

TFG-Teleco

pedro.arias

December 2019

Resumen

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Capítulo 1

Introducción

Este primer capítulo recoge lo esencial del trabajo. En él, se explica el contexto en el cual se enmarca el estudio, el problema a solucionar, la motivación del mismo y los objetivos extraídos del problema. Además, se presenta también cual será la estructura seguida en la memoria.

1.1. Contexto

El término *robot* aparece por primera vez en 1920, en la obra teatral *Rossum's Universal Robots* del escritor checo Karel Čapek en cuyo idioma la palabra “robota” significa fuerza o servidumbre [40]. El nacimiento de la robótica y los robots surge asociado a la idea de trabajo y producción tras la Segunda Revolución Industrial y a lo largo del siglo XX. La automatización industrial de aquella época da lugar a los primeros sistemas de control automático que se extienden rápidamente a todos los sectores industriales, y que dan lugar a la robótica industrial tal como la conocemos hoy en día [4].

El término *robótica* es acuñado por Isaac Asimov, definiendo a la ciencia que estudia a los robots. El propio Asimov postuló también las Tres Leyes de la Robótica en su libro *Yo Robot* publicado en 1950 [40]. El término ha evolucionado mucho desde sus inicios, hoy entendemos la robótica como la ciencia o rama de la tecnología que estudia el diseño y construcción de máquinas capaces de desempeñar tareas realizadas por el ser humano o que requieren el uso de inteligencia [24]. Como se puede entender, la visión actual es mucho más amplia que en sus inicios y abarca muchas áreas de la ingeniería.

Existen diversas clasificaciones en función de su arquitectura, de su aplicación, de su cronología, etc. Una de estas clasificaciones distingue robots en función de su morfología [11], que suele distinguir los siguientes tipos:

- **Poliarticulados:** Son artilugios mecánicos y electrónicos destinados a realizar de forma automática determinados procesos de fabricación o manipulación. Suelen ser fijos, aunque también pueden realizar desplazamientos limitados y poseen un espacio de trabajo concreto y limitado. Los mejores ejemplos son los robots industriales, manipuladores o cartesianos.
- **Móviles:** Están provistos de algún tipo de mecanismo que les permite desplazarse autónomamente, como patas o ruedas, y reciben información de su entorno a través de sus propios sensores. Son ampliamente utilizados en el transporte de mercancías o en la exploración de lugares de difícil acceso. Pueden ser terrestres, acuáticos, aéreos o espaciales.

- **Androides:** Intentan reproducir total o parcialmente la forma y el comportamiento del ser humano. No solo imitan la apariencia humana (antropomorfismo), si no que emulan también la conducta de forma autónoma.
- **Zoomórficos:** Son aquellos que trata de reproducir en mayor o menor grado de realismo los sistemas de locomoción de diversos seres vivos. Este tipo podría incluir también a la morfología androide, en función del autor de la clasificación. Una subclasificación distingue entre robots caminadores y no caminadores.

Este trabajo se enmarca dentro de la robótica móvil, y más en concreto, dentro de la robótica aérea. La robótica aérea es la rama de la robótica que se encarga del estudio del comportamiento autónomo de aeronaves no tripuladas. Se entiende como una aeronave no tripulada (UAV, *Unmanned Aerial Vehicle*, o más recientemente UAVS, *Unmanned Aircraft Vehicle System*) a aquella que es capaz de realizar una misión sin necesidad de tener una tripulación embarcada [3].

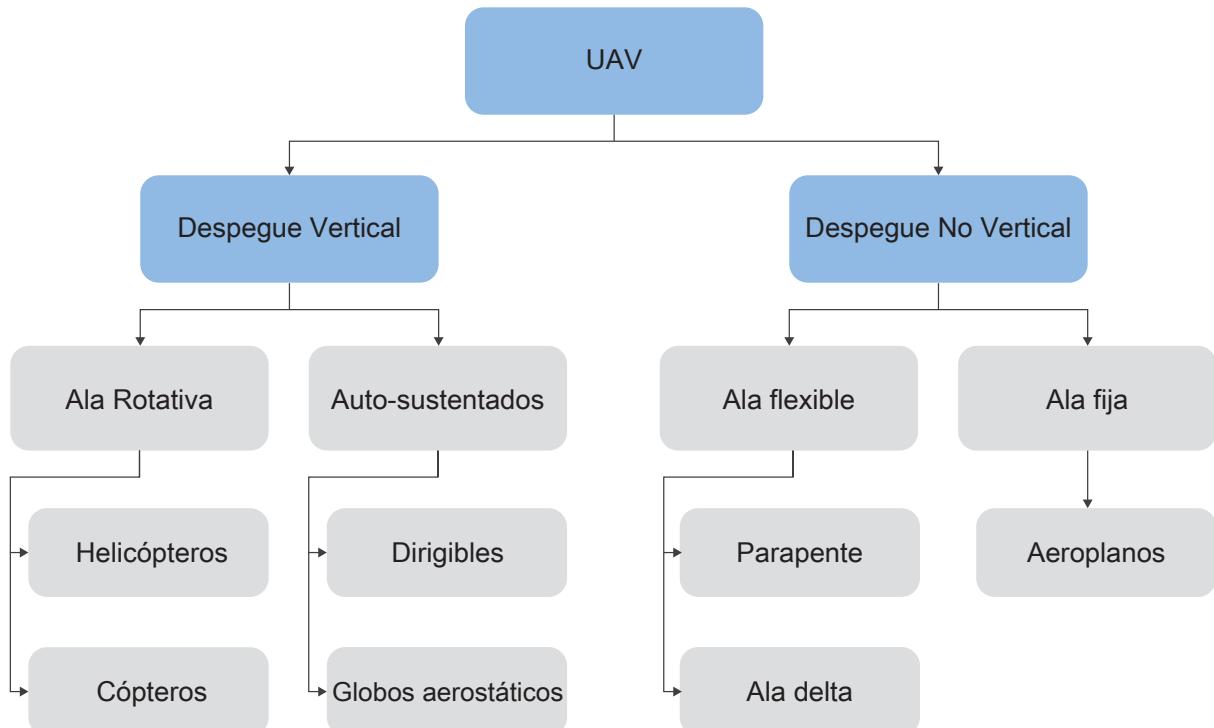


Figura 1.1: Clasificación de los UAV [3].

A la hora de establecer una clasificación de los UAV es posible atender a diferentes criterios. Siguiendo la clasificación propuesta por *Barrientos et al.* [3] se distinguen aeronaves en función del tipo de despegue, que puede ser vertical o no. A su vez, podemos subdividir las aeronaves en función del origen de su sustentación o del tipo de ala que poseen. Esta clasificación se representa en la Figura 1.1, mientras que en la Figura 1.2 se muestran ejemplos de los diferentes tipos de UAV.



Figura 1.2: Tipos de UAV.

Otros criterios de clasificación pueden responder a las capacidades de vuelo como el alcance, la altitud, la autonomía o la carga máxima. A su vez, también se clasificar las aeronaves en función de la actividad que realizan. Las principales aplicaciones se recogen en la siguiente lista:

- Militar: de apoyo, de combate, de reconocimiento, etc.
- Transporte, tanto de mercancías o de personas.
- Seguridad, vigilancia y salvamento.
- Ocio y entretenimiento: en cine, en deporte, etc.
- Educación e investigación.
- Agropecuario: fumigación, control de recursos, etc.
- Cartografía, topografía y fotografía aérea.

La siguiente figura (Fig. 1.3) recoge alguna de las principales aplicaciones mencionadas.

En la actualidad tiende a utilizarse el concepto de UAVS frente al de UAV. La extensión del concepto de vehículo a sistema refleja que el vehículo aéreo autónomo precisa para su funcionamiento de todo un sistema y no solo de la aeronave instrumentada. En concreto la instrumentación embarcada o segmento aire, debe verse complementada con la estación base o segmento tierra, debiéndose considerar las funcionalidades y características de ambos segmentos.



(a) Transporte de mercancías [5].



(b) Spot publicitario [44].



(c) Agricultura [44].



(d) Salvamento [5].

Figura 1.3: Principales aplicaciones en robótica aérea.

1.2. Problema y motivación

Este trabajo se centra en el diseño e implementación del primer prototipo de un software tipo *estación de tierra* que permita la operación automática remota de un Vehículo Aéreo no Tripulado (UAV, *Unmanned Aerial Vehicles*), a los que me dirigiré comúnmente como *drones* a lo largo de esta memoria.

Esta oportunidad surge de la colaboración entre la empresa madrileña CONYCA S.L. [47] y el grupo de investigación de robótica (@RoboticsLabURJC [12]) de la Universidad Rey Juan Carlos (URJC), con el proyecto *ROSpilot, software para operación y navegación de un UAV*, en la cual he tenido la suerte de ser invitado a colaborar.

Profundizando en la estación terrestre requerida, se busca automatizar procesos tediosos relacionados con labores de cartografía y toma de fotografía aérea. En este tipo de misiones es común utilizar drones de ala fija (tipo *Geodrone Mapper* [34], ver figura 1.4), lo cual agranda aún más el desafío propuesto debido a las limitaciones existentes en las operaciones de una aeronave de ala fija.

La inmensa mayoría de estaciones de tierra propone soluciones generales. Desde el inicio del proyecto se concibe un software íntimamente ligado a resolver y mitigar los problemas principales asociados a la cartografía y a la toma de fotografía aérea.



Figura 1.4: Geodrone Mapper, propiedad de CONYCA S.L., utilizado en misiones de cartografía y fotografía aérea.

En las siguientes secciones se describirá más en detalle el problema y como se afronta la solución del mismo.

1.3. Objetivos

Tras describir el problema a resolver en la sección anterior (Sección 1.2), en esta sección se quiere aclarar cuales son los objetivos perseguidos desde el inicio del proyecto. El objetivo principal es proporcionar una solución software que permita la operación automática remota a drones de ala fija. En concreto, la aplicación desarrollada será denominada como *UAVCommander* y parte del trabajo previo desarrollado por José Antonio Fernández Casillas en su proyecto fin de carrera *Navegación por posición para un avión autónomo con JdeRobot* [6].

En el proyecto acordado con la empresa se describen una serie de funcionalidades o requisitos que debe incluir la aplicación, los cuales se consideran objetivos secundarios:

- **Caracterización de la aeronave y carga de pago.** El usuario podrá configurar datos operativos de la aeronave y datos de la carga de pago, que consiste en una cámara con la que se tomarán las fotografías para realizar la cartografía.
- **Creación de misiones de vuelo automático.** El usuario podrá crear misiones de tipo multilínea como sucesión de puntos de paso (*waypoints*) o misiones de tipo superficie en las que la herramienta ha de planificar la trayectoria de barrido para cubrir toda la superficie. Para ambas misiones la referencia en la altitud podrá ser fija respecto al punto de despegue o variable según los diferentes puntos de paso.
- **Comprobación de pre-vuelo.** El usuario dispondrá de una lista de comprobaciones de seguridad previas al vuelo, entre las que se encuentran la calibración del tubo de Pitot, el estado de las baterías, etc.
- **Seguimiento del vuelo.** El usuario podrá visualizar la posición de la aeronave en todo momento acompañado de datos de telemetría y/o batería entre otros.

- **Análisis de datos post-vuelo.** El usuario podrá observar, descargar o eliminar los datos del log generado por el autopiloto de la aeronave.

Por último, se consideran otros objetivos ligados al enfoque cartográfico de la aplicación. Es lógico pensar que uno de los motivos por los que surge este proyecto es reducir y facilitar las tareas desempeñadas por el operario con el objetivo de una reducción sustancial en el tiempo empleado en este tipo de tareas. Además, otro objetivo que se persigue es la reducción o eliminación de errores que puedan surgir de una equivocación humana y que produzcan una repetición parcial o total de la misión.

1.4. Estructura de la memoria

En esta sección se describe la estructura de la memoria, se introducen los diferentes capítulos y los temas que se tratarán en cada uno de los mismos.

En primer lugar se encuentra un capítulo introductorio (Capítulo 1), en el cual se halla el lector al leer estas líneas, y que se concibe como una serie de aclaraciones previas y necesarias para el correcto entendimiento del trabajo en su conjunto. Incluye un contexto histórico de la robótica, la robótica aérea y las estaciones de tierra y sus aplicaciones, que establecen el marco en el que se sitúa este trabajo (Sección 1.1). A continuación incluye, el problema abordado en este trabajo y la motivación del mismo (Secc. 1.2), es decir, ¿qué se trata de resolver con este trabajo?, y ¿por qué surge la necesidad del mismo?. En la Sección 1.3 se incluyen los objetivos principales y secundarios que ha de cumplir la solución desarrollada. Por último, se encuentra la sección actual (Secc. 1.4) que muestra una visión global de lo que se presenta en esta memoria.

En segundo lugar, en el Capítulo 2 se detalla la infraestructura utilizada en la aplicación propuesta como solución al problema. Este capítulo recoge en diferentes secciones cada uno de los elementos que entran en juego. Por un lado, en la Sección 2.1 se explica el lado tierra y su composición. Por otro lado, en la Sección 2.2 se detalla el lado aire y sus componentes. Finalmente, en la Sección 2.3 se centra en el protocolo de comunicaciones entre ambos lados, tierra y aire.

En tercer lugar, en el Capítulo 3 se desarrolla el diseño y la implementación de la solución del problema. En una primera sección (Secc. 3.1) se explica detalladamente el diseño elegido y se explican también varias decisiones relevantes a la hora de seleccionar el diseño. En las secciones sucesivas (Seccs. 3.2, 3.3, 3.4 y 3.5) se describen la implementación de cada uno de los bloques de código identificados.

En el Capítulo 4, se presentan una serie de casos de uso que tratan de ejemplificar una posible situación real. Las diferentes secciones del capítulo corresponden cada una a un caso de estudio diferente. Los casos de uso que recoge esta memoria son la caracterización de la aeronave, la carga de mapas locales, la creación, carga y guardado de misiones, y el seguimiento de una misión.

Finalmente, en el Capítulo 5 se recogen las conclusiones extraídas con la finalización del trabajo y se evalúan los distintos objetivos iniciales propuestos (Secc. 5.1). A mayores, se exponen una serie de posibles vías de desarrollo futuro y de mejoras para la aplicación (Secc. 5.2).

Capítulo 2

Infraestructura

Durante este capítulo se explican las diferentes herramientas *software* y *hardware* que han servido de ingredientes en la realización de este trabajo. A la hora de proceder a desengranar los diferentes agentes que entran en acción, se realiza previamente una clasificación entre el lado tierra, el lado aire y la comunicación entre estos que se reflejan en las secciones de este capítulo.

En la Figura 2.1 se puede observar un esquema general que representa la arquitectura del sistema. El esquema muestra tanto el hardware como el software que integra el sistema, en los que se profundizará durante las secciones de este capítulo.

