



ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Curso Académico 2016/2017

Proyecto Fin de Carrera

Navegación por posición para un avión autónomo con
JdeRobot

Autor : José Antonio Fernández Casillas

Tutor : José María Cañas Plaza

Proyecto Fin de Carrera

Navegación por posición para un avión autónomo con JdeRobot

Autor : José Antonio Fernández Casillas **Tutor :** José María Cañas Plaza

La defensa del presente Proyecto Fin de Carrera se realizó el día de
de 2017, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2017

*A mis padres y a mi mujer,
sin vuestra perseverancia,
apoyo y paciencia
nada de esto
hubiese sido posible.
A mi hijo Pablo*

Resumen

Los UAV o Drones se han popularizado en los últimos años hasta el punto de formar parte de nuestro día a día con aplicaciones en muchos ámbitos de nuestra vida. El objetivo de este trabajo final es desarrollar tecnología que permita la programación de aplicaciones robóticas con un avión autónomo. Este proyecto se ubica en el contexto de robótica y drones, en robótica aérea. En él se documenta el análisis, diseño e implementación de un driver y una aplicación de teleoperación por puntos de paso en JdeRobot. A lo largo del mismo se detallan tanto el hardware como el software utilizados para la implementación y se describen los puntos más importantes. Como plataforma robótica se ha empleado JdeRobot 5.3. El lenguaje de programación ha sido Python y se han empleado las bibliotecas OpenCV, OWSLib y PyQt, para tratamiento de imágenes, apoyo a la descarga de mapas y para construir la interfaz gráfica de usuario de la aplicación.

Índice general

1. Introducción	1
1.1. Robótica Aérea	1
1.2. Tipos de Aeronaves	4
1.3. Aplicaciones	7
1.4. Robótica aérea en el laboratorio de Robótica de la URJC	13
2. Objetivos	19
2.1. Problema abordado	19
2.2. Metodología	20
3. Infraestructura utilizada	23
3.1. Hardware	23
3.2. JdeRobot	25
3.2.1. Interfaces	26
3.3. Protocolo MAVLink	28
3.4. Biblioteca pymavlink	29
3.5. Lenguaje Python y biblioteca PyQt5	30
3.6. Mapas y Geo-referenciación	30
3.7. Simulador SITL	32
4. Driver APM Server	35
4.1. Diseño	35
4.2. Bloque de conexión MAVLink	37
4.2.1. Listado de mensajes MAVLink	37
4.2.2. Conexión y configuración de la comunicación con el APM	43

4.2.3.	Lectura de datos del APM	45
4.2.4.	Envío de misiones al APM	46
4.3.	Capa de conexión con aplicaciones JdeRobot	48
4.4.	Bloque de interpretación	50
5.	Aplicación AUV Commander	53
5.1.	Diseño	53
5.2.	Bloque de sensores	56
5.2.1.	Cámara	57
5.2.2.	Sensores de vuelo	57
5.3.	Bloque de Mapas	58
5.3.1.	Obtención de Mapas	59
5.3.2.	Conversión de coordenadas	59
5.3.3.	Visualización de mapas	62
5.4.	Bloque de misiones	62
5.4.1.	Creación y envío de misiones	63
5.4.2.	Control del vuelo	65
6.	Experimentos	69
6.1.	Experimentos con misiones en simulador	69
6.1.1.	Integración de todo el software	69
6.1.2.	Envío de misión con avión en vuelo	70
6.1.3.	Envío de misión con despegue y varios puntos de paso	72
6.1.4.	Envío de misión completa con despegue y aterrizaje	72
6.1.5.	Experimento de autozoom ante la salida del avión del mapa	73
6.2.	Interconexión del driver y aplicación con el avión real	74
6.2.1.	Experimento de recepción de vídeo	74
6.2.2.	Recepción de información sensorial y seguimiento del avión a través del UAV Commander	75
6.2.3.	Pruebas de alcance del enlace entre el avión y el ordenador base	76
6.2.4.	Experimento de comportamiento ante pérdidas de señal	77

<i>ÍNDICE GENERAL</i>	VII
7. Conclusiones	79
7.1. Conclusiones	79
7.2. Trabajos futuros	81
Bibliografía	83

Índice de figuras

1.1. Infografía de la Agencia EFE con la regulación	5
1.2. Drone grabando en el interior de un cráter	8
1.3. Drone Yamaha RMAX fumigando	9
1.4. Drone Volt Z18 UF	10
1.5. AR Drone 2	14
1.6. Ventanas de UAV Viewer	14
1.7. AR Drone Simulado en Gazebo	15
1.8. SOLO Drone de 3D Robotics	16
1.9. Drone teleoperado con teléfono móvil	17
2.1.	20
2.2. Evolución de nuestra pizarra Kanban	21
3.1. Placa estabilizadora ardupilot	24
3.2. Imágen geo-referenciada del club de aeromodelismo Icaro	31
3.3. Arquitectura de SITL	33
4.1. Diseño de entradas y salidas de APM Server	36
4.2. Bloques del driver APM Server	37
4.3. Conexión con el APM	44
4.4. configuración de la conexión	44
4.5. Hilo de alimentación de mensajes MAVLink	45
4.6. Protocolo de envío de misiones MAVLink	46
4.7. Creación y arranque de los hilos encargados de servir los interfaces ICE	49
4.8. Servidor ICE encargado de servir objetos de la interfaz Pose3D	50

4.9. Código encargado de detectar misiones entrantes	51
4.10. Código de ejemplo de <code>refreshAPMPose3D()</code> encargado de recorrer los mensajes y obtener los datos de actitud que se van a servir a través de Pose3D .	52
5.1. Diseño de caja negra de UAV Commander, entradas y salidas	54
5.2. Esquema de conexión UAV Commander	54
5.3. Bloques dentro de la aplicación UAV Commander	55
5.4. Vista general del interfaz gráfico de UAV Commander	56
5.5. Ventana Camera de UAV Commander	57
5.6. Creación del servicio ICE camera y asignación al GUI	57
5.8. Creación de servicios ICE y asignación al GUI	58
5.7. Ventana actitud y batería de UAV Commander	58
5.9. Método <code>retrieve_new_map(lat, lon, radius, width, height)</code>	60
5.10. Método <code>getBoundingBox(lat, lon, distance)</code> encargado de calcular la esquina inferior izquierda y la esquina superior derecha de un cuadrado de ancho <code>distance</code>	60
5.11. Sistemas de referencia utilizados	61
5.12. Método <code>posCoords2Image(lonMin, latMin, lonMax, latMax, lat, lon, tamImageX, tamImageY)</code> , encargado de calcular la proyección de un punto en coordenadas geográficas sobre la matriz de píxeles de la imagen que representa el mapa	62
5.13. Código encargado de mostrar por pantalla el mapa	62
5.14. Pantalla principal de UAV Commander	63
5.15. Método <code>set_waypoints(self, wayPoints, currentTrue)</code> , encargado de pintar los puntos de paso en el mapa	64
5.16. Método <code>sendWP(self, send2APM)</code> , encargado de enviar la misión creada por el usuario al driver APM Server	65
5.17. Métodos encargados de actualizar la posición y el recorrido efectuado en el mapa	68
6.1. Seguimiento del experimento 6.1.2 dentro de la aplicación UAV Commander .	71
6.2. Seguimiento del experimento 6.1.3 dentro de la aplicación UAV Commander .	72
6.3. Seguimiento del experimento 6.1.3 dentro de la aplicación UAV Commander .	73

6.4. Autozoom en UAV Commander	74
6.5. Prueba de recepción de vídeo desde cameraserver a UAV Commander	75
6.6. Seguimiento del experimento 6.2.3 dentro de la aplicación UAV Commander .	76
6.7.	77

Capítulo 1

Introducción

Los UAV o Drones se han popularizado en los últimos años hasta el punto de formar parte de nuestro día a día con aplicaciones en muchos ámbitos de nuestra vida. Si bien se están utilizando ya de forma habitual en sectores como el cine o la ingeniería civil, aún se están explorando muchas de las posibles utilidades que estos robots pueden ofrecer.

El objetivo de este trabajo es desarrollar tecnología que permita la programación de aplicaciones robóticas con un avión autónomo. Este proyecto se ubica por lo tanto en el contexto de robótica y drones, en robótica aérea.

1.1. Robótica Aérea

Los orígenes de la robótica aérea tienen origen militar y su avance ha estado intrínsecamente ligado a este ámbito durante todo el siglo XX. Se consideran el origen de los aviones no tripulados los experimentos llevados a cabo a principios del siglo XX durante la 1ª guerra mundial como el “*Aerial Target*” desarrollado por el capitán A. H. Low para su uso como blanco aéreo. Si bien eran vehículos no tripulados (*Unmanned Aerial Vehicles*) no eran autónomos y eran manejados desde tierra a través de una radio. No es hasta el final del siglo XX cuando bajo el escenario de la guerra de Vietnam y ante la creciente pérdida de vidas de los pilotos, estos vehículos vuelven de nuevo a ser objeto de desarrollo y se conviertan en vehículos autónomos.

Desde ese momento y hasta nuestros días se utilizan de forma habitual en el ámbito militar en misiones de reconocimiento, bombardeos o apoyo sin arriesgar vidas humanas.

A los largo de los primeros años de este siglo, debido al abaratamiento de los componentes

electrónicos y a su minituarización y potencia, la robótica aérea se ha 'desmilitarizado' y está experimentado un enorme crecimiento en aplicaciones civiles.

Hoy en día es común encontrar en cualquier juguetería quadracópteros radio-pilotados por poco menos de 30 euros y en tiendas especializadas podemos encontrarlos ya con el hardware y software integrados que les permiten seguir una serie de puntos de control y comportarse de forma autónoma por poco más de 200 €.

El uso de AUV o drones se ha popularizado tanto que es una de las industrias en las que más ha crecido la inversión, y es que, según la empresa analista especializada en drones Droneii con sede en Hamburgo en un estudio sobre la inversión en el sector[3], en Europa se invirtió en proyectos domésticos en 2016 cerca de 65 millones de dólares incrementándose esta cifra hasta los 314 millones si atendemos al mercado norteamericano.

Estos datos se asientan en un mercado cada vez más extendido y con una gran proyección de crecimiento. La publicación BI Intelligence¹[1] espera que las ventas de drones alcancen los 12.000 millones en 2021. La venta de drones es sólo una de las piezas de este negocio incipiente, empresas como DJI, Xiaomi o 3DR están en la vanguardia de la innovación de éstos, pero es tan sólo la punta del iceberg. Esta industria está potenciando otras como la de las videocámaras deportivas GoPro e incluso están surgiendo nuevos puestos de trabajo como el de operador de drones. Las grandes empresas tecnológicas ven el potencial económico que pueden aportar a sus balances y se está produciendo una pugna por adquirir las principales empresas especializadas en drones, Verizon compró en febrero Skyward, Facebook compró Acenta y Google Titan Aerospace.

La enorme aceptación y expansión de los drones se ha producido de un modo tan explosivo que en ciertos aspectos de la sociedad no se ha avanzado lo suficiente como para que su utilización se haga de forma segura.

- Problemas de seguridad. Los drones y en especial los drones de gran envergadura, más allá de su uso profesional o lúdico, pueden ser muy peligrosos. Sus palas giran a más de 1000RPM y pueden producir cortes o amputaciones o producir daños personales y/o materiales en el caso de una pérdida de control del mismo o una caída. Esto se hizo patente cuando durante la filmación de un concierto del cantante Enrique Iglesias en Tijuana (México) éste agarró el drone que le grababa de forma espontánea y esto le produjo severos

¹<http://www.businessinsider.com>

cortes en su mano derecha que hizo que sangrase profusamente.

También pueden utilizarse con fines terroristas, como se sospechó en Francia cuando se detectaron en octubre de 2014 volando en las proximidades de varias centrales nucleares. Además, en la noche del 19 al 20 de enero de 2015 otro dron sobrevoló el Palacio del Elíseo, donde tiene su residencia el presidente de la república francesa. Y a ellos hay que sumar los 19 drones que han sido avistados sobrevolando 17 centrales nucleares francesas de octubre a febrero de 2015. Su bajo coste y su versatilidad producen que proliferen muchos drones contruidos para fines oscuros.

Algunas empresas del sector se han hecho eco de estos ataques y han deshabilitado sus drones en zonas de conflicto para evitar que se utilicen como armas de guerra como es el caso de la empresa china DJI, líder en el sector.²

A estos problemas de seguridad hay que unir el problema del *hacking*. Muchos de los más extendidos drones comerciales no tienen protección alguna contra el ataque de un ciberdelincuente. Por poner un ejemplo, el modo en el que el Parrot AR-Drone se empareja con el móvil es una conexión plana sin contraseña, hace posible conectarse a él por un tercero con unas pocas líneas de código³⁴⁵.

Existen técnicas como el *GPS Spoofing* que pueden “secuestrar” cualquier dron que se guíe por GPS como ocurrió en Irán en 2011 dónde hackers iraníes se hicieron con el control de un dron militar americano Lockheed Martin RQ-170 Sentinel⁶ mediante ésta técnica que engaña a la antena GPS del UAV haciéndole pensar que se encuentra en otro lugar. Si bien para este ataque se necesitó un hardware caro que haría el ataque prohibitivo, hoy es posible realizarlo por unos pocos cientos de euros con placas como *hackrf* o *braderf* como se demostró por la compañía china Qihoo 360 en las conferencias de Hacking DEFCON de 2015.[8]

- Problemas regulatorios. Otro problema importante que apenas se está empezando a abordar es el regulatorio. Con el fin de minimizar el riesgo para las personas se está llevando

²<http://clipset.20minutos.es/dji-bloquea-drones-guerra-irak-siria/>

³Guía sencilla de hacking del AR-Drone 2.0 - <http://www.xdrones.es/guia-para-hackear-el-ar-drone-2-0/>

⁴Cómo atacar el AR-Drone 2.0 con nodecopter - <http://www.nodecopter.com/hack>

⁵Entorno para *hackear* drones a través de WIFI - <https://github.com/samyk/skyjack>

⁶https://en.wikipedia.org/wiki/Iran%E2%80%93U.S._RQ-170_incident

a cabo una regulación del sector en todo el mundo. En España por ejemplo hasta abril de 2014, volar un drone para uso civil estaba absolutamente prohibido. La no regulación no indicaba que se pudiese volar, indicaba más bien todo lo contrario. En este marco regulatorio no se puede por ejemplo, volar un drone en núcleos urbanos, sólo debe hacerse en zonas previstas a tal efecto. Tampoco es posible volar un drone de más de 2Kg más allá de donde puedas verlo (500m), y es que antes era muy usual ponerle una cámara y volar varios kilómetros con él.

Para hacernos una idea clara del marco regulatorio actual para su uso no profesional basta con fijarnos en la siguiente Figura 1.1 publicada por la agencia EFE.

Para poder utilizar los drones con fines profesionales la ley es aún más restrictiva. La Agencia Española de Seguridad Aérea AESA exige el registro de las aeronaves civiles pilotadas por control remoto cuya masa máxima al despegue exceda de 25 kg, que deberán estar inscritas en el Registro de matrícula de aeronaves y disponer de certificado de aeronavegabilidad. Para drones de menor peso, el piloto deberá presentar ante la Agencia Estatal de Seguridad Aérea una comunicación previa y declaración responsable con una antelación mínima de cinco días al día del inicio de la operación, y éste ha de estar habilitado como operador de drones. La ley completa se puede consultar el BOE⁷

Con su uso ya ampliamente extendido en sectores como el cine, la televisión, fotografía, agropecuario, forestal, ingeniería civil y presencia en sectores como el de salvamento o seguridad y protección, el nicho de mercado de los UAV está lejos de su cima. Día a día se investigan nuevos usos en sectores como el de la logística o las telecomunicaciones, como veremos más adelante, suponiendo un reto constante para investigadores, desarrolladores, ingenieros, inversores y entidades regulatorias.

1.2. Tipos de Aeronaves

Las aeronaves son la base que permite que nuestro robot vuele, de ahí que convenga dedicar unas líneas a entender los fundamentos de las mismas y en particular las que son objeto de estudio y desarrollo en este PFC, los llamados aerodinos.

⁷http://www.seguridadaerea.gob.es/media/4243006/rdl_8_2014_4julio.pdf

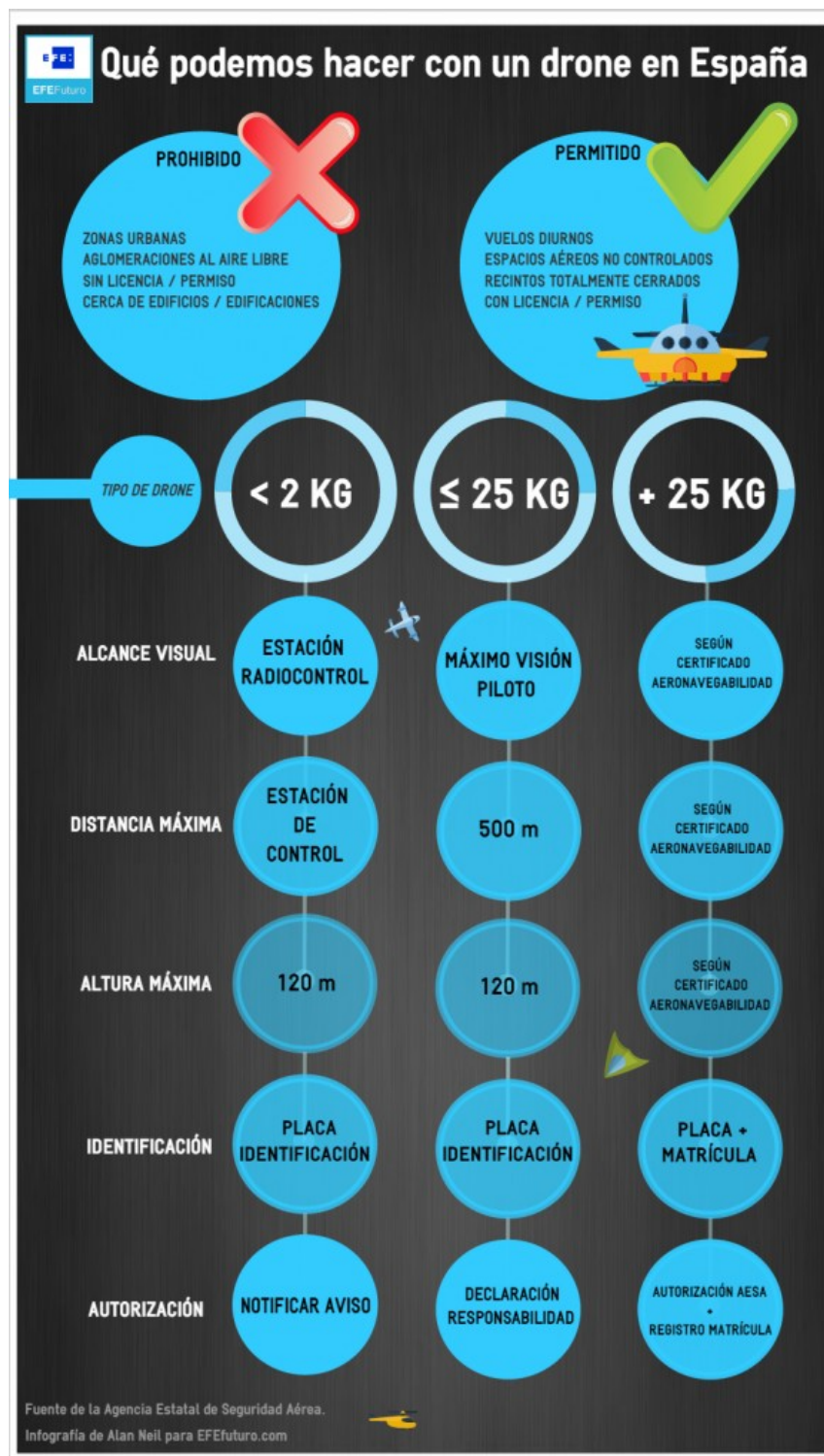


Figura 1.1: Infografía de la Agencia EFE con la regulación

Los aerodinos son aquellas aeronaves que vuelan a pesar de pesar más que el aire, son capaces de generar sustentación por sus propios medios a diferencia de los aerostatos como por ejemplo los globos aerostáticos. Existen principalmente 2 tipos de aerodinos si atendemos al modo en que generan su sustentación con sus alas: de ala fija y las de ala rotatoria.

Dentro de los primeros tenemos aquellos aerodinos que tienen sus alas fijas al fuselaje, y que comúnmente conocemos como aviones. Según la OACI, un avión es un «Aerodino propulsado por motor, que debe su sustentación en vuelo principalmente a reacciones aerodinámicas ejercidas sobre superficies que permanecen fijas en determinadas condiciones de vuelo». Algunos ejemplos de aerodinos de ala fija son los aeroplanos, planeadores/veleros, aladeltas, parapentes, paramotores y ultraligeros.

