



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

UAVCOMMANDER, ESTACIÓN TERRESTRE PARA
OPERACIÓN AUTOMÁTICA DE UAVs.

Autor: Pedro Arias Pérez
Tutor: José María Cañas Plaza
Cotutor: Diego Martín Martín

*A miña irmá Raquel,
o meu principal apoio e motivación.*

Resumen

Los robots aéreos forman parte de nuestro día a día con aplicaciones en muchos ámbitos de nuestra vida. Este trabajo proporciona una solución software para la operación automática y remota de aeronaves. Esta solución consiste en una aplicación de escritorio tipo estación terrestre que permite la programación de misiones para robots aéreos. La aplicación, UAVCommander, se ha diseñado e implementado en lenguaje Python. Entre las funciones del software destacan la especificación de misiones de navegación polilínea (secuencia de puntos de paso) así como de misiones por patrón, en las que se especifica un área a barrer, en forma de polígono, y la aplicación calcula automáticamente la ruta a seguir de modo que se recorra todo ese área. Además incluye la caracterización paramétrica de la aeronave, la comprobación de requisitos antes de cargar esas misiones en la aeronave y el seguimiento en vuelo de la ejecución de esas misiones junto a datos de navegación. La aplicación programada se ha validado experimentalmente, tanto con drones simulados como con una aeronave real en un campo de vuelo. Además de las pruebas unitarias, el sistema final ha funcionado satisfactoriamente en las pruebas integrales.

Palabras clave— Estación de tierra, UAVS, robótica, robótica aérea, drones, MAVLink.

Agradecimientos

Tras cinco años, finalizo, con esta memoria, mi etapa universitaria como estudiante de grado. Ha sido un trayecto emocionante en el que me he cruzado con grandes profesores, compañeros y amigos. Es por eso, por lo que me gustaría agradecer a la Escuela Técnica Superior de Ingeniería de Telecomunicación, a la Universidad Rey Juan Carlos y a todos sus profesionales por la atención y formación ofrecida.

Además, en particular, me gustaría dar las gracias a...

A mi tutor José María, por darme la oportunidad de participar en un proyecto tan apasionante y en el que he aprendido tanto. También por su ayuda y su comprensión en todo momento.

A Diego Martín y José Antonio Fernández, por su inestimable ayuda que me ha permitido avanzar ante cualquier problema.

A mi familia, en especial a mis padres, por dedicar gran parte de su vida a mi educación. También a mi hermana Raquel, por todo nuestro amor en silencio. Sin vuestra ayuda no habría podido llegar hasta aquí.

A Carme, por toda la inspiración y motivación que me aportas, y en particular, por ayudarme con el diseño de las figuras que esta memoria contiene.

A mis compañeros de viaje; Andrés, Juan, Raquel, Irene y Vladi, por lo sufrido y por lo disfrutado por partida doble. También a los que se han convertido en grandes amigos durante estos años, Paula, Carmen, Álvaro, Pauliña, Muffin y Anaïs.

A mis okupas; Luci, Silvi, Alejandro, Castell, Javi y Miranda, lo mejor de Madrid siempre estará con vosotros al lado.

A Juan, Dani, Sunil, Verto y Ana, porque a pesar de que pasen los años, seguimos en contacto.

¡Gracias a todos!

Índice general

Resumen	v
Agradecimientos	vii
Índice general	viii
Índice de figuras	x
Índice de tablas	xii
Índice de códigos	xiii
Acrónimos	xv
1 Introducción	1
1.1. Robótica Aérea	1
1.2. Problema y motivación	4
1.3. Objetivos	5
1.4. Estructura de la memoria	6
2 Infraestructura	7
2.1. Lado Tierra	7
2.2. Lado Aire	8
2.3. Protocolo de comunicaciones	10
3 Estación de Tierra, diseño e implementación	13
3.1. Diseño	13
3.2. Interfaz Gráfico de Usuario	17
3.3. Sistema de Mapeado	19
3.4. Creador de Misiones	23
3.5. MAVLink Driver	27
4 Manual de usuario y casos de uso	41
4.1. Manual de Usuario	41
4.2. Casos de uso	54
5 Validación Experimental	67
5.1. Vuelo en simulador con misiones polilínea.	68
5.2. Vuelo real con misiones polilínea.	68
5.3. Vuelo en simulador con misiones por patrón.	70

5.4. Vuelo real con misiones por patrón.	71
6 Conclusiones y líneas futuras	73
6.1. Conclusiones	73
6.2. Líneas futuras	74
Bibliografía	77

Índice de figuras

1.1.	Clasificación de los UAV [2].	2
1.2.	Tipos de UAV.	3
1.3.	Principales aplicaciones en robótica aérea.	4
1.4.	Geodrone Mapper utilizado en misiones de cartografía y fotografía aérea.	5
2.1.	Bucle de control de un UAV [13].	8
2.2.	Cuacóptero simulado sobre Gazebo9.	9
2.3.	Esquema de PX4 sobre SITL [24].	9
2.4.	Aeronaves utilizadas durante el proyecto.	10
3.1.	Esquema general del sistema.	14
3.2.	Capas que componen la aplicación.	15
3.3.	Esquema de clases del sistema.	16
3.4.	Ventana principal con la pestaña <i>Setup</i> activa.	18
3.5.	Ventana secundaria de configuración.	19
3.6.	Estructura vistas-escena.	20
3.7.	Esquema de vista en mosaico.	21
3.8.	Diferentes teselas disponibles para visualizar el mapa.	21
3.9.	Secuencia de llamadas para obtener una escena concreta.	22
3.10.	Esquema de conversión de coordenadas.	23
3.11.	Tipos de misión disponibles.	25
3.12.	Diagrama del protocolo de misiones de MAVLink.	30
3.13.	Ventana secundaria de sensores.	35
3.14.	Diagrama del control de vuelo.	37
3.15.	Diagrama del protocolo de comandos de MAVLink.	38
4.1.	Menú de fichero.	42
4.2.	Panel de navegación.	42
4.3.	Pestaña <i>Setup</i>	43
4.4.	Pestaña <i>Mission</i>	44
4.5.	Creador de misiones polilínea.	45
4.6.	Creador de misiones por patrón.	46
4.7.	Pestaña <i>Checklist</i>	47
4.8.	Pestaña <i>Follow</i>	48
4.9.	Pestaña <i>log</i>	49
4.10.	Ventana secundaria de configuración.	50
4.11.	Ventana secundaria de carga de mapas locales.	51
4.12.	Ventana secundaria de puntos de paso.	52
4.13.	Ventana secundaria de sensores.	53

4.14. Caso de uso 1	56
4.15. Caso de uso 2	57
4.16. Caso de uso 3	59
4.17. Caso de uso 4	60
4.18. Caso de uso 5	63
4.19. Caso de uso 6	64
4.20. Caso de uso 7	65
4.21. Caso de uso 8	66
5.1. Campo de vuelo de Seseña	67
5.2. Simulación de vuelo para misiones polilínea	68
5.3. Vídeo con la prueba de vuelo para misiones polilínea	69
5.4. Simulación de vuelo para misiones por patrón	70
5.5. Vídeo con la prueba de vuelo para misiones por patrón	71

Índice de tablas

3.1. Mensaje <i>HEARTBEAT</i> [20].	28
3.2. Mensaje <i>MISSION_ITEM</i> [21].	30
3.3. Comparación entre parámetros de tres comandos <i>MAV_CMD_</i>	31
3.4. Parámetros del comando <i>MAV_CMD_DO_SET_CAM_TRIGG_DIST</i>	31
3.5. Diccionario de mensajes soportados por cada autopiloto.	35
3.6. Mensaje <i>COMMAND_LONG</i> [19].	38
3.7. Comando <i>MAV_CMD_DO_CHANGE_SPEED</i> [14].	39
4.1. Menú de fichero.	42
4.2. Panel de navegación.	42
4.3. Pestaña <i>Setup</i>	43
4.4. Pestaña <i>Mission</i>	44
4.5. Creador de misiones polilínea.	45
4.6. Creador de misiones por patrón.	46
4.7. Pestaña <i>Checklist</i>	47
4.8. Pestaña <i>Checklist</i>	48
4.9. Pestaña <i>log</i>	49
4.10. Ventana secundaria de configuración.	50
4.11. Ventana secundaria de carga de mapas locales.	51
4.12. Ventana secundaria de puntos de paso.	52
4.13. Ventana secundaria de sensores.	53

Índice de códigos

1.	Mensaje MAVLink tipo.	11
2.	Comando MAVLink tipo.	12
3.	Estructura de pestañas en la ventana principal.	18
4.	Filtro de eventos de la escena.	24
5.	Algoritmo de barrido para las misiones de patrón.	26
6.	Establecimiento de conexión.	28
7.	Handler de conexión.	29
8.	Implementación del protocolo de misión de MAVLink.	33
9.	Procedimiento <i>refreshAPMnavdata()</i> , encargado de leer y actualizar los datos de navegación enviados por la aeronave.	34
10.	Procedimiento <i>refreshAPMPose3D()</i> , encargado de leer y actualizar los datos de posición enviados por la aeronave.	36
11.	Activación de modos de vuelo.	38
12.	Mensaje <i>COMMAND_LONG</i> para el cambio de velocidad de vuelo.	39
13.	Armado y desarmado de la aeronave.	39

Acrónimos

Entrada	Descripción
ECW	Enhanced Compressed Wavelet.
GeoTIFF	Georeferenced Tagged Image File Format.
GMP	Google Maps Platform.
GPS	Global Positioning System.
GTS	Google Tile Server.
GUI	Graphical User Interface.
IDE	Integrated Development Environment.
IGN	Instituto Geográfico Nacional.
IMU	Inertial Measurement Unit.
JSON	JavaScript Object Notation.
LTS	Long Term Support.
MAVLink	Micro Air Vehicle Link.
OMT	Open Map Tiles.
OSGeo	Open Source Geospatial Foundation.
OSM	Open Street Maps.
PIL	Python Image Library.
PNOA	Plan Nacional de Ortofotografía Aérea.
QFI	QFlight Instruments.
RTL	Return To Launch.
SITL	Software In The Loop.
TIFF	Tagged Image File Format.
UAV	Unmanned Aerial Vehicle.

Entrada Descripción

UAVS Unmanned Aircraft Vehicle System.

URJC Universidad Rey Juan Carlos.

URL Uniform Resource Locator.

WMS Web Map Services.

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. Robótica Aérea

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 [43]. 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 [3].

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 [43]. El término ha evolucionado mucho desde sus inicios, hoy entendemos la robótica como la 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 [25]. 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 [10], 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 [2].

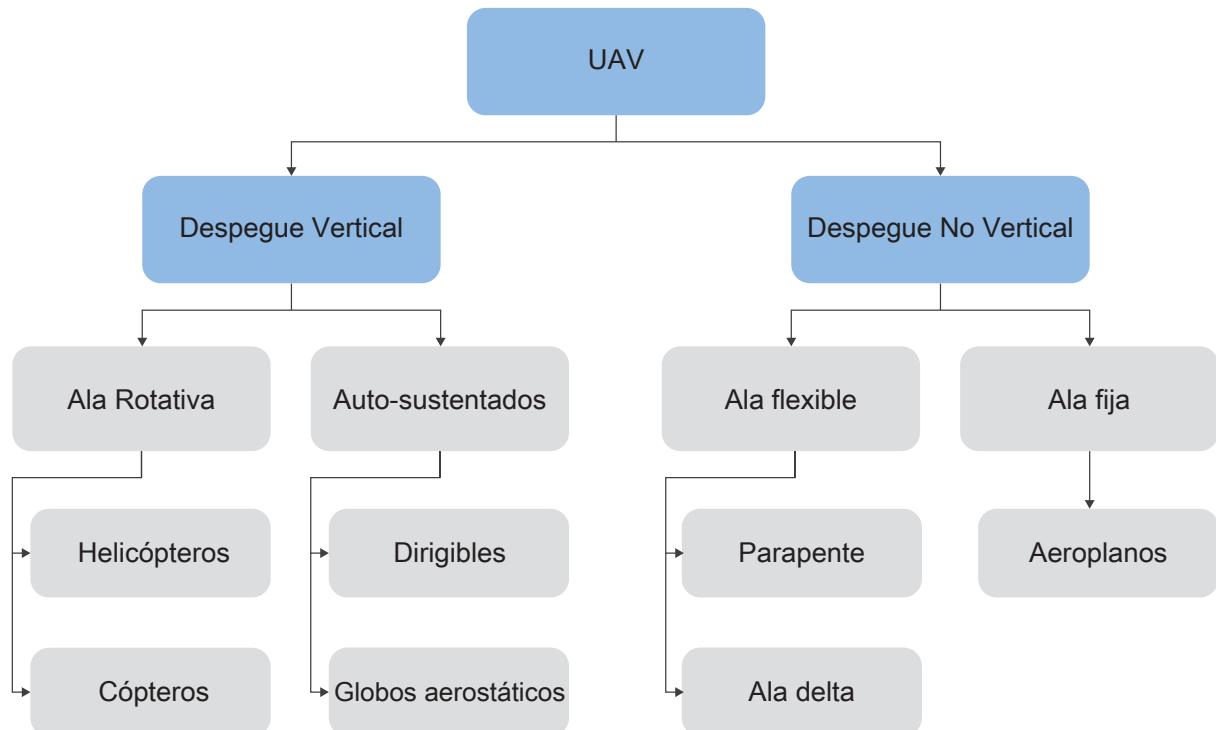


Figura 1.1: Clasificación de los UAV [2].

A la hora de establecer una clasificación de los UAV es posible atender a diferentes criterios. Siguiendo la clasificación propuesta por *Barrientos et al.* [2] 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.



(a) Aeroplano.



(b) Multicóptero.

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 Figura 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. Típicamente, el segmento tierra se compone de un ordenador donde se ejecuta un software tipo estación de tierra, mientras que el segmento aire se compone por una aeronave. La comunicación entre ellos se realiza a través de un protocolo de comunicaciones.



(a) Transporte de mercancías [4].



(b) Spot publicitario [48].



(c) Agricultura [48].



(d) Salvamento [4].

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 del desarrollo del grupo de investigación de robótica (@RoboticsLabURJC [11]) de la Universidad Rey Juan Carlos (URJC) y la asociación JdeRobot [38], con el proyecto *ROSpilot, software para operación y navegación de un UAV*, en la cual he tenido la suerte de 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* [35], 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 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 presentar el problema a resolver en la sección anterior (Sección 1.2), en esta sección se quiere aclarar cuáles son los objetivos perseguidos.

El objetivo principal es proporcionar una solución software que permita la operación automática remota a drones. 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* [5].

En el proyecto 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, se persigue 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 este capítulo introductorio 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. A continuación incluye, el problema abordado en este trabajo y la motivación del mismo, es decir, ¿qué se trata de resolver con este trabajo?, y ¿por qué surge la necesidad del mismo?. Se incluyen también los objetivos principales y secundarios que ha de cumplir la solución desarrollada. Por último, se encuentra la sección actual que muestra una visión global de lo que se presenta en esta memoria.

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 una primera sección se explica el segmento tierra y su composición. Por otro lado, en la siguiente sección se detalla el segmento aire y sus componentes. Finalmente, el capítulo se centra en el protocolo de comunicaciones entre ambos lados, tierra y aire.

En el Capítulo 3 se desarrolla el diseño y la implementación de la solución del problema. En una primera sección se explica detalladamente el diseño elegido y se explican también varias decisiones relevantes a la hora de seleccionar el diseño. En las secciones sucesivas se describen la implementación de cada uno de los bloques de código identificados.

En el Capítulo 4 se presentan los resultados obtenidos del desarrollo. En una primera sección se muestra un manual de usuario con la apariencia y las funcionalidades de la aplicación. En la segunda sección del capítulo se añaden una serie de casos de uso que tratan de ejemplificar posibles situaciones reales y los procedimientos a realizar por el usuario para completar dicha tarea. 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. En el Capítulo 5 se aportan diferentes validaciones experimentales que demuestran el funcionamiento y la validez del software. Estos experimentos consisten en distintas pruebas de vuelo en real.

Finalmente, en el Capítulo 6 se recogen las conclusiones extraídas con la finalización del trabajo y se evalúan los distintos objetivos iniciales propuestos. A mayores, se exponen una serie de posibles vías de desarrollo futuro y de mejoras para la aplicación.

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 desgranar 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.

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)** [42]. 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 [31]. Este 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** [39]. PyQt5 es un *binding* de la biblioteca gráfica Qt5 [8] para el lenguaje de programación Python. Qt es un entorno multiplataforma escrito en C++ que permite el desarrollo de interfaces gráficas de usuario de forma sencilla.

Otra biblioteca relevante es **PyMavlink** [1], 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* [40] u *Open Map Tiles* [37] a través de librerías como *urllib* para el manejo de URLs [34] o *PIL (Python Image Library)* y *Pillow*, para el manejo de imágenes [7].

Por otro lado, para la lectura de imágenes locales geo-referenciadas procedentes del Instituto Geográfico Nacional (IGN) [9] a través del Plan Nacional de Ortofotografía Aérea (PNOA) [44] se ha utilizado la librería **GDAL** [46]. 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* [30], *numpy* [12], *threading* [32], *json* [29], *collections* [27], *datetime* [28], *time* [33] o *qfi* [6].

A la hora de desarrollo se ha utilizado la plataforma GitHub [36] y el entorno de desarrollo PyCharm [51]. GitHub es plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. 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 es el drone. Se distinguen dos posibilidades, el drone puede ser real o simulado. Ambas posibilidades siguen un bucle de control similar. La Figura 2.1 representa el bucle estándar de control disponible en los autopilotos más comunes.

