Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Ampliación de la máquina de estados usada en un sistema de navegación autónoma para RPAS

Autor: Javier Moreno Prieto

Tutor: Aníbal Ollero Baturone

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Ampliación de la máquina de estados usada en un sistema de navegación autónoma para RPAS

Autor:
Javier Moreno Prieto

Tutor:
Aníbal Ollero Baturone
Catedrático de Universidad

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado: Ampliación de la máquina de estados usada en un sistema de navegación autónoma para RPAS

Autor: Javier Moreno Prieto
Tutor: Aníbal Ollero Baturone

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

En primer lugar, quiero agradecer a mis padres, por su esmero en mi educación, por apoyarme y aconsejarme en mis decisiones y por confiar en mí.

También, a mis compañeros de la universidad, en especial a Fátima, Irene y Pablo, por acompañarme durante esta etapa de mi vida y ofrecerme su ayuda siempre que la he necesitado.

Y por último y no por ello menos importante, a mis colegas de CATEC, por acogerme a mi llegada y ayudarme en la realización de este proyecto.

Gracias a todos, pues sin vosotros no estaría donde estoy.

Javier Moreno Prieto
Sevilla, 2022

Resumen

La fundación FADA-CATEC desarrolló en 2020 un *framework* para la creación y ejecución de misiones de seguimiento de *waypoints* para autopilotos de DJI. El fin, era poder controlar el dron de forma autónoma en escenarios donde la cobertura GPS no estaba disponible, pues esta es necesaria para poder utilizar las herramientas propias del fabricante que ofrecen este tipo de funcionalidades.

Dotando a la aeronave de un sistema de localización alternativo y gracias al *framework*, se consiguió recuperar la funcionalidad de las misiones. Este, además, ofrece la posibilidad de: controlar el armado de los motores, realizar maniobras de despegue y aterrizaje y alternar entre distintos modos de vuelo (manual, automático y asistido).

El presente Trabajo de Fin de Grado tiene como objetivo añadir nuevas funcionalidades al *framework*, centradas principalmente en:

- Mejorar la seguridad del sistema mediante la detección de fallos y ejecución de rutinas de prevención de accidentes.
- Añadir compatibilidad con el estándar para autopilotos *open-source* Pixhawk, adaptando las comunicaciones y funciones a la arquitectura de este tipo de sistemas.

Se partirá como base del código ya existente, siendo necesario un previo estudio del mismo, con el fin de poder reutilizar las funcionalidades ya implementadas y seguir la línea de diseño ya establecida por los autores originales.

Abstract

In 2020, the FADA-CATEC foundation developed a framework for the creation and execution of waypoint following missions for DJI autopilots. The purpose was to be able to control the drone autonomously in scenarios where GPS coverage was not available, since it is necessary to be use the manufacturer's own tools that offer this type of functionality.

By equipping the aircraft with an alternative location system and thanks to the framework, it was possible to recover the functionality of executing missions. The framework, in addition, offers the possibility of: arming the motors, executing takeoff and landing maneuvers and alternating between different flight modes (manual, automatic and assisted).

The purpose of this project is to add new functionalities to the framework, mainly focused on:

- Improving system security by detecting faults and executing accident prevention routines.
- Adding compatibility with the open-source Pixhawk autopilot standard, adapting communications and functionalities to this new architecture.

It will be based on the existing code, requiring a previous study of it, in order to be able to reuse the functionalities already implemented and follow the design line already established by the original authors.

Índice Abreviado

Resumen III
Abstract V
Índice Abreviado VII

1 Introducción 1
1.1 Introducción 1
1.2 Motivación 3
1.3 Objetivos 4

2 Adaptación para el uso del simulador 5
2.1 DjiSimulatorMainNode 6
2.2 DjiSimulatorMainData 6
2.3 JoystickPublisher 7
2.4 PoseConverter 8
2.5 RefConverter 11
2.6 GCSJoyConverter 12
2.7 UserInterface 15
2.8 Resultado 17

3 Definición e integración de mejoras 19
3.1 Detección de eventos 19
3.2 Nivel de batería crítica 19
3.3 Pérdida de señal RC 20
3.4 Exceso de límites 20
3.5 Gestión de failsafes 22
3.6 Rutinas de seguridad 23
3.7 Return to Home (RTH) 23
3.8 Estado FAILSAFE 24

4 Adaptación a Pixhawk 29
4.1 Cambios en Safety Manager 29
4.2 Cambios en Main State Machine 30
4.3 Cambios en Auto State Machine 31
4.4 Adaptación de la simulación 31

5 Unificación de códigos 33

6 Resultados experimentales 37
6.1 Pruebas de vuelo 41

7 Conclusiones 47

Apéndice A Hardware **49**
A.1 DJI 49
A.2 Pixhawk 50

Apéndice B Software **51**
B.1 ROS 51
B.2 Entornos de desarrollo y comunicación 52
B.3 CATEC GCS 53

Apéndice C Framework preexistente **55**
C.1 Safety Manager 55
C.2 Main State Machine 59
C.3 Auto State Machine 61

Índice de Figuras 65
Índice de Tablas 67
Índice de Códigos 69
Bibliografía 71

Índice

Resumen III
Abstract V
Índice Abreviado VII

1 Introducción 1
1.1 Introducción 1
1.2 Motivación 3
1.3 Objetivos 4

2 Adaptación para el uso del simulador 5
2.1 DjiSimulatorMainNode 6
2.2 DjiSimulatorMainData 6
2.3 JoystickPublisher 7
2.4 PoseConverter 8
2.5 RefConverter 11
2.6 GCSJoyConverter 12
2.7 UserInterface 15
2.8 Resultado 17

3 Definición e integración de mejoras 19
3.1 Detección de eventos 19
3.2 Nivel de batería crítica 19
3.3 Pérdida de señal RC 20
3.4 Exceso de límites 20
3.5 Gestión de failsafes 22
3.6 Rutinas de seguridad 23
3.7 Return to Home (RTH) 23
3.8 Estado FAILSAFE 24

4 Adaptación a Pixhawk 29
4.1 Cambios en Safety Manager 29
4.2 Cambios en Main State Machine 30
4.3 Cambios en Auto State Machine 31
4.4 Adaptación de la simulación 31

5 Unificación de códigos 33

6 Resultados experimentales 37
6.1 Pruebas de vuelo 41
6.1.1 Día 1 41
6.1.2 Día 2 41

6.1.3 Día 3 42

7 Conclusiones 47

Apéndice A Hardware 49
A.1 DJI 49
A.2 Pixhawk 50

Apéndice B Software 51
B.1 ROS 51
B.2 Entornos de desarrollo y comunicación 52
B.2.1 DJI Onboard SDK 52
B.2.2 DJI Onboard SDK ROS 52
B.2.3 Mavlink 52
B.2.4 Mavros 53
B.3 CATEC GCS 53

Apéndice C Framework preexistente 55
C.1 Safety Manager 55
C.1.1 Parámetros 55
C.1.2 Comprobación de topics 56
SubscriberChecker 56
TopicChecker 56
TopicsCheckerManager 57
C.1.3 Máquina de estados 58
C.1.4 Inicializador del nodo 58
C.2 Main State Machine 59
C.2.1 Estados 59
UNINITIALIZED 59
MANUAL 59
STOPPED 59
ARMED 59
TAKINGOFF 60
AUTO 60
ASSISTED 60
OFFBOARD 60
LANDING 60
C.2.2 Servicios 60
C.2.3 Topics 61
C.3 Auto State Machine 61
C.3.1 Estados 62
IDLE 62
HOVERING 62
GOTOWP 62
PRELANDING 62
C.3.2 Servicios 63
C.3.3 Topics 64
C.3.4 Concepto de *home* y transformación de coordenadas 64

Índice de Figuras 65
Índice de Tablas 67
Índice de Códigos 69
Bibliografía 71

1 Introducción

Este capítulo actuará como breve introducción al proyecto *Ampliación de la máquina de estados usada en un sistema de navegación autónoma para RPAS*. A continuación, se realizará una presentación del campo de investigación en el que se enmarca el proyecto, se justificará la motivación tras el desarrollo del mismo y se establecerán los objetivos a cumplir.

En los capítulos posteriores se explicará el trabajo realizado, mostrando el razonamiento tras las diferentes decisiones de diseño que se han tomado para implementar las nuevas funcionalidades y las adaptaciones necesarias para añadir la compatibilidad con Pixhawk; para acabar exponiendo los resultados obtenidos en pruebas de vuelo.

Adicionalmente, en el apéndice, se incluirá una explicación del hardware y software utilizados, además de un análisis en profundidad del *framework* preexistente, que será de gran importancia para la comprensión de la ampliación realizada.

1.1 Introducción

Se conoce como RPAS (*Remotely Piloted Aircraft System*) al conjunto de elementos configurables integrado por una aeronave pilotada a distancia, también conocida como UAV (*Unmanned Aerial Vehicle*) o de forma más común, dron[1], sus estaciones de piloto remoto conexas, los necesarios enlaces de mando y control y cualquier otro elemento de sistema que pueda requerirse en cualquier punto durante la operación de vuelo [2].

En los últimos años, la popularidad de estos sistemas se ha incrementado considerablemente debido al avance de la tecnología y la reducción de costes, que ha aumentado la accesibilidad tanto para particulares como empresas. En España, por ejemplo, la demanda de operaciones con drones aumentó un 370 % entre los años 2020 y 2021 según los datos de Enaire[3].