Este tipo de aerodinos tienen como principal ventaja que la carga de aire que necesitan en sus alas puede ser producida de muchas formas distintas (los veleros no tienen ningún tipo de propulsión). Esta carga es variable en función de la superficie alar del mismo y permite por tanto cargas más grandes que si instalásemos el mismo propulsor en un ala rotatoria. Pongamos como ejemplo el A380 de Airbus, es el avión de pasajeros más grande del mundo y cuenta con 4 motores que producen un empuje de entre 70.000 y 80.000lbs, unas 32-36 toneladas de empuje cada uno generando por tanto entre los 4 a máximo rendimiento y optimas condiciones alrededor de 144 toneladas de empuje. Este avión tiene un peso máximo al despegue⁸ de entre 560 y 590 toneladas. Tenemos por tanto que necesitamos en este caso $\frac{1}{4}$ del peso total en empuje para despegar este avión. Si hiciésemos este mismo ejercicio con un aerodino de ala rotatoria como el Boeing AH-64 o Apache con un peso máximo al despegue de 9,5 toneladas necesitaríamos que la combinación que realizan empuje y palas superase esos 9,5 toneladas para siguiera levantar del suelo. Este tipo de aerodinos son por tanto más eficientes, rápidos, con mayor carga de pago, mayor alcance debido a su menor consumo y más estables.

Dentro de la tipificación de ala rotatoria tenemos aquellos aerodinos que producen su sustentación con el movimiento (rotación) de sus alas. En este tipo de aerodinos las alas, también llamadas 'palas', que giran en torno a un eje produciendo con este giro la sustentación necesaria para despegar del suelo. Algunos ejemplos de este tipo de aerodinos son los helicópteros, autogiros, convertibles o los, ampliamente conocidos en robótica aérea, quadricópteros. Este tipo de aerodino tiene como principal ventaja frente a los ala fija en su versatilidad a la hora de realizar

⁸Peso máximo que es capaz de soportar un avión en su maniobra de despegue

las maniobras de despegue y aterrizaje que pueden realizarse de forma vertical (VTOL⁹), además de la capacidad de realizar vuelo estacionario¹⁰ que le hacen imprescindible en escenarios poco accesibles o donde no es posible aterrizar, como el rescate marítimo.

1.3. Aplicaciones

La robótica aérea ha experimentado un crecimiento exponencial en los últimos años, se ha popularizado su uso y se ha extendido la comercialización de drones.

El sector donde más rápida acogida ha tenido la robótica aérea ha sido el sector audiovisual, se usa de forma habitual en cine, grabación de espectáculos en directo televisión y fotografía. El porqué de tal acogida se basa principalmente en dos factores, los costes y la viabilidad técnica. Los costes de realizar una toma aérea en una producción antes pasaban por el alquiler de un helicóptero, dependiendo de la toma, así como del material y la contratación de los medios humanos a bordo para realizarlas podía ascender a entre 4000 y 6000 euros la hora. Hoy en día basta con un pequeño drone de entre 400 y 1800 euros¹¹ por jornada, e incluirían en el tramo más elevado piloto y operador de cámara. Como se puede constatar fácilmente, el ahorro no es desdeñable y ofrece a pequeñas productoras acceder a este tipo de grabaciones que de otro modo serían privativas. Aunque el aspecto económico es especialmente importante para decantarse por el uso de este tipo de medios, existen en el mundo audiovisual trabajos que no hubiesen sido posibles sin los drones. Grabaciones en las que el riesgo humano y material que habría que asumir es tan alto que tan sólo son factibles de este modo. Debemos por tanto a los drones fotografías como la de la Figura 1.2 que si bien no se tomó como parte de ningún proyecto cinematográfico nos hace una idea de lo que la robótica aérea puede ofrecernos.

⁹Vertical take off and landing

¹⁰Mantenerse estáticamente en un punto elevado

¹¹Precios aproximados extraídos de la empresa especializada World Aviation Helicopters <https://www.worldaviation.es/es/servicio-drones.aspx>



Figura 1.2: Drone grabando en el interior de un cráter

Otro sector que ha adaptado rápidamente el uso de éstas pequeñas aeronaves es el agropecuario, donde se utiliza para medir las condiciones del terreno, con el fin de recoger información sobre la hidratación, la temperatura o el ritmo de crecimiento de los cultivos. Con el punto de vista e imágenes de los drones el agricultor puede tener en tiempo real una foto del estado de su viñedo localizando con facilidad zonas afectadas por plagas o incluso vigilar el cultivo para evitar la entrada de intrusos. Controlan el riego e incluso esparcen los pesticidas de manera eficiente siendo un arma eficaz contra las plagas, se utilizan incluso como espantapájaros. La aplicación de drones en este sector se remonta a 1983 cuando el Ministerio de Agricultura de Japón preocupado por el envejecimiento de su población rural encargó a Yamaha el desarrollo de una aeronave no tripulada capaz de realizar varias tareas de las anteriormente descritas, a fin de atraer más gente al medio rural. En 1990 se entregaron las primeras unidades del Yamaha RMAX y actualmente el 40 % de los arrozales japoneses cuentan con un drone sobrevolándolos. Como método de riego, el drone tiene dos contenedores que pueden albergar una cantidad de 8 litros cada uno y en caso de utilizarse para esparcimiento de semillas o material granulado, posee otros dos contenedores que se pueden adaptar los cuales tienen una capacidad de 13 litros cada uno. Puede elevarse a una altura de 400 metros y dispone de un sistema de GPS que no sólo le permite mayor estabilidad durante el vuelo, sino que también le proporciona al usuario

la posibilidad de programar la ruta desde antes de ser lanzado. Tiene una autonomía de una hora durante el vuelo.



Figura 1.3: Drone Yamaha RMAX fumigando

Los drones están empezando a popularizarse en las empresas de seguridad privada. El punto de vista y alcance que proporcionan hacen de los drones herramientas muy potentes y versátiles. Se utilizan principalmente en vigilancia perimetral, dónde un drone puede recorrer una ruta de forma autónoma proporcionando imágenes de la misma a los vigilantes del centro de cámaras o a los *smartphones* o *tablets* de los guardias de seguridad. Se les utiliza también como apoyo a la vigilancia tradicional, sobrevolando con ellos zonas de grandes aglomeraciones o apoyando desde las alturas a las patrullas de vigilancia proporcionando, por ejemplo, imágenes térmicas de una zona durante la noche. Empresas como la francesa Drone Volt, líder en el sector de los drones profesionales, ha realizado un avance en materia de seguridad con el modelo *Z18 UF (Unlimited Flight)* un drone umbilical capaz monitorizar 24 horas de forma ininterrumpida. Esto es posible gracias a que va unido con un cable a una fuente de alimentación en tierra y por tanto se reduce la principal limitación de los drones, el tiempo de vuelo que proporcionan baterías pequeñas y ligeras.



Figura 1.4: Drone Volt Z18 UF

En España varias empresas de vigilancia privada ofrecen sus servicios con drones, algunas de ellas son:

- Aero Cámaras - <http://aerocamaras.es/servicios-drones-profesionales/drones-vigilancia>
- Prosegur - <https://www.prosegur.es/newsdetails/drone-vigilancia-interiores>
- SkyDron - <https://www.skydron.es/seguridad-aerea-privada-drones/>

Son utilizados también por los servicios de rescate tras una catástrofe para evaluar los daños experimentados y ayudar a encontrar supervivientes entre los escombros. Algunos son capaces de enviar a los supervivientes paquetes de supervivencia con salvavidas, alimentos o agua mientras esperan su rescate¹²

Es muy extendido en el mantenimiento de infraestructuras, en trabajos como:

- La inspección de tendidos eléctricos. Donde antes se utilizaban helicópteros ahora con los drones se obtienen mejores imágenes e inspecciones más precisas y meticulosas sin arriesgar vidas humanas y por una mínima parte del coste que se asumía.

¹²<http://www.elmundo.es/economia/2015/09/15/55f8239546163fc6598b45c3.html>

- Estos mismos drones también se utilizan para la revisión de las palas de los aerogeneradores, evitando que el personal de mantenimiento tenga que descolgarse desde ellos para su análisis.
- Se utilizan también drones para revisar oleoductos, especialmente en zonas de difícil acceso.
- Conservación de carreteras, detectando el estado de conservación de las calzadas, obstáculos o potenciales situaciones de peligro debido a accidentes o condiciones meteorológicas adversas.
- Supervisión y mantenimiento de presas y embalses.

La empresa española Ferrovial fue la pionera en España en obtener la licencia para este tipo de trabajos¹³

Es muy importante su uso en topografía, principalmente para obtener topografía aérea mediante técnicas de fotogrametría¹⁴. La empresa española OHL dispone de varios drones a tal efecto y tiene varios proyectos de I+D+I para seguir profundizando en esta materia. De esta manera se pueden estudiar obras en su fase de licitación, realizar cálculos de volúmenes y superficies en acopios, control de certificaciones, estudio de patologías como deslizamiento de taludes y realizar seguimientos.

Se utilizan en minas a cielo abierto para entre otras muchas funciones:

- Control y monitorización de explotación de minerales y su impacto ambiental
- Observación de operaciones con necesidad de supervisión aérea y seguimiento de movimientos de tierra, residuos, balsas, control de relaves, control de pilas, transporte de implementos de un punto a otro.
- Realizar la reconstrucción de una mina mediante fotogrametría para medir los volúmenes extraídos.
- Analizar la variación de alturas entre periodos para determinar los volúmenes explotados por periodo.

¹³<http://www.ferrovial.com/es/prensa/noticias/ferrovial-servicios-obtiene-la-licencia-para-operar-drones-en-espana/>

¹⁴Técnica para obtener mapas y planos de grandes extensiones de terreno por medio de la fotografía aérea.

E incluso se han empezado a utilizar por el Ministerio de Hacienda de España con fines recaudatorios. Sobrevuelan las viviendas o fincas localizando edificaciones no registradas por el catastro con el fin de regularizar su situación.

El uso de drones es especialmente importante en el ámbito científico para, por ejemplo, tomar medidas de temperatura y CO₂ en zonas peligrosas¹⁵, estudiar las nubes volcánicas¹⁶ o volar dentro de una grieta en un glaciar¹⁷.

El futuro del uso de los drones se está escribiendo en este mismo momento y es que poco a poco se investigan con nuevos usos o cómo mejorar los ya actuales. Uno de los usos más prometedores que está siendo investigado es el de logística y paquetería. Amazon estudió en 2015 la viabilidad de utilizar drones para el reparto, especialmente en zonas de difícil acceso o bien alejadas de las zonas habituales de reparto. A finales de dicho año realizó pruebas de entregas en el Reino Unido y desarrolló un prototipo de alrededor de 25Kg de peso. El objetivo de Amazon en éste experimento era realizar entregas de hasta 3Kg de peso en menos de 30 minutos y ver la viabilidad de la entrega de paquetes en zonas pobladas.

Esto ha dado pie a las principales empresas de logística y paquetería a realizar sus propios desarrollos y construir sus propios prototipos. DHL ha realizado sendas pruebas con drones de cerca de 5 Kg para la entrega de paquetes de hasta 1,2 Kg. La empresa francesa GeoPost ha comprado recientemente la empresa Atechsyst, especializada en el desarrollo de sistemas autónomos para aeronaves no tripuladas. Operación que ha dado como resultado un drone que cuenta con seis rotores eléctricos y estructura de fibra de carbono, con capacidad para llevar paquetes de un peso aproximado de hasta 2 kilos. UPS ha realizado un experimento en Tampa con un octocóptero que despegaría desde el techo de la furgoneta de reparto para entregar los paquetes en zonas rurales y ahorrar en kilometraje el drone tendría una carga de pago de unos 4,5Kg.

Otro de los estudios que más llama la atención es el de Google y Facebook que compiten en llevar internet a zonas aisladas a través de una red de drones y satélites. Facebook presentó en su F8 en Marzo de 2015 su prototipo de drone a tal efecto, Aquila, fruto de la adquisición de la empresa especializada en robótica aérea Acenta. Este proyecto se enmarca dentro del plan

¹⁵<http://www.igepn.edu.ec/servicios/noticias/1395-medidas-de-temperatura-y-co2-de-las-fumarolas-muestreo-y-fotografias-con-drone>

¹⁶<https://www.nasa.gov/topics/earth/earthmonth/volcanic-plume-uavs.html>

¹⁷http://tn.com.ar/tecnorecomendados/increible-un-drone-volo-dentro-de-un-glaciar_648661

internet.org y está liderado por Connectivity Lab¹⁸ que pretende ofrecer internet con un coste reducido a todo aquel que no lo tenga.

En 2014 Google por su parte compró Titan Aerospace para crear una flota de drones propulsados por energía solar, capaces de volar más de una semana mientras tomaban fotos de la superficie y proveían de acceso a Internet a lugares remotos y aportar nueva información para sus mapas. Este desarrollo sin embargo se ha abandonado en enero de este año en pro de la utilización de globos aerostáticos para tal propósito.

1.4. Robótica aérea en el laboratorio de Robótica de la URJC

Dentro del laboratorio de robótica de la Universidad Rey Juan Carlos cabe destacar varios trabajos realizados para profundizar, investigar y experimentar con drones y que han servido de antecedentes directos y base para este Proyecto Fin de Carrera. Todos ellos han usado como plataforma software JdeRobot <http://jderobot.org/>. Se resumen a continuación los más importantes.

Trabajos como el de Alberto Martín[4] que permitió controlar un AR-Drone 2 de Parrot real desde una aplicación cliente, desarrollando para ello un driver para el dron, `ardrone_server` y una aplicación cliente UAV Viewer. El driver `ardrone_server` es capaz de conectar con el AR-DRONE a través de comandos AT tanto para obtener los datos de todos sus sensores como para controlarlo y expone todo ello en interfaces JdeRobot para su interconexión con el software del entorno.

El AR-Drone es un quadricóptero que en su versión actual cuenta con acelerómetros, giróscopos y magnetómetros en 3 ejes que determinan junto con su barómetro y un sensor de ultrasonidos la actitud del mismo, dispone además de 2 cámaras, una ventral y otra frontal, y algunos modelos traen sensor GPS.

¹⁸Un equipo formado por 50 expertos en aeronáutica y ciencia espacial



Figura 1.5: AR Drone 2

Alberto desarrolló también el software de control UAV Viewer. Esta aplicación cliente es capaz de controlar con cualquier drone y mostrar de forma visual su actitud, lo que captan sus cámaras.

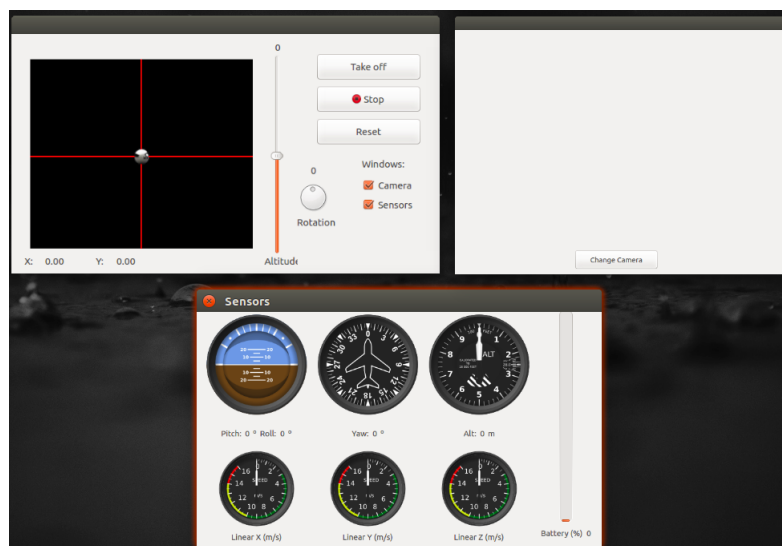


Figura 1.6: Ventanas de UAV Viewer

Daniel Yague[10] desarrolló un driver (plugin) para emular el comportamiento del AD-Drone en el simulador Gazebo, de referencia en el Laboratorio de Robótica de la universidad y contenido dentro de la suite JdeRobot. Con este driver se hizo posible probar, anticipar los problemas que puedan surgir en el vuelo del drone antes siquiera de volarlo físicamente. De esta forma es posible el desarrollo de aplicaciones de navegación complejas sin disponer de él, sin arriesgarlo e independientemente de factores externos como la climatología. Para mostrar las posibilidades que se abrían desarrolló varias aplicaciones en el ar-drone simulado como un

gato-ratón donde un drone trataba de perseguir autónomamente a otro que se teleoperaba o una aplicación donde el drone seguía una carretera.

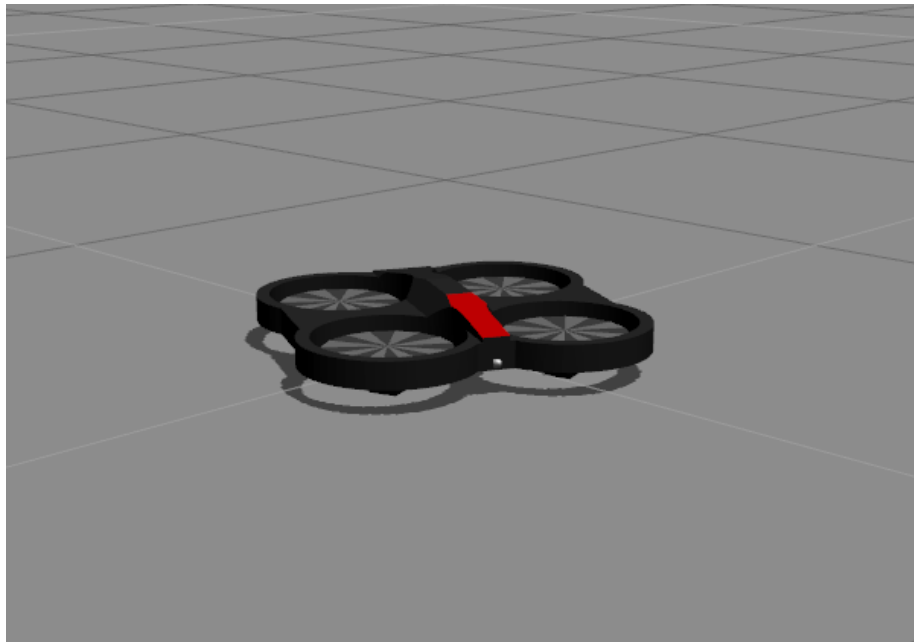


Figura 1.7: AR Drone Simulado en Gazebo

Jorge Cano construyó físicamente su propio drone utilizando como base un quadricóptero con un procesador Intel Compute Stick (ICS) STCK1A8LFC como ordenador de abordo y una placa Pixhawk como estabilizadora/piloto automático[2]. Esta placa utiliza como protocolo de comunicación MAVLink¹⁹ y dispone de acelerómetros, giróscopos y magnetómetros para determinar la actitud. Además de la construcción del drone Jorge, desarrolló el driver MAVLink-Server donde apoyándose en MAVProxy adaptó los comandos MAVLink a interfaces JdeRobot permitiendo acceder a la actitud y controlar el drone con comandos tipo GotoXY enviados a través del interfaz Pose3D²⁰

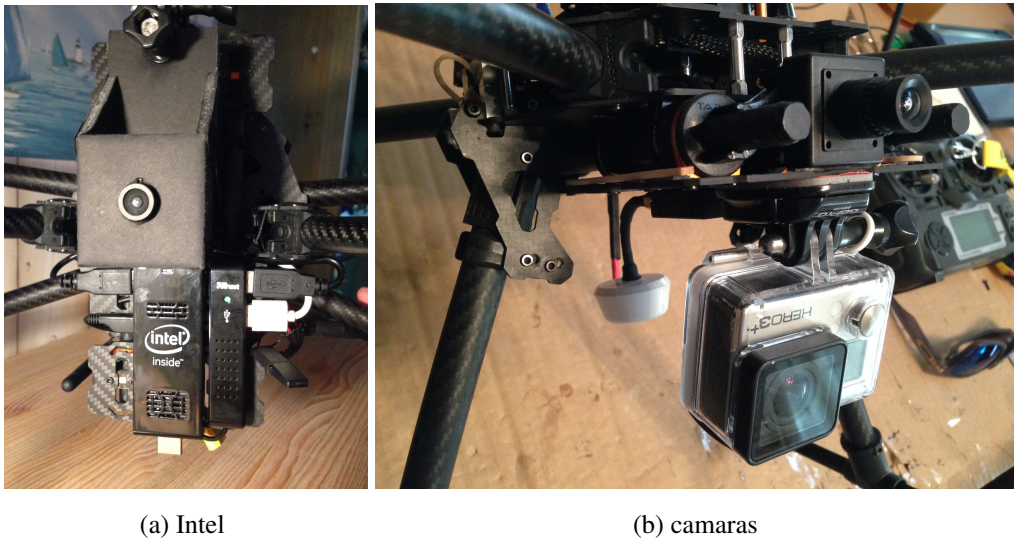
En la actualidad se están desarrollando varios trabajos sobre este tipo de placas estabilizadoras que tiene como software base ArduCopter/Ardupilot o son compatibles. Por un lado Diego Jiménez[5] trabaja en controlar un Solo Drone de la empresa 3DR²¹ mediante el interfaz de velocidades CMDVel²². Este quadracótero tiene como placa de control una Pixhawk como la

¹⁹Micro Air Vehicle Link <https://en.wikipedia.org/wiki/MAVLink>

²⁰Ver capítulo 3 Arquitectura utilizada

²¹Empresa norteamericana con sede en California especializada en robótica aérea. Se sitúa en 2017 como la 3ª empresa del sector

²²Ver capítulo 3 Arquitectura utilizada



utilizada por Jorge Cano en el drone que se construyó y utiliza también comandos MAVLink.