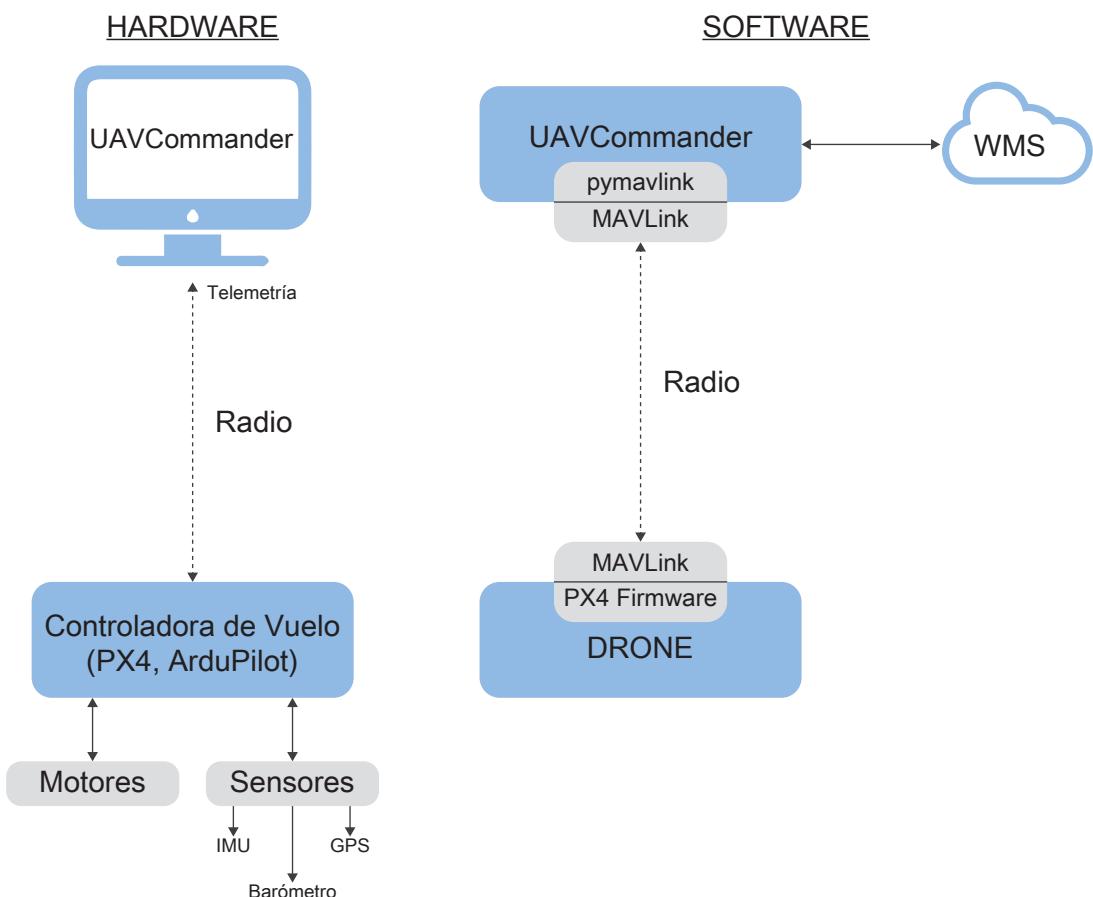


Figura 2.1: Esquema general del sistema.

2.1. Lado Tierra

El lado tierra se compone por un ordenador donde se ejecuta *UAVCommander*, la aplicación desarrollada. El software se idea como multiplataforma, por lo que el sistema operativo del ordenador puede ser cualquiera de las principales soluciones en el mercado. Sin embargo, la plataforma utilizada durante el desarrollo y las pruebas ha sido Ubuntu, la distribución de Linux basada en Debian. En concreto, la edición utilizada es la última versión con soporte de largo plazo, **Ubuntu 18.04.3 LTS (Bionic Beaver)** [39]. El motivo de esta elección es debido a que Ubuntu suele ser la primera opción en aplicaciones relacionadas con el software libre y la robótica.

El lenguaje elegido para el desarrollo de la aplicación ha sido Python [30]. El lenguaje creado a principios de 1990 por Guido van Rossum en los Países Bajos es un lenguaje de programación interpretado, interactivo y orientado a objetos. Sus principales ventajas que han motivado su elección para este proyecto son una sintaxis muy clara y su portabilidad. En concreto, la versión utilizada es **Python v3.6.9**.

Son varias las bibliotecas de Python utilizadas, entre ellas se quiere destacar **PyQt5** [37]. PyQt5 es un *binding* de la biblioteca gráfica Qt5 [9] para el lenguaje de programación Python. Qt es un framework multiplataforma escrito en C++ que permite el desarrollo de interfaces gráficas de usuario de forma sencilla.

Otra biblioteca relevante es **PyMavlink** [2], una implementación en Python del protocolo de comunicaciones MAVLink, el cual se explicará más adelante. El uso de esta librería facilita el uso del protocolo de comunicaciones, simplificando el uso de comandos y reduciendo el riesgo de cometer errores.

Para el manejo de mapas geo-referenciados se han utilizado diversas librerías. Un mapa geo-referenciado es aquel en el que conocemos o podemos calcular la posición real que representa cada píxel del mismo. Por un lado, se ha utilizado servicios de mapa web (WMS, *Web Map Services*) para obtener las imágenes como *Google Maps Platform* [38] u *Open Map Tiles* [36] a través de librerías como *urllib* para el manejo de URLs [33] o *PIL (Python Image Library)* y *Pillow*, para el manejo de imágenes [8].

Por otro lado, para la lectura de imágenes locales geo-referenciadas procedentes del Instituto Geográfico Nacional (IGN) [10] a través del Plan Nacional de Ortofotografía Aérea (PNOA) [41] se ha utilizado la librería **GDAL** [43]. GDAL es una biblioteca que permite leer más de 200 formatos de datos geoespaciales ráster y vectoriales, entre los cuales se encuentran formatos como GeoTIFF o ECW, utilizados por la aplicación.

Otras librerías utilizadas son *math* [29], *numpy* [13], *threading* [31], *json* [28], *collections* [26], *datetime* [27], *time* [32] o *qfi* [7].

A la hora de desarrollo se ha utilizado la plataforma GitHub [35] y el entorno de desarrollo PyCharm [48]. GitHub es plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. El código fuente del proyecto se encuentra en un repositorio privado debido a que está ligado a un proyecto de desarrollo con una empresa privada. De todas formas, en el siguiente capítulo se presentarán diferentes fragmentos de código mostrando algún aspecto de la implementación. En cambio, la memoria junto con cierta información sobre el trabajo se aloja en un repositorio público [45].

PyCharm es un entorno de desarrollo integrado (IDE, *Integrated Development Environment*), ampliamente utilizado con Python. Entre sus ventajas destacan su depurador, la refactorización, entre otros aspectos.

2.2. Lado Aire

El lado aire se compone por el drone. Se distinguen dos posibilidades, el drone puede ser real o simulado. Ambas posibilidades siguen un bucle de control similar. La Figura 2.2 trata de representar un bucle estándar de control.

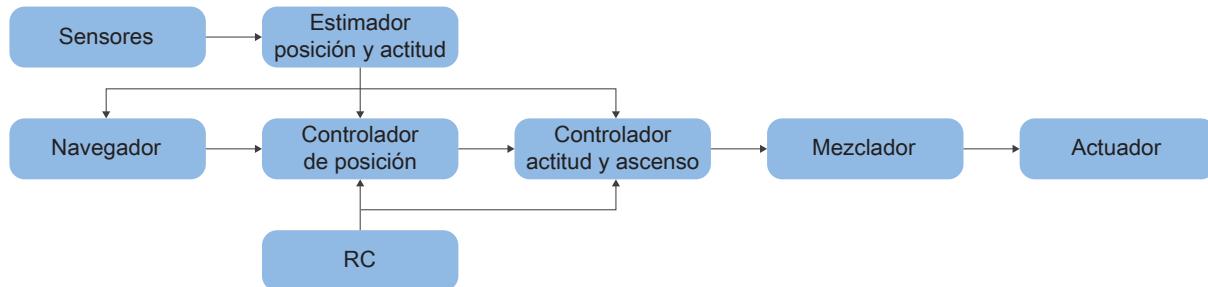


Figura 2.2: Bucle de control de un UAV [14].

El estimador toma una o más entradas de diferentes sensores, las combina y calcula el estado del vehículo. La controladora toma un punto de ajuste y una medición o estado estimado como entradas. Su objetivo es ajustar el valor del estado estimado de modo que coincida con el punto de ajuste. La salida es una corrección para eventualmente alcanzar ese punto de ajuste. Por ejemplo, el controlador de posición toma los puntos de ajuste de posición como entradas, y según la posición estimada calcula la salida que es un punto de ajuste de actitud y empuje que mueve el vehículo hacia la posición deseada. Finalmente, el mezclador toma comandos concretos, como girar a la derecha, y los traduce a comando de motor individuales, al tiempo que garantiza que no se excedan algunos límites. Esta traducción es específica para cada vehículo y depende de varios factores, como la disposición del motor con respecto al centro de gravedad o la inercia rotacional del vehículo.

La aeronave real puede ser de diversa índole, como ya se ha visto en el capítulo anterior. A lo largo de este proyecto se han utilizado las siguientes aeronaves:

- **3DR Solo Drone:** Cuadricóptero fabricado por 3DR Robotics [46] (Fig. 2.3a). Posee una controladora con firmware Ardupilot [49], una de las soluciones más usada en el mercado. Una de las principales ventajas que decidieron el uso de este drone en las primeras fases de vuelo frente a otros equipos del laboratorio es la existencia de una emisora (o mando de control), el cual permite controlar la aeronave ante cualquier problema que no este previsto en el software.
- **Geodrone training:** Ala fija propiedad de Conyca S.L [47] (Fig. 2.3b). Posee una controladora con firmware Ardupilot [49]. Al igual que el anterior, posee una emisora que aporta robustez y permite actuar ante cualquier imprevisto.

Sin embargo, el uso de drone real se ha limitado a las últimas fases de desarrollo, que muy frecuentemente se ha visto sustituido por un drone simulado (ver Fig. 2.4). Sobre el sistema propuesto en la sección anterior (Ubuntu 18.04.3) se ejecuta una simulación en software (**SITL**, *Software-In-The-Loop*) de una aeronave. El SITL permite enviar y recibir comandos a una controladora sin necesidad de tener el equipo real y evitando la pérdida de la aeronave en caso de error del programa.



(a) 3DR Solo Drone.



(b) Geodrone Training.

Figura 2.3: Aeronaves utilizadas durante el proyecto.

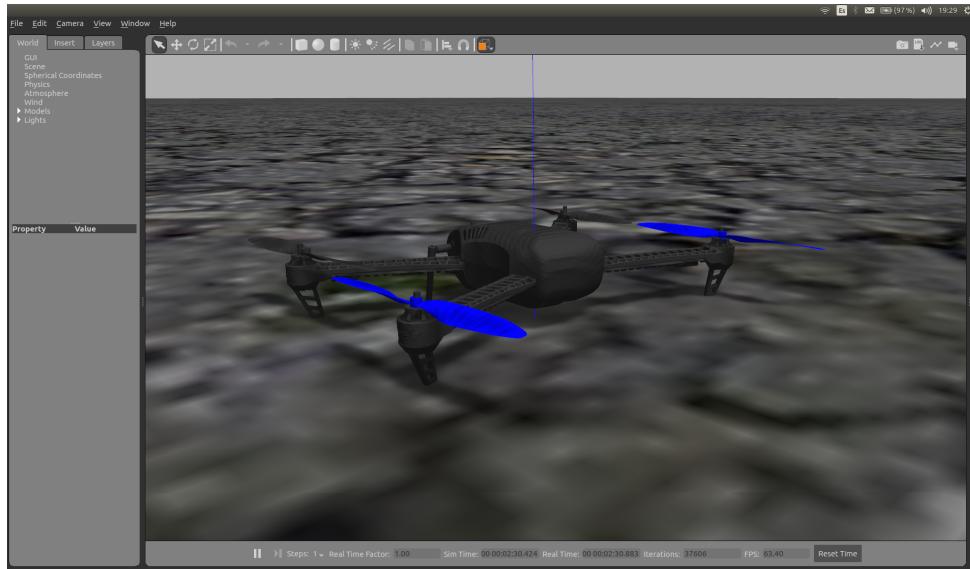


Figura 2.4: Cuacóptero simulado sobre Gazebo9.

El firmware utilizado como controladora es **PX4** [23], cuyo esquema de SITL se puede observar en la Figura 2.5. Además, la simulación se puede conectar con Gazebo [25] para hacer más completa la experiencia y mostrar el vuelo en un entorno de pruebas con elementos del mundo real. La versión utilizada durante el proyecto es **Gazebo9**.

2.3. Protocolo de comunicaciones

Tal como se ha adelantado en las secciones anteriores, el protocolo de comunicaciones utilizado es MAVLink [18]. MAVLink son las siglas de *Micro Air Vehicle Link*, un protocolo de comunicaciones muy ligero para el intercambio de mensajes con un drone y sus componentes a bordo. La versión utilizada es **MAVLink v2**.

Los extremos de la comunicación se han explicado en las secciones anteriores, siendo este protocolo el puente de comunicación entre el lado tierra y el lado aire. Es considerado como el protocolo estándar de comunicaciones en robótica aérea y las principales soluciones

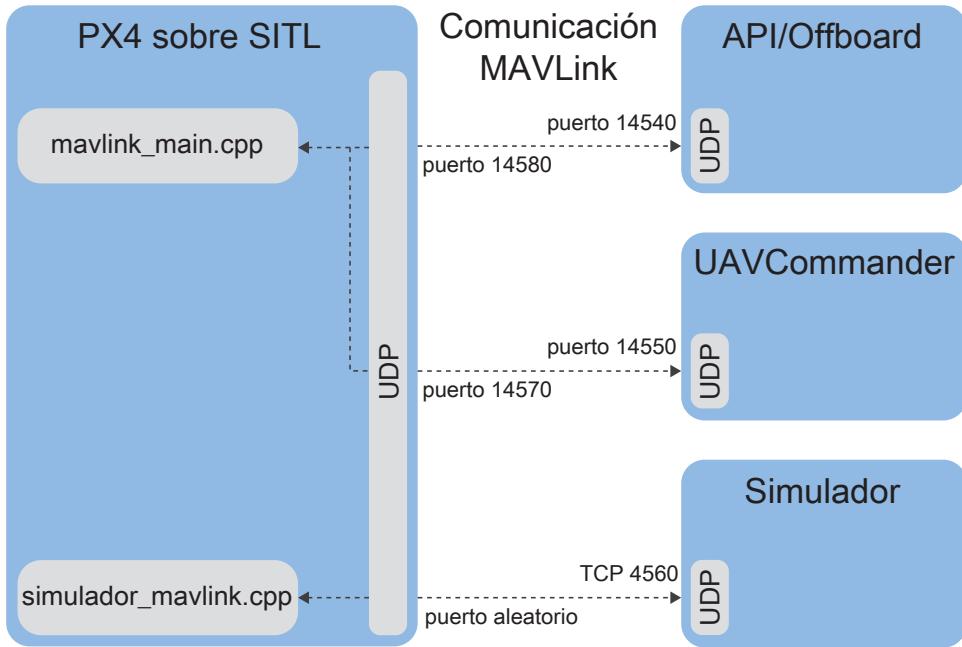


Figura 2.5: Esquema de PX4 sobre SITL [23].

comerciales lo utilizan. Existen diferentes tipos de mensajes, comandos (de navegación, de *perform*, etc.), enumerados y microservicios.

Capítulo 3

Estación de Tierra, diseño e implementación

Este capítulo aborda el diseño e implementación del software. En primer lugar se explicará el esquema seguido junto a las motivaciones de cada decisión de diseño tomada. Se presentarán los diferentes bloques en los que se organiza el código, que a su vez se desarrollarán en las secciones sucesivas.

3.1. Diseño

La aplicación ha sido diseñada desde el primer momento teniendo en cuenta los objetivos iniciales y requisitos del proyecto. A lo largo de esta sección, se harán múltiples referencias a los objetivos introducidos durante la Sección 1.3.

En la aplicación se integran múltiples funciones como la configuración en función de la aeronave, la programación de misiones, el seguimiento y cumplimiento de las mismas y la visualización de los sensores a bordo, entre otras funciones.

Existen diferentes formas de agrupar o segmentar la aplicación para un mejor análisis del código y una mejor comprensión del diseño realizado. En primer lugar, se presenta un diseño que agrupa la aplicación en bloques o módulos con funciones bien distinguidas entre sí. Se distinguen los siguientes grandes bloques:

- **Interfaz gráfica:** Está compuesto por una serie de ventanas, *widgets* y herramientas que dan soporte al resto de bloques y que permiten al usuario interactuar con la aplicación. La Sección 3.2 muestra el diseño e implementación seguida y explica más en detalle el bloque.
- **Mapas:** Se compone de una serie de clases que permiten la visualización e interacción con los mapas. Utiliza un sistema de mapeado basado en teselas que aligera la navegación a través del mapa. Dispone de diferentes fuentes cartográficas que permiten visualizar diferentes mapas sobre la misma herramienta. Durante la Sección 3.3 se detalla en profundidad aspectos relacionados con el diseño e implementación de este bloque.
- **Misiones:** Este bloque engloba las clases que intervienen en la creación de misiones. Se distinguen dos tipos de misión: por polilínea y por patrón. Más detalles y cuestiones relacionadas con el diseño e implementación se aportan en la Sección 3.4.

