Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Diseño y configuración de un sistema aéreo no tripulado con capacidad de procesamiento de imagen

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Miguel Marco Estevens

Tutor: Ángel Rodas Jordá

2018 - 2019

Agradecimientos

Dada aquí la oportunidad de expresarme, quisiera agradecer, en primer lugar, a mi familia, en especial padres y suegros junto con mi pareja quienes me han brindado todo el apoyo, el interés en todo momento, así como toda la ayuda posible a la hora de realizar pruebas y grabaciones.

En segundo lugar, a mis amigos quienes, a pesar de no poder verme ya que no disponía de tiempo para ellos en estos momentos tan tensos, han estado ahí mostrándome su apoyo y animándome a continuar.

Por último, y por supuesto no menos importante, a mi tutor Ángel Rodas por esa insistencia, ayuda y disponibilidad en todo momento para lo que pudiera necesitar y, como no, a Pablo Morcillo por rascar tiempo de donde no lo tenía para ayudarnos cuando estábamos con dudas de si el dron estaba bien constituido, así como los consejos que nos dio.

A todos, muchísimas gracias.

Resumen

En el presente trabajo final de grado se aborda el diseño y construcción de un sistema aéreo no tripulado —de ahora en adelante UAS[1]— que está formado por un dron hexacóptero, una estación de control en tierra implementada en un computador basado en Ubuntu y el sistema de comunicación entre ambos. Se han diseñado los componentes tanto hardware como software necesarios basándose en una filosofía de código abierto para así poder realizar el montaje del vehículo y la configuración de estos partiendo de un sistema operativo virgen al que se le ha ido instalando todo lo necesario según iba siendo requerido. El sistema tiene un modo básico de trabajo denominado RPAS en el que el vehículo es pilotado remotamente desde la estación de control en tierra, otro modo —el guiado— en el que se le indicará por comandos las acciones a realizar y, finalmente, un tercer modo en el que el vehículo tiene un comportamiento más autónomo, pues es capaz de desplazarse gracias a la retroalimentación obtenida a través de las imágenes que recibe a partir de la cámara que tiene instalada en su cara delantera.

Palabras clave: *dron*, procesamiento de imagen, *Open CV*, *Open Source*, *Raspberry Pi*, Python, MAVLink, dronekit, visión por computador y Ubuntu.

Resum

En el present treball final de grau s'aborda el disseny i construcció d'un sistema aeri no tripulat —d'ara endavant UAS— que està format per un *dron* hexacòpter, una estació de control en terra implementada en un ordinador basat en Ubuntu i el sistema de comunicació entre tots dos. S'han dissenyat els components tant maquinari com programari necessaris basant-se en una filosofia de codi obert per així poder realitzar el muntatge del vehicle i la configuració dels mateixos partint d'un sistema operatiu verge al qual se li ha anat instal·lant tot el necessari segons anava sent requerit. El sistema té una manera bàsica de treball denominat RPAS (*Remotely-Piloted Aircraft System*) en el qual el vehicle és pilotat remotament des de l'estació de control en terra, una altra manera —el guiat— en què se li indicarà per comandos les accions a realitzar i, finalment, una tercera manera en què el vehicle té un comportament més autònom, ja que és capaç de desplaçar-se gràcies a la retroalimentació obtinguda a través de les imatges que rep a partir de la càmera que té instal·lada a la seva cara davantera.

Paraules clau: dron, processament d'imatge, Open CV, Open Source, Raspberry Pi, Python, MavLink, dronekit, visió per computador i Ubuntu

[1] Los términos considerados de carácter específico serán explicados en el glosario.

Abstract

In the present final grade work the design and construction of an unmanned aerial system is addressed —hereafter UAS— which is formed by a hexacopter drone, a ground control station implemented in a computer based on Ubuntu and the system of communication between both. The necessary hardware and software components have been designed based on an open source philosophy to be able to assemble the vehicle and its configuration starting from a virgin operating system that has been installed everything necessary as it was being required. The system has a basic mode of work called RPAS (Remotely-Piloted Aircraft System) in which the vehicle is piloted remotely from the ground control station, another mode —guidance— in which it will be indicated by commands the actions to be carried out and, finally, a third way in which the vehicle has a more autonomous behaviour, because it is able to move thanks to the feedback obtained through the images it receives from the camera it has installed in your front face.

Keywords: dron, image processing, Open CV, Open Source, Raspberry Pi, Python, MAVLink, dronekit, computer vision and Ubuntu.

Tabla de contenidos

Tabla de ilustraciones

Contenido

Glosario

Actitud	Posición (en ángulos o radianes) del dron en función de sus tres valores fundamentales *(pitch, roll, yaw)* respecto al horizonte.
Dron	Aeronave no tripulada.
Dronekit	Librería de código abierto existente en diversos lenguajes de programación capaz de controlar vehículos ya sean aéreos, terrestres o marinos.
Hexacóptero	Aeronave formada por seis hélices.
MAVLink (*Micro Air Vehicle Link*)	Protocolo de comunicación punto a punto con posibles retransmisiones cuyos mensajes se empaquetan en ficheros XML y son desempaquetados por el receptor.
Pitch	Cabeceo. Inclinación del morro de la aeronave hacia arriba o hacia abajo. En los drones simula el funcionamiento de los elevadores de los aviones.
QGC (*QGroundControl*)	Aplicación software que actúa como estación de tierra. Es capaz de conectar con el vehículo, controlar todas sus variables y controlar el vuelo tanto para ver la ruta realizada como para controlar sus acciones (despegue / aterrizaje, realizar una serie de puntos o ruta, etc.)
Roll	Alabeo. Inclinación de la aeronave hacia la izquierda o hacia la derecha. En los drones simula el funcionamiento de los alerones de los aviones.
RPAS (*Remotely Piloted Aircraft System*)	Sistema de aeronaves pilotado de forma remota (aeronave + estación en tierra + enlace de comunicaciones)
SITL (Software In The Loop)	Simulador de Dronekit capaz de emular el comportamiento de un vehículo interpretando las órdenes que recibe.
Thrust	Potencia de los motores. Según la orientación del dron, afectará a su comportamiento.
UAS (*Unmaned Aerial System*)	Sistema Aéreo No Tripulado (aeronave + estación en tierra + enlace de comunicaciones).
Yaw	Guiñada. Capacidad de girar hacia la izquierda o hacia la derecha respecto al eje vertical. En los drones simula el funcionamiento del timón de cola de los aviones.

1. Introducción

Dado el auge de los vehículos no tripulados —junto con su respectivo auge laboral—, así como la diversidad en su usabilidad, hemos decidido diseñar un UAS viéndonos motivados por el carácter multidisciplinario del proyecto, ya que abarcamos ámbitos informáticos como son, por ejemplo, la visión por computador o la programación en *Python*; pero todo ello no nos serviría si no nos familiarizamos con algunos aspectos aeronáuticos como es la navegación aérea. Todo esto, con el añadido del ímpetu por querer demostrar que los no siempre estamos encerrados en un despacho —rompiendo así el estereotipo tan conocido—, ya que hemos debido de salir al campo para poder realizar las pruebas pertinentes.

A lo largo de este documento, podremos ver de manera detallada el proceso seguido para llevar a cabo este proyecto, parándonos a analizar y describir cada uno de los puntos que hemos creído conveniente puntualizar con tal de impedir que se puedan escapar ciertos aspectos que se podrían pasar por alto si no se reflejan aquí, así como los problemas que se nos han ido presentando y el modo en que los hemos ido solucionando sin apartar la vista de nuestros objetivos.

Por tanto, para que se pueda apreciar la calidad del trabajo realizado, podremos ver a continuación los objetivos que nos hemos marcado, así como los pasos en los que lo hemos segmentado. A estos, les sigue el estado del arte con la finalidad de saber en qué punto se encuentra en estos momentos esta tecnología comparando entre otras cosas los distintos tipos de vehículos aéreos, las posibles comunicaciones con tierra y hasta algunos de los distintos fabricantes de este tipo de vehículos.

Tras él, explicaremos en profundidad los materiales y métodos utilizados para el desarrollo de este trabajo, analizaremos desde todas las perspectivas posibles nuestro caso, el diseño y desarrollo de nuestra solución, veremos qué códigos hemos utilizado para comprobar la correcta configuración de vehículos, los códigos que hemos adaptado para que desempeñen las funciones que nosotros hemos deseado, su implantación y las pruebas realizadas tanto con el simulador como con el dron, finalizando con las conclusiones y algunos de los posibles futuros trabajos a realizar en caso de que se decidiese continuar la vida laboral en este ámbito.

1.1. Objetivos

Tomada ya la decisión, el presente trabajo final de grado tiene los siguientes objetivos principales:

El primero de ellos es construir un dron capaz de ser pilotado remotamente mediante una estación de tierra formada por un mando de radiofrecuencia y un software estándar de control.

Otro de los objetivos es crear programas para que pueda realizar ciertos trabajos “a ciegas”, es decir, trabajos basados en desplazamientos simples lo cual nos permitiese entender su funcionamiento interno.

El último de los objetivos principales es implementar código de manera que sea capaz de interactuar con el entorno a través de visión por computador dotándolo así de cierta autonomía.

Para ello, acordamos un proceso a través del cual logramos alcanzar nuestra meta. Para empezar, debemos construir el esqueleto del dron armando su esqueleto y soldando el cableado en sus emplazamientos correspondientes.

A continuación, debemos instalar el sistema operativo en la *Raspberry* de modo que seamos capaces de manipularla en caliente, es decir, estando instalada en el interior del vehículo y conectándonos a ella para realizar los cambios pertinentes de manera que evitamos tener que desacoplarla del esqueleto.

El siguiente paso es configurar el mando que dirige el dron por radiofrecuencia de modo que quedase personalizado como más cómodo nos resulte.

Tras esto, faltaría configurar la *Raspberry* para que se comunique con los otros componentes instalando las librerías necesarias para poder trabajar tanto con el dron como con el simulador.

Finalmente, tan sólo quedaría programar códigos con funcionalidades varias para ser ejecutados primero en el simulador de modo que, una vez verificado su correcto funcionamiento, nos desplacemos al campo de vuelo para experimentar con el dron real.

Desde un punto de vista empresarial, el impacto que cabría esperar de este vehículo respecto a los usuarios sería la posibilidad de adaptar cualquier código creado a las necesidades de cada cliente.

Siendo así, el hecho de que el cliente requiriese nuevas funcionalidades se convertiría en reuniones para acordar los cambios a realizar garantizando así un correcto funcionamiento testeando los cambios como corresponde antes de la entrega y, en caso necesario, acordar pruebas con el cliente para verificar dicho funcionamiento. De este modo evitaríamos posibles daños graves cuya reparación podría repercutir en un coste mucho mayor o incluso en daños personales.

2. Estado del arte

La idea de utilizar vehículos aéreos no tripulados se remonta al mes de julio del año 1849 cuando se pusieron en marcha 200 globos aerostáticos cargados de bombas que hicieron estallar sobre la ciudad italiana de Venecia. Acto que posteriormente se le atribuiría al ejército austriaco.

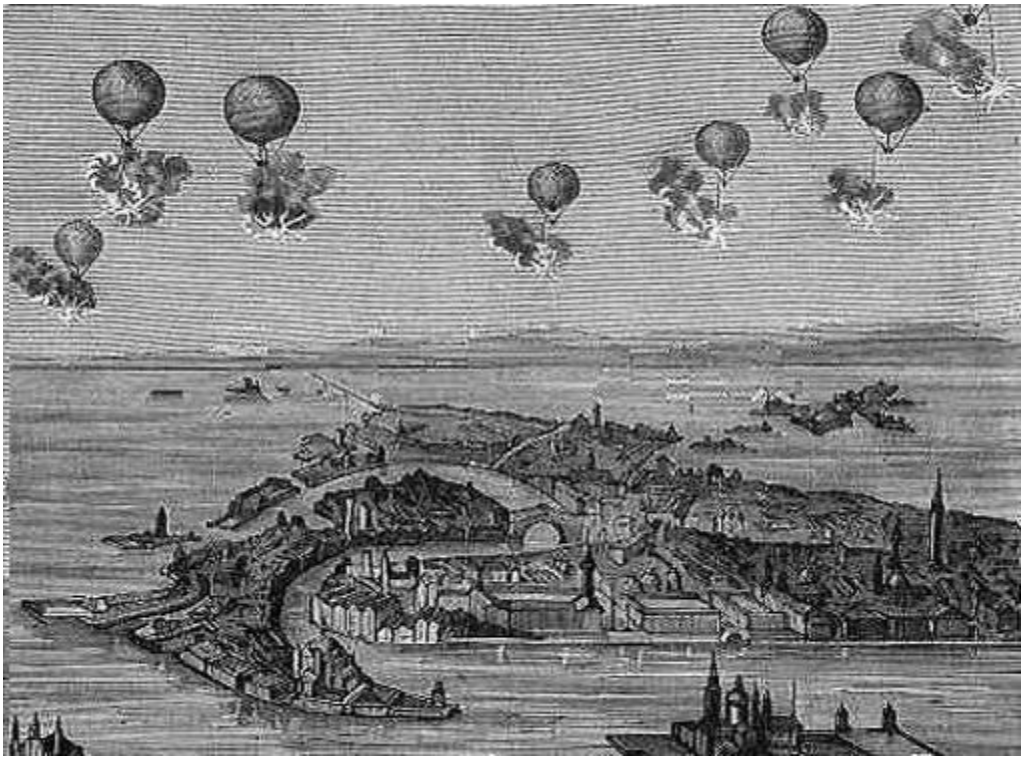


Figura 1: Globos aerostáticos sobrevolando Venecia

El inicio de las aeronaves no tripuladas se remonta al año 1907, cuando los hermanos Jacques y Louis Bréguet crearon el vehículo que aparece en la Figura 2, el cual requería de cuatro personas para estabilizarlo y que logró alzarse dos pies del suelo.

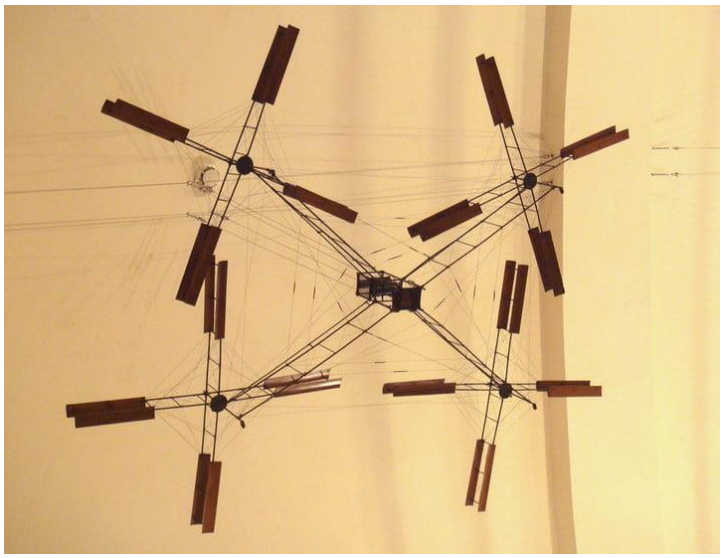


Figura 2: Primer cuadricóptero de la historia

Pasados 10 años de aquel momento y 16 del vuelo pionero de Kitty Hawk, en 1917 el Ruston Proctor Aerial Target se fue el primer avión sin piloto de la historia. Este avión, controlado por radio basada en la tecnología radiocontrol de Nikola Tesla, estaba pensado para usarse como bomba voladora, aunque nunca llegó a usarse, aunque sí impulsó otros proyectos como Kettering Bug.

Figura 3: Kettering Bug

Más adelante, en 1943 el conocido con el apodo “Fritz X” aunque su verdadero nombre era FX-1400 fue creado para el ejército alemán. Esta fue la primera arma de control remoto que realmente estuvo operativa. Lanzaba bombas de 2300 libras para hundir barcos, convirtiéndose así en el predecesor de los modernos misiles antibuque y otras armas guiadas de precisión.

Durante los años 60, el avance en la tecnología de transistores significó el abaratamiento de los componentes miniaturizados controlados por radio lo cual facilitó el alcance de este componente a un mayor abanico de clientela, fomentando así una industria cuyo más potente auge no se conocería hasta aproximadamente después de medio siglo.

Tras el atentado del 11 de septiembre de 2001, la CIA empezó a volar drones armados sobre Afganistán como parte de la guerra que mantenían con los talibanes, siendo llevada a cabo la primer a operación basada en drones en febrero de 2002 cuando se usó el dron Preadator para apuntar a un sospechoso que se creía que era Osama Bin Laden.

En 2006, percatándose ya del potencial de los drones no militares, la *Federal Aviation Administration* emitió los primeros comerciales para drones. Estos permisos eliminaron algunas de las limitaciones impuestas a los aviones no tripulados que volaban con fines recreativos, por lo que muchas industrias trataron de usarlos.

En 2010, la compañía francesa Parrot lanzó su Parrot AR Drone (Figura 4), el cual pudieron controlar completamente a través de Wi-Fi utilizando un teléfono inteligente. Tras él, numerosas empresas han continuado con este negocio, estando entre ellas *Amazon* la cual ha iniciado el reparto de paquetes utilizando drones o el fatídico caso del dron Lily Camera, cuya empresa, a pesar de acumular 34 millones de dólares en pedidos, tuvo que declararse en bancarrota tras una serie de retrasos.

Figura 4: Parrot AR Drone

Ya en el 2016, DJI introdujo, con el Phantom 4, la visión inteligente de una computadora, así como el aprendizaje automático. Esto permitió evitar obstáculos y rastrear (y fotografiar) de forma inteligente a personas, animales u objetos, en lugar de únicamente poder obedecer a señales de GPS.

Figura 5: Phantom 4 de DJI

Dentro del ámbito de los drones, cabe destacar ciertas diferencias, pues hay multitud de términos semejantes, pero con diferencias sutiles. Para empezar, damos a conocer los UA (*Unmanned Aircraft*) o UAV (*Unmanned Aerial Vehicle*), que son las aeronaves —o vehículos aéreos— no tripuladas, las cuales, por tanto, hacen referencia a cualquiera de este tipo de vehículos a los que comúnmente se les ha llamado “dron”. Si a estos vehículos les añadimos un enlace de comunicaciones —lo cual repercute con todo un sistema de control desde tierra ya sea por un computador o por un mando controlador por radio manejado por una persona— pasarían a llamarse UAS.

Por otro lado, existen los vehículos que se conocen como RPA (Remotely Piloted Aircraft) los cuales, a diferencia de los UA, hacen referencia únicamente a las aeronaves sin personas embarcadas las cuales estan tripuladas de manera remota comunicándose a través de un enlace de comunicaciones —y ésta es la diferencia—

con la estación de tierra. Es por ello, y aclaradas estas diferencias, queda claro que no todos los UAS son considerados RPAS, pero sí todos los RPAS son considerados UAS.

***Figura 6**: Unmaned Aerial Vehicle*

Dejando ahora del lado la terminología referente a qué componentes forman nuestro vehículo, pasamos a hablar sobre un detalle referente a su forma. Es aquí donde debemos diferenciar el tipo de movimientos que va a realizar nuestro vehículo, ya que existen los vehículos aéreos de ala fija y los *multirrotores*, Puesto que los vehículos de ala fija realizan movimientos más típicos de una aeronave convencional, mientras que los *multirrotores*, tienen mayor libertad de movimiento e incluso son capaces de mantenerse estáticos en el aire. Esto hace que este tipo de vehículo proporcione una mayor precisión a la hora de realizar sus tareas, más incluso si se trata de tareas relacionadas con la visión por computador, a costa de tener una menor autonomía que los vehículos de ala fija.

Por otro lado, es necesario recalcar la diferencia entre los sistemas con código abierto y los sistemas de código cerrado, ya que un sistema con código cerrado nos habría limitado notablemente el trabajo a realizar, pues nos tendríamos que ceñir a las funciones que nos aportaría el fabricante sin posibilidad de poder trabajar con las variables propias del vehículo, como sí sucede con los sistemas de código abierto entre los que está el que hemos escogido.

Es por ello, que a sabiendas de que va a suponer un mayor consumo de batería, nos hemos decantado por los *multirrotores*, concretamente por un hexacóptero, es decir, un *multirrotor* de seis hélices, dado que es el término medio entre un *multirrotor* de cuatro hélices y un *multirrotor* de ocho hélices, es decir, hemos escogido el vehículo con valores de consumo y precisión intermedios.

3. Materiales y métodos

A continuación, procedemos a citar y describir los elementos utilizados, tanto hardware como software, para tener una primera impresión de todo lo que se ha requerido para llevar a cabo este proyecto.

Figura 7: Relación de elementos utilizados.

En la Figura 7 podemos observar multitud de conceptos que, por el momento, parecen no estar relacionados, pero en las siguientes líneas los daremos a conocer para así poder entender cómo se ha llevado a cabo el proyecto. Para ello se ha optado por separar la parte hardware de la parte software con la finalidad de no entrelazar conceptos.

En la sección hardware vamos a analizar todos los elementos que componen nuestro vehículo, así como ciertos dispositivos que hemos necesitado en nuestro ordenador para comunicarnos con él. En la Figura 7 podemos observar algunos de estos elementos, entre ellos podemos encontrar nuestro vehículo —el dron hexacóptero F550 de DJI—, el mando que actúa como emisora controladora por radiofrecuencia, la Raspberry Pi 3 y la Emlid Navio2.

En la sección software vamos a analizar cada uno de los elementos que ha sido necesario descargar e instalar para poder trabajar con ellos y completar así el trabajo. Entre ellos, y basándonos en la Figura 7, podemos encontrar el sistema operativo Ubuntu 16.04 con soporte técnico a largo plazo (LTS, *Long-Term Support*), las librerías Dronekit, las librerías OpenCV, la estación de tierra (QGroundControl) y el lenguaje Python versión 2.7.

3.1. Hardware

Empezamos pues uno de los dos grandes bloques, el hardware, donde encontramos todos los elementos físicos utilizados para el montaje del vehículo.

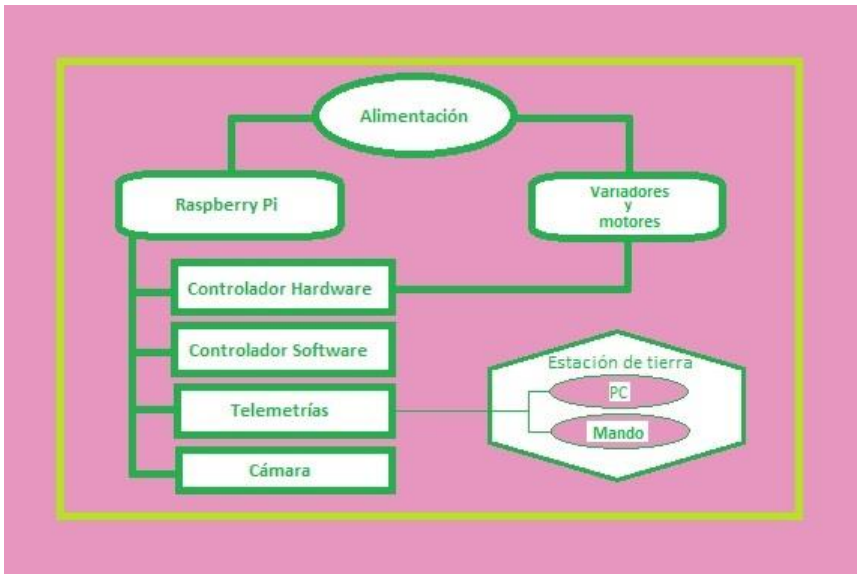


Figura 8: Composición del dron.

Dentro del dron —centrándonos en el hardware— existen diversos elementos que, aun pudiendo estar estrechamente relacionados con los elementos software, nos conviene citar, entre ellos está la Raspberry, que se encarga de todo el cómputo de código por parte del vehículo; las antenas de telemetría que permiten la comunicación con la estación de tierra (aunque en nuestro caso la estación de tierra se divide en dos partes: el mando y el QGroundControl); la cámara que permite la visión por computador; la batería, que transmite energía a todo el vehículo y, por supuesto, los variadores que transmiten la señal a los motores para controlar el vuelo del dron.

Figura 9: Dron completado.

3.1.1. Frame y motorización

El elemento del cual no nos podemos olvidar es el propio vehículo. En nuestro caso hemos escogido el modelo F550 de la gama "*Flame Wheel ARF Kit*" propia de DJI.

Figura 10*: Kit de montaje para DJI Flamewheel F550*

Este dispositivo —visible en la Figura 10*: Kit de montaje para DJI Flamewheel F550*—, siendo el mayor de su gama con una distancia diagonal entre ejes de 550mm., consta de 478g. de peso neto tal cual lo recibimos por defecto, pudiendo llegar a levantar los 2400g. requiriendo una batería LiPo de 3 o 4 celdas, motores de 22x12mm y variadores ESC 15A OPTO.

Sus brazos están formados por un material ultrarresistente proporcionando así una mejor seguridad contra las posibles caídas. Del mismo modo, el uso de materiales altamente resistentes para componer la PCB de la placa del *frame* garantiza un mejor estado del cableado que se dirige a cada ESC, así como mantiene más segura la propia batería. A esta estructura se le debe añadir motorización la cual veremos a continuación.

Inicialmente utilizamos los variadores modelo 420 Lite de la serie E pues, a pesar de las recomendaciones del fabricante debido al menor consumo del 15A OPTO, estos eran los que incluía nuestro paquete. Posteriormente, serían reemplazados por los variadores 430 Lite de la misma serie.

(a) Variador 420 Lite de la serie E

(b) Variador 430 Lite de la serie E

Figura 11*: Variadores utilizados a lo largo del proyecto*

En la Figura 11(a) podemos observar cómo del propio variador, por la parte superior según la imagen, sobresalen 3 cables. El más grueso es el que alimenta el variador, se trata de un cable coaxial en el que su núcleo es el cable de alimentación que está protegido por un aislante dieléctrico de color rojo para aclarárnoslo y éste último recubierto por el cable correspondiente a la masa que aparece justo por debajo de la funda protectora de color negro que podemos observar en dicha imagen.

Por otra parte, aparecen los cables naranja y marrón los cuales se conectan a la placa Navio2 para recibir de ella las señales que deben convertir para su respectivo motor siendo el cable naranja el que se corresponde con la señal y el cable marrón el que se corresponde con la masa de modo que ésta es usada para poder calcular la diferencia de potencial con la señal recibida a través del cable naranja.

Como último detalle respecto a esta figura, encontramos los tres bornes hembras alojados en la parte inferior de la misma. Estos bornes no tienen ninguna polaridad fijada, sino que, dependiendo de sus conexiones con los respectivos del motor, lo hacen girar en un sentido o en otro.

Respecto a la Figura 11(b) encontramos los variadores usados a última instancia debido a la avería de los anteriores, los cuales habían dejado de fabricar para cuando debíamos realizar el cambio. En este caso, los cables para la señal se sustituyeron por blanco (para alimentación) y negro (para masa).

Algunas mejoras notables que remarcar en estos dispositivos serían el paso del cable coaxial a dos cables unipolares con aislante de modo que no era necesario ya deshacer el trenzado de la masa como sí era indispensable en los anteriores, o el paso de tener los bornes incrustados dentro del dispositivo a tenerlos exteriorizados de modo que era más fácil separarlos en caso de querer alternar la polaridad del motor.

Para este proyecto hemos usado motores sin escobilla, lo cual, aunque los hace más caros, genera un mayor rendimiento del motor, así como una mayor eficiencia energética además de retrasar el deterioro de los mismos ya que, al carecer de escobillas, evitamos la rozadura de las mismas con su pertinente generación de chispas, calor, consumo y ruido.

Figura 12: Motor DJI 2312

En la Figura 12 podemos encontrar los motores utilizados en nuestro proyecto con sus respectivos bornes los cuales conectaríamos a los del variador de su respectivo brazo. Con esto, deberíamos probar una combinación que hiciese girar el motor en el sentido que él mismo nos indica (*clockwise* o *counter-clock wise*) con las flechas pintadas en él o con las propias abreviaturas (CW y CCW) de manera que, si una combinación lo hacía girar en un sentido, bastaría con alternar un par de bornes para alterar el sentido de giro.

El fabricante del vehículo recomienda usar hélices de dimensiones comprendidas entre 10 × 3.8” y 8 × 4.5”. De modo que nosotros usamos unas cuyas dimensiones son 9.4 × 5” hechas de material plástico y reforzadas por fibra de vidrio.

Figura 13: Hélices de dimensiones 9.4 × 5"

Estas hélices se caracterizan por ser autorroscantes, es decir, nosotros a la hora de montarlas, debemos ajustarlas hasta que lleguen a su límite y, en el momento del vuelo, éstas girarán en el sentido de su desenrosque mas la fuerza que ejerce el aire contra ellas provoca un empuje en sentido contrario que provoca su apriete.

3.1.2. Sistema de control

A continuación, nos encontramos con una de las piezas fundamentales de este proyecto, ya que ella ha sido la que ha dirigido el autopiloto, así como todo el sistema operativo que éste requiere.

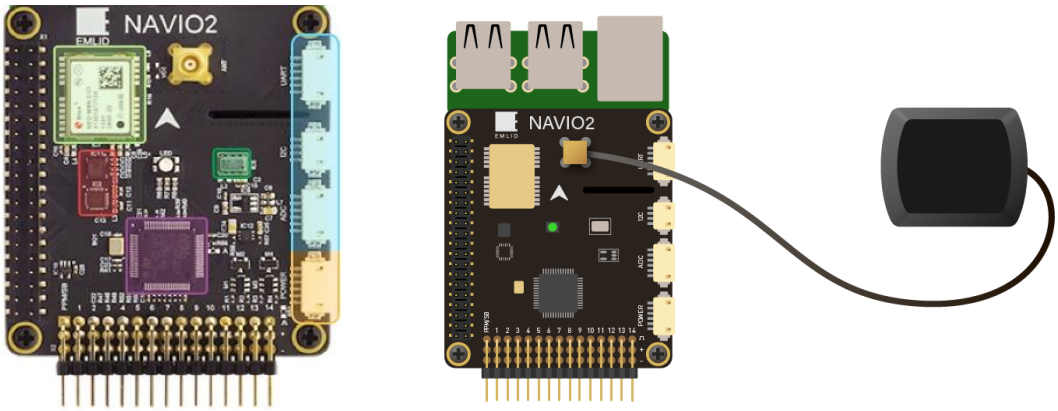
Figura 14: Raspberry Pi 3 Model B

En la Figura 14, por tanto, podemos encontrar esta placa, la cual se corresponde con la tercera generación del modelo B de Raspberry. Este modelo, se diferencia del A por tener un procesador con una frecuencia de 200 MHz menos, por sustituir los 512 MB de SDRAM por 1GB de RAM, por disponer de puerto 100 Base Ethernet, una mejora en la conexión de alimentación, y 4 puertos USB 2.0 en lugar del único del que dispondríamos con el modelo A+.

Este dispositivo, totalmente compatible con la Emlid Navio2 —la cual vamos a estudiar en el siguiente punto—, nos ha permitido la instalación de un sistema operativo completo de manera que podíamos realizar cualquier operación en él, desde la instalación de librerías, pasando por la modificación de ficheros propios de un sistema operativo hasta la interoperabilidad de los dispositivos que le hemos ido conectando.

En definitiva, este modelo dispone de mejoras respecto al modelo A+ que la convierten en indicada debido, entre otras cosas, a la mejor protección contra subidas de tensión ya que, si la alimentamos con baterías, en caso de avería pueden generar alguna sobretensión que quemase la propia placa. Además, el puerto Ethernet nos garantizaba una conexión directa a ella a través de un cable de manera que, en caso de desconfigurar la conexión inalámbrica, podríamos acceder a ella sin necesidad de retirarle la tarjeta micro-SD que incorpora a través de ssh por la IP que el router le hubiera asignado.

En la Figura 15 podemos observar la placa que nos ha permitido, junto con la Raspberry, llevar a cabo este proyecto debido a que esta es la encargada de la obtención de todos los datos tanto telemétricos como los parametrizables.



(a) Emlid Navio2 **(b)** Emlid Navio2 con GPS conectado

Figura 15: Emlid Navio2

En dicha figura podemos encontrar una serie de zonas coloreadas, de las cuales podemos citar la zona con un color verde más claro que se corresponde con la descodificadora de las señales GNSS la cual recibe información a partir de la antena que se conecta justo a su derecha. Esta descodificadora es capaz de tratar señales procedentes de satélites GPS, GLONASS, Beidou, Galileo y SBAS que recibe de la antena visible en la Figura 15(b).

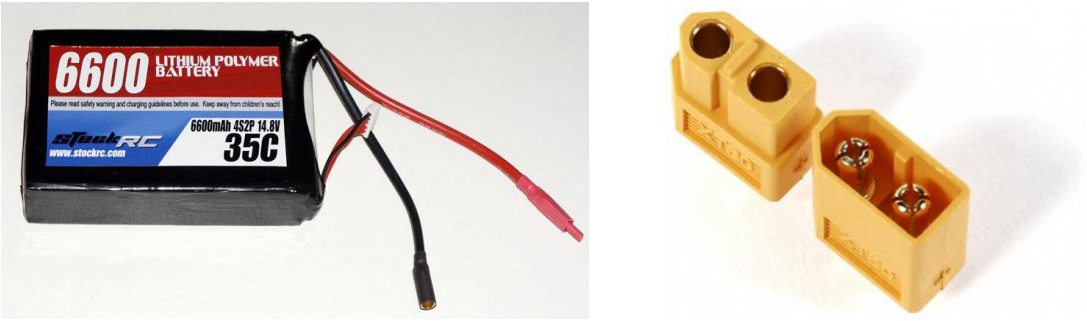
Por otra parte, tenemos la zona enmarcada con un verde más oscuro y se corresponde con el barómetro de alta resolución que lleva incrustado cuya precisión es de 10 cm. Además, dispone también de una zona coloreada de azul la cual se corresponde con los puertos ADC, I2C y UART los cuales permiten comunicarse con otros sensores y antenas de radiofrecuencia.

La zona remarcada con un color amarillo anaranjado se corresponde con la entrada de alimentación la cual cuenta con una protección contra sobretensiones de voltaje. Es necesario también mencionar la zona marcada por un color rojizo, ya que en ella encontramos la IMU dual que incorpora los acelerómetros, giroscopios y magnetómetros utilizados para obtener la orientación y la detección de movimiento.

Por último, queda hablar de la zona enmarcada de color púrpura, la cual es la que gestiona la entrada y salida de señales de radiocontrol. Este dispositivo acepta entrada PPM y SBUS siendo este último el que utilizamos nosotros debido a nuestra antena receptora de las señales radiocontrol procedentes del mando. Las señales recibidas las puede convertir hasta en 14 canales de salida diferentes utilizando modulación por ancho de pulso (PWM) para los motores, servos y variadores.