Por otro lado, Jorge Vela se encuentra desarrollando cómo realizar la maniobra de aterrizaje de forma automática al localizar un patrón o baliza mediante visión[9]. Como drone de referencia está utilizando un Solo Drone con un procesador Intel Stick como ordenador de a bordo.



Figura 1.8: SOLO Drone de 3D Robotics

Crear clientes web para las principales aplicaciones JdeRobot, incluyendo una con drones, fue tarea de Aitor Martínez[6]. En desarrolló seis clientes entre los que destaca la creación del cliente web similar a la herramienta UavViewer para teleoperar drones tanto reales como simu-

lados y ver los datos de sus sensores. También la creación de `IntrorobUavJS` que permite programar comportamientos autónomos en JavaScript una en una interfaz Web similar a UAV Viewer. Gracias a estos desarrollos, se podía teleoperar un drone desde cualquier dispositivo que tenga un navegador web, como un móvil o una tablet.



Figura 1.9: Drone teleoperado con teléfono móvil

En este contexto este PFC aborda el soporte de un nuevo robot aéreo, diferente a todos los usados en el Laboratorio de Robótica hasta la fecha: un avión de ala fija. Además se integrará con las herramientas ya existentes para robots aéreos dentro de la plataforma JdeRobot.

La memoria de este PFC se ha vertebrado en siete capítulos. En el segundo se fijan los objetivos concretos que se persiguen y los requisitos deseables. A continuación se describe la infraestructura, tanto hardware como software, utilizada en el desarrollo de este trabajo. En el capítulo 4 se detalla el desarrollo del driver `APM Server` y se desgrena la aplicación `UAV Commander` en el capítulo 5. En el sexto se muestran los experimentos realizados con todo el software desarrollado y en el séptimo las conclusiones alcanzadas.

Capítulo 2

Objetivos

2.1. Problema abordado

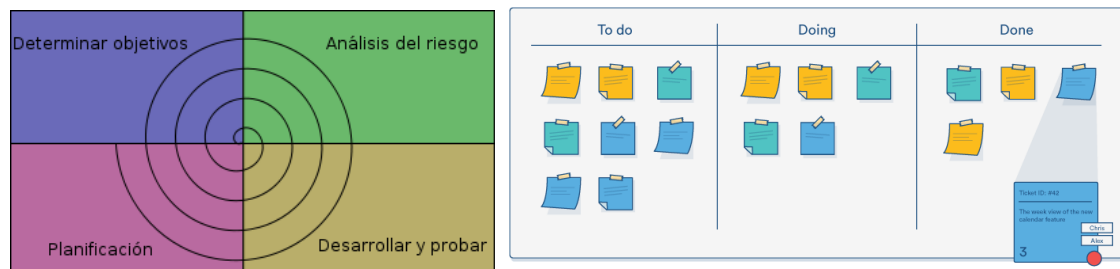
El objetivo general de este proyecto final de carrera es dar soporte en la plataforma software JdeRobot a drones de ala fija que utilicen el protocolo de comunicación MAVLink. Este objetivo general lo hemos articulado en 3 subobjetivos:

1. Desarrollo de un driver que acceda a los sensores y actuadores de drones de ala fija que utilicen MAVLink, y dar soporte a la actuación de misiones, que hasta ahora no soportaba JdeRobot.
2. Desarrollo de una aplicación GCS *Ground Control Station* que permita al operador humano introducir misiones y seguir el cumplimiento de las mismas a través de ella. Permitiendo el acceso a la actitud y a las cámaras de abordo.
3. Hacer experimentos con aviones MAVLink tanto simulados como con un prototipo real. Conectaremos nuestro driver al simulador SITL y la aplicación al servidor para verificar la correcta especificación y materialización de las misiones. Construiremos un prototipo de avión MAVLink y comprobaremos la conexión con el driver y la recepción de toda la información sensorial.

Tanto el driver como la aplicación, han de ser multiplataforma, utilizar únicamente librerías de software libre y ser 100 % compatibles con los actuales interfaces JdeRobot

2.2. Metodología

Este proyecto se ha abordado con dos metodologías de trabajo distintas combinadas, la primera es el desarrollo en espiral que se ha utilizado como metodología principal, y la segunda la metodología ágil Kanban, para un control de tareas y subtareas más eficiente.



(a) Representación gráfica del desarrollo en espiral.

(b) Representación de una pizarra kanban

Figura 2.1:

Metodologías empleadas

El modelo de desarrollo en espiral define una serie de ciclos que se repiten en bucle hasta el final del proyecto, dividiéndolo en varias subtareas más sencillas y estableciendo puntos de control al final de cada iteración, en los que se evalúa el trabajo realizado y se enfocan las nuevas tareas para continuar. Esta metodología recibe su nombre por la forma de espiral que tiene su representación gráfica o diagrama de flujo, que podemos ver en la figura 2.1. En cada iteración se llevan a cabo las siguientes actividades:

1. Determinar los objetivos, dividir en subobjetivos y fijar requisitos.
2. Analizar los riesgos y factores que impidan o dificulten el trabajo y las consecuencias negativas que éste pueda ocasionar.
3. Desarrollar los objetivos cubriendo los requisitos analizados anteriormente.
4. Se analizan los resultados del desarrollo y las pruebas y en base a esto se decide el inicio del siguiente ciclo en una reunión con el tutor.

Durante el ciclo de vida del proyecto se han llevado a cabo reuniones semanales de seguimiento con el tutor. En ellas se evaluaban las tareas realizadas y se marcaba qué dirección tomar

para la siguiente iteración o incremento. Si los puntos marcados en la anterior reunión no se habían alcanzado se ampliaba el plazo o se discutían otras vías para avanzar. En caso contrario se proponían nuevos subobjetivos.

Para desarrollar las tareas especificadas en cada incremento, fijadas en cada reunión de seguimiento, se utilizó la metodología kanban. Esta metodología de desarrollo no es más que una adaptación de su versión industrial que surgió en Toyota. A finales de los años 40, Toyota empezó a optimizar sus procesos de ingeniería a partir del modelo que empleaban los supermercados para llenar los estantes. Los supermercados almacenan los productos suficientes para suplir la demanda del cliente, una práctica que optimiza el flujo entre el supermercado y el cliente. En su versión TIC ésta metodología se centra, al igual que su versión industrial, en el *just in time*¹ y permite apreciar de una forma muy visual las tareas que hay en vuelo, desarrolladas, pendientes o bloqueadas pudiendo anticiparnos a cuellos de botella o bloqueos de forma sencilla.



Figura 2.2: Evolución de nuestra pizarra Kanban

¹El JIT es una política de mantenimiento de inventarios al mínimo nivel posible donde los suministradores entregan justo lo necesario en el momento necesario para completar el proceso productivo.

Para apoyarnos en nuestro desarrollo hemos utilizado cuatro herramientas:

- GitHub como forja y control de versiones. En el repositorio <https://github.com/RoboticsURJC-students/2014-pfc-JoseAntonio-Fernandez> se almacenan todos los desarrollos de este PFC. También se encuentran subproductos de desarrollo que han ido surgiendo como apoyo o pruebas a los desarrollos principales.
- Bitácora web en JdeRobot donde hemos actualizado periódicamente nuestros avances acompañados con explicaciones, vídeos e imágenes. Su url es <http://jderobot.org/Jafernandez> y aquí se puede contemplar con más detalle la construcción del UAV.
- YouTube y ftp. como herramientas de almacenamiento de los vídeos de nuestra bitácora web y de nuestros experimentos.

Capítulo 3

Infraestructura utilizada

Se describen en este capítulo las herramientas, ingredientes hardware y software en las que se apoya este PFC.

3.1. Hardware

Este PFC utiliza principalmente tres componentes hardware: un avión de radio control, un procesador Raspberry PI 3 y una placa estabilizadora.

Como el avión de radio-control hemos elegido el avión Bix3, distribuido por la empresa china www.hobbyking.com. Hemos elegido este modelo por ser un avión muy estable debido principalmente a sus cualidades como velero y su alta superficie alar. Otro aspecto importante de la construcción del avión es que el propulsor no se encuentra en el frontal del avión lo que nos permitirá aprovechar al máximo esa zona pudiendo poner incluso una cámara frontal.



(a) Bix3



(b) Vista de cerca



(c) Bix3 en vuelo

Para poder cargar con el equipo necesario a bordo ha sido necesario cambiar el motor, el variador y las baterías de serie por otras de mayor rendimiento que nos permitan volar con tanta

carga de pago.

Una raspberry PI3 es el ordenador de abordo y en el que se ha instalado el software necesario para dotarle de inteligencia. Hemos elegido este dispositivo debido al compromiso peso/potencia que otorga, así como porque es un hardware muy asequible y extendido. Conectado a esta PI3 va una PiCam que nos permitirá ver lo que el avión vea.

Para este PFC hemos utilizado una placa Ardupilot Mega¹ como estabilizador/piloto automático. Este dispositivo tiene como base una placa Arduino Mega a la que se le han incorporado giróscopos y acelerómetros en 3 ejes para su estabilización y que trae de serie un receptor GPS con brújula. El kit que adquirimos incluye también todo lo necesario para integrarlo en nuestro avión y proporcionarle la alimentación necesaria. Esta placa actúa directamente sobre los servos y el motor del avión a través de señales PWM.

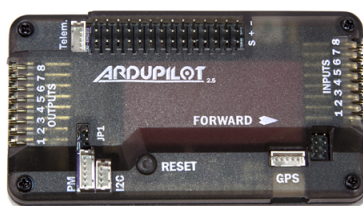


Figura 3.1: Placa estabilizadora ardupilot

Esta placa ha de *flashearse* con un *firmware* en función del tipo de aeronave que utilicemos como base, Arducopter para alas rotatorias o Ardupilot para alas fijas. Ésta y otras placas similares como PixHawk, se comunican y aceptan comandos a través de un protocolo llamado MAVLink. A través de comandos MAVLink se puede acceder a los sensores, tanto los que incorpora la placa nativa cómo a los que le añadamos a la misma, como el GPS o sensores de velocidad del aire. A través de estos comandos se le puede también enviar órdenes al piloto automático, quien las ejecutará. Más adelante trataremos el protocolo MAVLink en profundidad. Cabe destacar que algunos de los comandos que mandemos tendrán distinto comportamiento en la placa en función del *firmware* escogido, y que hay un conjunto de comandos que no pueden ser utilizados en uno o en otro. El objetivo de este PFC es crear un driver que nos permita comunicarnos con esta placa y una aplicación en JdeRobot para el manejo de un avión robotizado.

¹<http://www.ardupilot.co.uk/>

3.2. JdeRobot

JdeRobot es un entorno de software creado por el laboratorio de robótica de la Universidad Rey Juan Carlos, para el desarrollo de aplicaciones de robótica. Su última versión, la 5.5, se liberó el 15 de Marzo de 2017 pudiendo ver los detalles de ésta en el github oficial². JdeRobot se compone de interfaces, drivers, utilidades y aplicaciones para el desarrollo de cualquier proyecto de robótica. Gracias a estos interfaces y al *middleware* ICE (de la empresa ZeroC) se interconectan entre sí todos los componentes de una aplicación robótica. Algunos de los drivers más importantes que contiene, y más relacionados con este trabajo, son:

1. `Cameraserver`. Un driver para enviar imágenes y video a través del interfaz `camera`
2. `Gazeboserver`. Conjunto de *plugins* de Gazebo para conectar las aplicaciones con distintos robots simulados (con ruedas, cuadricópteros, etc).
3. `MAVLinkServer`. Driver desarrollado para intercomunicar JdeRobot con placas de estabilización y autopiloto que utilicen el protocolo de comunicación MAVLink.
4. `Ardrone_server`. Driver que conecta el Parrot Ar-Drone a JdeRobot. Este driver escrito en c++ transforma el conjunto de comandos AT del drone en interfaces ICE y viceversa. Implementa los interfaces `camera`, `cmdvel`, `navdata`, `extra` y `pose3D` y permite acceder a la actitud del drone así como a sus 2 cámaras. Sirve también datos como el nivel de la batería y permite grabar vídeo o tomar fotos.

Algunas de las herramientas contenidas en JdeRobot y más relacionadas con este PFC son:

1. `Camerview`. Es una aplicación desarrollada en c++ capaz de recibir vídeo a través del interfaz `camera`.
2. `UAV viewer`. Aplicación desarrollada para teleoperar robots aéreos. Esta aplicación permite teleoperar cualquier tipo de robot aéreo y ofrece de forma visualmente atractiva datos como la actitud, velocidades lineales y angulares, ofrece también la posibilidad de visualizar imágenes servidas por el interfaz `camera`.

²<https://github.com/JdeRobot/JdeRobot/wiki/JdeRobot-5.5.0>

3.2.1. Interfaces

JdeRobot expone más de 30 interfaces pero en este capítulo explicaremos los que durante nuestro desarrollo hemos implementado en el driver para el avión o usado en la aplicación:

- **Pose3D.** Utilizado para recoger los datos de actitud y la posición 3D de la aeronave. Devuelve la posición del robot y su orientación en el espacio, empleando para expresar la orientación cuaterniones. Comparados con los ángulos de Euler, son más simples de componer y evitan el problema del bloqueo del cardán³, `pymavlink` nos permite transformar de uno a otro de forma sencilla.

```
Pose3DData
{
    float x; //latitude
    float y; //longitude
    float z; //altitude
    float h; //not used now
    float q0; //quaternion component 1
    float q1; //quaternion component 2
    float q2; //quaternion component 3
    float q3; //quaternion component 4
};
```

- **Camera.** Utilizado para servir imágenes.

```
class CameraDescription
{
    string name;
    string shortDescription;
    string streamingUri;
    float fdistx; //focal distance in x
    float fdisty; //focal distance in y
    float u0;
    float v0;
    float skew;
    float posx; //position in x of the camera
    float posy; //position in y of the camera
    float posz; //position in z of the camera
    float foax; //position of a focused object
    float foay; //position of a focused object
    float foaz; //position of a focused object
    float roll; //roll of the camera
};
```

³El bloqueo del cardán consiste en la pérdida de un grado de libertad en una suspensión cardán de tres rotores, que ocurre cuando los ejes de dos de los tres rotores se colocan en paralelo, bloqueando el sistema en una rotación en un espacio bidimensional degenerado.

Los parámetros del interfaz se pueden dividir en 2 bloques: intrínsecos y extrínsecos.

- Intrínsecos: fdistx, fdisty, u0, v0 y skew.
 - Extrínsecos: posx, posy, posz, foax, foay, foaz y roll.
- NavData. Utilizado para servir datos secundarios de actuación como velocidades lineales o angulares o el estado de la batería.

```
class NavdataData
{
    int vehicle; //0-> ArDrone1, 1-> ArDrone2
    int state; // landed, flying,...
    float batteryPercent; //The remaing charge of baterry %

    //Magnetometer Ardron2.0
    int magX;
    int magY;
    int magZ;

    int pressure; //Barometer Ardron2.0
    int temp;      //Temperature sensor Ardron2.0
    float windSpeed; //Estimated wind speed Ardron2.0

    float windAngle;
    float windCompAngle;

    float rotX; //rotation about the X axis
    float rotY; //rotation about the Y axis
    float rotZ; //rotation about the Z axis

    int altd; //Estimated altitude (mm)

    //linear velocities (mm/sec)
    float vx;
    float vy;
    float vz;

    //linear accelerations (unit: g) ¿Ardron2.0?
    float ax;
    float ay;
    float az;

    //Tags in Vision Detectoion
    //Should be unsigned
    int tagsCount;
    arrayInt tagsType;
    arrayInt tagsXc;
```

```

arrayInt tagsYc;
arrayInt tagsWidth;
arrayInt tagsHeight;
arrayFloat tagsOrientation;
arrayFloat tagsDistance;

float tm; //time stamp
};

```

- Extra. Utilizado principalmente para las órdenes de despegue y aterrizaje.

```

void land() - land drone.
void takeoff() - takeoff drone.
void reset()
void recordOnUsb(bool record)
void ledAnimation(int type,float duration, float req)
void flightAnimation(int type, float duration)
void flatTrim()
void toggleCam() - switch camera.

```

3.3. Protocolo MAVLink

MAVLink son siglas de Micro Air Vehicle Link, un protocolo de comunicación desarrollado para comunicar las placas estabilizadoras dotadas de piloto automático con los GCS o *Ground control station*, es decir, con las aplicaciones desde las que se podía enviar misiones y seguir el cumplimiento de las mismas desde tierra.

MAVLink se publicó en 2009 por Lorenz Meier, publicado bajo licencia LGPL. Aspira a convertirse en el protocolo standard en robótica aérea y se ha probado su funcionamiento en PX4, PIXHAWK, APM⁴ y Parrot AR.Drone.

Un ejemplo de comando MAVLink sería este mensaje que porta la información del GPS y se envía periódicamente en ciclos que decidimos según la configuración de conexión con el dispositivo.

```

type GpsStatus struct {
    SatellitesVisible uint8      Número de satélites visibles
    SatellitePrn       [20]uint8 Id Global de cada satélite
    SatelliteUsed       [20]uint8 Lista con el uso de cada sat\elite
    SatelliteElevation [20]uint8 Elevación, nos da el ángulo sobre el horizonte.
    SatelliteAzimuth   [20]uint8 Dirección del satélite, 0: 0 grados, 255: 360 grados.
}

```

⁴Ardupilot Mega

```

SatelliteSnr      [20]uint8  Señal/ruido de cada uno de los satélites
}

```

Otro mensaje, esta vez vinculado a la actuación es:

```

type MissionItem struct {
    Param1      float32  parámetro variable en función del comando.
    Param2      float32  parámetro variable en función del comando.
    Param3      float32  parámetro variable en función del comando.
    Param4      float32  parámetro variable en función del comando.
    X           float32  latitud
    Y           float32  longitud
    Z           float32  altitud
    Seq         uint16   Número del item en la misión
    Command     uint16   Tipo de comando de navegación.
    TargetSystem uint8    ID del sistema
    TargetComponent uint8
    Frame       uint8    Sistema de coordenadas que se utiliza.
    Current     uint8    Misión actual no:0, si:1
    Autocontinue uint8   Autocontinuar al siguiente objeto de misión.
}

```

Este mensaje representa un hito de misión, una misión se compone de uno o varios de estos hitos. Un hito de misión en MAVLink puede ser un punto de paso (o *waypoint*), una maniobra de despegue o aterrizaje, o instrucciones de cambio de altura o velocidad del robot.

3.4. Biblioteca pymavlink

Pymavlink trae la definición de los comandos MAVLink a utilizar desde un programa escrito en python. Gracias a esta biblioteca no se tienen que construir los comandos MAVLink a mano, evitando cometer errores y simplificando la creación y envío de los mismos. Su uso facilita también la identificación de los comandos recibidos. Para instalarla tan sólo se necesita ejecutar:

```
sudo pip2 install -U pymavlink
```

Y tiene como dependencias:

- future. Se necesita future como soporte en Python 3.X de Python 2.7 (<http://python-future.org/>)
- lxml. Para analizar xml (<http://lxml.de/installation.html>)

- python-dev
- MavLink. (<http://qgroundcontrol.org/mavlink/start>)

3.5. Lenguaje Python y biblioteca PyQt5

Python es un lenguaje de programación interpretado y multiplataforma que nació en los años 80 en los países bajos con idea de hacer más legible el código. El lenguaje de programación que inicialmente se utilizaba principalmente para scripting, ha sabido crecer con los años y con la publicación de Python3 en 2009 ha recibido el impulso que necesitaba para ser hoy en día el quinto lenguaje más utilizado, por encima de PHP, .NET y JavaScript (que baja hasta el 8º puesto según TIOBE⁵ en un estudio de Abril de 2017).

Para este PFC se eligió Python, porque mantiene el carácter multiplataforma de JdeRobot, su código es simple y legible y trabaja muy bien con dependencias muy utilizadas en robótica como OpenCV.