- **MAVLink Driver:** Este último bloque contiene toda vía de comunicación con la aeronave. Es el encargado de traducir las acciones lanzadas a través de la aplicación a mensajes MAVLink que la aeronave comprende. Este bloque es una evolución del código desarrollado por J.A. Fernández durante su proyecto fin de carrera [6]. Se aportan más detalles en la Sección 3.5 sobre el diseño e implementación del *driver*.

En la Figura 3.1 se observa como configuran la aplicación los diferentes bloques presentados. En la parte superior de la figura se encuentra el *front-end* de la aplicación, que está compuesto en su totalidad por la interfaz de usuario (GUI, *Graphical User Interface*). En una segunda capa se encuentra la parte lógica de la aplicación que se divide en distintos bloques, principalmente compuestos por el sistema de mapeado y el creador de misiones. En una tercera y última capa se encuentran una serie de servicios que aportan comunicación a la aplicación con el exterior. Se destacan dos servicios principales, los mapas web y el *driver* de MAVLink. Estas dos capas internas conforman el *back-end* de la aplicación.

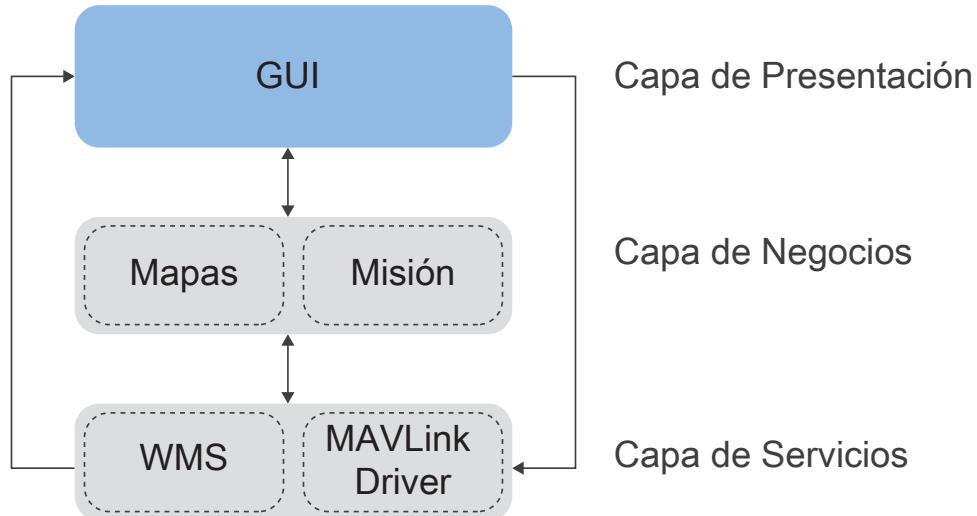


Figura 3.1: Capas que componen la aplicación.

Esta estructura en capas que se presenta ha permitido dividir el problema inicial en subproblemas, para afrontarlos individualmente de una forma más sencilla. Además esta estructura al ser tan gráfica permite entender los diferentes módulos y su función dentro de la aplicación.

Por otro lado, desde un punto de vista de ejecución la aplicación contiene cuatro hilos. Cada uno de estos hilo realiza tareas diferentes en uno o en varios de los bloques ya explicados. Previo paso a la descripción de los hilos de ejecución, es necesario introducir las interfaces utilizadas en la aplicación. Son cuatro, *Pose3D*, *NavData*, *mission* y *extra*, todas reciclamientos de las antiguas interfaces de JdeRobot con ICE [6].

A continuación se presentan estos hilos de ejecución con una breve descripción de los mismos:

- **Hilo principal:** Tiene asociado las principales acciones de la aplicación. Las diferentes ventanas y herramientas surgen de él, al igual que el sistema de mapeado y el creador de misiones.

- **Handler de mensajes:** Se encarga de escuchar y leer los mensajes enviados por la aeronave. Tras leer los mensajes recibidos, extrae la información relevante de los mismos y escribe esta información sobre las dos interfaces.
- **Hilo de navegación:** Se encarga de recoger los datos de navegación leyendo del interfaz *NavData* para actualizar los sensores de navegación y otros valores importantes sobre el estado de navegación de la aeronave.
- **Hilo de posición:** De forma similar al hilo anterior, lee el interfaz *Pose3D* y actualiza la posición de la aeronave sobre el mapa de la aplicación.

Sobre los hilos de ejecución es importante mencionar que no todos se lanzan al ejecutar la aplicación. Al inicio de la ejecución se crean tres hilos, el principal, el de navegación y el de posición, y durante el establecimiento de conexión con la aeronave se genera el *handler*. Sin embargo, tanto el hilo de navegación como el de posición se encuentran latentes o inactivos hasta el establecimiento de conexión. Se darán más detalles sobre su funcionamiento durante la sección 3.5.

Por último, y previo paso a la vista en detalle de cada bloque, se presenta una figura (Fig. 3.2) con la estructura general de clases que contiene la aplicación. Esta figura contiene toda la información hasta ahora mencionada sobre el diseño de la aplicación: bloques, hilos de ejecución e interfaces. Esta figura supone una vista en conjunto cuya intención es proveer al lector de una visión global detallada sobre el software previo a ese análisis pormenorizado sobre cada uno de los bloques.

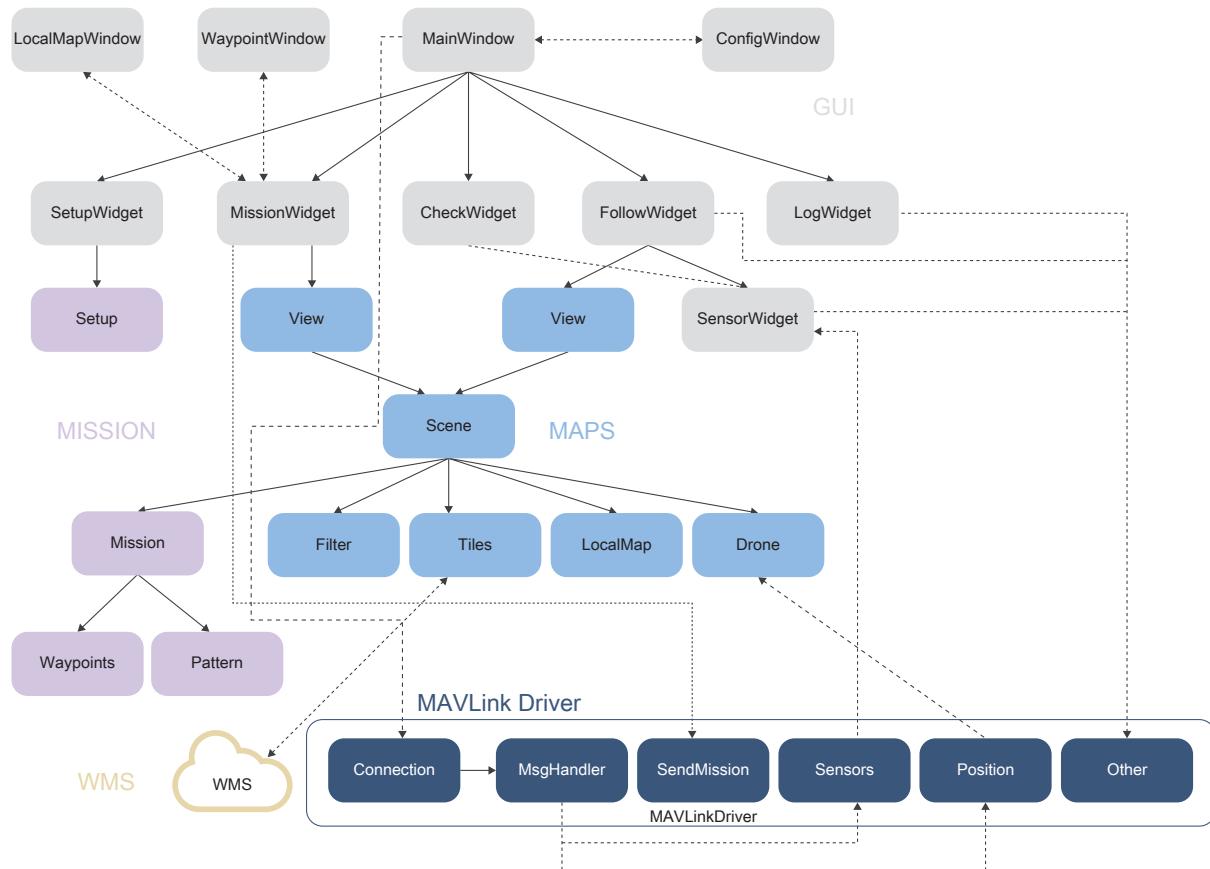


Figura 3.2: Esquema de clases del sistema.

También cabe destacar que este software no ha sido desarrollado en su totalidad por el autor, sino que hace uso de cierta infraestructura anteriormente desarrollada. Además del *driver* de comunicaciones MAVLink (desarrollado por J.A. Fernández [6]) y las interfaces de JdeRobot (*NavData* y *Pose3D*), también se hace uso de una librería de sensores que permite visualizar de forma más realista los datos de navegación [7]. A continuación, cada uno de los bloques de software de la aplicación se detalla en las sucesivas secciones de este capítulo.

3.2. Interfaz Gráfico de Usuario

La interfaz gráfica de usuario (GUI, *Graphical User Interface*) se compone de una serie de objetos gráficos que presentan la información y acciones disponibles al usuario. Se organiza en un conjunto de ventanas y *widgets*. La aplicación dispone de una ventana principal, cuatro ventanas secundarias y varias ventanas de carga y guardado. También se hace uso de diálogos para mostrar mensajes al usuario.

Ventana principal

La ventana principal se divide en cinco diferentes pestañas según las funcionalidades de la aplicación. Estas pestañas coinciden con los objetivos secundarios presentados durante la Sección 1.3 y reciben el nombre de *Setup* para la caracterización de la aeronave y la carga de pago, *Mission* para la creación de misiones de vuelo automático, *Checklist* para la comprobación de pre-vuelo, *Follow* para el seguimiento de vuelo y *Log* para el análisis de datos post-vuelo. En la Figura 3.3 se puede observar la primera pestaña de la aplicación, tal y como se encuentra recién iniciada la aplicación.

Esta estructura se ha conseguido haciendo uso de diferentes *QWidgets* asociado a cada una de las pestañas organizados a través de un *QStackedWidget* y una serie de *QAction* que permiten activar y visualizar un *QWidget* u otro. A continuación se muestra un fragmento del código (Cód. 1) donde se observa la estructura utilizada.

Para cada una de las pestañas se crea una clase nueva a través de una herencia simple de la clase abstracta *QWidget*. De esta forma, las clases creadas comparten los atributos y procedimientos de la clase base, pudiendo ser personalizadas para que cumplan sus requisitos individuales.

Ventanas secundarias

Entre las ventanas secundarias anteriormente mencionadas se encuentran la ventana de configuración, la ventana de carga de mapa local, la ventana de sensores y la ventana de puntos de paso en la misión. En la Figura 3.4 se pueden observar algunas de estas ventanas secundarias.

En la ventana de configuración se pueden observar algunos de los datos persistentes entre ejecuciones. El almacenamiento persistente y la recuperación de la configuración previa se consigue gracias a la clase *QSettings* que proporciona Qt5. Además, en esta ventana se puede observar también la posibilidad de cambiar el idioma de la interfaz durante la ejecución. Esta traducción dinámica es posible gracias a la clase *QTranslator*.

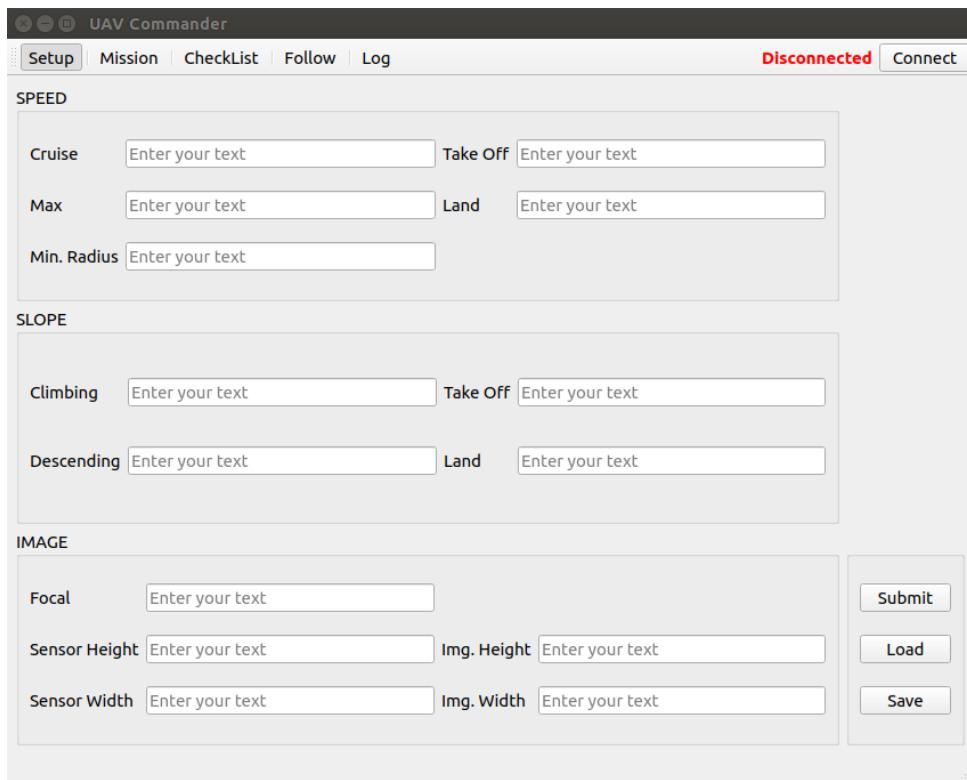


Figura 3.3: Ventana principal con la pestaña *Setup* activa.

de Qt5 que permite cargar ficheros de traducción previamente generados y realizar las traducciones. Actualmente la aplicación se encuentra disponible en castellano e inglés.

3.3. Sistema de Mapeado

El bloque de mapas permite la visualización e interacción a través de la interfaz gráfica de la superficie terrestre. Son muchos los actores que entran en juego y que permite su funcionamiento. La principal herramienta sobre la que trabajan el resto de herramientas es una *QGraphicsScene* (escena). Esta clase proporciona una superficie para desplegar y administrar una serie de elementos gráficos 2D.

Entre las diferentes acciones de las que se encarga el sistema se distinguen la visualización, la obtención de la imagen, la conversión de coordenadas y el manejo de las interacciones del usuario.

Visualización del mapa

Para dar soporte a la visualización del mapa a través de la interfaz gráfica no es suficiente con la escena. La clase *QGraphicsView* proporciona un *widget* para mostrar el contenido de una *QGraphicsScene*. Así pues, la *QGraphicsView* (vista) sirve de nexo entre la escena y la interfaz gráfica. Son dos las vistas utilizadas sobre una única escena, pues en dos pestañas es necesario visualizar el mapa, en *Mission* y en *Follow*.