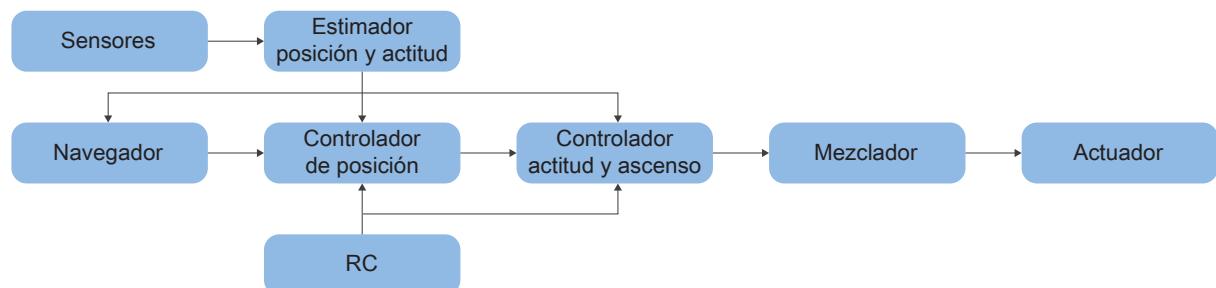


Figura 2.1: Bucle de control de un UAV [13].

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.

El uso de drone real se ha limitado a las últimas fases de desarrollo de este proyecto, que muy frecuentemente se ha visto sustituido por un drone simulado (ver Fig. 2.2). 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.

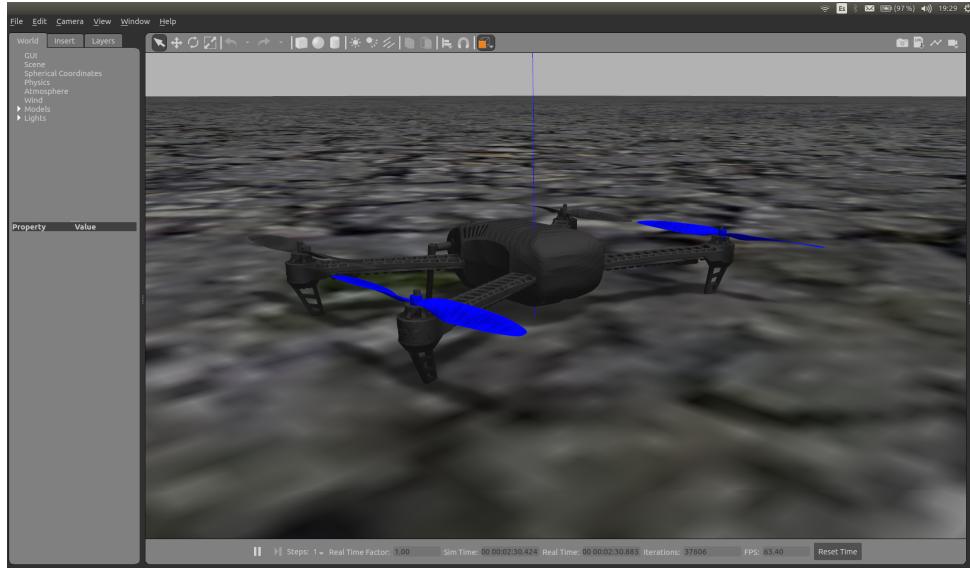


Figura 2.2: Cuacóptero simulado sobre Gazebo9.

El firmware utilizado como controladora es **PX4** [24], cuyo esquema de SITL se puede observar en la Figura 2.3. PX4 es un software de control de vuelo de código abierto para drones y otros vehículos no tripulados. PX4 proporciona un estándar para ofrecer soporte de hardware de drones, lo que permite que un ecosistema construya y mantenga hardware y software de forma escalable.

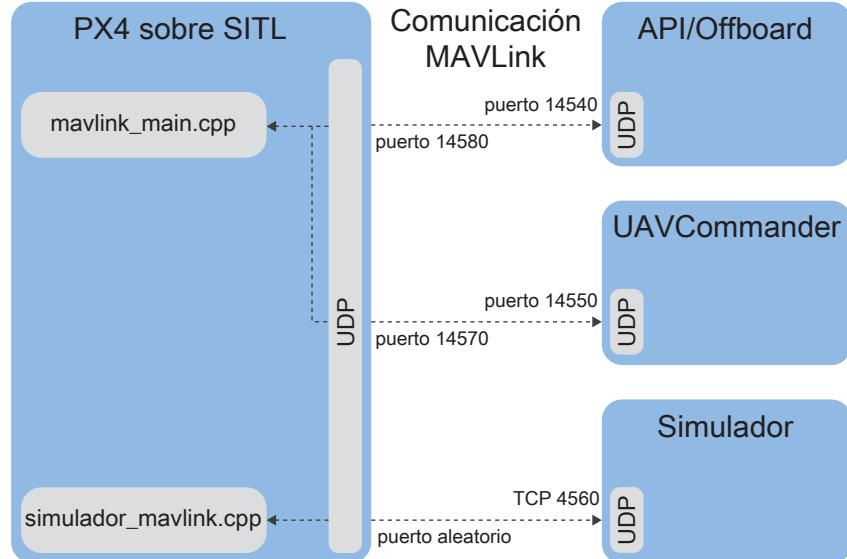


Figura 2.3: Esquema de PX4 sobre SITL [24].

Además, la simulación se puede conectar con un simulador para hacer más completa la experiencia y mostrar el vuelo en un entorno de pruebas con elementos del mundo real. Entre los distintos simuladores se ha elegido Gazebo [26]. Este simulador de código abierto es por excelencia el más utilizado en aplicaciones de robótica y visión artificial. Debido a su gestión

abierta permite la integración de múltiples vehículos, mundos, sensores, físicas, etc. La versión utilizada durante el proyecto es **Gazebo9**.

La aeronave real puede ser de diversa índole, como ya se ha visto en el capítulo anterior. A lo largo de este proyecto se prevé utilizar:

- **3DR Solo Drone:** Cuadricóptero fabricado por 3DR Robotics [49] (Fig. 2.4a). Posee una controladora con firmware Pixhawk v1 [41] con Ardupilot [52] *flasheado*, 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 esté previsto en el software.



(a) 3DR Solo Drone.

Figura 2.4: Aeronaves utilizadas durante el proyecto.

2.3. Protocolo de comunicaciones

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

Los mensajes son el objeto más pequeño de intercambio de información del protocolo. Son muchos los mensajes existentes y de diversa utilidad. Los diferentes mensajes de MAVLink se

pueden observar en su documentación [16]. Un ejemplo de un mensaje MAVLink se puede observar en el Código 1.

```

1 <message id="54" name="SAFETY_SET_ALLOWED_AREA">
2   <description>Set a safety zone (volume), which is defined by two corners of a cube.
3   This message can be used to tell the MAV which setpoints/waypoints
4   to accept and which to reject. Safety areas are often enforced
5   by national or competition regulations.</description>
6   <field type="uint8_t" name="target_system">System ID</field>
7   <field type="uint8_t" name="target_component">Component ID</field>
8   <field type="uint8_t" name="frame" enum="MAV_FRAME">Coordinate frame. Can be either
9       global, GPS, right-handed with Z axis up or local,
10      right handed, Z axis down.</field>
11  <field type="float" name="p1x" units="m">x position 1 / Latitude 1</field>
12  <field type="float" name="p1y" units="m">y position 1 / Longitude 1</field>
13  <field type="float" name="p1z" units="m">z position 1 / Altitude 1</field>
14  <field type="float" name="p2x" units="m">x position 2 / Latitude 2</field>
15  <field type="float" name="p2y" units="m">y position 2 / Longitude 2</field>
16  <field type="float" name="p2z" units="m">z position 2 / Altitude 2</field>
17 </message>
```

Código 1: Mensaje MAVLink tipo.

Existe un tipo especial de mensajes que encapsulan a los comandos, órdenes a ejecutar por la aeronave. Los comandos pueden ser de tres tipos: de navegación, de acción y de condición. Los comandos utilizados en el protocolo se pueden comprobar también en la documentación de MAVLink [16]. Un ejemplo de comando MAVLink se puede observar en el Código 2.

Los enumerados se usan para definir valores con nombre que se pueden usar como opciones en los mensajes, por ejemplo, para representar errores, estados o modos. Cada enumerado tiene un atributo de nombre obligatorio y puede contener una serie de elementos de entrada (con nombres exclusivos de enumeración) para los valores admitidos. Al igual que los mensajes y los comandos, los enumerados disponibles se pueden comprobar en la documentación de MAVLink [16].

Finalmente, los microservicios representan protocolos de alto nivel que los sistemas MAVLink pueden adoptar para una mejor interacción. Los microservicios se utilizan para intercambiar muchos tipos de datos, incluidos: parámetros, misiones, trayectorias, imágenes y otros archivos. Los datos pueden ser mucho más grandes de lo que cabe en un solo mensaje, por lo que los microservicios definirán cómo se dividen y se vuelven a ensamblar los datos, y cómo garantizar que los datos perdidos se vuelvan a transmitir. Otros microservicios proporcionan reconocimiento de comandos, informes de errores, etc. Los diferentes microservicios se definen en la documentación de MAVLink [22].

```
1 <enum name="MAV_CMD">
2     <description>Commands to be executed by the MAV. They can be executed on user
3         request, or as part of a mission script. If the action is used in
4         a mission, the parameter mapping to the waypoint/mission message is
5         as follows: Param 1, Param 2, Param 3, Param 4, X: Param 5, Y:Param 6,
6         Z:Param 7. This command list is similar what ARINC 424 is for
7         commercial aircraft: A data format how to interpret waypoint/mission
8         data.</description>
9     <entry value="16" name="MAV_CMD_NAV_WAYPOINT">
10        <description>Navigate to waypoint.</description>
11        <param index="1">Hold time in decimal seconds. (ignored by fixed wing, time
12            to stay at waypoint for rotary wing)</param>
13        <param index="2">Acceptance radius in meters (if the sphere with this radius
14            is hit, the waypoint counts as reached)</param>
15        <param index="3">0 to pass through the WP, if &gt; 0 radius in meters to pass
16            by WP. Positive value for clockwise orbit, negative value for
17            counter-clockwise orbit. Allows trajectory control.</param>
18        <param index="4">Desired yaw angle at waypoint (rotary wing). NaN for
19            unchanged.</param>
20        <param index="5">Latitude</param>
21        <param index="6">Longitude</param>
22        <param index="7">Altitude</param>
23    </entry>
24    ...
25 </enum>
```

Código 2: Comando MAVLink tipo.

Capítulo 3

Estación de Tierra, diseño e implementación

Este capítulo aborda el diseño e implementación del software. En primer lugar se explica 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 explicarán 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. En la Figura 3.1 se presenta un esquema general que representa la arquitectura del sistema. El esquema muestra tanto el hardware como el software que integra el sistema, y da una visión global inicial del funcionamiento de la aplicación dentro del sistema completo.

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 distintas 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 detallan en profundidad aspectos relacionados con la implementación de este bloque.

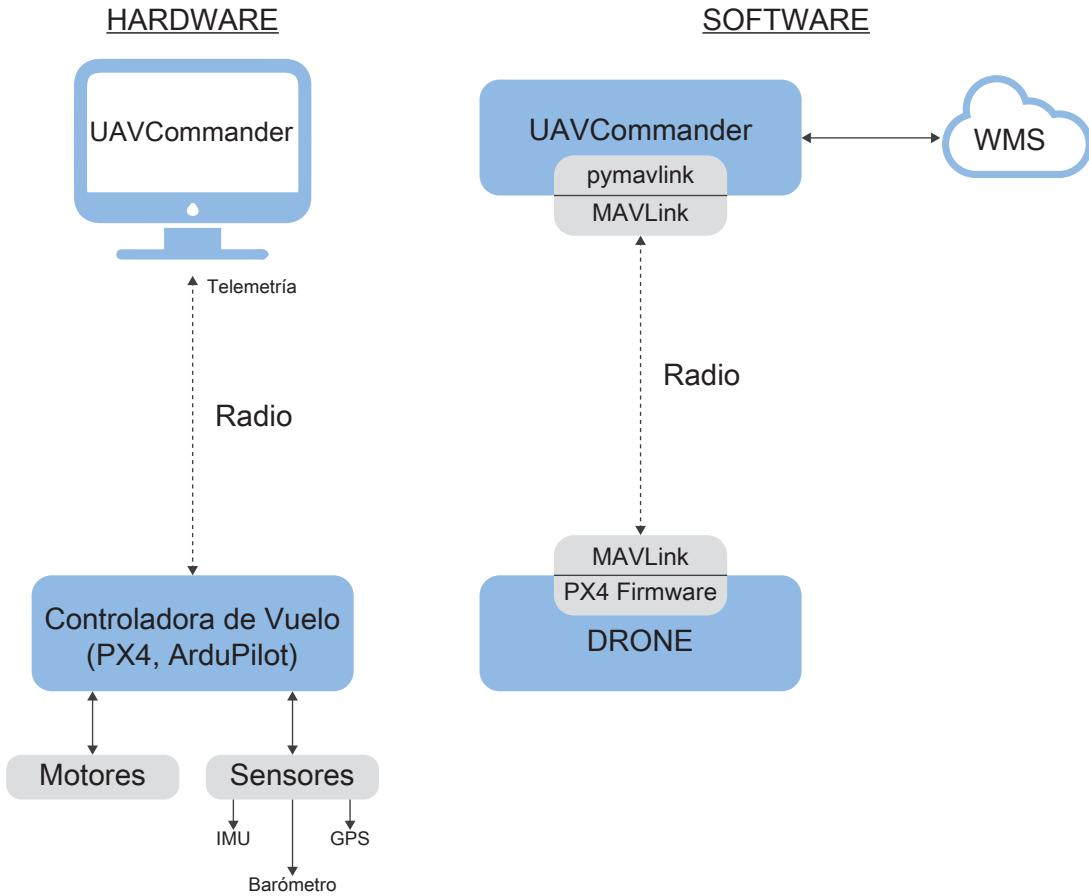


Figura 3.1: Esquema general del sistema.

- **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 la 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 [5]. Se aportan más detalles en la Sección 3.5 sobre la implementación del *driver*.

En la Figura 3.2 se observa cómo conforman 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 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.

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.

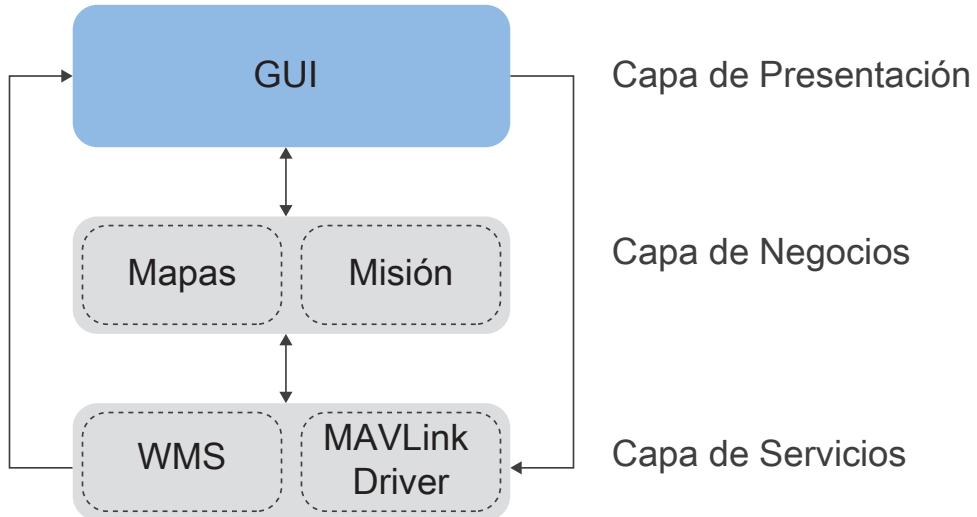


Figura 3.2: Capas que componen la aplicación.

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 recicajes de las antiguas interfaces de JdeRobot con ICE [5].

- **Pose3D:** Contiene los datos de posición en el espacio de la aeronave y su orientación. La orientación en el espacio es expresada mediante el uso de cuaterniones.
- **NavData:** Sirve datos secundarios de actuación como velocidades lineales o angulares o el estado de la batería.
- **mission:** Almacena los puntos de la misión con una posición en el espacio para cada uno.
- **extra:** Permite distinguir los puntos y asociar acciones de despegue y aterrizaje a puntos concretos.

A continuación se presentan los hilos de ejecución de la aplicación:

- **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 en la Figura 3.3 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 vista en conjunto provee al lector de una visión global detallada sobre el software previo a ese análisis pormenorizado sobre cada uno de los bloques.

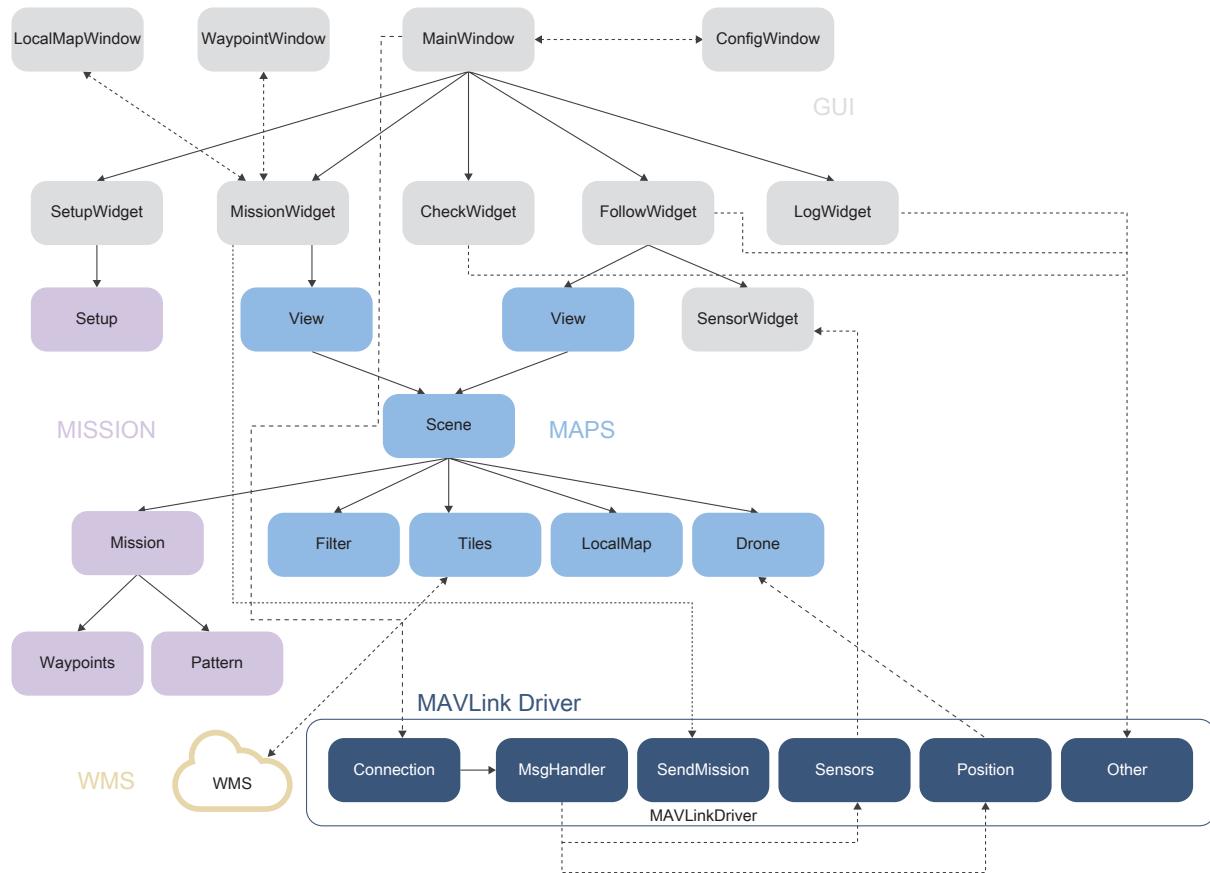


Figura 3.3: Esquema de clases del sistema.