Para nuestro desarrollo hemos utilizado PyQt5 para desarrollar el interfaz gráfico. PyQt5 es un *binding* de Qt5 en forma de librería Python que nos permite acceder a toda la funcionalidad de Qt5. Qt, propiedad de Nokia, es un conjunto de librerías escritas en C++ para interfaces gráficas.

3.6. Mapas y Geo-referenciación

En la aplicación de navegación autónoma hemos tenido que trabajar con mapas geo-referenciados. Estos mapas, que tanto se han popularizado gracias a Google, son una herramienta imprescindible en robótica aérea si se quiere trabajar con largas distancias. Un mapa geo-referenciado es aquel en el que conocemos o podemos calcular la posición en el planeta que representa cada píxel del mismo.

La forma más común de obtener estos mapas geo-referenciados es a través de WMS, siglas de *Web Map Service*. Un WMS nos es más que un servicio web que recibe como entrada unas coordenadas y una serie de parámetros y devuelve una imagen encuadrada en los datos enviados.

⁵<https://www.tiobe.com/tiobe-index/>



Figura 3.2: Imágen geo-referenciada del club de aeromodelismo Icaro

En nuestro desarrollo hemos utilizado dos WMS: el del Instituto Geográfico Nacional de España (IGN) y el de Google. A continuación vamos a desgranar un WMS, el de PNOA del IGN. PNOA son las siglas de Plan Nacional de Ortofotografía Aérea y contiene los mapas más actuales del territorio español. El WMS de PNOA requiere como entrada cuatro atributos principales:

1. La posición GPS.
2. *Bounding box*. El *bounding box* son dos puntos que corresponden con la posición GPS que queremos que sea el extremo inferior izquierdo de nuestra imagen y con el punto superior derecho de la misma.
3. Datum. El datum es el sistema de referencia o proyección de la tierra a utilizar. En el caso de IGN pese a que soporta varias utilizamos WGS84 que es el estándar.
4. Tamaño de la imagen, el tamaño que queremos que tenga la imagen obtenida.

El resultado de la consulta es un mapa del que conocemos el punto central, y los extremos inferior izquierdo y superior derecho y el número de píxeles que tiene a lo ancho y alto del mismo, es decir, un mapa geo-referenciado.

Con el fin de facilitar el montaje y envío de las peticiones al WMS de PNOA utilizamos la librería OWSLib⁶ en su versión 0.14.0.

3.7. Simulador SITL

Software In The Loop (o SITL⁷) es un simulador que permite a cualquier programa que envíe o reciba comandos MAVLink ejecutar pruebas sin necesidad de tener ninguna placa estabilizadora real y evitando la pérdida de la aeronave en caso de error del programa. SITL se conecta con JSBSim, un simulador de vuelo de software libre para ejecutar el aspecto físico de la simulación y con MAVProxy para el envío de comandos y seguimiento de misiones. Desde MAVProxy podemos enviar directamente comandos de vuelo al simulador sin necesidad de ninguna herramienta externa, y ha sido muy útil a la hora de articular nuestras pruebas. Se trata de una compilación en C++ de ardupilot y se podría asemejar al driver para cuadricópteros existente en JdeRobot gazebo_server, pero para aviones de ala fija. SITL puede conectarse también con Gazebo o con FlightGear para hacer más completa su simulación mostrando el vuelo en un entorno desarrollado para pruebas con elementos del mundo real.

⁶<https://geopython.github.io/OWSLib/>

⁷<http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html>

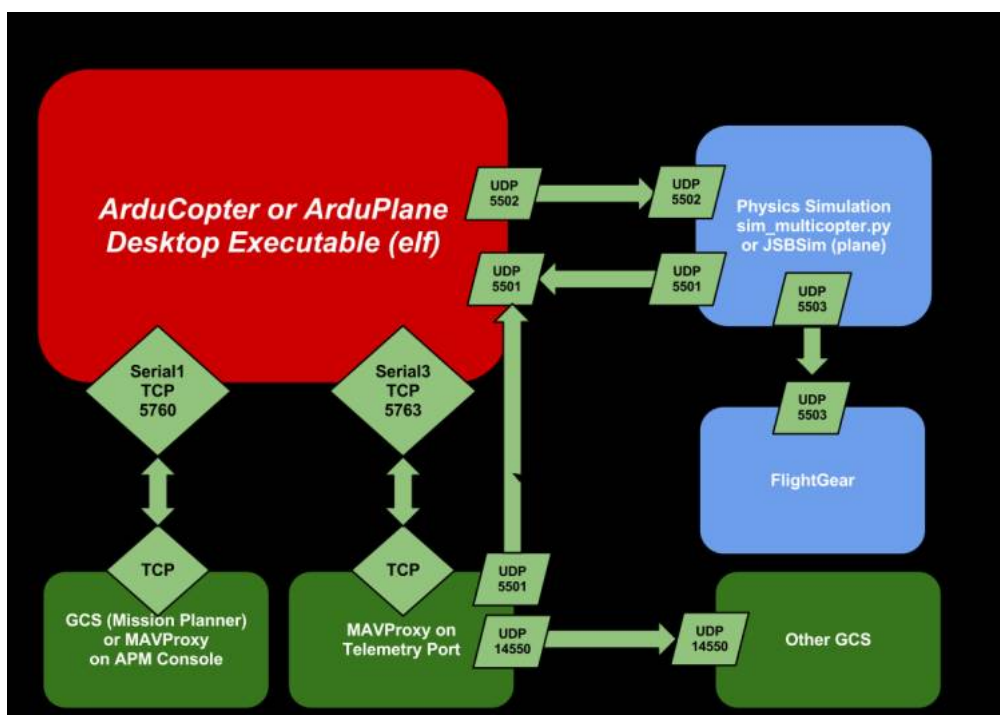


Figura 3.3: Arquitectura de SITL

Capítulo 4

Driver APM Server

El driver `APM Server` es quien va a mediar entre las aplicaciones de JdeRobot y el robot aéreo con sus sensores y actuadores físicos o simulados a través de SITL. De esta manera las aplicaciones pueden correr en máquinas distintas, no obligatoriamente a bordo, y pueden estar implementadas en distintos lenguajes de programación. Estas ventajas vienen de utilizar la división habitual en JdeRobot entre componentes drivers y componentes aplicación, que se pueden conectar entre sí remotamente.

En nuestro caso este driver se ejecuta sobre una Raspberry Pi3 con raspbian a la que hemos instalado el entorno JdeRobot y el driver `APM Server`, e irá a bordo del robot físico conectado por puerto serie a la placa estabilizadora (APM).

4.1. Diseño

El driver tiene que cubrir las siguientes funcionalidades:

- Debe ser capaz de conectar con dispositivos físicos de estabilización como APM 2.8 o PixHawk, así como al simulador SITL
- Debe acceder a los sensores del robot aéreo, interpretar sus datos y servirlos en forma de interfaz ICE a las aplicaciones de control.
- Debe ser capaz de recibir órdenes y comandos a través de interfaces JdeRobot codificarlas y enviarlas los actuadores del robot aéreo.

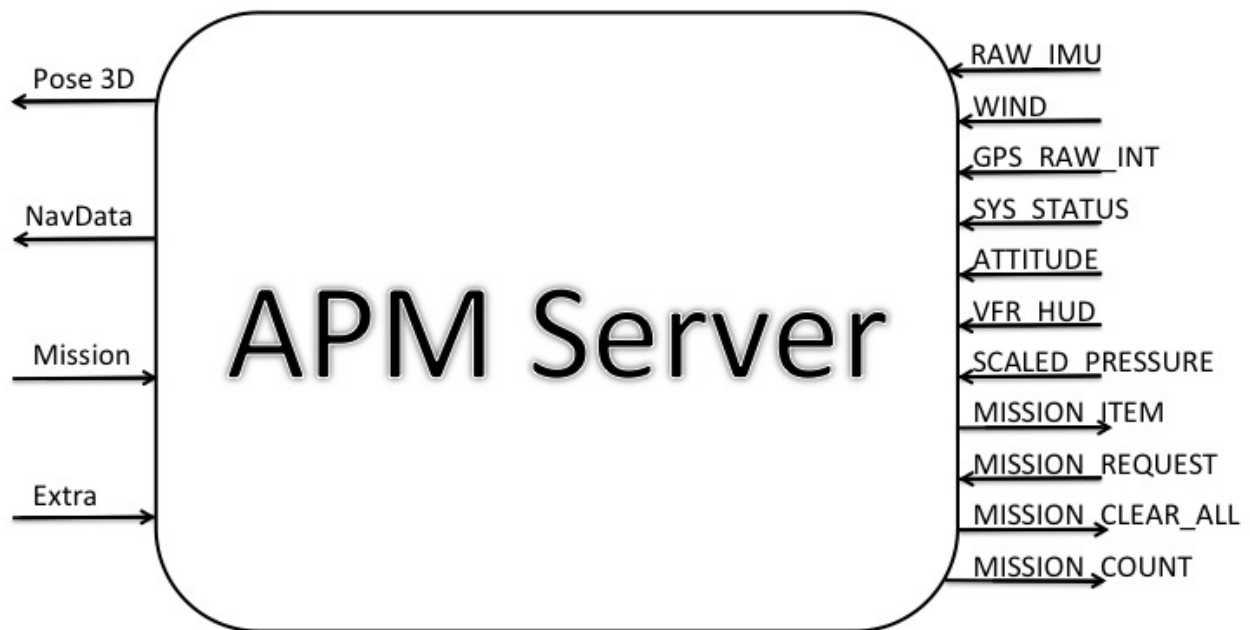


Figura 4.1: Diseño de entradas y salidas de APM Server

Se ha diseñado y programado un componente de JdeRobot en Python que hace de driver y que satisface esos requerimientos. Se ha organizado en tres capas, desde la más cercana al hardware hasta la más cercana a las aplicaciones:

- Capa de comunicación con el dispositivo APM. Esta capa se encarga de la comunicación del driver con el dispositivo APM a través del protocolo de comunicación MavLink.
- Capa de negocio e interpretación. En esta capa el driver transforma los comandos MAVLink y los interfaces JdeRobot interpretando la información de ambas capas y haciendo ésta legible en ambos sentidos.
- Capa de comunicación con aplicaciones JdeRobot. En ésta capa se materializan los servicios ICE necesarios para la recepción y envío desde/hacia las aplicaciones, a través de los interfaces ICE que implementa.

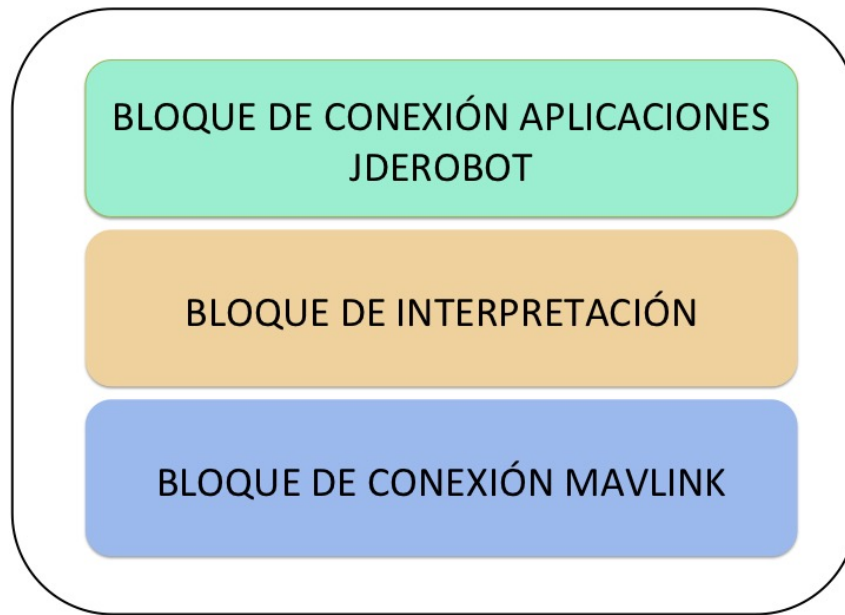


Figura 4.2: Bloques del driver APM Server

4.2. Bloque de conexión MAVLink

En la capa de comunicación con el APM es dónde se van a recibir comandos MAVLink desde la placa estabilizadora y autopiloto y se van a consumir y enviar los siguientes mensajes. Para cada uno de ellos se detalla su nombre, su sintaxis y su semántica.

4.2.1. Listado de mensajes MAVLink

- Mensaje RAW_IMU, donde recibimos la información de los acelerómetros giróscopos y magnetrómetros en crudo para el NavData.

```

<message id="27" name="RAW_IMU">
  <description>The RAW IMU readings for the usual 9DOF sensor setup.
    This message should always contain the true raw values without any
    scaling to allow data capture and system debugging.</description>
  <field type="uint64_t" name="time_usec" units="us">Timestamp
    (microseconds since UNIX epoch or microseconds since system boot)</field>
  <field type="int16_t" name="xacc">X acceleration (raw)</field>
  <field type="int16_t" name="yacc">Y acceleration (raw)</field>
  <field type="int16_t" name="zacc">Z acceleration (raw)</field>
  <field type="int16_t" name="xgyro">Angular speed around X axis (raw)</field>
  <field type="int16_t" name="ygyro">Angular speed around Y axis (raw)</field>
  <field type="int16_t" name="zgyro">Angular speed around Z axis (raw)</field>

```

```
<field type="int16_t" name="xmag">X Magnetic field (raw)</field>
<field type="int16_t" name="ymag">Y Magnetic field (raw)</field>
<field type="int16_t" name="zmag">Z Magnetic field (raw)</field>
</message>
```

Un ejemplo de mensaje recibido sería: RAW_IMU {time_usec : 480794000, xacc : 244, yacc : 0, zacc : -968, xgyro : 1, ygyro : 1, zgyro : 1, xmag : 335, ymag : 60, zmag : -452}

- Mensaje VFR_HUD, donde se presentan los típicos datos de navegación en ala fija como, la velocidad en el aire, velocidad en tierra o la velocidad de ascenso. De aquí obtendremos la altitud

```
<message id="74" name="VFR_HUD">
  <description>Metrics typically displayed on a HUD for fixed wing aircraft</description>
  <field type="float" name="airspeed" units="m/s">Current airspeed in m/s</field>
  <field type="float" name="groundspeed" units="m/s">Current ground speed in m/s</field>
  <field type="int16_t" name="heading" units="deg">Current heading in degrees,
    in compass units (0..360, 0=north)</field>
  <field type="uint16_t" name="throttle" units="%">Current throttle setting in
    integer percent, 0 to 100</field>
  <field type="float" name="alt" units="m">Current altitude (MSL), in meters</field>
  <field type="float" name="climb" units="m/s">Current climb rate in meters/second</field>
</message>
```

Un ejemplo de mensaje recibido sería: VFR_HUD {airspeed : 0.022061701864004135, groundspeed : 0.08534427732229233, heading : 356, throttle : 0, alt : 584.1099853515625, climb : -0.33149993419647217}

- Mensaje ATTITUDE, donde recibimos la actitud del avión.

```
<message id="30" name="ATTITUDE">
  <description>The attitude in the aeronautical frame
    (right-handed, Z-down, X-front, Y-right).</description>
  <field type="uint32_t" name="time_boot_ms" units="ms">Timestamp
    (milliseconds since system boot)</field>
  <field type="float" name="roll" units="rad">Roll angle (rad, -pi..pi)</field>
  <field type="float" name="pitch" units="rad">Pitch angle (rad, -pi..pi)</field>
  <field type="float" name="yaw" units="rad">Yaw angle (rad, -pi..pi)</field>
  <field type="float" name="rollspeed" units="rad/s">Roll angular speed (rad/s)</field>
  <field type="float" name="pitchspeed" units="rad/s">Pitch angular speed (rad/s)</field>
  <field type="float" name="yawspeed" units="rad/s">Yaw angular speed (rad/s)</field>
</message>
```

Un ejemplo de mensaje recibido sería: ATTITUDE {time_boot_ms : 480794, roll : -0.005003694910556078, pitch : 0.2430807501077652, yaw : -0.06844368577003479,

rollspeed : -0.0010073177982121706, pitchspeed : -0.0008516315137967467, yawspeed : -0.0006909647490829229}

- Mensaje SYS_STATUS, donde se obtiene el nivel de batería.

```
<message id="1" name="SYS_STATUS">
  <description>The general system state. If the system is following the MAVLink standard,
    the system state is mainly defined by three orthogonal states/modes: The system mode,
    which is either LOCKED (motors shut down and locked), MANUAL (system under RC control),
    GUIDED (system with autonomous position control, position setpoint controlled manually)
    or AUTO (system guided by path/waypoint planner).
    The NAV_MODE defined the current flight state:
      LIFTOFF (often an open-loop maneuver),
      LANDING
      WAYPOINTS
      VECTOR.
    This represents the internal navigation state machine. The system status shows wether
    the system is currently active or not and if an emergency occured.
    During the CRITICAL and EMERGENCY states the MAV is still considered to be active,
    but should start emergency procedures autonomously. After a failure occured it should
    first move from active to critical to allow manual intervention and then move to
    emergency after a certain timeout.</description>
  <field type="uint32_t" name="onboard_control_sensors_present" enum="MAV_SYS_STATUS_SENSOR"
    display="bitmask" print_format="0x%04x">
    Bitmask showing which onboard controllers and sensors are present. Value of 0: not
    present. Value of 1: present. Indices defined by ENUM MAV_SYS_STATUS_SENSOR</field>
  <field type="uint32_t" name="onboard_control_sensors_enabled" enum="MAV_SYS_STATUS_SENSOR"
    display="bitmask" print_format="0x%04x">
    Bitmask showing which onboard controllers and sensors are enabled: Value of 0:
    not enabled. Value of 1: enabled. Indices defined by ENUM MAV_SYS_STATUS_SENSOR</field>
  <field type="uint32_t" name="onboard_control_sensors_health" enum="MAV_SYS_STATUS_SENSOR"
    display="bitmask" print_format="0x%04x">
    Bitmask showing which onboard controllers and sensors are operational or have an error:
    Value of 0: not enabled. Value of 1: enabled.
    Indices defined by ENUM MAV_SYS_STATUS_SENSOR</field>
  <field type="uint16_t" name="load" units="d%">Maximum usage in percent of the mainloop
    time, (0%: 0, 100%: 1000) should be always below 1000</field>
  <field type="uint16_t" name="voltage_battery" units="mV">Battery voltage, in millivolts
    (1 = 1 millivolt)</field>
  <field type="int16_t" name="current_battery" units="cA">Battery current, in
    10*milliamperes (1 = 10 milliampere), -1: autopilot does not measure the current</field>
  <field type="int8_t" name="battery_remaining" units="%">Remaining battery energy:
    (0%: 0, 100%: 100), -1: autopilot estimate the remaining battery</field>
  <field type="uint16_t" name="drop_rate_comm" units="c%">Communication drops in percent,
    (0%: 0, 100%: 10'000), (UART, I2C, SPI, CAN), dropped packets on all links
    (packets that were corrupted on reception on the MAV)</field>
  <field type="uint16_t" name="errors_comm">Communication errors (UART, I2C, SPI, CAN),
```

```

    dropped packets on all links (packets that were corrupted on reception on the MAV)</field>
    <field type="uint16_t" name="errors_count1">Autopilot-specific errors</field>
    <field type="uint16_t" name="errors_count2">Autopilot-specific errors</field>
    <field type="uint16_t" name="errors_count3">Autopilot-specific errors</field>
    <field type="uint16_t" name="errors_count4">Autopilot-specific errors</field>
</message>

```

Un mensaje recibido sería: SYS_STATUS {onboard_control_sensors_present : 23198783, onboard_control_sensors_enabled : 23198783, onboard_control_sensors_health : 24247359, load : 0, voltage_battery : 12587, current_battery : 0, battery_remaining : 100, drop_rate_comm : 0, errors_comm : 0, errors_count1 : 0, errors_count2 : 0, errors_count3 : 0, errors_count4 : 0}

- Mensaje SCALED_PRESSURE, donde obtendremos la presión absoluta y la temperatura.

```

<message id="29" name="SCALED_PRESSURE">
  <description>The pressure readings for the typical setup of one absolute and differential
    pressure sensor. The units are as specified in each field.</description>
  <field type="uint32_t" name="time_boot_ms" units="ms">Timestamp
    (milliseconds since system boot)</field>
  <field type="float" name="press_abs" units="hPa">Absolute pressure (hectopascal)</field>
  <field type="float" name="press_diff" units="hPa">Differential pressure 1 </field>
  <field type="int16_t" name="temperature" units="cdegC">Temperature measurement
    (0.01 degrees celsius)</field>
</message>

```

Un mensaje recibido sería: SCALED_PRESSURE {time_boot_ms : 480794, press_abs : 945.0001831054688, press_diff : 0.021015625447034836, temperature : 2600}