A pesar del origen militar de este tipo de sistemas, en la actualidad, los drones son empleados para múltiples aplicaciones tanto civiles como industriales, como puede ser la inspección de edificios, el envío de paquetes, la grabación de vídeo, el recuento de inventario en almacenes, fotogrametría...

Para cumplir este tipo de tareas, se pueden pilotar las aeronaves de forma manual, es decir, con un piloto que mediante un mando radio control controle el movimiento, o, como alternativa, muchos fabricantes ofrecen la posibilidad de programar lo que se conoce como misiones, en las cuales se pueden especificar múltiples puntos de rutas o más conocidos como *waypoints*, de forma que el dron calcule la ruta y navegue hasta ellos de forma autónoma, desplazándose a la velocidad indicada e incluso, en algunos modelos, siendo capaz de evitar obstáculos tanto estáticos como móviles que detecte, recalculando la ruta sobre la marcha. Cada uno de los *waypoints* se define respecto a un sistema de coordenadas geográficas, que se compone de latitud, longitud y altitud, esta última pudiendo ser respecto al punto de despegue, al elipsoide, el geoide o el terreno. Gracias a esta herramienta se puede hacer un uso sencillo y seguro del vehículo por parte de los operarios, siendo este solamente necesario para diseñar el plan de vuelo y de estar en alerta para controlar la aeronave de forma manual ante algún inconveniente.

Figura 1.1 Visualización de *waypoints* en la app de DJI.

Sin embargo, la navegación autónoma presenta la desventaja de que, para poder realizar el seguimiento de los *waypoints*, la aeronave debe conocer su posición en un sistema de coordenadas geográfico. Depende, por tanto, de la señal GPS, pues es aquella que ofrece esta información. Esta señal, puede no estar disponible si la aplicación tiene lugar en entornos cerrados o con cobertura pobre, como en bosques frondosos o entre edificios, inhabilitando por tanto la ejecución de las misiones.

Así pues, se hace necesario implementar un método alternativo de localización. El uso de los sensores LiDAR se ha convertido en la solución más robusta y popular. Este tipo de sensores mide las distancias hasta los objetos de su entorno mediante pulsos de haces de luz, proporcionando una nube de puntos que siendo procesada mediante métodos de SLAM (*Simultaneous Localization And Mapping*) permite obtener un mapeado tridimensional del espacio y la estimación de la posición en un sistema de referencia local, donde se trabaja con coordenadas cartesianos en vez de geográficas. Las técnicas de SLAM también pueden ser realizadas mediante las imágenes obtenidas por cámaras de vídeo a bordo del dron, teniendo como desventaja ser dependiente de las condiciones de iluminación. Esta metodología resuelve el problema de localización, aunque con el inconveniente de que las herramientas de ejecución de misiones ofrecidas por los fabricantes de los UAVs únicamente trabajan con coordenadas geográficas.

Figura 1.3 DJI Matrice 600 equipado con LiDAR Velodyne VLP 16.

Figura 1.2 Dron para inspección de superficies por contacto, proyecto AEROARMS H2020.

1.2 Motivación

Figura 1.4 CATEC.

El Centro Avanzado de Tecnologías Aeroespaciales (CATEC) es un centro tecnológico establecido y gestionado por la Fundación Andaluza para el Desarrollo Aeroespacial (FADA) [4]. CATEC participa en múltiples proyectos de I+D+i tanto a nivel nacional como europeo, gran parte de ellos relacionados con el uso de RPAS, como el que se puede ver en la Figura 1.2.

En el año 2020, desarrolló un *framework* de navegación para multirrotores del fabricante DJI con el objetivo de poder recuperar la funcionalidad de programar misiones en situaciones donde la señal GPS no estaba disponible, pero se podía obtener la localización por algunos de los métodos alternativos anteriormente mencionados, pues algunos proyectos requerían realizar vuelos en interiores. Este software se pudo desarrollar gracias a las herramientas para desarrolladores proporcionadas por DJI y se compone de tres máquinas de estados, que permiten alternar quién posee el control del dron, gestionar los estados de vuelo y la creación y ejecución de misiones.

El *framework* es funcional y se ha implementado con éxito en diversos proyectos de CATEC, sin embargo, había planes de continuar el desarrollo del sistema, añadiendo nuevas funcionalidades, centradas sobre todo en la seguridad, y realizando una adaptación para ampliar su uso a autopilotos basados en Pixhawk.

Por tanto, el objetivo de este Trabajo de Fin de Grado será llevar a cabo dichos planes durante el periodo en el que el autor se encuentra realizando sus prácticas de empresa en la fundación.

1.3 Objetivos

Antes de comenzar, se definirán una serie de objetivos a cumplir para el proyecto:

- Estudio y comprensión del código ya existente: previamente al inicio del desarrollo, es de vital importancia estar familiarizado con el código ya existente para poder entender su estructura y adaptar los futuros añadidos de la mejor forma posible.
- Implementación de nuevas funcionalidades: en primer lugar habrá que decidir qué mejoras se van a incluir y posteriormente programar el código.
- Adaptación para autopilotos Pixhawk: será necesario primero documentarse sobre el funcionamiento de este tipo de sistemas y luego analizar qué partes del código son reutilizables y cuáles necesitan ser adaptadas.
- Creación de *framework* único para ambos entornos: una vez el sistema sea compatible con ambos autopilotos, se unificarán los dos códigos para obtener un único *framework* que pueda ser usado en los dos.
- Realización de tests para comprobar el correcto funcionamiento del código desarrollado: se diseñarán planes de vuelo para verificar que las mejoras se han implementado con éxito.

2 Adaptación para el uso del simulador

Las distintas máquinas de estados del *framework* requieren información procedente de varios topics para su funcionamiento, algunos de los cuales son publicados por el autopiloto y otros, por el control implementado en MATLAB. Esto implica que, para poder probar las mejoras, habría que realizar un vuelo. CATEC dispone de un banco de pruebas para realizar vuelos en interior, sin embargo, existe una gran demanda para su uso, además de ser necesaria la presencia de un piloto por motivos de seguridad, por lo que sería difícil probar las nuevas funcionalidades conforme se van añadiendo.

Como alternativa, se decide hacer uso del programa DJI Assistant 2. Este software es proporcionado por DJI con el fin de poder configurar distintos parámetros de la aeronave antes del vuelo, y además, dispone de un simulador de vuelo con el que probar la configuración y realizar prácticas con el mando. Con este simulador, se pretende poder probar fácilmente los añadidos realizados al *framework*.

Para la instalación del software se requiere un PC con Windows, que irá conectado al autopiloto A3, y este a su vez, al ordenador con Ubuntu, que contiene el Onboard SDK, como se puede observar en la Figura 2.1.

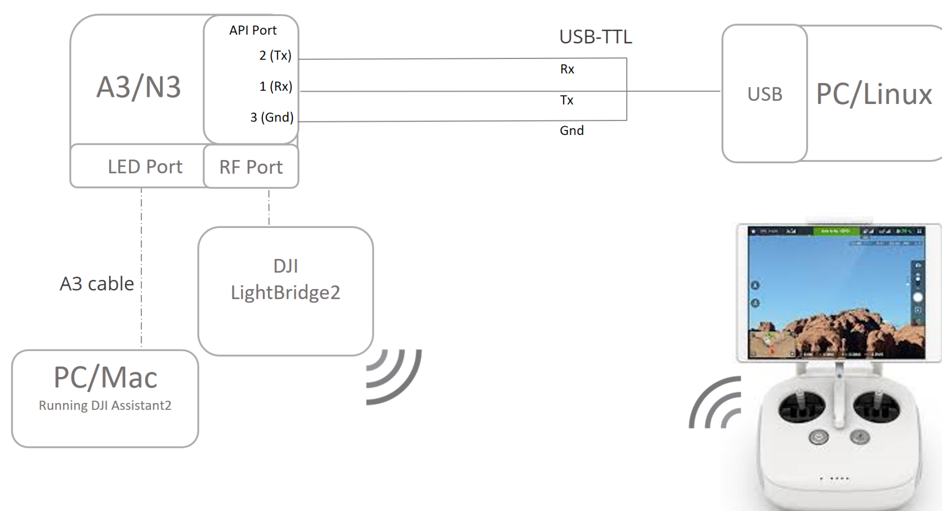

Figura 2.1 Conexionado del simulador.

Una vez realizado el conexionado, se comprueba que el DJI Assistant reconoce el autopiloto y se puede acceder a la configuración de este.

Después, desde Ubuntu, se debe configurar el archivo dji_sdk.launch, especificando el puerto USB usado para conectarse al autopiloto A3 y la tasa de baudios de la conexión. Una vez actualizado, se ejecuta el *launch* a través de la terminal con el comando *roslaunch* y se comprueba que el nodo "dji_sdk" se encuentra en ejecución, y se pone a prueba su correcto funcionamiento haciendo *echo* de los topics y probando la llamada a servicios de armado/desarmado y despegue/aterrizaje, observando que en el simulador el dron realiza estas acciones.

Con el simulador ya en funcionamiento, todavía no es posible la ejecución del framework de navegación por diversos motivos:

- El topic "/uav_main_control/pose" no es publicado por nadie, pues en principio no tenemos forma de conocer la posición del dron.
- No se dispone del control de MATLAB, por lo que las referencias de posición publicadas en "/uav_-main_control/auto_control_ref" no tienen efecto alguno.
- Se está trabajando sin un radio control, por lo que no se podrá controlar la autoridad.
- Faltan algunos topics necesarios.