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 [5]) 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 [6]. 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 3.4 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. 3) donde se observa la estructura utilizada.

```
1 def __init__(self, *args, **kwargs):
2
3     ....
4
5     self.widgets = QStackedWidget()
6     self.widgets.addWidget(MySetupWidget(self))
7     self.widgets.addWidget(MyMissionWidget(self))
8     self.widgets.addWidget(MyChecklistWidget(self))
9     self.widgets.addWidget(MyFollowWidget(self))
10    self.widgets.addWidget(MyLogWidget(self))
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    toolbar.addSeparator()
21    ....
22
23    def onSetupToolBarClick(self):
24        self.setup_action.setChecked(True)
25        self.mission_action.setChecked(False)
```

```

25     self.checklist_action.setChecked(False)
26     self.follow_action.setChecked(False)
27     self.log_action.setChecked(False)
28
29     self.widgets.setCurrentIndex(0) # Fija el widget visible

```

Código 3: Estructura de pestañas en la ventana principal.

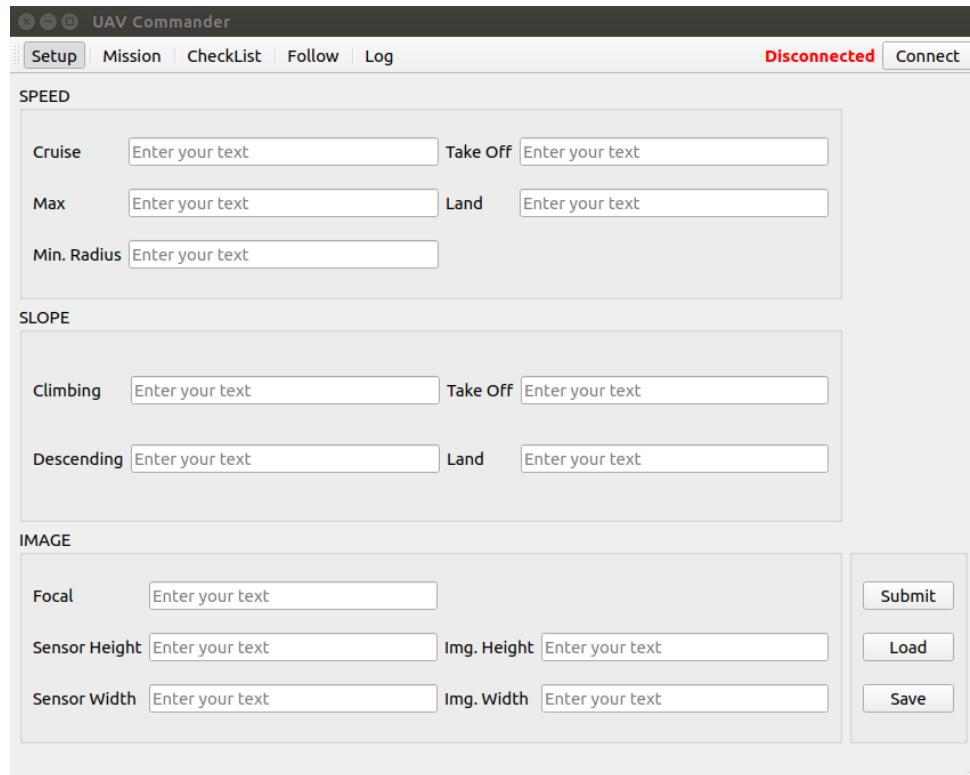


Figura 3.4: Ventana principal con la pestaña *Setup* activa.

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 3.5 se puede observar alguna 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* de Qt5 que permite cargar ficheros

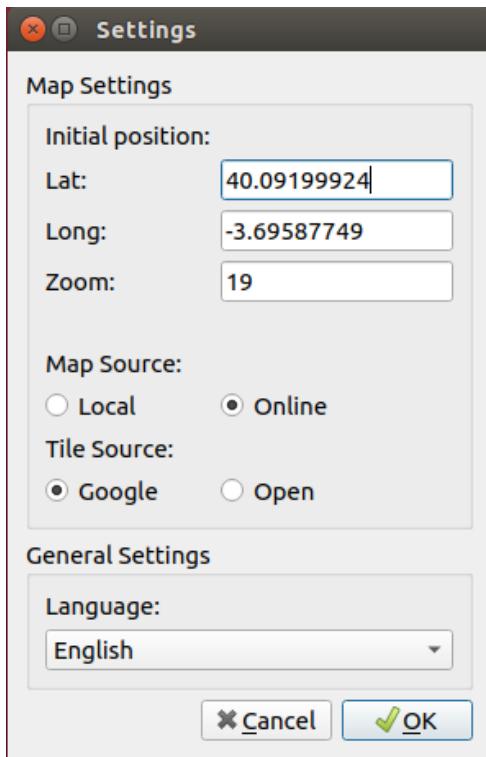


Figura 3.5: Ventana secundaria de configuración.

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 permiten 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*. Esta estructura se muestra en la Figura 3.6.

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. Esta estructura en mosaico se puede observar en la Figura 3.7. Así pues, se introduce la primera

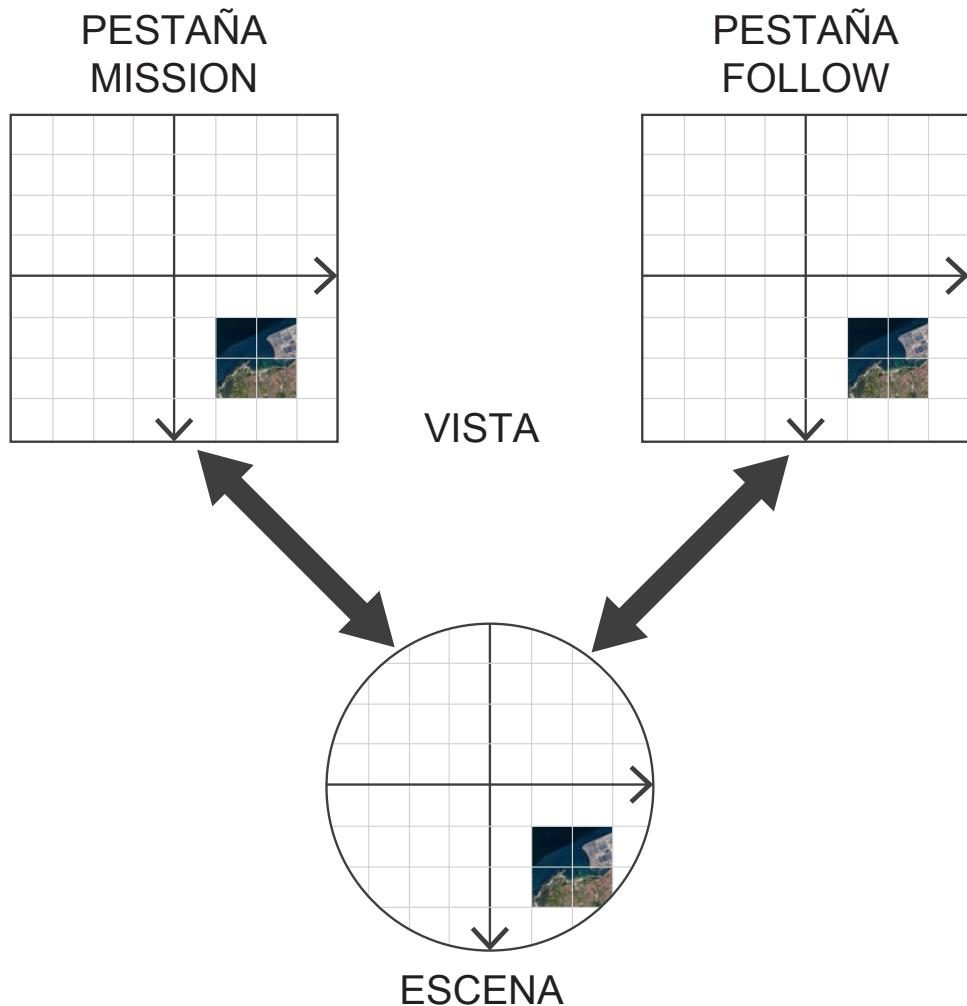


Figura 3.6: Estructura vistas-escena.

herramienta bajo la escena, las teselas. Las teselas constituyen un grupo de elementos gráficos 2D que son gobernados bajo las mismas reglas. Estos elementos gráficos son en su totalidad imágenes con una posición asociada que permite la construcción del mosaico. En esta primera versión del sistema de mapas, las teselas utilizadas son nueve.

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 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 georeferenciadas, 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-

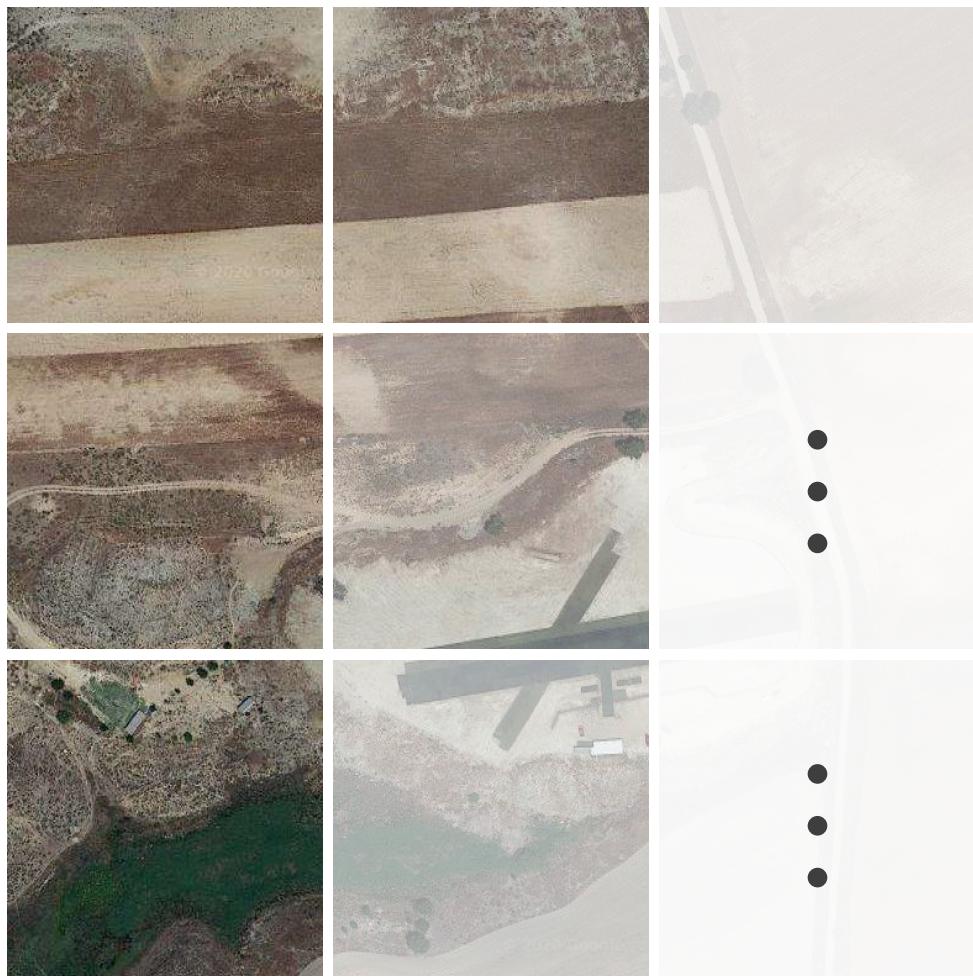


Figura 3.7: Esquema de vista en mosaico.

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 3.8 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.



(a) Tesela de Google Earth.

(b) Tesela de Open Maps.

(c) Tesela de mapa local.

Figura 3.8: Diferentes teselas disponibles para visualizar el mapa.

La obtención de la imagen depende del tipo de mapa a visualizar. En la Figura 3.9 se representa la secuencia de llamadas necesaria para obtener una escena.

De la secuencia de llamadas se pueden 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 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.

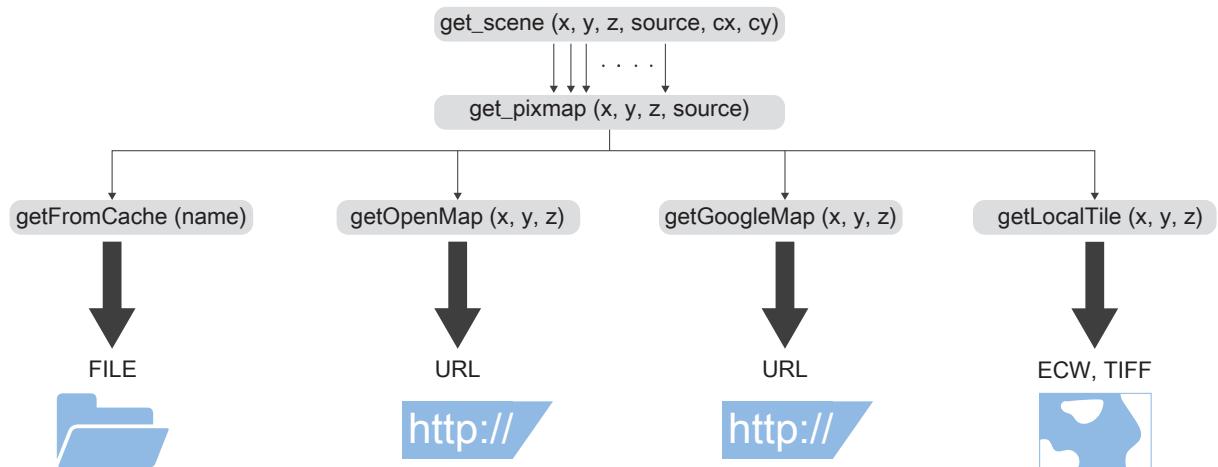


Figura 3.9: Secuencia de llamadas para obtener una escena concreta.

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 exacta 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 3.10 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.

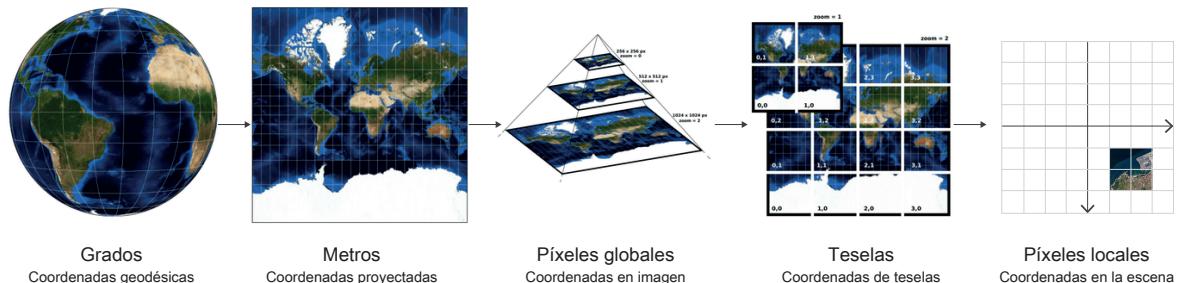


Figura 3.10: Esquema de conversión de coordenadas.

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. 4) que se encarga de filtrar los diferentes eventos.

Los eventos filtrados son tres. En la línea 2 (Cód. 4) 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 dron sobre la escena. Ambas serán explicadas en las siguientes secciones.

3.4. Creador de Misiones

El creador de misiones permite, como su propio nombre indica, la creación y el envío de misiones de navegación y cartografía 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

```

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

```

Código 4: Filtro de eventos de la escena.

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.

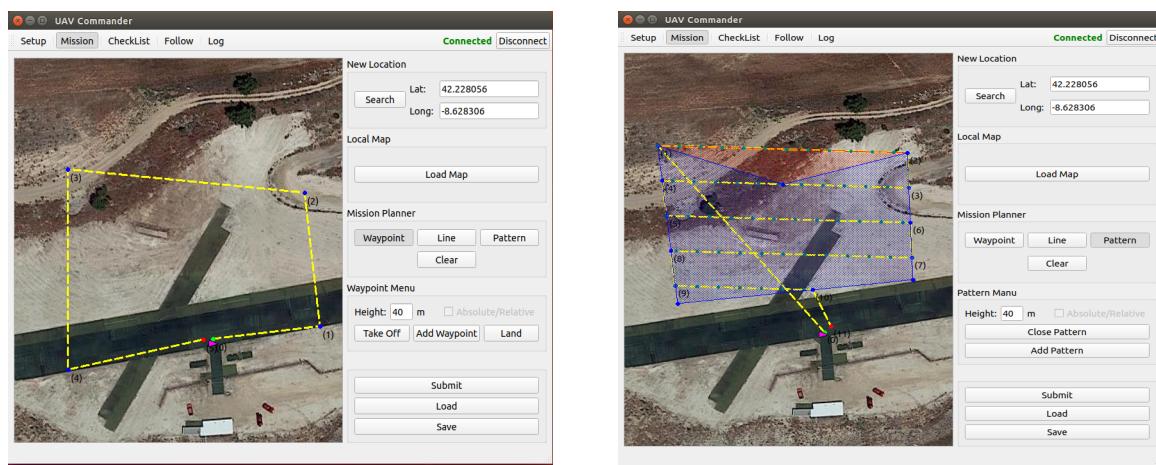
Es importante destacar que ambas informaciones se pueden guardar en diferentes archivos de forma persistente en el disco. De la misma forma, estos archivos se pueden cargar en memoria. El formato seguido para la creación de estos ficheros de configuración y de misión es un formato plano de texto [17].