- Mensaje WIND, donde se obtienen las lecturas del viento estimadas.

```

<message id="168" name="WIND">
  <description>Wind estimation</description>
  <field name="direction" type="float">wind direction that wind is coming from </field>
  <field name="speed" type="float">wind speed in ground plane (m/s)</field>
  <field name="speed_z" type="float">vertical wind speed (m/s)</field>
</message>

```

Un mensaje recibido sería: WIND {direction : -179.99998474121094, speed : 0.0, speed_z : 0.0}

- Mensaje GLOBAL_POSITION_INT, donde obtendremos la posición GPS.

```

<message id="33" name="GLOBAL_POSITION_INT">
  <description>The filtered global position (e.g. fused GPS and accelerometers).
    The position is in GPS-frame (right-handed, Z-up). It is designed as scaled integer
    message since the resolution of float is not sufficient.</description>
  <field type="uint32_t" name="time_boot_ms" units="ms">Timestamp
    (milliseconds since system boot)</field>
  <field type="int32_t" name="lat" units="degE7">Latitude, expressed as degrees * 1E7</field>
  <field type="int32_t" name="lon" units="degE7">Longitude, expressed as degrees * 1E7</field>
  <field type="int32_t" name="alt" units="mm">Altitude in meters, expressed as
    * 1000 (millimeters), AMSL (not WGS84 - note that virtually all GPS modules provide
    the AMSL as well)</field>
  <field type="int32_t" name="relative_alt" units="mm">Altitude above ground in meters,
    expressed as * 1000 (millimeters)</field>
  <field type="int16_t" name="vx" units="cm/s">Ground X Speed (Latitude, positive north),
    expressed as m/s * 100</field>
  <field type="int16_t" name="vy" units="cm/s">Ground Y Speed (Longitude, positive east),
    expressed as m/s * 100</field>
  <field type="int16_t" name="vz" units="cm/s">Ground Z Speed (Altitude, positive down),
    expressed as m/s * 100</field>
  <field type="uint16_t" name="hdg" units="cdeg">Vehicle heading (yaw angle) in degrees
    * 100, 0.0..359.99 degrees. If unknown, set to: UINT16_MAX</field>
</message>

```

Un mensaje recibido sería: GLOBAL_POSITION_INT {time_boot_ms : 480614, lat : -353632612, lon : 1491652301, alt : 584110, relative_alt : -179, vx : 0, vy : 0, vz : 0, hdg : 35608}

- Mensaje MISSION_ITEM, son objetos de misión, en ellos se mandan los puntos de paso o comandos, como fijar una velocidad o una altitud o bien aterrizar o despegar en función del parámetro command.

```

<message id="39" name="MISSION_ITEM">
  <description>Message encoding a mission item. This message is emitted to announce the
    presence of a mission item and to set a mission item on the system. The mission item
    can be either in x, y, z meters (type: LOCAL) or x:lat, y:lon, z:altitude.
    Local frame is Z-down, right handed (NED), global frame is Z-up, right handed (ENU).
    See also http://qgroundcontrol.org/mavlink/waypoint\_protocol.</description>
  <field type="uint8_t" name="target_system">System ID</field>
  <field type="uint8_t" name="target_component">Component ID</field>
  <field type="uint16_t" name="seq">Sequence</field>
  <field type="uint8_t" name="frame" enum="MAV_FRAME">The coordinate system of the MISSION.
    see MAV_FRAME in mavlink_types.h</field>
  <field type="uint16_t" name="command" enum="MAV_CMD">The scheduled action for the MISSION.
    see MAV_CMD in common.xml MAVLink specs</field>
  <field type="uint8_t" name="current">false:0, true:1</field>
  <field type="uint8_t" name="autocontinue">autocontinue to next wp</field>

```

```

<field type="float" name="param1">PARAM1, see MAV_CMD enum</field>
<field type="float" name="param2">PARAM2, see MAV_CMD enum</field>
<field type="float" name="param3">PARAM3, see MAV_CMD enum</field>
<field type="float" name="param4">PARAM4, see MAV_CMD enum</field>
<field type="float" name="x">PARAM5 / local: x position, global: latitude</field>
<field type="float" name="y">PARAM6 / y position: global: longitude</field>
<field type="float" name="z">PARAM7 / z position: global: altitude (relative or absolute,
    depending on frame.</field>
<extensions/>
<field type="uint8_t" name="mission_type" enum="MAV_MISSION_TYPE">Mission type,
    see MAV_MISSION_TYPE</field>
</message>

```

Un mensaje enviado sería: MISSION_ITEM {target_system : 1, target_component : 1, seq : 0, frame : 3, command : 16, current : 0, autocontinue : 0, param1 : 0, param2 : 10, param3 : 0, param4 : 0, x : 40.33024215698242, y : -3.8008816242218018, z : 40.0}

- Mensaje MISSION_REQUEST, donde el estabilizador nos requiere el siguiente MISSION_ITEM. Este comando se recibe una vez enviado al APM nuestra intención de enviarle mensajes de misión y cuántos objetos vamos a enviarle con el comando MISSION_COUNT. También se recibe este comando tras la recepción de alguno de estos MISSION_ITEM, como confirmación de la recepción de éste, y solicitando el siguiente objeto de misión.

```

<message id="40" name="MISSION_REQUEST">
  <description>Request the information of the mission item with the sequence number seq.
    The response of the system to this message should be a MISSION_ITEM message.
    http://qgroundcontrol.org/mavlink/waypoint_protocol</description>
  <field type="uint8_t" name="target_system">System ID</field>
  <field type="uint8_t" name="target_component">Component ID</field>
  <field type="uint16_t" name="seq">Sequence</field>
  <extensions/>
  <field type="uint8_t" name="mission_type" enum="MAV_MISSION_TYPE">Mission type,
    see MAV_MISSION_TYPE</field>
</message>

```

Un mensaje recibido sería: MISSION_REQUEST {target_system : 0, target_component : 0, seq : 2} En este mensaje el APM nos estaría solicitando el item de misión 3, ya que empieza solicitando el 0. Este mensaje termina al llegar al máximo de mission item que se ha indicado que le se le va a mandar.

- Mensaje MISSION_COUNT, éste comando se envía para comenzar una comunicación con el APM con el objetivo de enviarle una misión.


```
<message id="44" name="MISSION_COUNT">
  <description>This message is emitted as response to MISSION_REQUEST_LIST by the MAV
    and to initiate a write transaction. The GCS can then request the individual
    mission item based on the knowledge of the total number of MISSIONs.</description>
  <field type="uint8_t" name="target_system">System ID</field>
  <field type="uint8_t" name="target_component">Component ID</field>
  <field type="uint16_t" name="count">Number of mission items in the sequence</field>
  <extensions/>
  <field type="uint8_t" name="mission_type" enum="MAV_MISSION_TYPE">Mission type,
    see MAV_MISSION_TYPE</field>
</message>
```

Un mensaje enviado sería: MISSION_COUNT {target_system : 0, target_component : 0, count : 3} Donde comunicamos al APM nuestra intención de enviarle 3 objetos de misión.

- Mensaje MISSION_CLEAR_ALL, este comando sirve para limpiar el APM de misiones, si éste tenía alguna misión previa se desecha.

```
<message id="45" name="MISSION_CLEAR_ALL">
  <description>Delete all mission items at once.</description>
  <field type="uint8_t" name="target_system">System ID</field>
  <field type="uint8_t" name="target_component">Component ID</field>
  <extensions/>
  <field type="uint8_t" name="mission_type" enum="MAV_MISSION_TYPE">Mission type,
    see MAV_MISSION_TYPE</field>
</message>
```

Un mensaje enviado sería MISSION_CLEAR_ALL {target_system : 0, target_component : 0}

4.2.2. Conexión y configuración de la comunicación con el APM

De cara a facilitar la implementación del software, ahorrarnos crear los comandos MAVLink a mano, y hacer el código más legible, utilizaremos la librería Pymavlink.

Las importaciones que se utilizan son las siguientes:

```
1 from pymavlink import mavutil, quaternion, mavwp
2 from pymavlink.dialects.v10 import ardupilotmega as mavlink
```

Se importa mavutil y ardupilotmega para obtener los interfaces en python desde pymavlink y mavwp y quaternion como soporte a la transformación a cuaterniones y viceversa.

```

1  class Server:
2
3      def __init__(self, port, baudrate):
4          self.master = mavutil.mavlink_connection(port, baudrate, autoreconnect=True)
5          print('Connection established to device')
6          self.master.wait_heartbeat()
7          print("Heartbeat Recieved")
8
9  #test = Server("/dev/ttyUSB0", 57600) # Connection to the real APM device
10 test = Server("udp:192.168.1.133:14558",57600) # Connection to SITL

```

Figura 4.3: Conexión con el APM

```

1  RATE = 50
2  self.master.mav.request_data_stream_send(self.master.target_system,
3                                          self.master.target_component,
4                                          mavutil.mavlink.MAV_DATA_STREAM_ALL,
5                                          RATE, 1)

```

Figura 4.4: configuración de la conexión

El primer paso para interactuar con el APM es el proceso de conexión (Figura 4.3). En el comando de conexión se le indica una de las siguiente tuplas:

- Dispositivo_serie, velocidad de conexión en baudios (Línea 9 de la Figura 4.3).
- Tipo_de_puerto:IP:puerto, velocidad de conexión en baudios. En este caso los baudios son obligatorios pero se desechan (Línea 10 de la Figura 4.3).

Con estas líneas se inicia el servidor de APM `Server` y se realiza la conexión con el dispositivo APM o con el simulador. Se puede apreciar cómo se realiza la conexión en el método `__init__` del servidor. Y la creación de un objeto de servidor, que conecta con el dispositivo real y en el comentario de la instrucción que hace lo propio con el dispositivo simulado en SITL (Línea 9 de la Figura 4.3).

Una vez conectado, se debe indicar al APM qué mensajes se quieren recibir y la frecuencia con la que se quieren recibir (Figura 4.4). Debido a que se trata de un prototipo y se trata de extraer el máximo potencial del mismo, se indicará el juego de instrucciones completo y una frecuencia de moderada a alta 50 hz, lo máximo que soporta el APM adquirido.

Desde este momento se van a estar volcando a un *buffer* los mensajes que manda nuestro APM, con lo que ahora ya se puede acceder a ellos desde Python para procesarlos.

4.2.3. Lectura de datos del APM

Una vez establecida la conexión y fijados los parámetros de configuración ya se puede acceder a los datos. Para ésto en el bloque de transformación se levantará el siguiente hilo que procesará los mensajes. Con el fin de evitar una desconexión con el APM envíamos periódicamente heartbeat al mismo. heartbeat es un comando que sirve para conocer si el dispositivo esta disponible y atiende peticiones, de esta forma nos aseguramos cada cierto tiempo que el APM continúa activo.

```
1  MsgHandler = threading.Thread(target=self.mavMsgHandler, args=(self.master,), name='
    msg_Handler')
2  MsgHandler.start()
3
4
5  def mavMsgHandler(self, m):
6
7      while True:
8          msg = m.recv_msg()
9
10         if time.time() - self.lastSentHeartbeat > 1.0:
11             self.master.mav.heartbeat_send(mavlink.MAV_TYPE_GCS, mavlink.MAV_AUTOPILOT_INVALID
                , 0, 0, 0)
12             self.lastSentHeartbeat = time.time()
13
14             # refresh the attitude
15             self.refreshAPMPose3D()
16             self.refreshAPMnavdata()
17
18         elif msg is None or msg.get_type() == "BAD_DATA":
19             time.sleep(0.01)
20             continue
```

Figura 4.5: Hilo de alimentación de mensajes MAVLink

4.2.4. Envío de misiones al APM

Con el fin de entender mejor la implementación del método `self.setMission(mission)`, en la Figura 4.6 podemos observar un gráfico de cómo es el protocolo de envío de misiones al APM a través de MAVLink, obtenido de la universidad University of Colorado Boulder [7]

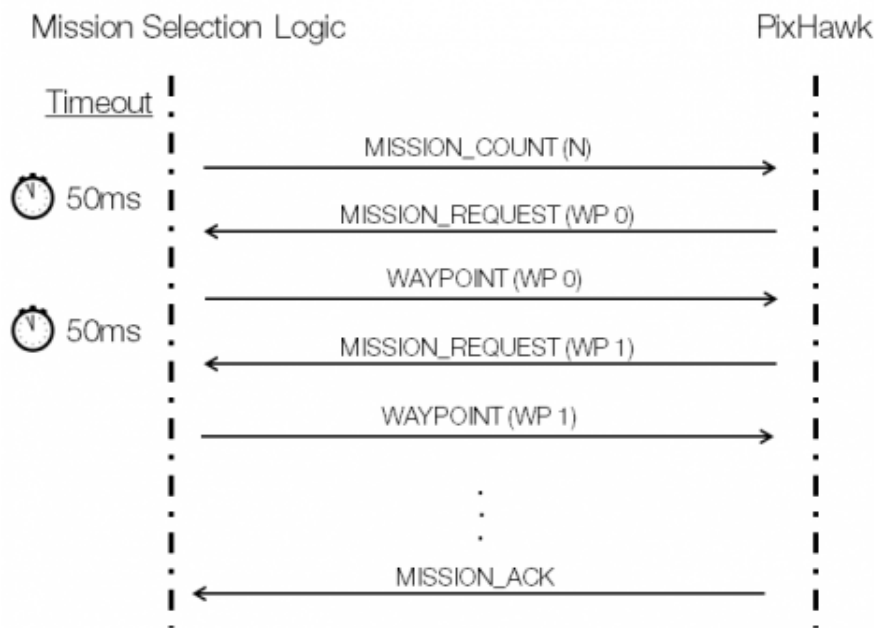


Figura 4.6: Protocolo de envío de misiones MAVLink

Para abordar el problema e implementar dicho protocolo partimos de la solución que nos propone la Universidad de Colorado Boulder [7] y la adaptamos a nuestras necesidades, recibiendo de entrada un objeto de la interfaz `mission` de JdeRobot, transformándolo y enviándolo como misión de MavLink al APM. También revisamos si se han enviado órdenes de despegue y aterrizaje y en el caso de que así sea añadimos sus *mission item* al principio o al final de la misión. En este punto se entrelazan la capa de interpretación con la capa de comunicación con el APM

```

1  def setMission(self, mission):
2
3      wp = mavwp.MAVWPLoader()
4      seq = 1
5      frame = mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT
6      radius = 10
7      pose3Dwaypoints = mission.mission
8      N = len(pose3Dwaypoints)
  
```

```

9         if (self.extra.takeOffDecision):
10             navData = pose3Dwaypoints[seq-1]
11             toff = mavutil.mavlink.MAVLink_mission_item_message(self.master.target_system,
12                                                                 self.master.target_component,
13                                                                 seq,
14                                                                 frame,
15                                                                 mavutil.mavlink.
16                                                                 MAV_CMD_NAV_TAKEOFF, # 22
17                                                                 0, 0, 0, radius, 0, 0,
18                                                                 navData.x, navData.y, navData.
19                                                                 h)
20
21             ...
22
23         for i in range(N):
24             navData = pose3Dwaypoints[i]
25             wayPoint_tmp = mavutil.mavlink.MAVLink_mission_item_message(self.master.
26                                                                 target_system,
27                                                                 self.master.target_component,
28                                                                 seq,
29                                                                 frame,
30                                                                 mavutil.mavlink.
31                                                                 MAV_CMD_NAV_WAYPOINT, # 16
32                                                                 0, 0, 0, radius, 0, 0,
33                                                                 navData.x, navData.y, navData.
34                                                                 h)
35
36             ...
37
38         if (self.extra.landDecision):
39             navData = pose3Dwaypoints[N-1]
40             land = mavutil.mavlink.MAVLink_mission_item_message(self.master.target_system,
41                                                                 self.master.target_component,
42                                                                 seq,
43                                                                 frame,
44                                                                 mavutil.mavlink.
45                                                                 MAV_CMD_NAV_LAND, # 21
46                                                                 0, 0, 0, radius, 0, 0,
47                                                                 navData.x, navData.y, navData.
48                                                                 h)
49
50             ...
51
52         self.master.waypoint_clear_all_send()
53         self.master.waypoint_count_send(wp.count())
54
55         for i in range(wp.count()):
56             msg = self.master.recv_match(type=['MISSION_REQUEST'], blocking=True)
57             self.master.mav.send(wp.wp(i))
58

```

```

49     self.master.arducopter_arm()
50     self.master.set_mode_auto() # start mission
51     ...

```

Como se puede observar en el código, en caso de tener `takeOffDecision` o `landDecision` activos en el atributo `extra` (enlazado a través de ICE) se añaden los comandos de despegue y aterrizaje, que interpreta son el primero y el último. Se puede observar también al final del código cómo se arman los motores con la línea `self.master.arducopter_arm()` y cómo se pone el dispositivo en modo automático con `self.master.set_mode_auto()`. El driver está implementado para ejecutarse en modo “auto”. Se puede ejecutar también en modo “guided” pero el modo “auto”, a diferencia del “guided”, no inhabilita la radio y se puede recuperar en cualquier momento el control del robot aéreo. Si bien hay que decir que el modo “guided” o modo guiado es más preciso que el modo “auto”.

4.3. Capa de conexión con aplicaciones JdeRobot

En esta capa se sirven y reciben los objetos de interfaces ICE JdeRobot a través de los cuales se habla con la aplicación (que es el destino final de la información de sensores y la fuente original de los comandos de actuación para el avión), de modo que sean entendibles por el resto del driver. Para esto se han implementado cuatro hilos, uno por cada interfaz JdeRobot que se sirve o se recibe. En cada uno de estos hilos se levanta un servidor ICE para recibir o servir objetos de interfaces JdeRobot mapeando atributos de la clase del servidor a estos servicios ICE. La figura 4.7 muestra dos ejemplos.

Los 4 interfaces que van a ser servidos desde APM Server son: Pose3D, NavData, Extra y Mission. Pose3D, NavData y Extra se describieron en la sección 3.1.1. Adicionalmente a éstos se ha desarrollado un nuevo interfaz para dar soporte al uso de misiones, el interfaz `mission`.

```

class Pose3DData //we consumes Pose3DData
{
float x; //latitude
float y; //longitude
float z; //altitude
float h; //not used now
float q0; //quaternion component 1
float q1; //quaternion component 2
float q2; //quaternion component 3
float q3; //quaternion component 4

```

```

};

["python:seq:list"] sequence<Pose3DData> PoseSequence; // list of Pose3DData
/**
 * Mission data information, Pose3DData sequence.
 */
class MissionData
{
    PoseSequence mission;
};

/**
 * Interface to the Mission.
 */
interface Mission
{
    idempotent MissionData getMissionData();
    int setMissionData(MissionData data);
};

```

```

1
2     PoseTheading = threading.Thread(target=self.openPose3DChannel, args=(self.pose3D,),
3                                     name='Pose_Theading')
4     PoseTheading.daemon = True
5     PoseTheading.start()
6     ...
7
8     ExtraTheading = threading.Thread(target=self.openExtraChannel, args=(self.extra,),
9                                     name='Extra_Theading')
10    ExtraTheading.daemon = True
11    ExtraTheading.start()

```

Figura 4.7: Creación y arranque de los hilos encargados de servir los interfaces ICE

Cada hilo abre un servicio ICE, como todos los servicios ICE que implementamos son similares podremos como ejemplo de implementación el servicio de Pose3D. Como se puede apreciar en esta capa tenemos cuatro hilos de ejecución simultáneos sirviendo objetos ICE a través de servidores como el de la Figura 4.8. Como se ha descrito en la sección 4.2.3 en paralelo se tiene otro hilo que procesa los mensajes recibidos del APM o SITL, los interpreta y cumplimenta estos objetos ICE, con lo que se ha tenido que cuidar el código para evitar problemas de concurrencia. En las Figuras 4.7 y 4.8 se puede observar cómo se crea un hilo

asocado a un servidor ICE y cómo se implementa un servidor ICE respectivamente.

```

1  def openPose3DChannel(self, pose3D):
2      status = 0
3      ic = None
4      # recovering the attitude
5      Pose2Tx = pose3D
6      try:
7          ic = ICE.initialize(sys.argv)
8          adapter = ic.createObjectAdapterWithEndpoints("Pose3DAdapter", "default -p 9998")
9          object = Pose2Tx
10         # print object.getPose3DData()
11         adapter.add(object, ic.stringToIdentity("ardrone_pose3d")) #ardrone_pose3d Pose3D
12         adapter.activate()
13         ic.waitForShutdown()
14     except:
15         traceback.print_exc()
16         status = 1
17     if ic:
18         # Clean up
19         try:
20             ic.destroy()
21         except:
22             traceback.print_exc()
23             status = 1
24
25     sys.exit(status)

```

Figura 4.8: Servidor ICE encargado de servir objetos de la interfaz Pose3D

4.4. Bloque de interpretación

En esta capa se interpretan los mensajes recibidos de una y otra capa y se transforman en el formato que sea necesario para que lo interprete la opuesta.