Se decide crear el paquete "dji_framework_simulator", con el fin de solventar estos problemas, además de ofrecer una interfaz de usuario para poder realizar algunas acciones básicas. El código se programará en Python y se estará estructurado en diferentes clases, cada una de ellas con una función específica que se detallará a continuación. En la Figura 2.2 se puede observar un diagrama con las clases creadas.

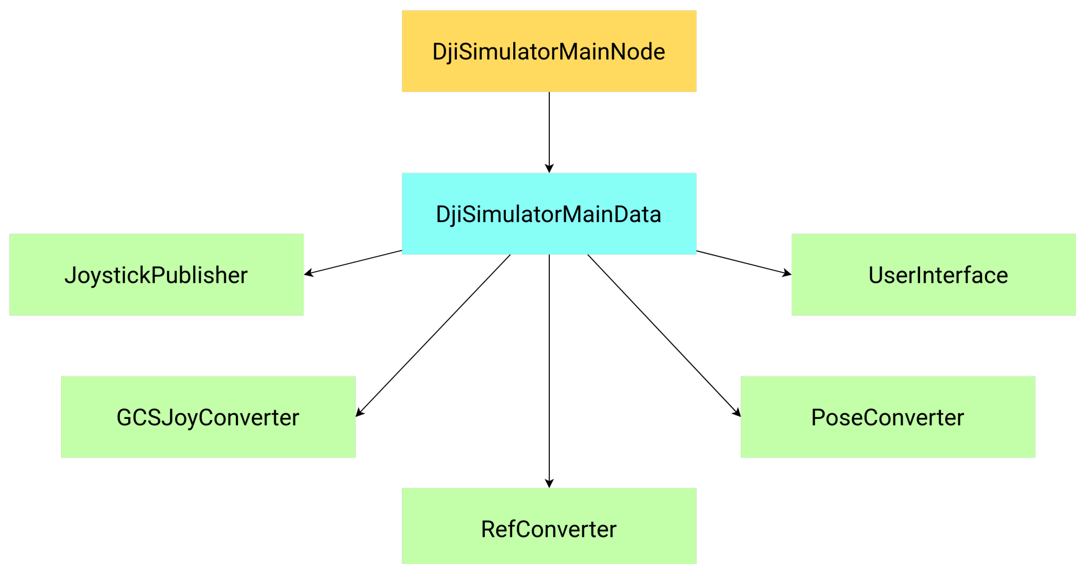

Figura 2.2 Clases presentes en dji_framework_simulator.

2.1 DjiSimulatorMainNode

Esta clase inicializará el nodo de ROS y creará una instancia de **DjiSimulatorMainData**.

2.2 DjiSimulatorMainData

En esta clase se creará una instancia de varias clases (ver Figura 2.2) y después, se procederá a la creación de la interfaz de usuario (GUI). Para esta última tarea, se ha utilizado la librería PyQt5, una adaptación a Python de la popular biblioteca Qt de C++ para la creación de GUIs. Se crea una interfaz sencilla, compuesta por una única ventana con varios botones:

- **MANUAL/AUTO**: permite cambiar la autoridad del dron. Llama al método "automode_change" de **JoystickPublisher** para ello.
- **Take off**: solicita un despegue. Llama al método "takeoff_client" de **UserInterface**.
- **Land**: solicita un aterrizaje. Llama al método "land_client" de **UserInterface**.
- **Preland**: cambia a modo Preland de Auto State Machine. Llama al método "preland_client" de **UserInterface**.
- **Set home**: actualiza el home de Auto State Machine con la posición actual. Llama al método "set_-home_tf" de **UserInterface**.
- **Arm/Disarm**: arma o desarma el dron. Llama al método "arm" de **UserInterface**.

- **Set home simulator**: establece en el simulador que la posición actual del dron es el origen de coordenadas. Llama al método propio "__set_home", donde se llama al método "set_home_simulator" de **UserInterface** y además se actualiza el cuaternión relativo (se explicará más adelante).

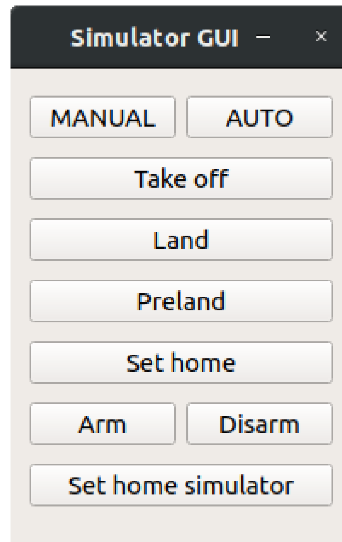

Figura 2.3 Interfaz de usuario.

Adicionalmente, para una mejor experiencia de usuario, leyendo la información de los topics "dji_-sdk/flight_status" y "dji_safety_manager/status", se podrá conocer el estado del dron con el fin de poder activar o desactivar ciertos botones. Por ejemplo, cuando la aeronave se encuentre en el aire, el botón de despegue estará inhabilitado, pues carece de sentido en ese momento. También se colorearán de verde los botones MANUAL y AUTO en función del propietario de la autoridad.

2.3 JoystickPublisher

Durante las primeras fases del proyecto, no se tuvo a disposición un mando radio control. Mediante este dispositivo se realizan los cambios de autoridad, por lo que sin él, no se podrá establecer el modo SERIAL en el autopiloto y por tanto, comandar acciones al dron a través del SDK.

Surge la necesidad entonces de simular de algún modo la señal emitida por el RC. Accediendo a la documentación del "dji_sdk'de ROS Wiki [5], se puede observar que la información recibida del mando es publicada en el topic "/dji_sdk/rc", en un mensaje de tipo Joy. Este mensaje se compone de dos vectores: "axes" y "buttons". Para los cambios de autoridad, se tienen en cuenta axes[4] y axes[5], por tanto, si se publicase en este topic alternando los valores de estas posiciones entre *p_mode_selector_max/min* y *set_authority_max/min*, se podría conseguir emular los flancos de subida y bajada necesarios.

Esta clase consistirá entonces en un publicador a "dji_sdk/rc". Empezará publicando un mensaje que equivaldría a modo MANUAL, y mediante el método "automode_change" se podrá alternar entre MANUAL y CONTROL. Este método es el llamado por la GUI al presionar los botones MANUAL y AUTO.

Una vez terminada la clase, existe otro problema a resolver: a pesar de que no hay un mando conectado, en "/dji_sdk/rc" el autopiloto está publicando mensajes con todos los valores a cero. Si se lanzara nuestro nodo, habría dos publicadores sobre el topic, lo que causaría conflictos. La solución empleada consiste en modificar el archivo "dji_sdk.launch", remapeando dicho topic, es decir, cambiando la cola donde se envían los mensajes que iban destinados a "/dji_sdk/rc".

```
class JoystickPublisher:
    def __init__(self):
        rospy.loginfo("[JOYSTICK] Initiating...")
        self.__rc_pub = rospy.Publisher('dji_sdk/rc', Joy, queue_size=10)
        self.__rc_connection_pub = rospy.Publisher('dji_sdk/
            rc_connection_status', UInt8Stamped, queue_size=10)

        self.__automode = False
        self.__rate = rospy.Rate(50) # 10hz
        self.__publish_data_timer = rospy.Timer(rospy.Duration(1/50.), self.
            __publish_data_cb)

        self.__p_mode_selector_max = rospy.get_param('dji_safety_manager/
            p_mode_selector_max')
        self.__p_mode_selector_min = rospy.get_param('dji_safety_manager/
            p_mode_selector_min')
        self.__set_authority_min = rospy.get_param('dji_safety_manager/
            set_authority_min')
        self.__set_authority_max = rospy.get_param('dji_safety_manager/
            set_authority_max')

    def automode_change(self, answer):
        if answer == 1:
            rospy.loginfo("[JOYSTICK] Mode set to CONTROL")
            self.__automode = True
        else:
            rospy.loginfo("[JOYSTICK] Mode set to MANUAL")
            self.__automode = False

    def __publish_data_cb(self, event):
        rc_msg        = Joy()
        rc_status_msg = UInt8Stamped()

        if self.__automode:
            rc_msg.axes = (0.0, 0.0, 0.0, 0.0, self.__p_mode_selector_max, self.
                __set_authority_max)
        else:
            rc_msg.axes = (0.0, 0.0, 0.0, 0.0, self.__p_mode_selector_min, self.
                __set_authority_min)

        rc_status_msg.data = 1

        self.__rc_pub.publish(rc_msg)
        self.__rc_connection_pub.publish(rc_status_msg)
        self.__rate.sleep()
```

Código 2.1 Código de la clase JoystickPublisher.

2.4 PoseConverter

Conocer la posición de la aeronave es vital para la ejecución de las máquinas de estados, pues sin esta, es imposible realizar las tareas de seguimiento de *waypoints*. Ante la falta de disponer de esta información por parte de algún sensor, se necesita encontrar una alternativa de localización.

En la documentación, se descubre la existencia de dos topics que de los que se puede obtener información de la posición:

- **/dji_sdk/local_position**: publica un mensaje tipo PointStamped con la posición xyz del dron en un sistema de referencia ENU (East, North, Up), siendo el origen el del punto inicial al empezar la simulación, o aquel en el que se encuentre el UAV al llamar al servicio "/dji_sdk/set_local_pos_ref".
- **/dji_sdk/attitude**: publica un mensaje tipo QuaternionStamped con el cuaternión de la rotación entre el sistema de referencia FLU (Front, Left, Up) y ENU.