Tipos de misión

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 3.11a se puede observar un ejemplo de una misión polilínea ya creada.

En cambio, las misiones 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. Una vez creado el polígono se genera automáticamente 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 3.11b se puede observar un ejemplo de una misión de patrón ya creada. Además, en el Código 5 se muestra un fragmento del algoritmo de barrido creado para recorrer un superficie delimitada por una lista de vértices.



(a) Misińn polilínea.

(b) Misińn de patrón.

Figura 3.11: Tipos de misińn disponibles.

Los puntos de aterrizaje y despegue son algo diferentes a los puntos de paso convencionales. Con respecto al despegue, la altura asociada a ese punto se corresponde con una altura de seguridad, la cual se debe haber sobrepasado para poder iniciar la misión y avanzar hacia el primer punto de paso. La senda de despegue se configura en línea recta según la orientación en el despegue. Para multicópteros esta senda es completamente vertical, mientras que para las aeronaves de ala fija se utilizará el valor asociado a la inclinación de despegue.

Por otro lado, el aterrizaje funciona de forma similar. La senda de aterrizaje se creará a través de dos puntos. El primero de ellos será el último punto de paso introducido y fijará la altura a la que se inicia la aproximación. El segundo punto fija el punto de aterrizaje y la orientación de la aproximación. Para multicópteros esta aproximación será vertical, mientras que para alas fija se utilizará el valor de la inclinación de aterrizaje introducida durante la caracterización de la aeronave.

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 (absoluta, sobre el nivel del mar) 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 la Figura 3.11 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.

```

1  def calc_path(self):
2      # Fija los parámetros iniciales del escaneo
3      self.set_scan(self.p_init)
4
5      # Primera pasada
6      self.wayp.append(self.v_scan)
7      self.calc_intermediates(self.v_scan,
8          self.get_first_vertex(self.v_scan, self.l_scan))
9      self.wayp.append(self.get_first_vertex(self.v_scan, self.l_scan))
10
11     i = 1
12     while True:
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 5: Algoritmo de barrido para las misiones de patrón.

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.

Captura de imágenes

El uso principal del software es cartográfico, para ello, es necesario la toma de fotografías por parte de la aplicación. Esta funcionalidad solo estaría disponible para la misiones por patrón, que están ideadas para el escaneo de superficies. Este cálculo de posiciones de captura

de imagen se realiza de forma automática a partir de los datos del sensor (cámara) y la altura de vuelo. Estas posiciones son las calculadas por el procedimiento *calc_intermediates(init, end)* en las líneas 6 y 26 del Código 5.

El procedimiento anterior utiliza constantes de la misión para calcular la posición de captura. Estas constantes de misión son la distancia entre fotografías consecutivas (*fotobase*, Ec. 3.1) y entre pasadas de la aeronave (*espaciamiento*, Ec. 3.2), y dependen de los datos del sensor y la altura de vuelo [47]. Las ecuaciones que comandan estos valores son:

$$fotobase = sh * (1 - p \%) * E_v \quad (3.1)$$

$$espaciamiento = sw * (1 - q \%) * E_v \quad (3.2)$$

Siendo *sh* y *sw* el alto y ancho del sensor, *p* y *q* el solapamiento longitudinal y transversal de las fotografías, cuyos valores típicos en tanto por ciento suelen ser *p* = 60 % y *q* = 30 %, y finalmente, *E_v* la escala de vuelo, que se calcula siguiendo *E_v* = *H*/*f*, donde *H* es la altura de vuelo y *f* la distancia focal del sensor.

La comunicación MAVLink para la toma de fotografías se explicará en la siguiente sección junto con las principales comunicaciones de la aplicación.

3.5. MAVLink Driver

El último bloque de la aplicación desarrollada 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 [5].

Sus 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, propone una implementación del protocolo de comunicaciones MAVLink facilitando acciones como esta.

En el fragmento de código (Cód. 6) 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 def uav_connect(port, baudrate=None):
15     master = mavutil.mavlink_connection(port, baudrate, autoreconnect=True)
16     print('Connection established to device')
17     heartbeat = master.wait_heartbeat()
18
19     print("Heartbeat Received", heartbeat)
20
21     # Set the complete set of commands
22     master.mav.request_data_stream_send(master.target_system,
23                                         master.target_component,
24                                         mavutil.mavlink.MAV_DATA_STREAM_ALL,
25                                         RATE, 1)
26
return master

```

Código 6: 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 7 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 7: Handler de conexión.

Envío de misión

El envío de misiones se realiza a través del protocolo de misiones que propone MAVLink. La Figura 3.12 recoge un diagrama del protocolo, el cual ha sido sacado de la propia web de MAVLink [23].

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* o comandos. Alguno ejemplo pueden ser los mostrados durante el Código 3.12 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 [15]. 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.

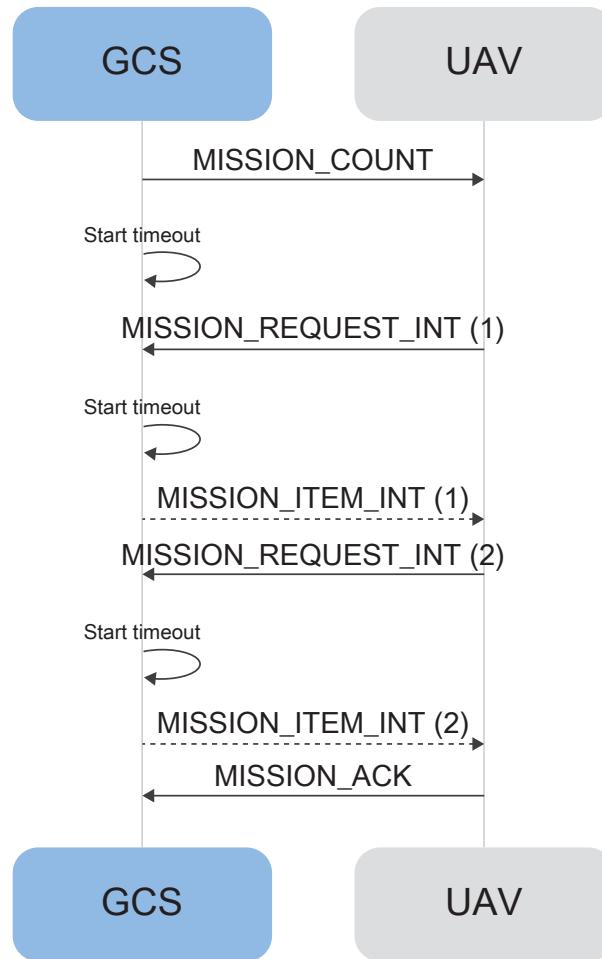


Figura 3.12: Diagrama del protocolo de misiones de MAVLink.

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

Tabla 3.2: Mensaje `MISSION_ITEM` [21].

Parámetros	NAV_TAKEOFF	NAV_LAND	NAV_WAYPOINT
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_*.

Entre los comandos que se pueden encapsular en los mensajes de misión destaca el usado para la toma de fotografías, *MAV_CMD_DO_SET_CAM_TRIGG_DIST*. En la Tabla 3.4 se muestran los parámetros asociados a este comando. Este comando permite establecer una distancia de disparo donde la cámara captura cada vez que se supera esta distancia. El funcionamiento pues, es sencillo. Al inicio de la pasada se manda el mensaje de misión con la *distancia* entre fotografías deseada, mientras que al final de la pasada se manda una *distancia* nula para detener la captura de fotos.

Parámetros	Descripción
Distancia	Distancia de disparo.
Obturación	Tiempo de obturación.
Disparo	Disparo inmediato.
param4	-
param5	-
param6	-
param7	-

Tabla 3.4: Parámetros del comando *MAV_CMD_DO_SET_CAM_TRIGG_DIST*.

Para la implementación del protocolo se parte de un pseudo-código propuesto por la Universidad de Colorado Boulder [45]. 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 8 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     if frame_type:
4         frame = mavutil.mavlink.MAV_FRAME_GLOBAL # 0
5     else:
6         frame = mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT # 3
7
8     seq = 1
9     radius = 10

```

```

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

```

```

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

```

Código 8: 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. 9).

```

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     if mav_autopilot == 3: # ArduPilot
7         status_msg = 'SYS_STATUS'
8         imu_msg = 'RAW_IMU'
9     elif mav_autopilot == 12: # PX4
10        status_msg = 'BATTERY_STATUS'
11        imu_msg = 'HIGHRES_IMU'
12    else:
13        status_msg = 'BATTERY_STATUS'
14        imu_msg = 'HIGHRES_IMU'
15
16    # get battery_remaining

```

```

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

```

Código 9: 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 3.5.

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 3.5: Diccionario de mensajes soportados por cada autopiloto.

Además, la tabla anterior (Tab. 3.5) también indica qué 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 3.13 muestra esta ventana.



Figura 3.13: Ventana secundaria de sensores.

Por último, se presenta un esquema que representa 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 3.14.

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 en el Código 10. Los mensajes enviados por cada autopiloto se presentan en la Tabla 3.5. Los parámetros de estos mensajes enviados pueden comprobarse en la documentación de MAVLink [16].

```

1 def refreshAPMPose3D(master, pose):
2     # get attitude
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
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 = Pose3DData()
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.setPose3DData(data)

```

Código 10: Procedimiento *refreshAPMPose3D()*, encargado de leer y actualizar los datos de posición enviados por la aeronave.

El hilo de posición accede periódicamente a la interfaz y actualiza en la escena gráfica 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.

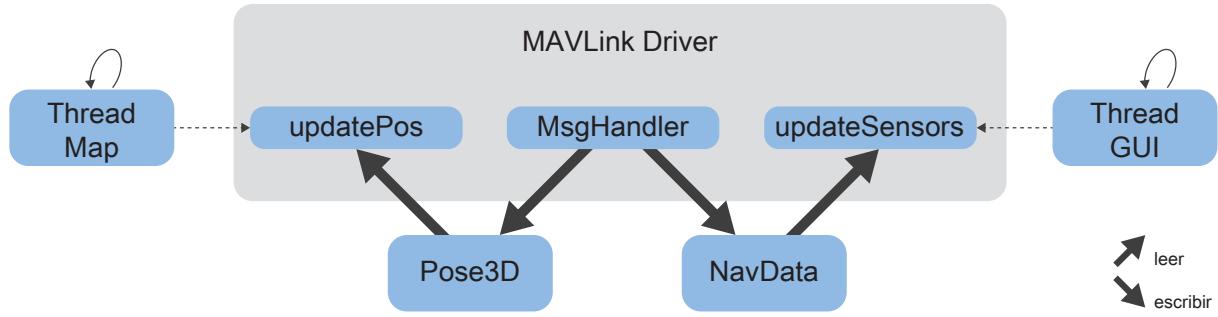


Figura 3.14: Diagrama del control de vuelo.

Finalmente, en la Figura 3.14 se muestra un diagrama con el flujo de información y la secuencia de llamadas de ambas partes del control de vuelo en conjunto.

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 11.

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 3.15 muestra un diagrama del protocolo.

La estructura del mensaje *COMMAND_LONG* se muestra en la Tabla 3.6. Para un cambio de velocidad el comando enviado es *MAV_CMD_DO_CHANGE_SPEED* cuyos parámetros se pueden observar en la Tabla 3.7. Para más 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 12.

```

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 11: Activación de modos de vuelo.

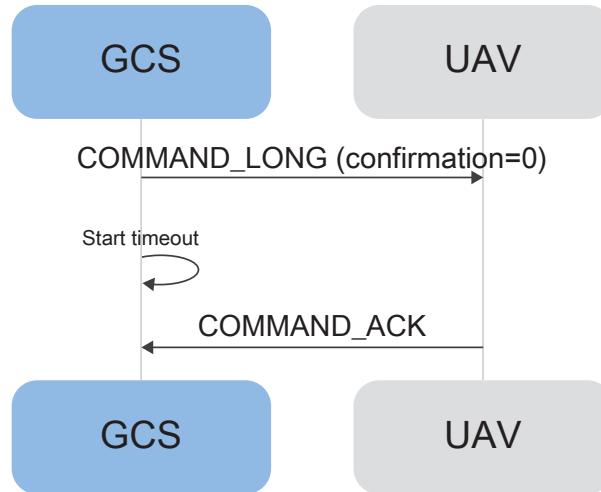


Figura 3.15: Diagrama del protocolo de comandos de MAVLink.

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 3.6: Mensaje *COMMAND_LONG* [19].

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 3.7: Comando *MAV_CMD_DO_CHANGE_SPEED* [14].

```

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 12: Mensaje *COMMAND_LONG* para el cambio de velocidad de vuelo.

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. 13).

```

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 13: Armado y desarmado de la aeronave.

Capítulo 4

Manual de usuario y casos de uso

Este capítulo presenta en primer lugar, un manual de usuario sobre la aplicación. Este manual realiza un viaje gráfico por cada una de las ventanas y pestañas mostrando cada una de las funciones de la aplicación.

En segundo lugar, recoge en forma de casos de uso diferentes acciones típicas que se pueden realizar con la aplicación. El motivo de los casos de uso es exemplificar y poner en práctica lo explicado durante el manual de usuario. Un caso de uso trata de mostrar al lector la secuencia de acciones a realizar para completar una tarea. Suponen pues una especie de guía de aprendizaje de la aplicación. El lector tiene que ser consciente que no se representan todas las acciones disponibles en la aplicación, sino las más comunes en un uso estándar. Existen acciones que sin estar especificadas en los casos de uso se podrán realizar según lo mostrado en el manual de usuario.

4.1. Manual de Usuario

El manual de usuario se divide en tres diferentes secciones. Se distingue entre menús, ventana principal y ventanas secundarias. Para cada una de estas secciones y sus sucesivas secciones internas se muestra una captura de pantalla junto con una tabla que expone las diferentes opciones disponibles y una breve explicación de la misma. Esta sección imita el formato la sencillez y brevedad de un manual de usuario, donde la escasez de texto es una virtud.

Menús

La aplicación solo dispone de un menú, el menú de fichero (Secc. 4.1). Para acceder a los menús es necesario pinchar sobre la esquina superior izquierda de la ventana.

Menú de Fichero

El menú de fichero *File* dispone de dos acciones, de configuración y de salida. La acción de configuración permite cambiar el origen del mapa mostrado en la escena y abrir la ventana secundaria de configuración (Secc. 4.1). La Figura 4.1 muestra el menú de fichero, mientras que la Tabla 4.1 recoge sus funcionalidades.

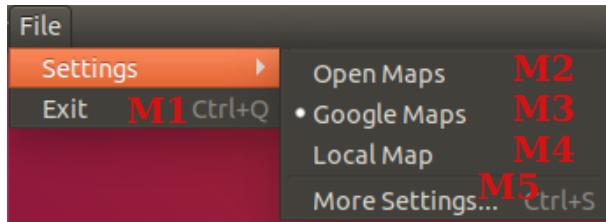


Figura 4.1: Menú de fichero.

Nº	Descripción
M1	Opción para cerrar la aplicación.
M2	Opción de cambio de escena a mapas abiertos.
M3	Opción de cambio de escena a mapas de Google.
M4	Opción de cambio de escena a mapas locales.
M5	Opción para abrir ventana de configuración (Ver 4.1).

Tabla 4.1: Menú de fichero.

Ventana Principal

Dentro de la ventana principal de la aplicación se distinguen el panel de navegación (Secc. 4.1) y las cinco pestañas disponibles; *Setup* (Secc. 4.1), *Mission* (Secc. 4.1), *Checklist* (Secc. 4.1), *Follow* (Secc. 4.1) y *Log* (Secc. 4.1).

Panel de Navegación

El panel de navegación permite fundamentalmente la navegación entre las diferentes pestañas de la aplicación y el establecimiento/cierre de la conexión con la aeronave. La Figura 4.2 muestra el panel de navegación, mientras que la Tabla 4.2 recoge sus funcionalidades.

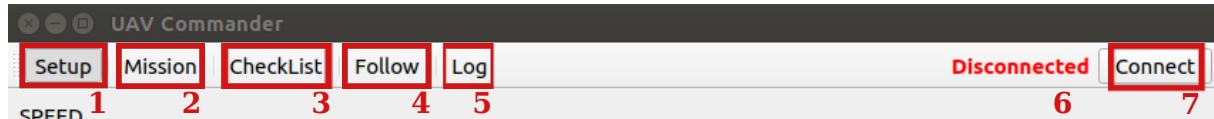


Figura 4.2: Panel de navegación.

Nº	Descripción
1	Botón para acceder a la pestaña <i>Setup</i> (Ver 4.1).
2	Botón para acceder a la pestaña <i>Mission</i> (Ver 4.1).
3	Botón para acceder a la pestaña <i>Checklist</i> (Ver 4.1).
4	Botón para acceder a la pestaña <i>Follow</i> (Ver 4.1).
5	Botón para acceder a la pestaña <i>Log</i> (Ver 4.1).
6	Etiqueta que muestra el estado de la conexión.
7	Botón para establecer/cortar la conexión.

Tabla 4.2: Panel de navegación.