Por otro lado, con el objetivo de agilizar la visualización y la navegación a través del mapa se presenta un mapa en mosaico, compuesto por un número fijo de teselas o imágenes. Así pues, se introduce la primera herramienta bajo la escena, las teselas. Las teselas constituyen un grupo de elementos gráficos 2D que son gobernados bajo las mismas reglas. Estos

```

1 def __init__(self, *args, **kwargs):
2
3     ....
4     self.widgets = QStackedWidget()
5     self.widgets.addWidget(MySetupWidget(self))
6     self.widgets.addWidget(MyMissionWidget(self))
7     self.widgets.addWidget(MyChecklistWidget(self))
8     self.widgets.addWidget(MyFollowWidget(self))
9     self.widgets.addWidget(MyLogWidget(self))
10
11    self.setCentralWidget(self.widgets)
12
13    toolbar = QToolBar()
14    self.addToolBar(toolbar)
15
16    self.setup_action = QAction("Setup", self)
17    self.setup_action.triggered.connect(self.onSetupToolBarClick)
18    self.setup_action.setCheckable(True)
19    toolbar.addAction(self.setup_action)
20
21    toolbar.addSeparator()
22    ....
23
24 def onSetupToolBarClick(self):
25     self.setup_action.setChecked(True)
26     self.mission_action.setChecked(False)
27     self.checklist_action.setChecked(False)
28     self.follow_action.setChecked(False)
29     self.log_action.setChecked(False)
30
31     self.widgets.setCurrentIndex(0) # Fija el widget visible

```

Código 1: Estructura de pestañas en la ventana principal.

elementos gráficos son en su totalidad imágenes con una posición asociada que permite la construcción del mosaico. En esta primer versión del software las teselas utilizadas son nueve, aunque se está trabajando para que el número utilizado sea variable permitiendo así modificar el tamaño de la escena y dar más versatilidad al sistema de mapeado.

Obtención de la imagen

El sistema de mapas soporta diferentes formatos u orígenes de las imágenes. Actualmente existen tres alternativas para visualizar el mapa:

- **Imagen satélite de Google Earth:** Se obtiene a través del servidor de teselas de

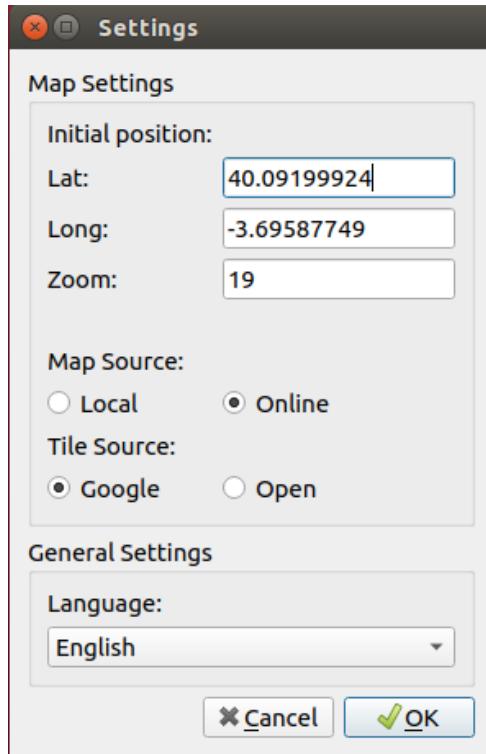


Figura 3.4: Ventana secundaria de configuración.

Google (GTS, *Google Tile Server*).

- **Imagen de Open Maps:** Se obtiene a través del servidor de teselas de Open Street Maps.
- **Imagen satélite del IGN:** Se obtiene cargando archivos locales con fotografías geo-referenciadas, como los archivos ECW o TIFF. Estos archivos provienen del Instituto Geográfico Nacional a través del programa nacional de ortofotografía aérea (PNOA).

Paralelamente al sistema de teselas existe otra herramienta que permite el uso de mapas locales. Esta herramienta permite leer y guardar información relevante sobre los archivos geo-referenciados (ECW o TIFF) para extraer así las teselas sobre la imagen completa, que serán utilizadas por la herramienta de teselas anterior.

En la Figura 3.5 se pueden comparar las tres opciones con las que se puede visualizar el mapa. Las imágenes mostradas corresponden a una tesela de tamaño 256x256 píxeles sobre la misma posición.

La obtención de la imagen depende del tipo de mapa a visualizar. En la Figura 3.6 se representa la secuencia de llamadas necesaria para obtener una escena.

De la secuencia de llamadas se puede deducir diferentes aspectos relevantes de la obtención de mapas. En primer lugar, la aplicación utiliza una caché para reducir la latencia y agilizar así la obtención del mosaico. Esto es debido a que los servicios WMS o la lectura de grandes ficheros geo-referenciados son procesos lentos y pesados.

También se puede observar que los primeros pasos para la obtención de las teselas son comunes, y solo en el último paso difieren los caminos según el *source* (origen) introducido como parámetro. Además, la función *get-scene()* llamará a *get-pixmap()* tantas veces como



(a) Tesela de Google Earth.

(b) Tesela de Open Maps.

(c) Tesela de mapa local.

Figura 3.5: Diferentes teselas disponibles para visualizar el mapa.

teselas posea la escena.

Por último, para obtener una escena concreta es necesario conocer las coordenadas (x, y, z), el origen (*source*) y el desplazamiento del centro (cx, cy). Las diferentes coordenadas utilizadas se explicarán en la siguiente subsección.

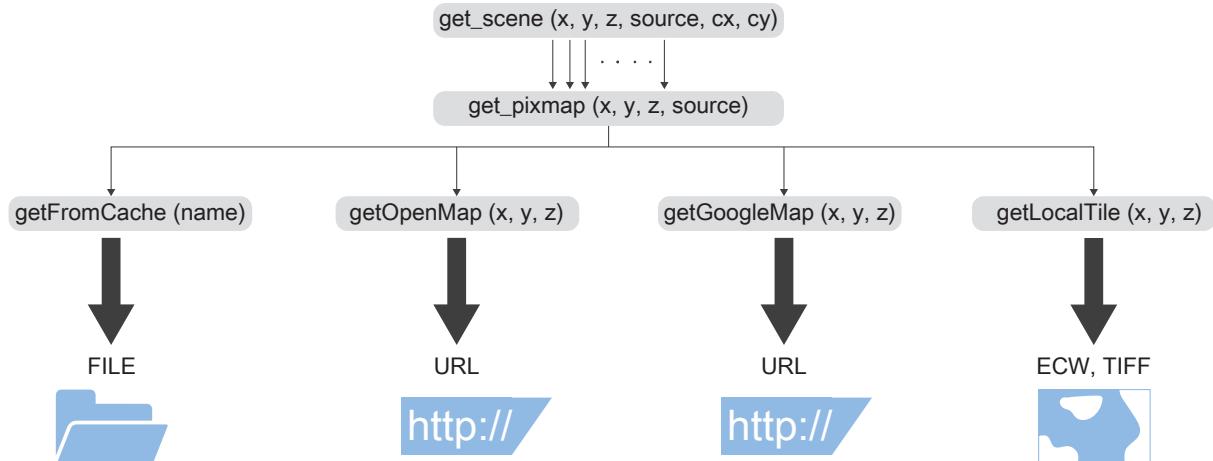


Figura 3.6: Secuencia de llamadas para obtener una escena concreta.

Conversión de coordenadas

La conversión de coordenadas de latitud y longitud a píxeles en la escena es un proceso complejo e inexacto. El simple posicionamiento sobre el globo terrestre es una aproximación más o menos exacto según el datum utilizado. El geoide o el elipsoide seleccionado trata de aproximarse a la forma real del globo terráqueo o a una zona de la superficie de la Tierra. Desde 1984 el geoide utilizado de forma mundial es el WGS84.

En segundo lugar, la conversión de una esfera (o algo que se asemeja a una esfera) a un plano conlleva también otro error asociado. La proyección más utilizada mundialmente es la proyección de Mercator.

El siguiente paso supone la digitalización de la proyección. En función del tamaño y resolución del mismo, se asocia un nivel de zoom. Para un zoom igual a cero, la imagen tiene normalmente 256x256 píxeles, el siguiente nivel tiene 512x512 píxeles, y así sucesivamente. A este esquema de zooms se le conoce como los diferentes niveles de una pirámide.

Por último, cada nivel de la pirámide se divide en teselas de tamaño normalmente igual a 256x256 píxeles. De esta forma, para un zoom igual a uno habría cuatro diferentes teselas.

Así pues, en la cima de la pirámide habría una tesela, en el siguiente nivel 4 teselas, 16 en el siguiente, etc. A cada una de estas teselas se le asocia un coordenada x y otra y para posicionarla sobre el mosaico.

En la Figura 3.7 se representa este sistema de conversión de coordenadas explicado. Este método es ampliamente utilizado y supone un estándar en cartografía para la representación de la superficie terrestre en prácticamente cualquier herramienta de navegación existente hoy en día. Al ser una solución muy utilizada, existen multitud de útiles que facilitan su uso. En la aplicación se ha creado un archivo *TileUtils.py* que recoge estas funciones para la obtención y conversión de coordenadas.

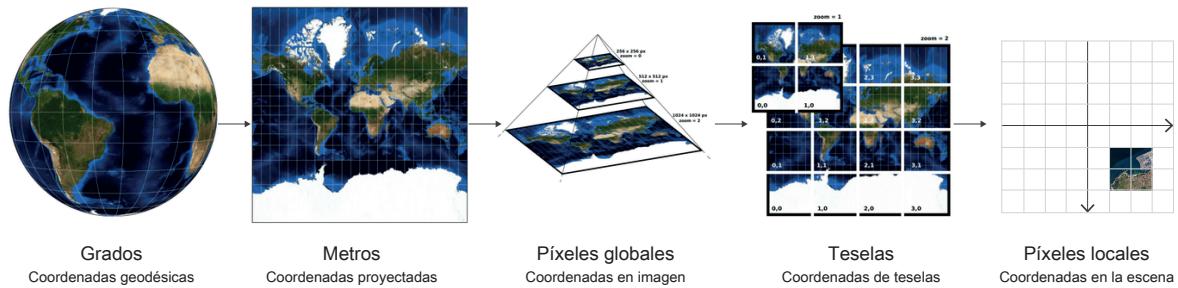


Figura 3.7: Esquema de conversión de coordenadas.

Debido al uso de una escena con un sistema de coordenadas propio es necesario una conversión más, entre el sistema de coordenadas de píxeles global y el sistema de píxeles local asociado a la escena. Estas funciones también se encuentran en el fichero de útiles *TileUtils.py*.

Interacción con el usuario

El sistema de mapas soporta acciones típicas de navegación como el arrastre, el aumento o la disminución del mapa. Para esto, es necesario manejar las interacciones del usuario sobre el mapa. Se presenta entonces la tercera herramienta de la escena, un filtro que distingue entre los diferentes eventos recibidos por la escena y los redirige para tratarlos de forma diferente. A continuación se presenta un fragmento del código (Cód. 2) que se encarga de filtrar los diferentes eventos.

Los eventos filtrados son tres. En la línea 2 (Cód. 2) se detectan las pulsaciones de ratón, registrando la posición del mismo, mientras que en la línea 6 se detecta la posición en la que se suelta el pinchazo de ratón sobre la escena. Estas dos posiciones se comparan (línea 9), si son iguales, la acción realizada por el usuario se considera un clic y entonces el evento se trata como la suma de un nuevo punto de paso (línea 11). En cambio, si las posiciones detectadas entre pulsación y suelte son diferentes, la acción se considera un arrastre sobre el mapa mostrado, por lo que la escena se actualizará (línea 17).

Por otro lado, en la línea 20 se detectan los eventos relacionados con la rueda del ratón que se tratan como aumentos o disminuciones sobre el mapa. Además, cualquier otro evento sobre la escena sería fácilmente detectable y manejable modificando solamente el filtro.

Quedan por presentar dos herramientas más de la escena, una relacionada con la creación de misiones y otra con la representación del drone sobre la escena. Ambas serán explicadas en las siguientes secciones.

```

1  def sceneEventFilter(self, source, event):
2      if event.type() == QtCore.QEvent.GraphicsSceneMousePress:
3          if event.button() == QtCore.Qt.LeftButton:
4              self.initPos = event.scenePos()
5              return True
6
7      elif event.type() == QtCore.QEvent.GraphicsSceneMouseRelease:
8          if event.button() == QtCore.Qt.LeftButton:
9              self.pos = event.scenePos()
10             if self.initPos == self.pos:
11                 # Add waypoint
12                 self.scene().addWayp(event.scenePos())
13             else:
14                 # Drag & Drop
15
16             ....
17
18             self.scene().updateScene(desp_tx, desp_ty, desp_tz, cx, cy)
19             self.initPos = None
20             return True
21
22     elif event.type() == QtCore.QEvent.GraphicsSceneWheel:
23         # Zomm-in Zomm-out by mouse wheel movement
24
25         ....
26
27         self.scene().updateZoom(desp_tz, pos)
28         return True
29
30
31     return QGraphicsItem.sceneEventFilter(self, source, event) # Another event

```

Código 2: Filtro de eventos de la escena.

3.4. Creador de Misiones

El creador de misiones permite, como su propio nombre indica, la creación y el envío de misiones a la aeronave a través del driver de MAVLink. Está íntimamente ligado a la escena, pues las misiones tienen un componente gráfico relevante que se representa sobre la escena.

Una misión se compone por los datos de configuración de la aeronave y la carga de pago y por una sucesión de puntos de paso. Ambos datos se pueden introducir a través de las dos primeras pestañas de la aplicación, *Setup* y *Mission*. Como es lógico pensar, los datos de configuración no dependen de la escena, mientras que la misión en sí misma (los puntos de paso) depende de la escena al tener elementos gráficos que representar.

Es importante destacar que ambas informaciones se pueden guardar en diferentes archivos de forma persistente en el disco. De la misma forma, estos archivos se pueden cargar en memoria. El formato seguido para la creación de estos ficheros de configuración y de misión es un formato plano de texto [17].

Existen dos tipos de misiones, las misiones polilínea y las misiones de patrón. Las misiones polilínea se componen por una sucesión de puntos de paso y se crean introduciendo los diferentes puntos por orden. Es el tipo de misión más sencillo. Existen tres tipos de puntos, despegue, de paso o aterrizaje. Para introducir estos puntos existen dos posibilidades, bien pinchando sobre la escena directamente o introduciendo a mano una latitud y longitud deseada. En ambos casos es necesario introducir a mano la altura de vuelo deseada. En la Figura 3.8a se puede observar un ejemplo de una misión polilínea ya creada.

En cambio, las misión por patrón son más complejas. Se generan introduciendo los vértices de un polígono cerrado. Al igual que con la misión polilínea estos vértices pueden ser introducidos pinchando sobre el mapa o introduciendo la latitud y longitud del punto. El polígono puede ser irregular y convexo, la única restricción es que sea un polígono cerrado. Una vez creado el polígono se genera un algoritmo de barrido que recorre todo la superficie abarcada mediante una sucesión de puntos de paso. La altura introducida debe ser igual para todos los puntos de paso y se selecciona al introducir los vértices. En la Figura 3.8b se puede observar un ejemplo de una misión de patrón ya creada. Además, en el Código 3 se muestra un fragmento del algoritmo de barrido creado para recorrer un superficie delimitada por una lista de vértices.