El topic de posición necesario para el framework es "/uav_main_control/pose", en el cual se deberá publicar un mensaje de tipo PoseStamped, que contiene tanto la posición xyz como el cuaternión de la aeronave. Por tanto, se puede crear este mensaje como composición de los dos mencionados anteriormente.

La clase **PoseConverter** se suscribirá a los topics de posición con el fin de crear un mensaje PoseStamped extrayendo la información necesaria de PointStamped y QuaternionStamped.

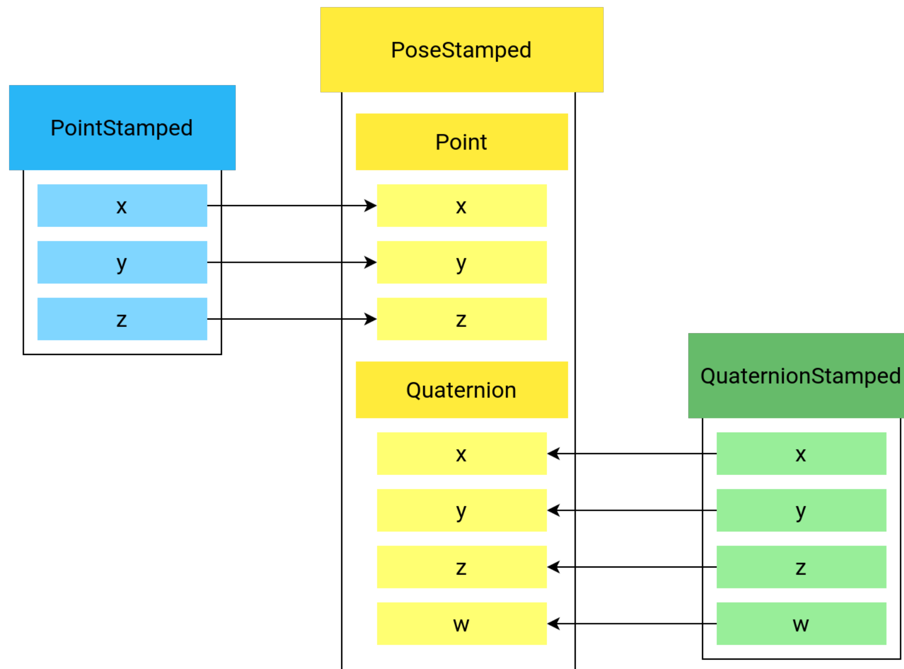

Figura 2.4 Composición de mensaje PoseStamped con PointStamped y QuaternionStamped.

Sin embargo, el proceso no copia directamente los valores leídos. La información de posición publicada por el autopiloto está referenciada a un sistema de coordenadas ENU, donde el eje x apunta al este, el eje y al norte y el z hacia arriba. La máquina de estados trabaja con un sistema de coordenadas FLU, donde el eje x apunta hacia la parte frontal de la aeronave, el eje y hacia la izquierda y el z hacia arriba. Por tanto, habrá que realizar una transformación de ENU a FLU para poder trabajar correctamente.

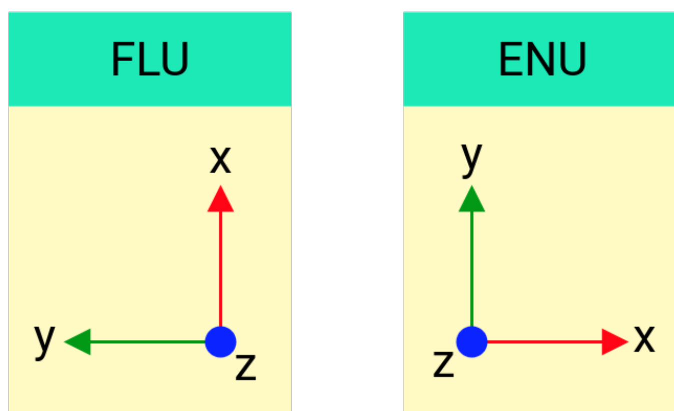

Figura 2.5 Sistemas de referencia FLU y ENU.

La conversión será entonces:

$$x_{FLU} = y_{ENU} \tag{2.1}$$

$$y_{FLU} = -x_{ENU} \tag{2.2}$$

$$z_{FLU} = z_{ENU} + offset \tag{2.3}$$

Adicionalmente, se ha añadido un offset al valor del eje z, pues en el simulador, el origen de coordenadas se sitúa en el punto más bajo del dron, que corresponde con el patín de aterrizaje, y en un vuelo real, el origen se encontraría en el cuerpo del dron, por lo que con este parámetro se puede conseguir una aproximación más fiel.

La orientación proporcionada por el autopiloto tampoco puede ser usada directamente, puesto que lo que representa es el ángulo entre el sistema de referencia FLU solidario a la aeronave y el ENU del mundo. Es decir, cuando este ángulo vale 0, el dron esta orientado con la parte frontal apuntando en el sentido negativo del eje y del sistema FLU.

Habrá que realizar entonces un cálculo para conseguir que el ángulo sea 0 cuando el dron esté alineado con el eje x FLU, o lo que es lo mismo, que el cuaternión sea:

$$q_{init} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.4}$$

Si se toma el primer mensaje recibido de la orientación como referencia ($\mathrm{q}_{\mathrm{ENU}}$), asumiendo que ese valor se corresponde con $\mathrm{q}_{\mathrm{init}}$, se puede calcular el giro entre ambos cuaterniones como:

$$q_{giro} = q_{ENU} \cdot q_{init}^{-1} \tag{2.5}$$

Si ahora, se aplica ese giro a los futuros mensajes recibidos, se podrá obtener el valor de la rotación en el sistema de coordenadas FLU:

$$q_{FLU} = q_{giro} \cdot q_{ENU} \tag{2.6}$$

Teniendo ya adecuada la información de la posición, el siguiente paso es crear el mensaje PoseStamped con los valores calculados y publicarlo en “/uav_main_control/pose”.

Una última función de **PoseStamped** es publicar también en el topic “/lightware_range” para simular el altímetro láser. Para ello, se crea un mensaje de tipo Range, al que se le asigna como valor medido el valor del eje z más un offset, que simula la altura a la que se encontraría el láser en un dron real.

```python
def __uav_pose_cb(self, point, quat):
    if self.__obtained_atti_offset is False:
        self.__quat_relative = self.__obtain_relative_rotation_quat(quat.
            quaternion, self.__quat_init)
        (roll, pitch, yaw) = euler_from_quaternion(self.__quat_relative)
        print("[POSE CONVERTER] Initial relative rotation [deg]: r:{}, p:{}, y
            :{}".format(math.degrees(roll), math.degrees(pitch), math.degrees(
            yaw)))
        self.__obtained_atti_offset = True
        return

    if (self.__quat_relative is None):
        return

    # ENU to FLU
    self.__pose.pose.position.x = point.point.y
    self.__pose.pose.position.y = -point.point.x
    self.__pose.pose.position.z = point.point.z + self.__odom_offset
```

```
    self.__pose.header = point.header
    self.__pose.header.frame_id = "/odom_r"

    self.__pose.pose.orientation = self.__apply_relative_rotation(quat.
        quaternion)

    self.__lightware.header.stamp = point.header.stamp
    self.__lightware.range = point.point.z + self.__lightware_height

    self.__publish_data()
```

Código 2.2 Extracto de la clase PoseConverter.

2.5 RefConverter

La máquina de estados publica la referencia de control en “/uav_main_control/auto_control_ref”, la cual sería leída por el control de MATLAB, que comandaría el movimiento al dron para alcanzarla.

Como el control no está disponible en la simulación, esa referencia no es gestionada, por lo que la aeronave no recibe ninguna orden. Con esta clase, se pretende hacer de puente para poder mandar al autopiloto las instrucciones necesarias para llegar al objetivo.

En la documentación se puede ver que el SDK se suscribe a un topic llamado “/dji_sdk/flight_control_-setpoint_ENUposition_yaw”, donde se puede comandar un offset de posición X e Y, una altura Z y un ángulo de guiñada (rotación sobre el eje vertical) en el sistema de referencia ENU para que el UAV se desplace hasta allí. Mediante este topic, se puede por tanto mover la aeronave a la posición deseada publicada en “/uav_main_control/auto_control_ref”, siendo necesario un previo procesamiento de la información.

De igual manera que en **PoseConverter** se realizaba la transformación de ENU a FLU, ahora será necesaria la transformación de la referencia publicada en FLU a ENU para que sea interpretada adecuadamente por el autopiloto.

Para la posición xyz, se puede despejar fácilmente de las ecuaciones 2.1, 2.2 y 2.3:

$$x_{ENU} = -y_{FLU} \tag{2.7}$$

$$y_{ENU} = x_{FLU} \tag{2.8}$$

$$z_{ENU} = z_{FLU} - offset \tag{2.9}$$

Sin embargo, esto no es suficiente para adaptarlo, puesto que como anteriormente se ha comentado, en la documentación se especifica que en X e Y hay que indicar el offset de posición, es decir, no se indica directamente el punto X e Y al que se quiere ir, si no que desplazamiento que se quiere realizar en cada uno de estos ejes. Por tanto, a la posición de referencia habrá que restarle la posición actual, para obtener así el desplazamiento deseado:

$$x_{ENU} = -y_{FLU} - x_{ENU}^{actual} \tag{2.10}$$

$$y_{ENU} = x_{FLU} - y_{ENU}^{actual} \tag{2.11}$$

Para la orientación, habrá que realizar la transformación inversa a la realizada en **PoseConverter**. Invirtiendo q_{giro}, se puede realizar la rotación como:

$$q_{ENU} = q_{giro}^{-1} \cdot q_{FLU} \tag{2.12}$$

Este cuaternión deberá ser luego convertido a ángulos de Euler para poder obtener la rotación del vehículo en el eje verical, conocida como guiñada.