3.1.3. Alimentación

Para alimentar el dron, hemos utilizado una batería de polímero de litio (Li-Po) recargable de 4 celdas de 3.7V cada una (alcanzando un voltaje de 14.8V en total), aunque pueden llegar a 4.2V por celda, consiguiendo así un total de 16.8V. Posteriormente, hemos requerido otra adicional —tratando de que tuviera las mismas características para que en caso de necesidad poder alternarlas indistintamente— para poder alimentar la antena receptora de vídeo.



(a) Batería LiPo 6600mAh 35C 4S2P 14.8V ***(b)*** Conectores XT60 (macho y hembra)

Figura 16: Batería LiPo junto con su conector

Como dato que en posteriores explicaciones nos va a ser de utilidad, queremos remarcar el uso de los cables visibles en la Figura 16(a). El par de cables más gruesos son los usados para suministrar o sustraer energía a la batería, mientras que los cables más finos son los cables balanceadores, los cuales están conectados cada uno a una de las celdas y, al conectarlos al cargador, le indican el voltaje actual de dicha celda. Para evitar invertir polaridades, hemos eliminado este extremo de la batería para sustituirlo por un conector XT60 como el de la Figura 16(b).

Estas baterías, como la visible en la Figura 16(a) siempre han sido cargadas bajo supervisión controlando que no se hinchen ni se calienten —dado que ello nos impediría seguir utilizándolas ya que podrían explotar en cualquier momento poniendo en peligro la integridad del dron y, por encima de todo, de los usuarios—, evitando siempre las descargas profundas y sobrepasar la carga máxima (4.2V por celda). Ambas baterías se han almacenado siempre en un lugar fresco para evitar también que el calor las deteriorase.

Cabe aclarar también el concepto 35C, el cual se refiere al factor de descarga de la batería. Siendo el caso de que una batería de 1000mAh 1C se descargaría a 1A en una hora, realizando los cálculos:

$$t = \frac{6600mAh}{35C} = \frac{6600}{35000} = 0.18h = 11{,}314\ min. = 11min.\ 18{,}84s.$$

Es decir, a 35A, la batería se descargaría en 11 minutos y 18.84 segundos, lo cual significa que debíamos hacer las pruebas lo más rápido posible para evitar tener que parar a recargar la batería en mitad de la batería de pruebas.

No podemos hablar de las baterías sin hablar de su cargador. Este está compuesto por dos piezas las cuales vamos a analizar en las líneas siguientes.

(a) Fuente de alimentación conmutada SPS 230V 12A Robbe 1-8480

(b) *Vista global del instrumento*

(c) Vista del conexionado de salida

(d) Conjunto de cableado

Figura 17: Fuente de alimentación, cargador y cableado necesario

En la Figura 17(a) podemos encontrar la fuente de alimentación que dará corriente al cargador del que pronto hablaremos. Esta fuente de alimentación con un voltaje de salida de 13.8V es ideal para los cargadores de 12V en una red eléctrica que suministre hasta 230V, así como también es útil para herramientas eléctricas que requieran el mismo voltaje. Este instrumento admite una corriente máxima de 12A lo cual cubre una amplia gama de aplicaciones además de estar protegido contra la posible inversión de polaridad y sobrecarga con un interruptor de alimentación separado y ventilador refrigerante.

Así pues, las 3 figuras siguientes: Figura 17(b), Figura **17**(c) y Figura **17**(d), tratan sobre el cargador de la batería. En la primera de ellas, la Figura 17(b), podemos observar el propio cargador desde un ángulo donde podemos visualizar la conexión de entrada de alimentación a la parte izquierda del lateral más estrecho. Para conectar este dispositivo con la fuente de alimentación, basta con usar el conjunto de cables situados en la parte más superior de la Figura 17(d), conectando las pinzas de cocodrilo a la fuente de alimentación y el extremo contrario a la conexión de entrada de alimentación anteriormente citada.

En esta imagen encontramos también los botones que nos permiten cargar o descargar la batería seleccionando en el menú la opción "*CHARGE*" o "*DISCHARGE*" respectivamente, así como equilibrar sus celdas usando la opción "*BALANCE*" para que todas ellas tengan la misma carga.

Para conectar la batería al cargador, requerimos de la Figura 17(c) de modo que, en caso de querer realizar una carga o descarga, debemos conectar únicamente el conector principal de la batería, formado por un conector XT60 hembra con su respectivo macho, tal y como podemos apreciar en el conjunto de cables situados a la izquierda de la Figura 17(d) los cuales se conectan al bloque "*OUTPUT*" visible en la Figura 17(c), en caso de querer realizar un equilibrado de la carga, es necesario conectar también el conector de balanceo en su respectiva casilla situada al lado de la salida principal. Para saber en qué casilla se debe conectar, basta con observar el

esquema situado sobre este panel de conexiones para buscar en él el número de celdas de nuestra batería.

Una vez encontrado el puerto al que debemos conectarnos, lo introducimos de la única manera posible ya que, en caso de tratar de introducirlo al revés, el diseño del conector no nos lo permite, lo cual nos protege también contra la inversión de polaridad.

Para activar la función, sea cual sea, debemos conectar el cargador a la fuente de alimentación, activar su interruptor y, una vez encendido el cargador, buscamos en el menú nuestro tipo de batería —puede ser de litio polimerizado (como la nuestra), de níquel-cadmio (NiCd) o de níquel-metal hidruro(NiMH)— usando los 2 botones centrales visibles en la Figura 17(b) para así, una vez encontrado, validar con el botón situado a la derecha.

Entonces, estando ya en el menú de nuestro tipo de baterías, presionamos la tecla de validación para poder ajustar los parámetros a los nuestros. Primeramente, parpadea el número de celdas con unidad de medida “S” —sub-baterías— buscamos entre las opciones la que se corresponde con nuestra cantidad de celdas la cual está acompañada del voltaje total añadido de todas las sub-baterías a 3.7V cada una —en nuestro caso 14.8V—, validamos nuevamente para que pase al siguiente valor donde debemos escoger la intensidad de corriente a la cual queremos que se cargue la batería, por lo que, ya dependiendo de la corriente que admita la batería y también según la necesidad de volver a utilizarla, siempre podremos incrementar o decrementar este valor.

Una vez hecho esto, mantenemos presionado el botón derecho (el de validación) hasta oír un sonido distinto al de una simple pulsación. Este sonido, el cual es un poco más largo, en caso de estar la batería en buen estado es acompañado por un menú de confirmación en el que, al presionar el botón de validación, empieza la carga de la batería.

En caso de haber algún error de carga, puede ser debido a que la batería se ha descargado notablemente. En este caso, la batería no puede cargarse del modo habitual, por lo que debemos conectar el cable balanceador y seleccionar “*BALANCE*” para que se equilibren las cargas y, una vez alcanzado un nivel óptimo, ya podemos volver a seleccionar la opción de “*CHARGE*” para cargar normalmente.

3.1.4. Herramientas de comunicación

Así pues, damos paso al elemento que nos ha permitido comunicarnos con el vehículo una vez estaba en el aire. Estas son las antenas telemétricas, las cuales vamos a poder ver a continuación de manera que cada una de ellas tiene una función especial dentro de los enlaces de comunicación.

Figura 18: Telemetrías usadas

En la Figura 18, podemos observar las telemetrías usadas para dicha comunicación. La imagen de la izquierda representa la telemetría utilizada para comunicar con él desde el ordenador de modo que la antena más corta es la que se ha instalado en el vehículo conectándola a él con los cables que corresponden a la conexión UART (rojo para alimentación, verde para transmisión, amarillo para recepción y negro para masa). La antena más larga se ha conectado al ordenador utilizando el cable USB de tipo micro-B que aparece en pantalla.

En la Figura 18, la imagen de la derecha se corresponde con la telemetría utilizada para comunicar el mando Taranis con el vehículo. Este dispositivo dispone de una serie de pines que se corresponden con sus canales —concretamente 8 canales, a tres pines por canal (uno para la señal, otro para la alimentación y otro para la masa)— más otros tres pines que se corresponden con los pines conocidos como SBUS que generan la salida multiplexada de todos los canales que recibe la antena receptora del mando.

Estos tres pines son los que hemos utilizado para conectar con la Raspberry, pues ha bastado con utilizar una columna de pines de esta para recoger por ella las señales de todos los canales. Los pines que le continúan se corresponden con las conexiones de los variadores siguiendo un orden específico el cual se muestra en la Figura 32 con posterioridad.

Para llevar a cabo algunos de nuestros objetivos, hemos tenido que agregar a nuestro ordenador una antena GPS la cual se ejecute a través de un demonio y facilite coordenadas al resto de servicios. Para ello hemos utilizado la antena Receptor GPS USB Globalsat BU-353S4 visible en la Figura 19

Figura 19: Receptor GPS USB Globalsat BU-353S4

También es necesario mencionar la cámara que hemos instalado en el dron, la cual está conectada a una digitalizadora de vídeo emitiendo la señal por radiofrecuencia. Dicha señal es recibida por una descodificadora de vídeo y transmitida al ordenador a través de un puerto USB, cuyos datos serán interpretados por la librería OpenCV. A continuación, se adjunta una imagen en la que podemos apreciar tanto la cámara, el conversor digital-analógico, la antena emisora de imagen junto con la receptora y su digitalizadora.

Figura 20: Set de retransmisión de vídeo.

Entre los dispositivos de la Figura 20 encontramos la cámara que se ha instalado en el dron, la cual es alimentada por el transmisor —a quien transmite la imagen y que está situado a su derecha— seguidos por el receptor y la digitalizadora de vídeo la cual descodificará lo que el receptor le transmita a través de los cables RCA de los cuales hay que tener en cuenta que, dado que no existe un estándar, cada fabricante establece un orden distinto en el *jack* que los une y que, por tanto, el receptor emita por una banda la imagen, pero que realmente el *jack* no lo recoja por la banda correspondiente y que, en su lugar, la transmita por la banda de alguno de los audios.

Por otro lado, tenemos el mando capaz de comunicarse con el dron sin necesidad de ninguna otra herramienta. Este mando (llamado "*Taranis X9D*") dispone de dos palancas —apreciables en la imagen como cada uno de los "*Stick*"— que son capaces de desplazarse en dos dimensiones con lo que obtenemos 4 dimensiones independientes (representadas en la Figura 21 como JX, donde X es cada una de las dimensiones) las cuales se corresponden con *pitch*, *roll*, *yaw* y *thust*.

Figura 21: Mando Taranis X9D

Además, dispone de una serie de botones —visibles en la parte inferior de la Figura 21— que nos servirán para configurar el mando junto con otros interruptores —situados en la parte superior de esta figura, representados de la forma SX— los cuales tienen comportamientos diversos, es decir, hay algunos que tienen 3 posiciones, otros que tienen dos posiciones y otro que, a pesar de tener también dos posiciones, se mantiene en una posición determinada mientras no ejerzamos ninguna fuerza sobre él.

Lo que diferencia a este interruptor del resto es que al ejercer la fuerza de nuestro dedo sobre ella le cambiamos la posición, pero en el momento en el que dejamos de ejercer dicha fuerza, el interruptor vuelve a su posición de reposo. De ahora en adelante, este interruptor —al cual Taranis reconoce en su hoja de referencia como "momentáneo" o "SH"— le llamaremos SH.

Existen diversos modos de configuración de la radio ya sea para aviones, helicópteros o multirrotores (pues dentro de cada tipo de RPAS era necesario especificar otros detalles significativos), por lo que tuvimos que detenernos a analizar cuál era la opción que más acertaba con nuestro diseño y, a la hora de configurar las palancas de actitud del dron, qué palanca y en qué dirección era más recomendable configurar cada dimensión de navegación, así como con los interruptores, cuáles de ellos podían activar un modo u otro de funcionamiento y cuál de ellos debe superponerse al resto para, por ejemplo, tener un modo de regreso seguro en caso de haber finalizado las pruebas de manera que optimice su regreso y aterrizaje. Toda esta configuración está disponible en el anexo C para evitar desviarnos del principal objetivo de este proyecto.

3.1.1. Estación de soldadura

Para la soldadura del cableado hemos requerido de la estación visible en la Figura 22(a) la cual es capaz de alcanzar los 480 ºC, aunque nosotros hemos estado trabajando con 370 ºC de modo al aplicar calor a nuestro hilo soldador —visible en la Figura 22(b)— cuya composición es un 40% de plomo y el 60% restante de estaño, tiene un punto de fusión de 183 ºC se derrite con facilidad permitiéndonos trabajar con él y amoldarlo a la placa para poder cubrir el cableado y mantenerlo bien sujeto a la placa.

(a) Estación de soldadura HoLife de 60W

(b) Hilo soldador 60% Sn y 40% Pb

(c) Extractor de soldadura

Figura 22: Conjunto de herramientas de soldadura

Aunque teníamos la opción de utilizar un hilo 100% de estaño, decidimos usar esta aleación debido a que, aunque el estaño se funde a bajas temperaturas, la adición del plomo facilita una más fuerte sujeción.

Por otra parte, en la Figura 22(c) podemos observar el extractor de soldadura el cual nos fue útil a la hora de sustituir unos variadores por otros ya que, con él, pudimos eliminar el sobrante de estaño consiguiendo así poder volver a soldar cableado como si la placa fuera nueva. Para ello, solo bastaba con ejercer presión sobre el extractor para accionar la bomba de presión, calentar la soldadura y, cuando a simple vista podíamos apreciar que empezaba a derretirse, acercábamos el extractor, accionábamos el botón amarillo central y él absorbía el estaño.

3.2. Software

3.2.1. Sistema Operativo

Empezando por la parte más centrada en la informática, encontramos el computador con el que a partir de una distribución Ubuntu 16.04 LTS, hemos instalado en él la versión 2.7 de Python —elegido entre otras cosas por ser multiplataforma y por su versatilidad a la hora de programar— y las librerías *dronekit* y *dronekit-sitl*, las cuales son las librerías que nos han permitido trabajar con todo lo que el dron es capaz de entender y con el simulador respectivamente, siendo así que forman un bloque fundamental en el que unos elementos se complementan con el resto.

Por otro lado, tenemos que hablar también del sistema operativo que incluye la Raspberry. En ella hemos instalado Raspbian el sistema operativo que los propios fabricantes de Raspberry han adaptado a partir de Debian. Para ello accedimos a la web https://www.raspberrypi.org/downloads/raspbian/ y descargamos la versión Lite. En él instalaríamos las librerías necesarias para poder interactuar con el vehículo.

3.2.2. Dronekit y Dronekit-SITL

La librería *Dronekit* es la encargada de manejar con un conjunto de funciones y parámetros todo el hardware del dron a través del autopiloto —en nuestro caso ArduCopter— incluyendo en él los variadores de potencia y, por tanto, cada uno de los motores, así como también calcula los cuaterniones que participan en la función que rectifica la actitud del dron. Esta librería de código abierto está implementada en Android y Python, por lo que desde cualquier dispositivo Android podríamos ejecutar nuestros códigos adaptándolos a dicho lenguaje, aunque Python no deja de ser el predilecto debido a que Android posee librerías privadas y debido a que a nivel industrial los trabajos suelen llevarse a cabo desde ordenadores ya que tienen mayor capacidad de cómputo que los móviles y mayor comodidad para alterar el código.

Existe también la librería *dronekit-SITL* (*Software-in-the-loop simulator*), que es la encargada de poder ejecutar en el ordenador un proceso que simula la actitud que tendría el dron real, con lo que hemos podido confiar en esta herramienta para evitar caídas por un mal funcionamiento del código o incluso, en el peor de los casos, poder haber perdido el dron en caso de que hubiera continuado en línea recta cuando debería haber girado, frenado o aterrizado. Esta herramienta es también capaz de generar conexiones, las cuales se aprovechan para comunicarse desde el computador con el vehículo o, en caso de haber creado un proxy, con el puerto pertinente.

El código que hemos implementado con estas librerías es ejecutado desde el computador —aunque si deseásemos que formase parte del comportamiento del dron, tan solo necesitaríamos registrarlo como modo de funcionamiento añadiéndolo al conjunto de modos de funcionamiento de ArduCopter y configurando un interruptor del mando para que lo llevase a cabo— de modo que para ejecutarlo en el dron es requisito necesario realizar una conexión con el vehículo para que se le pueda transmitir, así como que el dron nos transmita la información necesaria. Para ello, utilizamos MAVProxy, el cual —junto con su protocolo de comunicación— es descrito en las siguientes líneas.

3.2.3. ArduPilot y ArduCopter

ArduPilot es software libre cuya función es ser el autopiloto del vehículo de tal manera que, si ajustamos el modo de funcionamiento del dron a alguno de los ya definidos por el fabricante, el vehículo es capaz de autogestionarse como, por ejemplo, el modo "*PosHold*" en el que el dron deberá mantenerse estable en el aire forzando aquellos motores que más lo necesiten en caso de perder estabilidad. Además, el autopiloto también traduce en señales eléctricas cada uno de los comandos que reciba el dron a través de los programas Python que diseñemos. Dichas señales se envían a través de los pines de salida de la Raspberry a los variadores los cuales traducen dicha señal en velocidad de giro para dicho motor siendo, por tanto, independiente al resto.

Por otra parte, cabe destacar que *ArduCopter* —y por este motivo lo englobamos en el mismo punto—, es un subsistema que hereda de *ArduPilot* y que se utiliza para todos los vehículos rotores que comparten semejanza comportamental con un helicóptero. Así pues, existen también otras familias dentro de *ArduPilot*, ellas son: *ArduPlane* para aeroplanos, *ArduRover* para los vehículos terrestres y los acuáticos no sumergibles y, por último, *ArduSub* para vehículos submarinos. Todos estos autopilotos son alterables desde el ordenador utilizando la libería Dronekit.

Es necesario también en este momento hablar de los modos de funcionamiento del dron o, almenos, de aquellos que más vamos a utilizar.

MODO DE FUNCIONAMIENTO	Descripción
AltHold (*Altitude Hold*)	Se encarga de mantenerse a la altura actual sin tener en cuenta la posición.
PosHold (*Position* Hold)	Se encarga de mantener la posición fija, estabilizándose en el aire y contrarrestando las posibles rachas de viento.
Guided	Permite la ejecución de códigos diseñados por el usuario abriendo así el gran abanico de posibilidades que nos brindan estas librerías junto con el autopiloto.
RTL (*Return To Launch*)	Su principal función es alzarse a una determinada altura —por defecto 10 metros— para así evitar posibles obstáculos y, una vez alcanzada esa altura, desplazarse hasta el punto de despegue. Dependiendo de la configuración insertada en el vehículo, el vehículo puede aterrizar una vez ubicado en dicha posición o mantenerse en el aire esperando a que el usuario lo aterrice manualmente

3.2.4. MAVLink y MAVProxy

MAVLink (*Micro Air Vehicle Link*) es un protocolo de comunicación muy ligero —la versión 1 tiene un tamaño de mensaje de 8 bytes y la versión 2 tiene 14 bytes— que nos permite interactuar con el dron a través de distintos canales posibles de modo que aún en caso de funcionar a baja frecuencia, el bajo coste espacial de cada paquete asegura una alta comunicación. Sus mensajes son ficheros XML los cuales se transmiten de emisor a receptor y, en caso de requerirlo, se responden con ficheros del mismo formato. Dado que cada mensaje contiene información completa, es decir, su información no depende de la de otros mensajes, la pérdida de un mensaje no afecta al resto de ellos.

MAVProxy es como conocemos al proxy que genera *mavgen* (generador de proxys MAVLink) que comunica vehículo y ordenador a través de un enlace. Este enlace puede generarse a través de UDP o a través de telemetría (entre otros). Este enlace —conocido en el momento de generación del proxy como *master*— genera una salida que permite ser replicada a través de varios puertos a la vez —conocidas en el momento de generación del proxy como *out*—, de tal manera que disponemos de varios puertos para comunicarnos con el mismo dron. Esto nos ha sido útil para poder enlazar el vehículo con la estación de tierra (QGroundControl en nuestro caso) a la vez que lanzábamos código comunicándonos con el vehículo a través de otro puerto.

3.2.5. OpenCV

La visión por computador es uno de los mayores campos de trabajo debido a su amplia gama de posibles usos, ya sea en vigilancia de viviendas, control del tráfico, de incendios, de calidad de producción, etc. por lo que hemos creído necesario adjuntarla a nuestro proyecto.

Para ello, hemos instalado en nuestro vehículo una cámara junto con su antena transmisora de imagen con la que, desde nuestro ordenador, descodificamos con la antena receptora junto con un descodificador siendo así posible ver lo que el dron en tiempo real. Para ello, utilizamos estas librerías Python capaces de aplicar filtros a una imagen, identificando así los distintos objetos y trabajar con ellos según parámetros.

En nuestro caso hemos trabajado utilizando como filtro el color, aunque podríamos usar siluetas (sean figuras geométricas, números, etc.) en caso de poder profundizar en este campo.

3.2.6. GPSD

Este es el software utilizado internamente en el ordenador para poder tratar los datos recibidos a través de la antena GPS que conectamos a través de un puerto USB. Este software requiere la implantación de un demonio el cual sea capaz de detectar la conexión del dispositivo gracias a la regla de USB visible en la Figura 38: Reglas USB.

3.2.7. Estación de tierra: QGroundControl

Es lo que conocemos como estación de tierra. Esta aplicación —la cual requiere librerías Qt— nos ha permitido de manera cómoda configurar algunos parámetros que, en caso de no disponer de esta aplicación, habrían sido modificados a través de MAVLink, así como también habrían tenido que ser contrastados solicitando la información al dron a través del mismo protocolo para asegurarnos de que el valor es modificado.

Además, esta aplicación nos ha permitido observar la trayectoria del dron (ya sea real o simulado), lo cual nos ha ayudado notablemente a la hora de simular ciertos códigos evitando posibles accidentes. Esta aplicación nos permitía enlazar con el vehículo usando UDP, TCP o mediante alguna conexión serial (telemetría) pudiendo ajustarle los baudios según el funcionamiento del dispositivo. En las Figura 23: Menú principal de los ajustes y Figura **24**: Menú de los parámetros del vehículo podemos observar un par de capturas de pantalla de la aplicación.

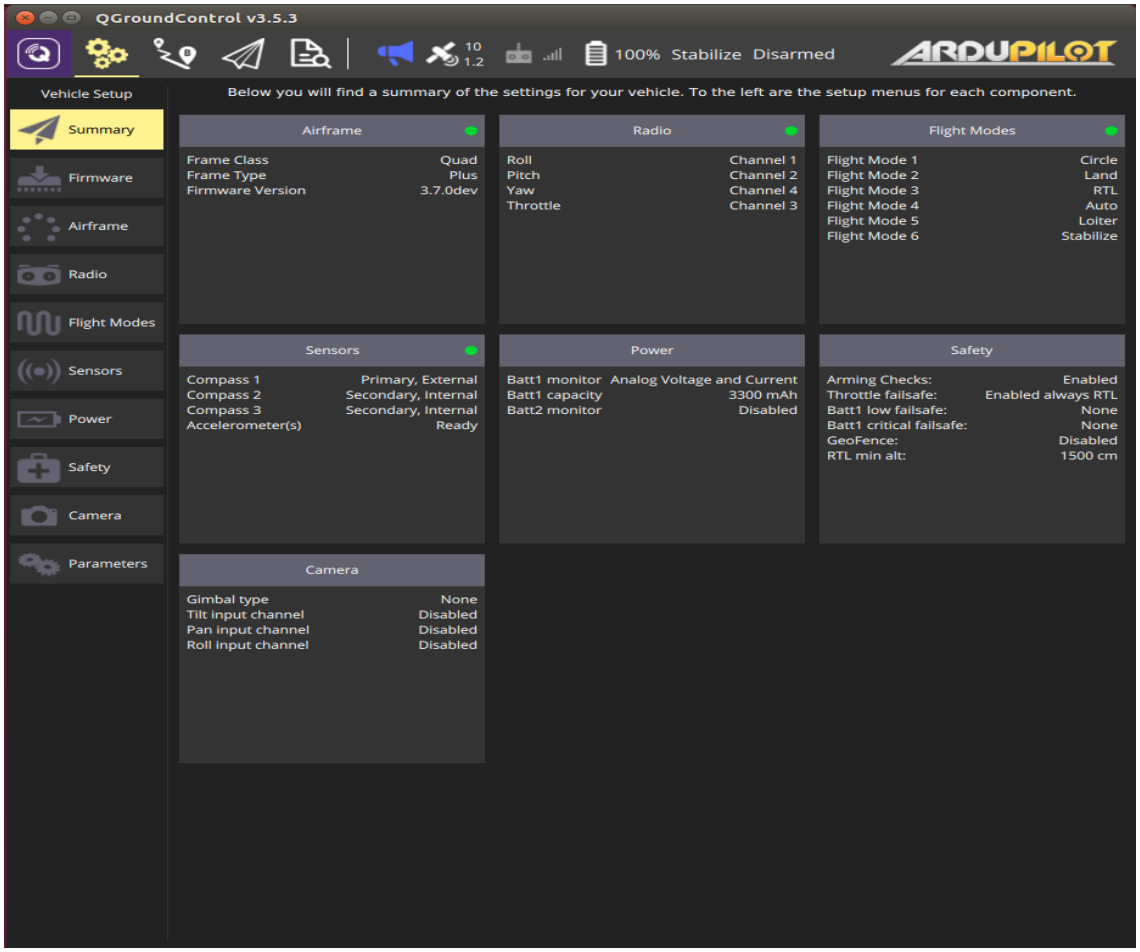


Figura 23: Menú principal de los ajustes

En la Figura 23 encontramos una imagen que representa la pantalla principal cuando tratamos de acceder a los ajustes del vehículo. Esta y las demás pantallas de ajustes aparecerían totalmente despejadas en caso de no estar conectados a ningún vehículo. En este menú principal encontramos un resumen de todas las características

del vehículo estando entre ellas la configuración de la radio, algunos de los parámetros de seguridad o de los modos de vuelo.

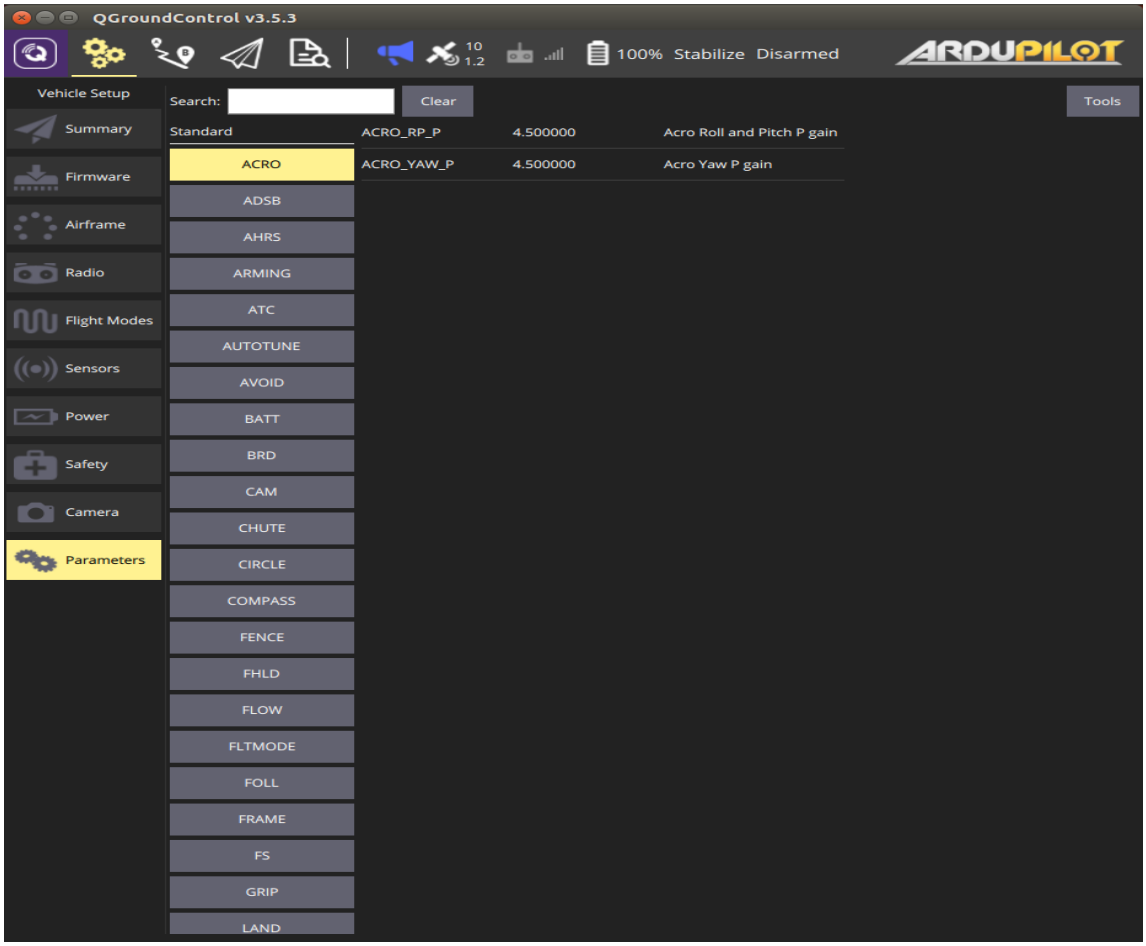


Figura 24: Menú de los parámetros del vehículo

En la Figura 24 encontramos el menú de configuración de todos los parámetros propios del vehículo. Podríamos remarcar en este caso la barra de búsqueda situada encima de esta lista, pues en ella podemos poner alguna palabra significativa filtrándose así los parámetros pudiendo encontrar así más cómodamente la deseada.

3.2.8. Simulador

Ya terminando con esta sección, solo nos falta hablar del simulador, el cual, utilizando las librerías dronekit y dronekit-sitl junto con MAVProxy, es capaz de crear un vehículo virtual capaz de comportarse igual que el vehículo real creando a su vez un proxy con el que podemos comunicarnos.

Dicho simulador no tiene ninguna implementación gráfica en la que podamos apoyarnos para ver el funcionamiento de dicho vehículo, no obstante, sí que podemos ver el comportamiento del dron gracias a que la conexión del proxy es compatible con el programa que utilizaremos para controlar la ubicación, la actitud y las características del dron —QGroundControl—. Aun así, en ocasiones ha sido necesario analizar la actitud del dron a cada instante de ejecución, por lo que también era indispensable que cada código que ejecutásemos mantuviera un reporte constante de dicha actitud a través de un *log*.

Para llevar a cabo nuestros objetivos, hemos añadido al fichero *locations.txt* (ubicado en la carpeta donde se encuentra el simulador) las coordenadas —siguiendo el patrón de las coordenadas ya existentes— del lugar donde pretendíamos realizar las prácticas de vuelo a modo de familiarizarse con dicha ubicación a través de la estación de tierra. Para lanzar el simulador, debemos ubicarnos en la carpeta del tipo de vehículo que queremos simular y lanzamos la siguiente orden:

sudo sim_vehicle.py -L My_Home

Con esta orden, se ejecuta el simulador cargando como ubicación inicial la indicada con el parámetro "-L" y que ha sido guardada en el fichero *locations.txt* propio del simulador. Con el lanzamiento, se crea un proceso que carga la configuración de todo un vehículo obteniéndola de los ficheros ubicados en el directorio "ardupilot/Arducopter". A continuación, con el arranque, se crea un proxy local que proporciona dos puertos a los que nos podemos conectar para interactuar con el supuesto vehículo siendo uno de ellos para la estación de tierra y otro para enviarle los comandos del programa Python.

Una vez terminado el arranque, el sistema emite los mensajes que emitiría el vehículo real, así como es también capaz de simular que consigue obtener cobertura GPS y el desgaste de la supuesta batería. Tras estos mensajes, el simulador queda a la espera de órdenes siendo nosotros incapaces de observar nada si no utilizamos la estación de tierra (QGroundControl) para ver su actitud.

Para interactuar con este vehículo virtual, simplemente debemos lanzar a ejecución en otro terminal el programa que deseamos testear introduciendo como dirección de conexión la IP y el puerto que el simulador nos ha facilitado, siendo en nuestro caso, la dirección local (127.0.0.1) y uno de los dos puertos (14550 o 14551) que el simulador habilita para este uso tal y como podemos observar a continuación:

sudo python vehicle_state.py --connect 127.0.0.1:14551

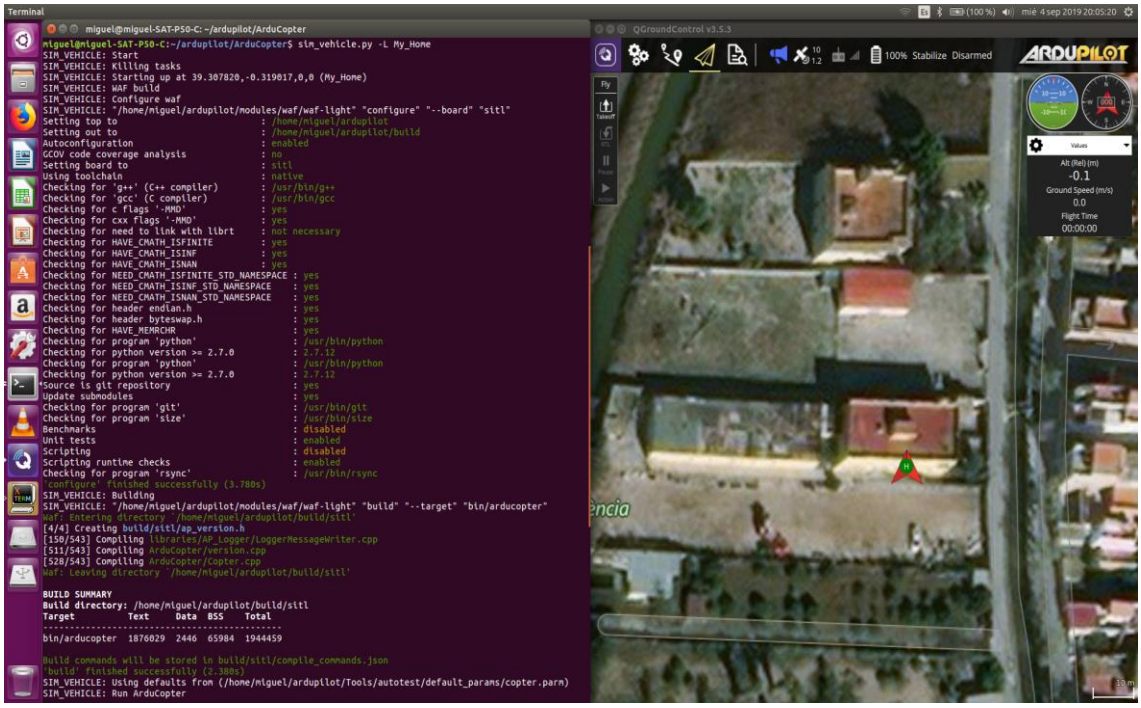


Figura 25: Simulación del dron con reconocimiento desde estación de tierra

4. Análisis y diseño de la solución

A continuación, vamos a analizar desde diversas perspectivas nuestro proyecto. Analizando el problema, respecto a los aspectos de eficiencia tanto energética como algorítmica cabe decir que, dado que siempre trabajamos con librerías ya optimizadas por sus autores, nosotros nos hemos ceñido a desarrollar códigos lo más eficientemente posible para mantener así la mejor eficiencia.