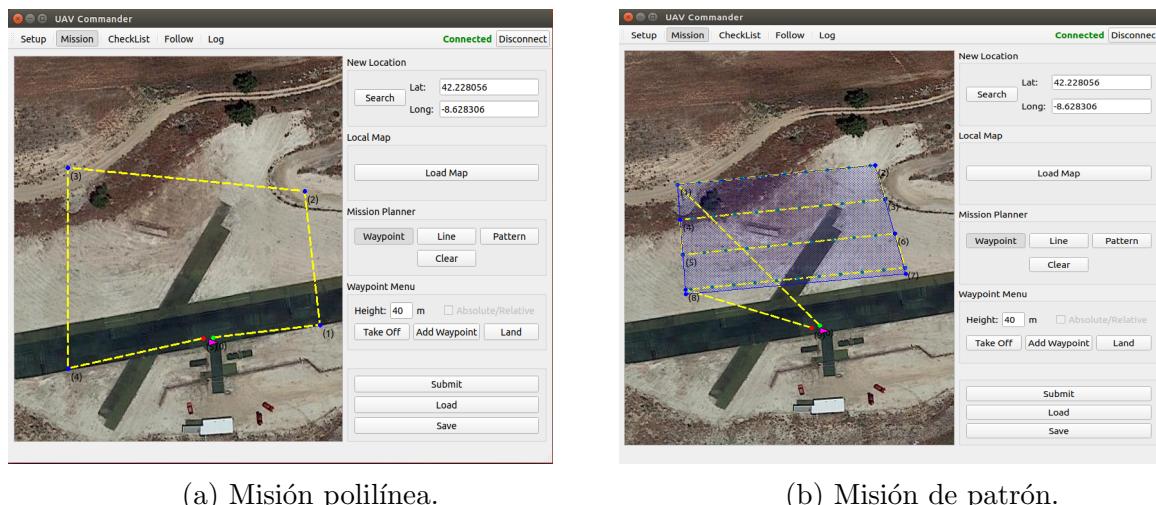


Figura 3.8: Tipos de misión disponibles.

La altura seleccionada para los puntos de paso o el barrido puede ser global o relativa. El término *global* apunta a una altura de referencia constante (global) según el punto de despegue para toda la misión. En cambio, el término *relativo* utiliza una referencia variable en función de la altitud del terreno en cada punto.

En las Figuras 3.8 se pueden observar dos ventanas secundarias que muestran la lista de puntos de paso o vértices introducidos. Es a través de estas ventanas como se pueden añadir tanto los puntos de paso como los vértices a partir de su latitud y longitud. En una futura versión se planea introducir la posibilidad de modificar los puntos desde estas ventanas, aportando más versatilidad a esta herramienta.

Cierta parte de este bloque se integra como una herramienta dentro de la escena, pues toda la representación gráfica mediante puntos, líneas o polígonos cobra cierta relevancia a la hora de visualizar la misión que se está creando. Así pues, la cuarta herramienta de la escena se ha ido explicando a lo largo de esta sección.

```

1  def calc_path(self):
2      self.set_scan(self.p_init) # Fija los parámetros iniciales del escaneo
3
4      # Primera pasada
5      self.wayp.append(self.v_scan)
6      self.calc_intermediates(self.v_scan,
7          self.get_first_vertex(self.v_scan, self.l_scan))
8      self.wayp.append(self.get_first_vertex(self.v_scan, self.l_scan))
9
10     i = 1
11     while True:
12
13         # Calcula la siguiente pasada
14         aux = self.get_next_aux(i)
15         # Comprueba si la pasada es válida
16         vertexes_valid = self.get_valid_vertex(aux)
17         # Ordena los puntos de la pasada
18         vertexes_valid = self.get_next_vertex(i, vertexes_valid, self.u_scan)
19
20         if len(vertexes_valid) == 0: # FIN
21             break
22         elif len(vertexes_valid) == 1:
23             self.wayp.append(vertexes_valid[0])
24         elif len(vertexes_valid) >= 2:
25             self.wayp.append(vertexes_valid[0]) # Siguiente pasada
26             self.calc_intermediates(vertexes_valid[0], vertexes_valid[-1])
27             self.wayp.append(vertexes_valid[-1])
28
29         i = i + 1

```

Código 3: Algoritmo de barrido para las misiones de patrón.

3.5. MAVLink Driver

El último bloque integra diferentes funciones que dependen directamente de la comunicación con la aeronave. El driver de comunicaciones es bidireccional, y sirve tanto para recibir como para enviar información a la aeronave. Una vez más, se quiere resaltar que este bloque parte del trabajo inicial de J.A. Fernández en su proyecto fin de carrera [6]. Las diferentes funciones se pueden clasificar de la siguiente forma:

- Establecimiento de conexión.
- Envío de misión.
- Control de vuelo: Navegación y sensores.
- Control de vuelo: Posicionamiento de aeronave.

- Otros: modo de vuelo, velocidad crucero, etc.

Las diferentes funcionalidades se analizarán una por una en las siguientes subsecciones.

Establecimiento de conexión

La conexión con la aeronave se establece gracias a la librería pyMavlink, que como se ha explicado durante el Capítulo 2, propone una implementación del protocolo de comunicaciones MAVLink facilitando acciones como esta.

En el fragmento de código (Cód. 4) se pueden observar los dos procedimientos principales. Cabe destacar que el establecimiento no solo implica atarse a una dupla IP-puerto, sino que también incluye la creación de un *handler* que se encarga de recibir los mensajes de la aeronave.

```

1 def establishConnection(self):
2     # Launch the MAVLink messages handler
3     if self.connection == 0:
4         print("New connection")
5         self.connection = MAVLinkDriver.uav_connect("udp:127.0.0.1:14540") # SITL
6         # self.connection = MAVLinkDriver.uav_connect("udpin:0.0.0.0:14550") # 3DRSolo
7
8     msgHandler = threading.Thread(target=MAVLinkDriver.mavMsgHandler, args=(
9             self.connection, self.pose, self.navdata,
10            self.msg_handler_thread_kill_signal), name='msg_Handler')
11    msgHandler.start()
12
13    ....
14
15 def uav_connect(port, baudrate=None):
16     master = mavutil.mavlink_connection(port, baudrate, autoreconnect=True)
17     print('Connection established to device')
18     heartbeat = master.wait_heartbeat()
19
20     print("Heartbeat Received", heartbeat)
21
22     # Set the complete set of commands
23     master.mav.request_data_stream_send(master.target_system,
24                                         master.target_component,
25                                         mavutil.mavlink.MAV_DATA_STREAM_ALL,
26                                         RATE, 1)
27
28     return master

```

Código 4: Establecimiento de conexión.

El protocolo de conexión de MAVLink es muy sencillo y se basa en el envío de mensajes *HEARTBEAT*. Se utiliza para anunciar la existencia de un sistema en la red MAVLink, junto con su identificación del sistema y componente, tipo de vehículo, tipo de autopiloto, estado y modo de vuelo. Los distintos componentes deben transmitir regularmente sus *HEARTBEAT* y controlar los recibidos desde otros componentes y/o sistemas. La estructura del mensaje *HEARTBEAT* se presenta en la Tabla 3.1. Para más detalles sobre el tipo y los valores asociados a cada campo, se recomienda visitar la documentación de MAVLink [20].

Campo	Tipo	Valores	Descripción
type	uint8_t	MAV_TYPE	Tipo de vehículo o componente.
autopilot	uint8_t	MAV_AUTOPILOT	Tipo de autopiloto.
base_mode	uint8_t	MAV_MODE_FLAG	Mapa de bits del modo del sistema.
custom_mode	uint32_t		Flags del autopiloto.
system_status	uint8_t	MAV_STATE	Flags de estado del sistema.
mavlink_version	uint8_t		Versión de MAVLink.

Tabla 3.1: Mensaje *HEARTBEAT* [20].

Entre las tareas que realiza el *handler* se encuentran el manejo de mensajes enviados por el autopiloto, el envío de los *heartbeats* o el refresco de las interfaces *Pose3D* y *NavData* con datos de la aeronave. El Código 5 recoge la implementación en código de estas tareas. Más adelante se explicarán también las funciones de las interfaces y el control de vuelo.

```

1 def mavMsgHandler(master, pose, navdata, stop_event):
2     lastSentHeartbeat = 0
3     while not stop_event.isSet():
4         msg = master.recv_msg()
5
6         # send heartbeats to autopilot
7         if time.time() - lastSentHeartbeat > 0.5:
8             master.mav.heartbeat_send(mavlink2.MAV_TYPE_GCS,
9                             mavlink2.MAV_AUTOPILOT_INVALID, 0, 0, 0)
10            lastSentHeartbeat = time.time()
11
12            # refresh the attitude
13            refreshAPMPose3D(master, pose)
14            refreshAPMnavdata(master, navdata)
15
16        elif msg is None or msg.get_type() == "BAD_DATA":
17            time.sleep(0.01)
18            continue

```

Código 5: Handler de conexión.

Envío de misión

El envío de misiones se realiza a través del protocolo de misiones que propone MAVLink. La Figura 3.9 recoge un diagrama del protocolo, el cual ha sido sacado de la propia web de MAVLink [22].

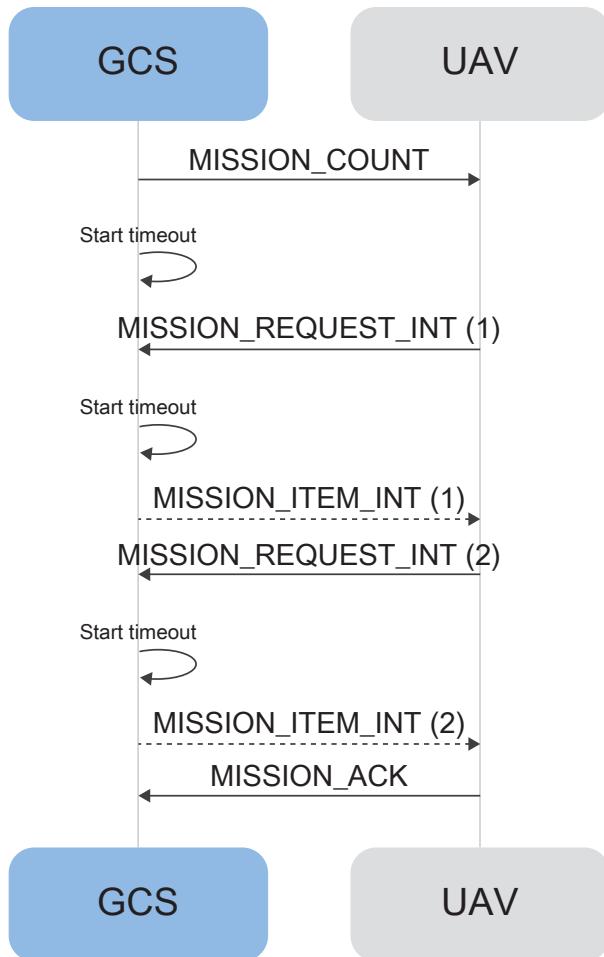


Figura 3.9: Diagrama del protocolo de misiones de MAVLink.

Los elementos que componen la misión se envían mediante mensajes `MISSION_ITEM`. La estructura de estos mensajes se representa en la Tabla 3.2. Para más detalles sobre los campos del mensaje [21] o sobre otros mensajes que intervienen en el protocolo se recomienda visitar la documentación de MAVLink [16].

Estos mensajes encapsulan otros mensajes de tipo `MAV_CMD`. Algunos ejemplos pueden ser los mostrados durante el Código 3.9 en la línea 20 (`MAV_CMD_NAV_TAKEOFF`), línea 32 (`MAV_CMD_NAV_LAND`) o en la línea 43 (`MAV_CMD_NAV_WAYPOINT`), aunque existen muchos más [1]. En función del comando especificado en el campo `command` los siete parámetros tomarán un significado u otro. La Tabla 3.3 recoge las diferencias entre los tres comandos mencionados.

Campo	Tipo	Valores	Descripción
target_system	uint8_t		ID Sistema.
target_component	uint8_t		ID Componente.
seq	uint16_t		Secuencia.
frame	uint8_t	MAV_FRAME	Sistema de coordenadas.
command	uint16_t	MAV_CMD	Acción programada.
current	uint8_t		Falso (0) o verdadero (1).
autocontinue	uint8_t		Continuar automático.
param1	float		PARAM1.
param2	float		PARAM2.
param3	float		PARAM3.
param4	float		PARAM4.
x	float		PARAM5 (Coord. X).
y	float		PARAM6 (Coord. Y).
z	float		PARAM7 (Coord. Z).
mission_type	uint8_t	MAV_MISSION_TYPE	Tipo de misión.

Tabla 3.2: Mensaje *MISSION_ITEM* [21].

Parámetros	<i>NAV_TAKEOFF</i>	<i>NAV_LAND</i>	<i>NAV_WAYPOINT</i>
param1	Cabeceo.	Altura de decisión.	Tiempo de espera.
param2	-	Modo de aterrizaje.	Radio aceptado.
param3	-	-	radio de paso.
param4	Guiñada.	Guiñada.	Guiñada.
param5	Latitud.	Latitud.	Latitud.
param6	Longitud.	Longitud.	Longitud.
param7	Altitud.	Altitud.	Altitud.

Tabla 3.3: Comparación entre parámetros de tres comandos *MAV_CMD_*.

Para la implementación del protocolo se parte de un pseudo-código propuesto por la Universidad de Colorado Boulder [42]. Este boceto de implementación se adapta a la estructura de misiones de la aplicación, para construir los diferentes mensajes y el envío de misión. En el Código 6 se recoge el resultado final de la implementación del protocolo.

```

1 def set_px4_mission(master, mission, extra, frame_type):
2     wp = mavwp.MAVWPLoader()
3     seq = 1
4     if frame_type:
5         frame = mavutil.mavlink.MAV_FRAME_GLOBAL # 0
6     else:
7         frame = mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT # 3
8
9     radius = 10

```

```

10    pose3Dwaypoints = mission.getMission()
11
12    N = len(pose3Dwaypoints)
13
14    for i in range(N):
15        if (i==0 and extra.takeOffDecision()):
16            navData, is_img = pose3Dwaypoints[0]
17            toff = mavutil.mavlink.MAVLink_mission_item_message(master.target_system,
18                                                    master.target_component,
19                                                    seq, frame,
20                                                    mavutil.mavlink.MAV_CMD_NAV_TAKEOFF,
21                                                    0, 1, 0, 0, 0, 0,
22                                                    navData.x, navData.y, navData.h)
23            wp.add(toff)
24            print("[MAVLink Driver] Takeoff to " + str(toff))
25            seq += 1
26            extra.setTakeoff(False)
27        elif (i==N-1 and extra.landDecision()):
28            navData, is_img = pose3Dwaypoints[N - 1]
29            land = mavutil.mavlink.MAVLink_mission_item_message(master.target_system,
30                                                    master.target_component,
31                                                    seq, frame,
32                                                    mavutil.mavlink.MAV_CMD_NAV_LAND,
33                                                    0, 1, 0, 0, 0, 0,
34                                                    navData.x, navData.y, 0)
35            seq += 1
36            wp.add(land)
37            print("[MAVLink Driver] Land on " + str(land))
38        else:
39            navData, is_img = pose3Dwaypoints[i]
40            wayPoint = mavutil.mavlink.MAVLink_mission_item_message(master.target_system,
41                                                    master.target_component,
42                                                    seq, frame,
43                                                    mavutil.mavlink.MAV_CMD_NAV_WAYPOINT,
44                                                    0, 1, 0, radius, 0, 0,
45                                                    navData.x, navData.y, navData.h)
46            wp.add(wayPoint)
47            seq += 1
48            print("[MAVLink Driver] Waypoint on " + str(wayPoint))
49
50    master.waypoint_clear_all_send()
51
52    print("[MAVLink Driver] " + str(wp.count()) + " mission items to send")
53    master.waypoint_count_send(wp.count())
54
55    for i in range(wp.count()):
56        msg = master.recv_match(type=['MISSION_REQUEST'], blocking=True)
57        print(msg)

```

```

58     master.mav.send(wp.wp(i))
59     print ('[MAVLink Driver] Sending waypoint {0} '.format(i)
60                         + format(wp.wp(msg.seq)))
61
62     mission_validation = master.recv_match(type=['MISSION_ACK'], blocking=True)
63     print("[MAVLink Driver] Mission ACK message (type = 0 means successful)"
64           + str(mission_validation))
65
66     if getattr(mission_validation, 'type') == 0:
67         print('[MAVLink Driver] Mission SENDED')
68         empty_mission = Mission()
69         mission.setMission(empty_mission)
70         return 0
71     else:
72         return 1

```

Código 6: Implementación del protocolo de misión de MAVLink.