Al método `setMission(mission)` se llama al recibir vía ICE una misión. Para identificar cuándo llega, debido a que tenemos mapeadas las variables con ICE, se ha construido un *listener* que valida el valor de la variable `mission` y si tiene contenido articula el método `setMission(mission)`.

Ya se visto cómo se realiza la transformación entre mensajes de misión en el método `setMission(mission)`, ahora nos centraremos en interpretar los mensajes recibidos del APM


```

1  MissionListener = threading.Thread(target=self.missionListener, name='MissionListener')
2  MissionListener.daemon = True
3  MissionListener.start()
4
5  def missionListener(self):
6      """
7      Function who listen to a new mission recieved from ICE MissionChannel thread and send
        it to APM
8      :return: None
9      """
10     while True:
11         if not self.mission.is_empty():
12             self.lastMission = self.mission
13             self.setMission(self.mission.getMissionData())
14             time.sleep(1)

```

Figura 4.9: Código encargado de detectar misiones entrantes

y crear los objetos de las interfaces que sean necesarias para su interpretación por las aplicaciones JdeRobot. Esto lo realizamos principalmente en dos métodos, `refreshAPMPose3D()` y `refreshAPMnavdata()`. Estos métodos se centran en buscar dentro de los paquetes recibidos del APM aquellos que contienen información relevante, los descritos en la sección 4.2, analizarlos y extraer de ellos la información necesaria y guardarla en objetos de interfaces JdeRobot para ser servidos a través de ICE.

En la Figura 4.10 se pueden observar los métodos que recogen la actitud, podemos ver cómo se buscan los mensajes que necesitamos para extraer la información de los sensores desde `self.master.messages`. En esta variable se vuelcan los datos de todos los mensajes recibidos de la placa estabilizadora.

```

1  def refreshAPMPose3D(self):
2      # get attitude of APM
3      if 'ATTITUDE' not in self.master.messages:
4          self.attitudeStatus = 1
5          q=[0,0,0,0]
6      else:
7          attitude = self.master.messages['ATTITUDE']
8          yaw = getattr(attitude, "yaw")
9          pitch = getattr(attitude, "pitch") * -1
10         roll = getattr(attitude, "roll")
11         q = quaternion.Quaternion([roll, pitch, yaw])
12
13     # get altitude of APM
14     altitude = self.master.field('VFR_HUD', 'alt', None)
15     if altitude is None:
16         self.altitudeStatus = 1
17
18     # get GPS position from APM
19     latitude = 0
20     longitude = 0
21     if 'GPS_RAW_INT' not in self.master.messages:
22         self.gpsStatus = 1
23     else:
24         gps = self.master.messages['GPS_RAW_INT']
25         latitude = getattr(gps, "lat") / 10e6
26         longitude = getattr(gps, "lon") / 10e6
27         self.GPS_fix_type = getattr(gps, "fix_type")
28
29     # refresh the pose3D
30     data = jderobot.Pose3DData
31     data.x = latitude
32     data.y = longitude
33     data.z = altitude
34     data.h = altitude
35     data.q0 = q.__getitem__(0)
36     data.q1 = q.__getitem__(1)
37     data.q2 = q.__getitem__(2)
38     data.q3 = q.__getitem__(3)
39     self.pose3D.setPose3DData(data)

```

Figura 4.10: Código de ejemplo de `refreshAPMPose3D()` encargado de recorrer los mensajes y obtener los datos de actitud que se van a servir a través de Pose3D

Capítulo 5

Aplicación AUV Commander

UAV Commander es una aplicación JdeRobot que hace las veces de estación de tierra o GCS (*Ground Control Station*) para robots aéreos y que ha sido íntegramente desarrollada dentro de este proyecto. Desde UAV Commander se pueden programar misiones, enviar éstas al driver de un avión autónomo MAVLink, verificar el cumplimiento de las mismas y visualizar los sensores a bordo del robot aéreo. Sirve de cliente del driver desarrollado en la primera parte de este Proyecto Fin de Carrera y proporciona su primera validación experimental.

5.1. Diseño

La aplicación debe cubrir las siguientes funcionalidades:

1. Presentar toda la información sensorial que nos ofrezca el driver, la actitud y las cámaras de abordo
2. Permitir la creación de misiones como secuencias de puntos por los que debe pasar el drone, ubicándolos dentro de un mapa geo-referenciado.
3. Enviar misiones a través de interfaces ICE al driver de JdeRobot que ofrece acceso al avión y es capaz de materializar esas misiones, de gobernar al avión para que las cumpla.

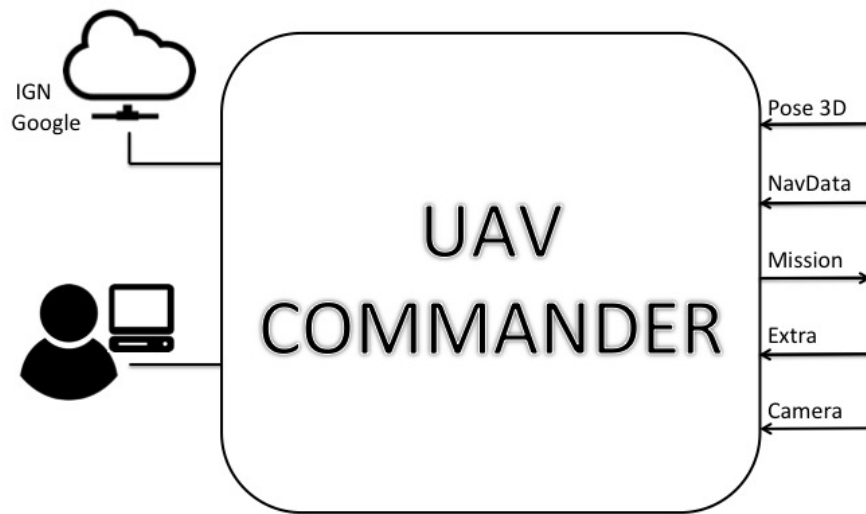


Figura 5.1: Diseño de caja negra de UAV Commander, entradas y salidas

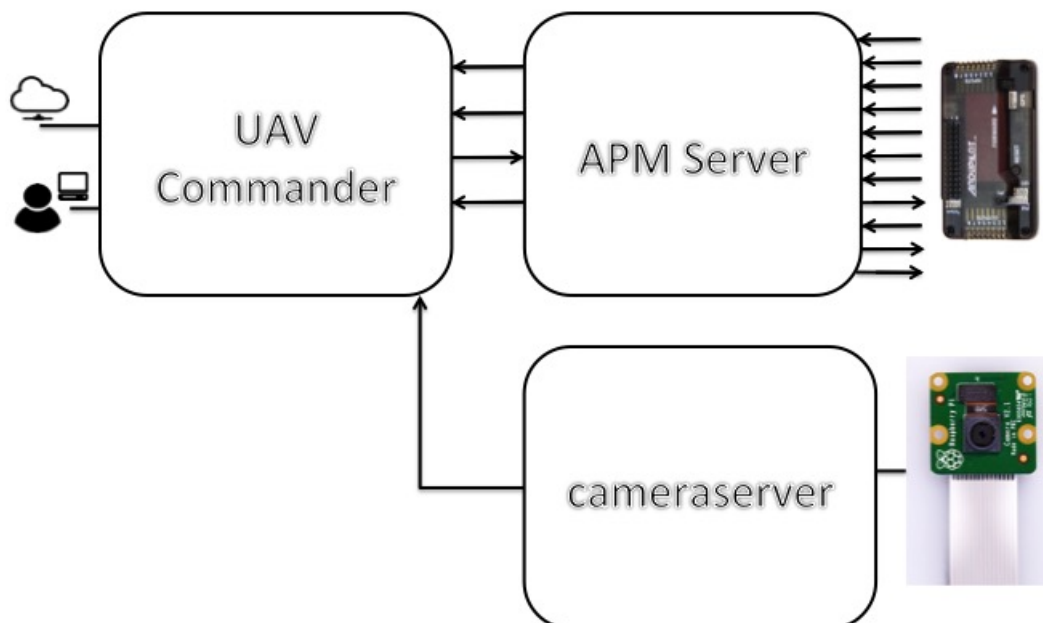


Figura 5.2: Esquema de conexión UAV Commander

La aplicación utiliza los interfaces Pose3D, NavData, Extra, camera y mission y recupera, o bien de Google o bien del Instituto Geográfico Nacional de España (IGN), el mapa geo-referenciado de esa zona. Estos datos se presentan por pantalla al usuario a través de las distintas ventanas que conforman el interfaz gráfico.

La aplicación UAV Commander se estructura en 3 grandes bloques:

- El bloque de sensores es el encargado de recuperar desde los interfaces ICE toda la información sensorial a bordo del avión, tratarla y mostrarla
- El bloque de Localización es el encargado de obtener los mapas geo-referenciados de la zona del mundo donde va a trabajar el avión, realizar las transformaciones entre sistemas de coordenadas y mostrar de forma interactiva éstos a través de la interfaz gráfica de usuario.
- Bloque de control de misiones es quien se encarga de facilitar la creación de misiones a través del interfaz gráfico de usuario y las envía a través de los interfaces ICE involucrados.

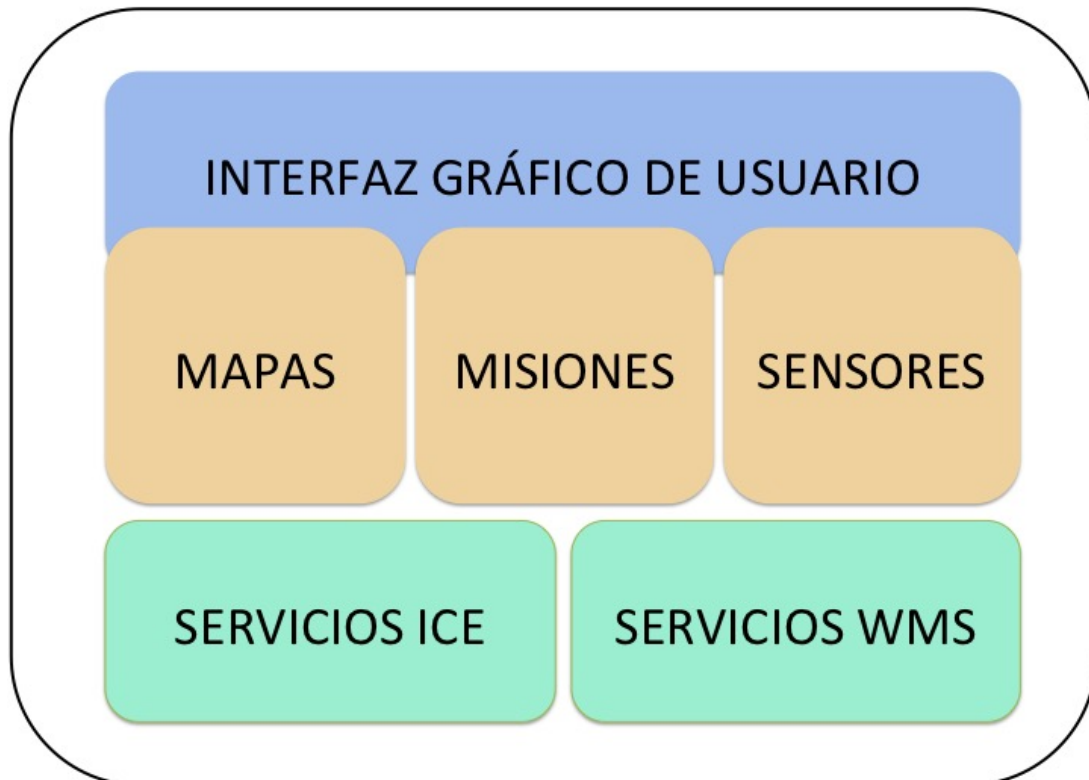


Figura 5.3: Bloques dentro de la aplicación UAV Commander

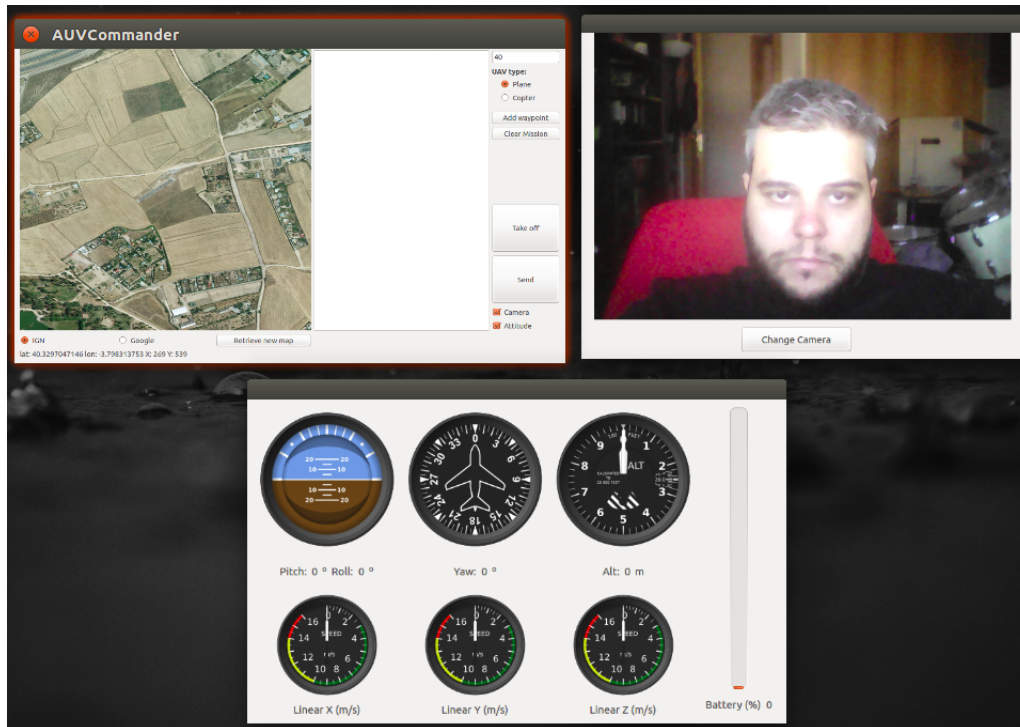


Figura 5.4: Vista general del interfaz gráfico de UAV Commander

La aplicación contiene cinco hilos asociados a cada uno de los servicios ICE que utiliza y dos hilos más para actualizar en el interfaz gráfico de usuario, la posición en el mapa del avión (`threadMap`), y los datos recibidos de los sensores (`threadGUI`).

Cada uno de estos bloques se detalla a continuación en las secciones siguientes.

5.2. Bloque de sensores

En este bloque se implementan todos los servidores ICE necesarios para acceder a la información sensorial y para enviar los comandos de actuación. La aplicación UAV Commander utiliza 5 interfaces ICE a diferencia de los cuatro que usa en el driver APM Server. Además de los que expone APM Server (a saber: `Pose3D` y `NavData` para recibir la posición medida por los sensores a bordo como el GPS y `Extra` y `Mission` para el envío de misiones con puntos de paso y maniobras de aterrizaje o despegue), también utiliza el interfaz `Camera` para recoger las imágenes y vídeo de la cámara a bordo del avión que se muestran en el interfaz gráfico.

5.2.1. Cámara

UAV Commander utiliza el interfaz Camera para obtener las imágenes y vídeos recogidos desde el driver y mostrarlos en su interfaz gráfico. Para esto primero se crea el servicio ICE y se asigna al interfaz de usuario. Una vez hecho esto, cuando el usuario hace *click* en el *checkBox* de camera, se abre la ventana de camera. En esta pantalla se recogen los datos de imagen y se asignan al área dedicado a mostrar las imágenes tal, y como se puede ver en la Figura 5.4

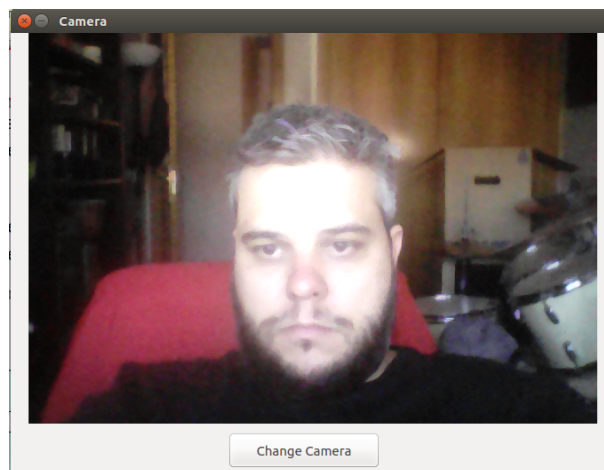


Figura 5.5: Ventana Camera de UAV Commander

```
1 ic = EasyIce.initialize(sys.argv)
2 camera = CameraClient(ic, "UavViewer.Camera", True)
3 screen.setCamera(camera)
```

Figura 5.6: Creación del servicio ICE camera y asignación al GUI

5.2.2. Sensores de vuelo

Para simplificar el proceso de creación y enlazado de los servicios ICE hemos optado por utilizar los servidores (en el sentido de servidores de ICE, sin atender al sentido en el que se transmiten los datos) ya implementadas en JdeRobot, CameraClient, NavDataClient, Pose3DClient y Extra y la clase easyiceconfig. De este modo este bloque queda simplificado y su inicialización queda reducida a unas pocas líneas de código como vemos en la Figura 5.7.

```

1  ic = EasyIce.initialize(sys.argv)
2  camera = CameraClient(ic, "UavViewer.Camera", True)
3  navdata = NavDataClient(ic, "UavViewer.Navdata", True)
4  pose = Pose3DClient(ic, "UavViewer.Pose3D", True)
5  extra = Extra(ic, "UavViewer.Extra")
6  mission = MissionI(ic, "UavViewer.Mission")
7  screen.setPose3D(pose)
8  screen.set_initial_pose3D(pose3D)
9  screen.setNavData(navdata)
10 screen.setExtra(extra)
11 screen.setMission(mission)

```

Figura 5.8: Creación de servicios ICE y asignación al GUI



Figura 5.7: Ventana actitud y batería de UAV Commander

5.3. Bloque de Mapas

Este bloque es el encargado de la obtención de los mapas geo-referenciados y la interpretación de la posición actual sobre ellos. Este bloque realiza las conversiones de sistemas de

referencia necesarios para inferir la posición de un punto geográfico en el planeta a partir de un píxel de la imagen y viceversa, así como las operaciones requeridas para representar ese punto geográfico en la imagen.

5.3.1. Obtención de Mapas

Para obtener los mapas geo-referenciados se ha recurrido a WebMapService (WMS) que es el método más empleado para la obtención de estos mapas. Se han utilizado dos WMS como fuentes, el PNOA del Instituto Geográfico Nacional (IGN) como fuente principal y Google como fuente secundaria.

Con el fin de abstraernos del montaje de la URL y envío y tratamiento de la respuesta utilizamos la librería OWSLib¹, en su versión 0.14.0, que nos ayuda con estos trámites. En el código mostrado en la Figura 5.8 podemos observar cómo recuperamos un nuevo mapa del IGN², que publica uno de sus numerosos WebMapService (WMS) en ‘http://www.ign.es/wms-inspire/pnoa-ma’. Para obtener un mapa geo-referenciado necesitamos principalmente cuatro cosas: la posición, el *bounding box*, el tamaño de la imagen y la proyección a utilizar. Primero obtenemos el *bounding box*. El *bounding box* es la zona geográfica que vamos a recuperar, se calcula a partir de la posición GPS central y, en nuestro caso su lado (*radius*). El código para calcularlo podemos observarlo en el método `getBoundingBox(lat, lon, distance)` de la Figura 5.10.

Una vez obtenido el *bounding box* únicamente rellenamos el resto de parámetros del objeto WMS y lanzamos la petición. El WMS devuelve una imagen en bytes a la que debemos añadir el *disclaimer* que nos indica el IGN para poder utilizar los mapas de forma gratuita. Creamos la petición WMS con la URL del WMS PNOA y asignamos los parámetros necesarios para obtener el mapa. Al final montamos la imagen geo-referenciada, que se implementa como un diccionario con la imagen en OpenCV, el *bounding box*, y el tamaño de la imagen.