Pestaña Setup

La pestaña de *Setup* reúne las acciones de caracterización de la aeronave y su carga de pago. La Figura 4.3 muestra esta pestaña, mientras que la Tabla 4.3 recoge sus funcionalidades.

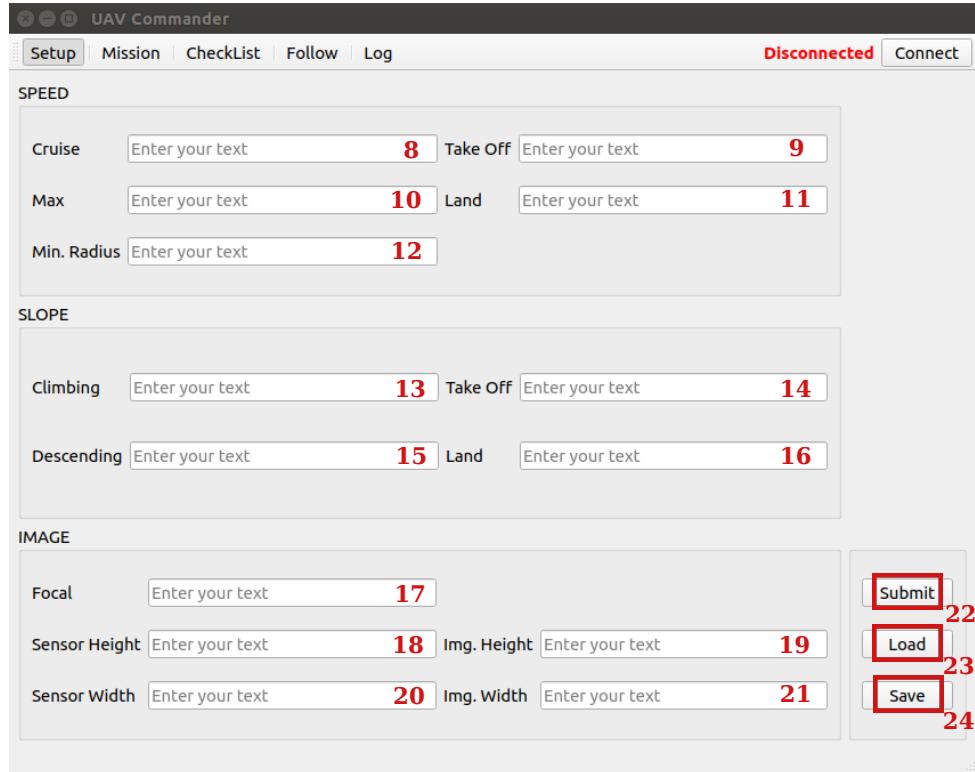


Figura 4.3: Pestaña *Setup*.

Nº	Descripción
8	Velocidad de crucero (m/s).
9	Velocidad de despegue (m/s).
10	Velocidad máxima (m/s).
11	Velocidad de aterrizaje (m/s).
12	Radio mínimo de giro (m).
13	Pendiente de ascenso (°).
14	Pendiente de despegue (°).
15	Pendiente de descenso (°).
16	Pendiente de aterrizaje (°).
17	Distancia focal del sensor (mm).
18	Alto del sensor (mm).
19	Alto de la imagen (px).
20	Ancho del sensor (mm).
21	Ancho de la imagen (px).
22	Botón crear la configuración.
23	Botón para cargar una configuración.
24	Botón para guardar la configuración.

Tabla 4.3: Pestaña *Setup*.

Pestaña Mission

La pestaña de *Misión* permite la creación de misiones. Dispone de una escena donde se muestra el mapa y de una serie de herramientas de creación de misión. Sobre esta pestaña se puede acceder al creador de misiones polilínea (Secc. 4.1) y al creador de misiones por patrón (Secc. 4.1). La Figura 4.4 muestra esta pestaña, mientras que la Tabla 4.4 recoge sus funcionalidades.

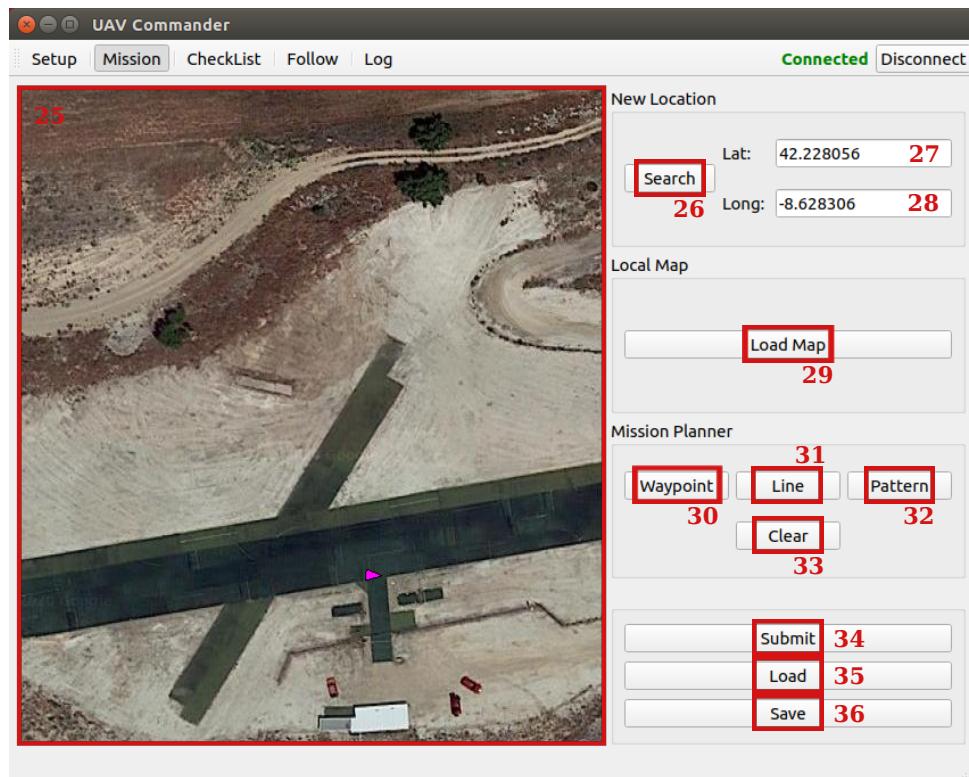


Figura 4.4: Pestaña *Misión*.

Nº	Descripción
25	Escena con el mapa cargado.
26	Botón para buscar determinada posición en el mapa.
27	Latitud (°).
28	Longitud (°).
29	Botón de carga de mapa local.
30	Botón para acceder al creador de misiones multilínea (Ver 4.1).
31	No disponible.
32	Botón para acceder al creador de misiones por patrón (Ver 4.1).
33	Botón para borrar la misión activa.
34	Botón para crear la misión.
35	Botón para cargar una misión.
36	Botón para guardar la misión.

Tabla 4.4: Pestaña *Misión*.

Creador de misiones polilínea

Para acceder al creador de misiones polilínea es necesario tener pulsada la acción 30 en la pestaña *Mission*. La Figura 4.5 muestra el creador de misiones polilínea, mientras que la Tabla 4.5 recoge sus funcionalidades.

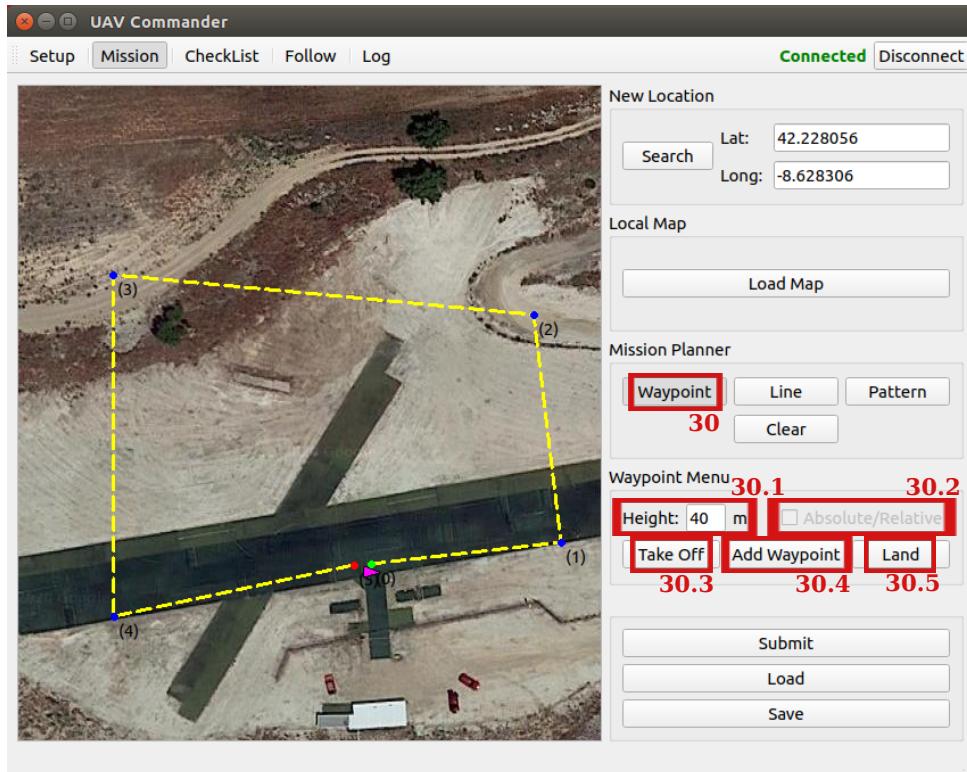


Figura 4.5: Creador de misiones polilínea.

Nº	Descripción
30	Botón para acceder al creador de misiones multilínea.
30.1	Altura (m).
30.2	Altura absoluta o relativa.
30.3	Botón para añadir punto de despegue.
30.4	Botón para abrir ventana de puntos de paso (Ver 4.1).
30.5	Botón para añadir punto de aterrizaje.

Tabla 4.5: Creador de misiones polilínea.

Creador de misiones por patrón

Para acceder al creador de misiones por patrón es necesario tener pulsada la acción 32 en la pestaña *Mission*. La Figura 4.6 muestra el creador de misiones por patrón, mientras que la Tabla 4.6 recoge sus funcionalidades.

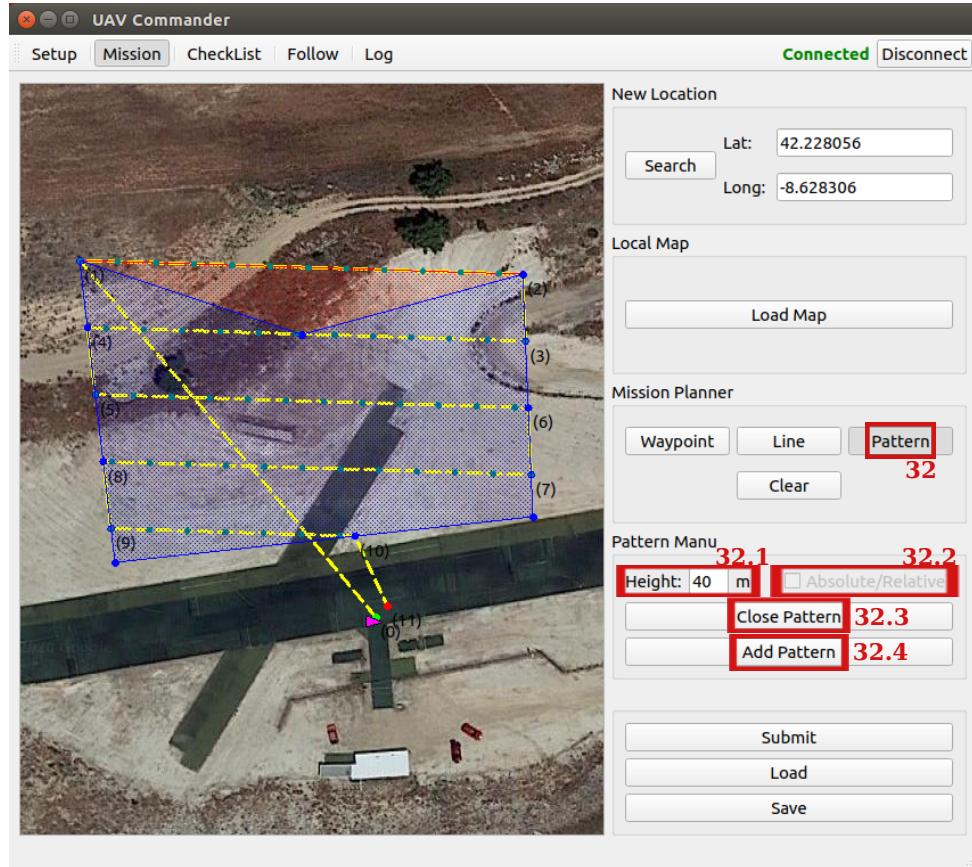


Figura 4.6: Creador de misiones por patrón.

Nº	Descripción
32	Botón para acceder al creador de misiones por patrón.
32.1	Altura (m).
32.2	Altura absoluta o relativa.
32.3	Botón para cerrar el patrón.
32.4	Botón para abrir ventana de vértices (Ver 4.1).

Tabla 4.6: Creador de misiones por patrón.

Pestaña Checklist

La pestaña *Checklist* recoge las comprobaciones pre-vuelo necesarias para poder iniciar una misión. La Figura 4.7 muestra esta pestaña, mientras que la Tabla 4.7 recoge sus funcionalidades.

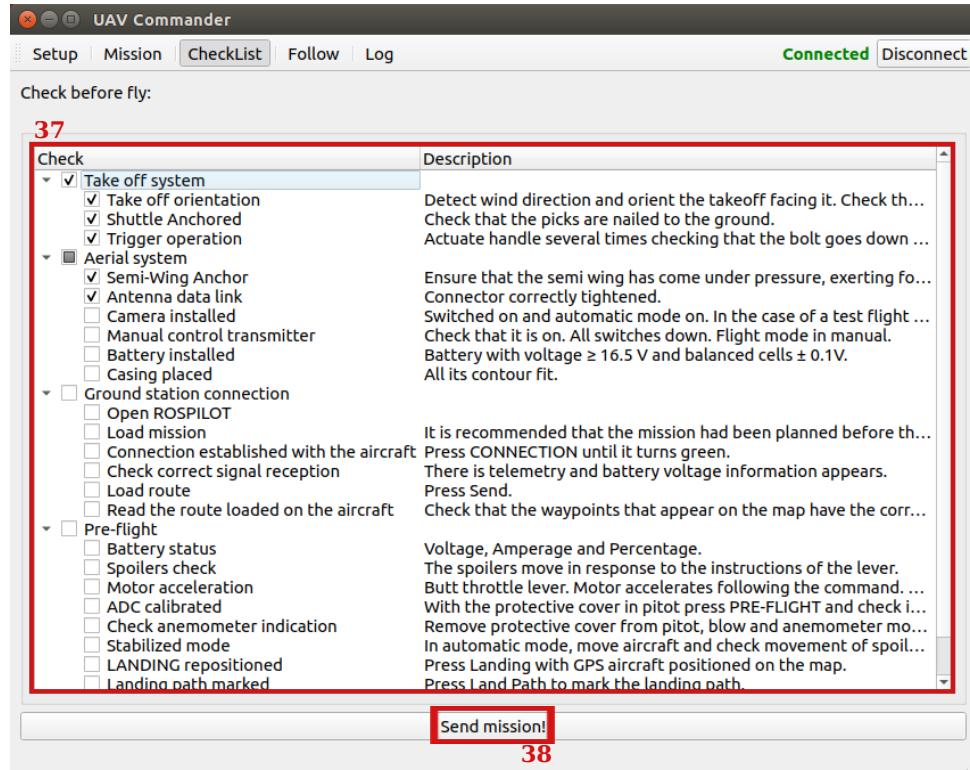


Figura 4.7: Pestaña *Checklist*.

Nº	Descripción
37	Listado de comprobaciones pre-vuelo.
38	Botón para enviar la misión a la aeronave.

Tabla 4.7: Pestaña *Checklist*.

Pestaña *Follow*

La pestaña *Follow* permite el seguimiento en tiempo real de la aeronave. El seguimiento se realiza a través de la posición gracias a la escena y a través de los datos de navegación gracias a la ventana secundaria de sensores (Secc. 4.1).

La Figura 4.8 muestra esta pestaña, mientras que la Tabla 4.8 recoge sus funcionalidades.

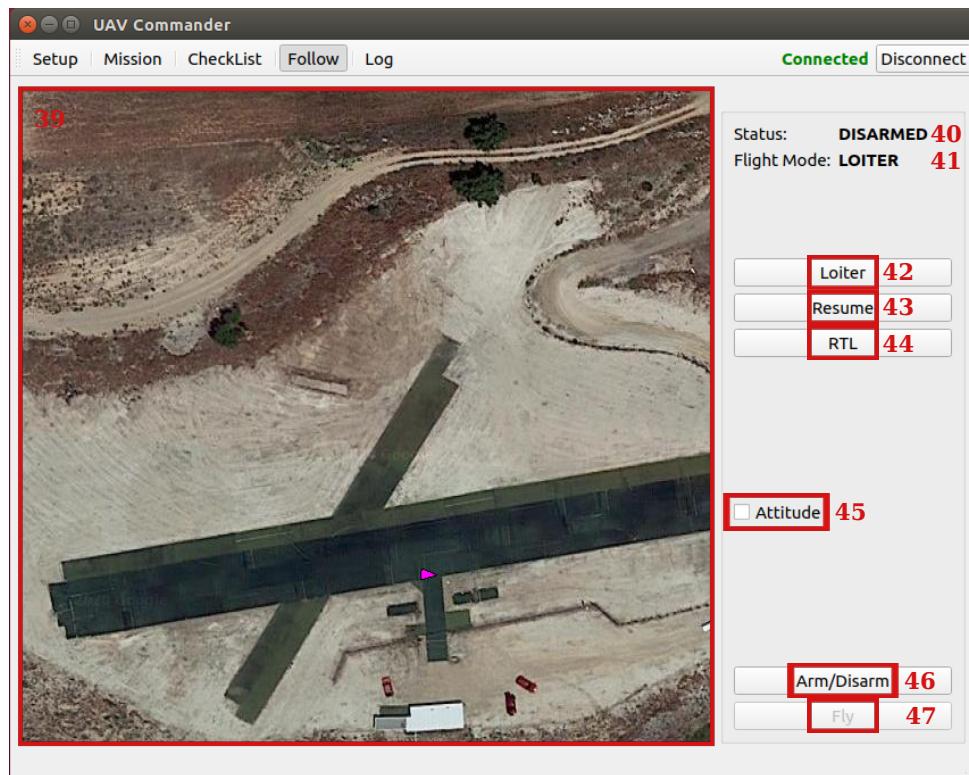


Figura 4.8: Pestaña *Follow*.