Una vez transformada la referencia, se crea un mensaje de tipo Joy, donde en el vector "axes", habrá que añadir lo valores de los offsets de X e Y, la altura Z y la guiñada, y luego, publicarlo a "/dji_sdk/flight_-control_setpoint_ENUposition_yaw".

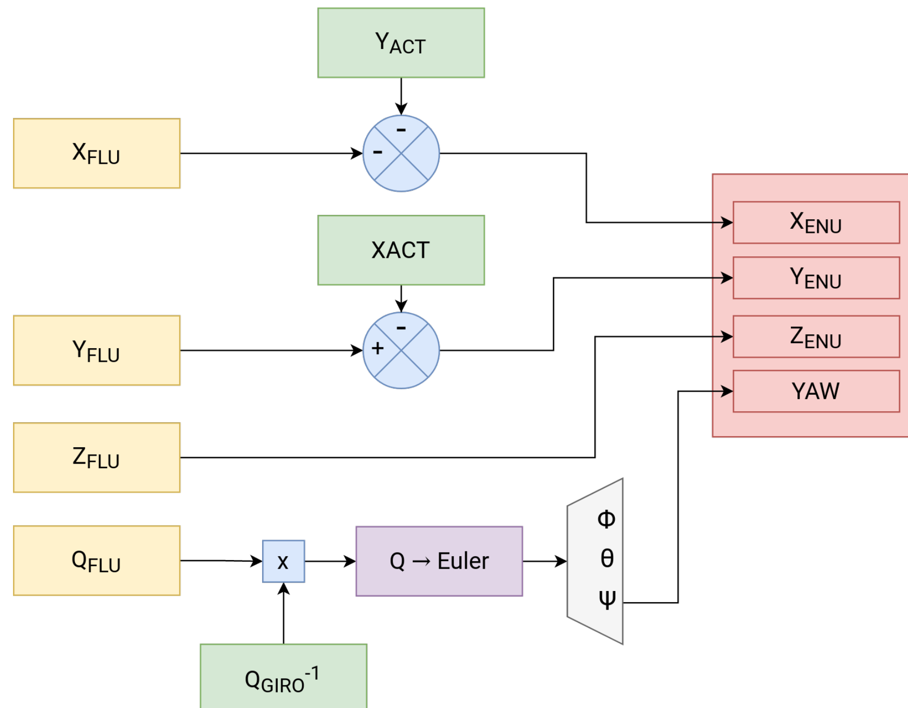

Figura 2.6 Adaptación de la referencia de control.

```
def __ref_cb(self, msg):
    # FLU to ENU
    self.__xref = -(msg.pose.position.y + self.__x_act)
    self.__yref = msg.pose.position.x - self.__y_act

    self.__zref = msg.pose.position.z - self.__odom_offset

    self.__quat_rotated = self.__apply_relative_rotation(msg.pose.orientation)

    self.__yaw = self.__yaw_from_quaternion(self.__quat_rotated.x, self.
        __quat_rotated.y,
                                      self.__quat_rotated.z, self.__quat_rotated.
                                          w)

    self.__joy.axes = [self.__xref, self.__yref, self.__zref, self.__yaw]
    self.__publish_data(self.__joy)
```

Código 2.3 Extracto de la clase RefConverter.

2.6 GCSJoyConverter

En el modo de control ASSISTED, la aeronave puede ser controlada de una forma semi-automática mediante un joystick. La GCS posee un widget con un joystick para el desplazamiento en X e Y, y varios botones para controlar la altura y la guiñada. La posición de estas entradas se publica en el topic "/uav_main_-control/assisted_control_ref" en un mensaje TwistStamped. Este tipo de mensaje contiene información sobre velocidad tanto lineal como angular, que servirá como referencia para el dron.

El control de MATLAB es el encargado de leer este topic y comandar las velocidades al UAV, pero nuevamente, será necesario un módulo intermedio que sustituya al control para la simulación.

Mediante el topic "/dji_sdk/flight_control_setpoint_generic" se puede comandar órdenes de movimiento de múltiples formas dependiendo del flag que se envíe, pudiendo configurar independientemente el tipo de control en el eje horizontal, en el eje vertical y en la guiñada, y el sistema de referencia. Esta configuración del flag puede ser consultada en la Figura 2.7, que se encuentra disponible en la documentación del SDK.

Horizontal	**description**	**reference**	**limit**
0x00	Command roll and pitch angle	Ground/Body	0.611 rad (35 degree)
0x40	Command horizontal velocities	Ground/Body	30 m/s
0x80	Command position offsets	Ground/Body	N/A
0xC0	Command angular rates	Ground/Body	5⁄6π rad/s
Vertical	**description**	**reference**	**limit**
0x00	Command the vertical speed	Ground	-5 to 5 m/s
0x10	Command altitude	Ground	0 to 120 m
0x20	Command thrust	Body	0% to 100%
Yaw	**description**	**reference**	**limit**
0x00	Command yaw angle	Ground	-π to π
0x08	Command yaw rate	Ground	5⁄6π rad/s
Coordinate	**description**		
0x00	Horizontal command is ground_ENU frame		
0x02	Horizontal command is body_FLU frame		
Active Break	**description**		
0x00	No active break		
0x01	Actively break to hold position after stop sending setpoint		

Figura 2.7 Flags para control.

Para este caso, el objetivo es enviar referencias de velocidad, tanto en angular como lineal, por lo que el flag será:

Tabla 2.1 Flags usadas.

Parámetro	**Flag**	**Descripción**
Horizontal	0x40	Velocidad horizontal(m/s)
Vertical	0x00	Velocidad horizontal(m/s)
Yaw	0x08	Velocidad angular(rad/s)
Coordinate	0x02	Sistema FLU solidario a la aeronave

Una vez configurado el flag, faltaría extraer del mensaje Twist recibido los valores de velocidad lineal y la velocidad angular en el eje z y copiar estos valores en la primeras posiciones del vector "axes" perteneciente al mensaje Joy que se va a publicar.

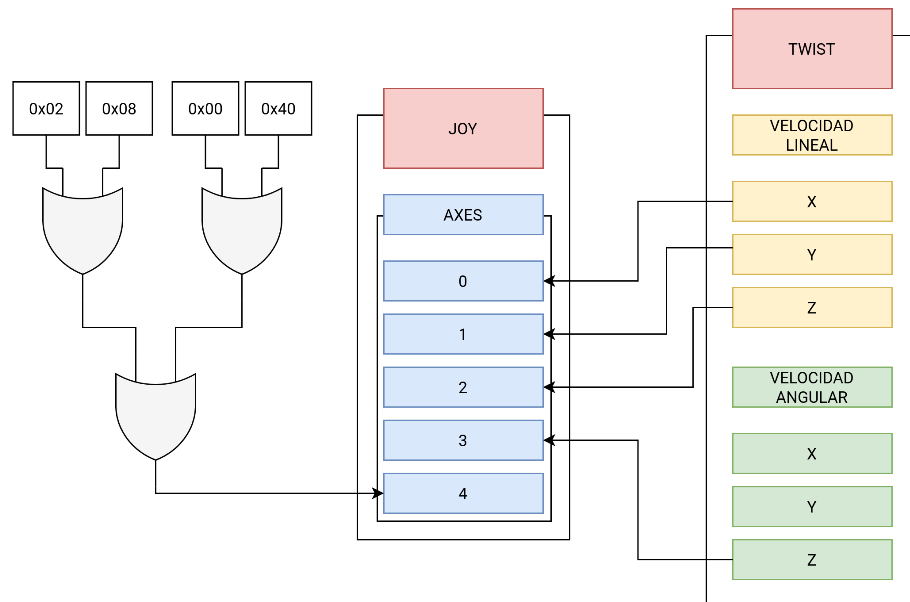

Figura 2.8 Composición de mensaje Joy para modo ASSISTED.

```
class GCSJoyConverter:
    def __init__(self):

      rospy.loginfo("[GCS JOY CONVERTER] Initiating...")

      self.__assisted_ref_sub = rospy.Subscriber("/uav_main_control/
          assisted_control_ref", TwistStamped, self.__assisted_ref_cb)
      self.__ref_pub = rospy.Publisher("/dji_sdk/
          flight_control_setpoint_generic", Joy, queue_size=10)

      self.__joy = Joy()
      self.__joy.header.frame_id = "body_FLU"
      for i in range(0,4):
        self.__joy.axes.append(0) # fill x, y, z and yaw with 0

      self.__flag = 0x40 | 0x00 | 0x08 | 0x02 # h: vel(m/s), v: vel(m/s), y:
          vel(rad/s), c: body_FLU
      self.__joy.axes.append(self.__flag)

      string = bin(self.__flag).replace("0b","")
      rospy.loginfo("[GCS JOY CONVERTER] Flag set to: %s", string)

    def __assisted_ref_cb(self, msg):
      #self.__joy.header.stamp = msg.header.stamp
      self.__joy.axes[0] = msg.twist.linear.x
      self.__joy.axes[1] = msg.twist.linear.y
      self.__joy.axes[2] = msg.twist.linear.z
      self.__joy.axes[3] = msg.twist.angular.z

      self.__ref_pub.publish(self.__joy)
```

Código 2.4 Código de la clase GCSJoyConverter.