5.3.2. Conversión de coordenadas

La conversión de coordenadas o proyección se basa en “ubicar” un punto dado en un determinado sistema de coordenadas en otro sistema completamente distinto. Los datos recibidos

¹<https://geopython.github.io/OWSLib/>

²Instituto geográfico nacional

```

1  def retrieve_new_map(lat, lon, radius, width, height):
2      bbox = getBoundingBox(lat, lon, radius)
3      wms = WebMapService('http://www.ign.es/wms-inspire/pnoa-ma', version='1.3.0')
4      img = wms.getmap(layers=['OI.OrthoimageCoverage'],
5                          styles=['default'],
6                          srs='EPSG:4326',
7                          bbox=bbox,
8                          size=(width, height),
9                          format='image/png',
10                         transparent=True)
11      ...
12      ImageUtils.prepareInitialImage(img.read(), width, height)
13      opencv_image = cv2.imread("images/imageWithDisclaimer.png", 1)
14      image = {'bytes': opencv_image, 'bbox': bbox, 'size': (width, height)}
15      return image

```

Figura 5.9: Método `retrieve_new_map(lat, lon, radius, width, height)`

```

1  def getBoundingBox(lat, lon, distance):
2      radValues = from_degrees(lat, lon)
3      rad_dist = distance / EARTH_RADIUS_WGS84
4      min_lat = radValues[0] - rad_dist
5      max_lat = radValues[0] + rad_dist
6      if min_lat > MIN_LAT and max_lat < MAX_LAT:
7          delta_lon = asin(sin(rad_dist) / cos(radValues[1]))
8          min_lon = radValues[1] - delta_lon
9          if min_lon < MIN_LON:
10             min_lon += 2 * pi
11             max_lon = radValues[1] + delta_lon
12             if max_lon > MAX_LON:
13                 max_lon -= 2 * pi
14         ...
15         southWestPoint = from_radians(min_lat, min_lon)
16         northEastpoint = from_radians(max_lat, max_lon)
17         return southWestPoint[1], southWestPoint[0], northEastpoint[1], northEastpoint[0]

```

Figura 5.10: Método `getBoundingBox(lat, lon, distance)` encargado de calcular la esquina inferior izquierda y la esquina superior derecha de un cuadrado de ancho `distance`

del GPS y los solicitados al WMS referencian a una proyección geográfica. La más extendida y que utilizamos aquí es la WGS84. Los datos del GPS los recibimos con este sistema de referencia y solicitamos los mapas en base a él. En nuestro caso trabajamos hasta con 3 sistemas de coordenadas distintos: (a) el de las imágenes con $X[0, \text{MAX_WIDTH}]$ creciente de izquierda a derecha e $Y[0, \text{MAX_HEIGHT}]$ creciente de arriba a abajo, (b) el sistema de coordenadas geográficas WGS84 y (c) el cartesiano. Las tres se mueven en órdenes de magnitud diferentes y un movimiento de $1 * 10^{-5}$ puede suponer varias decenas de píxeles. Podemos ver la proyección de posición geográfica a imagen en la Figura 5.11.

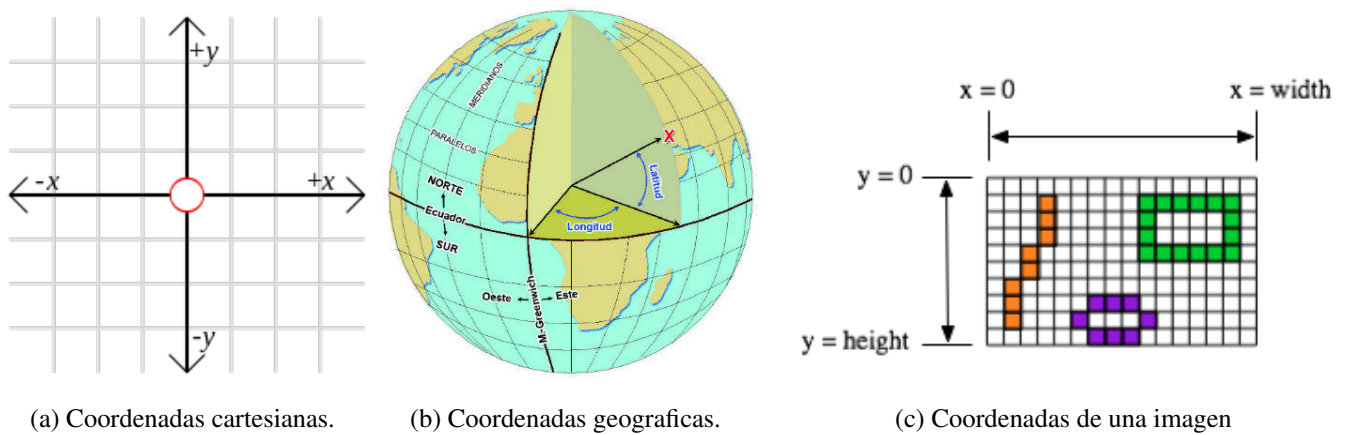


Figura 5.11: Sistemas de referencia utilizados

Para abordar este problema, y dado que se ha trabajado con robots aéreos con una autonomía relativamente reducida, se ha asumido que la tierra es plana en un área rectangular de menos de 2 kilómetros de ancho. Con esta convención conseguimos simplificar notablemente la complejidad del problema, que prácticamente se convierte en una normalización de un sistema a otro. Si en un futuro se decidiese utilizar la aplicación para vuelos de mayor envergadura se debería tener en cuenta la curvatura de la tierra. La proyección en UAV Commander se realiza en ambos sentidos, de imagen a posición geográfica y de sistema de coordenadas geográficas a imagen.

```

1  def posCoords2Image(lonMin, latMin, lonMax, latMax, lat, lon, tamImageX, tamImageY):
2      distCoordX = round(latMax - latMin, 7)
3      distCoordY = round(lonMax - lonMin, 7)
4      x = ((latMax-lat) *(tamImageX))/distCoordX
5      y = ((lon-lonMin) *(tamImageY))/distCoordY
6      return round(y), round(x)

```

Figura 5.12: Método `posCoords2Image(lonMin, latMin, lonMax, latMax, lat, lon, tamImageX, tamImageY)`, encargado de calcular la proyección de un punto en coordenadas geográficas sobre la matriz de píxeles de la imagen que representa el mapa

5.3.3. Visualización de mapas

El mapa se presenta como un elemento más del control de misiones de la aplicación. Para visualizarlo por pantalla se ha creado una capa en PyQt5 con un QLabel que carga la imagen. Para realizar esta tarea de forma ágil y evitar el colapso de los recursos de la máquina donde se ejecuta, trabajamos con la biblioteca de visión artificial OpenCV, con lo que para mostrarla por pantalla hay que realizar ciertas transformaciones. El código encargado de mostrar por pantalla el mapa se puede ver en la Figura 5.12 y la apariencia de un mapa visualizado como ejemplo en la Figura 5.13.

```

1  def refreshImage(self):
2      height, width, channel = self.cvImage.shape
3      bytesPerLine = 3 * width
4      qimg = QImage(self.im_to_show.data, width, height, bytesPerLine, QImage.Format_RGB888)
5      qpm = QPixmap.fromImage(qimg)
6      self.imageLabel.setPixmap(qpm)

```

Figura 5.13: Código encargado de mostrar por pantalla el mapa

5.4. Bloque de misiones

Este bloque es el encargado de facilitar la creación y envío de misiones. Las misiones se definen con una secuencia de puntos de paso (*waypoints*) que se quiere que el dron recorra, incluyendo un punto inicial de despegue y un punto final de aterrizaje. Ha de permitir crear misiones

de forma interactiva además de identificar y convertir en objetos de interfaz ICE `mission` las maniobras de despegue y aterrizaje y los propios puntos de paso.

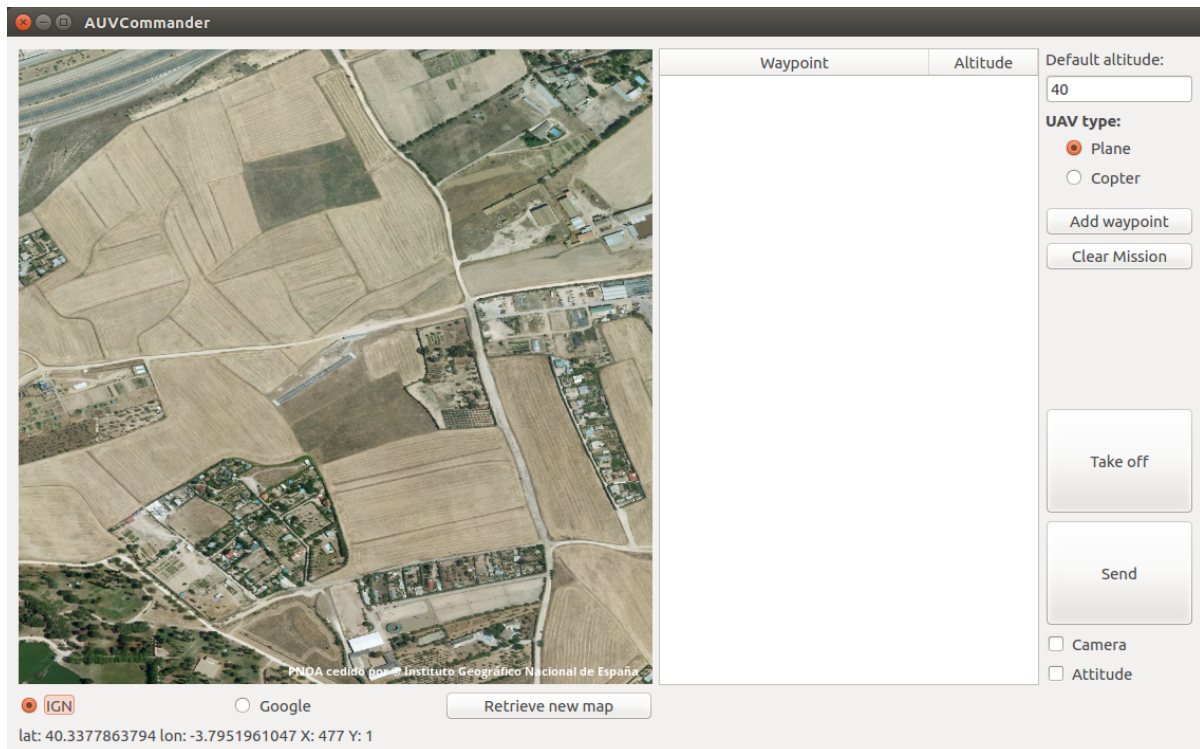


Figura 5.14: Pantalla principal de UAV Commander

5.4.1. Creación y envío de misiones

Para dar esta funcionalidad de una forma intuitiva y sencilla se han implementado varios ingredientes:

- Un mapa interactivo donde un click crea un punto de misión, dibujando un número secuencial que muestra el orden la misión. Además, según van añadiéndose nuevos puntos, la aplicación UAV Commander va dibujando la ruta que “debería” seguir el robot para el cumplimiento de su misión. Un simple click en el mapa desencadena:

1. Inclusión del punto de paso en la lista de puntos de paso.
2. Llamada al método de proyección `posImage2Coords(x, y, sizeX, sizeY, latMin, lonMin, latMax, lonMax)` para obtener a partir del x e y de la imagen la latitud y longitud del punto en el espacio a donde se desea ir.

```

1  def set_waypoints(self, waypoints, current=True):
2      n=0
3      pts = np.array(waypoints, np.int32)
4      pts = pts.reshape((-1, 1, 2))
5      cv2.polylines(self.cvImageShadow, [pts], False, (250, 250, 250), thickness=1)
6      for waypoint in waypoints:
7          n+=1
8          s = str(n)
9          cv2.circle(self.cvImageShadow, (waypoint[0], waypoint[1]), 1, [0, 0, 255],
10                  thickness=-1, lineType=1)
11          cv2.putText(self.cvImageShadow, s, (waypoint[0] + 3, waypoint[1]), cv2.
12                      FONT_HERSHEY_COMPLEX_SMALL, 1, [0, 0, 255], thickness=1)
13      self.refreshImage()

```

Figura 5.15: Método `set_waypoints(self, waypoints, current=True)`, encargado de pintar los puntos de paso en el mapa

3. Control de misión a realizar, verificando si se trata de un despegue, aterrizaje o un punto de paso intermedio.
 4. Inserción en la tabla de seguimiento de puntos de paso.
 5. En la próxima iteración del hilo `threadMap` a través de la llamada a `update_position()` se actualizará el mapa con éste cambio.
 6. Al actualizar el mapa se llama al método `set_waypoints(self, waypoints, current=True)` quien pinta el nuevo punto de paso y la ruta a seguir en el mapa. Las líneas de código de la figura 5.14 muestran la implementación del método encargado de pintar los puntos de paso y las rutas en el mapa `set_waypoints(self, waypoints, current=True)`
- Botón de “Take Off”/“Land”. Este botón apoya al mapa para crear misiones con aterrizaje y despegue, si se quiere despegar hacia un punto, basta con pulsarlo y pulsar después hacia dónde se quiere despegar. El comportamiento para el aterrizaje es análogo.
 - Tabla de seguimiento de puntos de paso. En esta tabla se almacenan los puntos de paso que se van creando al hacer click sobre el mapa, cada fila es editable, de tal forma que ofrece una gran precisión en caso que sea necesario.

```

1  def sendWP(self, send2APM):
2      ...
3      for row in range(colCount):
4          ...
5          pos_lat = text.find("lat:")
6          pos_lon = text.find("lon:")
7          if pos_lat != -1:
8              lat = (text[pos_lat + 4:pos_lon])
9              pose.x = float(lat)
10             lon = (text[pos_lon + 4:])
11             pose.y = float(lon)
12         else:
13             if "LAND in " in text:
14                 ...
15                 self.extra.land()
16             else:
17                 pos = text.find("TAKE OFF to ") + 12
18                 ...
19                 self.extra.takeoff()
20         alt = self.table.item(row, 1)
21         pose.h = int(alt.text())
22         mission.mission.append(pose)
23         if i==0:
24             mission.mission.append(pose)
25         i += 1

```

Figura 5.16: Método `sendWP(self, send2APM)`, encargado de enviar la misión creada por el usuario al driver APM Server

- Campo “default altitude” Establece la altitud de crucero, el valor de este campo se irá propagando en cada punto de paso creado en el atributo `altitude`.
- Botón “Send”. Este botón llama al método `sendWP(self, send2APM)` (Figura 5.15) quien recoge los objetos de misión, los procesa y los envía al bloque de comunicación con drivers a través de los interfaces ICE necesarios, a saber, `mission` y `extra` en el caso en el que haya aterrizaje o despegue en la misión.

5.4.2. Control del vuelo

El control se materializa en la placa estabilizadora a bordo del propio dron, en la aplicación UAV Commander se visualiza su efecto, monitorizando en todo momento la posición 3D del

avión y su relación con los puntos de paso de la misión.

Para llevar ésto a cabo se ha dividido el mapa en tres capas:

1. Mapa original con el disclaimer. `self.cvImage`
2. Mapa original con la estela que ha dejado el robot aéreo. `self.cvImageShadow`, sus posiciones pasadas.
3. Mapa que va a mostrarse, donde se pinta la posición actual del robot aéreo `self.im_to_show`

Este planteamiento por capas permite, por ejemplo, mantener la estela del robot y pintar la posición actual, añadir puntos de paso sin alterar lo ya dibujado, etc. Aunque complica la implementación de todos los métodos que interactúan en este bloque. Algunos de los métodos más importantes del desarrollo se encuentran aquí y se listan a continuación:

- `update_position(self)`. Este método es al que el hilo `treadMap` llama en cada iteración. Es el encargado de desencadenar las llamadas suficientes para actualizar visualmente el mapa en el interfaz de usuario.
- `self.setPosition(self, x, y, angle)`. Este método es el encargado de realizar las modificaciones sobre el mapa y asignar el nuevo mapa al interfaz de usuario.
- `draw_triangle(self, x, y, angle)`. Este método es el encargado de actualizar la posición actual pintando en el mapa un triángulo que muestra la posición y rumbo del avión. El centro del triángulo se sitúa en el píxel que representa la posición geográfica actual, y se gira para que marque el rumbo que sigue el avión.
- `refresh_shadow(self,x,y)`. Este método es el encargado de actualizar el mapa de sombras con las posiciones por dónde ha pasado el avión.
- `set_wayPoints(self.wayPoints)` Este método consulta los puntos de paso pintando éstos sobre el mapa junto con la trayectoria a seguir, si en este momento hay un punto de paso nuevo se añade al mapa con el resto.
- `refreshImage(self)`. Este método asigna el nuevo mapa en el interfaz gráfico de usuario.

Se ha desarrollado un hilo que actualiza periódicamente el estado del mapa `threadMap`. Este hilo da la señal al interfaz de usuario de que se tiene que refrescar el mapa, se actualiza a través del método enlazado a esta señal, el método `update_position(self)`.

Para que se pueda ver bien el proceso de una actualización del mapa vamos a describir qué tareas se realizan en cada ciclo de actualización.

En cada iteración del hilo `threadMap` se valida si el avión se encuentra en una posición del mapa cercana al borde (menor del 10 % del ancho o alto), si es así se descargará un nuevo mapa de un área el doble que la primera. Se obtiene la nueva posición y rumbo para que el método `setPosition(self, x, y, angle)` la actualice en el mapa. Se actualiza el mapa de sombras con un nuevo punto correspondiente a la posición actual y se pintan sobre él los puntos de paso en `set_wayPoints(self.wayPoints)`. La siguiente tarea a realizar es actualizar la posición del avión actual sobre el mapa, para ello se dibuja un triángulo de color rojo con el método `draw_triangle(self, x, y, angle)` que representa la posición y el rumbo del avión. Por último se refresca en el interfaz de usuario el mapa que hemos ido modificando con el método `refreshImage(self)`.

A continuación se muestra esta secuencia de llamadas en forma de árbol para que pueda verse mejor que métodos se llaman en qué momento.

- `update_position(self)`
 - `GeoUtils.is_position_close_border(lat, lon , bbox)`
 - `ImageUtils.posCoords2Image(lonMin, latMin, lonMax, latMax, lat, lon, tamImageX, tamImageY)`
 - `setPosition(self, x, y, angle)`
 - `refresh_shadow(self, x, y)`
 - `set_wayPoints(self.wayPoints)`
 - `draw_triangle(self, x, y, angle)`
 - ◇ `GeoUtils.center_of_triangle(triangle)`
 - ◇ `GeoUtils.change_coordinate_system(triangle, center, True)`
 - ◇ `rotate_polygon(triangleCartesian, angle)`
 - `set_waypoints(self.wayPoints)`

- `self.refreshImage()`

En la Figura 5.16 se muestran las implementaciones de los métodos con más peso de éste bloque.

```

1  def update_position(self):
2
3      if self.pose != None:
4          pose = self.getPose3D().getPose3D()
5          lat = pose.x
6          lon = pose.y
7          bbox = self.imageMetadata["bbox"]
8          self.limit_warning = GeoUtils.is_position_close_border(lat, lon, bbox)
9          if self.limit_warning:
10             self.download_zoomed_map()
11         else:
12             bbox = self.imageMetadata["bbox"]
13             imagesize = self.imageMetadata["size"]
14             x, y = ImageUtils.posCoords2Image(bbox[0], bbox[1], bbox[2], bbox[3], lat, lon
15                 , imagesize[0], imagesize[1])
16             angle = self.sensorsWidget.quatToYaw(pose.q0, pose.q1, pose.q2, pose.q3)
17             self.setPosition(x, y, angle)
18
19     def setPosition(self, x, y, angle):
20         self.refresh_shadow(x, y)
21         self.set_waypoints(self.wayPoints)
22         self.draw_triangle(x, y, angle)
23         self.refreshImage()
24
25     def refresh_shadow(self, x, y):
26         cv2.circle(self.cvImageShadow, (x, y), 1, [255, 255, 102], thickness=-1, lineType=8,
27             shift=0)

```

Figura 5.17: Métodos encargados de actualizar la posición y el recorrido efectuado en el mapa

Capítulo 6

Experimentos

En este capítulo se presentan las pruebas realizadas al sistema que proporcionan su validación experimental y el hardware necesario para la reproducción de las mismas. Estos experimentos han de validar los subobjetivos descritos en el capítulo 2 y se dividen en nueve pruebas repartidas en dos bloques: experimentos en simulación y experimentos con el avión real.

6.1. Experimentos con misiones en simulador

En este apartado se presentan todos los experimentos realizados en simulación uno a uno, a saber:

1. Experimento de integración de todo el software.
2. Experimento de envío de misión con avión en vuelo.
3. Experimento de envío de misión con despegue y varios puntos de paso.
4. Experimento de envío de misión con despegue y aterrizaje.
5. Experimento de autozoom ante la salida del avión del mapa.

6.1.1. Integración de todo el software

El objetivo de este experimento es probar que la integración del sistema y el simulador SITL se ha realizado correctamente. Vamos a verificar que se realiza correctamente la conexión con el

drone simulado y constatar que el driver recoge correctamente los datos sensoriales del drone. Se verifica también que estos datos se envían correctamente desde el driver a la aplicación y que ésta los representa fielmente. Para ello se han seguido los siguientes pasos:

1. Arranque y configuración de SITL.
2. Se carga una misión de prueba *en el simulador* a través de MAVProxy, donde se le ordena al drone seguir una serie de puntos de paso. Como adelantamos en el capítulo 3 el simulador SITL se integra con MAVProxy y ésto es muy útil, con él podemos cargar misiones desde un fichero, o o comenzar o parar una misión directamente sin necesidad de utilizar nuestra aplicación y driver.
3. Se da comienzo a la misión simulada desde MAVProxy.
4. Se arranca el driver APM Server, es en este momento donde validaremos que se ha realizado la conexión con el drone simulado correctamente.
5. Se arranca la aplicación UAV Commander y seguiremos la trayectoria del avión simulado a través del mapa de la aplicación, comprobando que pasa por los puntos de paso que hemos especificado en el simulador. Se comprobará que la ventana de actitud responde correctamente a los giros del drone simulado durante su misión.