Nº	Descripción
39	Escena con el mapa cargado.
40	Etiqueta que muestra el estado de la aeronave.
41	Etiqueta que muestra el modo de vuelo de la aeronave.
42	Botón para activar el modo de vuelo <i>Loiter</i> .
43	Botón para activar el modo de vuelo <i>Auto</i> .
44	Botón para activar el modo de vuelo <i>RTL</i> .
45	Botón para abrir la pestaña secundaria de sensores (Ver 4.1).
46	Botón para armar/desarmar.
47	Botón para despegar e iniciar misión.

Tabla 4.8: Pestaña *Checklist*.

Pestaña Log

La pestaña *Log* permite la obtención de datos post-vuelo de la aeronave. La Figura 4.9 muestra esta pestaña, mientras que la Tabla 4.9 recoge sus funcionalidades.

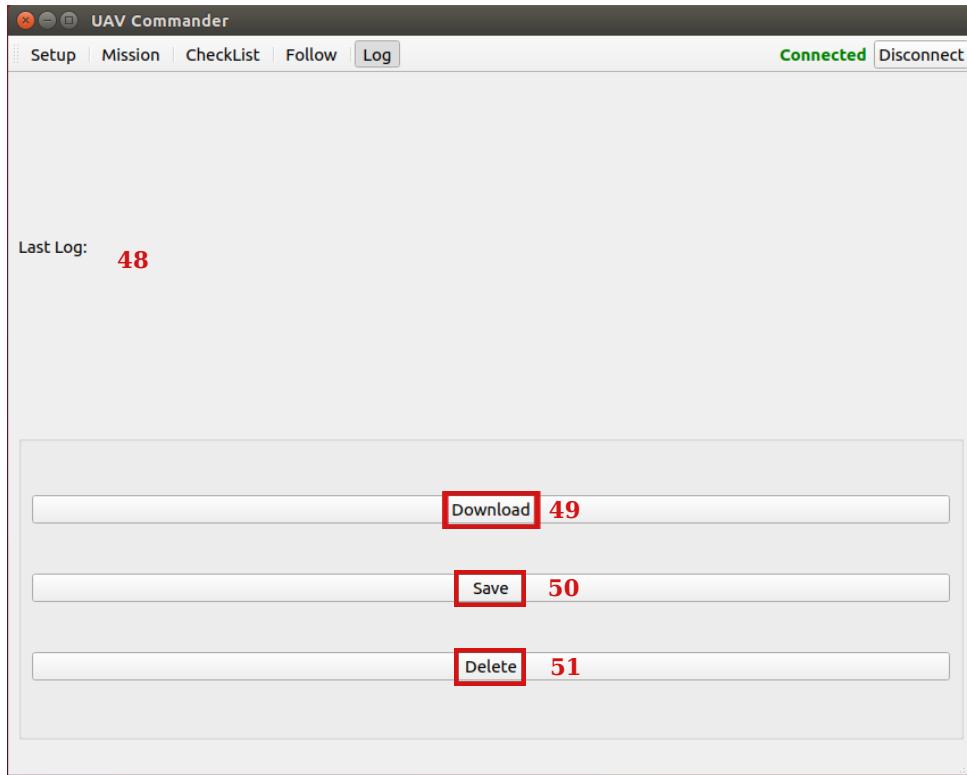


Figura 4.9: Pestaña *log*.

Nº	Descripción
48	Etiqueta que muestra último log descargado.
49	Botón para descargar los logs del autopiloto.
50	Botón para guardar los logs descargados.
51	Botón para borrar los logs del autopiloto.

Tabla 4.9: Pestaña *log*.

Ventanas Secundarias

Las ventanas secundarias son desplegadas a través del menú o de la ventana principal y complementan las acciones de la aplicación. Entre las ventanas secundarias disponibles se encuentran la ventana de configuración (Secc. 4.1), la ventana de carga de mapas locales (Secc. 4.1), la ventana de puntos de paso (Secc. 4.1) y la ventana de sensores (Secc. 4.1).

Ventana de Configuración

La ventana de configuración permite modificar parámetros de la escena y el idioma de la aplicación de forma dinámica. La Figura 4.10 muestra esta ventana, mientras que la Tabla 4.10 recoge sus funcionalidades.



Figura 4.10: Ventana secundaria de configuración.

Nº	Descripción
C1	Latitud por defecto de la escena (°).
C2	Longitud por defecto de la escena (°).
C3	Zoom por defecto de la escena.
C4	Botón para activar mapas locales sobre la escena.
C5	Botón para activar mapas en línea sobre la escena.
C6	Botón para activar mapas de Google sobre la escena.
C7	Botón para activar mapas abiertos sobre la escena.
C8	Desplegable para seleccionar el idioma de la interfaz gráfica de usuario.
C9	Botón para cancelar cambios.
C10	Botón para confirmar cambios.

Tabla 4.10: Ventana secundaria de configuración.

Ventana de carga de Mapa Local

La ventana de carga de mapas locales permite abrir ficheros locales con datos ortofotográficos y desplegarlos sobre la escena. La Figura 4.11 muestra esta ventana, mientras que la Tabla 4.11 recoge sus funcionalidades.

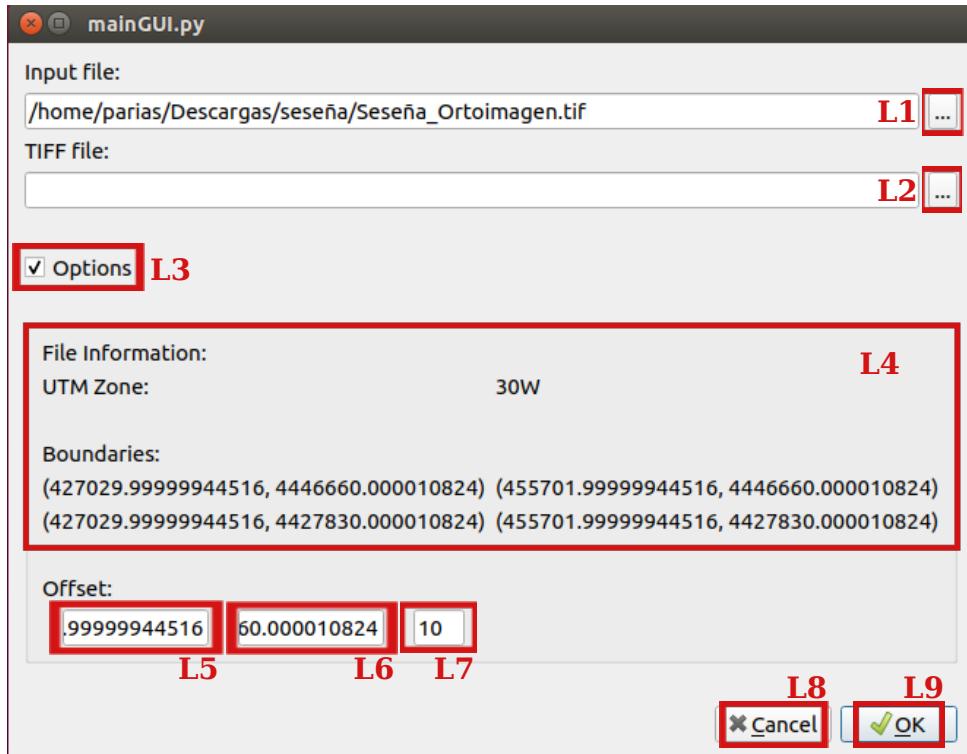


Figura 4.11: Ventana secundaria de carga de mapas locales.

Nº	Descripción
L1	Botón para cargar archivo con ortoimagen.
L2	Botón para cargar archivo con elevaciones.
L3	Botón parastrar/ocultar opciones.
L4	Información sobre ortoimagen cargada.
L5	Coordenada X (UTM) a cargar (m).
L6	Coordenada Y (UTM) a cargar (m).
L7	Zoom a cargar.
L8	Botón para cancelar cambios.
L9	Botón para confirmar cambios.

Tabla 4.11: Ventana secundaria de carga de mapas locales.

Ventana de Puntos de Paso

La ventana de puntos de paso permite desplegar una lista con los puntos de paso que componen una misión. La Figura 4.12 muestra esta ventana, mientras que la Tabla 4.12 recoge sus funcionalidades.

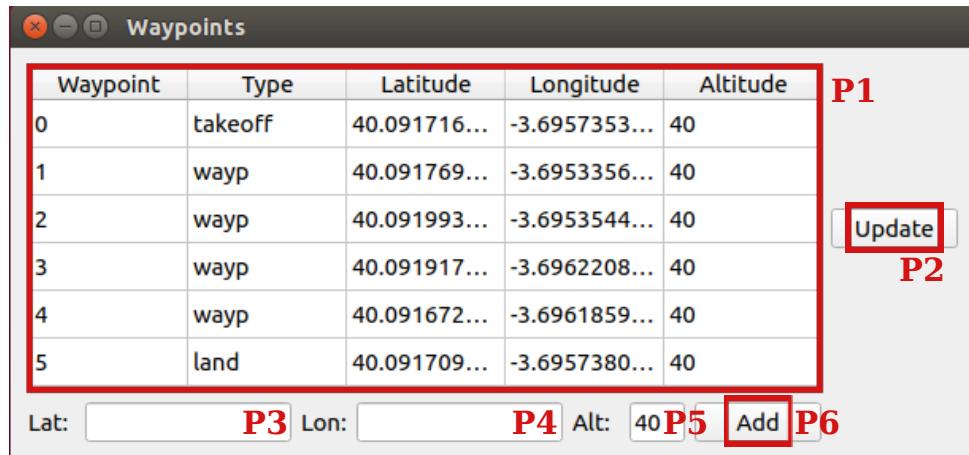


Figura 4.12: Ventana secundaria de puntos de paso.

Nº	Descripción
P1	Lista con los puntos de paso de la misión.
P2	Botón para actualizar los puntos de la lista sobre la escena.
P3	Latitud (°).
P4	Longitud (°).
P5	Altura (m).
P6	Botón para añadir un punto de paso a la lista.

Tabla 4.12: Ventana secundaria de puntos de paso.

Ventana de Sensores

La ventana de sensores muestra los datos de navegación de la aeronave a través de útiles de aeronavegación estándar. La Figura 4.13 muestra esta ventana, mientras que la Tabla 4.13 recoge sus funcionalidades.



Figura 4.13: Ventana secundaria de sensores.

Nº	Descripción
S1	Horizonte artificial.
S2	Indicador de rumbo.
S3	Altímetro.
S4	Velocidad lineal X.
S5	Velocidad lineal Y.
S6	Velocidad lineal Z.
S7	Batería.

Tabla 4.13: Ventana secundaria de sensores.

4.2. Casos de uso

Las diferentes pruebas realizadas durante el desarrollo se reúnen en los diferentes casos de uso, a cada uno de los cuales se les dedica una subsección. 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, junto a cada caso de uso se añaden una serie de tablas que esquematizan los procedimientos y ejemplifican la interacción entre el usuario y la aplicación para cada caso.

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 la Figura 4.14.

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.

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. La Figura 4.15 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 obtenidos desde la página web del Instituto Geográfico Nacional [9]. 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 abre 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 cambiar 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 este caso de uso.

CU-01	Caracterización de una aeronave y su carga de pago.	
Versión	v1.0 - 25/01/2020 - Pedro Arias Pérez	
Dependencias	•	
Descripción	La aplicación deberá comportarse como se describe en el caso de uso cuando <i>el operario introduzca los parámetros de caracterización de la aeronave y su carga de pago.</i>	
Precondición	La aplicación debe estar abierta y en ejecución.	
Secuencia normal	Paso	Acción
	1	El operario pincha sobre <i>Setup</i> para desplegar dicha pestaña.
	2	El operario completa 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 <i>Submit</i> para caracterizar la aeronave y su carga de pago.
	4	En caso de que el operario detectase un error en los datos introducidos tras haber completado el Paso 3.
	4.1	El operario modifica los parámetros incorrectos sobre el formulario, dejando los datos correctos intactos.
	4.2	El operario pincha sobre el botón <i>Submit</i> para confirmar los cambios.
	5	Si el operario desease guardar dicha configuración para algún uso futuro durante otra ejecución de la aplicación,
	5.1	El operario debe pinchar sobre el botón <i>Save</i> para guardar de forma persistente la configuración. Una ventana de guardado emergirá.
	5.2	Sobre la ventana emergente el operario debe elegir un nombre y una ubicación de guardado para el nuevo archivo de configuración y pinchar sobre el botón <i>Save</i> .
Postcondición	-	
Excepciones / Rutas alternativas	Paso	Acción
	2	En caso de disponer de un archivo de configuración previamente creado,
	2A.1	El operario pincha sobre el botón <i>Load</i> para cargar el archivo de configuración.
	2A.2	Sobre la ventana emergente, el operario selecciona el archivo de configuración deseado y pincha sobre el botón <i>Open</i> .
Comentarios	No es posible tener más de un archivo de configuración abierto al mismo tiempo. En caso de tener un archivo de configuración abierto, cualquier cambio sobre el mismo debe ser confirmado pinchando sobre el botón <i>Submit</i> . Para guardar de forma persistente los cambios realizados sobre un archivo de configuración se debe pinchar sobre el botón <i>Save</i> . Esta acción sobrescribirá el archivo abierto y NO creará un archivo nuevo.	

Figura 4.14: Caso de uso 1.

CU-02	Carga de un mapa local en una determinada posición.	
Versión	v1.0 - 25/01/2020 - Pedro Arias Pérez	
Dependencias	<ul style="list-style-type: none"> El mapa local deberá ser una imagen georreferenciada con extensión <i>.tif</i> o <i>.ecw</i>. 	
Descripción	La aplicación deberá comportarse como se describe en el caso de uso cuando <i>el operario abra un mapa local y desee modificar la posición inicial del mismo</i> .	
Precondición	La aplicación debe estar abierta y en ejecución. El archivo con el mapa local debe haber sido previamente descargado.	
Secuencia normal	Paso	Acción
Secuencia normal	1	El operario pincha sobre <i>Mission</i> para desplegar dicha pestaña.
	2	El operario pincha sobre el botón <i>Load Map</i> para abrir la ventana de carga de mapas locales.
	3	El operario pincha sobre el primer botón ... para proceder a seleccionar el archivo con el mapa local.
	4	Sobre la ventana emergente el operario debe buscar y seleccionar el archivo con el mapa local y, a continuación, pinchar sobre el botón <i>Open</i> .
	5	Si el operario desease cargar un archivo con la elevación del terreno cargado,
	5.1	El operario pincha sobre el segundo botón ... para proceder a seleccionar el archivo con la elevación del terreno.
	5.2	Sobre la ventana emergente el operario debe buscar y seleccionar el archivo con la elevación del terreno y, a continuación, pinchar sobre el botón <i>Open</i> .
	6	Si el operario necesitase modificar la posición inicial en la que se despliega el mapa,
	6.1	El operario pincha sobre el cuadro <i>Options</i> para desplegar las opciones del mapa.
	6.2	El operario modifica los campos <i>Offset</i> con la posición inicial deseada.
	7	El operario pincha sobre el botón <i>OK</i> para confirmar los datos sobre el mapa local a abrir.
Postcondición	-	
Excepciones	Paso	Acción
Excepciones	6.2	Si la posición inicial introducida no es válida,
	6.2E.1	La aplicación no mostrará ningún aviso. La escena no mostrará ninguna imagen y permanecerá en blanco.
	6.2E.2	Para cargar una posición válida se deberá repetir el caso de uso CU-02.
Comentarios	-	

Figura 4.15: Caso de uso 2.

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. La Figura 4.16 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 que 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 como 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 abre 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.

CU-03	Creación y posterior guardado en fichero de una misión polilínea.	
Versión	v1.0 - 25/01/2020 - Pedro Arias Pérez	
Dependencias	<ul style="list-style-type: none"> CU-02: Carga de mapas locales. 	
Descripción	La aplicación deberá comportarse como se describe en el caso de uso cuando <i>el operario cree una misión polilínea y decida guardar la misión en un archivo</i> .	
Precondición	La aplicación debe estar abierta y en ejecución.	
Secuencia normal	Paso	Acción
	1	El operario pincha sobre <i>Mission</i> para desplegar dicha pestaña.
	2	El operario introduce la latitud y la longitud de la posición deseada y pincha en <i>Search</i> para obtener el mapa en dicha posición.
	3	El operario pincha sobre <i>Waypoint</i> para abrir el creador de misiones polilínea.
	4	El operario pincha sobre <i>Take Off</i> y posteriormente pincha sobre el mapa para seleccionar el punto de despegue de la aeronave. La altura introducida en el campo <i>Height</i> es la altura necesaria a alcanzar antes de comenzar la misión.
	5	El operario introduce la altura de vuelo deseada en el campo <i>Height</i> y posteriormente pincha sobre el mapa para añadir un punto de paso en la posición deseada.
	6	El operario repetirá el Paso 5 tantas veces como puntos de paso desee añadir.
	7	El operario pincha sobre <i>Land</i> y posteriormente pincha sobre el mapa para seleccionar el punto de aterrizaje de la aeronave.
	8	Si el usuario quisiera guardar la misión,
	8.1	El operario debe pinchar sobre el botón <i>Save</i> para guardar de forma persistente la misión. Una ventana de guardado emergirá.
	8.2	Sobre la ventana emergente el operario debe elegir un nombre y una ubicación de guardado para el nuevo archivo de misión y pinchar sobre el botón <i>Save</i> .
Postcondición	-	
Excepciones	Paso	Acción
	2	Si lo desea, el usuario puede utilizar un mapa local, para ello,
	2A.1	El operario debe seguir el caso de uso <i>Carga de un mapa local en una determinada posición</i> (CU-02).
	2	Si el operario no dispone de conexión a internet, de un mapa local de la zona o el mapa no ha sido cacheado previamente,
	2E.1	Se cancela el caso de uso. La acción no es realizable.
	5	Si el operario desea introducir los puntos de paso mediante latitud y longitud,
	5A.1	El operario pincha sobre el botón <i>Add Waypoint</i> para desplegar una ventana con la lista de puntos de paso.
	5A.2	Sobre esta ventana, el operario introduce en los campos <i>Lat</i> ,
		<i>Lon</i> y <i>Alt</i> la latitud, longitud y la altura deseada para el nuevo punto de paso.
	5A.3	El operario pincha sobre <i>Add</i> para añadir el nuevo punto de paso a la misión.
	5	Si el usuario introduciera erróneamente alguno de los puntos de paso,
	5E.1	El operario pincha sobre el botón <i>Clear</i> para borrar la misión.
	5E.2	El operario debe repetir este caso de uso para crear de nuevo una misión valida.
Comentarios	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 (al pinchar en el botón <i>Save</i>) y NO creará un archivo de misión nuevo.	