Para el correcto funcionamiento se hace uso de dos interfaces de JdeRobot *mission* y *extra* que almacenan la información necesaria para construir cada uno de los elementos de la misión. Como se puede observar en el código, en caso de tener *takeOffDecision()* o *landDecision()* activos en el atributo extra se añaden elementos de despegue y aterrizaje en vez de puntos de paso o navegación convencionales.

Control de vuelo: Navegación y sensores

El control de vuelo es subdividido en dos partes. En esta subsección se explicará la parte encargada de los datos de navegación y sensores. Aunque el funcionamiento es similar en ambas partes, se ha decidido separar debido a que las tareas que realiza cada uno son diferentes.

La clave de su funcionamiento radica en dos aspectos, la interfaz *NavData* y el hilo de navegación. Como ya se ha explicado, el *handler* actualiza la interfaz leyendo los mensajes recibidos del autopiloto. Esto se realiza a través del procedimiento *refreshAPMnavdata()* el cual se presenta a continuación (Cód. 7).

```

1 def refreshAPMnavdata(master, navdata):
2     mav_type = master.mav_type
3     mav_autopilot = master.field('HEARTBEAT', 'autopilot', None)
4     mav_state = master.field('HEARTBEAT', 'system_status', None)
5
6     # PX4 - BATTERY_STATUS, ArduPilot - SYS_STATUS
7     # PX4 - HIGHRES_IMU, ArduPilot - RAW_IMU
8
9     if mav_autopilot == 3: # ArduPilot
10        status_msg = 'SYS_STATUS'
11        imu_msg = 'RAW_IMU'
12    elif mav_autopilot == 12: # PX4
13        status_msg = 'BATTERY_STATUS'

```

```

14     imu_msg = 'HIGHRES_IMU'
15 else:
16     status_msg = 'BATTERY_STATUS'
17     imu_msg = 'HIGHRES_IMU'
18
19 # get battery_remaining
20 battery_remaining = master.field(status_msg, "battery_remaining", None)
21 if battery_remaining is None:
22     print("[MAVLink Server] Error: " + status_msg + " not received")
23     battery_remaining = 0
24
25 # get RAW_IMU APM
26 if imu_msg not in master.messages:
27     print("[MAVLink Server] Error: " + imu_msg + " not received")
28     rawIMU = None
29 else:
30     rawIMU = master.messages[imu_msg]
31
32 # get GLOBAL_POSITION_INT
33 if 'GLOBAL_POSITION_INT' not in master.messages:
34     print("[MAVLink Server] Error: GLOBAL_POSITION_INT not received")
35     global_position = None
36 else:
37     global_position = master.messages['GLOBAL_POSITION_INT']
38
39 # refresh the navdata
40 ndata = NavdataData()
41
42 ndata.batteryPercent = battery_remaining
43
44 try:
45     ndata.vx = getattr(global_position, "vx")
46 except Exception as e:
47     print("[MAVLink Server] Error: " + str(e))
48 try:
49     ndata.vy = getattr(global_position, "vy")
50 except Exception as e:
51     print("[MAVLink Server] Error: " + str(e))
52
53 ....
54
55 ndata.vehicle = mav_type
56 ndata.state = mav_state
57
58 navdata.setNavdataData(ndata)

```

Código 7: Procedimiento `refreshAPMnavdata()`, encargado de leer y actualizar los datos de navegación enviados por la aeronave.

Como se puede observar en el código, existen diferencias entre los mensajes enviados por los diferentes autopilotos. Esta aplicación está preparada para trabajar con las dos principales soluciones en el mercado, PX4 y ArduPilot. Las diferencias entre ambos autopilotos se refleja en la Tabla 3.4.

Datos	PX4	ArduPilot
Batería	<i>BATTERY_STATUS</i>	<i>SYS_STATUS</i>
IMU	<i>HIGHRES_IMU</i>	<i>RAW_IMU</i>
posición	<i>GLOBAL_POSITION_INT</i>	<i>GLOBAL_POSITION_INT</i>
Actitud	<i>ATTITUDE</i>	<i>ATTITUDE</i>
Altitud	<i>VFR_HUD</i>	<i>VFR_HUD</i>
GPS	<i>GPS_RAW_INT</i>	<i>GPS_RAW_INT</i>

Tabla 3.4: Diccionario de mensajes soportados por cada autopiloto.

Además, la tabla anterior (Tab. 3.4) también indica que información se extrae de cada mensaje. Los parámetros de los mensajes enviados puede comprobarse en la documentación de MAVLink [16].

El hilo de navegación se encarga de acceder periódicamente a la interfaz y actualizar en la interfaz gráfica de usuario esta información. La lectura y escritura sobre la interfaz se protege mediante semáforos para evitar conflictos entre entes. Entre la información actualizada destaca la ventana secundaria de sensores que se puede desplegar desde la pestaña de seguimiento (*follow*). Esta pestaña muestra de forma sencilla y muy visual el estado de la aeronave en vuelo replicando los instrumentos de vuelo de una cabina convencional. La Figura 3.10 muestra esta ventana.



Figura 3.10: Ventana secundaria de sensores.

Por último, se presenta un esquema que trata de representar el flujo de información y la secuencia de llamadas del control de vuelo. Este diagrama se muestra en conjunto con

el otro control de vuelo y se puede observar en la Figura 3.11.

Control de vuelo: Posicionamiento de aeronave

El control de vuelo que se encarga del posicionamiento de la aeronave funciona de forma similar al anterior. El *handler* actualiza la interfaz *Pose3D* con los datos de posicionamiento a través del procedimiento *refreshAPMPose3D()*. Dicho fragmento de código se presenta a continuación (Cód. 8). Los mensajes enviados por cada autopiloto se presentan en la Tabla 3.4. Los parámetros estos mensajes enviados puede comprobarse en la documentación de MAVLink [16].

El hilo de posición accede periódicamente a la interfaz y actualiza en la escena la posición de la aeronave. La lectura y escritura sobre la interfaz se protege mediante semáforos para evitar conflictos entre entes. La posición de la aeronave se muestra sobre la escena gracias a la última herramienta de la escena que faltaba por introducir. Esta herramienta aporta un elemento gráfico que puede ser desplegado sobre la escena, y cuya posición se actualiza con la información de la interfaz.

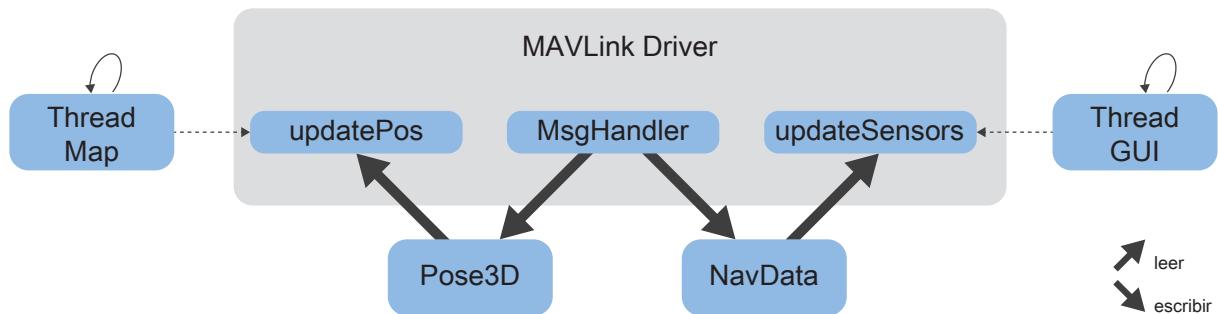


Figura 3.11: Diagrama del control de vuelo.

Finalmente, en la Figura 3.11 se muestra un diagrama con el flujo de información y la secuencia de llamadas de ambas partes del control de vuelo en conjunto.

```

1 def refreshAPMPose3D(master, pose):
2     # get attitude of APM
3     if 'ATTITUDE' not in master.messages:
4         print("[MAVLink Server] Error: ATTITUDE not received")
5     else:
6         attitude = master.messages['ATTITUDE']
7         yaw = getattr(attitude, "yaw")
8         pitch = getattr(attitude, "pitch") * -1
9         roll = getattr(attitude, "roll")
10        q = quaternion.Quaternion([roll, pitch, yaw])
11
12        # get altitude of APM
13        altitude = master.field('VFR_HUD', 'alt', None)
14        if altitude is None:
15            print("[MAVLink Server] Error: VFR_HUD not received")

```

```

16
17     # get GPS position from APM
18     latitude = 0
19     longitude = 0
20     if 'GPS_RAW_INT' not in master.messages:
21         gpsStatus = 1
22     else:
23         gps = master.messages['GPS_RAW_INT']
24
25         latitude = getattr(gps, "lat")/ 10e6
26         longitude = getattr(gps, "lon") / 10e6
27         GPS_fix_type = getattr(gps, "fix_type")
28         sat_visible = getattr(gps, "satellites_visible")
29
30     # refresh the pose3D
31     data = Pose3DDData()
32
33     data.x = latitude
34     data.y = longitude
35     data.z = altitude
36     data.h = altitude
37     data.q0 = q.__getitem__(0)
38     data.q1 = q.__getitem__(1)
39     data.q2 = q.__getitem__(2)
40     data.q3 = q.__getitem__(3)
41     pose.setPose3DDData(data)

```

Código 8: Procedimiento *refreshAPMPose3D()*, encargado de leer y actualizar los datos de posición enviados por la aeronave.

Otros

Entre otras funciones que incluye el *driver* son el modo de cambio de vuelo, el cambio de velocidad de crucero o el armado y el desarmado.

Los modos de vuelo disponibles son:

- **Auto/Mission:** Con el modo automático la aeronave seguirá una misión pre-programada previamente almacenada en el piloto automático.
- **Hold/Loiter:** Este modo ordena mantener la posición, el rumbo y la altitud en el momento de su activación. En el caso de un ala fija, la aeronave realizará círculos alrededor de la posición de activación.
- **Return To Launch:** El modo RTL activa la vuelta de la aeronave al punto de despegue o a un punto fijado como *casa* antes del despegue.

El cambio de modo de vuelo se realiza de forma sencilla gracias a la implementación del protocolo MAVLink de pyMavlink. Los procedimientos utilizados por el *driver* se presentan en el Código 9.

```

1 def set_loiter_mode(master):
2     master.set_mode("LOITER")
3     print('[MAVLink Driver] Flight mode set to LOITER')
4
5
6 def set_rtl_mode(master):
7     master.set_mode_rtl()
8     print('[MAVLink Driver] Flight mode set to RTL')
9
10
11 def set_auto_mode(master):
12     master.set_mode_auto()
13     print('[MAVLink Driver] Flight mode set to MISSION')

```

Código 9: Activación de modos de vuelo.

Para el cambio de velocidad de crucero es necesario utilizar otro protocolo de MAVLink. El protocolo de comandos es muy sencillo, pues únicamente es necesario el envío de un mensaje de tipo *COMMAND_LONG* y esperar su confirmación. La Figura 3.12 muestra un diagrama del protocolo.

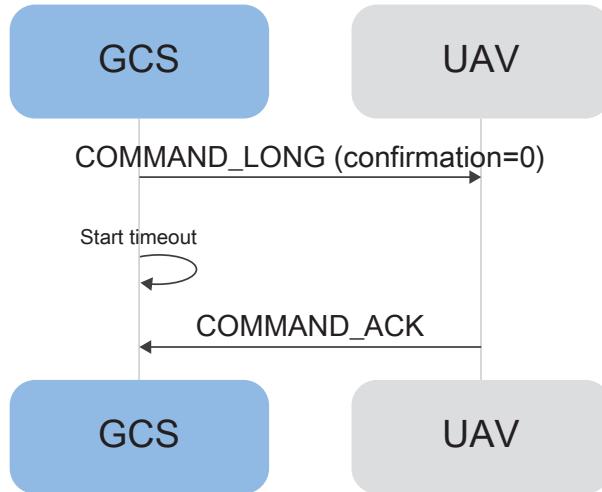


Figura 3.12: Diagrama del protocolo de comandos de MAVLink.

La estructura del mensaje *COMMAND_LONG* se muestra en la Tabla 3.5. Para un cambio de velocidad el comando enviado es *MAV_CMD_DO_CHANGE_SPEED* cuyos parámetros se pueden observar en la Tabla 3.6. Para mas detalles acerca de los campos del mensaje se recomienda visitar la documentación de MAVLink [19]. Por otro lado, la implementación en código del envío del mensaje para el cambio de velocidad se muestra en el Código 10.

Campo	Tipo	Valores	Descripción
target_system	uint8_t		ID Sistema.
target_component	uint8_t		ID Componente.
command	uint16_t	MAV_CMD	ID Comando.
confirmation	uint8_t		Nº de transmisión.
param1	float		PARAM1.
param2	float		PARAM2.
param3	float		PARAM3.
param4	float		PARAM4.
param5	float		PARAM5.
param6	float		PARAM6.
param7	float		PARAM7.

Tabla 3.5: Mensaje *COMMAND_LONG* [19].

```

1 master.mav.command_long_send(master.target_system,
2                         master.target_component,
3                         mavutil.mavlink.MAV_CMD_DO_CHANGE_SPEED,
4                         0,
5                         0, speed, 0, 0, 0, 0, 0)

```

Código 10: Mensaje *COMMAND_LONG* para el cambio de velocidad de vuelo.

Parámetro	Descripción
Tipo de velocidad	Airspeed (0), Ground speed (1), Velocidad de ascenso (2), Velocidad de descenso (3)
Velocidad	Velocidad (m/s)
Empuje	Empuje (%)
Relativa	Absoluta (0) o Relativa (1)
param5	-
param6	-
param7	-

Tabla 3.6: Comando *MAV_CMD_DO_CHANGE_SPEED* [15].

Finalmente, el armado y desarmado de la aeronave también resulta trivial gracias a pyMavlink. El código utilizado por el *driver* se muestra en el siguiente fragmento (Cód. 11).

```
1 def arm_dissarm(master, arm):
2     if arm:
3         master.arducopter_arm()
4         print('[MAVLinkDriver] Vehicle armed')
5     else:
6         master.arducopter_disarm()
7         print('[MAVLinkDriver] Vehicle disarmed')
```

Código 11: Armado y desarmado de la aeronave.

Capítulo 4

Resultados

En este capítulo se recogen las pruebas realizadas a la estación de tierra. A partir de los resultados obtenidos tanto en simulación como en pruebas reales se persigue su validación experimental. Estos experimentos han de comprobar que los objetivos descritos durante el Capítulo 1.3 se cumplen. Las diferentes pruebas se reúnen en los diferentes casos de uso, a los cual se les dedica una de las secciones de este capítulo. Los casos de uso expuestos en este capítulo son ocho:

1. Caracterización de una aeronave y su carga de pago.
2. Carga de un mapa local en una determinada posición.
3. Creación y posterior guardado en fichero de una misión polilínea.
4. Establecimiento de conexión con la aeronave.
5. Creación de una misión por patrón y validación de la misma.
6. Carga de una misión en fichero, envío y seguimiento de la misión.
7. Interrupción de una misión y vuelta a casa.
8. Cambio de idioma de la aplicación.

Además, el Apéndice ?? recoge una tabla que esquematiza cada uno de los casos de uso. Junto a las tablas se añaden una serie de diagramas de caso de uso que tratan de ejemplificar la interacción entre el usuario y la aplicación para cada caso.

Por último, y previo paso a las secciones con los casos de uso, se adjuntan dos enlaces a los vídeos con pruebas experimentales reales, uno con misiones polilíneas¹ y otro con misiones por patrón². Además, se adjunta también la lista de reproducción³ que muestra la evolución de la aplicación a lo largo del tiempo con las distintas versiones realizadas.

¹https://youtu.be/UOv8Go_Q22k

²<https://youtu.be/tppdkG0CjM>

³<https://www.youtube.com/playlist?list=PLGIx46StCA-QAXmeQp5omNhllGJ3CMjcO>

4.1. Caso de uso 1: Caracterización de una aeronave y su carga de pago.

El primer caso de uso describe una situación en la que un operario cualquiera desee caracterizar una aeronave y su carga de pago mediante la introducción de sus parámetros en la aplicación. La tabla con el caso de uso se presenta en el Apéndice ??.