Se comprueba que funciona correctamente y se evidencia en el siguiente vídeo <https://www.youtube.com/watch?v=rHTi0buUETg>

6.1.2. Envío de misión con avión en vuelo

El objetivo de este experimento es probar que el envío de misiones al drone simulado desde nuestra aplicación funciona correctamente y validar su cumplimiento. Para ello se realizaron los siguientes pasos:

1. Arranque y configuración de SITL.
2. Se carga una misión de prueba a través de MAVProxy, en este caso a diferencia del caso anterior la misión tan solo consta de un despegue. Cuando el autopiloto comprueba que se han alcanzado todas las misiones enviadas entra en modo RTL. En este modo el drone

da vueltas trazando una circunferencia de un radio establecido por defecto en el *firmware* de forma indefinida permitiéndonos mandarle misiones estando en el aire (Figura 6.1).

3. Se da comienzo a la misión simulada desde MAVProxy.
4. Se arranca el driver APM Server conectando el driver al avión simulado.
5. Se arranca la aplicación UAV Commander conectándose con el driver.
6. Se crea una misión en UAV Commander haciendo click en los puntos del mapa interactivo por donde queramos que pase el avión.
7. Se envía la misión a APM Server a través del botón “Send” de UAV Commander



Figura 6.1: Seguimiento del experimento 6.1.2 dentro de la aplicación UAV Commander

En el experimento se comprueba que el avión ha seguido la ruta indicada desde UAV Commander correctamente. La prueba se evidencia en el vídeo https://www.youtube.com/watch?v=_X8WGSMIxug

v=_X8WGSMIxug

6.1.3. Envío de misión con despegue y varios puntos de paso

El objetivo de este experimento es probar que el envío de misiones incluyendo una maniobra de despegue al avión simulado funciona correctamente y validar su cumplimiento.

Seguimos los mismos pasos que en el experimento anterior salvo por una particularidad, antes de clicar en el primer punto de paso (que será el primer punto al que vuele tras el despegue) pulsaremos el botón de “Take Off”. De este modo estamos indicando al avión simulado «Despega hacia el punto de paso X, Y» donde X e Y serán latitud y longitud de la proyección del píxel de la imagen a coordenadas geográficas.

Una vez realizada la secuencia arriba descrita se comprueba el seguimiento de la misión enviada a través del mapa, validando tanto que realiza la maniobra de despegue correctamente como el seguimiento de los puntos de paso. La prueba se evidencia en el vídeo <https://www.youtube.com/watch?v=0j3QttXIRbg>



Figura 6.2: Seguimiento del experimento 6.1.3 dentro de la aplicación UAV Commander

6.1.4. Envío de misión completa con despegue y aterrizaje

El objetivo de este experimento es probar que el envío de misiones completas con maniobra de despegue y aterrizaje al avión simulado funciona correctamente y validar su cumplimiento.

Seguiremos los mismos pasos que en el experimento anterior salvo porque antes de clicar en el último punto de paso pulsaremos el botón “Land”. De este modo estamos indicando al avión simulado «Aterriza en el punto de paso X, Y» donde X e Y serán latitud y longitud de la proyección del píxel de la imagen a coordenadas geográficas.

Durante el experimento se puede comprobar que hay que ajustar muy bien la altitud del avión en los últimos puntos de paso, para que el avión aterrice en lo más cerca posible del último. Como vemos en la Figura 6.3 el avión necesita mucho terreno para aterrizar aún ajustando la altura en sus últimos puntos de paso. La prueba se evidencia en el vídeo https://www.youtube.com/watch?v=4yf_PmFgR5Y



Figura 6.3: Seguimiento del experimento 6.1.3 dentro de la aplicación UAV Commander

6.1.5. Experimento de autozoom ante la salida del avión del mapa

El objetivo de este experimento es probar que el autozoom de UAV Commander funciona correctamente. Para ello se han realizado los mismos pasos que en el experimento 6.1.1 estableciendo desde MAVProxy un punto de paso que quede claramente fuera del mapa descargado por defecto por UAV Commander.

Una vez realizada la secuencia arriba descrita se comprueba que UAV Commander descarga un nuevo mapa con una perspectiva más amplia para seguir al avión. Un ejemplo de este funcionamiento lo tenemos en la Figura 6.4. La prueba se evidencia en el vídeo <https://www.youtube.com/watch?v=vEcMbJoJ-Qo>



Figura 6.4: Autozoom en UAV Commander

6.2. Interconexión del driver y aplicación con el avión real

En este apartado se presentan todos los experimentos realizados con el avión real, a saber:

1. Experimento de recepción de vídeo.
2. Pruebas de alcance del enlace entre el avión simulado y el ordenador base.
3. Experimento de recepción de información sensorial y seguimiento del avión a través del UAV Commander.
4. Experimento de comportamiento ante pérdidas de señal.

6.2.1. Experimento de recepción de vídeo

El objetivo de este experimento es verificar que el envío del vídeo capturado desde el ordenador de a bordo con `cameraserver` se recibe correctamente en la aplicación UAV Commander.

Para ello se han realizado los siguiente pasos:

1. Arranque del ordenador abordo.
2. Arranque del driver `cameraserver`.
3. Arranque de la aplicación `UAV Commander` donde clicaremos en el *checkbox camera* y validaremos que se recibe correctamente el video transmitido.

Esta prueba valida la recepción por parte de `AUV Commander` del vídeo que en este caso se emite desde otro driver distinto de `APM Server`, tal y como se aprecia en la Figura 6.5. La prueba se evidencia en el vídeo <https://youtu.be/U-y8xstytyY>

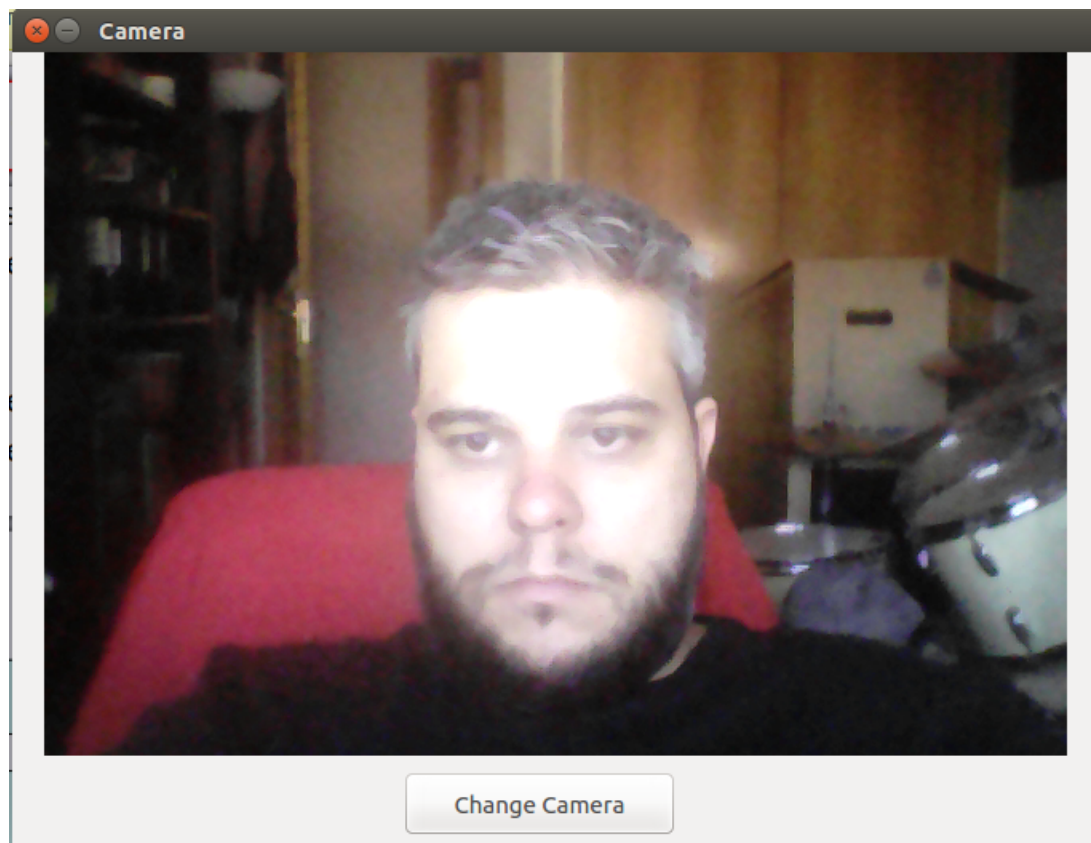


Figura 6.5: Prueba de recepción de vídeo desde `cameraserver` a `UAV Commander`

6.2.2. Recepción de información sensorial y seguimiento del avión a través del `UAV Commander`

El objetivo de este experimento es probar que la integración del sistema y el avión real a radiocontrol se ha realizado correctamente. Se valida que se reciben correctamente los datos

entregados por el driver y que se presentan fielmente a través del interfaz de usuario. Para ello se realizaron los pasos del experimento anterior salvo que en este caso también moveremos el prototipo en sus 3 ejes de libertad para comprobar que se reciben y se presentan correctamente los datos en la ventana de actitud.



Figura 6.6: Seguimiento del experimento 6.2.3 dentro de la aplicación UAV Commander

La prueba se evidencia en el vídeo <https://youtu.be/HFR7JjD-ZIs>

6.2.3. Pruebas de alcance del enlace entre el avión y el ordenador base

El objetivo de este experimento es probar que se reciben correctamente los datos de actitud del avión desde el ordenador base, según nos vamos alejando. El driver (ejecutando abordo del avión real) y aplicación están conectados a través de wifi y por tanto ésto nos delimitará el alcance al que vamos a volar. Para ello se van a realizar los siguiente pasos:

1. Arranque del avión a radiocontrol y del ordenador de abordo.
2. Arranque del driver APM Server, este se conecta nada más arrancar a la placa estabilizadora APM a través del puerto de serie.
3. Arranque de la aplicación UAV Commander quien se conecta a su vez al driver.
4. Se separará el avión del ordenador base hasta que se pierda la señal del wifi. De este modo sabremos la distancia máxima a la que podemos volar

Este experimento nos va a permitir conocer a qué distancia vamos a poder volar. Este experimento se realizó dos veces: en la primera ocasión los datos arrojados por el experimento, apenas 10 metros, impedían cualquier tipo de vuelo del avión real. Con estos datos sobre la mesa se decidió adquirir un adaptador wifi mas potente para el ordenador de abord, y se utilizó un *router* con mas alcance para conectar ambos ordenadores. Con este nuevo hardware en una segunda prueba se alcanzó una distancia cercana a los 20 metros que permitiría un vuelo ceñido. Podemos comparar los resultados de ambos experimentos en la Figura 6.6.



(a) Alcance logrado en la primera prueba de alcance.

(b) Alcance logrado en la segunda prueba de alcance.

Figura 6.7:

Resultados de las pruebas de alcance wifi

6.2.4. Experimento de comportamiento ante pérdidas de señal

El objetivo de este experimento es ver cómo se comporta UAV Commander ante la pérdida de la recepción de APM Server. A tenor de los datos obtenidos en el experimento 6.2.2 este experimento es imprescindible para conocer de antemano qué puede pasar en un vuelo real.

Para llevarlo a cabo se simuló una pérdida de alcance apagando el adaptador wifi del ordenador base donde corre UAV Commander. El resultado esperado era que tras la desconexión no se rompiera la ejecución de APM Server o de UAV Commander y se recuperase tras volver de nuevo a una zona de cobertura. En efecto el resultado obtenido corrobora que aplicación y driver retoman con éxito la conexión después de un tiempo sin estar conectados. El experimento

demuestra la robustez de ambos desarrollos ante pérdidas de señal, la evidencia puede verse en el vídeo <https://youtu.be/HFR7JjD-ZIs>

Capítulo 7

Conclusiones

Este PFC abre las puertas a futuros desarrollos con este tipo de robots aéreos, que hasta ahora JdeRobot no soportaba, sentando las bases del desarrollo de aplicaciones en JdeRobot para aviones autónomos. A lo largo del documento se han descrito los objetivos de este Proyecto Fin de Carrera, la plataforma hardware utilizada y la infraestructura software realizada para abordar el problema descrito en el capítulo dos. Se ha detallado el diseño e implementación de todos los desarrollos implementados en este proyecto, APM Server y UAV Commander, y se han validado con sendos experimentos en el capítulo seis. El objetivo general de dar soporte en JdeRobot a drones tipo avión, incluyendo misiones como secuencia de puntos de paso se ha cumplido satisfactoriamente, al igual que los tres subobjetivos definidos en el capítulo dos.

7.1. Conclusiones

1. El primer subobjetivo del proyecto era el desarrollo de un driver para JdeRobot que nos permitiese conectarnos tanto al avión real como a uno simulado a través de SITL. El driver APM Server (ver capítulo 4) es capaz de conectarse tanto con aviones físicos que utilicen el protocolo de comunicación MAVLink, como con aviones simulados a través de SITL. El driver recupera los datos recogidos de los sensores del hardware de abordo del avión como el GPS y la IMU y los transforma en objetos que cumplen las interfaces ICE ya desarrolladas en JdeRobot Pose3D y NavData.

Para dar soporte a la navegación por posición se ha diseñado un nuevo interfaz, el interfaz `mission`, que junto con el `ExtA` completan los interfaces que este driver recibe

como órdenes y que se envían a través de los comandos MAVLink `MISSION_ITEM`. El driver da soporte en actuación tanto a las maniobras de despegue y aterrizaje como a la navegación por puntos intermedios.

La concurrencia del componente permite que éste pueda responder a las peticiones de otros componentes mientras realiza otras acciones como la obtención de las imágenes del avión o el envío de órdenes de movimiento.

2. El segundo subobjetivo era el desarrollo de una aplicación GCS *Ground Control Station* que permita al operador humano introducir misiones y seguir el cumplimiento de las mismas a través de ella y el acceso a sus datos sensoriales. Para abordarlo se ha desarrollado la aplicación `JdeRobot UAV Commander`, para dar soporte al control de misiones en robots aéreos de ala fija, aviones. Esta aplicación muestra un interfaz gráfico en el cual el usuario humano puede visualizar los datos de los sensores a bordo (cámaras, posición, actitud), el mapa de la zona de trabajo del avión y componer gráficamente misiones de vuelo gracias al mapa interactivo.

`UAV Commander` es capaz de descargar el mapa de la zona del mundo donde el avión se encuentra a través de WMS, y presentarlos por pantalla en forma de mapa interactivo. Clicando sobre el mapa se pueden crear de forma sencilla las misiones que el operador humano desee. Este mapa permite validar el cumplimiento de las misiones en vuelo gracias a elementos visuales añadidos como la ruta teórica a seguir o la estela dejada por el avión durante su vuelo. El control de misiones se complementa con una tabla donde se puede ver la secuencia de puntos de paso que se va a enviar al avión junto a la altura de vuelo. Esta tabla es editable para un ajuste mas exacto, y junto con los botones “TakeOff/Land”, “Clear” y “send” completan el interfaz de creación de misiones.

`UAV Commander` ofrece toda la información sensorial del avión de forma visual en las ventanas de “Camera” y “Attitude”, en ellas veremos los datos enviados por los sensores de abordó del avión como los acelerómetros, giróscopos o las cámaras de abordó. La aplicación hace uso de los interfaces ICE desarrollados en `JdeRobot`, incluyendo el interfaz `mission` desarrollado íntegramente en este proyecto para dar soporte al control de misiones. Ésto permite que sea compatible con el resto de aplicaciones y drivers de `JdeRobot` y le confiere las ventajas de este tipo de desarrollos como la independencia del

lenguaje de programación empleado o del hardware en que se esté ejecutando.

3. El tercer subobjetivo era la validación experimental de ambos tanto en simulación como en el prototipo de avión real construido a tal efecto. Se ha validado el funcionamiento tanto de `APM Server` como de `UAV Commander` a través experimentos diseñados a tal efecto y descritos en el capítulo 6. Los experimentos han sido diseñados para ir validando de menos a más el correcto funcionamiento del driver y la aplicación tanto en un ambiente simulado (experimentos del 6.1.1 al 6.1.5) como en el prototipo de avión MAVLink (experimentos del 6.2.1 al 6.2.3)

En los experimentos con misiones en simulador se prueba de forma progresiva el comportamiento de `APM Server` y `UAV Commander` conectados a SITL. Empezamos comprobando la correcta conexión de `APM Server` a SITL, de `UAV Commander` a `APM Server` y que la aplicación `UAV Commander` presenta correctamente los datos en el experimento 6.1.1 donde queda además probada parte sensorial. En el experimento 6.1.2 se empieza a probar la actuación con el envío de puntos de paso y se le añaden progresivamente las maniobras de despegue y aterrizaje en los experimentos 6.1.3 y 6.1.4 respectivamente. El experimento 6.1.5 valida que `UAV Commander` no pierda el avión cuando éste se acerque al límite del mapa descargado.

En los experimentos de interconexión de driver y aplicación con el avión se comprueba la correcta recepción de toda la información sensorial(experimento 6.2.1 y 6.2.3). Tras probar en el experimento 6.2.2 que el alcance es mas limitado de lo esperado en un inicio, en el experimento 6.2.4 comprobamos la robustez de nuestros desarrollos ante una hipotética pérdida de señal al quedar fuera del alcance del wifi.

El seguimiento y evolución de este PFC, así como el código, vídeos y material utilizado puede encontrarse en la web <http://jderobot.org/Jafernandez>

7.2. Trabajos futuros

Durante el desarrollo de `APM Server` y `UAV Commander` se han identificado varias evoluciones de los desarrollos aquí expuestos que mejorarían las funcionalidades actuales. Las describimos a continuación.

- Se puede ampliar el soporte de misiones de `APM Server` hasta cubrir la totalidad de los comandos de misión de MAVLink. De esta forma se podría por ejemplo enviar misiones del tipo “dar X vueltas de Y radio en torno al punto de paso Z” o “aumenta la velocidad durante la misión a X Km/h”.
- Se pueden ampliar en `APM Server` los interfaces para enviar información sobre la misión, el objetivo que he alcanzado, el que estoy ejecutando, a qué distancia se encuentra del objetivo, etc.
- En `UAV Commander` se pueden mejorar los cálculos de proyección para que sean mas precisos independientemente de la zona del globo donde se encuentre, e independientemente del WSM de origen del mapa.
- En `UAV Commander` se puede ampliar la lista de orígenes de mapas añadiendo los WMS de OpenStreetMap, Yahoo Maps y Bing Maps .
- En `UAV Commander` se puede añadir el control por velocidades `CMDVel` para dar soporte a todo tipo de drones. Y que esta aplicación se pudiera emplear con drones tipo cuadricóptero, capaces de cernirse en un sitio fijo y obedecer órdenes de velocidad en varios ejes.
- Como desarrollo adicional se podría conectar SITL a Gazebo, de ésta forma tendríamos la simulación completa incluyendo el sensor de cámara y se podrían probar las misiones en entornos más realistas. Esto facilitaría la simulación de desarrollos que utilicen la visión en aviones.

Bibliografía

- [1] businessinsider.com. Drones are about to fill the skies within the next 5 years, Jun 2016.
<http://www.businessinsider.com/the-drones-report-research-use-cases-regulations-and-issues-2016-4>.
- [2] J. cano. Mediawiki, 2016.
<http://jderobot.org/J.canoma-tfg>.
- [3] Droneii.com. Drone investment trends 2016, May 2016.
<http://www.droneii.com/drone-investment-trends-2016>.
- [4] A. M. Florido. Mediawiki, 2014.
<http://jderobot.org/Amartinflorido-tfg>.
- [5] D. Jiménez. Mediawiki, 2017.
<http://jderobot.org/Jimenez-tfg>.
- [6] A. Martínez. Mediawiki, 2015.
<http://jderobot.org/Aitormf-tfg>.
- [7] U. of Colorado Boulder. Mavlink protocol: Waypoints, May 2015.
<http://www.colorado.edu/recuv/2015/05/25/mavlink-protocol-waypoints>.
- [8] rtl sdr.com. Spoofing gps locations with low cost tx sdrs, Sep 2015.
<http://www.rtl-sdr.com/spoofing-gps-locations-with-low-cost-tx-sdrs/>.
- [9] J. Vela. Mediawiki, 2017.
<http://jderobot.org/Jvela-tfg>.

- [10] D. Yagüe. Mediawiki, 2014.
<http://jderobot.org/Daniyague-pfc>.