Figura 4.16: Caso de uso 3.

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 la Figura 4.17 una tabla resumen 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 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.

CU-04	Establecimiento de conexión con la aeronave.		
Versión	v1.0 - 25/01/2020 - Pedro Arias Pérez		
Dependencias	•		
Descripción	La aplicación deberá comportarse como se describe en el caso de uso cuando <i>el operario establezca la conexión entre la aeronave y la aplicación</i> .		
Precondición	La aplicación debe estar abierta y en ejecución. La aeronave debe estar encendida y con la posición GPS detectada.		
Secuencia normal	Paso	Acción	
	1	El operario pincha sobre el botón <i>Connect</i> para iniciar la conexión.	
	2	Si la conexión se ha establecido correctamente,	
		2.1	La etiqueta cambiará a color verde mostrando el texto <i>Connected</i> .
Postcondición	-		
Excepciones	Paso	Acción	
	2	Si la conexión no se ha establecido o algo inesperado ha ocurrido,	
		2E.1	La aplicación debe ser reiniciada y este caso de uso debe iniciarse de nuevo.
Comentarios	El establecimiento de conexión puede durar unos segundos en función de la aeronave en cuestión.		

Figura 4.17: Caso de uso 4.

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

El quinto caso de uso muestra cómo 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. 4.2). La Figura 4.18 recoge en una tabla los principales aspectos de este caso de uso. La secuencia normal de ejecución es la siguiente:

1. La pestaña *Mission* debe estar activa. Para ello hay que pinchar en el menú con sí 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 automáticamente 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. 4.2) 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.

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 desde un 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. 4.2). 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. 4.2 y 4.2).

La Figura 4.19 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 se podrá observar los parámetros de vuelo pinchando sobre la opción *Attitude* que desplegará una ventana 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. 4.2).

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. 4.2 y 4.2).

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. 4.2).

CU-05	Creación de una misión por patrón y posterior validación de la misión.	
Versión	v1.0 - 25/01/2020 - Pedro Arias Pérez	
Dependencias	<ul style="list-style-type: none"> • CU-02: Carga de mapas locales. • CU-01: Caracterización de una aeronave y su carga de pago. 	
Descripción	La aplicación deberá comportarse como se describe en el caso de uso cuando <i>el operario cree una misión por patrón y la validación de la misma</i> .	
Precondición	La aplicación debe estar abierta y en ejecución. La aeronave debe estar caracterizada siguiente el caso de uso <i>Caracterización de una aeronave y su carga de pago</i> (CU-01).	
Secuencia normal	Paso	Acción
	1	El operario pincha sobre <i>Mission</i> para desplegar dicha pestaña.
	2	El operario introduce la latitud y la longitud de la posición deseada y pincha en <i>Search</i> para obtener el mapa en dicha posición.
	3	El operario pincha sobre <i>Pattern</i> para abrir el creador de misiones por patrón.
	4	El operario introduce la altura de vuelo deseada en el campo <i>Height</i> para la misión de patrón.
	5	El operario pincha sobre el mapa para añadir un vértice del polígono en la posición deseada.
	6	El operario repetirá el Paso 5 tantas veces como vértices desee añadir al polígono.
	7	El operario pincha sobre el botón <i>Close Pattern</i> para generar la ruta de escaneo y la misión por patrón.
	8	Si el operario quisiera guardar la misión,
	8.1	El operario debe pinchar sobre el botón <i>Save</i> para guardar de forma persistente la misión. Una ventana de guardado emergirá.
	8.2	Sobre la ventana emergente el operario debe elegir un nombre y una ubicación de guardado para el nuevo archivo de misión y pinchar sobre el botón <i>Save</i> .
	9	Si el operario desea validar la misión,
	9.1	El operario pincha sobre el botón <i>Submit</i> . Una ventana emergente mostrará si la misión ha sido aceptada o no.
Postcondición	-	
Excepciones	Paso	Acción
	2	Si lo desea, el usuario puede utilizar un mapa local, para ello,
	2A.1	El operario debe seguir el caso de uso <i>Carga de un mapa local en una determinada posición</i> (CU-02).
	2	Si el operario no dispone de conexión a internet, de un mapa local de la zona o el mapa no ha sido cacheado previamente,
	2E.1	Se cancela el caso de uso. La acción no es realizable.
	5	Si el operario desea introducir los vértices mediante latitud y longitud,
	5A.1	El operario pincha sobre el botón <i>Add Pattern</i> para desplegar
		una ventana con la lista de vértices.
	5A.2	Sobre esta ventana, el operario introduce en los campos <i>Lat</i> y <i>Lon</i> la latitud y longitud deseada para el nuevo vértice.
	5A.3	El operario pincha sobre <i>Add</i> para añadir el nuevo vértice al polígono.
	5	Si el usuario introduciera erróneamente alguno de los vértices,
	5E.1	El operario pincha sobre el botón <i>Clear</i> para borrar los vértices.
	5E.2	El operario debe repetir este caso de uso para crear de nuevo una misión valida.
	7	Si la aeronave no ha sido caracterizado previamente,
	7E.1	Una ventana emergente mostrará al usuario que un modelo de configuración de la aeronave no ha sido enviado.
	7E.2	El usuario debe seguir el caso de uso <i>Caracterización de una aeronave y su carga de pago</i> (CU-01) y después continuar creando la misión por patrón siguiendo el caso de uso actual.
Comentarios	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 sobreescribirá al archivo al guardar los cambios (al pinchar en el botón <i>Save</i>) y NO creará un archivo de misión nuevo.	

Figura 4.18: Caso de uso 5.

CU-06	Carga de una misión en fichero, envío y seguimiento de la misión.	
Versión	v1.0 - 25/01/2020 - Pedro Arias Pérez	
Dependencias	<ul style="list-style-type: none"> • CU-04: Establecimiento de conexión con la aeronave. • CU-03: Creación y posterior guardado en fichero de una misión polilínea. • CU-05: Creación de una misión por patrón y validación de la misión. • CU-01: Caracterización de una aeronave y su carga de pago. 	
Descripción	La aplicación deberá comportarse como se describe en el caso de uso cuando <i>el operario cargue, envíe y realice el seguimiento sobre una misión</i> .	
Precondición	La aplicación debe estar abierta y en ejecución. La conexión con la aeronave debe estar establecida siguiendo el caso de uso <i>Establecimiento de conexión con la aeronave</i> (CU-04). Una misión debe haber sido creada y guardada en un fichero de misión previamente según los casos de uso CU-03 o CU-05.	
Secuencia normal	Paso	Acción
	1	El operario pincha sobre <i>Mission</i> para desplegar dicha pestaña.
	2	El operario pincha sobre el botón <i>Load</i> para cargar un archivo de misión.
	3	Sobre la ventana emergente, el operario selecciona el archivo de misión deseado y pincha sobre el botón <i>Open</i> .
	4	El operario debe pinchar en el botón <i>Submit</i> para validar la misión. Una ventana emergente mostrará si la misión ha sido aceptada o no. Tras aceptar la misión, se mostrará la pestaña <i>Checklist</i> .
	5	El operario debe completar y validar cada una de las caja de validación. Tras ello, el usuario pinchará sobre el botón <i>Send mission</i> para enviar la misión. Tras enviar la misión, se mostrará la pestaña <i>Follow</i> .
	6	El operario debe armar la aeronave pinchando sobre el botón <i>Arm</i> y después pinchar sobre el botón <i>Fly</i> para despegar e iniciar la misión.
	7	Si el operario desea observar los parámetros de vuelo,
	7.1	El operario pinchará sobre el cuadro de validación <i>Attitude</i> para desplegar una ventana con los útiles de navegación.
Postcondición	-	
Excepciones	Paso	Acción
	3	Si el archivo de misión seleccionado es una misión por patrón,
	3E.1	La aeronave debe haber sido previamente caracterizada. El operario debe seguir el caso de uso <i>Caracterización de una aeronave y su carga de pago</i> (CU-01).
	4	Si la misión no ha sido aceptada,
	4E.1	El operario deberá crear una misión válida o cargar alguna misión válida. Para crear una misión válida el operario debe seguir los casos de uso CU-03 o CU-05.
	5	Si la conexión no ha sido establecida,
	5E.1	Un aviso mostrará al operario que debe establecer la conexión previamente. El operario puede seguir el caso de uso <i>Establecimiento de conexión con la aeronave</i> (CU-04).
Comentarios	-	

Figura 4.19: Caso de uso 6.

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. 4.2) se explica el procedimiento para establecer la conexión, mientras que en el sexto caso de uso (Secc. 4.2) 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 la Figura 4.20. La secuencia de ejecución en condiciones normales es la siguiente:

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.

CU-07	Interrupción de una misión y vuelta a casa.				
Versión	v1.0 - 25/01/2020 - Pedro Arias Pérez				
Dependencias	<ul style="list-style-type: none"> CU-04: Establecimiento de conexión con la aeronave. CU-06: Carga de una misión en fichero, envío y seguimiento de la misión. 				
Descripción	La aplicación deberá comportarse como se describe en el caso de uso cuando <i>el operario decide interrumpir una misión en curso e iniciar una vuelta a casa</i> .				
Precondición	La aplicación debe estar abierta y en ejecución. La conexión con la aeronave debe estar establecida siguiendo el caso de uso <i>Establecimiento de conexión con la aeronave</i> (CU-04). Una misión debe estar en curso, para realizar una misión el caso de uso <i>Carga de una misión en fichero, envío y seguimiento de la misión</i> (CU-06) explica el procedimiento.				
Secuencia normal	Paso	Acción			
	1	El operario pincha sobre <i>Follow</i> para desplegar dicha pestaña.			
Postcondición	2	El operario pincha sobre el botón <i>RTL</i> para iniciar la vuelta a casa. La aeronave cambiará su modo de vuelo a <i>RTL</i> e iniciará el retomo.			
	-				
Excepciones	Paso	Acción			
	2	Si el operario decide anular la vuelta a casa, <table border="1" style="margin-left: 20px;"> <tr> <td>2A.1.1</td> <td>El operario puede pinchar sobre el botón <i>Loiter</i> para mantener la posición actual.</td> </tr> <tr> <td>2A.1.2</td> <td>El operario puede pinchar sobre el botón <i>Resume</i> para restaurar la misión interrumpida.</td> </tr> </table>	2A.1.1	El operario puede pinchar sobre el botón <i>Loiter</i> para mantener la posición actual.	2A.1.2
2A.1.1	El operario puede pinchar sobre el botón <i>Loiter</i> para mantener la posición actual.				
2A.1.2	El operario puede pinchar sobre el botón <i>Resume</i> para restaurar la misión interrumpida.				
Comentarios	-				

Figura 4.20: Caso de uso 7.

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

Finalmente, el último caso de uso presentado muestra cómo cambiar dinámicamente el idioma del texto de la aplicación. La aplicación debe estar abierta y en ejecución. La Figura 4.21 incluye una tabla con este 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 menú de fichero de la aplicación.
2. Sobre la ventana de configuración desplegada se debe elegir en la lista despegable *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* que desplegaría la ventana sin necesidad de abrir el menú de fichero.

CU-08	Cambio del idioma de la aplicación.	
Versión	v1.0 - 25/01/2020 - Pedro Arias Pérez	
Dependencias	•	
Descripción	La aplicación deberá comportarse como se describe en el caso de uso cuando <i>el operario decida cambiar dinámicamente el idioma de la aplicación</i> .	
Precondición	La aplicación debe estar abierta y en ejecución.	
Secuencia normal	Paso	Acción
	1	El operario accede a la ventana secundaria de configuración siguiendo la ruta <i>File → Settings → More Settings</i> en el menú de fichero de la aplicación.
	2	Sobre la ventana de configuración el operario elige en la lista despegable <i>Language</i> uno de los idiomas disponibles. El operario confirma el cambio de idioma pinchando sobre el botón <i>OK</i> .
Postcondición	-	
Excepciones	Paso	Acción
	1	Para acceder a la ventana de configuración,
	2A.1	El operario puede introducir el atajo <i>Ctrl+S</i> .
Comentarios	Es posible que idiomas distintos al idioma base (Inglés) contengan errores o funciones no traducidas.	

Figura 4.21: Caso de uso 8.

Capítulo 5

Validación Experimental

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 planteados en este proyecto se cumplen.

Además de las pruebas unitarias de diferentes partes de la aplicación realizadas para verificar los casos de uso, en este capítulo se detallan las pruebas integrales de vuelo realizadas. Estas pruebas se han realizado tanto en simulación como con la aeronave real.

Principalmente, se han realizado dos pruebas de vuelo. Ambos experimentos en simulación y con una aeronave real de ala rotatoria, el 3DR Solo Drone (ver Figura 2.4a). En uno se ha realizado con una misión polilínea mientras que el otro con una misión por patrón. Estos test de vuelo de la aplicación se verán con profundidad en las secciones de este capítulo.

También se adjunta una lista de reproducción¹ que recoge la evolución de la aplicación a lo largo del tiempo con las distintas versiones realizadas. Aproximadamente cada mes se ha grabado un vídeo con las principales novedades de desarrollo durante ese periodo. Esta lista muestra una visión de las características de la aplicación y de los tiempos de desarrollo empleados.

Para realizar las pruebas de vuelo en real ha sido necesario desplazarse a un campo de vuelo donde se cumpliese la legislación actual. En campo de vuelo elegido ha sido el de Seseña [50]. La Figura 5.1 muestra parte de las instalaciones y el despliegue realizado durante las pruebas.



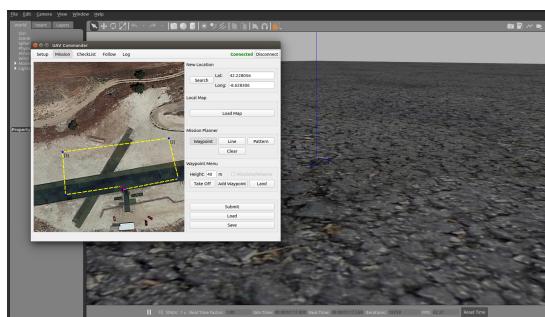
Figura 5.1: Campo de vuelo de Seseña.

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

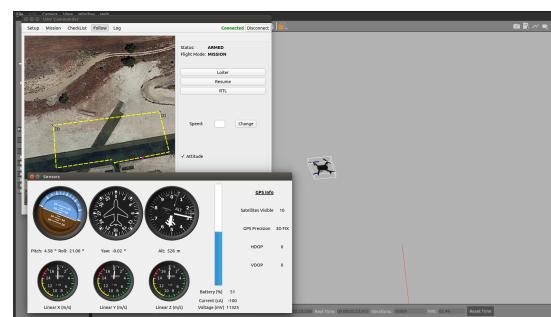
5.1. Vuelo en simulador con misiones polilínea.

Este experimento realiza en un entorno de simulación la creación de una misión polilínea, el envío a la aeronave y el seguimiento de la misión. La simulación permite la detección de errores sin poner en riesgo a la aeronave. La Figura 5.2 contiene unas capturas de pantalla realizadas durante la simulación. El siguiente enlace ² muestra un vídeo con la simulación realizada. La secuencia de ejecución seguida ha sido:

1. Lanzamiento de la simulación mediante un SITL de PX4 con Gazebo9. Ejecución de la aplicación.
2. Establecimiento de la conexión entre estación terrestre y aeronave (ver caso de uso 4.2).
3. Caracterización de la aeronave y carga de pago (ver caso de uso 4.2). Este paso, pese haberlo realizado, resulta prescindible para las misiones polilínea.
4. Creación de la misión polilínea (ver caso de uso 4.2 y Fig. 5.2a).
5. Validación de la misión creada.
6. Comprobaciones pre-vuelo y envío de la misión a la aeronave.
7. Inicio de la misión y seguimiento de la misma (ver caso de uso 4.2 y Fig. 5.2b).
8. Aterrizaje y fin de la misión.



(a) Simulación de la creación de la misión.



(b) Simulación del seguimiento de la misión.

Figura 5.2: Simulación de vuelo para misiones polilínea.

Tras realizar la prueba en simulación y comprobar el buen funcionamiento de la aplicación se considera que el software ya está listo para realizar pruebas en real. Esta prueba en real se muestra en la siguiente sección.

5.2. Vuelo real con misiones polilínea.

Este experimento reproduce un supuesto en el cual un operario realiza una misión polilínea *in situ*, la envía a la aeronave y realiza el seguimiento de la misma. Para documentar la prueba de vuelo se ha generado un vídeo del que se adjunta su enlace ³. La Figura 5.3 muestra una

²https://youtu.be/SEdT_Rq_59A

³https://youtu.be/UOv8Go_Q22k

serie de fotogramas del vídeo mencionado.