Para poder realizar la acción requerida, la aplicación debe estar abierta y en ejecución. Los diferentes pasos de una secuencia normal serían:

1. En primer lugar, hay que pinchar sobre *Setup* para desplegar dicha pestaña.
2. A continuación, el usuario debe completar los diferentes campos requeridos sobre la configuración de la aeronave. Todo aquel parámetro que permanezca en blanco tomará un valor nulo.
3. El operario, tras comprobar los datos introducidos, pincha sobre el botón *Submit* para caracterizar la aeronave y su carga de pago.
4. En caso de cometer algún error al introducir los datos o de querer modificar algún parámetro, se podrían modificar los campos necesarios y confirmar estos cambios pinchando sobre el botón *Submit*.
5. Para guardar una configuración creada para un uso futuro de la misma, es necesario pinchar sobre el botón *Save* para generar un archivo de configuración. Sobre la ventana emergente hay que elegir un nombre y una ubicación de guardado, y posteriormente pinchar en el botón *Save*.

En el paso dos, si el operario hubiese creado un fichero de configuración anteriormente, podría cargarlo directamente en la aplicación, evitando repetir el proceso. Para ello se debe pinchar en el botón *Load* y seleccionar el archivo de configuración a cargar. Tras seleccionar el archivo es necesario pinchar en el botón *Open* sobre la ventana emergente. Es importante destacar que no es posible tener más de un archivo de configuración abierto al mismo tiempo. Además, en caso de tener un fichero de configuración abierto, cualquier cambio realizado sobre el mismo debe ser confirmado pinchando sobre el botón *Submit* para que tenga efecto sobre el modelo de configuración. Por otro lado, si estos cambios quieren ser guardados de forma persistente sobre el archivo de configuración, es necesario pinchar sobre el botón *Save*. Esta acción sobrescribirá el archivo abierto y no creará un archivo de configuración nuevo.

4.2. Caso de uso 2: Carga de un mapa local en una determinada posición.

En este caso de uso se explica el proceso de carga de un mapa local mostrando una posición determinada en la aplicación. En el Apéndice ?? se muestra una tabla que recoge el caso de uso.

Este ejemplo requiere que la aplicación se encuentre abierta y en ejecución. Además, es necesario disponer de un archivo de imagen geo-referenciada con la ubicación precisada. Estos archivos pueden ser obtenido desde la página web del Instituto Geográfico Nacional

[10]. Solo dos formatos son admitidos, las extensiones TIFF y ECW.

Una secuencia normal ejecutada por un usuario cualquiera sería:

1. Para comenzar, el operario debe pinchar sobre *Mission* para abrir la pestaña con el mismo nombre.
2. Con el objetivo de abrir la ventana secundaria de carga de mapas locales, hay que pinchar sobre el botón *Local Map*.
3. Pinchando sobre el primer botón ... del campo *Input File* se abriría una ventana de carga sobre la que se debe buscar y seleccionar el archivo con el mapa local. A continuación, confirmar el archivo pinchando sobre el botón *Open*.
4. Existe la posibilidad de cargar también un archivo con información acerca de la altitud del terreno. Ambos archivos deben coincidir en la ubicación comprendida en sus datos. Pinchando sobre el segundo botón ... del campo *TIFF File* se abriría otra ventana de carga. Al igual que en el paso anterior, se debe buscar el archivo de elevaciones, seleccionarlo y pinchar sobre el botón *Open* para cargar el fichero.
5. Para modificar la posición inicial desplegada en la escena, hay que pinchar sobre la opción *Options*. Esto muestra la información asociada al mapa local. Modificando los campos *Offset* se puede modificar la posición inicial cargada, que por defecto muestra la esquina superior izquierda de la imagen.
6. Finalmente, para confirmar la acción, el usuario debe pinchar sobre el botón *OK* y en la escena se podría observar el mapa cargado.

En caso de que la posición inicial introducida no fuese válida por encontrarse fuera de la superficie abarcada por el fichero, la aplicación no mostraría ningún error al usuario. Sin embargo, la escena no mostraría ninguna imagen y permanecería mostrando una imagen en blanco, pues sobre esa posición no dispone de información a mostrar. Para modificar la posición mostrada en la escena, se recomienda repetir el caso de uso.

4.3. Caso de uso 3: Creación y posterior guardado en fichero de una misión polilínea.

Este caso de uso prueba las acciones necesarias para la creación y posterior guardado de una misión polilínea. El Apéndice ?? recoge el caso de uso simplificado en una tabla. El único requisito para poder completar este experimento radica en tener abierta y en ejecución la aplicación. La secuencia normal de ejecución se resume en los siguientes puntos:

1. Primeramente, la aplicación debe mostrar la pestaña *Mission*. Para ello, hay pinchar en *Mission* sobre el menú de la ventana principal.
2. Si la escena no mostrase la ubicación sobre la que se quiere crear la misión, la posición se podría modificar introduciendo la latitud y longitud sobre los campos *Lat* y *Long* y pinchando sobre el botón *Search*.
3. Tras obtener el mapa deseado, el usuario debe pinchar sobre el botón *Waypoint* para abrir el creador de misiones por polilínea.

4. A la hora de crear la misión, es necesario introducir el punto de despegue en primer lugar. Para ello hay que pinchar sobre el botón *Take Off* y después pinchar sobre el mapa en el punto deseado de despegue. La altura introducida en el campo *Height* es la altura necesaria a alcanzar antes de comenzar la misión.
5. A continuación se añadirán los puntos de paso deseados por el usuario. Para añadir un punto de paso se debe pinchar sobre el mapa en la posición querida y con la altura asociada al campo *Height*.
6. Para crear el punto de aterrizaje, el usuario pinchará sobre el botón *Land* y posteriormente pinchará sobre el mapa la posición deseada de aterrizaje.
7. Finalmente, para guardar la misión, el usuario debe pinchar sobre el botón *Save* con la misión ya creada completamente. Una ventana de guardado emergirá donde se debe seleccionar un nombre y una ubicación de guardado para el nuevo fichero de misión. Para confirmar el guardado, se debe pinchar sobre el botón *Save* de esta ventana.

El mapa utilizado puede ser cualquiera de los soportados por la aplicación. Si se desea utilizar un mapa local, este se puede cargar siguiendo el caso de uso 2 (Secc. 4.2). Sin embargo, si el usuario no dispusiese de una archivo de mapa local ni tampoco de conexión a internet y los mapas no estuvieran previamente cacheados, no se podría obtener un mapa válido sobre el que crear la misión. Por ello, la acción no sería realizable y habría que posponerla a una situación en la que se dispusiese de algún mapa.

Por otro lado, los puntos se pueden introducir también mediante los campos de latitud, longitud y altura que lo posicionan en el espacio. Tanto el punto de despegue, el punto de aterrizaje o los puntos de paso se pueden añadir de esta forma alternativa. Para ello es necesario abrir la ventana secundaria con la lista de los puntos en la misión. Esta ventana se puede abrir pinchando en el botón *Add Waypoint* en el menú de creador de misiones. Sobre la ventana se encuentran los campos *Lat*, *Lon* y *Alt* junto al botón *Add* con los que se pueden añadir puntos a la misión.

Si el usuario cometiese un error introduciendo alguno de los puntos de la misión, sería necesario pinchar el botón *Clear* para borrar la misión y empezar de nuevo la creación de la misma.

Es importante destacar que no es posible crear más de una misión al mismo tiempo. En caso de tener un archivo de misión abierto, cualquier cambio sobre la misma sobrescribirá el archivo al guardar los cambios (pinchando en el botón *Save*) y no creará un archivo de misión nuevo.

Se adjunta un enlace de un vídeo ⁴ de un experimento real en el que junto a otras acciones se completa el caso de uso actual. Es importante destacar que algún aspecto menor puede diferir con la versión actual, pues en el momento de grabación del vídeo la aplicación se encontraba todavía en proceso de desarrollo.

4.4. Caso de uso 4: Establecimiento de conexión con la aeronave.

El cuarto caso de uso explica el procedimiento a seguir para establecer la conexión entre la aeronave y la estación de tierra. Se agrega en el Apéndice ?? una tabla resumen

⁴https://youtu.be/UOv8Go_Q22k

y simplificada de lo que en esta sección se describe.

Para una perfecta ejecución del caso de uso es necesario que la aplicación se encuentre abierta y en ejecución. A su vez, la aeronave debe estar encendida y con el posicionamiento GPS establecido. La siguiente secuencia recoge los pasos a seguir en situaciones normales:

1. Para tratar de establecer la conexión hay que pinchar sobre el botón *Connect* situado en el menú de la ventana principal.
2. En caso de un establecimiento de conexión favorable, la etiqueta de estado de conexión en el menú cambiará a color verde mostrando el texto *Connected*.

Es importante resaltar que si la conexión no se puede establecer debido a algún error, la aplicación debe ser reiniciada para poder volver a realizar un intento de establecimiento de conexión. También hay que tener en cuenta que el establecimiento puede durar unos segundos en función del estado de la aeronave.

4.5. Caso de uso 5: Creación de una misión por patrón y validación de la misión.

El quinto caso de uso muestra como deberá comportarse la aplicación frente a la intención por parte de un usuario para crear una misión y la validación de la misma. Esta acción conlleva los requisitos de que la aplicación se encuentre abierta y en ejecución, y de que la aeronave se encuentre caracterizada. Para una correcta configuración de la aeronave se puede seguir el caso de uso 1 (Secc. 4.1). Con caso de uso se presenta también una tabla en el Apéndice ?? que recoge sus principales aspectos. La secuencia normal de ejecución es la siguiente:

1. La pestaña *Mission* debe estar activa. Para ello hay que pinchar en el menú con si mismo nombre *Mission*.
2. Si la escena no mostrase la ubicación sobre la que se quiere crear la misión, la posición se podría modificar introduciendo la latitud y longitud sobre los campos *Lat* y *Long* y pinchando sobre el botón *Search*.
3. Tras obtener el mapa deseado, el usuario debe pinchar sobre el botón *Pattern* para abrir el creador de misiones por patrón.
4. Para fijar la altura de vuelo durante la ruta de escaneo, se debe modificar el valor del campo *Height*.
5. A continuación se añadirán los vértices del polígono a escanear. Se podrán añadir tantos vértices como el usuario desee con un mínimo de tres vértices. Para añadir cada vértice, el operario debe pinchar sobre el mapa en la posición donde desee añadir cada vértice.
6. Para finalizar y cerrar el polígono se debe pinchar sobre el botón *Close Pattern*. Este paso genera la ruta de escaneo y la misión se mostrará sobre la escena.

7. El usuario puede guardar la misión creada para un uso futuro pinchando sobre el botón *Save*. Sobre la ventana emergente se debe elegir el nombre y la ubicación de guardado para el archivo de misión y pinchar sobre el botón *Save* para confirmar el guardado.
8. Finalmente, para validar la misión hay que pinchar sobre el botón *Submit*. Una ventana mostrará un aviso conforme la misión ha sido aceptada o rechazada.

El mapa utilizado puede ser cualquiera de los soportados por la aplicación. Si se desea utilizar un mapa local, este se puede cargar siguiendo el caso de uso 2 (Secc. 4.2). Sin embargo, si el usuario no dispusiese de una archivo de mapa local ni tampoco de conexión a internet y los mapas no estuvieran previamente cacheados, no se podría obtener un mapa válido sobre el que crear la misión. Por ello, la acción no sería realizable y habría que posponerla a una situación en la que se dispusiese de algún mapa.

Por otro lado, los vértices se pueden introducir también mediante los campos de latitud y longitud. Para ello es necesario abrir la ventana secundaria con la lista de los vértices del polígono. Esta ventana se puede abrir pinchando en el botón *Add Pattern* en el menú de creador de misiones por patrón. Sobre la ventana se encuentran los campos *Lat* y *Lon* junto al botón *Add* con los que se pueden añadir los vértices al polígono.

Si el usuario cometiese un error introduciendo alguno de los vértices del polígono, sería necesario pinchar el botón *Clear* para borrar el polígono y empezar de nuevo la creación del mismo.

Finalmente, la ruta de escaneo solo se puede crear si la aeronave ha sido correctamente caracterizada. Por ello, si la aeronave no ha sido configurada en la aplicación, durante el paso 6 saldrá un mensaje de error avisando al usuario que debe caracterizar a la aeronave antes de cerrar el polígono y crear la misión por patrón. La aeronave se puede caracterizar siguiendo el caso de uso 1 (Secc. 4.1) y a continuación, se puede continuar este caso de uso desde el paso 6.

Es importante destacar que al igual que sucedía con las misiones polilínea, no es posible crear más de una misión al mismo tiempo. Además, en caso de tener un archivo de misión abierto, cualquier cambio que se grabe de forma persistente pinchando sobre el botón *Save* sobrescribirá al archivo abierto y no creará un archivo de misión nuevo.

4.6. Caso de uso 6: Carga de una misión en fichero, envío y seguimiento de la misión.

Este caso de uso describe la carga de una misión en fichero a la aplicación, la validación y el envío de la misión, seguido de un posterior seguimiento de la misión. Son varios los requisitos asociados a este caso de uso. En primer lugar, la aplicación debe estar abierta y en ejecución. En segundo lugar, la conexión a la aeronave debe haber sido correctamente establecida siguiendo el caso de uso 4 (Secc. 4.4). Además, una misión debe haber sido creada y guardada en un fichero previamente. Los casos de usos 3 y 5 muestran como crear los distintos tipos de misión (Secc. 4.3 y 4.5).

En el Apéndice ?? se presenta este caso de uso en formato de tabla con los principales aspectos esquematizados. La secuencia normal de ejecución es la siguiente:

1. En primer lugar, se debe pinchar sobre *Mission* para desplegar dicha pestaña.

2. A continuación, para cargar una misión es necesario pinchar sobre el botón *Load*. Sobre la ventana emergente, el operario selecciona el archivo de misión deseado y pincha sobre el botón *Open* para confirmar la selección.
3. Para validar la misión se pincha sobre el botón *Submit*. Un aviso mostrará al usuario si la misión ha sido aceptada o rechazada. Si la misión es válida, la aplicación pasará a mostrar la pestaña *Checklist*.
4. Sobre la nueva pestaña se debe completar y validar cada caja de validación. Cada elemento corresponde a una medida de seguridad que se debe comprobar antes de un vuelo. Tras completar cada una de las comprobaciones, el usuario podrá pinchar sobre el botón *Send Mission* para el envío de la misión a la aeronave. Al pinchar sobre el botón se mostrará la pestaña *Follow*.
5. Para iniciar la misión es necesario armar la aeronave pinchando sobre el botón *Arm* y a continuación pinchar sobre el botón *Fly* para despegar.
6. Durante el seguimiento de vuelo el operario podrá observar los parámetros de vuelo pinchando sobre la opción *Attitude* que desplegará una ventana secundaria con los útiles de navegación.

A la hora de cargar la misión en la aplicación, si esta es una misión de patrón, la aeronave debe haber sido previamente caracterizada, de lo contrario no se podrá cargar la misión. Para caracterizar la aeronave correctamente se recomienda seguir el caso de uso 1 (Secc. 4.1).

Si durante el paso 3 la misión no ha sido aceptada, el usuario deberá crear o cargar otra misión válida. Los casos de uso 3 y 5 muestran la creación de misiones válidas (Secc. 4.3 y 4.5).

Finalmente, para poder enviar la misión a la aeronave la conexión tiene que haber sido establecida previamente. Si la conexión no ha sido establecida, un error se mostrará durante el paso 4. Para establecer la conexión se recomienda seguir el caso de uso 4 (Secc. 4.4).