En nuestro caso, debemos realizar un pequeño análisis sobre la seguridad informática del sistema, dado que solo una persona conocedora de la red en la que se encuentra nuestro vehículo sería capaz de interactuar con él, ya que nuestro UAS se encuentra en una red privada ajena a servidores de internet. Además, para poder acceder al dron, es necesario conocer la contraseña de conexión por *ssh*.

Con todo esto, nos faltaría aclarar algunos aspectos de la normativa establecida en nuestro territorio en 2017, como es la restricción de las zonas de vuelo, las cuales podemos consultar en este enlace https://drones.enaire.es/, el cual es una aplicación donde muestra cada una de las zonas restringidas junto con los elementos destacables en este ámbito.

A fin de facilitar la información adjuntamos a continuación una captura de pantalla de dicha aplicación donde podemos ver las zonas de vuelo permitidas a nivel nacional.

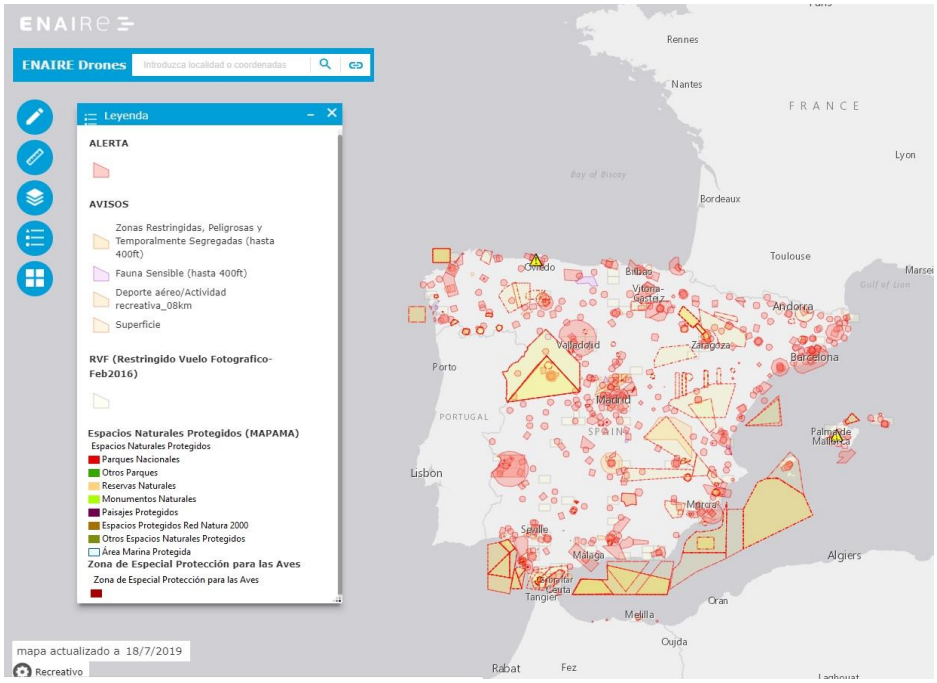


Figura 26: Captura de pantalla de la aplicación ENAIRE

También sería necesario añadir otros conceptos como la exención de obligatoriedad a la hora de requerirse un certificado de pilotaje para drones menores de 25 kilos o, así como indica el artículo 20 en el capítulo 3 del BOE del día 29 de diciembre de 2017 (que se corresponde con el Real Decreto 1036/2017) y que nos limita —debido a la ausencia de licencia— a zonas sobre aglomeraciones de edificios en ciudades, pueblos o lugares habitados o reuniones de personas al aire libre, en espacio aéreo no controlado y fuera de una zona de información de

vuelo (FIZ), únicamente por aeronaves pilotadas por control remoto (RPA) cuya masa máxima al despegue no exceda de 10 kg, dentro del alcance visual del piloto (VLOS), a una distancia horizontal máxima del piloto de 100 m, y a una altura máxima sobre el terreno no mayor de 400 pies (120 m), o sobre el obstáculo más alto situado dentro de un radio de 600 m desde la aeronave."

Otros datos que creemos conveniente citar en este punto sería, siendo que volamos de forma recreativa —ya que de momento no podemos categorizarlo de profesional— la obligatoriedad de volarlo siempre de día y con buenas condiciones meteorológicas, lo cual ha sido, en ocasiones, un problema debido a los fuertes vientos que nos han hecho interrumpir las sesiones de vuelo o la restricción de no exceder de los 10kg en su despegue —en caso de necesitar volar en zonas urbanas— aunque esto no haya supuesto un problema para nosotros ya que no era nuestro propósito, así como la necesidad de acordonar dicha zona o el requisito de tener un sistema de limitación de energía tal como un airbag o un paracaídas además de requerir una autorización procedente de AESA. También, aunque no se nos obligue, se nos recomienda un seguro de responsabilidad civil para poder así subsanar posibles accidentes.

Basándonos en el marco ético y legal, podemos también decir que hemos evitado correr cualquier riesgo que pudiera derivarse de la presencia de gente ajena a nosotros puesto que la velocidad de giro de las aspas podría producir cortes que podrían poner en grave peligro la salud de la gente. Por el mismo motivo, a la hora de realizar las pruebas, siempre lo hemos colocado a una distancia bastante prudencial para evitar que cualquier comportamiento anómalo causado por cualquier problema electrónico pudiera ocasionarnos daños a cualquiera de nosotros.

Dado que, tanto ArduPilot como Dronekit, son capaces de responder a los comandos que reciban, bien desde la radio o bien desde el ordenador, hemos optado por diseñar un código que dotara de autonomía al vehículo a modo de prueba para un posible futuro desarrollo a nivel empresarial que fuera capaz de perseguir algún objeto sin almacenar ningún dato. Es por ello, que barajamos como posibilidades definir un contorno o definir un color. Terminamos decantándonos por la definición de un color puesto que un contorno puede ser muy semejante a otro, y más, si se trata de un turismo, pero el color es más único debido a que la pintura de cada turismo se deteriora de manera distinta dependiendo de su exposición al sol.

El plan de trabajo a seguir ha sido, a grandes rasgos, que se trabaje de manera autónoma durante toda la semana añadiendo una reunión semanal en la que se iban aclarando dudas, se indicaban futuros pasos a seguir y se concretaba la fecha de la siguiente reunión. Aun así, también hemos mantenido contacto a través de aplicaciones de mensajería móvil para algún caso especial que pudiera paralizar durante el resto de semana el desarrollo de las tareas asignadas. La planificación ha sido la siguiente:

<table>
<tr><th colspan="4">TABLA DE PLANIFICACIÓN</th></tr>
<tr><th>MES</th><th>SEMANA</th><th>SOFTWARE</th><th>HARDWARE</th></tr>
<tr><td rowspan="4">FEBRERO</td><td>1</td><td rowspan="2">Instalar del sistema operativo en la Raspberry y en el PC</td><td rowspan="2">Montaje inicial: Tornillos y soldadura</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td><td rowspan="2">Instalar librerías básicas: Dronekit, Python, Configuración en red</td><td rowspan="2">Comprar y montar separadores y tornillos</td></tr>
<tr><td>4</td></tr>
<tr><td rowspan="4">MARZO</td><td>1</td><td rowspan="2">Leer API de Dronekit Probar: vehicle_state.py</td><td rowspan="2">Buscar patas en 3D</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td><td rowspan="2">Estudiar ejemplos Dronekit Modificar ejemplos para simularlos</td><td rowspan="2">Imprimir patas 3D</td></tr>
<tr><td>4</td></tr>
<tr><td rowspan="4">ABRIL</td><td>1</td><td rowspan="2">Configurar del dron Configurar el mando Familiarizarse con el mando</td><td rowspan="2">Pruebas de funcionamiento de la cámara en un televisor Instalar la cámara</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td><td rowspan="2">Configurar QGroundControl Primeras pruebas de vuelo</td><td rowspan="2">Aislar el GPS con una plancha de cobre para crear una jaula de Faraday</td></tr>
<tr><td>4</td></tr>
<tr><td rowspan="4">MAYO</td><td>1</td><td rowspan="2">Implementar códigos de movimiento básicos Testear en simulador</td><td rowspan="12">Revisar estructura Reforzar estructura Mantenimiento de la batería</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td><td rowspan="2">Implementar códigos de visión por computador Testear en simulador</td></tr>
<tr><td>4</td></tr>
<tr><td rowspan="4">JUNIO</td><td>1</td><td rowspan="2">Implementar códigos de movimiento complejos Testear con el simulador</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td><td rowspan="2">Crear MAVProxy a través de UDP Intento fallido de crear MAVProxy a través de tty</td></tr>
<tr><td>4</td></tr>
<tr><td rowspan="4">JULIO</td><td>1</td><td rowspan="4">Pruebas definitivas de vuelo Crear MAVProxy a través de tty (telemetría)</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>AGOSTO</td><td>Todo el mes</td><td>Redactar memoria</td></tr>
</table>

De este modo podemos observar cómo hemos ido alternando tareas físicas junto con las virtuales de modo que no solamente destinábamos el tiempo semanal a una única tarea. Así, conseguíamos que, en caso de quedar atascados con alguna duda, siempre hubiera algo por hacer manteniéndonos de esta manera ocupados hasta la siguiente reunión.

En relación con el presupuesto, el coste total del proyecto —visible en el Anexo A - Presupuesto— asciende a menos de 969,60 €, lo cual establece el mínimo precio al que deberíamos poner nuestro producto si quisiéramos ponerlo en el mercado "a precio de coste". En un caso más realista, suponiendo un salario medio anual de 25.363 € (según *Indeed*), calculando 160 horas al mes, nos sale un salario de 13,20€ por hora. Por lo que suponiendo que hemos realizado las 300 horas del proyecto, el precio final del producto completo sería 969,60 + (13,20 * 300) = 4.929,60 €.

Concepto	Cantidad	Precio	Subtotal
Producto	1	969,60	969,60 €
Precio por hora	300	13,20	3.960,00 €
	Total		4.929,60 €

A continuación, podemos apreciar en la Figura 27: Esquema software de nuestro diseño un esquema que representa los distintos componentes de un modo más esquematizado con el que podemos hacernos una mejor idea de qué tiene cada uno de nuestros elementos.

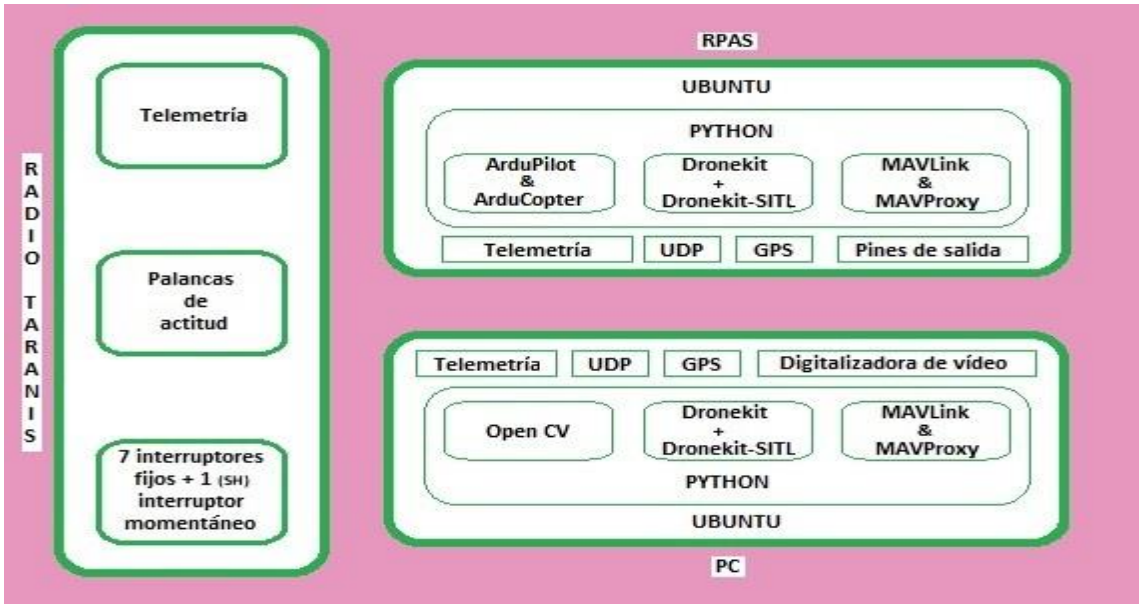


Figura 27: Esquema software de nuestro diseño

Como se puede apreciar en la Figura 27, existen 3 bloques fundamentales para el correcto desarrollo de este proyecto. En primer lugar, tenemos la radio, el mando de control remoto Taranis X9D, la cual dispone de una telemetría incorporada para comunicarse directamente con el dron. Además, dispone de un par de palancas que se pueden desplazar en dos direcciones, lo cual nos permite trabajar con cuatro

variables simultáneamente de modo que afectarán a la actitud del vehículo. Como último aspecto en referencia al mando, cabe destacar los 8 interruptores que también incluye, de entre los cuales, existen siete con comportamiento fijo y otro momentáneo. Esto nos ha sido de mucha utilidad para administrar los modos de funcionamiento del vehículo para así, tener un control seguro del dron en todo momento evitando que tras un mal comportamiento se alejase de nuestro alcance impidiendo así la recuperación del hexacóptero.

En segundo lugar, tenemos el portátil, el cual ha sido indispensable para la configuración de la Raspberry que el vehículo incorpora, la implementación de código y la simulación de este de manera previa a la experimentación con el dron real. Este ordenador, con un sistema operativo basado en Ubuntu, dispone de las librerías Python 2.7 sobre las que se ha instalado todo lo necesario para llevar a cabo nuestros objetivos: Dronekit y Dronekit-SITL para el desarrollo comportamental del dron, MAVLink y MAVProxy para la creación del proxy que nos ha facilitado la interacción con el dron utilizando a la vez varios puertos los cuales eran requeridos por diversas aplicaciones y, por último, OpenCV para poder tratar las imágenes recibidas de la digitalizadora de video.

Además de todo esto, disponemos también de unos componentes hardware sin los que algunos de nuestros objetivos no podrían haberse realizado. Ellos son: la digitalizadora de vídeo recientemente mencionada, el GPS —con el que el dron conocía nuestra posición (en este punto profundizaremos en los puntos posteriores)— y la telemetría, con la que nos hemos comunicado con el dron y a través de la cual hemos conocido su estado. Antes de pasar al siguiente punto, es necesario referirnos a UDP, pues en los momentos previos a la comunicación con el dron a través de telemetría, este protocolo ha sido indispensable para realizar tanto simulaciones como experimentación real. Así pues, cuando conseguimos establecer la comunicación por telemetría con el vehículo, los puertos locales a través de los que el MAVProxy retransmitía los datos recibidos, utilizaban UDP como protocolo de transporte.

En tercer lugar, y no por ello el menos importante, tenemos el bloque RPAS, que se corresponde con nuestro vehículo, el cual contiene una Raspberry que, bajo un sistema operativo Ubuntu especialmente diseñado para ella, regirá todo el vehículo a través de sus pines de salida cuyas señales son transmitidas a sus respectivos variadores, los cuales convierten dichas señales en un pulso eléctrico capaz de ser interpretado por cada motor de modo que hará girar su respectiva hélice a cierta velocidad.

En este bloque contamos también con un conjunto de librerías Python 2.7 sobre el que se apoyan Dronekit y Dronekit-SITL para poder interactuar con ArduCopter (y, por tanto, ArduPilot) y con MAVProxy. En este punto es preciso mencionar también la participación de UDP debido a la interacción con MAVProxy y, por tanto, con las comunicaciones entre aplicaciones y servicios.

Respecto al hardware, encontramos también el GPS, el cual se encontrará en la parte más alta del cuerpo del vehículo para así tener una mejor cobertura, y, la telemetría, aunque en este caso sería más apropiado referirnos a las telemetrías, dado que disponemos de dos: una para que pueda comunicarse con el mando y otra para comunicarse con el PC.

Con estos tres bloques, el diseño de la solución ha sido, en primer lugar, configurar el controlador por radio y, una vez realizada esta tarea, trabajar directamente en el ordenador, simular su funcionamiento con SITL y, una vez corroborado el correcto funcionamiento, experimentarlo en el vehículo. Para ello, en primer lugar, hemos hecho pruebas de comunicación gracias a un código que nos ha proporcionado Dronekit denominado “vehicle_state.py”, el cual le solicita información al vehículo y la transmite por pantalla.

Una vez conseguido este punto, experimentamos con otro código facilitado por Dronekit, el denominado “follow_me.py” el cual, a través del GPS que conectamos a nuestro ordenador, el dron se encarga de desplazarse en cada momento a la posición que el programa le facilita. Posteriormente, basándonos en el código del ejemplo denominado “set_attitude_target.py”, realizamos varios ejercicios en los cuales alterábamos cada uno de los parámetros que afectan a la actitud del vehículo para, seguidamente, unirlos de manera que se desplazase de un punto a otro siempre de cara al objetivo —reconociendo la cara como el espacio comprendido entre los dos brazos de color rojo del dron—. Tras esto, le instalamos la cámara, adaptamos parte del código y experimentamos con el simulador para comprobar que era capaz de seguir un tapón en la ruta que estuviera haciendo, ya fuera *in situ* o persiguiéndolo. Todo este proceso va a ser detallado a continuación.

5. Desarrollo de la solución propuesta

Llegados a este punto, es el momento de detallar todo el proceso separando por bloques cada parte del trabajo realizado, así como ciertos detalles que han ido quedando pendientes a lo largo del escrito y que hemos reservado para este momento, pues ayudarán a entender la interrelación existente.

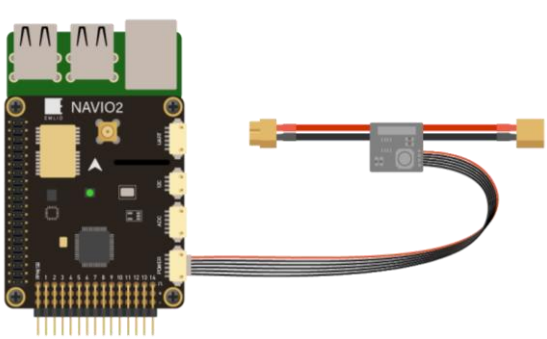
5.1. Montaje del dron

Inicialmente, tras desempaquetar todos los elementos, tuvimos que atornillar cada brazo a una de las planchas que pasaría a ser la “panza” del dron. El orden seguido para instalar estos brazos no es importante, no obstante, sí lo es la presencia de dos brazos en concreto: los rojos, ya que estos dos —a pesar de no tener ninguna función especial— acordamos usarlos para indicar donde se sitúa la parte delantera de nuestro vehículo dado que, si no lo aclarásemos de ninguna forma, no sería posible saber si el dron está comportándose correctamente o no.

Seguidamente, procedimos a instalar una cuadrado de velcro en la parte interna de la plancha, ya que esto nos servirá en los próximos pasos como sujeción de la Raspberry. Una vez hecho esto, utilizando una impresora 3D, imprimimos la carcasa de dos piezas que protegería la Raspberry. Con todo este material, unimos la placa Emlid a la Raspberry conectando a ésta última una tira doble de zócalos que vienen incluidos dentro de la placa Emlid Navio2 y sobre éstos acoplamos la placa Emlid, encapsulamos este bloque dentro de la carcasa y tratamos de cerrarla a base de tornillos. Aquí se presenta la primera complicación, dado que el tamaño del orificio requiere una métrica de 2.5mm, la cual es poco común —por lo que tuvimos que buscar en ferreterías especializadas— y, al menos 3mm de longitud para alcanzar la placa.

Figura 28: Soldadura del cableado e instalación del velcro

Dado que había que atornillar la plancha por ambas partes y, dado que los tornillos debían fijar tanto la Emlid como la Raspberry —como se puede observar en la Figura 31—, optamos por adquirir también unos separadores hembra-hembra con métrica de 2.5mm y una longitud de 12mm de modo que cuando los tornillos adquiridos atravesasen la carcasa protectora, entrasen al separador consiguiendo así mantener rígida la pieza. Estos tornillos finalmente fueron de una longitud de 6mm, con lo que conseguían atravesar la carcasa, atravesar la placa y penetrar en el separador sin impedir que el otro entrase en ella.



(a) Conexión de la Navio2 a la fuente de alimentación

(b) Punto de acceso de la alimentación a la base del dron

Figura 29: Alimentación a la Emlid Navio2 y a los variadores

Posteriormente decidimos instalar a la parte inferior de la carcasa una base contra vibraciones a la que los tornillos atravesaron también sin problemas. Seguidamente, a través del orificio ubicado en la carcasa que corresponde con la placa Emlid, conectamos a ésta la antena GPS a través del agujero visible en la Figura 29(a) el cual se corresponde con el enganche saliente que se puede apreciar en la Figura 31**¡Error! No se encuentra el origen de la referencia.** al que acompaña una flecha que debe estar orientada en el mismo sentido que la de la placa visible en la Figura 29(a). Una vez encapsulado, conectamos a la Emlid la alimentación a través del puerto Power y la telemetría que interactúa con el PC a través del puerto UART siendo, por tanto, Emlid la que alimenta a la Raspberry a través de los pines interconectados gracias a la tira de zócalos que los unifica.

Para poder alimentar la Navio2 desde la batería, era necesario utilizar el dispositivo visible en la Figura 29(a), pues gracias a esto, la Navio2 tiene acceso a la energía aun permitiéndole de igual manera llegar hasta la placa —como podemos ver en la zona central de la Figura 29(b)— donde debe suministrar energía a variadores y, por tanto, a motores.

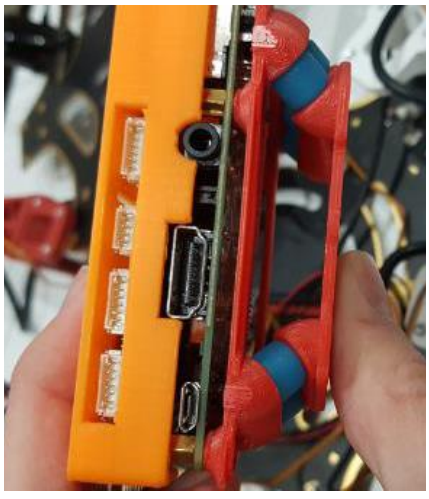


Figura 30: Base contra vibraciones (roja) con amortiguadores (azules)

A la parte inferior de la base contra vibraciones —visible en la Figura 30 y formada por dos planchas creadas con una impresora 3D que están unidas a través de unos amortiguadores (de color azul) para que absorban posibles vibraciones y caídas que puedan afectar a este conjunto— instalamos la parte contraria al velcro que

habíamos instalado previamente en la base del dron, con lo que conseguimos mantener fijada la pieza. Tras esto, instalamos a una longitud aproximada de la mitad de cada brazo uno de los variadores fijándolo con una brida y atornillando al extremo de cada brazo un motor de tal manera que cada motor debe girar en sentido opuesto al de sus vecinos. Tras esto, enrollamos el cable sobrante alrededor del brazo y soldamos el extremo a la placa base respetando la polaridad de cada cable.

(a) Acoplamiento de la Emlid Navio2 con la Raspberry Pi 3 Model B

(b) *Carcasa 3D para el bloque "Raspberry - Emlid"*

Figura 31*: Acople y carcasa de Emlid Navio2*

Debido a que la tarjeta Emlid es sensible a la posición en la que se encuentre, ha sido vital ubicarla lo más encarada posible a la cara delantera, por lo que aquí ya nos ha sido de utilidad las dos patas rojas que indicaban la zona frontal. Para ello, tan solo tuvimos que acoplar el bloque “Raspberry – Emlid” al velcro situando la flecha de la carcasa (visible en la Figura 29(a)), así como la flecha pintada en la Navio2 visible en la Figura 31(b), apuntando hacia el punto medio entre ambas patas rojas.

Habiendo llegado ya a este punto, es el momento de configurar el mando Taranis —cuyo proceso aparece extendido en el anexo B— y de realizar la conexión de cada variador con los pines de la Raspberry siguiendo el orden indicado en la Figura 32 debido a que las salidas demultiplexadas siguen dicho orden.

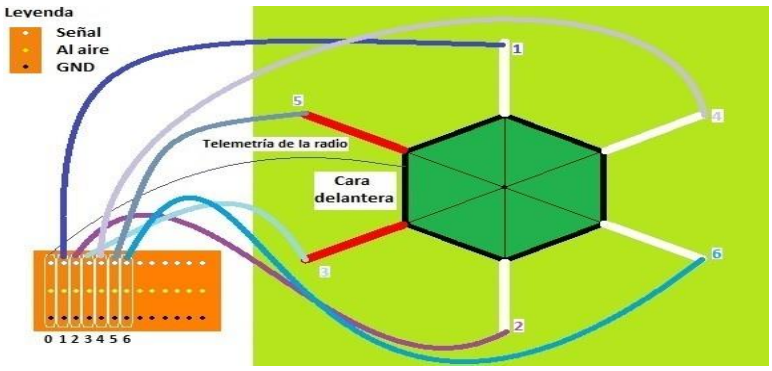


Figura 32: Conexionado de los pines de la placa Emlid Navio2

Con ella podemos ver que —como ya habíamos dicho— la radio ocupa la primera columna de pines y que la siguen los variadores siguiendo el orden de acuerdo con las líneas trazadas acompañadas del número que indica la posición en las columnas. Apoyándonos en la Figura 32, también podemos observar otro detalle, la columna central de pines está al aire, es decir, no está conectada a ningún cable y

esto es debido a que los variadores reciben la alimentación de la propia placa del dron gracias a las soldaduras realizadas por nosotros.

Hecho esto, es el momento de proceder a calibrar cada variador. Para ello, siempre sin hélices en ningún motor, basta con desconectar el variador de la Raspberry y conectarlo directamente al canal 3 de la radio, ya que dicho canal se corresponde en nuestro caso con *thrust*, es decir, la potencia del motor a la vez que dejamos la palanca de *thrust* (en el mando Taranis) a su valor máximo. Una vez hecho esto, se conecta la batería para encender el dron, de modo que la luz verde de la Raspberry debería empezar a parpadear, lo cual significa que la próxima vez que se encienda entrará en modo calibración, por lo que debemos apagar el dron y volverlo a encender (manteniendo la palanca de *thrust* a su valor máximo).

Cuando se oiga un sonido característico de la calibración, bajamos la palanca hasta su valor mínimo, momento en el que el dispositivo emite otro sonido el cual significa que la calibración ha finalizado con éxito. Llegados a este punto, solo queda probar si el motor responde correctamente levantando ligeramente la palanca de *thrust* para comprobar si realmente el variador es sensible a diferentes potencias reaccionando distintamente a cada valor. Como último paso para calibrado, basta con quitar la alimentación del dron y devolver la conexión del variador a su set de pines correspondiente.

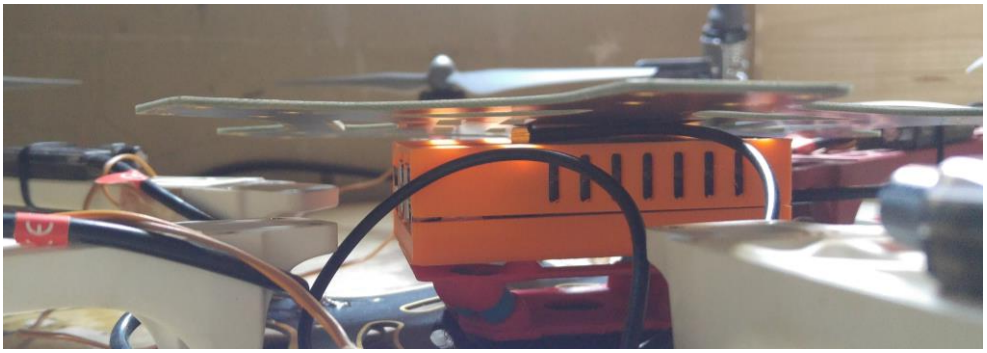


Figura 33: La placa superior no consigue alcanzar los orificios de los brazos

A continuación, tuvimos que adquirir unos separadores macho-hembra de 12mm de longitud para acoplarlos a la parte superior de los brazos del robot, ya que la carcasa del bloque “Raspberry – Emlid” junto con la base anti-vibraciones y el cable de la antena GPS superaban 1cm la altura del brazo del dron como podemos apreciar en la Figura 33, por lo que no podíamos cerrar la tapa superior. Aquí tuvimos un contratiempo debido a que uno de los separadores, al ser de nylon, se rompió al intentar enroscarlo en el orificio debido al calentamiento por la fricción entre los materiales. Es por ello por lo que tuvimos que, con ayuda de una taladradora muy fina que nos prestaron, forzar la salida del material pudiendo así sustituirlo, pero esta vez de un modo pausado para evitar que se calentase.

Una vez alcanzado este punto, atornillamos la placa superior al dron con los 24 tornillos que disponíamos, instalamos un velcro en la zona central y pusimos la contraparte del velcro a la batería para que así estuviese fijada durante los vuelos. Dado que en algunas ocasiones se producían interferencias con el GPS, decidimos adquirir una plancha de cobre a la que soldamos un cable que conectaba con masa. A esta plancha se pegaría el GPS con velcro para evitar darle calor a un dispositivo tan sensible a la vez que evitamos interferencias provocadas por la propia masa. De este

modo se generaría una jaula de Faraday sobre la plancha aislando el GPS de las ondas que circulan por debajo de él como la telemetría, pero manteniendo la comunicación con los satélites. A la parte inferior de dicha plancha se acoplaría otro velcro de modo que se pudiera anexionar a la parte superior de la batería quedando así en la parte más alta del dron.

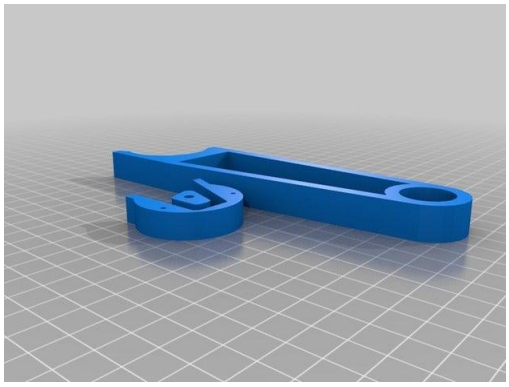


Figura 34: Pieza 3D que se corresponde con cada una de las patas.

Para poder alzar el vuelo, quisimos imprimir unas patas 3D que le permitieran aterrizar a cierta altura para así evitar que aterrizase sobre su plancha inferior pudiendo dañar algún componente como puede ser alguna antena. Por ello, buscamos entre las disponibles a través de la red decantándonos por la que aparece en la Figura 34, ya que esta pieza proporciona una fácil instalación, con lo que tras varios días acudiendo a la impresora 3D conseguimos obtenerlas todas.

La instalación de la cámara ha sido un sencillo paso debido a que hemos pasado a través de dos de sus orificios visibles en la imagen superior izquierda de la Figura 20 una brida, de modo que cuando le fijamos una posición no puede moverse de esta, quedando así fijada. Respecto a la alimentación, está conectada al transmisor de vídeo a través de los pines que éste tiene reservados para dicho fin, mientras que el transmisor es alimentado directamente desde la batería conectando cables de puente a los conectores balanceadores primero y cuarto empezando desde GND siendo el primero GND y el cuarto el que transmite una tensión de 11.1V.

Tras una serie de pruebas de vuelo, tuvimos que sustituir todos los variadores, pero en el momento de adquirir otros nos percatamos de que el fabricante ya no distribuía este dispositivo, por lo que tuvimos que adquirir la nueva versión que sería más compacta, pero con mayor amperaje. Con esto, tuvimos que desoldar todos los variadores, limpiar la superficie de soldadura, soldar los nuevos y recalibrar todo el dron quedando como se puede observar en la Figura 9.

5.2. Instalación del software

5.2.1. Configuración del computador

Respecto a la configuración software, en primer lugar tratamos de usar la versión más reciente disponible de Ubuntu (18.04), dado que queríamos utilizar la tecnología más reciente y adaptarnos a ella, pero en alguna de las instalaciones se requería de ciertas dependencias cuyas instalaciones no se podían llevar a cabo debido a la obsolescencia de algunas de ellas, por lo que tuvimos que retroceder a la versión 16.04 LTS —cuyas siglas se refieren a *Long Term Support* (Soporte a Largo Plazo)— lo cual representa que lo que llegásemos a hacer podría utilizarse durante un tiempo considerable ya que esta versión de Ubuntu no caería en desuso por falta de soporte en algún tiempo. Siendo pues así, procedemos a conocer cada uno de nuestros pasos:

Tras la instalación de nuestro sistema operativo, el primer paso es dotar de conexión a Internet a nuestro computador a través de la interfaz que el propio sistema operativo nos facilita. Una vez conectados exitosamente, actualizamos nuestra distribución para actualizar el sistema con la versión más reciente de cada paquete utilizando el comando:

sudo apt-get update & sudo apt-get dist-upgrade

A continuación, procedimos a la instalación del editor de texto vim, del intérprete Python que necesitábamos, pues ArduPilot es únicamente compatible con la versión 2.7, así como ipython para algunas pruebas sencillas y ocasionales sin la necesidad de crear códigos enteros, por lo que ejecutamos:

sudo apt-get install vim python2.7 ipython

Seguidamente, instalamos *pip*, un controlador de paquetes de Python que facilita la obtención e instalación de los paquetes necesarios. Para ello ejecutamos:

sudo apt-get install python-pip

Previamente a instalar las librerías es recomendable (y en nuestro caso necesario para algunas librerías) actualizar la version de pip, para ello lanzamos:

sudo pip install --upgrade pip

Una vez instalado y actualizado, podemos descargar las librerías necesarias para que Python pueda interpretar el código de dronekit. Para ello ejecutamos:

sudo pip install pymavlink dronekit dronekit-sitl opencv-python

Gracias al reporte del log, podemos observar que no solo instala las librerías, sino que también busca las dependencias propias de cada librería, las descarga y las instala para evitar problemas futuros.

Con esto, ahora solo queda descargar de la página oficial de Raspberry (https://www.raspberrypi.org/downloads/) el sistema operativo Raspbian para instalarlo posteriormente en la Raspberry, además de crear un punto de acceso del que el propio computador sea el punto de acceso.