La secuencia de ejecución seguida durante el experimento ha sido la siguiente:

1. Ejecución de la aplicación y encendido del 3DR Solo junto a su emisora.
2. Establecimiento de conexión entre ambos cuando el 3DR Solo ha establecido señal GPS (ver caso de uso 4.2).
3. Caracterización de la aeronave y carga de pago (ver caso de uso 4.2). Este paso, pese haberlo realizado, resulta prescindible para las misiones polilínea.
4. Creación de la misión polilínea (ver caso de uso 4.2 y Fig. 5.3a).
5. Validación de la misión creada.
6. Comprobaciones pre-vuelo y envío de la misión a la aeronave (Fig. 5.3b).
7. Inicio de la misión y seguimiento de la misma (ver caso de uso 4.2 y Fig. 5.3c).
8. Aterrizaje y fin de la misión (Fig. 5.3d).

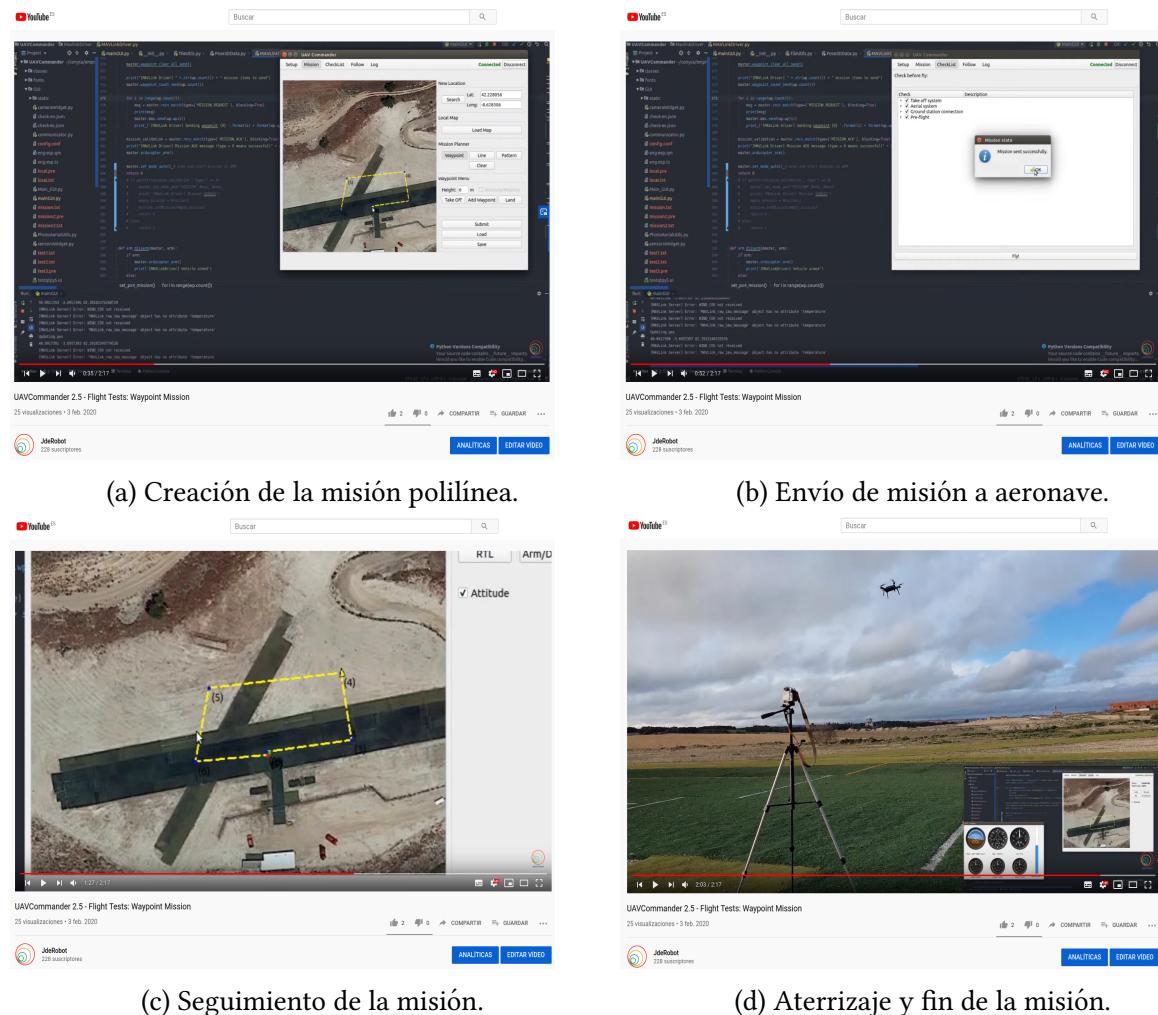


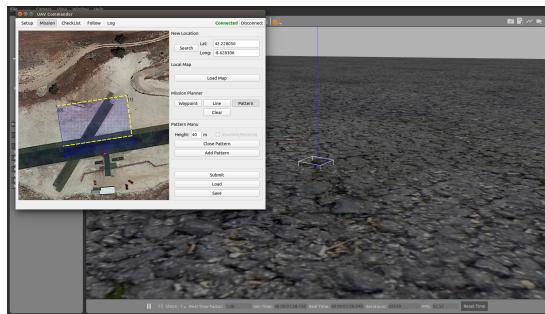
Figura 5.3: Vídeo con la prueba de vuelo para misiones polilínea.

Tras realizar esta prueba integral se concluye que la aeronave completa satisfactoriamente la misión polilínea. El drone despegue, recorre la secuencia de puntos de paso y aterriza en los puntos de paso especificados por el usuario a través de la aplicación desarrollada. Además, la posición de la aeronave se puede seguir en todo momento a través de la estación de tierra. Lo mismo sucede con otros parámetros de navegación como las velocidades o la actitud de la aeronave.

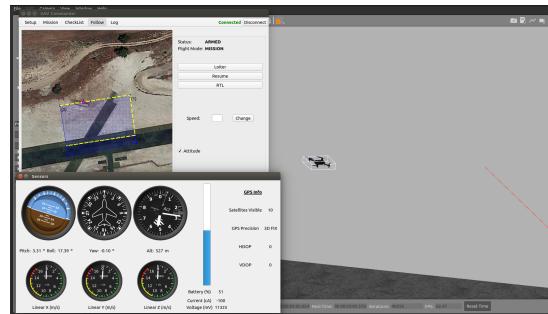
5.3. Vuelo en simulador con misiones por patrón.

Este experimento realiza en un entorno de simulación la creación de una misión por patrón, el envío a la aeronave y el seguimiento de la misión. La simulación permite la detección de errores sin poner en riesgo a la aeronave. La Figura 5.4 contiene unas capturas de pantalla realizadas durante la simulación. El siguiente enlace ⁴ muestra un vídeo con la simulación realizada. La secuencia de ejecución seguida ha sido:

1. Lanzamiento de la simulación mediante un SITL de PX4 con Gazebo9. Ejecución de la aplicación.
2. Establecimiento de la conexión entre estación terrestre y aeronave (ver caso de uso 4.2).
3. Caracterización de la aeronave y carga de pago (ver caso de uso 4.2).
4. Carga de una misión por patrón previamente confeccionada (ver caso de uso 4.2 y Fig. 5.4a).
5. Validación de la misión creada.
6. Comprobaciones pre-vuelo y envío de la misión a la aeronave.
7. Inicio de la misión y seguimiento de la misma (ver caso de uso 4.2 y Fig. 5.4b).
8. Aterrizaje y fin de la misión.



(a) Simulación de la creación de la misión.



(b) Simulación del seguimiento de la misión.

Figura 5.4: Simulación de vuelo para misiones por patrón.

Tras realizar la prueba en simulación y comprobar el buen funcionamiento de la aplicación se considera que el software ya está listo para realizar pruebas en real. Esta prueba en real se muestra en la siguiente sección.

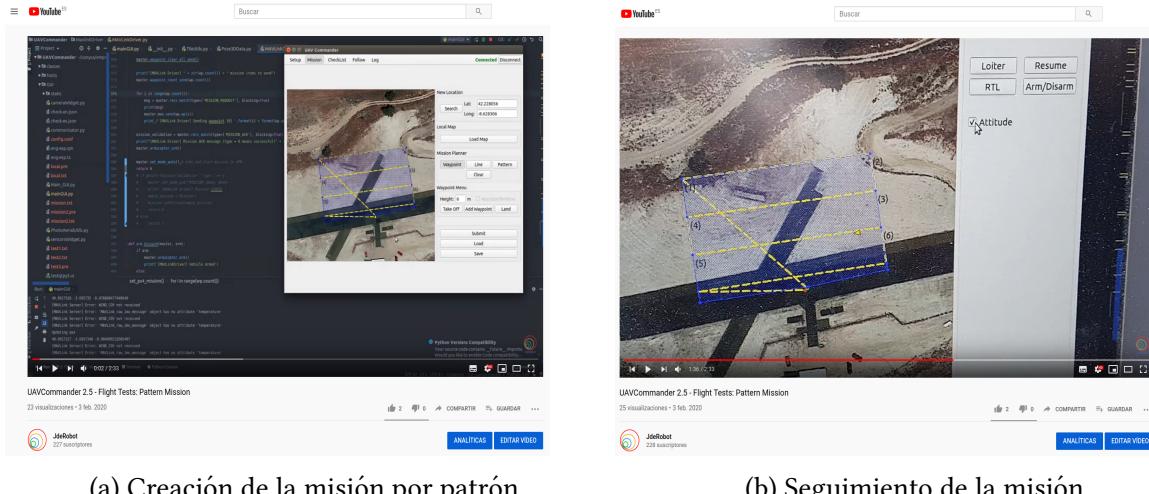
⁴<https://youtu.be/0FcyvPnj5B0>

5.4. Vuelo real con misiones por patrón.

Este ensayo de vuelo reproduce una situación en la cual un operario carga una misión por patrón previamente creada, la envía a la aeronave y realiza el seguimiento de la misma. Se adjunta un enlace de un vídeo⁵ con el experimento real. Además, la Figura 5.3 muestra un fotograma del vídeo adjunto.

La secuencia de ejecución seguida durante la prueba de vuelo ha sido la siguiente:

1. Ejecución de la aplicación y encendido del 3DR Solo junto a su emisora.
2. Establecimiento de conexión entre ambos cuando el 3DR Solo ha establecido señal GPS (ver caso de uso 4.2).
3. Caracterización de la aeronave y carga de pago (ver caso de uso 4.2).
4. Carga de una misión por patrón previamente confeccionada (ver caso de uso 4.2).
5. Validación de la misión creada.
6. Comprobaciones pre-vuelo y envío de la misión a la aeronave.
7. Inicio de la misión y seguimiento de la misma (ver caso de uso 4.2).
8. Aterrizaje y fin de la misión.



(a) Creación de la misión por patrón.

(b) Seguimiento de la misión.

Figura 5.5: Vídeo con la prueba de vuelo para misiones por patrón.

Tras realizar esta prueba integral se concluye que la aeronave completa satisfactoriamente la misión por patrón. La ruta se genera automáticamente tras seleccionar la región de escaneo. El drone recorre dicha ruta según la forma indicada. Además, la posición de la aeronave se puede seguir en todo momento a través de la estación de tierra. Lo mismo sucede con otros parámetros de navegación como las velocidades o la actitud de la aeronave.

⁵<https://youtu.be/tppmpdkG0CjM>

Capítulo 6

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 futuras versiones de la aplicación.

6.1. Conclusiones

El desarrollo concluye en una nueva 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 precomercial. 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 satisfecho.

El producto final desarrollado consiste en una aplicación con cerca de 6000 líneas de código en lenguaje Python dividido en diferentes ficheros. La aplicación se ha desarrollado siguiendo un control de versiones y una metodología en espiral con incidencias y parches. Además, el procedimiento seguido incluye validaciones semanales con los tutores.

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 validación experimental (Cap. 5). 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:

- 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.

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

- 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 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 en vuelo. La cuarta pestaña de la aplicación cumplimenta este requisito. El seguimiento permite visualizar la posición de la aeronave sobre el mapa en todo momento 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 desarrollado siguiendo la línea de aplicaciones existentes similares (como *QGroundControl* o *MissionPlanner*), pero manteniendo la sencillez allí donde las principales soluciones del mercado pecan al incluir multitud de opciones y configuraciones. Así pues, el operario solo interacciona con la aplicación cuando es estrictamente necesario, y los automatismos se tratan de llevar a máximos.

6.2. Líneas futuras

Este trabajo desarrolla un 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 de la aplicación en pantalla, 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.
- Internacionalización, dar soporte a la aplicación en otros idiomas como portugués, francés, etc.

Bibliografía

- [1] ArduPilot. Pymavlink. <https://github.com/ArduPilot/pymavlink>. Último acceso: 10-02-2020.
- [2] 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.
- [3] Aníbal Ollero Baturone. *Robótica: manipuladores y robots móviles*. Marcombo, 2005.
- [4] Diego Jiménez Bravo. Drones con protocolo mavlink en el entorno jderobot. <https://github.com/RoboticsLabURJC/2016-tfg-Diego-Jimenez>. Último acceso: 24-02-2020.
- [5] 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.
- [6] Marek Cel and Aitor Martinez (JdeRobot). Qflightinstruments. <https://github.com/JdeRobot/ThirdParty/tree/master/qflightinstruments>. Último acceso: 10-02-2020.
- [7] Alex Clark and Contributors. Pil. <https://pillow.readthedocs.io/en/stable/>. Último acceso: 10-02-2020.
- [8] The Qt Company. Qt5. <https://www.qt.io/>. Último acceso: 10-02-2020.
- [9] Ministerio de Fomento. Instituto geográfico nacional. <https://www.ign.es/web/ign/portal>. Último acceso: 10-02-2020.
- [10] Jonathan Ruiz de Garibay Pascual. Robótica: Estado del arte. *Universidad de Deusto. Número. Fecha*, page 54, 2006.
- [11] Grupo de robótica de la URJC. Twitter @roboticslaburjc. <https://twitter.com/RoboticsLabURJC?lang=es>. Último acceso: 24-01-2020.
- [12] NumPy developers. Numpy. <https://numpy.org/>. Último acceso: 10-02-2020.
- [13] Inc. Dronecode Project. Bucle de control de un uav. <https://dev.px4.io/master/en/concept/architecture.html>. Último acceso: 24-02-2020.
- [14] 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.

- [15] Inc. Dronecode Project. Comandos *MAV_CMD*. https://mavlink.io/en/messages/common.html#mav_commands. 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. Microservicios de mavlink. <https://mavlink.io/en/services/>. Último acceso: 02-03-2020.
- [23] Inc. Dronecode Project. Protocolo de misiones de mavlink. https://mavlink.io/en/services/mission.html#uploading_mission. Último acceso: 22-02-2020.
- [24] Inc. Dronecode Project. Px4. <https://px4.io/>. Último acceso: 11-02-2020.
- [25] Real Academia Española. Diccionario de la lengua española, 23.^º ed. <https://dle.rae.es>. Último acceso: 02-02-2020.
- [26] Open Source Robotics Foundation. Gazebo. <http://gazebosim.org/>. Último acceso: 11-02-2020.
- [27] Python Software Foundation. Collections python library. <https://docs.python.org/3/library/collections.html>. Último acceso: 10-02-2020.
- [28] Python Software Foundation. Datetime python library. <https://docs.python.org/3/library/datetime.html>. Último acceso: 10-02-2020.
- [29] Python Software Foundation. Json python library. <https://docs.python.org/3/library/json.html>. Último acceso: 10-02-2020.
- [30] Python Software Foundation. Math python library. <https://docs.python.org/3/library/math.html>. Último acceso: 10-02-2020.
- [31] Python Software Foundation. Pyhton. <https://www.python.org/>. Último acceso: 09-02-2020.
- [32] Python Software Foundation. Threading python library. <https://docs.python.org/3/library/threading.html>. Último acceso: 10-02-2020.
- [33] Python Software Foundation. Time python library. <https://docs.python.org/3/library/time.html>. Último acceso: 10-02-2020.

- [34] Python Software Foundation. Urllib. <https://docs.python.org/3/library/urllib.html>. Último acceso: 10-02-2020.
- [35] Geodrone. Geodrone mapper. <http://geodrone.es/>. Último acceso: 24-01-2020.
- [36] Inc. GitHub. Github. <https://github.com/>. Último acceso: 26-02-2020.
- [37] Klokan Technologies GmbH and OSM community. Open map tiles. <https://openmaptiles.org/about/>. Último acceso: 10-02-2020.
- [38] JdeRobot. Asociación de robótica jderobot. <https://jderobot.github.io/>. Último acceso: 02-03-2020.
- [39] Riverbank Computing Limited. Pyqt5. <https://riverbankcomputing.com/software/pyqt/intro>. Último acceso: 10-02-2020.
- [40] Google LLC. Google map server. <https://developers.google.com/maps/documentation?hl=es>. Último acceso: 10-02-2020.
- [41] Auterion Ltd. Pixhawk. <https://pixhawk.org/>. Último acceso: 07-03-2020.
- [42] Canonical Ltd. Ubuntu 18.04.3 lts. <https://ubuntu.com/download/desktop>. Último acceso: 09-02-2020.
- [43] 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.
- [44] Instituto Geográfico Nacional. Plan nacional de ortofotografía aérea. <https://pnoa.ign.es/>. Último acceso: 10-02-2020.
- [45] 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.
- [46] OSGeo. Gdal. <https://www.osgeo.org/projects/gdal/>. Último acceso: 10-02-2020.
- [47] I Otero, A Ezquerra, R Rodríguez-Solano, L Martín, and I Bachiller. Fotogrametría. 2010.
- [48] 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.
- [49] 3DR Robotics. 3dr solo drone. <https://newthreedee.wengine.com/>. Último acceso: 11-02-2020.
- [50] R.C. Seseña. Campo de vuelo de seseña. <https://aeromodelismosesena.wordpress.com/>. Último acceso: 26-02-2020.
- [51] JetBrains s.r.o. Pycharm. <https://www.jetbrains.com/es-es/pycharm/>. Último acceso: 26-02-2020.
- [52] Ardupilot Development Team and Community. Ardupilot. <https://ardupilot.org/>. Último acceso: 11-02-2020.