Se adjunta un enlace de un vídeo ⁵ de un experimento real en el que se completa el caso de uso actual. Es importante destacar que algún aspecto menor puede diferir con la versión actual, pues en el momento de grabación del vídeo la aplicación se encontraba todavía en proceso de desarrollo.

4.7. Caso de uso 7: Interrupción de una misión y vuelta a casa.

Este caso de uso recoge los pasos necesarios para la interrupción de una misión y la vuelta a casa de la aeronave. Para que esta situación tenga lugar, la aplicación debe estar abierta, en ejecución, con conexión establecida y con una misión en curso. En el cuarto caso de uso (Secc. 4.4) se explica el procedimiento para establecer la conexión, mientras que en el sexto caso de uso (Secc. 4.6) se recogen los pasos para el inicio y el seguimiento de una misión en modo automático.

Este caso de uso se presenta en formato de tabla en el Apéndice ???. La secuencia de ejecución en condiciones normales es la siguiente:

⁵<https://youtu.be/tppmpdkG0CjM>

1. La pestaña *Follow* debe estar activa. Para ello es necesario pinchar en el menú con el mismo nombre en la ventana principal. En esta pestaña se pueden observar las actuaciones de la aeronave en todo momento.
2. Para iniciar la vuelta a casa, el operario debe pinchar sobre el botón *RTL*. Tras esto, la aeronave cambiará su modo de vuelo a *RTL* e iniciará la ruta de retorno.

El usuario puede decidir anular el retorno a casa. Para ello dispone de dos modos de vuelo. En primer lugar, pinchando sobre el botón *Loiter* se activaría el modo de vuelo *Hold* o *Loiter*, donde la aeronave mantendría la posición en el momento de activación. En segundo lugar, pinchando sobre el botón *Resume*, la aeronave volvería al modo de vuelo automático y continuaría con la misión interrumpida.

4.8. Caso de uso 8: Cambio de idioma de la aplicación.

Finalmente, el último caso de uso presentado muestra como cambiar dinámicamente el idioma del texto de la aplicación. La aplicación debe estar abierta y en ejecución. El Apéndice ?? incluye una tabla con el caso de uso esquematizado. La secuencia de ejecución en condiciones normales es la siguiente:

1. En primer lugar, es necesario acceder a la ventana secundaria de ejecución siguiendo la ruta *File - Settings - More Settings* en el menu de fichero de la aplicación.
2. Sobre la ventana de configuración desplegada se debe elegir en la lista desplegable *Language* uno de los idiomas disponibles. Para confirmar los cambios hay que pinchar sobre el botón *OK*.

Otra opción para acceder a la ventana de configuración es introduciendo el atajo *Ctrl+S*. Cabe destacar que con idiomas distintos al idioma base (inglés), es posible que la aplicación contenga errores en la traducción o funciones no traducidas.

Capítulo 5

Conclusiones y líneas futuras

Este capítulo reúne las conclusiones extraídas tras la realización del proyecto y describe posibles líneas futuras de trabajo que surgen a partir del mismo. Estas líneas de trabajo suponen una vía de desarrollo para la elaboración de una segunda versión más completa de la aplicación.

5.1. Conclusiones

El desarrollo concluye en una primera versión de UAVCommander. Estos meses de trabajo han permitido que la aplicación haya avanzado desde un punto inicial de desarrollo hasta en un estado actual de prototipo. Esta evolución se puede observar en la lista de reproducción¹ donde se han subido los vídeos con las diferentes versiones y funciones que se han ido lanzando. Los vídeos demuestran como los objetivos descritos durante la Sección 1.3 se han visto satisfechos.

El objetivo principal era el desarrollo software de una estación tierra. A lo largo de los capítulos de esta memoria se han explicado las diferentes funcionalidades y logros conseguidos. Si bien es cierto que la aplicación se encuentra en fase de maduración, ya permite operar con aeronaves reales, tanto con ala fija como con ala rotatoria. Son varios los vuelos de prueba que se han realizado, como se ha podido observar a lo largo del capítulo de resultados (Cap. 4). Por ello, se concluye que el objetivo principal ha sido ampliamente satisfecho con un software funcional que sirve de base para futuras versiones que incluyan otras mejoras.

Entre los objetivos secundarios se encontraban una serie de requisitos funcionales. Estas funciones también han sido satisfechas, se explican a continuación:

- El primer objetivo secundario era dotar a la aplicación de una caracterización de la aeronave y carga de pago. Para dar soporte a esta funcionalidad, se ha desarrollado una pestaña dentro de la interfaz gráfica de usuario que recoge los diferentes parámetros de configuración. Además, se ha dotado a la aplicación de un sistema de carga y guardado en ficheros para la reutilización de configuraciones. Esto agiliza mucho su uso, pues el usuario solo realizaría la caracterización una vez y podría reutilizar el archivo de configuración entre distintas ejecuciones de la aplicación.
- El segundo objetivo secundario consistía en poder crear misiones automáticas desde la aplicación. En la segunda pestaña de la aplicación se incluyen un sistema de

¹<https://www.youtube.com/playlist?list=PLGIx46StCA-QAXmeQp5omNhllGJ3CMjcO>

mapas y un creador de misiones para dar soporte a esta funcionalidad. Son varios los mapas soportados como se ha explicado durante la Sección 3.3. Además, las misiones creadas pueden ser de dos tipos, misiones polilínea y misiones por patrón. La creación de misiones y sus tipos se explica a lo largo de la Sección 3.4.

- El tercer objetivo secundario recogía la necesidad de disponer de una lista de comprobaciones de seguridad previas al vuelo. Esta lista se encuentra en la tercera pestaña de la aplicación.
- El cuarto objetivo secundario proponía un sistema de seguimiento sobre la aeronave. La cuarta pestaña de la aplicación cumplimenta este requisito. El seguimiento permite visualizar la posición de la aeronave sobre el mapa en todo momentos junto con una serie de datos de telemetría y navegación.
- Finalmente, el quinto objetivo secundario incluye la posibilidad de un análisis de datos post-vuelo. La quinta pestaña de la aplicación proporciona este servicio, donde los logs de vuelo se pueden descargar y eliminar.

Las diferentes funcionalidades secundarias se ven cumplimentadas en cada una de las pestañas de la aplicación. Por tanto, se concluye que los requisitos funcionales se ven cumplimentados dentro del conjunto de la aplicación.

Por último, la aplicación se ha diseñado e implementado en todo momento primando la simplicidad y reduciendo al mínimo la interacción con el operario. Con esta visión se pretende facilitar el aprendizaje y el uso de la aplicación, junto con la reducción de los errores que puedan surgir por motivo de equivocaciones humanas. Además, se ha tratado de desarrollar siguiendo el estándar en este tipo de software manteniendo el enclave de sencillez donde las principales soluciones del mercado pecan a la hora de incluir multitud de opciones y configuraciones. Así pues, el operario solo interacciona con la aplicación cuando es estrictamente necesario, y donde los automatismos se tratan de llevar a máximos.

5.2. Líneas futuras

Este trabajo desarrolla un primer prototipo funcional de una estación de tierra. Son muchos los aspectos a mejorar y las funciones a incluir que se han detectado a lo largo del desarrollo. A continuación se incluyen una lista de posibles mejoras futuras:

- Descarga de teselas del mapa mediante multihilo para agilizar la velocidad de navegación a través del mapa.
- Aumentar el número de comprobaciones sobre una misión creada para aumentar la seguridad de la misma. Algún ejemplo podría ser: comprobaciones de radio mínimo de giro, comprobaciones en la altitud del terreno sobre la ruta a realizar, etc.
- Añadir nuevos tipos de misión como misiones de corredor.
- Permitir un tamaño variable en la escena lo que permitiría modificar libremente el tamaño de la ventana. Para ello, sería necesario incluir un tamaño de teselas variable.
- Modificación dinámica de los puntos de paso que permita cambiar ciertos valores como el tipo de punto, el orden, la altura, etc.

- Habilitar funciones de *deshacer* o *rehacer* durante la creación de misiones. Mitigar así los errores introducidos al añadir puntos.
- Actualización automática de los puntos visibles sobre la lista de puntos.
- Mejorar visualmente la aplicación. Substituir texto por iconos, habilitar modo oscuro, etc.
- Dar soporte a la aplicación en otros idiomas como portugués, francés, etc.

Bibliografía

- [1] Comandos *MAV_CMD*, author = Dronecode Project, Inc., howpublished = https://mavlink.io/en/messages/common.html#mav_commands, note = Último acceso: 26-02-2020.
- [2] ArduPilot. Pymavlink. <https://github.com/ArduPilot/pymavlink>. Último acceso: 10-02-2020.
- [3] Antonio Barrientos, J Del Cerro, P Gutiérrez, R San Martín, A Martínez, and C Rossi. Vehículos aéreos no tripulados para uso civil. tecnología y aplicaciones. *Universidad politécnica de Madrid, Madrid*, 2007.
- [4] Aníbal Ollero Baturone. *Robótica: manipuladores y robots móviles*. Marcombo, 2005.
- [5] Diego Jiménez Bravo. Drones con protocolo mavlink en el entorno jderobot. <https://github.com/RoboticsLabURJC/2016-tfg-Diego-Jimenez>. Último acceso: 24-02-2020.
- [6] José Antonio Fernández Casillas. Navegación por posición para un avión autónomo con jderobot. <https://github.com/RoboticsLabURJC/2014-pfc-JoseAntonio-Fernandez>. Último acceso: 24-01-2020.
- [7] Marek Cel and Aitor Martinez (JdeRobot). Qflightinstruments. <https://github.com/JdeRobot/ThirdParty/tree/master/qflightinstruments>. Último acceso: 10-02-2020.
- [8] Alex Clark and Contributors. Pil. <https://pillow.readthedocs.io/en/stable/>. Último acceso: 10-02-2020.
- [9] The Qt Company. Qt5. <https://www.qt.io/>. Último acceso: 10-02-2020.
- [10] Ministerio de Fomento. Instituto geográfico nacional. <https://www.ign.es/web/ign/portal>. Último acceso: 10-02-2020.
- [11] Jonathan Ruiz de Garibay Pascual. Robótica: Estado del arte. *Universidad de Deuston. Número. Fecha*, page 54, 2006.
- [12] Grupo de robótica de la URJC. Twitter @roboticslaburjc. <https://twitter.com/RoboticsLabURJC?lang=es>. Último acceso: 24-01-2020.
- [13] NumPy developers. Numpy. <https://numpy.org/>. Último acceso: 10-02-2020.
- [14] Inc. Dronecode Project. Bucle de control de un uav. <https://dev.px4.io/master/en/concept/architecture.html>. Último acceso: 24-02-2020.

- [15] Inc. Dronecode Project. Comando *MAV_CMD_DO_CHANGE_SPEED*. https://mavlink.io/en/messages/common.html#MAV_CMD_DO_CHANGE_SPEED. Último acceso: 26-02-2020.
- [16] Inc. Dronecode Project. Documentación de mavlink con los mensajes disponibles. <https://mavlink.io/en/messages/common.html>. Último acceso: 26-02-2020.
- [17] Inc. Dronecode Project. Formato de ficheros. https://mavlink.io/en/file_formats/#mission_plain_text_file. Último acceso: 22-02-2020.
- [18] Inc. Dronecode Project. Mavlink. <https://mavlink.io/en/>. Último acceso: 11-02-2020.
- [19] Inc. Dronecode Project. Mensaje *COMMAND_LONG*. https://mavlink.io/en/messages/common.html#COMMAND_LONG. Último acceso: 26-02-2020.
- [20] Inc. Dronecode Project. Mensaje *HEARTBEAT*. <https://mavlink.io/en/messages/common.html#HEARTBEAT>. Último acceso: 26-02-2020.
- [21] Inc. Dronecode Project. Mensaje *MISSION_ITEM*. https://mavlink.io/en/messages/common.html#MISSION_ITEM. Último acceso: 26-02-2020.
- [22] Inc. Dronecode Project. Protocolo de misiones de mavlink. https://mavlink.io/en/services/mission.html#uploading_mission. Último acceso: 22-02-2020.
- [23] Inc. Dronecode Project. Px4. <https://px4.io/>. Último acceso: 11-02-2020.
- [24] Real Academia Española. Diccionario de la lengua española, 23.^º ed. <https://dle.rae.es>. Último acceso: 02-02-2020.
- [25] Open Source Robotics Foundation. Gazebo. <http://gazebosim.org/>. Último acceso: 11-02-2020.
- [26] Python Software Foundation. Collections python library. <https://docs.python.org/3/library/collections.html>. Último acceso: 10-02-2020.
- [27] Python Software Foundation. Datetime python library. <https://docs.python.org/3/library/datetime.html>. Último acceso: 10-02-2020.
- [28] Python Software Foundation. Json python library. <https://docs.python.org/3/library/json.html>. Último acceso: 10-02-2020.
- [29] Python Software Foundation. Math python library. <https://docs.python.org/3/library/math.html>. Último acceso: 10-02-2020.
- [30] Python Software Foundation. Pyhton. <https://www.python.org/>. Último acceso: 09-02-2020.
- [31] Python Software Foundation. Threading python library. <https://docs.python.org/3/library/threading.html>. Último acceso: 10-02-2020.
- [32] Python Software Foundation. Time python library. <https://docs.python.org/3/library/time.html>. Último acceso: 10-02-2020.

- [33] Python Software Foundation. Urllib. <https://docs.python.org/3/library/urllib.html>. Último acceso: 10-02-2020.
- [34] Geodrone. Geodrone mapper. <http://geodrone.es/>. Último acceso: 24-01-2020.
- [35] Inc. GitHub. Github. <https://github.com/>. Último acceso: 26-02-2020.
- [36] Klokan Technologies GmbH and OSM community. Open map tiles. <https://openmaptiles.org/about/>. Último acceso: 10-02-2020.
- [37] Riverbank Computing Limited. Pyqt5. <https://riverbankcomputing.com/software/pyqt/intro>. Último acceso: 10-02-2020.
- [38] Google LLC. Google map server. <https://developers.google.com/maps/documentation?hl=es>. Último acceso: 10-02-2020.
- [39] Canonical Ltd. Ubuntu 18.04.3 lts. <https://ubuntu.com/download/desktop>. Último acceso: 09-02-2020.
- [40] FM Sánchez Martín, F Millán Rodríguez, J Salvador Bayarri, J Palou Redorta, F Rodríguez Escobar, S Esquena Fernández, and H Villavicencio Mavrich. Historia de la robótica: de arquitas de tarento al robot da vinci (parte i). *Actas Urológicas Españolas*, 31(2):69–76, 2007.
- [41] Instituto Geográfico Nacional. Plan nacional de ortofotografía aérea. <https://pnoa.ign.es/>. Último acceso: 10-02-2020.
- [42] University of Colorado Boulder. Pseudo-código con la implementación del protocolo de misiones de mavlink. <https://www.colorado.edu/recuv/2015/05/25/mavlink-protocol-waypoints>. Último acceso: 22-02-2020.
- [43] OSGeo. Gdal. <https://www.osgeo.org/projects/gdal/>. Último acceso: 10-02-2020.
- [44] Jorge Vela Peña. Despegue, navegación y aterrizaje visuales de un drone usando jderobot. <https://github.com/RoboticsLabURJC/2016-tfg-jorge-vela>. Último acceso: 24-02-2020.
- [45] Pedro Arias Pérez. Repositorio github del trabajo. <https://github.com/RoboticsLabURJC/2019-tfg-pedro-arias>. Último acceso: 26-02-2020.
- [46] 3DR Robotics. 3dr solo drone. <https://newthreedee.wpengine.com/>. Último acceso: 11-02-2020.
- [47] Conyca S.L. Web. <http://www.conyca.es/>. Último acceso: 24-01-2020.
- [48] JetBrains s.r.o. Pycharm. <https://www.jetbrains.com/es-es/pycharm/>. Último acceso: 26-02-2020.
- [49] Ardupilot Development Team and Community. Ardupilot. <https://ardupilot.org/>. Último acceso: 11-02-2020.