Para la creación de dicho punto de acceso hay que seguir una serie de pasos:

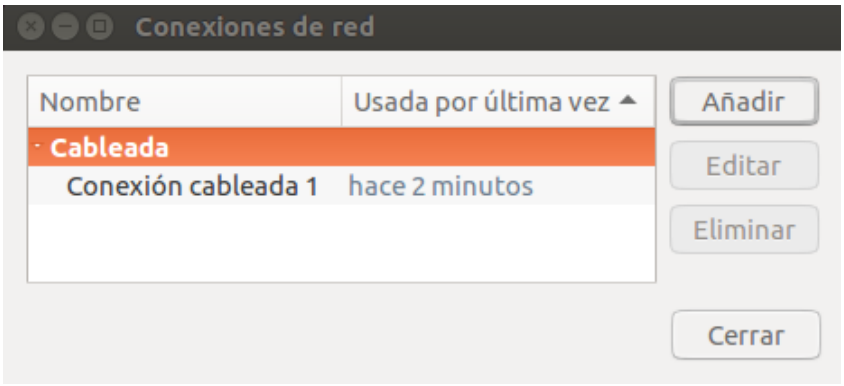


Figura 35: Menú de conexiones de red

En primer lugar, haremos click en el icono perteneciente a la red situado en la esquina superior derecha de la pantalla, seleccionamos “Editar conexiones las conexiones…” y elegimos “Añadir”.

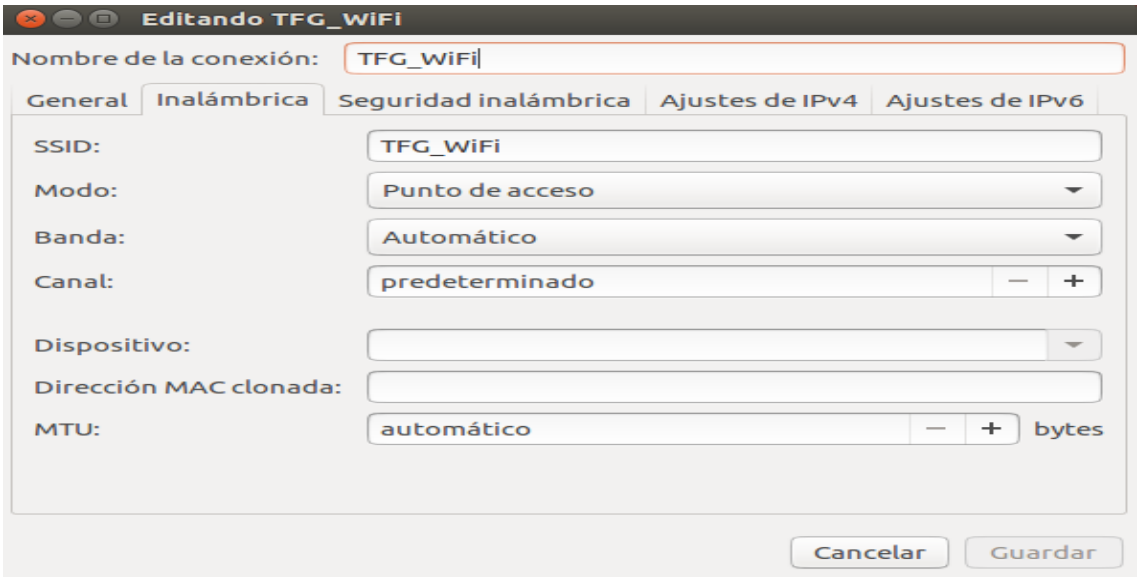


Figura 36: Menú de la creación de la red.

Seguidamente, en el menú que nos aparece en la Figura 36, seleccionamos el tipo de red que queremos crear. En nuestro caso, y dado que queremos crear una red Wi-Fi privada para poder conectarnos al dron, seleccionaremos la opción “Inalámbrica” y pasamos al siguiente punto seleccionando “Crear…”. En este menú, rellenaremos (como mínimo) el campo SSID —que será a partir del punto de creación el nombre de la red a la que tratará de conectarse el dron— y nos aseguramos de que la opción “Modo:” tenga el valor “Punto de acceso”. Recomendamos también cambiar el “Nombre de la conexión” ya que es el nombre con el que la veremos en nuestro menú a partir del modo de su creación. Este último valor es únicamente apreciable a nivel local para diferenciar entre redes que pudieran llamarse igual.

Habiendo realizado estos pasos, podemos continuar observando en la Figura 37 la contraseña que le hemos asignado a nuestra red para evitar que cualquiera pueda interferir en nuestro sistema pudiendo así tomar el control de nuestro vehículo. Por defecto la red tiene asignada la opción “Seguridad: Ninguna”, por lo que debemos cambiarla a un modo más seguro como es WPA2 y añadiendo una contraseña que

contenga a ser posible la mayor variedad de caracteres estando entre ellos: números, letras mayúsculas, letras minúsculas y símbolos.

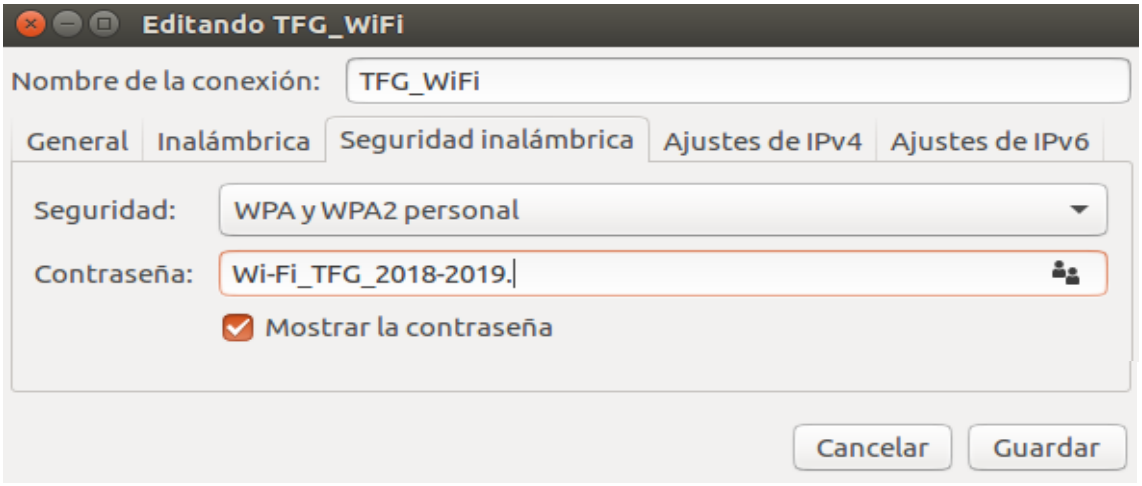


Figura 37: Menú de seguridad de la red.

Como último paso para terminar de configurar nuestra red, basta con seleccionar la opción "Guardar" y la red ha sido creada. En caso de que se desee comprobar el correcto funcionamiento de la red, se puede conectar cualquier dispositivo a ella, consultar su IP y tratar de hacer ping a este dispositivo y comprobar que se reciben los paquetes sin pérdidas.

Por otro lado, hemos tenido que crear un fichero en el computador en el que definir las reglas por las que deben regirse los dispositivos al conectarse al sistema. Este fichero —ubicado en */etc/udev/rules.d/* y con extensión *.rules*— es el que busca *udev* para gestionar los dispositivos que el usuario agrega o sustrae al sistema en caliente.

Para poder configurar este fichero, hay que conocer previamente los parámetros del dispositivo que pretendemos controlar a fin de poder aplicar un filtro al sistema de modo que solo aplique la regla si se cumple algún requisito el cual se aconseja que sea lo más específico posible. En nuestro caso, hemos decidido filtrar tanto por el identificador del fabricante como por el identificador del producto. Para ello, y dado que todos los dispositivos que hemos necesitado filtrar se conectan por USB, ejecutamos desde el terminal:

lsusb -t

lsusb -D /dev/bus/usb/NNN/XXX

Con la primera orden, logramos conocer todos los dispositivos conectados a través de USB a nuestro sistema. Con ello, conocemos el número de bus al que está conectado cada dispositivo y qué identificador se le ha asignado. Dichos valores, que se corresponden con *NNN* y *XXX* respectivamente en la segunda orden, nos han permitido acceder a los datos del dispositivo deseado (en nuestro caso la antena de telemetría) de modo que hemos podido obtener los identificadores de vendedor y de producto —*idVendor* e *idProduct*— del mismo con los que hemos podido crear la siguiente regla:

```
#Bus 001 Device 054: ID 10c4:ea60 Cygnal Integrated Products, Inc. CP210x UART Bridge / myAVR mySmartUSB light
#Bus 001 Device 055: ID 067b:2303 Prolific Technology, Inc. PL2303 Serial Port

SUBSYSTEM=="tty",ATTRS{idVendor}=="10c4",ATTRS{idProduct}=="ea60",SYMLINK:="ttyTelemetry"
SUBSYSTEM=="tty",ATTRS{idVendor}=="067b",ATTRS{idProduct}=="2303",SYMLINK:="ttyGPS"
```

Figura 38: Reglas USB

Con ella, conseguimos crear un enlace simbólico al tty correspondiente a la telemetría cuyo nombre queda previamente fijado por la variable ------- de la regla de modo que conseguimos la capacidad de poder llamar siempre a nuestra antena a través del nombre *ttyTelemetry* sin necesidad de preocuparnos si el sistema le asigna ttyUSB0 o cualquier otro como, por ejemplo, ttyUSB1. Es por esto que también definimos reglas para aquellos dispositivos que puedan interferir en nuestro proyecto, como es el GPS que utilizamos para las pruebas de seguimiento.

5.2.2. Configuración de la Raspberry

Llegado este momento abrimos una terminal en nuestro computador y ejecutamos:

lsblk

Lo cual nos mostrará todos los bloques de memoria conectados a nuestro ordenador. Insertamos nuestra tarjeta SD, bien a través del puerto apropiado para ello si nuestro computador dispone de él o, en caso contrario, a través de un lector de tarjetas SD que se conecte al PC a través de un puerto USB y volvemos a ejecutar el comando. En esta segunda ejecución aparecerá un bloque adicional el cual será nuestra tarjeta SD.

Estos bloques de memoria pueden aparecer listados de dos modos: */dev/mmcblk0* o bien */dev/sdX*, donde la X es una letra minúscula y cuyas particiones se representan como: */dev/mmcblk0pN* o */dev/sdXN* respectivamente siendo esta vez N un número. A la derecha de cada bloque de memoria puede aparecer una ruta la cual significa dónde está montado dicho bloque de memoria. En caso de aparecer en blanco, significa que no está montado. Antes de continuar se recomienda desmontar las otras particiones que no estemos utilizando, en caso de que alguna de ellas esté montada utilizando siguiente comando (en el que utilizamos las mismas siglas que en los ejemplos anteriores):

umount /dev/sdXN

Es ahora el momento en el que debemos transferir la imagen descargada a la tarjeta SD. Para ello hay que cerciorarse de que conocemos todas las rutas involucradas para no cometer errores, puesto que si, por ejemplo, en lugar de escribir los datos en la ruta */dev/sda3*, los escribiésemos en la ruta */dev/sda*, perderíamos toda la información almacenada en este bloque. Teniendo esto claro, pasamos a teclear la siguiente orden:

sudo dd bs=4M if=<Nombre del fichero .img> of=<Ruta de la tarjeta> conv=fsync

Donde *dd* corresponde al comando que copia un fichero convirtiéndolo en el formato indicado por *conv* usando bloques de tamaño 4MB (*bs*) tomando como entrada el fichero indicado en el parámetro *if* (*input file*) y como salida el fichero indicado en el parámetro *of* (*output file*) los cuales deben indicarse con la ruta completa en caso de no localizarse en la ubicación sobre la que se encuentre el terminal en dicho momento.

A continuación, antes de instalarla en la Raspberry, hemos procedido a indicarle cual es la red a la cual debe conectarse al inicio. Para ello, montamos la tarjeta SD en una carpeta del escritorio de nuestro computador con las órdenes:

mkdir ~/Escritorio/Raspberry

sudo mount /dev/sdb1 ~/Escritorio/Raspberry

Con ello, disponemos ahora de todo el sistema de ficheros de la tarjeta SD enraizado en la carpeta Raspberry creada en el escritorio. Siendo así, el siguiente paso es acceder al fichero ubicado dentro del sistema de ficheros de Raspberry

/etc/wpa_supplicant/wpa_supplicant.conf para indicarle aquí los datos de la conexión que debe establecer con el PC. Para ello, tecleamos la siguiente orden:

sudo vim ~/Escritorio/Raspberry/etc/wpa_supplicant/wpa_supplicant.conf

Y modificamos el fichero de manera que quede:

```
network={
    ssid=TFG_WiFi
    psk=Wi-Fi_TFG_2018-2019.
}
```

Aprovechando el momento, decidimos realizar una copia del fichero a la que denominamos *wpa_pro* (de provisional) para que no hubiese posibles solapamientos de nombres y a la que le suministramos los datos de la red que nos da acceso a Internet con la intención de alternarlas para la instalación de los futuros paquetes sin necesidad de estar extrayendo la tarjeta de la Raspberry en cada ocasión.

Habiendo finalizado este proceso, salimos del sistema de ficheros de la tarjeta SD ubicado en la carpeta Raspberry y lo desmontamos con la orden:

sudo umount /dev/sdb1

A continuación, ya podemos insertar nuestra tarjeta en la Raspberry para que no vuelva a ser extraída, por lo que la instalamos —tal y como ya se ha comentado en puntos anteriores— y conectamos la batería al conector XT60 soldado a la base del dron para alimentar todo el sistema. En cuanto arranca el sistema, se oyen unos pitidos intermitentes los cuales se detienen cuando el sistema al completo ha terminado de inicializarse. Estos pitidos provienen de los variadores y significan que el dron —junto con la Raspberry— se está inicializando, así como comprobando que todas las conexiones están realizadas correctamente dado que, si un variador está conectado a la alimentación, pero no está conectado a la tarjeta Emlid para recibir las señales, no deja de sonar.

Una vez inicializado, ya podemos conectarnos a nuestro vehículo. Para ello, abrimos un terminal en nuestro computador y nos conectamos a través de ssh usando los valores por defecto del sistema (usuario: pi@navio.local y contraseña: raspberry) ejecutando el comando:

sudo ssh pi@navio.local

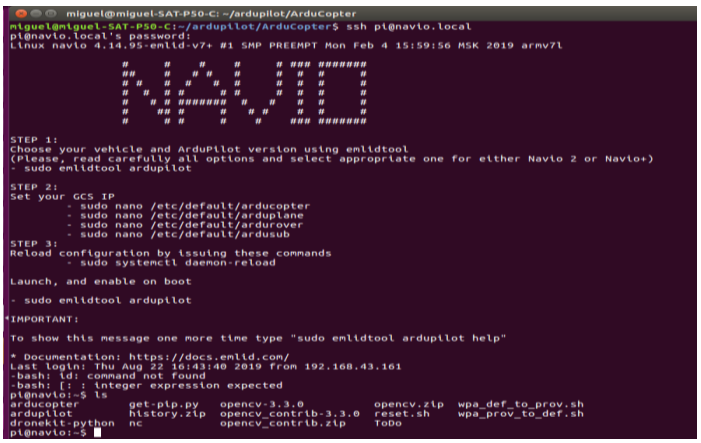


***Figura 39**: Conexión a la Raspberry por ssh*

Una conexión exitosa significa que el propio dron tiene acceso a nuestra red. En cuanto se establezca dicha conexión, recomendamos en primer lugar modificar la contraseña de acceso al sistema, pues de otro modo, cualquiera que conozca la contraseña que el sistema tiene asignada por defecto podría apropiarse del control del dron. Para ello, basta con ejecutar la orden:

sudo passwd

Y, una vez lanzada, el sistema nos solicita la contraseña actual para autenticarnos y, así, evitar que cualquier otra persona la modifique en nuestro lugar. Una vez insertada la contraseña y corroborada por el sistema, nos solicita la nueva contraseña la cual debemos repetir para verificar que no nos hemos equivocado al teclearla. Si la operación se ha tramitado exitosamente, el sistema nos lo hace saber.

Estando ya modificada la contraseña podemos empezar a trabajar sobre el dron. Para ello, primeramente, nos cercioramos de que el sistema tiene la versión Python que necesitamos, así como pip, por lo que requerimos acceso a la red y, para ello, deberemos sustituir el fichero *wpa_supplicant.conf* definitivo por el provisional del siguiente modo:

sudo mv /etc/wpa_supplicant/wpa_supplicant.conf /etc/wpa_supplicant/wpa_def

sudo mv /etc/wpa_supplicant/wpa_prov /etc/wpa_supplicant/wpa_supplicant.conf

Y, para que surta efecto, reiniciamos el sistema:

sudo reboot

Cuando ha cesado el sonido de los variadores, volvemos a conectarnos a la Raspberry y empezamos la configuración. Para la instalación de Python, previamente aconsejamos actualizar el sistema para que descargue los paquetes más recientes alojados en sus servidores. Para ello ejecutamos:

sudo apt-get update & sudo apt-get dist-upgrade

Ahora ya, procedemos a las instalaciones:

sudo apt-get install vim python2.7 ipython

sudo apt-get install python-pip

Actualizamos pip:

sudo pip install --upgrade pip

Y continuamos con las instalaciones, a las que añadiremos —aunque puede estar ya instalada por defecto— *emlidtool*, la herramienta que nos ayudará a configurar el tipo de vehículo que vamos a usar:

sudo pip install emlidtool

A continuación, instalamos las otras librerías que pueden ser requeridas en cualquier momento:

sudo pip install pymavlink dronekit dronekit-sitl opencv-python

Una vez finalizada la instalación, podemos volver a restaurar la conexión a nuestra red privada creada siendo el ordenador el punto de acceso a la misma quedando así, incomunicados con la red. Para ello, invertimos los ficheros *wpa_supplicant.conf*:

sudo mv /etc/wpa_supplicant/wpa_supplicant.conf /etc/wpa_supplicant/wpa_prov

sudo mv /etc/wpa_supplicant/wpa_def /etc/wpa_supplicant/wpa_supplicant.conf

Y, para que surta efecto, volvemos a reiniciar el sistema:

sudo reboot

Hecho esto, procedemos a configurar el vehículo abriendo la herramienta a través del comando:

sudo emlidtool ardupilot

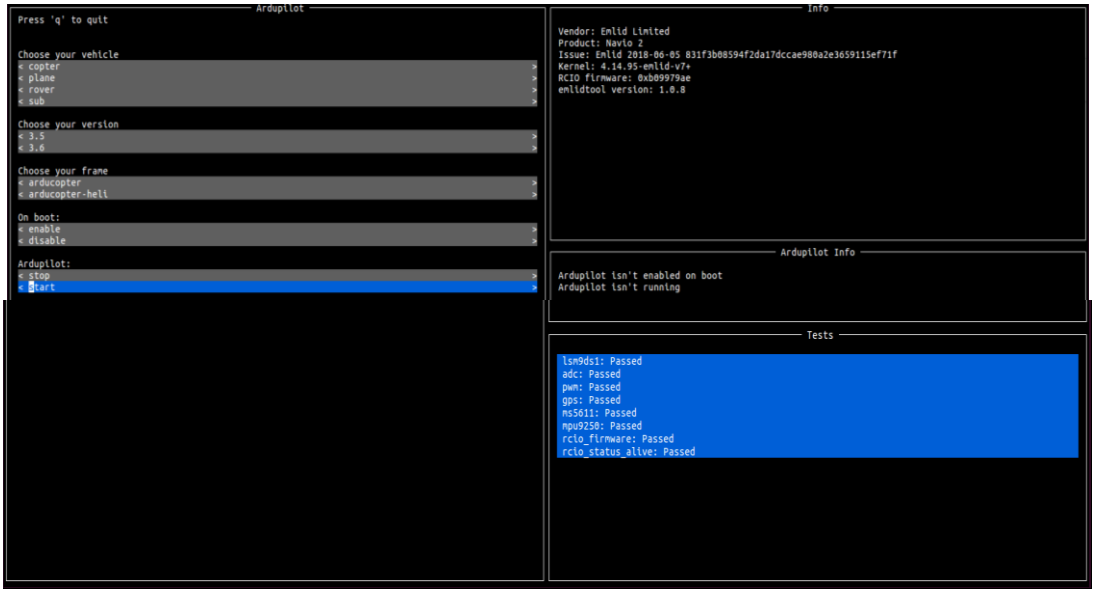


Figura 40: emlidtool ardupilot

La herramienta nos guía a través de un senillo proceso en el que, en primera instancia nos solicita que seleccionemos nuestro vehículo —en nuestro caso, *copter*, ya que se trata de un hexacóptero—, nos solicita también la versión que deseamos ejecutar (en nuestro caso seleccionamos la 3.6.5 la cual en aquel momento era la segunda más reciente para evitar problemas no resueltos), nos solicita a continuación el *frame* que deseamos utilizar, en nuestro caso arducopter, y, por último, nos pregunta si queremos habilitar el demonio de arranque (*enable/disable*) y si queremos dejar Ardupilot en marcha o parado (*start/stop*). Una vez completado el proceso, aplicamos los cambios, el sistema ejecuta automáticamente unos tests los cuales, en condiciones normales, deben superarse y, al finalizarlos, nos invita a salir de la herramienta.

Finalizada así la configuración del vehículo respecto a la herramienta emlidtool, podemos pasar a la configuración de las conexiones con el dispositivo. Para llevar a cabo este propósito es necesario acceder y editar el fichero */etc/default/arducopter utilizando el comando:*

sudo vim /etc/default/arducopter

En este fichero configuramos las conexiones con las que podíamos conectarnos al dron, tal y como se cita a continuación:

TELEM1=”-A udp:10.0.42.1:14550”

TELEM2=”-C /dev/ttyAMA0”

TELEM3=”-C /dev/ttyUSB0”

ARDUPILOT_OPTS=”$TELEM1 $TELEM2 $TELEM3”

En este fichero podemos observar como las distintas conexiones al dron están desglosadas de manera que cada línea es una conexión diferente. Siendo *TELEM1* la que —gracias al parámetro “-A”— indica que debe crear un switch por consola típicamente a través de un enlace Wi-Fi, en nuestro caso usando el protocolo UDP. Le siguen TELEM2 y TELEM3 los cuales indican —gracias al parámetro “-C”— que la conexión se establece a partir del puerto AMA0 de la tarjeta Emlid Navio2 o a través de una antena ubicada en un puerto USB (concretamente el primero que haya sido conectado o, en caso de haber varios conectados al arrancar, el de menor índice de entre los USB).[2] Finalmente, la última línea de nuestro código genera una variable de entorno que engloba las tres telemetrías para que nos podamos conectar a cualquiera de ellas indistintamente.

[2] Es necesario también mencionar la presencia de otros dos parámetros: “-E” y “-B” los cuales indican igualmente que la conexión no requiere GPS.

6. Implementación y simulación

A continuación, procedemos a explicar el funcionamiento de los distintos comandos que hacen rectificar la actitud de nuestro vehículo, así como algunos algoritmos como el de la realimentación visual

6.1. Guiado del dron mediante comandos

De entre los comandos a explicar, el primero de ellos sería el *simple_goto()*, dado que es el más sencillo. En él, se introducen las coordenadas GPS a las que se desea desplazar el dron además de un tercer elemento el cual se corresponde con la altura relativa final. Esta llamada hace cálculos internamente para desplazarse hasta esta posición.

Estos cálculos son mayormente cuaterniones los cuales afectan a las variables de actitud del dron dado que son ellas las que obligan al vehículo a moverse. Para llevar a cabo el movimiento se llama internamente —aunque en los próximos pasos nosotros la usaremos—a la función *set_attitude()* a la cual se le introducen la corrección de *pitch, roll, thrust* y *yaw* para aplicársela directamente al vehículo siendo estas órdenes las que Emlid Navio2 desempaqueta convirtiendo en señales para los variadores.

De este modo, usando la función *set_attitude(roll_angle, pitch_angle, yaw_angle, yaw_rate,use_yaw_rate, thrust, duration)* encontramos los parámetros de manera que primeramente nos solicita el *roll*, posteriormente el *pitch*, desoués el *yaw*, luego nos solicita la tasa de giro y si queremos usarla para terminar con *tthrust* que se expresa con un valor entre 0 y 1 (siendo 0 el mínimo, 0.5 mantener la altura, 1 el máximo y los valores intermedios para ascender, descender o para incrementar o decrementar, según el valor, la potencia de la acción tomada) junto con la duración de esta acción, aunque nosotros este último valor lo hemos omitido de la función ya que queremos acciones en tiempo real.

Para el cálculo de los cuaterniones se introducen los errores a corregir en grados para que la función *to_quaternion()* los convierta en radianes y les aplique los cálculos pertinentes.

Siendo así, procedemos a describir los códigos utilizados para familiarizarnos con los comandos y con la simulación, los cuales no se han incluido en esta memoria debido a su gran semejanza con sus sucesores, los cuales también se han simulado y posteriormente experimentado.

Estos códigos se han implementado con la filosofía de prevenir desastres de manera física, por lo que el primero que programamos fue yaw.py, ya que como mucho giraría sobre sí mismo mientras que los otros podrían alejarse e impactar contra cualquier objeto o ser vivo.

Por este motivo, *yaw.py* va a ser el primero en ser descrito. Este código, al igual que los otros dos, puede diseñarse utilizando realimentación visual o no. En caso de implementarlo a ciegas, tan solo era necesario generar una lista de puntos a los que quisiéramos que el dron apuntase. Para cada punto, se calcula el ángulo respecto

del dron y, con la función *set_attitude* se le indica dicho ángulo en grados para que el sistema calcule el cuaternión pertinente y lo ejecute.

Los siguientes códigos a analizar son *pitch.py* y *roll.py* de modo que con la misma batería de puntos, se calculaba la distancia lineal en su respectivo eje respecto al punto y, a este mismo valor, se le aplicaba un controlador proporcional para que, en caso del *pitch*, en caso de necesitar avanzar, el valor debe ser negativo y al revés en caso de desear retroceder.

Para poder evitar el caso de que el valor sea excesivamente grande como para poner en riesgo el posible giro completo del dron y, por tanto, su caída, se ha dispuesto de un par de limitadores (uno para cada dirección de movimiento) para el que, en caso de superar un umbral, se asigne como máximo este umbral respetándose el signo de modo que nunca se rebase tanto en sentido positivo como negativo.

6.2. Guiado del dron mediante visión

Para los códigos de realimentación visual, y siguiendo en la línea de los explicados con anterioridad, encontramos los mismos códigos, pero esta vez con realimentación visual. Para ello, se aplica un algoritmo de visión con el que se filtra la imagen por HSV y se obtiene el punto del momento del centro del objeto con mayor contorno que se haya filtrado. Es por ello por lo que, para no redundar, pasamos directamente a la explicación de los códigos que pretendemos utilizar en la experimentación real

Llegados a este punto, solo queda por explicar los códigos implementados para dotar de autonomía a nuestro vehículo. Inicialmente, debido a que era necesario simular cada movimiento, disponíamos de 4 códigos, pero, como vamos a poder apreciar a continuación, se ha simplificado a dos —*rotation.py* y *persecution.py*— ya que la genética es muy similar hasta el punto en que solo modificamos pequeñas órdenes manteniendo así la mayoría de la estructura, por lo que vamos a analizar ambos códigos paralelamente deteniéndonos a equiparar las posibles diferencias en caso de haberlas.

A fin de esclarecer mejor el funcionamiento del código, igual que con *follow_me.py* hemos decidido incluir un diagrama de flujo que aclare mejor todo este funcionamiento. Así pues, procedemos con el diagrama fragmentado para posteriormente continuar con su pertinente explicación más detallada.

Todos estos códigos, basados originalmente en el código *set_attitude_target.py* de *DroneKit*, inician su código con la importación de las librerías necesarias tanto para el funcionamiento original de *set_attitude_target.py* como los necesarios para el correcto funcionamiento de la cámara. Así pues, procedemos primeramente con un diagrama más genérico que englobe a grandes rasgos todo el proceso y, tras él, uno más específico que hemos segmentado para poder dar sus pertinentes explicaciones:

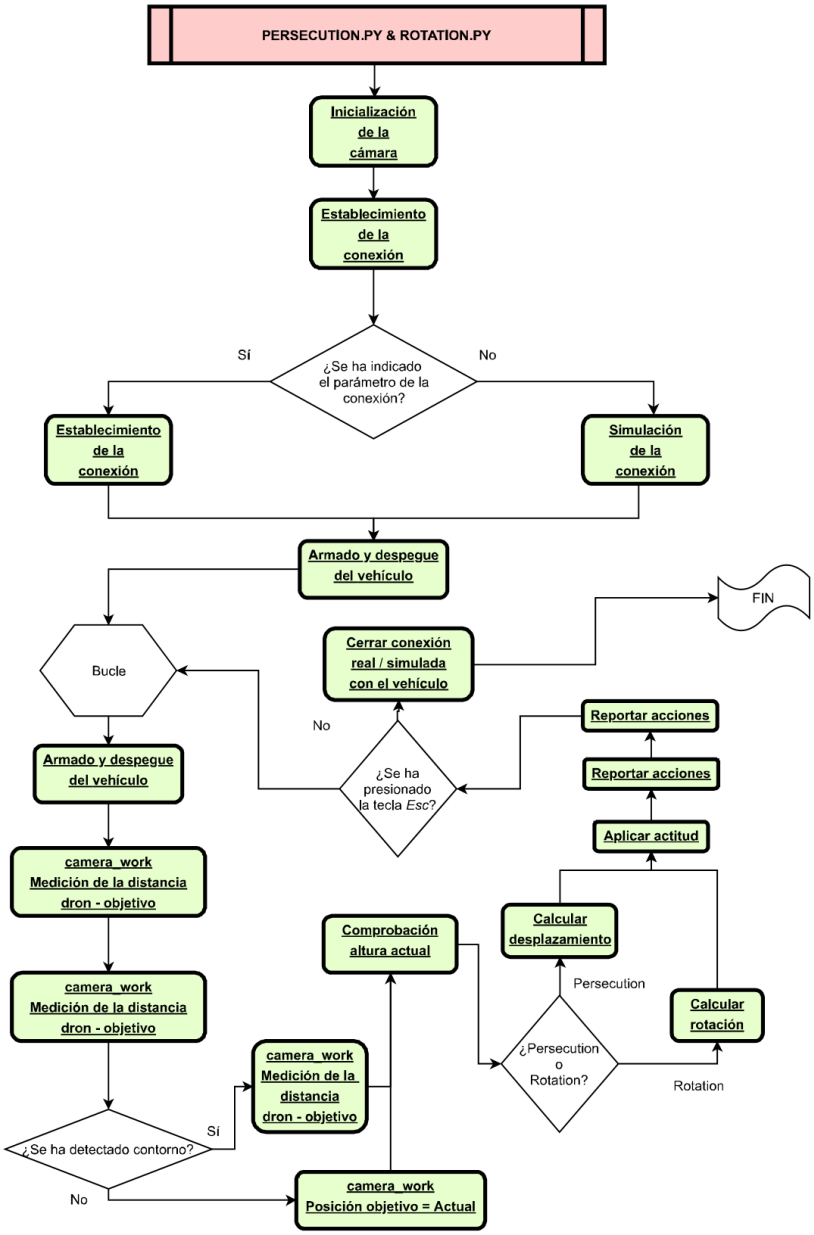


Figura 41: Diagrama de flujo global de persecution.py y rotation.py

Con la Figura 41 podemos hacernos una idea de qué hemos pretendido desarrollar, no obstante, creemos necesario profundizar un poco más para que se entienda el código realizado que es visible en los anexos, así como los fragmentos de diagramas de flujo que vamos a encontrar a continuación.

En ellos podremos encontrar un conjunto de bloques como los pertenecientes a “camera_work” los cuales hacen referencia a acciones más desglosadas que han sido llevadas a cabo dentro de esta función aun habiéndola resumido en este diagrama global en tres bloques.

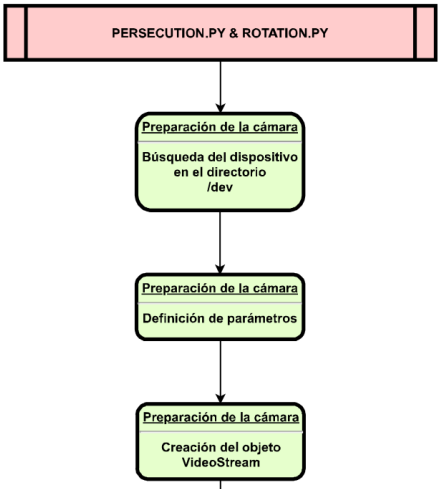
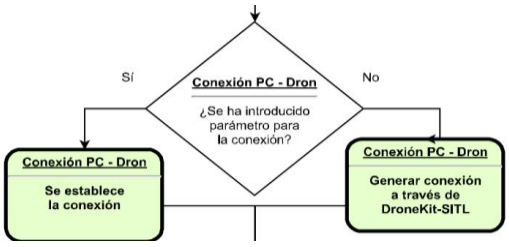


Figura 42: Diagrama de flujo persecution.py y rotation.py (parte 1)

Contando ya con las librerías, el sistema, al igual que en los códigos anteriores, trata de obtener con el *ArgumentParser* los datos para la conexión para, en caso de no obtenerlos, generar una conexión a través de SITL. A continuación, definimos las variables correspondientes a la resolución de la cámara junto con las variables *x_dest*, *y_dest* y *z_dest*, las cuales forman en su conjunto *pos_offset* cuya utilidad se basa en almacenar en los posibles parámetros iniciales que quisiéramos darle al dron respecto a la posición que tuviera que alcanzar. Estos códigos apenas utilizan esta variable de modo que podríamos haber prescindido de ella, pero la hemos dejado para dar mejor significado en el momento de manipular los valores que contienen.

Tras este punto, se genera la resolución a partir de los parámetros definidos con anterioridad junto con la cantidad de fotogramas por segundo y la variable *printedShape* cuya utilidad se va a definir próximamente. Antes de llegar a este punto, es necesario explicar el bucle encontrado en ambos códigos ya que, en él, se listan todas las entradas de vídeo almacenadas en el sistema y, para cada una de ellas, se listan todos sus parámetros para buscar entre ellos tanto su *idVendor* como su *idProduct* para almacenarlos en variables y compararlos con los de nuestra entrada de vídeo. En caso de encontrarlo, modificamos el valor de la variable *dev* y, en caso de terminar el bucle sin haber sido encontrada, abortamos la ejecución. Esta búsqueda la realizamos debido a que, tras cada conexión de nuestra digitalizadora, ésta tomaba un valor diferente, por lo que era imprescindible realizar la búsqueda en directo.

Una vez obtenido el valor del dispositivo, creamos e inicializamos el *stream* de vídeo dándole un segundo para que arranque. Tras esto, definimos para el filtro HSV los valores máximos y mínimos, creamos la ventana de visualización de imagen y las *trackbars* necesarios para cada uno de los valores definidos. En caso de que *printedShape* sea negativo, se crea el frame y aplicamos el filtro HSV a la imagen.