2.7 UserInterface

Esta clase contiene los distintos métodos llamados al pulsar los botones de la interfaz gráfica. Cada uno de los métodos realiza una llamada a un servicio, tanto de las máquinas de estados como del SDK.

Tabla 2.2 Servicios llamados por los métodos.

Método	Servicio
takeoff_client	dji_main_state_machine/takeoff
land_client	dji_sdk/drone_task_control
preland_client	dji_auto_state_machine/preland
set_home_tf	dji_auto_state_machine/set_home_tf
arm	dji_sdk/drone_arm_control
set_home_simulator	dji_sdk/set_local_pos_ref

```
class UserInterface():
    def __init__(self):
        rospy.loginfo("[USER INTERFACE] Initiating...")
        self.__set_mode = rospy.ServiceProxy('dji_main_state_machine/set_mode',
             SetMode)
        self.__takeoff = rospy.ServiceProxy('dji_main_state_machine/takeoff',
            Takeoff)
        self.__land = rospy.ServiceProxy('dji_sdk/drone_task_control',
            DroneTaskControl)
        self.__preland = rospy.ServiceProxy('dji_auto_state_machine/preland',
            Trigger)
        self.__gotowp = rospy.ServiceProxy('dji_auto_state_machine/
            go_to_waypoint', GoToWaypoint)
        self.__set_flight_plan = rospy.ServiceProxy('dji_auto_state_machine/
            set_flight_plan', SetFlightPlan)
        self.__sethome = rospy.ServiceProxy('dji_auto_state_machine/set_home_tf'
            , SetHomeTf)
        self.__stop_mission = rospy.ServiceProxy('dji_auto_state_machine/
            stop_mission', Trigger)
        self.__sethome_dji = rospy.ServiceProxy('dji_sdk/set_local_pos_ref',
            SetLocalPosRef)
        self.__arm_control = rospy.ServiceProxy('dji_sdk/drone_arm_control',
            DroneArmControl)

    def set_mode_client(self, req):
        rospy.wait_for_service('dji_main_state_machine/set_mode')
        try:
            self.__set_mode(req)
        except rospy.ServiceException as e:
            rospy.logerr("[USER INTERFACE] Service set_mode call failed: %s"%e)

    def takeoff_client(self, req):
        try:
            self.__takeoff(req)
        except rospy.ServiceException as e:
            rospy.logerr("[USER INTERFACE] Service takeoff call failed: %s"%e)

    def land_client(self):
        try:
            rospy.loginfo("[USER INTERFACE] Calling land service")
```

```
            self.__land(6)
        except rospy.ServiceException as e:
            rospy.logerr("[USER INTERFACE] Service land call failed: %s"%e)

    def preland_client(self):
        try:
            self.__preland()
        except rospy.ServiceException as e:
            rospy.logerr("[USER INTERFACE] Service preland call failed: %s"%e)

    def gotowp_client(self, req, req2):
        try:
            self.__gotowp(req,req2)
        except rospy.ServiceException as e:
            rospy.logerr("[USER INTERFACE] Service gotowp call failed: %s"%e)

    def set_flight_plan_client(self, wp_list):
        try:
            self.__set_flight_plan(wp_list)
        except rospy.ServiceException as e:
            rospy.logerr("[USER INTERFACE] Service set_flight_plan call failed:
                %s"%e)

    def set_home_tf(self, pose):
        try:
            self.__req = Vector3()
            self.__req2 = Quaternion()

            self.__req.x = pose.position.x
            self.__req.y = pose.position.y
            self.__req.z = pose.position.z

            self.__req2.x = pose.orientation.x
            self.__req2.y = pose.orientation.y
            self.__req2.z = pose.orientation.z
            self.__req2.w = pose.orientation.w

            self.__sethome(self.__req,self.__req2)
        except rospy.ServiceException as e:
            rospy.logerr("[USER INTERFACE] Service set_home call failed: %s"%e)

    def arm(self, arm):
        try:
            if arm == 1:
                rospy.loginfo("[USER INTERFACE] Arming")
            elif arm == 0:
                rospy.loginfo("[USER INTERFACE] Disarming")
            self.__arm_control(arm)
        except rospy.ServiceException as e:
            rospy.logerr("[USER INTERFACE] Service set_home call failed: %s"%e)

    def set_home_simulator(self):
        try:
            rospy.loginfo("[USER INTERFACE] Setting simulator home")
            self.__sethome_dji()
            return True
        except rospy.ServiceException as e:
```

```
            rospy.logerr("[USER INTERFACE] Service set_home_simulator call
                failed: %s"%e)
            return False

    def stop_mission(self):
        self.__stop_mission()
```

Código 2.5 Código de la clase UserInterface.

2.8 Resultado

Teniendo ya disponible esta herramienta, se pueden lanzar las máquinas de estados, la GCS y el simulador, consiguiendo que todo funcione tal y como lo haría en un vuelo real, pudiendo así probar las funciones originales y las que se añadan próximamente.

3 Definición e integración de mejoras

El *framework* de navegación tal y como se encuentra es completamente funcional, de hecho, ya ha sido usado en diferentes proyectos de CATEC. Sin embargo, la única medida de seguridad que dispone consiste en activar el modo MANUAL al dejar de recibir información de alguno de los topics especificados.

La implementación de rutinas para la actuación ante posibles fallos es de vital importancia en este tipo de vehículos, puesto que una pérdida del control del dron puede resultar en graves daños objetos, infraestructuras, personas e incluso a la misma aeronave, que puede suponer la pérdida de componentes de alto coste.

Por tanto, se decide que las mejoras a realizar consistirán en la inclusión de actuaciones que hagan el sistema a prueba de fallos, o como es conocido popularmente, *failsafe*.

Para determinar ante que eventos es necesario implementar estas medidas, se toma inspiración las soluciones incluidas por defecto en los drones de DJI[6], Pixhawk[7] y otros sistemas[8]. Como conclusión, se determina que los casos a contemplar serán:

- Nivel de batería crítica.
- Pérdida de la señal RC.
- Exceso de límites.

3.1 Detección de eventos

Al tratarse de inclusión de medidas de seguridad, parece lógico que la detección de los posibles fallos se realice en el Safety Manager, dado que es la máquina de estados de mayor prioridad y su función es precisamente implementar medidas seguridad. A continuación, se detallará cómo se ha llevado la inclusión de estas comprobaciones en este módulo.

3.2 Nivel de batería crítica

Al lanzar el SDK de DJI, uno de los topics publicados es "/dji_sdk/battery_state", en el cual se vuelcan mensajes de tipo *BatteryState*, que contiene distinta información sobre la batería, entre ellas, el voltaje.

Con el valor de la tensión, se puede determinar un valor mínimo de este en el que se quiere que la aeronave opere, acorde con las especificaciones de la batería a bordo.

Para realizar la comprobación del topic, se seguirá el ejemplo de las clases **DeviceTopicChecker** y **RcTopicChecker**, de forma que se pueda mantener la comprobación de la correcta recepción de mensajes gracias a la clase **SubscriberChecker** y se pueda leer la información contenida en ellos.

Primero, se definirá una enumeración, **BatteryStatus**, con los posibles estados, que serán:

- UNKNOWN: aún no se ha inicializado o ha habido algún error.
- OK: el nivel de tensión supera el umbral especificado.
- CRITIC: el nivel de tensión es menos que el umbral.

Después, en el *callback* del suscriptor, se hará la comparación entre el nivel de tensión y el del umbral, el cual se habrá definido como un nuevo parámetro en el fichero de configuración YAML. Dependiendo del resultado de la comprobación, se actualizará una variable con el valor de **BatteryStatus** correspondiente.

El estado de esta variable podrá luego ser obtenido por otra clase llamando al método "getStatus()".

3.3 Pérdida de señal RC

La forma que podría parecer más lógica para detectar una desconexión del mando sería comprobar si en el topic “/dji_sdk/rc” se dejan de publicar mensajes. Sin embargo, con el autopiloto A3, esto no sucede de esta forma. A pesar de que la emisora no esté conectada, en el topic se sigue publicando a la misma frecuencia, con la diferencia de que todos los valores se encuentran a cero.

Como alternativa, se encuentra una solución más sencilla. Existe un topic llamado “/dji_sdk/rc_connection_-status”, donde se publica un mensaje de tipo *UInt8*, cuyo valor es 0 cuando el mando no está conectado y 1 cuando sí.

Se aprovechará la existencia de la clase **RcTopicChecker** para la implementación de esta comprobación. Se añade un nuevo suscriptor al topic mencionado, y en el *callback* se comprueba el valor de mensaje. Si es 0, se cambia el valor del estado del RC a un nuevo estado de la enumeración **RcStatus** llamado LOST.

```
void RcTopicChecker::rc_status_callback(const dji_sdk::UInt8Stamped& msg)
{
   std::lock_guard<std::mutex> lock(_mutex);

   const auto rc_connection_status = msg.data;

   if (rc_connection_status == RcConnection::DISCONNECTED && _current_rc_status
        != RcStatus::LOST)
   {
      ROS_ERROR("[rc_status] RC lost");
      _current_rc_status = RcStatus::LOST;
   }
   else if (rc_connection_status == RcConnection::CONNECTED &&
       _current_rc_status == RcStatus::LOST)
   {
      ROS_WARN("[rc_status] RC connection restored");
      _current_rc_status = RcStatus::MANUAL; // Always return to Manual after
          reconnecion
   }
}
```

Código 3.1 Callback del topic /rc_connection_status.