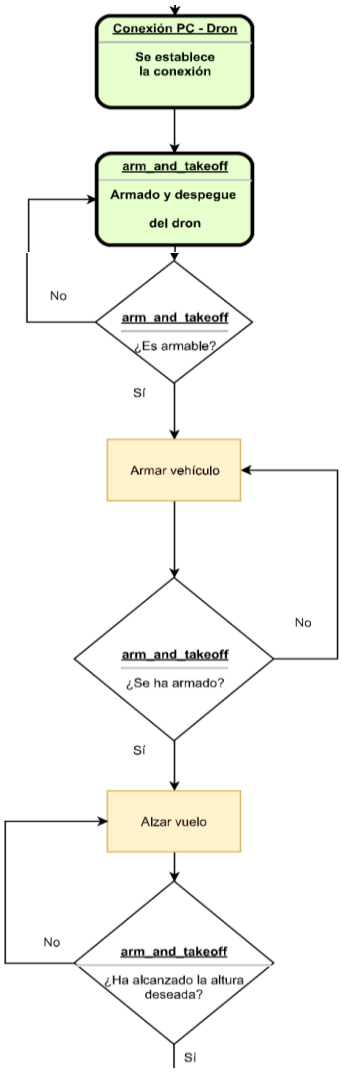


Figura 43: Diagrama de flujo persecution.py y rotation.py (parte 2)

A continuación, se da paso a las funciones que desglosan el funcionamiento del código para una mejor comprensión. La primera de ellas es *arm_and_takeoff*, la cual define un par de constantes que representan dos niveles de potencia distintos para las hélices. Tras estas definiciones, se comprueba que el vehículo es armable en un bucle bloqueante para una vez serlo, poder armarlo, asignarle el modo *GUIDED* para que tenga en cuenta la ubicación GPS —manteniendo así la posición estática en el aire cuando no requiera correcciones de actitud— y empezar así el despegue.

Es en este preciso instante cuándo se asigna la primera de las constantes al *thrust* consiguiendo así que las hélices empiecen a rotar para entrar en un bucle en el que vamos almacenando la altura relativa actual y la compramos con la deseada aplicando un factor de histéresis de 0, 05. En caso de encontrarnos a una altura dentro del rango permitido entre la histéresis y la altura deseada, salimos del bucle para dejar de ascender, pero si aún no hemos alcanzado este margen y la altura actual supera el 60% de la altura deseada, sustituimos el *thrust* actual por la segunda de las constantes declaradas al inicio de la función suavizando así el ascenso. En cualquiera de los casos utilizamos la función *set_attitude* para asignarle al dron la actitud pertinente.

Continuando con las funciones, y antes de continuar con el diagrama, le sigue la función *send_attitude_target*, la cual codifica un mensaje para enviarlo a través de

MAVLink a nuestro vehículo. Para ello tiene definidos unos valores por defecto a los que se les puede alterar el valor citándolos al utilizar la función. De este modo, la función empieza asignándole el valor *yaw* en caso de no haber sido definido al llamar a la función asignándole el valor propio del vehículo en ese momento. A continuación, crea un mensaje al que le asigna los valores correspondientes al mensaje, a excepción de los valores de *roll*, *pitch* y *yaw*, los cuales obtiene a partir de la función *to_quaternion*. Una vez codificado el mensaje, lo envía, tal y como hemos dicho anteriormente.

La siguiente función —*set_attitude*— llama a la función anterior con los valores introducidos en ella. Esta función originalmente tenía un contador temporal para poder utilizar una misma actitud durante un determinado tiempo el cual venía definido por la variable *duration* y es por este motivo por el que existe esta función ya que podíamos haber utilizado directamente la otra, pero ello repercutía en cambios del código qué podían comportar fallos a la hora de ejecutar el programa si no encontraba esta función.

Pasando pues a la siguiente función encontramos *to_quaternion* la cual a través de unas operaciones matemáticas convierte los 3 parámetros inicialmente indicados en el cuaternión resultante.

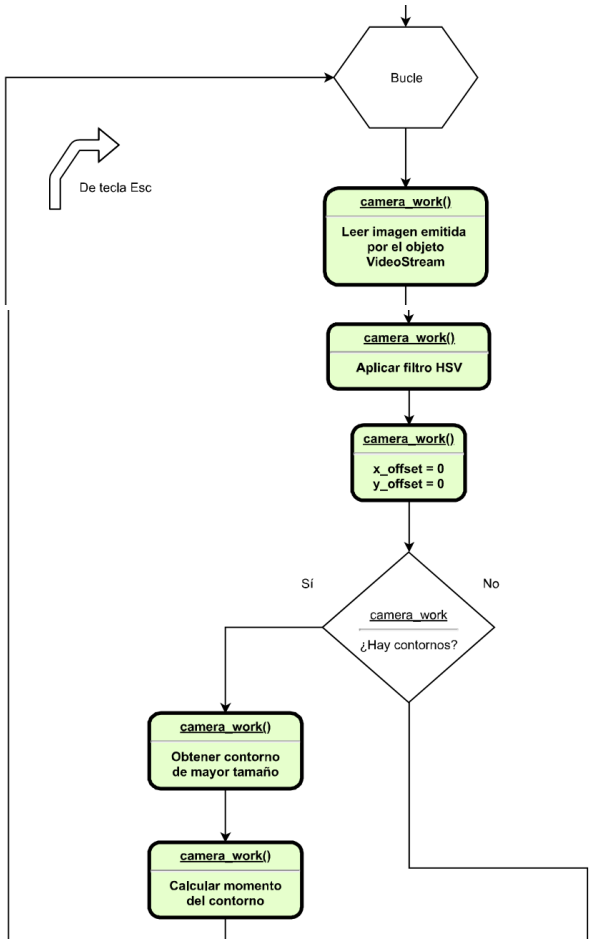


Figura 44: Diagrama de flujo persecution.py y rotation.py (parte 3)

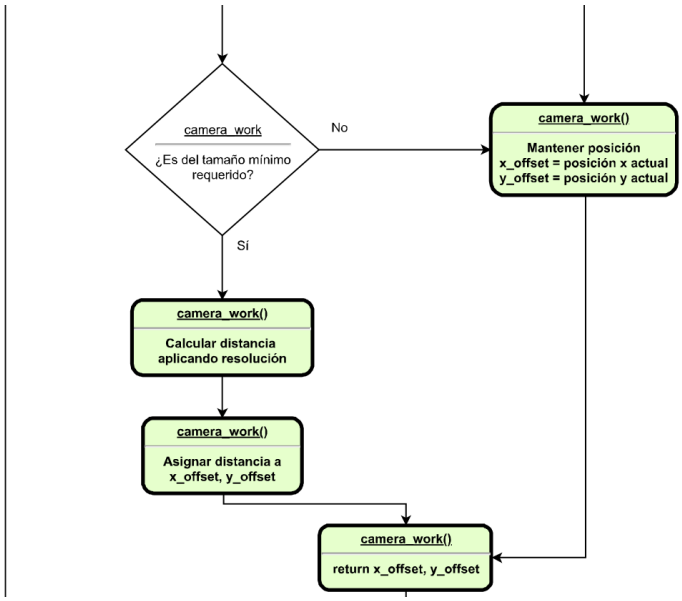


Figura 45: Diagrama de flujo persecution.py y rotation.py (parte 4)

Alcanzando ahora este punto encontramos la función *camera_work* la cual se encarga de obtener la posición del objetivo respecto al dron utilizando la cámara. Para ello, lee la imagen del stream de vídeo para aplicarle el filtro y obtener así los contornos. Si ha encontrado alguno, obtiene el centro del contorno de mayor tamaño para almacenar en sus respectivas variables las coordenadas X e Y respecto a la cámara, así como también el radio de dicho contorno. Con estos valores calculamos el momento ver contorno y si el momento es distinto de 0 y el radio es mayor que 10 calculamos las coordenadas del centro del objeto en función de los momentos de cada eje.

A partir de dichas coordenadas construimos el punto centro y lo dibujamos en el *frame* construido al inicio. Tras esto actualizamos la posición destino utilizando las coordenadas del centro del objeto apoyándonos en la resolución de la cámara de modo que el eje X es positivo hacia la derecha y negativo hacia la izquierda, mientras que el eje Y es negativo hacia adelante y positivo hacia atrás.

Esto es debido a que las pantallas numeran los píxeles en orden ascendente desde la esquina superior izquierda mientras que nosotros consideramos el punto (0, 0) el centro de la pantalla y, por tanto, un punto en el eje X a la derecha del centro de la pantalla es positivo, así como un punto a su izquierda es negativo, pero para el eje Y, un valor por debajo del centro de la pantalla tendrá un valor mayor que el centro de la pantalla, pero debería ser negativo, sucediendo lo mismo con la mitad superior de la pantalla. Una vez actualizado este valor se asigna el mismo valor a otras variables para ser éstas las que devolvamos al final de la función.

La siguiente función es *check_height* la cual se dedica a comprobar que la altura del vehículo sea la adecuada, es decir, que no se haya alterado debido a alguno de los recientes movimientos y, para ello, calculamos el cociente proporcional entre la altura actual y la almacenada inicialmente en *pos_offset* de modo que si la altura excede una histéresis de 0.05 respecto a la deseada, requiere un ajuste por el que calculamos una cantidad a añadir —si está por debajo— o sustraer —si está por encima— en función del cociente proporcional obtenido de modo que la modificación

será mayor si permanece por encima del doble o por debajo del 70% de la altura deseada y, en caso de permanecer dentro del rango, dejamos el *thrust* a 0.5 para que no varié. Finalmente, devolvemos el valor obtenido al terminar la función.

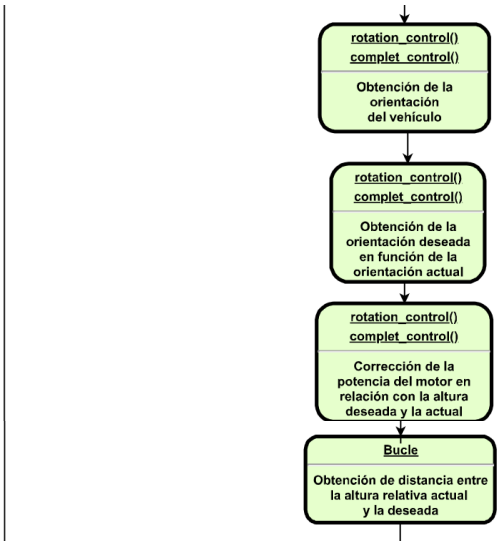
Continuamos con la última función de cada código, la que les hace comportarse de distinta manera. Así pues, vamos a empezar hablando de la función del código *rotation.py* —*rotation_control*— y seguiremos con la de *persecution.py* —*complet_control*— para terminar retomando ambos códigos simultáneamente explicando como acaban su ejecución de manera muy similar.

En el primero de ellos, *rotation_control* recibe como parámetros de entrada *x_FO, y_FO* y *z_FO* los cuales se refieren a la distancia en cada eje desde el final hasta el origen nombrándolos como los vectores que realmente representan. El primer paso dentro de la función es la obtención del ángulo que debe tomar el dron teniendo en cuenta el ángulo en el que se encuentra en dicho momento respecto al norte geográfico, por lo que sumamos ambos valores. A continuación, comprobamos la altura del vehículo y almacenamos la posible corrección en una variable.

Dado que este código se encarga de forzar que el vehículo observe al objetivo siendo innecesario el seguimiento, no son necesarias más operaciones, por lo que llamamos a la función *set_attitude* con los valores recién calculados para después mostrar por pantalla los errores de distancia y ángulo respecto al objetivo.

Por otra parte, el *complet_control* propio de *persecution.py*, calcula el ángulo de igual modo que acabamos de ver para *rotation.py* pero, además, calcula los errores de *pitch* y *roll* a corregir aplicándoles una constante de proporcionalidad para suavizar el movimiento siempre y cuando no exceda en valor el de un máximo asignado a lo largo del código de modo que, en caso de excederlo, se reajusta al valor máximo manteniendo el signo.

Tras esta acción, se comprueba la altura actual del vehículo, se almacena la posible corrección en una variable y se llama a la función *set_attitude* para corregir a la par todos los parámetros. Tras esta acción, se reportan por pantalla los errores actuales de distancia y los aplicados a cada variable de la actitud.



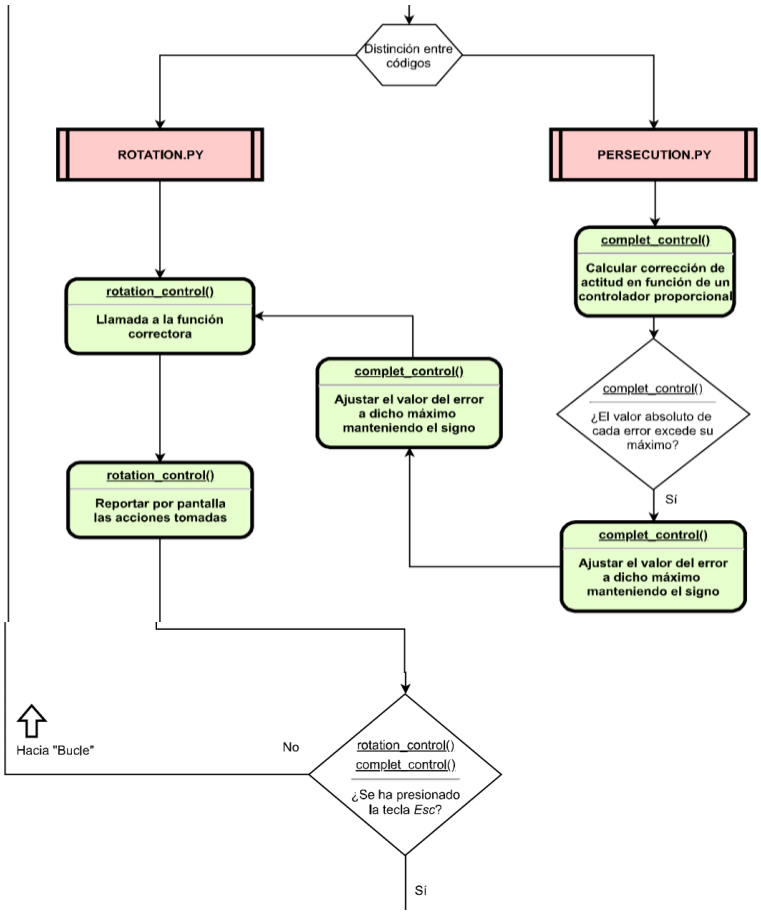


Figura 46: Diagrama de flujo de persecution.py y rotation.py (parte 5)

Llegado este momento, solo queda dar paso al resto de código el cual comparten ambos de igual manera. Esta parte representa tanto el despegue como el código que va a llevarse a cabo de manera indefinida mientras nosotros deseemos. Así pues, podemos observar que se llama a la función *arm_and_takeoff(5)* para pedirle que se sitúe a 5 metros de altura, asignamos los valores de las constantes de proporcionalidad, así como los límites máximos para roll y para pitch en caso de requerirlos.

Tras estas acciones, realizamos un *time.sleep* de 2 segundos para evitar entrar en el bucle sin haber realizado todas las acciones pertinentes, para crear a continuación las variables *x_FO*, *y_FO* y *z_FO* quedando así ya preparados para entrar en el bucle del que solo saldremos si presionamos la tecla *Esc*. En dicho bucle, llamamos a la función *camera_work* para que nos devuelva *x_offset* e *y_offset*, tomamos la altura que nos falta para alcanzar nuestro objetivo (esta variable solo la utilizamos para el reporte final de cada iteración) y llamamos a la función de control correspondiente a cada código —*rotation_control* o *complet_control*— para terminar comprobando si se ha presionado la tecla *Esc.*

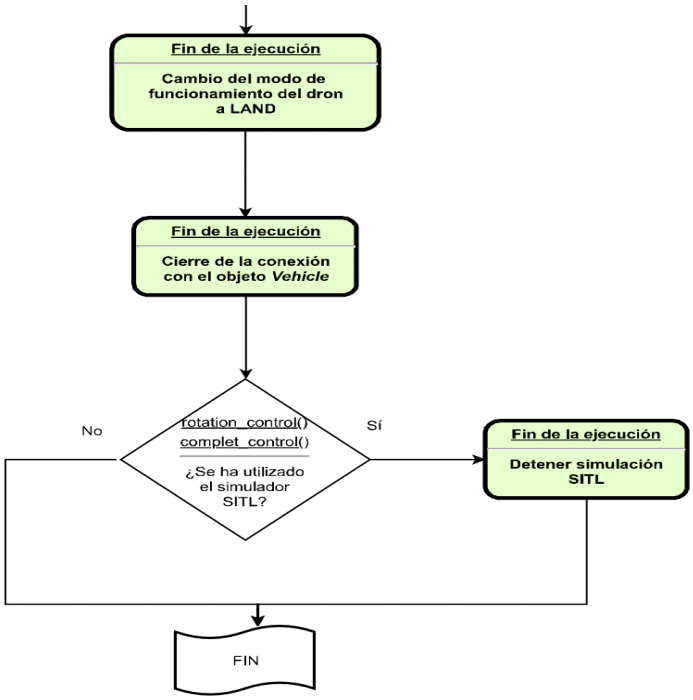


Figura 47: Diagrama de flujo de persecution.py y rotation.py (parte 6)

En caso de salir del bucle, se cambia el modo de funcionamiento a *LAND* para aterrizar donde se encuentre, se deja en espera 1 segundo al proceso para posteriormente cerrar el *frame*, el *stream* de vídeo y la conexión (incluyendo el cierre de la conexión SITL en caso de haberla creado).

6.3. Seguimiento mediante GPS

Para el seguimiento mediante GPS, encontramos el código follow_me.py cuya función principal es recabar la posición más actual posible de nuestra ubicación para tratar de situarse en ella. Para llevar a cabo esta operación, necesita de la función *simple_goto()* a la cual le introduce como parámetros nuestra más reciente ubicación.

Así pues, procedemos a analizar el código de *follow_me.py.*

Con la intención de aclarar y facilitar el entendimiento del proceso que sigue este código, hemos decidido añadir un diagrama de bloques que sea capaz de esquematizarlo segmentado en fragmentos que iremos describiendo.

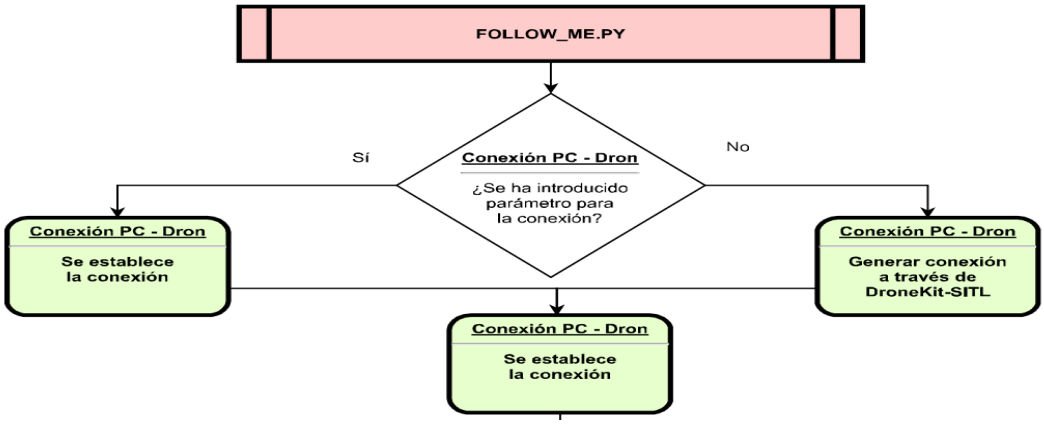


Figura 48: Diagrama de flujo follow_me.py (parte 1)

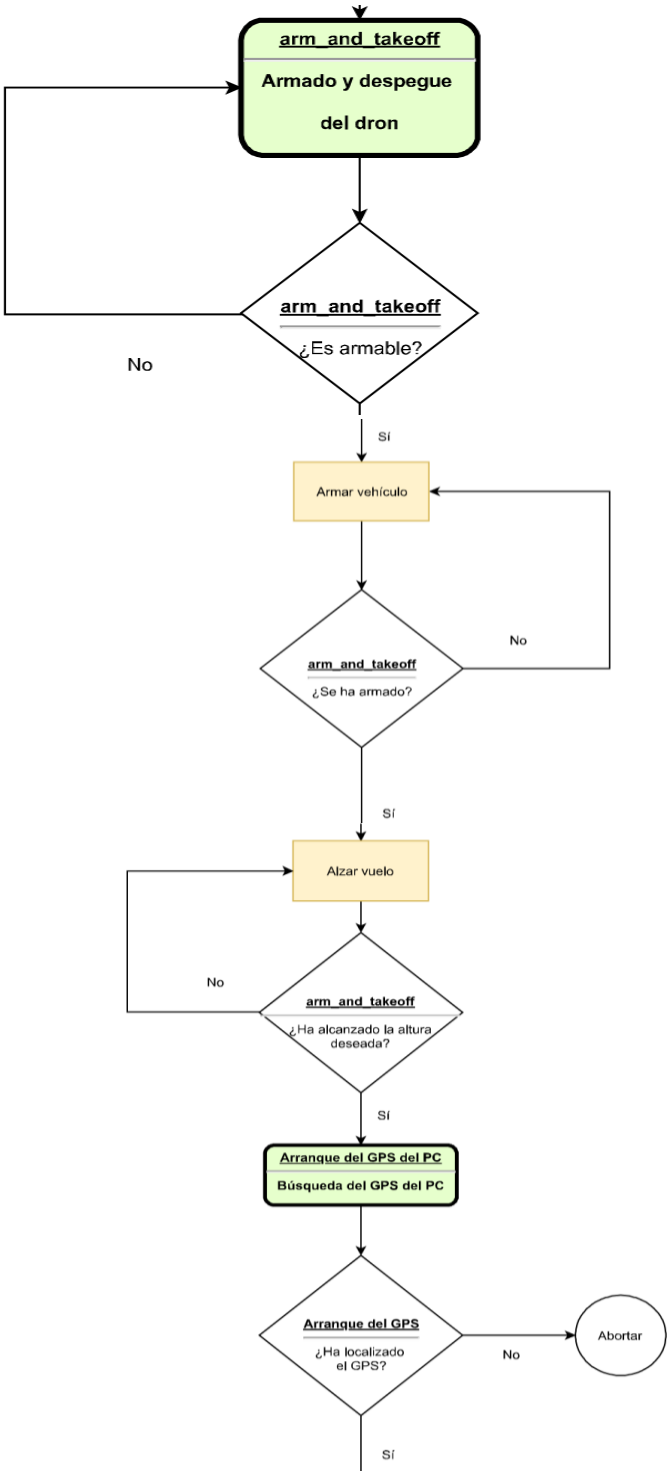


Figura 49: Diagrama de flujo follow_me.py (parte 2)

Para que este código funcione, en primer lugar, debe de buscar el objetivo para conectarse a través del *ArgumentParser* —como en el caso anterior— y tratar de establecer conexión con dicho parámetro con una espera bloqueante y con un *timeout* determinado.

A continuación, se define la función *arm_and_takeoff* en la que, como su propio nombre indica, se implementa el armado y el despegue del dron. Para ello, obviando los mensajes por pantalla, el sistema queda bloqueado esperando a que la variable *is_armable* del vehículo valga *True* para que dé luz verde al resto de la función. Una

vez validado este punto, se asigna el modo de funcionamiento a *GUIDED* para que acate las órdenes que le transmitimos para, seguidamente, modificarle el parámetro *armed* a True para solicitar el armado y quedar esperando a que el parámetro en el vehículo se corresponda con nuestra asignación.

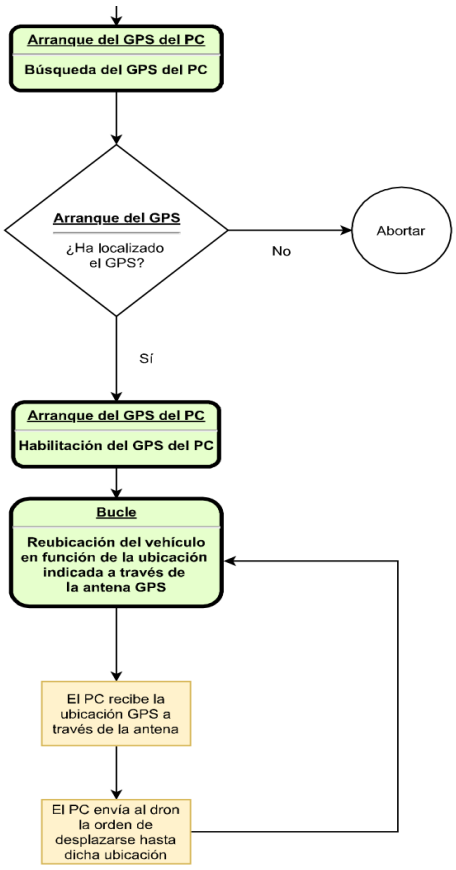


Figura 50: Diagrama de flujo follow_me.py (parte 3)

Una vez finalizado el armado, el sistema despega con una altura fijada de modo que entra en un bucle del que solo sale cuando alcanza una altura mayor o igual a la deseada reduciéndole un 5% de histéresis para que, en caso de alcanzar un valor muy cercano a la altura deseada, detenga el ascenso.

A esta función le sigue el código principal del programa. En éste, dentro de un *try-except* —el cual captura errores de socket y notifica debidamente para evitar que se ejecute sin GPS alguno—, habilita el GPS, ejecuta la función descrita con anterioridad con una altura fijada a 5 metros y, tras el despegue, entra en un bucle infinito del cual no se puede salir mientras el modo de funcionamiento sea *GUIDED*.

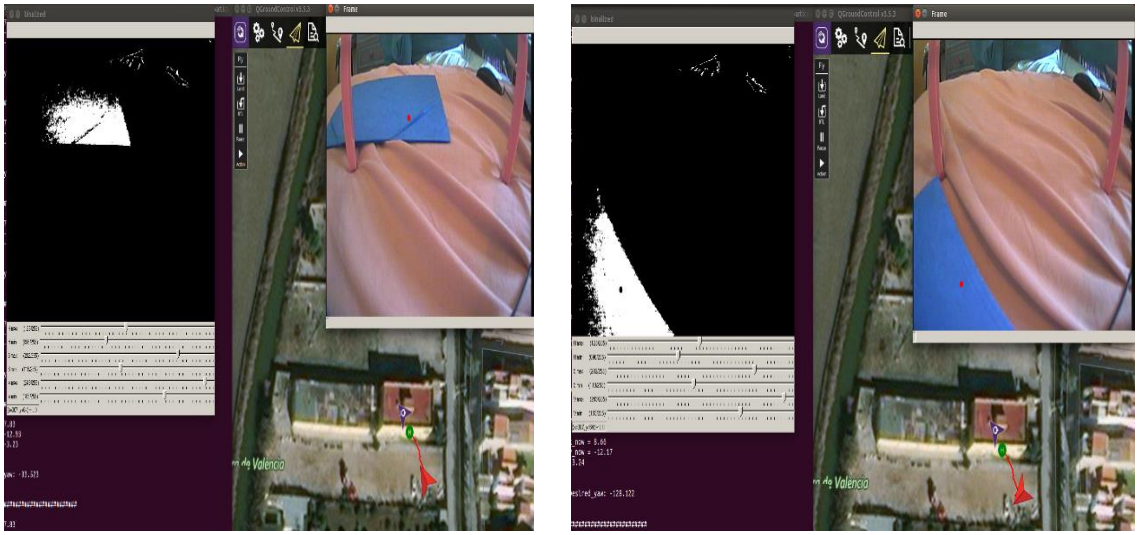
En dicho bucle, se lee el estado del GPS y, si el GPS tiene latitud y longitud fijados a valores válidos, es decir, ambos son distintos de 0, se define una nueva altitud (esta acción se puede omitir para trabajos más próximos al usuario) y se crea una nueva posición basándose en los valores obtenidos del GPS y la altura recién definida. Con esta nueva ubicación se solicita al vehículo que se traslade hasta ella y espere durante 2 segundos para realizar otra iteración de este bucle. Esta operación de espera es la que deberíamos modificar —u omitir— en caso de querer una reacción más temprana por parte del dron o, incluso, en tiempo real.

En caso de salir del bucle debido a que el modo de funcionamiento ya no sea GUIDED, se procede como en el código anterior cerrando primeramente la conexión con el objeto *vehicle* y, posteriormente, el SITL en caso de haberse usado.

6.4. Simulaciones

Habiendo llegado ya a este punto, es necesario esclarecer los funcionamientos de todos los códigos vistos hasta el momento. Siendo así, podemos empezar por el primer bloque. Entre ellos, podemos empezar por *yaw.py*, ya que es el que siempre probamos primero.

6.4.1. Yaw.py



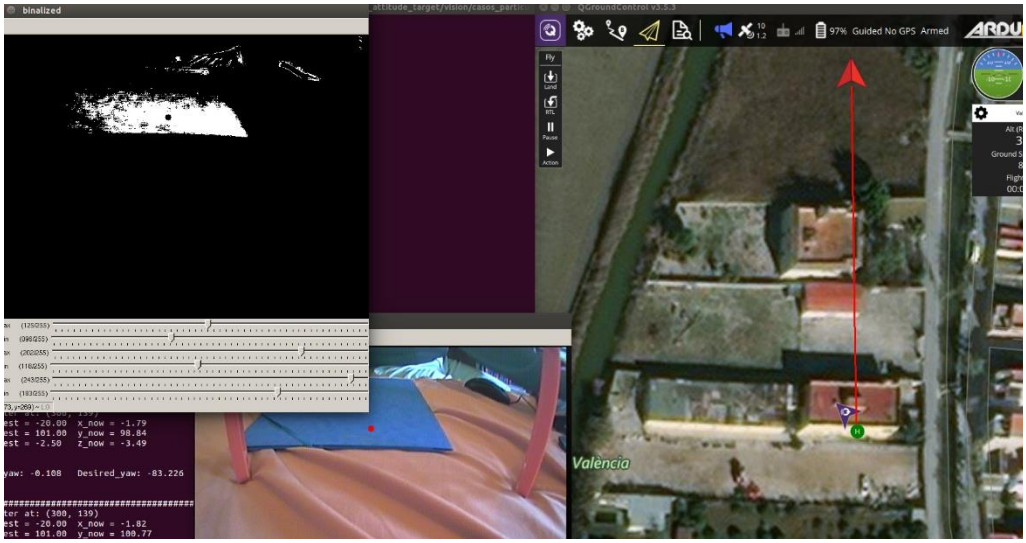
(a) *Carpeta arriba* ***(b)*** *Carpeta abajo*

Figura 51: Detección de la carpeta

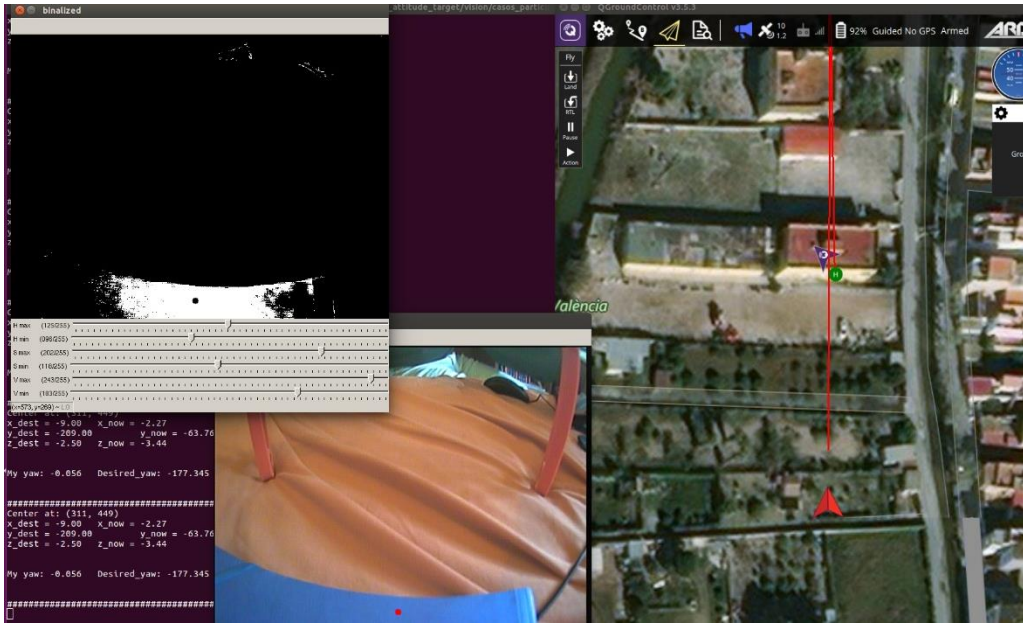
En la Figura 51 podemos observar como la carpeta es detectada gracias al filtro HSV donde vemos la silueta blanca que hace referencia a esa tonalidad azul —los trozos de la carpeta que no capta el filtro son debidos a la distinta tonalidad debido al brillo— la cual el sistema capta y marca con un punto rojo al cual hace girar el vehículo (visible en la estación de tierra como una flecha roja) para apuntarlo con su cara frontal. Apreciamos también una derivación de su ubicación debida a que el modo de funcionamiento del dron simulado es “Guided-NoGPS” lo cual repercute en que el sistema no tiene en cuenta la ubicación por lo que no trata de reubicarse.

6.4.2. Pitch.py

Los siguientes códigos que analizar son *pitch.py* y *roll.py*, siendo *pitch.py quien* genera las siguientes figuras:



(a) Carpeta arriba



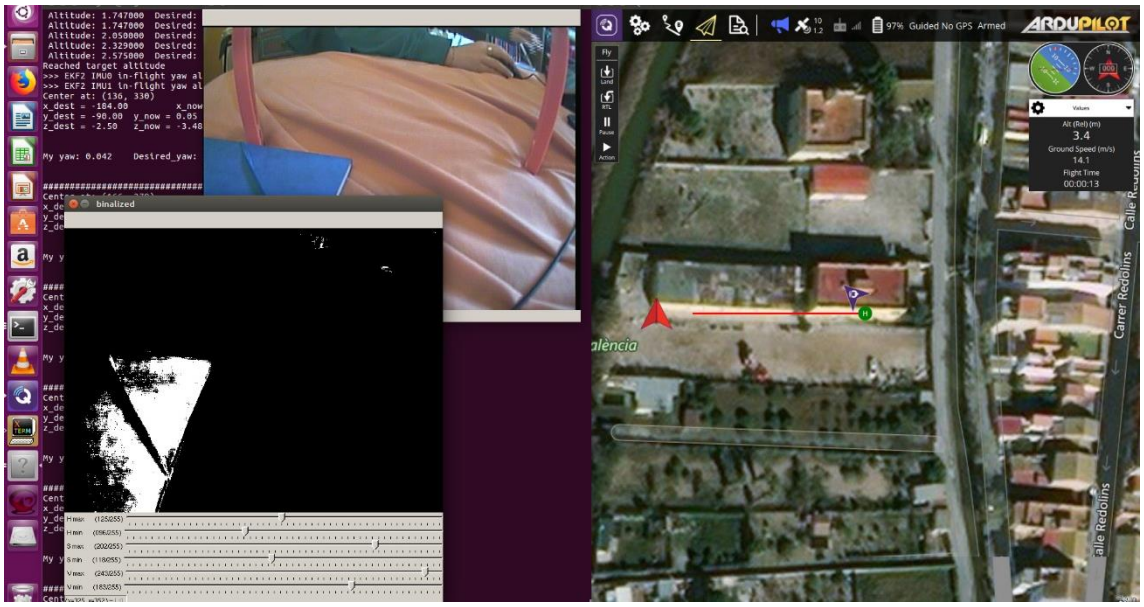
(b) Carpeta abajo

Figura 52: Silumación del código *pitch.py*

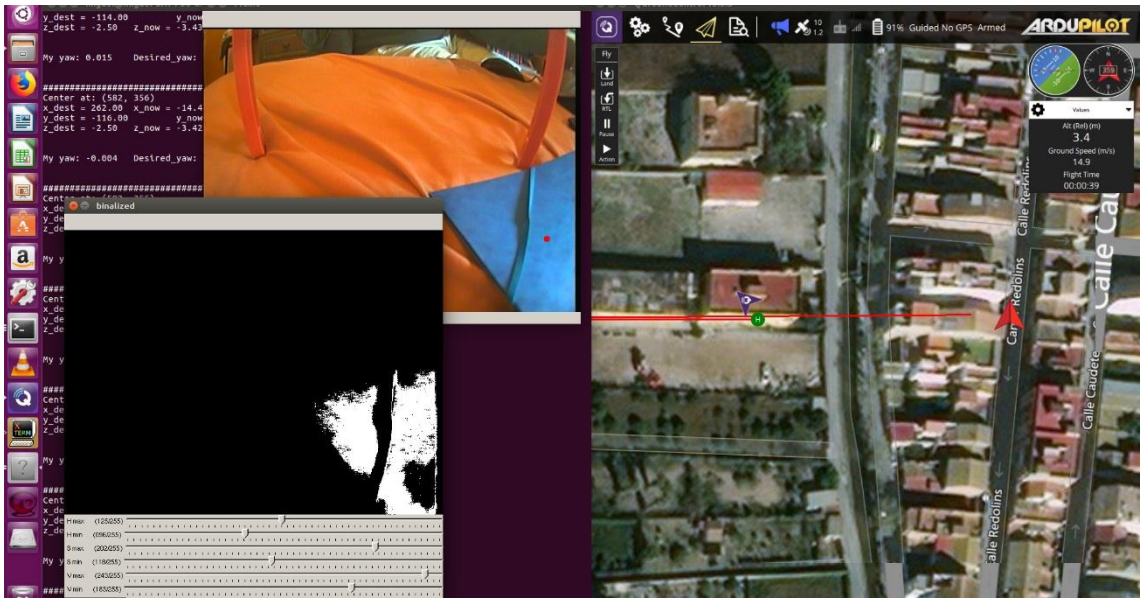
En la Figura 52 podemos apreciar como colocando el objetivo en la mitad superior del campo de visión de la cámara, el vehículo avanza, así como cuando lo colocamos en la mitad inferior y retrocede.

6.4.3. Roll.py

A esta simulación le sigue la pertinente de *roll.py*, cuyo resultado son las figuras siguientes:



(a) Carpeta a la izquierda



(b) *Carpeta a la derecha*

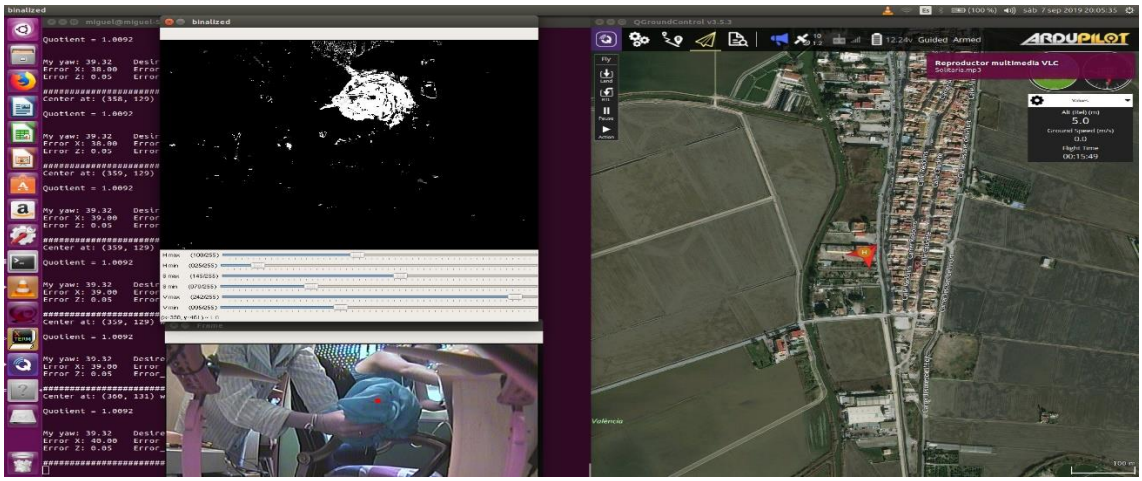
Figura 53: Simulación del código *roll.py*

Gracias a la Figura 53 podemos observar cómo, al igual que en los anteriores, el vehículo obedece nuestras órdenes desplazándose hacia izquierda y derecha cuando corresponde.

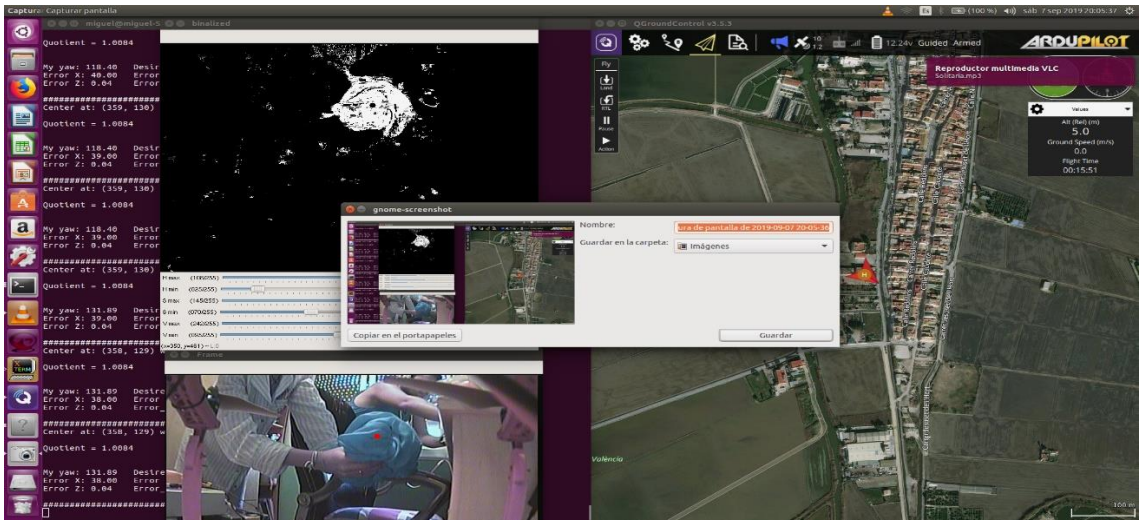
También es necesario analizar el código de *rotation.py* y *persecution.py*, dado que ambos integran visión, pero esta vez, se comportan de una manera un poco diferente. Así pues, procedemos a analizarlo:

6.4.4. Rotation.py

Siguiendo con las simulaciones, ahora disponemos de las simulaciones de los códigos que realmente queríamos probar con nuestro vehículo, aunque el análisis de dichos códigos se realiza más adelante. En primer lugar, vamos a analizar la simulación del código *rotation.py*:



(a) Captura realizada a las20:05:35



(b) Captura realizada a las 20:05:36

Figura 54: Capturas de pantalla del código *rotation.py*

En la Figura 54 podemos ver como el dron, al detectar un objeto —el cual el filtro detecta gracias a la sombra blanca— en la mitad derecha de su pantalla "*binalized*" éste gira en sentido horario, ya que se corresponde a su derecha. De mismo modo sucedería si lo colocásemos a la izquierda y el sentido antihorario, aunque no lo hayamos demostrado en las imágenes, así como que se detiene en caso de situar esta bufanda en el eje vertical de la cámara.

Por otro lado, también podemos ver en el lateral izquierdo de la pantalla los reportes utilizados para conocer cómo se está comportando el sistema de modo que si en algún momento detectábamos comportamientos anómalos bastaba con observar este *log* para comprender los posibles fallos existentes, aunque en este caso ya observamos un resultado exitoso.

6.4.5. Persecution.py

A continuación, vamos a analizar el resultado de *persecution.py* para el que, utilizando la misma bufanda, hemos filtrado su color obteniendo un movimiento basado en la ubicación de éste.

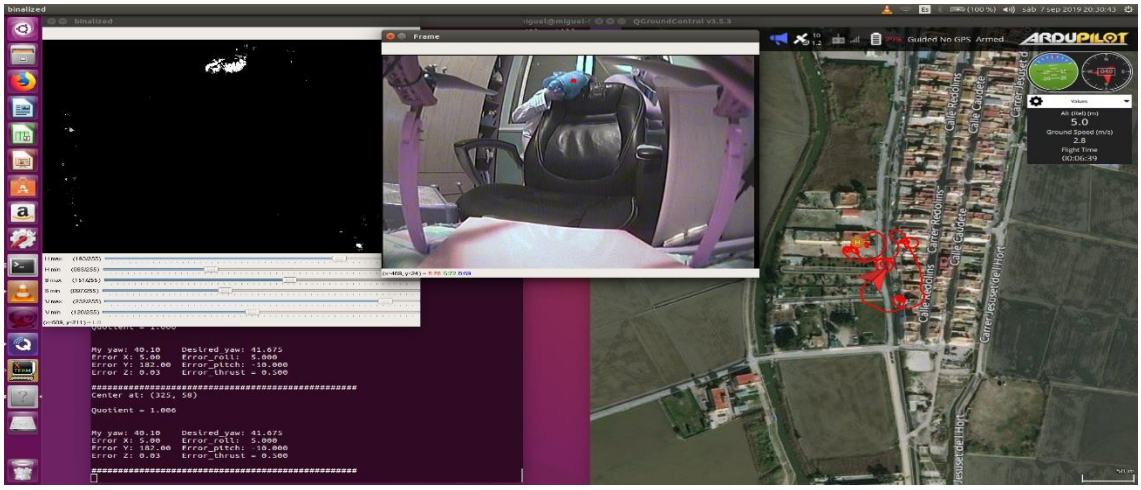


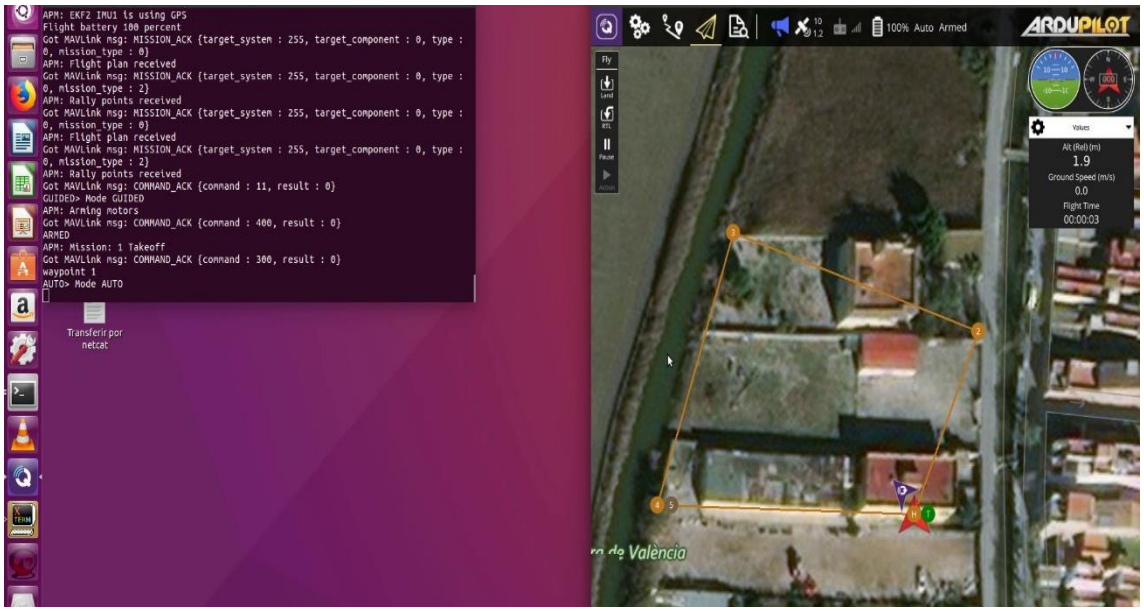
Figura 55: Simulación del código persecution.py

En la Figura 55 podemos observar el camino que ha ido siguiendo el dron durante su vuelo. Debido a que el objetivo se encuentra notablemente cerca en comparación a los que se encuentra normalmente a unos cuantos metros de distancia, los giros son bruscos, no obstante, fijando el objetivo en el eje vertical de la pantalla conseguimos una trayectoria casi perfecta, lo cual significa que tiene un comportamiento aceptable ya que las curvas que impiden la rectitud se deben a no conseguir mantener el objeto justo en el píxel central.

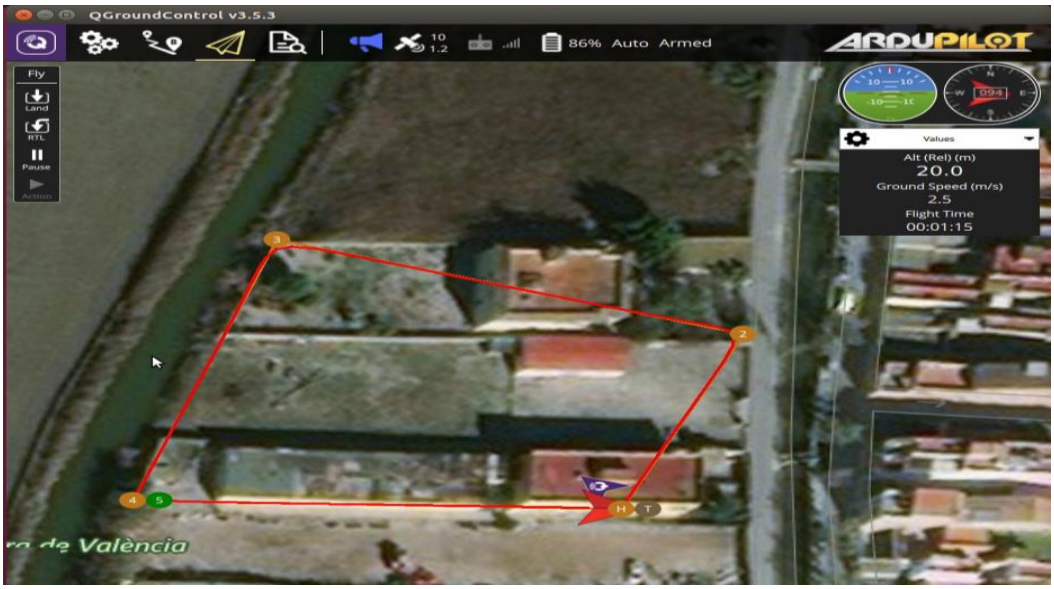
En la Figura 55 podemos apreciar también el *log* en el que conocemos tanto las acciones tomadas como los errores respecto al objeto, así como el punto en coordenadas X e Y para así tratar de acercarlo al píxel central vertical (320, dado que la resolución horizontal es de 640 píxeles).

También nos gustaría remarcar que el vehículo consigue mantenerse en la altura deseada (5 metros), lo cual cobrará sentido con la explicación del código en las líneas siguientes. Así pues, vamos a proceder a ello.

Por último, solo quedaría hablar de la simulación de una ruta a seguir, la cual, a primera instancia puede parecer innecesaria, no obstante, cuando lleguemos al punto de la experimentación real apreciaremos diferencias.



(a) Simulación de asignación de ruta



(b) Simulación de realización de ruta

Figura 56*: Simulación de ruta*

En la Figura 56 podemos apreciar como al inicio asignamos una ruta específica para terminar realizándola por encima de la línea durante todo el tiempo, excepto en la realización de las curvas.

7. Experimentación

Pasando pues a la parte de experimentación, para probar los códigos de vuelo, nos hemos desplazado al campo de vuelo y nos hemos instalado en él de la manera más práctica posible para llevar a cabo estos experimentos sin sufrir inconvenientes. Para ello cogimos una mesa que utilizábamos para manejar el ordenador estando como podemos apreciar en la

Figura 57. Cuando hemos alcanzado este punto, hemos preparado todos los elementos necesarios evitando el consumo de baterías y, en el momento en que todo estaba preparado, los encendíamos.

Figura 57: Inicio del montaje en el campo de vuelo.

En caso de querer realizar una comprobación de que todos los variadores funcionan bien conjuntamente, deberíamos darle alimentación al dron, armarlo y ponerlo en modo "*PositionHold*" de modo que al darle cierta potencia al *thrust* deberían girar todos los motores a la vez, y, en caso de estar en posición totalmente recta, todos a la misma velocidad. Si cogemos en este momento el dron y lo hacemos rotar a través de los ejes de tal manera que pierda la horizontalidad, los variadores deberían responder —este comportamiento es más perceptible al oído que a la vista— dándole más potencia a los motores situados más hacia abajo para intentar compensar el desequilibrio y tratar así de volver a colocarse horizontalmente.

Otra prueba sería la realizada para comprobar que la telemetría consigue comunicarse entre el dron y el computador. Para ello, se ha requerido de dos terminales, uno para conectarnos con el dron a través de ssh y otro para trabajar desde el ordenador. Una vez hecho esto, se han ejecutado estas órdenes:

Ordenador	**Raspberry**
echo -n 'Hola mundo' > /dev/ttyUSB0	***cat -v < /dev/ttyAMA0***

Aquí se nos presentó otra dificultad, ya que no lográbamos enlazar mediante telemetría con el vehículo real mientras que sí podíamos por Wi-Fi. Para tratar de solucionarlo, en primer lugar, tratamos de modificar el punto de conexión de la telemetría del vehículo, es decir, desconectar la antena conectada a UART y

conectarla a través de un cable USB a un puerto destinado a este tipo de comunicación.

Una vez hecho esto, ejecutamos las reglas:

sudo systemctl daemon-reload

sudo reboot

ssh pi@navio.local

sudo systemctl disable arducopter.service

sudo systemctl stop arducopter.service

sudo systemctl enable arducopter.service

sudo systemctl start arducopter.service

Con ellas, reiniciamos *systemd* para que se regenere todo el árbol de dependencias entre *sockets* oyentes de *systemd*, reiniciamos la Raspberry, reconectamos con ella, deshabilitamos su servicio lo paramos —prueba de que surte efecto es que los variadores empiezan a emitir sonidos periódicos de manera indefinida hasta que se reactive el servicio, lo volvimos a poner en marcha y lo habilitamos para que recargara la información releyendo todas las entradas de su fichero de configuración ubicado en */etc/default/arducopter* y buscase el puerto USB en el que estuviera conectada la antena.

Con ello, volvimos a tratar de comunicarnos entre el ordenador y la Raspberry a través de telemetría como hemos explicado con anterioridad con órdenes como *echo* y *cat*, comprobamos si existía conexión y el resultado fue negativo, por lo que sustituimos la antena del vehículo —la cual era una que ya disponíamos— por otra nueva, la volvimos a conectar al puerto UART, repetimos el proceso anterior para resetear el servicio de ArduCopter junto con *systemd*, tratamos de volver a comunicarnos a través de la telemetría y, a partir de este momento, la comunicación fue exitosa.

Con ellas, logramos que la antena telemétrica del ordenador reciba sin saltos de línea adicionales el texto que insertemos entre las comillas y que la antena telemétrica de la Raspberry muestre por pantalla todo lo recibido incluyendo los caracteres no visibles a excepción del salto de línea y la tabulación.

7.1. Pruebas previas

La primera de las pruebas de vuelo la realizamos para comprobar que el dron conseguía elevarse y descender sin problemas. El resultado fue negativo dado que se alzó, pero el aterrizaje fue estrepitoso al hacerlo diagonalmente. Todo fue debido a que el dron se mantenía en el modo mantener altitud (*AltitudeHold*), por lo que no deberíamos haberlo mantenido en este modo, sino que deberíamos haberlo permutado al modo *PositionHold* lo cual habría permitido que el dron no se desplazase durante el descenso y que hubiera aterrizado verticalmente.

Una vez todo estuvo listo, nos decidimos a despegar y, en pleno despegue, el dron se volteó y terminó chocando bruscamente contra el suelo. Al parecer, fue debido a que dos variadores se habían estropeado. Uno de ellos llegaba a funcionar con potencias de *thrust* significativas, pero el otro, ni siquiera elevando el *thrust* conseguía activarse, por lo que decidimos sustituirlos con la impertinencia de que ya no se fabricaban estos variadores, por lo que tuvimos que buscar el nuevo modelo, cuyo consumo era superior. Este detalle, aunque significativo por la duración de la batería, no debía ser un problema, ya que el dron al calibrarse debe ser capaz de compensar cualquier fuerza que lo desestabilice.

Aun así, tuvimos que recurrir a un compañero nuestro. Él sería quien comprobaría bajo sus vastos conocimientos en el manejo de vehículos aéreos el buen ensamblaje del nuestro, consiguiendo de este modo el visto bueno de un profesional para poder continuar nuestras pruebas, así como nos asesoraría sobre los nuevos variadores en función de si era conveniente sustituirlos todos o mantener los originales, siendo ésta última la decisión tomada ya que podía ser un incidente aislado.

No obstante, fue entonces cuando otro variador dejó de funcionar y, llegados a ese punto, tomamos la decisión de que —dado que podían fallar el resto en cualquier momento— era recomendable sustituir los otros cuatro variadores a la vez de tal manera que fue necesario desoldarlos, sustituirlos, soldar los nuevos y recalibrarlos para evitar fallos funcionales.

Figura 58: Prueba de vuelo con los variadores originales.[3]

[3] Será necesario aproximarse con el zoom para poder apreciar que se trata de los variadores originales.

7.2. Modo RPAS

A modo de prueba de que dominamos el pilotaje del dron, tuvimos que tomar el mando y realizar una serie de vuelos una vez seguros de que todo funcionaba correctamente.

Figura 59: Pilotaje manual del vehículo

Es por ello por lo que tuvimos que probamos a alzarlo, rotarlo sobre sí mismo, así como también a desplazarlo a la vez que rotaba de modo que actuara con movimientos naturales de cualquier ser animado. Este ha sido un importante aporte al desarrollo del proyecto ya que ha proporcionado confianza a la hora de desarrollar pruebas en cualquier momento con la tranquilidad de que en cualquier momento podemos tomar el mando y recuperar el control del vehículo.

7.3. Control a través de la estación de tierra

A partir de la estación de tierra es posible establecer rutas de igual manera que hemos hecho con anterioridad. Para ello, en el menú ubicado en la tercera pestaña de la aplicación podemos seleccionar cualquier punto del mapa para establecerlo como punto. En caso de desear que retorne al punto de inicio, al terminar de establecer la ruta, seleccionamos el primero de ellos y activamos la opción "*Return to Launch*" lo cual hará que al llegar al último punto regrese al de despegue.

Figura 60: Fin de la ruta

En la Figura 60 podemos avistar como el vehículo no ha seguido la línea con exactitud. Esto es debido a que se trata de un vehículo real que sufre la fuerza del viento además de calcular su movimiento respecto a su posición actual sin considerar la posible desviación debido al retraso entre el cálculo y la aplicación del movimiento.

7.4. Control desde programas Python

En este punto vamos a analizar en orden cronológico los códigos que hemos ido experimentando a través de la telemetría con el dron real.

7.4.1. Vehicle_state.py

En primer lugar, es necesario hablar de *vehicle_state.py* ya que, a fin de cuentas, es el primer programa testeado y el más seguro en cuanto a la integridad del vehículo se refiere, dado que en ningún momento se realiza ningún movimiento con el dron, simplemente recibimos la información de éste.

De modo que para esclarecer este código hemos diseñado un diagrama de flujo que posiblemente permita al lector hacerse a la idea de su funcionamiento.

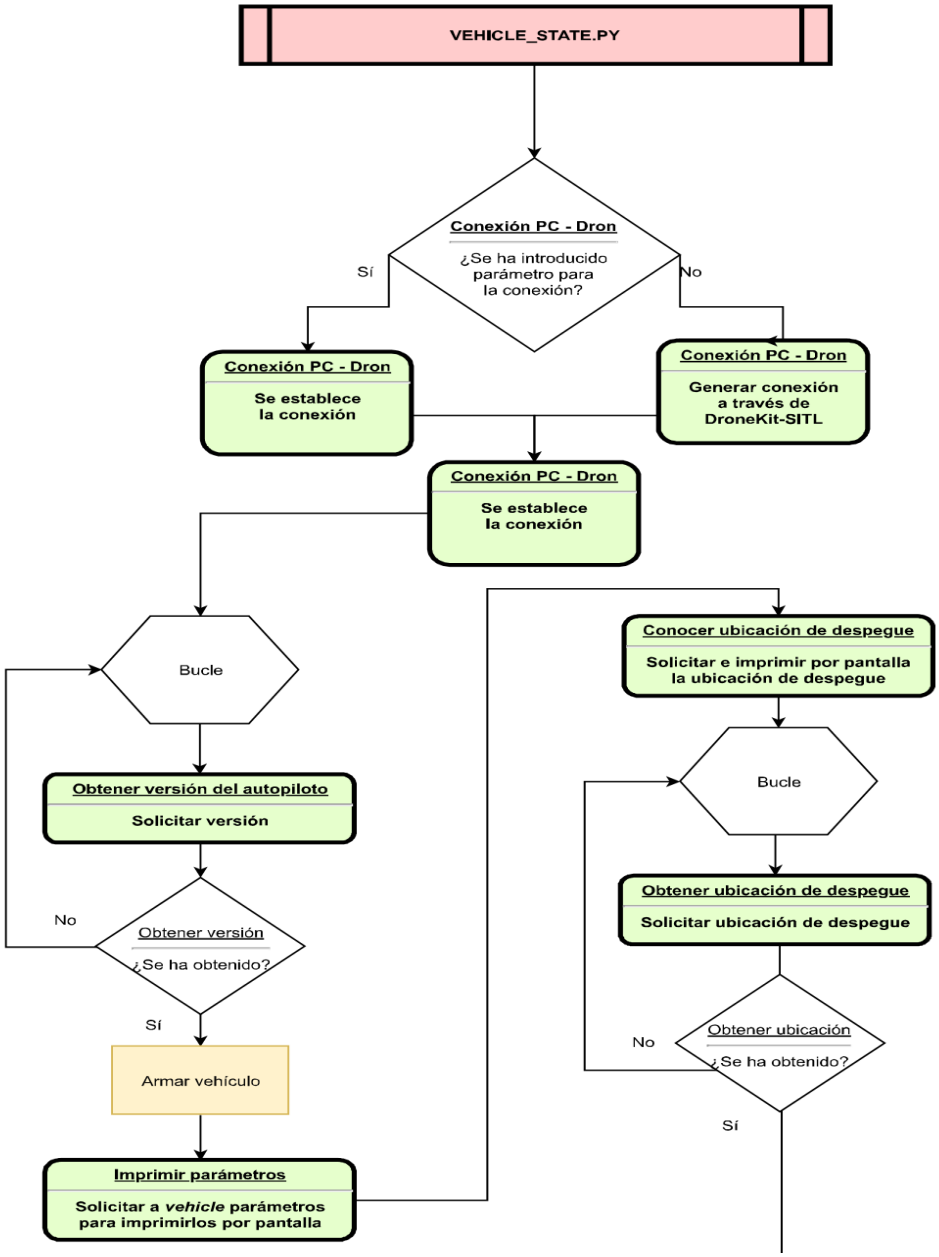


Figura 61: Diagrama de flujo de *vehicle_state.py* (parte 1)

Figura 62: Diagrama de flujo de *vehicle_state.py* (parte 2)

Así pues, este código inicia sus líneas de código buscando como conectar desde el ordenador con el propio vehículo, ya sea simulado o real. Habiendo inducido del *ArgumentParser* propio de Python los datos de la conexión, trata de establecerla y, en caso de no encontrar la cadena que indique la conexión, SITL se encarga de generarla. Tras conseguirlo, espera a conseguir la versión del autopiloto que tiene el vehículo para después solicitar algunos parámetros con la finalidad de mostrarlos por pantalla obteniendo todos estos valores a partir de la variable *vehicle* creada para este fin. De este modo somos capaces de conocer en qué estado se encuentra el vehículo en dicho momento y, entre otras cosas, saber si es armable o no.

Tras la retahíla de parámetros, este código se dispone a conocer la ubicación de despegue de nuestro vehículo la cual no puede estar disponible hasta que el autopiloto la asigne al menos por primera vez desde que hemos encendido el dron. Es por ello por lo que entra en un bucle del cual no puede salir mientras obtenga *None* como resultado de descargar los comandos del vehículo y buscar entre ellos el parámetro *home_location*.

Una vez obtenido, lo muestra por pantalla y trata de modificarlo variando la altitud a un valor fijado para mostrarlo también por pantalla dos veces siendo la primera la variable que se supone que debe haber sido modificada y, la segunda, la comprobación de que se ha modificado en el vehículo descargando de nuevo todos los datos del vehículo y comprobando de entre estos el propio valor asegurándose así de la correcta comunicación bidireccional entre el ordenador y el dron.

Dando por zanjado el asunto de la comunicación y la modificación del *home_location*, el siguiente bloque de código muestra por pantalla el modo de funcionamiento del dron para después modificarlo a *GUIDED* permaneciendo en un bucle hasta conseguirlo para dar paso a la espera de que el vehículo sea armable.

A continuación, se definen una serie de funciones de las que la primera saca por pantalla solamente los parámetros con valores distintos a los anteriores, lo cual es útil cuando se dispone de una flota de vehículos o, en caso de disponer de uno, para conocer los parámetros con valores distintos de *None*, es decir, los parámetros con valores conocidos. Esta función es utilizada en los pasos siguientes para dar tanto de alta —con la función *add_attribute_listener*— como de baja —con la función *remove_attribute_listener*— un atributo entre los observadores para nuestro vehículo.

El siguiente paso en este código es añadir un observador usando un decorador sobre el atributo *mode* del vehículo, lo cual le afecta de manera perpetua ya que el decorador va a estar siempre activo y va a ser imposible eliminarlo de los observadores. Es por ello por lo que se le puede modificar el modo de funcionamiento, pero requerimos de un *try-except* para eliminar el atributo *mode* de la lista de observadores ya que al intentar ejecutar este código debe fallar.

Los pasos siguientes son similares a los anteriores ya que se declara otra función que anuncia la modificación de cualquier parámetro, se testea añadiéndolos como observadores para después eliminarlos. Se continua con la modificación de algunos parámetros, se imprimen por pantalla todos aquellos de los que dispone el vehículo y, tras esto, se declara el decorador de otro parámetro y se testea con él.

Tras una serie de modificaciones simples de parámetros, el código finaliza con el cierre del objeto *vehicle* así como con SITL en caso de haber sido usado.

A la hora de probar este código, tanto en el simulador como en el dron real, en ambos casos, el resultado ha sido exitoso devolviéndonos todos los atributos del vehículo. Entre ellos están, por citar algunos: la versión del parche utilizada, la capacidad de soportar tipos de números (flotantes o enteros), transferencia de ficheros a través de FTP, la posición actual del vehículo y el porcentaje de la batería.

```
STABILIZE> Mode STABILIZE
Received 1138 parameters
Saved 1138 parameters to mav.parm
APM: Barometer 1 calibration complete
Init Gyro***

Ready to FLY APM: GPS 1: detected as u-blox at 115200 baud
APM: EKF2 IMU0 initial yaw alignment complete
APM: EKF2 IMU1 initial yaw alignment complete
APM: EKF2 IMU0 tilt alignment complete
APM: EKF2 IMU1 tilt alignment complete
APM: EKF2 IMU0 Origin set to GPS
APM: EKF2 IMU1 Origin set to GPS
APM: ArduCopter V3.7.0-dev (6774bab3)
APM: 0e68e9ef23d049839a595a803492b70d
APM: Frame: QUAD
Got MAVLink msg: COMMAND_ACK {command : 520, result : 0}
APM: EKF2 IMU0 is using GPS
APM: EKF2 IMU1 is using GPS
Got MAVLink msg: COMMAND_ACK {command : 11, result : 0}
APM: Arming motors
Got MAVLink msg: COMMAND_ACK {command : 400, result : 0}
Got MAVLink msg: COMMAND_ACK {command : 400, result : 0}
ARMED
GUIDED_NOGPS> Mode GUIDED_NOGPS
APM: EKF2 IMU0 in-flight yaw alignment complete
APM: EKF2 IMU1 in-flight yaw alignment complete
Flight battery 100 percent
Flight battery 90 percent
Flight battery 80 percent
Flight battery 70 percent
Flight battery 50 percent
Flight battery 40 percent
Flight battery 30 percent
Flight battery 20 percent
Flight battery warning
Flight battery 10 percent
Flight battery warning
APM: ArduCopter V3.7.0-dev (6774bab3)
APM: 0e68e9ef23d049839a595a803492b70d
APM: Frame: QUAD
Flight battery 0 percent
Flight battery warning
APM: ArduCopter V3.7.0-dev (6774bab3)
APM: 0e68e9ef23d049839a595a803492b70d
APM: Frame: QUAD
Flight battery warning
APM: ArduCopter V3.7.0-dev (6774bab3)
APM: 0e68e9ef23d049839a595a803492b70d
APM: Frame: QUAD
Got MAVLink msg: COMMAND_ACK {command : 520, result : 0}
Got MAVLink msg: COMMAND_ACK {command : 179, result : 4}
Got MAVLink msg: COMMAND_ACK {command : 11, result : 0}
GUIDED> Mode GUIDED
Got MAVLink msg: COMMAND_ACK {command : 11, result : 0}
STABILIZE> Mode STABILIZE
APM: Disarming motors
DISARMED
MAV> Flight battery warning
```

Figura 63: MAVProxy generado

```
miguel@miguel-SAT-P50-C:~/Descargas/dronekit-python-master/examples/vehicle_state$ sudo python vehicl
e_state.py.modificado --connect 127.0.0.1:14551

Connecting to vehicle on: 127.0.0.1:14551
>>> ArduCopter V3.7.0-dev (6774bab3)
>>> 0e68e9ef23d049839a595a803492b70d
>>> Frame: QUAD

Get all vehicle attribute values:
 Autopilot Firmware version: APM:Copter-3.7.0-dev0
   Major version number: 3
   Minor version number: 7
   Patch version number: 0
   Release type: dev
   Release version: 0
   Stable release?: False
 Autopilot capabilities
   Supports MISSION_FLOAT message type: True
   Supports PARAM_FLOAT message type: True
   Supports MISSION_INT message type: True
   Supports COMMAND_INT message type: True
   Supports PARAM_UNION message type: False
   Supports ftp for file transfers: False
   Supports commanding attitude offboard: True
   Supports commanding position and velocity targets in local NED frame: True
   Supports set position + velocity targets in global scaled integers: True
   Supports terrain protocol / data handling: True
   Supports direct actuator control: False
   Supports the flight termination command: True
   Supports mission_float message type: True
   Supports onboard compass calibration: True
 Global Location: LocationGlobal:lat=39.3075049,lon=-0.3184742,alt=5.04
 Global Location (relative altitude): LocationGlobalRelative:lat=39.3075049,lon=-0.3184742,alt=5.028
 Local Location: LocationLocal:north=-35.0856399536,east=46.7540664673,down=-5.02882194519
 Attitude: Attitude:pitch=0.000163919918123,yaw=-2.54421472549,roll=6.17438636255e-05
 Velocity: [0.0, 0.0, 0.0]
 GPS: GPSInfo:fix=6,num_sat=10
 Gimbal status: Gimbal: pitch=None, roll=None, yaw=None
 Battery: Battery:voltage=12.242,current=20.43,level=0
 EKF OK?: True
 Last Heartbeat: 0.0509142159999
 Rangefinder: Rangefinder: distance=None, voltage=None
 Rangefinder distance: None
 Rangefinder voltage: None
 Heading: 214
 Is Armable?: True
 System status: ACTIVE
 Groundspeed: 0.00578317465261
 Airspeed: 0.0490000024438
 Mode: GUIDED_NOGPS
 Armed: True

 Home location: LocationGlobal:lat=39.3078193665,lon=-0.319016993046,alt=0.019999999553

Set new home location
 New Home Location (from attribute - altitude should be 222): LocationGlobal:lat=39.3075049,lon=-0.31
84742,alt=222.0
 New Home Location (from vehicle - altitude should be 222): LocationGlobal:lat=39.3078193665,lon=-0.3
19016993046,alt=0.019999999553
```

Figura 64: Simulación del código vehicle_state.py

7.4.2. Follow_me.py

Habiendo ya realizado todos los simulacros, había llegado el momento de realizar las pruebas pertinentes con el vehículo real. Para ello, nos desplazamos al campo de vuelo y creamos un proxy que nos sirviera para comunicar con el vehículo y que nos permitiera enviarle un código a ejecutar a la vez que tener un control visual sobre sus movimientos, por lo que requeríamos de dos puertos: el 14500 y el 14501. Siendo así, enlazamos el proxy de MAVLink con la antena telemétrica conectada al ordenador y lanzamos la orden al terminal:

sudo mavproxy.py --master /dev/ttyTelemetry --out 127.0.0.1:14500 --out 127.0.0.1:14501

En segundo lugar, tenemos *follow_me.py*, otro código de los de ArduPilot cuya función principal es reubicarse en función de las coordenadas GPS que va recibiendo. Para ello, es requisito indispensable que el ordenador desde donde se emita la señal disponga de antena GPS, bien instalada internamente o bien como dispositivo externo que podamos agregar a través de un puerto USB.