En el *callback* ya existente, que estaba asociado al topic “/dji_sdk/rc” y gestionaba los cambios de autoridad, se añade una comprobación de si el estado es LOST, y en caso de ser así, no realizará los cambios a MANUAL y AUTO hasta que vuelva a haber conexión.

3.4 Exceso de límites

Una de las características que suelen incluir los UAVs consiste en la definición de *geofences* o *geocages*. Las *geofences* definen un perímetro geográfico donde el dron no puede entrar, mientras que las *geocages* definen un objeto geográfico donde el dron debe permanecer [9]. De este modo, se puede controlar que la aeronave no se interne en zonas peligrosas o donde el vuelo no está permitido, como es el caso de los aeropuertos o recintos militares.

Se decide incluir únicamente las *geocages* en el framework. Para ello, se definirá una nueva clase, **GeocageChecker**, en la que se podrá configurar dos tipos de perímetros: circular y poligonal. Estas serán definidas en el fichero YAML de configuración, indicando el radio en metros en el caso circular, o en el caso del polígono, la posición de los vértices sobre el mapa.

Además, para poder actualizar la *geocage* una vez ejecutado el framework, se hace disponible un servicio, “/set_geocage”, para el cual se ha creado un tipo de servicio personalizado, *SetGeocage.srv*:

```
float32 radius
float32[] x
float32[] y
---
bool success
```

Código 3.2 Mensaje SetGeocage.srv.

La clase heredará de **SubscriberChecker** y se suscribirá a la pose del dron, para poder realizar la comprobación de si la aeronave se encuentra dentro del límite. Esta comprobación se realizará de distinta forma en función del tipo de *geofence* activa:

- Circular: se calcula la distancia del UAV al centro como $\sqrt{x^2+y^2}$ y, si es mayor al radio definido, se considerá que se excedió el límite.
- Poligonal: para poder comprobar si la posición del dron está dentro del polígono definido, se utilizará el algoritmo de *Ray-Casting*, que se basa en el Teorema de la Curva de Jordan para demostrar que, al trazar un línea infinita en cualquiera dirección que parte del punto a estudiar, si el número de veces que corta a la figura es impar, este se encuentra dentro, y si es par, fuera [10].

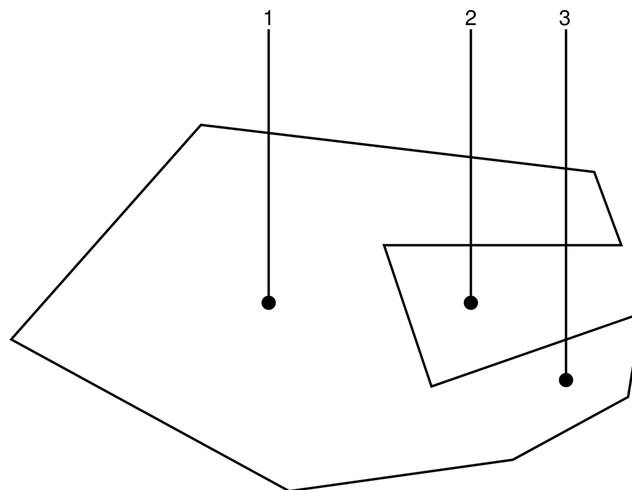

Figura 3.1 Ejemplo de Ray-Casting.

Se definen dos enumeraciones: **GeocageMode**, que define los tipos de *geocage*, pudiendo ser OFF, CIRCLE y POLYGON; y **GeocageStatus**, con las opciones UNKNOWN, IN y OUT para indicar si se ha excedido el límite.

```
void GeocageChecker::callback(const geometry_msgs::PoseStamped::ConstPtr& msg)
{
   std::lock_guard<std::mutex> lock(_mutex);

   ros::Time time = ros::Time::now();
   _buffer.push_back(time.toSec());

   static bool first_call = true;
   if (first_call)
   {
      ROS_INFO("[geocage_checker] First call");
      first_call = false;
```

```
      if (_geocage_polygon_x.size() == _geocage_polygon_y.size() &&
          _geocage_polygon_x.size() > 2)
      {
         ROS_INFO("[geocage_checker] Polygon geocage enabled");
         _geocage_mode = GeocageMode::POLYGON;
      }
      else if(_geocage_radius > 0)
      {
         ROS_INFO("[geocage_checker] Circle geocage enabled");
         _geocage_mode = GeocageMode::CIRCLE;
      }
      else
      {
         ROS_INFO("[geocage_checker] Geocage not available");
         _geocage_mode = GeocageMode::OFF;
         _current_geocage_status = GeocageStatus::IN;
      }
   }

   if (_geocage_mode != GeocageMode::OFF)
   {
      const float px = msg->pose.position.x;
      const float py = msg->pose.position.y;
      bool pose_inside_geocage = true;
      if (_geocage_mode == GeocageMode::POLYGON)
      {
         pose_inside_geocage = isPoseInsideGeocage(px, py);
      }
      else if (_geocage_mode == GeocageMode::CIRCLE)
      {
         pose_inside_geocage = isPoseInsideCircle(px, py);
      }
      if (!pose_inside_geocage)
      {
         ROS_WARN_THROTTLE(10, "[geocage_checker] Geocage exceeded!");
         _current_geocage_status = GeocageStatus::OUT;
      }
      else
         _current_geocage_status = GeocageStatus::IN;
   }
}
```

Código 3.3 Callback del topic /uav_main_control/pose.

3.5 Gestión de failsafes

Una vez detectados los eventos, es necesario un módulo que compruebe el estado de estos, haga alguna gestión si es necesaria y establezca prioridades en caso de que se den varios al mismo tiempo. Para ello, se crea la clase **Failsafe**. En ella, se configura un temporizador para realizar una comprobación recurrente del estados de los posibles eventos, llamando al método "getStatus()" de las clases mencionadas anteriormente.

Se define una nueva enumeración con los posibles estados de *failsafe*, siendo estos:

- BATTERY: el método "getStatus()" de **BatteryTopicChecker** ha devuelto **BatteryStatus::CRITIC**.
- RC_LOST: el método "getStatus()" de **RcTopicChecker** ha devuelto **RcStatus::LOST**. En el momento en que se detecta, se guarda el tiempo en que ha ocurrido.

- RC_TIMEOUT: el método "getStatus()" de **RcTopicChecker** ha devuelto **RcStatus::LOST** y han pasado más de 10 segundos (o el tiempo que se defina) desde que el estado es RC_LOST. Esto es así para dar un margen de tiempo de reconexión del mando antes de realizar una actuación.
- GEOCAGE: el método "getStatus()" de **GeocageChecker** ha devuelto **GeocageStatus::OUT**.
- OFF: ninguna de las condiciones anteriores se ha cumplido.

El estado de *failsafe* podrá ser consultado con el método "getStatus()" y además, se ha creado un nuevo publicador al topic "failsafe_status", de forma que también pueda ser accedido por las otras máquinas de estados.

Se tendrán que realizar cambios también en la máquina de estados del Safety Manager. Cuando el estado de *failsafe* sea provocado por la batería o por el mando, se necesita que la autoridad resida en el ordenador a bordo para poder realizar las maniobras que se programen.

3.6 Rutinas de seguridad

Siendo capaces ya de detectar y gestionar los distintos escenarios de *failsafe*, el último paso será llevar a cabo acciones para evitar los posibles riesgos.

Para ello, en Main State Machine se introducirán dos nuevos estados: RTH y FAILSAFE.

3.7 Return to Home (RTH)

Una de las rutinas de seguridad más comunes en los sistemas analizados consiste en la ejecución de una vuelta a casa (RTH). Al poner en marcha esta maniobra, el dron navegará de forma autónoma al punto designado como *home*, que puede ser el punto de despegue u otro que se haya especificado y, una vez alcanzado, podrá aterrizar o mantenerse *hovering* sobre él.

De igual manera que ocurre con otras funcionalidades implementadas por defecto en el autopiloto, esta es dependiente de la señal GPS para poder realizarse, por tanto, se decide añadir esta característica al *framework* y así poder llevarla a cabo con una localización alternativa.