Tras verificar las pruebas citadas anteriormente y estando ya solventado el tema de los variadores, fue cuando inicializamos las pruebas del follow_me. Fue entonces cuando ejecutamos el código *follow_me.py* el cual, al conectar el GPS a nuestro ordenador, funcionaba con cierto retraso, por lo que anulamos el *time.sleep(2)* situado al final del código de modo que el dron estaba situado prácticamente encima de nosotros, por lo que lo dimos por totalmente verificado y con funcionamiento exitoso.

Finalizadas estas pruebas, diseñamos unos códigos que forzaban que el vehículo modificase su actitud respecto a alguno (o algunos) de sus parámetros: *yaw*, *pitch* y *roll*. Estos serían los códigos que posteriormente se convertirían en aquellos que tendrían capacidad visual. Teniendo estos códigos, verificamos primeramente con el simulador que éstos acataban correctamente nuestro propósito, por lo que dimos el visto bueno para llevar a cabo la experimentación real.

Figura 65: Dron persiguiendo con follow_me.py

7.4.3. Realimentación visual

Llegado el momento de realizar las pruebas de visión, tomamos un tapón de color azul para la simulación que cogía imágenes de la cámara real y una manta de color rojizo, dado que en un paraje con vegetación era difícil de encontrarse un elemento con esta pigmentación, con la que uno de nosotros se cubriera con ella para que nuestro vehículo tuviera que interactuar con dicho color al detectarlo.

Tras darnos cuenta de que, en ocasiones, cuando el vehículo no la detectaba por estar fuera de rango, se acercaba a nuestra ubicación y detectaba los faros traseros del vehículo, decidimos cambiar de color, pasando a usar el color blanco de un vehículo, dado que esa tonalidad no se encontraba en nuestro escenario.

Habiendo configurado el color, realizamos las primeras pruebas con el simulador las cuales desglosamos en cuatro fases y, por tanto, cuatro códigos. Para poder simularlos, situábamos el tapón en un punto estratégico y comprobábamos si el dron simulado actuaba como se pretendía utilizando la estación de tierra.

El primero de los códigos se basaba en la variación del ángulo que forma el vehículo con el norte geográfico, o sea, la variación del *yaw* para que el dron estuviera siempre apuntando hacia el objetivo. Para que el dron simulado se detuviese, teníamos que ubicar el tapón en la línea vertical que separa la imagen en dos mitades, pero siempre en la parte superior de la imagen de modo que el vehículo simulado lo entendía como que está mirando al objetivo.

El segundo código pretendía desplazar al vehículo alrededor del eje X del propio vehículo, o sea, la variación del *roll* para que se situase enfrente o detrás del objetivo según su ubicación. Para que el dron virtual cesase en el intento de colocarse en línea con el tapón, debíamos colocar el tapón en la línea vertical que secciona la imagen por la mitad siendo esta vez indistinto si se encuentra en la parte superior o inferior de la imagen, dado que se limita a ubicarse en línea con el objetivo.

El tercer código consistiría en desplazar el vehículo alrededor del eje Y, o sea, la variación del *pitch* en función de la ubicación del objetivo, con lo que conseguiríamos hacerlo avanzar o retroceder. Para que el dron virtual cesase en el intento de colocarse en línea con el tapón, debíamos colocar el tapón en la línea horizontal que divide la imagen por la mitad siendo indistinto si se encuentra en la parte izquierda o derecha de la imagen, dado que se limita a alinearse con el objetivo.

El cuarto y último de estos códigos aunaba los tres anteriores de modo que el dron siempre trataría de ubicarse justo encima del objetivo encarándose a él utilizando el *yaw* y desplazándose con la ayuda tanto del *roll* como del *pitch*. De modo adicional, a este último código le añadimos una funcionalidad extra: el control de la altura. Para ello, a cada iteración del dron comprobaría si se había salido de la altura deseada con un margen de histéresis y, en caso de no estar en dicho rango, se reajustaría modificando el *thrust*.

Una vez creado el proxy, lanzamos a ejecución primeramente el código que manipulaba la orientación del vehículo respecto al norte geográfico por el simple motivo de que en caso de no responder bien a las órdenes indicadas —cosa que no nos preocupaba dado que ya sabíamos que el comportamiento iba a ser muy similar al

del simulador— sería más fácil devolverlo al origen. Para ejecutarlo, lanzamos la orden:

sudo python rotation.py --connect 127.0.0.1:14501

Como ya hemos visto anteriormente, en el código original, el modo de funcionamiento al que sometíamos al dron era "*GUIDED_NOGPS*". En un principio, con el simulador pudimos apreciar un funcionamiento correcto, pero cuando nos dispusimos a volar nuestro dron, tuvimos que reajustar el modo de funcionamiento a "*GUIDED*" de modo que sí tuviera en cuenta la ubicación GPS para que permaneciera quieto durante el proceso ya que, de lo contrario, poco a poco se desviaba.

Este detalle no lo valoramos con la simulación debido a que para lo que nosotros pudimos observar en el mapa, parecía ser un movimiento muy discreto, lo que nos hacía pensar que no iba a ser de gran notoriedad en el dron real. No obstante, llegado el momento de comprobarlo, pudimos apreciar que realmente sí era notorio ya que era necesario perseguir al dron el cual, aun mirando al objetivo, continuaba desplazándose debido a su modo de funcionamiento.

La última de las pruebas fue la ejecución del código que aunaba todos los parámetros (*pitch, roll, yaw* y *thrust*) ya que, si se había comprobado que nuestro vehículo se comportaba de igual modo que el simulador, no era menester comprobar cada uno de los códigos que segmentan el movimiento del dron, sino que era más eficiente comprobar el resultado que englobaba todas estas acciones. Así pues, nos decidimos a ejecutar este código al que llamamos "*persecution.py*" con la orden:

sudo python persecution.py --connect 127.0.0.1:14501

Este código nos llevó más tiempo en experimentación real, por lo que tuvimos que seccionar la experimentación debido a la duración de la batería y a los fuertes vientos que se producían a determinadas horas.

Fue durante estas pruebas cuando aún utilizábamos la manta roja mencionada con anterioridad y, tras algunos intentos, acababa focalizando el foco de la furgoneta, impidiendo la focalización sobre la manta y, como habíamos estado haciendo pruebas con el código *rotation.py*, tuvimos que parar debido al desgaste de la batería.

Continuando con las pruebas, nos dio la sensación de que el *pitch* funcionaba en sentido contrario, por lo que decidimos invertir el signo de nuestra variable proporcional, con lo que pudimos comprobar que nuestra sensación era errónea, ya que esta vez, con el signo invertido, su comportamiento sí que era notoriamente contrario al deseado y, por tanto, el movimiento previo era el correcto aunque debíamos tratar de mantenerlo fijado en el aire siempre y cuando no detectase su objetivo el cual sería esta vez un vehículo de color blanco.

Figura 66: Dron interactuando con vehículo

Figura 67*: Objetivo marcado por el punto rojo*

De modo que devolvimos el signo original a esta variable y comprobamos que lo que realmente sucedía con anterioridad era que en ocasiones la cámara no alcanzaba a visualizar el objetivo debido a su campo de visión, por lo que optamos por volver a modificar en este caso el modo de funcionamiento a "*GUIDED"* de manera que mientras no detectase el objeto, permanecería estático en el aire.

8. Conclusiones

En definitiva, este trabajo ha significado una experimentación muy cercana al mundo laboral, así como un refuerzo de todo lo aprendido a lo largo de los estudios. Entre todos estos aspectos, sería destacable la profundización en el lenguaje Python así como en algunas de sus librerías además del refuerzo en los conocimientos sobre las comunicaciones inalámbricas o el uso avanzado de terminales con base Ubuntu.

Respecto al trabajo, podemos considerar que se han alcanzado todos los objetivos planteados al inicio ya que hemos conseguido pilotar el vehículo con el mando y hemos conseguido que actúe autónomamente —tanto a ciegas como con visión— en base a un código que le transferimos aunque en los aspectos de visión podría ser necesario el refinamiento de los movimientos debido a que en ocasiones, como ya hemos indicado con anterioridad, la cámara perdía la imagen del objetivo teniendo así que reubicarnos para que nos volviese a localizar.

Algunos de los problemas encontrados han sido: la falta de conocimiento en soldaduras, los cuales solucionamos a base de practicar en planchas ajenas al trabajo para evitar roturas; la desaparición de la API oficial de DroneKit durante unos meses; la avería de algunos de los variadores los cuales tuvimos que ir reemplazando a medida que se estropeaban hasta que, habiéndose estropeado la mitad de ellos, decidimos reemplazar todos los restantes; la rotura de uno de los separadores que quedó atorado en una de las roscas necesitando de un taladro de métrica milimétrica que introducimos por un lateral de la rosca consiguiendo expulsar el fragmento de rosca atorada y la necesidad de formatear el ordenador para cambiar de sistema operativo, pues la versión 18.04 de Ubuntu no funcionaba correctamente con el simulador y, por tanto, era de mayor temeridad el tener que ejecutar un código en el vehículo sin haberlo probado antes.

Consideramos también destacable la necesidad de citar algunas de las competencias transversales que más hemos reforzado con la realización de este trabajo como son por ejemplo: la de la instrumental específica, pues tratamos de utilizar la versión más moderna de Ubuntu y tuvimos que retroceder por necesidad así como el uso de la última versión de ArduPilot, la necesidad de aprender a soldar en el mínimo tiempo posible y el aprendizaje sobre el mantenimiento de baterías; la competencia de planificación y gestión del tiempo, pues empezamos a mitad curso por problemas con otro trabajo de final de grado además de la competencia de aplicación y pensamiento práctico basándonos en los errores a la hora de experimentar durante el vuelo así como a la hora de programar teniendo que poner a prueba lo aprendido durante los estudios y, en ocasiones, durante las tutorías.

En suma, consideramos este un proyecto muy amplio ya que no solo utilizamos conocimientos teóricos, sino que nos obliga a trabajar con herramientas físicas así como la necesidad de adquirir información de ciertos conocimientos aeronáuticos de extrema importancia para poder manejar correctamente el dron teniendo que buscar la información en las páginas oficiales de los fabricantes y en foros —en ambos casos mayoritariamente en inglés— además de reforzar la comprensión de código sin existencia de la API pertinente.

9. Trabajos futuros

Como posibles trabajos futuros, existe la posibilidad de perfeccionar el funcionamiento del dron con un controlador PID que suavizase la brusquedad de algunos movimientos, así como que fuera capaz de relacionar la altura a la que se encuentra con la distancia que debe desplazarse.

Este vehículo podría destinarse a actividades de vigilancia perimetral para lo cual podríamos añadirle una tarjeta SIM telefónica y con la que pudiera ponerse en contacto con el usuario a través de mensajería para notificarle en caso de detectar alguna intrusión en la zona vigilada.

También podría utilizarse para la detección de incendios sustituyendo la cámara que disponemos ahora por una con visión térmica. De este modo, no solo sería más fácil rastrear los incendios, sino que también sería posible detectar posibles focos en los que se pudieran ocasionar otros nuevos.

Por último, quedaría también pendiente el estudio de nuevas y distintas formas de trabajar con la visión por computador de modo que se pudiera diferenciar formas e incluso la detección de rostros.

10. Bibliografía

[1] Diccionario de la Real Academia Española.
https://dle.rae.es/
[2] Historia de los drones
http://eldrone.es/historia-de-los-drones/
https://es.digitaltrends.com/drones/la-historia-de-los-drones/
[3] Principios básicos de vuelo – Vuelo artificial.
https://vueloartificial.com/introduccion/toma-de-contacto/principios-basicos-de-vuelo/
[4] ¿UAV, UAS, RPAS o drones?
http://www.inta.es/WEB/INTA/es/blogs/copernicus/BlogEntry_1553849310660
[5] ¿Qué diferencias hay entre un cuadricóptero y un dron de ala fija?
https://www.todrone.com/cuadricoptero-dron-ala-fija/
[6] Raspberry Pi 3 Model B
https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[7] Emlid Navio2
https://store.emlid.com/product/navio2/
[8] MAVLink - Wikipedia
https://en.wikipedia.org/wiki/MAVLink
[9] Normativa sobre drones en España [2019] - Aerial Insights
https://www.aerial-insights.co/blog/normativa-drones-espana/
[10] Disposición 15721 del BOE núm. 316 de 2017 - BOE-A-2017-15721.pdf
https://www.boe.es/boe/dias/2017/12/29/pdfs/BOE-A-2017-15721.pdf
[11] ArduPilot
http://ardupilot.org/
[12] Dronekit
https://dronekit.io/
[13] MAVLink
https://mavlink.io/en/
[14] Guía de cuidado de baterías: cuida las baterías LiPo de tu dron y mantenlas como el primer día
https://www.wikiversus.com/electronica-y-gadgets/drones/uso-cuidado-mantenimiento-baterias-lipo/
[15] Salarios para empleos de Ingeniero/a informático/a en España
https://www.indeed.es/salaries/Ingeniero/a-inform%C3%A1tico/a-Salaries
[16] Electronic Speed Controller (ESC) Calibration — Copter documentation
http://ardupilot.org/copter/docs/esc-calibration.html
[17] DJI F550 / 450 Folding Landing gear legs by CVMichael - Thingiverse
https://www.thingiverse.com/thing:759368
[18] A

11. Anexos

11.1. Anexo A - Presupuesto

Cantidad	Producto	Precio por unidad	Precio
1	DJI F550 Hexa Kit ARF	279,00 €	279,00 €
1	FrSky 2,4 GHz ACCST Taranis X9D Plus and X8R Combo Digital Telemetry Radio System with case (Mode 2)	279,00 €	279,00 €
1	Radio Telemetry Kit-433MHz Compatible PX4 & APM	69,00 €	69,00 €
1	Cargador Imax B6	29,90 €	29,90 €
1	Fuente de alimentación profesional Robbe SPS 230V 12A	49,90 €	49,90 €
2	Batería Lipo 6600mA 35-70C 4S2P 14,8V	119,00 €	238,00 €
1	14-AWG - Cable de silicona - 400 * 0,08 - Rojo - 2,5mm	3,80 €	3,80 €
1	14-AWG - Cable de silicona - 400 * 0,08 - Negro - 2,5mm	3,80 €	3,80 €
2	XT60 Macho original / GENUINE	1,30 €	2,60 €
2	Conector XT60 hembra original xt	1,30 €	2,60 €
15	Separador Hexagonal, Wurth Elektronik, Hexagonal, 12mm, M2.5	0,80 €	12,00 €
		Total	969,60 €

11.2. Anexo B – Configuración del mando Taranis X9D

El mando Taranis X9D se caracteriza por disponer de gran versatilidad a la hora de programar tanto sus palancas como sus interruptores ya que, por ejemplo —y como caso principal— los factores que modifican la actitud del dron pueden ir en cualquier canal ya que nosotros hemos conectado el receptor del mando en el dron al canal *SBUS* que recoge todas las señales, por lo que creemos necesario explicar aquí como lo hemos programado para así hacer entender de la mejor manera posible cómo manejábamos el vehículo cuando era necesario que usásemos el mando.

Para ello, en primer lugar, pulsamos "Menú" y nos desplazamos a una casilla (que se corresponde con modelo) y que según el vehículo que usemos podemos alternar. Una vez elegido el número de modelo lo seleccionamos (ahora, y en todo momento, la tecla de validación será "*Ent"*) manteniendo presionada la tecla hasta que se visualice un pop-up y seleccionamos "*Create model".*

En el siguiente menú seleccionamos —desplazándonos con las teclas - y +— el tipo de vehículo que vamos a usar, en nuestro caso, el dron. Una vez seleccionado, presionaremos el botón *Page* —dejando así por defecto las variables como el sistema las considera necesarias—hasta que nos pida escoger en qué canal deseamos insertar cada acción (*throttle, ailerons, elevator* y *rudder*) el cual será el último de dicho menú. Al salir de dicho menú, aparece la casilla que hemos seleccionado marcada con un asterisco, lo cual significa que tenemos seleccionada dicha opción.

A modo de comprobación, si seleccionamos el botón "*Exit"*, volvemos a la pantalla principal en la que esta vez veremos "*MODEL XX"* donde "XX" se corresponde con el número de modelo seleccionado. Volviendo al menú (presionando la tecla oportuna si hemos hecho esta comprobación), pasamos las páginas con el botón "*Page*" hasta llegar a la página 6/12 —visible en la esquina superior derecha de la pantalla del mando— que es la encargada de unir las acciones de todos los canales.

A nuestra manera de ver las palancas, la configuración ideal de nuestro mando sería que la palanca derecha controlase *pitch* y *roll* —*engine* y *aileron*— siendo el primero el que se controle con el eje vertical de dicha palanca y el segundo con el eje horizontal y, que la palanca derecha controlase *thrust* y *yaw* —*elevator* y *rudder*— correspondiéndose con los ejes vertical y horizontal respectivamente.

Para ello, presionamos con una larga pulsación la tecla "*Ent*" sobre el canal que queremos editar de modo que nos deje seleccionar la opción "*Edit*", bajamos a la opción "*Source*", presionamos ligeramente "*Ent*" para que se ponga a parpadear y zarandeamos la palanca que queremos asignarle a dicho canal en el sentido del eje con el que queremos que se corresponda siendo visible entonces el cambio. A modo de evitar confusiones, podemos modificar el nombre de "*Mix Name*" el cual será el que veamos en la estación de tierra y desde fuera de la configuración a algún nombre que nos facilite recordar qué acción activa ese canal.

Habiendo llegado a este punto, es necesario emparejar la emisora con la receptora para evitar que una emisora se comunique con varias receptoras a la vez. Así pues, con un objeto punzante (como un bolígrafo o una aguja) mantenemos

presionado el botón F/0 de la receptora y, sin soltarlo, le damos alimentación. Por otra parte, es necesario poner el mando emisor en modo escucha para que pueda captar la antena receptora que está tratando de enlazarse con él. Esto lo conseguimos accediendo al final de la página 2 del menú y buscamos la opción "*Bind*", la seleccionamos y queda en modo escucha a la vez que emite un sonido.

Estando pues en modo escucha, procedemos a alimentar la antena receptora mientras, como ya hemos dicho, presionamos el botón F/0 de la antena. Si no se ha llevado a cabo con éxito, el LED parpadeará en rojo y, en caso de haberse llevado a cabo con éxito, alterna parpadeando el rojo y el verde. Esto significa que se ha enlazado exitosamente, por lo que volvemos a presionar "*Ent*" en el mando para detener el enlazado y, tras esto, desconectamos la alimentación que sustentaba la antena receptora. Al volver a conectarla, debe quedarse en verde como significado de que se ha enlazado correctamente con el mando y, por otra parte, en el mando, al lado de la batería, debe visualizarse un símbolo de una antena junto con unas líneas que representan la cobertura.

Hecho esto, conectamos el dron a la estación de tierra (ya sea por telemetría o Wi-Fi) para configurar los modos de vuelo. Así pues, seleccionamos el canal en el que queremos que funcione y buscamos la opción "*Source*" en la que, como habíamos hecho con las acciones de vuelo anteriormente, presionamos la tecla "*Ent*" y accionamos el interruptor al que queremos asignar el modo de funcionamiento. Dado que nosotros queríamos experimentar 3 modos de vuelo además del *GUIDED*, escogimos un interruptor con 3 posiciones como es el caso del interruptor SG (visible en la Figura 21).

Habiendo accionado el botón y habiendo cambiado por tanto el valor de la variable "*Source*", nos desplazamos a la variable "*Switch*" ya que tenemos que definir en qué posición queremos cada modo de vuelo, por lo que volvemos a seleccionar "*Ent*" para que parpadee, colocamos el interruptor en dicha posición. Por último, en este apartado, es necesario definir si queremos que dicho modo de funcionamiento se añada al actual, lo reemplace o lo multiplexe. En nuestro caso, preferimos que lo reemplace para que solo exista un modo de funcionamiento activado a la vez por lo que, si queremos que se reemplacen entre ellos, una vez finalizada esta configuración, podemos añadir más volviendo al mismo canal, mantenemos presionada la tecla "*Ent*" y seleccionamos la opción "*Insert After*" repitiendo el proceso para un modo distinto y una posición del interruptor distinta.

A modo de comprobación, permaneciendo en el menú de los canales, se puede alternar la posición de este interruptor y podemos apreciar como se pone en negrita una línea u otra dependiendo de la configuración aplicada a dicho interruptor. Es recomendable instalar un modo de retorno a casa (*Return To Launch* o *RTL*) *de* modo que cuando se accione este interruptor, se establezca por encima de cualquier otra acción. Para ello, repetimos el proceso ya citado dejando el valor de "*Source*" con el valor de un interruptor distinto al que hemos estado usando hasta ahora y, a la opción "*Switch*", le adjudicamos dicho interruptor.

En nuestro caso hemos escogido SF debido a que tiene 2 posiciones y, mientras esté hacia adelante (o hacia abajo) esté inhabilitado y, si lo colocamos hacia atrás (o hacia arriba) lo activamos dejando inutilizados el resto de los interruptores. Esto es fácilmente comprobable volviendo al menú de canales y activar el interruptor

que tenga el modo RTL y, tras él, intentar cambiar de modo de funcionamiento accionando el otro interruptor (SG), siendo esto imposible.

A continuación, necesitamos acceder a la estación de tierra (QGroundControl) para ajustar todos estos parámetros. Para ello, dentro de la aplicación, seleccionamos el icono de los engranajes (Opciones) y accedemos al menú "*Radio*", en él encontramos los 4 ejes de funcionamiento del mando, donde al zarandear alguno de ellos, podemos observar cómo varía su barra en la pantalla a proporción.

El siguiente paso sería la configuración de los sensores, los cuales podemos encontrar en el siguiente menú ("*Sensors*") de modo que siguiendo las instrucciones que la pantalla te indica son fácilmente calibrables.

Para configurar los modos de vuelo, accedemos al menú "*Flight modes*" con el que podemos ver que, accionando el interruptor en diversas posiciones, los modos en los que entra no son los deseados. Esto lo podemos corregir entrando en el mando emisor en el menú de edición del modo de funcionamiento que queremos ajustar y nos desplazamos a la opción "*Offset*" —que se corresponde con la señal PWM del mando— la cual reduciremos si queremos que se desplace hacia modos situados en posiciones de menor índice o incrementaremos si queremos que se desplace hacia modos situados en posiciones de mayor índice. No es necesario detenernos justo en el umbral, es decir, en el valor en el que alcance la posición deseada, sino que podemos modificar un poco más el "*Offset*" de modo que no quede justo en la línea y pueda alterarse en pleno vuelo.

Una vez hecho esto, para seleccionar el modo de vuelo que queremos adjudicar a cada posición del interruptor, basta con abrir el menú desplegable de cada uno de estos modos de vuelo y seleccionar el que deseemos, siempre manteniendo el RTL en el modo que se superpone al resto para garantizar la vuelta al origen en circunstancias peligrosas.

11.3. Anexo C – Códigos Python

11.3.1. vehicle_state.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
© Copyright 2015-2016, 3D Robotics.
vehicle_state.py:

Demonstrates how to get and set vehicle state and parameter information,
and how to observe vehicle attribute (state) changes.

Full documentation is provided at http://python.dronekit.io/examples/vehicle_state.html
"""
from __future__ import print_function
from dronekit import connect, VehicleMode
import time

#Set up option parsing to get connection string
import argparse
parser = argparse.ArgumentParser(description='Print out vehicle state information. Connects to SITL on local PC by default.')
parser.add_argument('--connect',
                    help="vehicle connection target string. If not specified, SITL automatically started and used.")
args = parser.parse_args()
connection_string = args.connect
sitl = None

#Start SITL if no connection string specified
if not connection_string:
    import dronekit_sitl
    sitl = dronekit_sitl.start_default()
```

```
    connection_string = sitl.connection_string()
# Connect to the Vehicle.
#   Set `wait_ready=True` to ensure default attributes are populated before `connect()` returns.
print("\nConnecting to vehicle on: %s" % connection_string)
vehicle = connect(connection_string, wait_ready=True)


vehicle.wait_ready('autopilot_version')


# Get all vehicle attributes (state)
print("\nGet all vehicle attribute values:")
print(" Autopilot Firmware version: %s" % vehicle.version)
print("   Major version number: %s" % vehicle.version.major)
print("   Minor version number: %s" % vehicle.version.minor)
print("   Patch version number: %s" % vehicle.version.patch)
print("   Release type: %s" % vehicle.version.release_type())
print("   Release version: %s" % vehicle.version.release_version())
print("   Stable release?: %s" % vehicle.version.is_stable())
print(" Autopilot capabilities")
print("   Supports MISSION_FLOAT message type: %s" % vehicle.capabilities.mission_float)
print("   Supports PARAM_FLOAT message type: %s" % vehicle.capabilities.param_float)
print("   Supports MISSION_INT message type: %s" % vehicle.capabilities.mission_int)
print("   Supports COMMAND_INT message type: %s" % vehicle.capabilities.command_int)
print("   Supports PARAM_UNION message type: %s" % vehicle.capabilities.param_union)
print("   Supports ftp for file transfers: %s" % vehicle.capabilities.ftp)
print("   Supports commanding attitude offboard: %s" % vehicle.capabilities.set_attitude_target)
print("   Supports commanding position and velocity targets in local NED frame: %s" % vehicle.capabilities.set_attitude_target_local_ned)
print("   Supports set position + velocity targets in global scaled integers: %s" % vehicle.capabilities.set_altitude_target_global_int)
print("   Supports terrain protocol / data handling: %s" % vehicle.capabilities.terrain)
```

```
print("   Supports direct actuator control: %s" %
vehicle.capabilities.set_actuator_target)
print("   Supports the flight termination command: %s" %
vehicle.capabilities.flight_termination)
print("   Supports mission_float message type: %s" % vehicle.capabilities.mission_float)
print("   Supports onboard compass calibration: %s" %
vehicle.capabilities.compass_calibration)
print(" Global Location: %s" % vehicle.location.global_frame)
print(" Global Location (relative altitude): %s" % vehicle.location.global_relative_frame)
print(" Local Location: %s" % vehicle.location.local_frame)
print(" Attitude: %s" % vehicle.attitude)
print(" Velocity: %s" % vehicle.velocity)
print(" GPS: %s" % vehicle.gps_0)
print(" Gimbal status: %s" % vehicle.gimbal)
print(" Battery: %s" % vehicle.battery)
print(" EKF OK?: %s" % vehicle.ekf_ok)
print(" Last Heartbeat: %s" % vehicle.last_heartbeat)
print(" Rangefinder: %s" % vehicle.rangefinder)
print(" Rangefinder distance: %s" % vehicle.rangefinder.distance)
print(" Rangefinder voltage: %s" % vehicle.rangefinder.voltage)
print(" Heading: %s" % vehicle.heading)
print(" Is Armable?: %s" % vehicle.is_armable)
print(" System status: %s" % vehicle.system_status.state)
print(" Groundspeed: %s" % vehicle.groundspeed)    # settable
print(" Airspeed: %s" % vehicle.airspeed)    # settable
print(" Mode: %s" % vehicle.mode.name)    # settable
print(" Armed: %s" % vehicle.armed)    # settable

# Get Vehicle Home location - will be `None` until first set by autopilot
while not vehicle.home_location:
    cmds = vehicle.commands
    cmds.download()
    cmds.wait_ready()
    if not vehicle.home_location:
        print(" Waiting for home location ...")
```

```
# We have a home location, so print it!
print("\n Home location: %s" % vehicle.home_location)
# Set vehicle home_location, mode, and armed attributes (the only settable attributes)

print("\nSet new home location")
# Home location must be within 50km of EKF home location (or setting will fail silently)
# In this case, just set value to current location with an easily recognisable altitude (222)
my_location_alt = vehicle.location.global_frame
my_location_alt.alt = 222.0
vehicle.home_location = my_location_alt
print(" New Home Location (from attribute - altitude should be 222): %s" % vehicle.home_location)

#Confirm current value on vehicle by re-downloading commands
cmds = vehicle.commands
cmds.download()
cmds.wait_ready()
print(" New Home Location (from vehicle - altitude should be 222): %s" % vehicle.home_location)

print("\nSet Vehicle.mode = GUIDED (currently: %s)" % vehicle.mode.name)
vehicle.mode = VehicleMode("GUIDED")
while not vehicle.mode.name=='GUIDED':  #Wait until mode has changed
    print(" Waiting for mode change ...")
    time.sleep(1)

# Check that vehicle is armable
while not vehicle.is_armable:
    print(" Waiting for vehicle to initialise...")
    time.sleep(1)
    # If required, you can provide additional information about initialisation
    # using `vehicle.gps_0.fix_type` and `vehicle.mode.name`.

#print "\nSet Vehicle.armed=True (currently: %s)" % vehicle.armed
```

```
#vehicle.armed = True
#while not vehicle.armed:
#    print " Waiting for arming..."
#    time.sleep(1)
#print " Vehicle is armed: %s" % vehicle.armed

# Add and remove and attribute callbacks

#Define callback for `vehicle.attitude` observer
last_attitude_cache = None
def attitude_callback(self, attr_name, value):
    # `attr_name` - the observed attribute (used if callback is used for multiple attributes)
    # `self` - the associated vehicle object (used if a callback is different for multiple vehicles)
    # `value` is the updated attribute value.
    global last_attitude_cache
    # Only publish when value changes
    if value!=last_attitude_cache:
        print(" CALLBACK: Attitude changed to", value)
        last_attitude_cache=value

print("\nAdd `attitude` attribute callback/observer on `vehicle`")
vehicle.add_attribute_listener('attitude', attitude_callback)

print(" Wait 2s so callback invoked before observer removed")
time.sleep(2)

print(" Remove Vehicle.attitude observer")
# Remove observer added with `add_attribute_listener()` specifying the attribute and callback function
vehicle.remove_attribute_listener('attitude', attitude_callback)

# Add mode attribute callback using decorator (callbacks added this way cannot be removed).
print("\nAdd `mode` attribute callback/observer using decorator")
```

```
@vehicle.on_attribute('mode')
def decorated_mode_callback(self, attr_name, value):
    # `attr_name` is the observed attribute (used if callback is used for multiple attributes)
    # `attr_name` - the observed attribute (used if callback is used for multiple attributes)
    # `value` is the updated attribute value.
    print(" CALLBACK: Mode changed to", value)

print(" Set mode=STABILIZE (currently: %s) and wait for callback" % vehicle.mode.name)
vehicle.mode = VehicleMode("STABILIZE")

print(" Wait 2s so callback invoked before moving to next example")
time.sleep(2)

print("\n Attempt to remove observer added with `on_attribute` decorator (should fail)")
try:
    vehicle.remove_attribute_listener('mode', decorated_mode_callback)
except:
    print(" Exception: Cannot remove observer added using decorator")

# Demonstrate getting callback on any attribute change
def wildcard_callback(self, attr_name, value):
    print(" CALLBACK: (%s): %s" % (attr_name,value))

print("\nAdd attribute callback detecting ANY attribute change")
vehicle.add_attribute_listener('*', wildcard_callback)
print(" Wait 1s so callback invoked before observer removed")
time.sleep(1)
print(" Remove Vehicle attribute observer")
# Remove observer added with `add_attribute_listener()`
vehicle.remove_attribute_listener('*', wildcard_callback)

# Get/Set Vehicle Parameters
print("\nRead and write parameters")
```

```
print(" Read vehicle param 'THR_MIN': %s" % vehicle.parameters['THR_MIN'])

print(" Write vehicle param 'THR_MIN' : 10")
vehicle.parameters['THR_MIN']=10
print(" Read new value of param 'THR_MIN': %s" % vehicle.parameters['THR_MIN'])

print("\nPrint all parameters (iterate `vehicle.parameters`):")
for key, value in vehicle.parameters.iteritems():
    print(" Key:%s Value:%s" % (key,value))

print("\nCreate parameter observer using decorator")
# Parameter string is case-insensitive
# Value is cached (listeners are only updated on change)
# Observer added using decorator can't be removed.

@vehicle.parameters.on_attribute('THR_MIN')
def decorated_thr_min_callback(self, attr_name, value):
    print(" PARAMETER CALLBACK: %s changed to: %s" % (attr_name, value))

print("Write vehicle param 'THR_MIN' : 20 (and wait for callback)")
vehicle.parameters['THR_MIN']=20
for x in range(1,5):
    #Callbacks may not be updated for a few seconds
    if vehicle.parameters['THR_MIN']==20:
        break
    time.sleep(1)

#Callback function for "any" parameter
print("\nCreate (removable) observer for any parameter using wildcard string")
def any_parameter_callback(self, attr_name, value):
    print(" ANY PARAMETER CALLBACK: %s changed to: %s" % (attr_name, value))

#Add observer for the vehicle's any/all parameters parameter (defined using wildcard
string ``'*'``)
```

```
vehicle.parameters.add_attribute_listener('*', any_parameter_callback)
print(" Change THR_MID and THR_MIN parameters (and wait for callback)")
vehicle.parameters['THR_MID']=400
vehicle.parameters['THR_MIN']=30

## Reset variables to sensible values.
print("\nReset vehicle attributes/parameters and exit")
vehicle.mode = VehicleMode("STABILIZE")
#vehicle.armed = False
vehicle.parameters['THR_MIN']=130
vehicle.parameters['THR_MID']=500

#Close vehicle object before exiting script
print("\nClose vehicle object")
vehicle.close()

# Shut down simulator if it was started.
if sitl is not None:
    sitl.stop()

print("Completed")
```

11.3.2. follow_me.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
© Copyright 2015-2016, 3D Robotics.
followme - Tracks GPS position of your computer (Linux only).

This example uses the python gps package to read positions from a GPS attached to your
laptop and sends a new vehicle.simple_goto command every two seconds to move the
vehicle to the current point.

When you want to stop follow-me, either change vehicle modes or type Ctrl+C to exit the script.

Example documentation: http://python.dronekit.io/examples/follow_me.html
"""
from __future__ import print_function

from dronekit import connect, VehicleMode, LocationGlobalRelative
import gps
import socket
import time
import sys

#Set up option parsing to get connection string
import argparse
parser = argparse.ArgumentParser(description='Tracks GPS position of your computer (Linux only).')
parser.add_argument('--connect',
                    help="vehicle connection target string. If not specified, SITL automatically started and used.")
args = parser.parse_args()
```

```
connection_string = args.connect
sitl = None

#Start SITL if no connection string specified
if not connection_string:
    import dronekit_sitl
    sitl = dronekit_sitl.start_default()
    connection_string = sitl.connection_string()

# Connect to the Vehicle
print('Connecting to vehicle on: %s' % connection_string)
vehicle = connect(connection_string, wait_ready=True, timeout=300)

def arm_and_takeoff(aTargetAltitude):
    """
    Arms vehicle and fly to aTargetAltitude.
    """

    print("Basic pre-arm checks")
    # Don't let the user try to arm until autopilot is ready
    while not vehicle.is_armable:
        print(" Waiting for vehicle to initialise...")
        time.sleep(1)

    print("Arming motors")
    # Copter should arm in GUIDED mode
    vehicle.mode = VehicleMode("GUIDED")
    vehicle.armed = True

    while not vehicle.armed:
        print(" Waiting for arming...")
        time.sleep(1)

    print("Taking off!")
```

```
    vehicle.simple_takeoff(aTargetAltitude) # Take off to target altitude

    # Wait until the vehicle reaches a safe height before processing the goto (otherwise the command
    #  after Vehicle.simple_takeoff will execute immediately).
    while True:
        print(" Altitude: ", vehicle.location.global_relative_frame.alt)
        if vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95: #Trigger just below target alt.
            print("Reached target altitude")
            break
        time.sleep(1)


try:
    # Use the python gps package to access the laptop GPS
    gpsd = gps.gps(mode=gps.WATCH_ENABLE)

    #Arm and take off to altitude of 5 meters
    arm_and_takeoff(5)

    while True:
        if vehicle.mode.name != "GUIDED":
            print("User has changed flight modes - aborting follow-me")
            break

        # Read the GPS state from the laptop
        next(gpsd)

        # Once we have a valid location (see gpsd documentation) we can start moving our vehicle around
        if (gpsd.valid & gps.LATLON_SET) != 0:
            altitude = 30  # in meters
            dest = LocationGlobalRelative(gpsd.fix.latitude, gpsd.fix.longitude, altitude)
            print("Going to: %s" % dest)
```

```
        # A better implementation would only send new waypoints if the position had
changed significantly
        vehicle.simple_goto(dest)

        # Send a new target every two seconds
        # For a complete implementation of follow me you'd want adjust this delay
        time.sleep(2)

except socket.error:
    print("Error: gpsd service does not seem to be running, plug in USB GPS or run run-
fake-gps.sh")
    sys.exit(1)

#Close vehicle object before exiting script
print("Close vehicle object")
vehicle.close()

# Shut down simulator if it was started.
if sitl is not None:
    sitl.stop()

print("Completed")
```

11.3.3. rotation.py

```
#!/usr/bin/env python

"""
rotation.py: (Copter Only)
        - Based on set_attitude_target.py from DroneKit Python.
This example shows how to move/direct Copter and send commands
 in GUIDED_NOGPS mode using DroneKit Python.
Caution: A lot of unexpected behaviors may occur in GUIDED_NOGPS mode.
     Always watch the drone movement, and make sure that you are in dangerless environment.
     Land the drone as soon as possible when it shows any unexpected behavior.
Tested in Python 2.7.10
"""
#################################
# IMPORTS SET_ATTRIBUTE_TARGET #
#################################

from dronekit import connect, VehicleMode, LocationGlobal, LocationGlobalRelative
from pymavlink import mavutil # Needed for command message definitions
import time
import math
###################
# IMPORTS CAMERA #
###################

from imutils.video import VideoStream
import imutils
import cv2
import numpy as np
import threading

import glob as g
import os
```

```
# Set up option parsing to get connection string
import argparse
parser = argparse.ArgumentParser(description='Control Copter and send commands in GUIDED mode ')
parser.add_argument('--connect',
                    help="Vehicle connection target string. If not specified, SITL automatically started and used.")
args = parser.parse_args()

connection_string = args.connect
sitl = None

# Start SITL if no connection string specified
if not connection_string:
    import dronekit_sitl
    sitl = dronekit_sitl.start_default()
    connection_string = sitl.connection_string()

# Connect to the Vehicle
print('Connecting to vehicle on: %s' % connection_string)
vehicle = connect(connection_string, wait_ready=True)
#-------------------------------------------------------------------------------------
########################################
#                    VIDEO                    #
########################################
def nothing(x):
    pass
res_h = 640
res_v = 480

x_dest =  0
y_dest =  0
z_dest =  5
pos_offset = [ x_dest, y_dest, z_dest ]
resolution=(res_v, res_h)
```

```
fps = 60
printedShape=False

dev = -1
# Looking for video decoder
for i in g.glob("/dev/video?"):
    print("Analizing " + i)
    resVid = os.popen("udevadm info -a -n " + i + " | grep \'{idVendor}\' | head -n1 | awk -F \'\"\' '{print $2}\'").read()[:-1]
    resPid = os.popen("udevadm info -a -n " + i + " | grep \'{idProduct}\' | head -n1 | awk -F \'\"\' '{print $2}\'").read()[:-1]
    print("resVid = " +resVid)
    print("resPid = " +resPid)
    if resVid == "1f4d" and resPid == "0102" :
        print("This!")
        dev = int(i[-1])
        print("Selected src = " + str(dev))
        break   # Break loop because we have found it
    else :
        print("This not.") # Discard this input
if dev == -1 :
    print("Any match. Aborting...")
    quit(); # Abort execution due to we have not found it

vs = VideoStream(src=dev, resolution=resolution, framerate=fps).start()
time.sleep(1.0)
rows,cols=frame.shape[:2]

# Thresholds
minH = 0
maxH = 255
minS = 0
maxS = 255
minV = 0
maxV = 255
```

```
# Create a window
cv2.namedWindow('binalized');

cv2.createTrackbar('H max', 'binalized', 175, 255, nothing);
cv2.createTrackbar('H min', 'binalized', 131, 255, nothing);
cv2.createTrackbar('S max', 'binalized', 145, 255, nothing);
cv2.createTrackbar('S min', 'binalized',  76, 255, nothing);
cv2.createTrackbar('V max', 'binalized', 199, 255, nothing);
cv2.createTrackbar('V min', 'binalized',  31, 255, nothing);

image = vs.read()
if not printedShape:
    print(threading.active_count())
    print(frame.shape)
    printedShape=True
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
maxH = cv2.getTrackbarPos('H max', 'binalized')
minH = cv2.getTrackbarPos('H min', 'binalized')
maxS = cv2.getTrackbarPos('S max', 'binalized')
minS = cv2.getTrackbarPos('S min', 'binalized')
maxV = cv2.getTrackbarPos('V max', 'binalized')
minV = cv2.getTrackbarPos('V min', 'binalized')
cv2.imshow('binalized',mask)
"""
        This function manages arming and taking off of the drone
"""
def arm_and_takeoff(aTargetAltitude):
    """
    Arms vehicle and fly to aTargetAltitude with GPS data.
    """
    ##### CONSTANTS #####
    DEFAULT_TAKEOFF_THRUST = 0.7
    SMOOTH_TAKEOFF_THRUST = 0.6
    print("Basic pre-arm checks")
```

```
# Don't let the user try to arm until autopilot is ready
# If you need to disable the arming check,
# just comment it with your own responsibility.
while not vehicle.is_armable:
    print(" Waiting for vehicle to initialise...")
    time.sleep(1)

print("Arming motors")
# Copter should arm in GUIDED mode
vehicle.mode = VehicleMode("GUIDED")
vehicle.armed = True

while not vehicle.armed:
    print(" Waiting for arming...")
    vehicle.armed = True
    time.sleep(1)

print("Taking off!")
thrust = DEFAULT_TAKEOFF_THRUST

while True:
    current_altitude = vehicle.location.global_relative_frame.alt
    print(" Altitude: %f  Desired: %f" % (current_altitude, aTargetAltitude))

    if current_altitude >= aTargetAltitude*0.95: # Trigger just below target alt.
        print("Reached target altitude")
        break
    elif current_altitude >= aTargetAltitude*0.6:
        thrust = SMOOTH_TAKEOFF_THRUST
    set_attitude(thrust = thrust)
    time.sleep(0.2)
```

```
"""
        This function encodes into a Dronekit message our order
"""
def send_attitude_target(roll_angle = 0.0, pitch_angle = 0.0,
                         yaw_angle = None, yaw_rate = 0.0, use_yaw_rate = False,
                         thrust = 0.5):
    """
    use_yaw_rate: the yaw can be controlled using yaw_angle OR yaw_rate.
                  When one is used, the other is ignored by Ardupilot.
    thrust: 0 <= thrust <= 1, as a fraction of maximum vertical thrust.
        Note that as of Copter 3.5, thrust = 0.5 triggers a special case in
        the code for maintaining current altitude.
    """
    if yaw_angle is None:
        # this value may be unused by the vehicle, depending on use_yaw_rate
        yaw_angle = vehicle.attitude.yaw
    # Thrust >  0.5: Ascend
    # Thrust == 0.5: Hold the altitude
    # Thrust <  0.5: Descend
    msg = vehicle.message_factory.set_attitude_target_encode(
        0, # time_boot_ms
        1, # Target system
        1, # Target component
        0b00000000 if use_yaw_rate else 0b00000100,
        to_quaternion(roll_angle, pitch_angle, yaw_angle), # Quaternion
        0, # Body roll rate in radian
        0, # Body pitch rate in radian
        math.radians(yaw_rate), # Body yaw rate in radian/second
        thrust  # Thrust
    )
    vehicle.send_mavlink(msg)
```

```
"""
        This function reflects all our parameters to another function to
        encode them for sending into a message to the drone.

        There are two functions for the same purpose due to before,
        in this function, there were more actions which were affected
        by duration parameter.
"""
def set_attitude(roll_angle = 0.0, pitch_angle = 0.0,
                 yaw_angle = None, yaw_rate = 0.0, use_yaw_rate = False,
                 thrust = 0.5, duration = 0):

    send_attitude_target(roll_angle, pitch_angle,
                         yaw_angle, yaw_rate, False,
                         thrust)
def to_quaternion(roll = 0.0, pitch = 0.0, yaw = 0.0):
    """
    Convert degrees to quaternions
    """
    t0 = math.cos(math.radians(yaw * 0.5))
    t1 = math.sin(math.radians(yaw * 0.5))
    t2 = math.cos(math.radians(roll * 0.5))
    t3 = math.sin(math.radians(roll * 0.5))
    t4 = math.cos(math.radians(pitch * 0.5))
    t5 = math.sin(math.radians(pitch * 0.5))

    w = t0 * t2 * t4 + t1 * t3 * t5
    x = t0 * t3 * t4 - t1 * t2 * t5
    y = t0 * t2 * t5 + t1 * t3 * t4
    z = t1 * t2 * t4 - t0 * t3 * t5

    return [w, x, y, z]
```

```
"""
        This function reads from camera and obtains
        the distance between drone and its target
        using camera resolution
"""
def camera_work() :
        image = vs.read()
        hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

        # Building HSV Trackbars
        maxH = cv2.getTrackbarPos('H max', 'binalized')
        minH = cv2.getTrackbarPos('H min', 'binalized')
        maxS = cv2.getTrackbarPos('S max', 'binalized')
        minS = cv2.getTrackbarPos('S min', 'binalized')
        maxV = cv2.getTrackbarPos('V max', 'binalized')
        minV = cv2.getTrackbarPos('V min', 'binalized')
        # Define upper and lower colours.
        lower_color = np.array([minH, minS, minV])
        upper_color = np.array([maxH, maxS, maxV])
        # Threshold the HSV image to get only the selected color
        mask = cv2.inRange(hsv, lower_color, upper_color)

        cv2.imshow('binalized',mask)
        cnts = None

        # find contours in the mask and initialize the current (x, y) center of the ball
        cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)[-2]
        center = None
        center_x = None
        center_y = None

        x_offset = 0
        y_offset = 0
```

```
        # only proceed if at least one contour was found
        if len(cnts) > 0:
            # find the largest contour in the mask, then use it to compute the minimum enclosing circle and centroid
            c = max(cnts, key=cv2.contourArea)
            ((x, y), radius) = cv2.minEnclosingCircle(c)
            # Calculate largest contour moment
            M = cv2.moments(c)
            # only proceed if the radius meets a minimum size draw the circle and
            # centroid on the frame, then update the list of tracked points
            # and calculate the distance between drone and point
            if M["m00"] != 0 and radius > 10 :
                # Calculate each coordinate of the moment center point
                center_x = int(M["m10"] / M["m00"])
                center_y = int(M["m01"] / M["m00"])
                # Build center point
                center = ( center_x, center_y )

                # Draw a circle
                cv2.circle(image, center, 5, (0, 0, 255), -1)

                print("Center at: ({0:3d}, {1:3d}) with radius {2:3.2f}\n".format(center_x, center_y, radius) )

                # Update target position using camera resolution
                pos_dest[0] = center_x - res_h / 2
                pos_dest[1] = - (center_y - res_v / 2)

                # Calculate offset between the center of the sceen and the target
                x_offset = center_x - res_h / 2
                y_offset = - (center_y - res_v / 2)

        cv2.imshow("Frame", image)
        return x_offset, y_offset
```

```
min_thrust =   0.1
max_thrust =  1
Kp_Thrust  =  1.5


"""
        This function checks what is the height of the drone and check it with desired height. If
        it is correct 0.5 is returned for maintaining actual height but if it is not correct it
        will calculate according to a proportion if it must be higher or lower so the returned value
        will be a value between 0 and 1 being lower than 0.5 to decrease and higher for increase.
"""
def check_height() :


        # We will allow a hysteresis error = +/-0.05
        # So, if we calculate a quotient, it must be 1 to be equal to our desired height
        # So, with a hysteresis error = +/-0.05 our quotient must below to the range [0.95, 1.05]


        # To sum up, it is the same that vehicle.location.global_relative_frame.alt <= pos_offset[2] * 1.05
        #             and the same that vehicle.location.global_relative_frame.alt >= pos_offset[2] * 0.95


        thrust = 0.5


        # Quotient between relative actual height and the desired height which is stored in pos_offset[2]
        # Being:
        #         pos_offset[X][Y][Z]
        #         pos_offset[0][1][2]
        quotient = vehicle.location.global_relative_frame.alt / pos_offset[2]


        print("Quotient = " + str(quotient) + "\n" )
```

```
        # Apply hysteresis error
        if ( quotient > 1.05 ) :
                to_substract = ( 0.1 * min(4, quotient) )
                thrust = 0.5 - to_substract + (0 if quotient >= 2 else (to_substract / 2) )
        elif ( quotient < 0.95 ) :
                to_add = (0.1 * min( 5, math.pow( quotient, -1 ) ) )
                thrust = 0.5 + to_add - (0 if quotient <= 0.7 else (to_add / 2) )
        else : # if( quotient >= ( pos_offset[2] * 0.95 ) and quotient <= ( pos_offset[2] *
1.05 ) )
                thrust = 0.5
        return thrust


def rotation_control( x_FO, y_FO, z_FO ) :
                #Get actual yaw angle
    my_yaw = math.degrees( vehicle.attitude.yaw )
                # Calculate desired yaw angle
    desired_yaw = math.degrees( math.atan2( x_FO, y_FO ) ) + my_yaw
                # Check height for making sure drone is in the desired height
    error_thrust = check_height()
                # Set the attitude corrector
    set_attitude(yaw_angle = desired_yaw, thrust = error_thrust)
                # Report all needed info
    print("\nMy yaw: {0:5.2f}\t Desired_yaw: {1:3.3f}".format(my_yaw, desired_yaw))
    print("Error X: {0:4.2f}\t Error Y: {0:4.2f}".format(x_FO, y_FO))
    print("Error Z: {0:4.2f}\t Error_thrust = {1:3.3f}\n".format(z_FO, error_thrust ) )
    print("#"*50)


# Take off 2.5m in GUIDED_NOGPS mode.
arm_and_takeoff(5)
Kp_Pitch =    -0.1    # It is negative because if we want to go ahead, we will need
negative value for pointing nose down and if we need go back, we will need positive
value for pointing nose up.
Kp_Roll  =    0.1
max_pitch =   3
max_roll  = 3
```

```
time.sleep(2)


x_FO =          1.0
y_FO =          1.0
z_FO =          1.0
while True :
                # Taking values which will be needed from camera
        x_FO, y_FO = camera_work()
        z_FO = vehicle.location.global_relative_frame.alt - pos_offset[2]
                # Do rotation control about the drone
        rotation_control( x_FO, y_FO, z_FO )
                # If user press "Esc" key execution will be broken
        k = cv2.waitKey(5) & 0xFF
        if k == 27:
                break
print("Setting LAND mode...")
vehicle.mode = VehicleMode("LAND")
time.sleep(1)

cv2.destroyAllWindows()
vs.stop()
# Close vehicle object before exiting script
print("Close vehicle object")
vehicle.close()
# Shut down simulator if it was started.
if sitl is not None:
    sitl.stop()
print("Completed")
```

11.3.4. persecution.py

```
#!/usr/bin/env python

"""
persecution.py: (Copter Only)
        -       Based on set_attitude_target.py
This example shows how to move/direct Copter and send commands
 in GUIDED mode using DroneKit Python.

Caution: A lot of unexpected behaviors may occur in GUIDED_NOGPS mode.
    Always watch the drone movement, and make sure that you are in dangerless environment.
    Land the drone as soon as possible when it shows any unexpected behavior.

Tested in Python 2.7.10
"""
#####################################
# IMPORTS SET_ATTRIBUTE_TARGET #
#####################################
from dronekit import connect, VehicleMode, LocationGlobal, LocationGlobalRelative
from pymavlink import mavutil # Needed for command message definitions
import time
import math

####################
# IMPORTS CAMERA #
####################
from imutils.video import VideoStream
import imutils
import cv2
import numpy as np
import threading
import glob as g
import os
```

```
# Set up option parsing to get connection string
import argparse
parser = argparse.ArgumentParser(description='Control Copter and send commands in GUIDED mode ')
parser.add_argument('--connect',
                    help="Vehicle connection target string. If not specified, SITL automatically started and used.")
args = parser.parse_args()

connection_string = args.connect
sitl = None

# Start SITL if no connection string specified
if not connection_string:
    import dronekit_sitl
    sitl = dronekit_sitl.start_default()
    connection_string = sitl.connection_string()

# Connect to the Vehicle
print('Connecting to vehicle on: %s' % connection_string)
vehicle = connect(connection_string, wait_ready=True)
########################################
#                VIDEO                 #
########################################
def nothing(x):
    pass

res_h = 640
res_v = 480

x_dest =  0
y_dest =  0
z_dest =  5
pos_offset = [ x_dest, y_dest, z_dest ]
resolution=(res_v, res_h)
```

```
fps = 60
printedShape=False


dev = -1
for i in g.glob("/dev/video?"):
        print("Analizing " + i)
        resVid = os.popen("udevadm info -a -n " + i + " | grep \'{idVendor}\' | head -n1 | awk -F \'\"\' '{print $2}\'").read()[:-1]
        resPid = os.popen("udevadm info -a -n " + i + " | grep \'{idProduct}\' | head -n1 | awk -F \'\"\' '{print $2}\'").read()[:-1]
        print("resVid = " +resVid)
        print("resPid = " +resPid)
        if resVid == "1f4d" and resPid == "0102" :
                print("This!")
                dev = int(i[-1])
                print("Selected src = " + str(dev))
                break   # Exit because we have found it
        else :
                print("This not.") # Discard this input device
if dev == -1 :
        print("Any match. Aborting...")
        quit(); # Abort this execution due to we haven't  found it
vs = VideoStream(src=dev, resolution=resolution, framerate=fps).start()


time.sleep(1.0)
rows,cols=frame.shape[:2]


# Thresholds
minH = 0
maxH = 255
minS = 0
maxS = 255
minV = 0
maxV = 255
```

```
# Create a window
cv2.namedWindow('binalized');
cv2.createTrackbar('H max', 'binalized', 175, 255, nothing);
cv2.createTrackbar('H min', 'binalized', 131, 255, nothing);
cv2.createTrackbar('S max', 'binalized', 145, 255, nothing);
cv2.createTrackbar('S min', 'binalized',  76, 255, nothing);
cv2.createTrackbar('V max', 'binalized', 199, 255, nothing);
cv2.createTrackbar('V min', 'binalized',  31, 255, nothing);

image = vs.read()

if not printedShape:
    print(threading.active_count())
    print(frame.shape)
    printedShape=True

hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
maxH = cv2.getTrackbarPos('H max', 'binalized')
minH = cv2.getTrackbarPos('H min', 'binalized')
maxS = cv2.getTrackbarPos('S max', 'binalized')
minS = cv2.getTrackbarPos('S min', 'binalized')
maxV = cv2.getTrackbarPos('V max', 'binalized')
minV = cv2.getTrackbarPos('V min', 'binalized')

cv2.imshow('binalized',mask)
```

```
"""
        This function manages arming and taking off of the drone
"""
def arm_and_takeoff(aTargetAltitude):
    """
        Arms vehicle and fly to aTargetAltitude with GPS data.
    """
    ##### CONSTANTS #####
    DEFAULT_TAKEOFF_THRUST = 0.7
    SMOOTH_TAKEOFF_THRUST = 0.6

    print("Basic pre-arm checks")

    # Don't let the user try to arm until autopilot is ready
    # If you need to disable the arming check,
    # just comment it with your own responsibility.
    while not vehicle.is_armable:
        print(" Waiting for vehicle to initialise...")
        time.sleep(1)

    print("Arming motors")
    # Copter should arm in GUIDED mode
    vehicle.mode = VehicleMode("GUIDED")
    vehicle.armed = True

    while not vehicle.armed:
        print(" Waiting for arming...")
        vehicle.armed = True
        time.sleep(1)

    print("Taking off!")

    thrust = DEFAULT_TAKEOFF_THRUST
```

```
    while True:
        current_altitude = vehicle.location.global_relative_frame.alt
        print(" Altitude: %f  Desired: %f" %
              (current_altitude, aTargetAltitude))
        if current_altitude >= aTargetAltitude*0.95: # Trigger just below target alt.
            print("Reached target altitude")
            break
        elif current_altitude >= aTargetAltitude*0.6:
            thrust = SMOOTH_TAKEOFF_THRUST
        set_attitude(thrust = thrust)
        time.sleep(0.2)


"""
        This function encodes into a Dronekit message our order
"""
def send_attitude_target(roll_angle = 0.0, pitch_angle = 0.0,
                         yaw_angle = None, yaw_rate = 0.0, use_yaw_rate = False,
                         thrust = 0.5):
    """
    use_yaw_rate: the yaw can be controlled using yaw_angle OR yaw_rate.
                  When one is used, the other is ignored by Ardupilot.
    thrust: 0 <= thrust <= 1, as a fraction of maximum vertical thrust.
            Note that as of Copter 3.5, thrust = 0.5 triggers a special case in
            the code for maintaining current altitude.
    """


    if yaw_angle is None:
        # this value may be unused by the vehicle, depending on use_yaw_rate
        yaw_angle = vehicle.attitude.yaw


    # Thrust >  0.5: Ascend
    # Thrust == 0.5: Hold the altitude
    # Thrust <  0.5: Descend
```

```
    msg = vehicle.message_factory.set_attitude_target_encode(
        0, # time_boot_ms
        1, # Target system
        1, # Target component
        0b00000000 if use_yaw_rate else 0b00000100,
        to_quaternion(roll_angle, pitch_angle, yaw_angle), # Quaternion
        0, # Body roll rate in radian
        0, # Body pitch rate in radian
        math.radians(yaw_rate), # Body yaw rate in radian/second
        thrust  # Thrust
    )
    vehicle.send_mavlink(msg)

"""
        This function reflects all our parameters to another function to
        encode them for sending into a message to the drone.

        There are two functions for the same purpose due to before, in this function,
        there were more actions which were affected by duration parameter.
"""
def set_attitude(roll_angle = 0.0, pitch_angle = 0.0, yaw_angle = None, yaw_rate = 0.0,
                 use_yaw_rate = False, thrust = 0.5, duration = 0):
    """
    Note that from AC3.3 the message should be re-sent more often than every
    second, as an ATTITUDE_TARGET order has a timeout of 1s.
    In AC3.2.1 and earlier the specified attitude persists until it is canceled.
    The code below should work on either version.
    Sending the message multiple times is the recommended way.
    """
    send_attitude_target(roll_angle, pitch_angle,
                         yaw_angle, yaw_rate, False,
                         thrust)
```

```
def to_quaternion(roll = 0.0, pitch = 0.0, yaw = 0.0):
    """
    Convert degrees to quaternions
    """
    t0 = math.cos(math.radians(yaw * 0.5))
    t1 = math.sin(math.radians(yaw * 0.5))
    t2 = math.cos(math.radians(roll * 0.5))
    t3 = math.sin(math.radians(roll * 0.5))
    t4 = math.cos(math.radians(pitch * 0.5))
    t5 = math.sin(math.radians(pitch * 0.5))

    w = t0 * t2 * t4 + t1 * t3 * t5
    x = t0 * t3 * t4 - t1 * t2 * t5
    y = t0 * t2 * t5 + t1 * t3 * t4
    z = t1 * t2 * t4 - t0 * t3 * t5

    return [w, x, y, z]

"""
        This function reads from camera and obtains
        the distance between drone and its target
        using camera resolution
"""
def camera_work() :
        image = vs.read()
        hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

        # Building HSV Trackbars
        maxH = cv2.getTrackbarPos('H max', 'binalized')
        minH = cv2.getTrackbarPos('H min', 'binalized')
        maxS = cv2.getTrackbarPos('S max', 'binalized')
        minS = cv2.getTrackbarPos('S min', 'binalized')
        maxV = cv2.getTrackbarPos('V max', 'binalized')
        minV = cv2.getTrackbarPos('V min', 'binalized')
```

```
        # Define upper and lower colours.
        lower_color = np.array([minH, minS, minV])
        upper_color = np.array([maxH, maxS, maxV])
        # Threshold the HSV image to get only the selected color
        mask = cv2.inRange(hsv, lower_color, upper_color)

        cv2.imshow('binalized',mask)

        cnts = None

        # find contours in the mask and initialize the current (x, y) center of the ball
        cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[-2]
        center = None
        center_x = None
        center_y = None

        x_offset = 0
        y_offset = 0

        # only proceed if at least one contour was found
        if len(cnts) > 0:
                # find the largest contour in the mask, then use it to compute the minimum enclosing circle and centroid
                c = max(cnts, key=cv2.contourArea)
                ((x, y), radius) = cv2.minEnclosingCircle(c)
                # Calculate largest contour moment
                M = cv2.moments(c)
                # only proceed if the radius meets a minimum size
                # draw the circle and centroid on the frame,
                # then update the list of tracked points
                # and calculate the distance between drone and point
                if M["m00"] != 0 and radius > 10 :
```

```
            # Calculate each coordinate of the moment center point
            center_x = int(M["m10"] / M["m00"])
            center_y = int(M["m01"] / M["m00"])

            # Build center point
            center = ( center_x, center_y )

            # Draw a circle
            cv2.circle(image, center, 5, (0, 0, 255), -1)

            print("Center at: ("+ str(center_x) + ", " + str(center_y) + ")\n")

            # Update target position using camera resolution
            pos_dest[0] = center_x - res_h / 2
            pos_dest[1] = - (center_y - res_v / 2)

            # Calculate offset between the center of the sceen and the target
            x_offset = center_x - res_h / 2
            y_offset = - (center_y - res_v / 2)

        # Show this frame
        cv2.imshow("Frame", image)
        # Return distance
        return x_offset, y_offset

min_thrust = 0.1
max_thrust = 1
Kp_Thrust = 1.5
```

```
"""
        This function checks what is the height of the drone and check it with desired
        height. If it is correct 0.5 is returned for maintaining actual height but if it is not
        correct it will calculate according to a proportion if it must be higher or lower so
        the returned value will be a value between 0 and 1 being lower than 0.5 to
        decrease and higher for increase.
"""
def check_height() :
        # We will allow a hysteresis error = +/-0.02. So, if we calculate a quotient, it
        # must be 1 to be equal to our desired height. So, with a hysteresis = +/-0.05
        # our quotient must below to the range [0.95, 1.05]. To sum up, it is the same
        # that vehicle.location.global_relative_frame.alt <= pos_offset[2] * 1.02 and the
        # same that vehicle.location.global_relative_frame.alt >= pos_offset[2] * 0.98


        thrust = 0.5
        # Quotient between relative actual height and the desired height which is stored
in pos_offset[2]
        # Being:
        #       pos_offset[X][Y][Z]
        #       pos_offset[0][1][2]
        quotient = vehicle.location.global_relative_frame.alt / pos_offset[2]
        print("Quotient = " + str(quotient) + "\n" )


        # Apply hysteresis error
        if ( quotient > 1.05 ) :
                to_substract = ( 0.1 * min(4, quotient) )
                thrust = 0.5 - to_substract + (0 if quotient >= 2 else (to_substract / 2) )
        elif ( quotient < 0.95 ) :
                to_add = (0.1 * min( 5, math.pow( quotient, -1 ) ) )
                thrust = 0.5 + to_add - (0 if quotient <= 0.9 else (to_add / 2) )
        else : # if( quotient >= ( pos_offset[2] * 0.95 ) and quotient <= ( pos_offset[2] *
1.05 ) )
                thrust = 0.5
        # Return definitive thrust
        return thrust
```

```
"""
        This function does all attitude work calculating
        orientation, pitch to do and roll to do. After this,
        report all this info
"""
def complet_control( x_FO, y_FO, z_FO ) :

        # Take actual yaw angle
        my_yaw = math.degrees( vehicle.attitude.yaw )

        # Calculate the desired yaw according to actual orientation and target position
        desired_yaw = my_yaw + math.degrees( math.atan2( x_FO, y_FO ) )

        # Calculate errors to apply
        error_pitch = Kp_Pitch * y_FO
        error_roll  = Kp_Roll  * x_FO
        # If error_pitch is higher than maximum pitch value allowed,
        #       it will be forced to be max_pitch value with its correct sign
        if( math.fabs(error_pitch) > max_pitch ) :
                error_pitch = math.copysign(max_pitch, error_pitch)

        # If error_roll is higher than maximum roll value allowed,
        #       it will be forced to be max_roll value with its correct sign

        if( math.fabs(error_roll) > max_roll ) :
                error_roll = math.copysign(max_roll, error_roll)

        # Check height for making sure drone is in the desired height
        error_thrust = check_height()

        # Set the attitude corrector
        set_attitude(pitch_angle = error_pitch, roll_angle = error_roll, yaw_angle = desired_yaw, thrust = error_thrust)
```

```
        # Report all this info
        print("\nMy yaw: {0:5.2f}\t Desired_yaw: {1:3.3f}".format(my_yaw, desired_yaw))
        print("Error X: {0:4.2f}\t Error_roll:  {1:3.3f}".format(x_FO, error_roll))
        print("Error Y: {0:4.2f}\t Error_pitch: {1:3.3f}".format(y_FO, error_pitch))
        print("Error Z: {0:4.2f}\t Error_thrust = {1:3.3f}\n".format(z_FO, error_thrust ) )

        print("#"*50)

# Take off 5m in GUIDED mode.
arm_and_takeoff(5)

a = 0
Kp_Pitch =    -0.1    # It is negative because if we want to go ahead, we will need negative value for pointing nose down and if we need go back, we will need positive value for pointing nose up.
Kp_Roll        =       0.1

max_pitch = 3
max_roll  = 3

time.sleep(2)

x_FO = 1.0
y_FO = 1.0
z_FO = 1.0

while True :
        # Taking values which will be needed from camera
        x_FO, y_FO = camera_work()
        z_FO = vehicle.location.global_relative_frame.alt - pos_offset[2]

        # Do complet control about the drone
        complet_control( x_FO, y_FO, z_FO )
```

```
            # If user press "Esc" key execution will be broken
            k = cv2.waitKey(5) & 0xFF
            if k == 27:
                    break
print("Setting LAND mode...")
vehicle.mode = VehicleMode("LAND")
time.sleep(1)

cv2.destroyAllWindows()
vs.stop()

# Close vehicle object before exiting script
print("Close vehicle object")
vehicle.close()

# Shut down simulator if it was started.
if sitl is not None:
    sitl.stop()
print("Completed")
```