Se crea un nuevo estado, que tendrá asociada una nueva clase, **RTH**. Al ejecutarse, se publicará como referencia de control la posición actual del dron, cambiando X e Y por 0. De esta forma, la aeronave se dirigirá al *home*, pues este es el origen de coordenadas del sistema de referencia de la aeronave. Al llegar allí, habrá dos opciones: aterrizar o mantenerse sobrevolando el punto. La elección dependerá de una variable, que podrá estar en LAND o HOVER.

```
def execute(self, ud):
    rospy.loginfo('[RTH] - The vehicle is in RTH mode')
    self.common_data.set_uav_status(self.common_data.uav_states.RTH)

    self.common_data.get_rth_abort() # Used to reset abort rth request

    home_wp = copy.deepcopy(self.common_data.get_current_pose())
    home_wp.position.x = 0.0
    home_wp.position.y = 0.0

    self.common_data.set_uav_control_ref(home_wp)

    rospy.loginfo('[RTH] - Returning to [x: %.3f, y: %.3f, z: %.3f]',
                  home_wp.position.x, home_wp.position.y, home_wp.position.z)

    self.ref_pub_timer = rospy.Timer(rospy.Duration(1/50.), self.__pose_pub_cb)

    while not rospy.is_shutdown():

        if self.common_data.get_rth_abort():
            return self.__stop_publish_and_exit('manual')
```

```
        if (self.__check_if_current_waypoint_reached(home_wp)):
            if self.common_data.get_rth_mode() == self.common_data.rth_modes.
                LAND:
                rospy.logwarn('[RTH] - Home reached. Landing...')
                return self.__stop_publish_and_exit('landing')

            rospy.logwarn('[RTH] - Home reached. Hovering...')
            current_control_mode = self.common_data.get_uav_control_mode()
            if current_control_mode == self.common_data.uav_control_modes.AUTO:
                return self.__stop_publish_and_exit('auto')
            if current_control_mode == self.common_data.uav_control_modes.
                OFFBOARD:
                return self.__stop_publish_and_exit('offboard')
            return self.__stop_publish_and_exit('assisted')

        rospy.sleep(0.1)
    return 'shutdown'
```

Código 3.4 Extracto de la clase RTH.

Para poder entrar en este estado, se crea un nuevo servicio: "/rth". Para este, se crea un nuevo mensaje, ReqRTH.srv:

```
uint8 HOVER = 0
uint8 LAND = 1

uint8 mode
---
bool success
string message
```

Código 3.5 Mensaje ReqRTH.srv.

Gracias a este servicio, se podrá solicitar la realización de esta maniobra a voluntad siempre que el estado activo sea AUTO, indicando el comportamiento una vez alcanzado el *home*.

Sin embargo, el objetivo original es que esta maniobra se ejecute automáticamente ante uno de los escenarios de *failsafe* definidos. Por ello, también se podrá acceder a este estado a través del nuevo estado FAILSAFE, que se explica a continuación.

3.8 Estado FAILSAFE

Este nuevo estado entrará en juego cuando se detecte que hay un *failsafe* activo, dato que se podrá conocer suscribiéndose al topic "failsafe_status" publicado por Safety Manager. Se realizará la comprobación de cuál ha sido el evento causante y, en función de este, se actuará de distintas formas:

- BATTERY: se pasará al estado LANDING para realizar un aterrizaje de emergencia inmediato.
- RC_LOST: se mantendrá el dron *hovering* hasta que se produzca una reconexión o pase el tiempo límite establecido.
- RC_TIMEOUT: se saltará al estado RTH en modo LAND.
- GEOCAGE: se saltará al estado RTH en modo HOVER.
- OFF: se vuelve a modo MANUAL.

El salto a este estado puede darse desde todos los estados excepto en los estados de despegue y aterrizaje, puesto que la interrupción de estas maniobras puede resultar peligrosa. Una visión más detallada de las transiciones relacionadas con FAILSAFE puede verse en la Figura 3.2

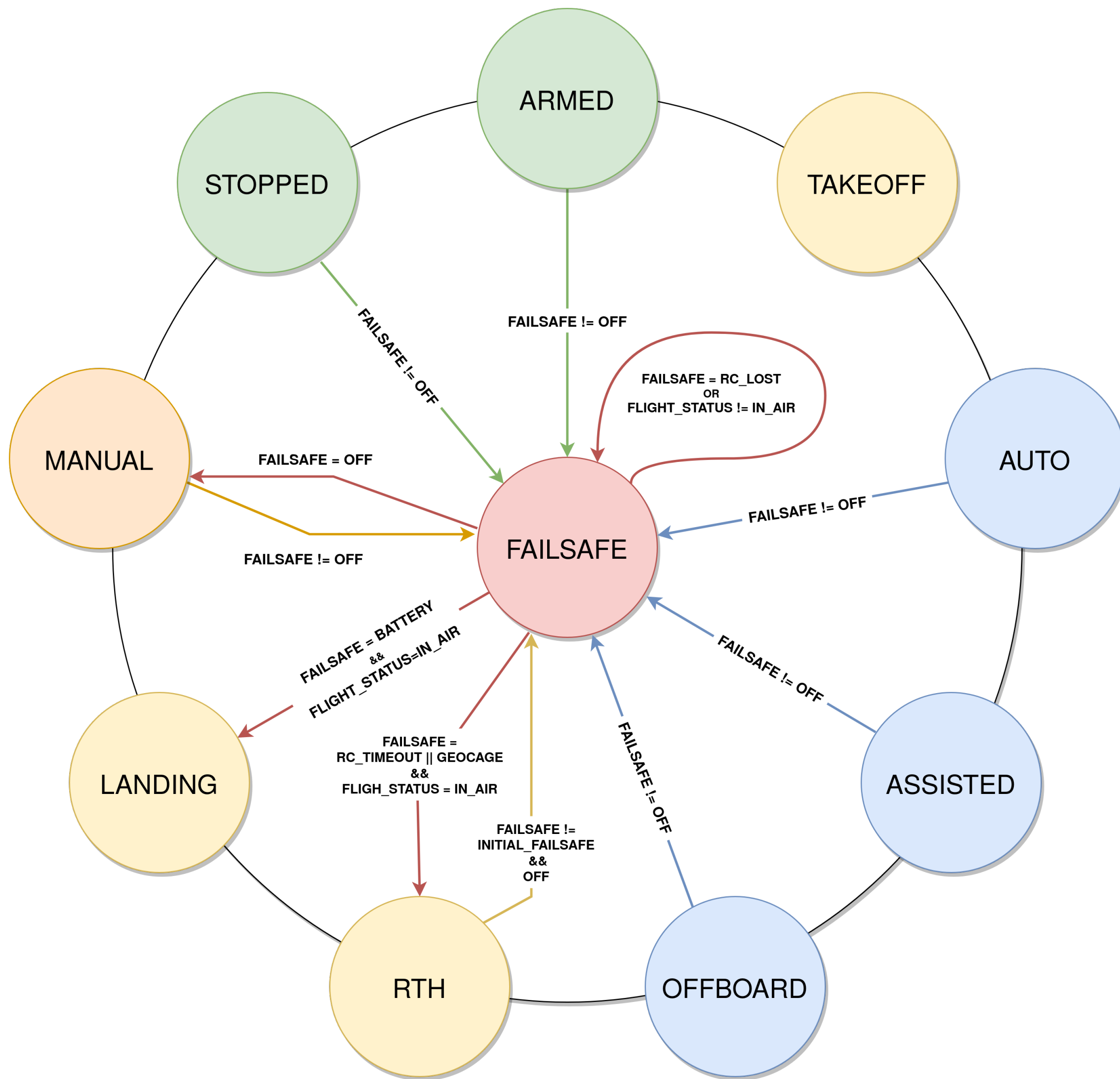

Figura 3.2 Saltos a/desde FAILSAFE.

Otro aspecto a destacar es que si se ha iniciado una maniobra de RTH por *failsafe*, si el estado de *failsafe* cambia a uno de mayor prioridad, se detendrá el movimiento y se realizará la acción correspondiente para este nuevo evento.

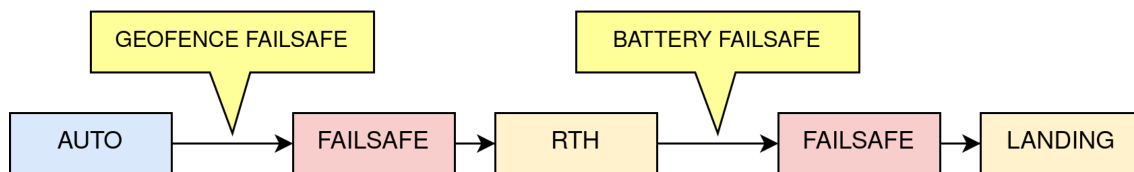

Figura 3.3 Interrupción de RTH por failsafe prioritario.

```
class Failsafe(smach.State):
    def __init__(self, common_data):
        smach.State.__init__(
            self, outcomes=['manual', 'landing', 'rth', 'shutdown'])

        self.common_data = common_data

    def execute(self, ud):
        rospy.loginfo('[Failsafe] - The vehicle is in failsafe mode')
        self.common_data.set_uav_status(self.common_data.uav_states.FAILSAFE)

        while not rospy.is_shutdown():
            failsafe_status = self.common_data.get_failsafe_status()
            if failsafe_status == self.common_data.failsafe_states.OFF: # Not in
                 FAILSAFE anymore
                return 'manual'

            rospy.logerr_throttle(5, '[Failsafe] - Failsafe status: [%s]' %(
                failsafe_status.name))

            uav_flight_status = self.common_data.get_flight_status()
            if uav_flight_status == self.common_data.flight_states.IN_AIR:

                if failsafe_status == self.common_data.failsafe_states.BATTERY:
                    rospy.logerr('[Failsafe] - Critic battery level, vehicle is
                        going to land')
                    return 'landing'
                if failsafe_status == self.common_data.failsafe_states.
                    RC_TIMEOUT:
                    rospy.logerr('[Failsafe] - RC connection lost, vehicle is
                        going to return home')
                    self.common_data.set_rth_mode(self.common_data.rth_modes.LAND
                        )
                    return 'rth'
                if failsafe_status == self.common_data.failsafe_states.GEOCAGE:
                    rospy.logerr('[Failsafe] - Geocage exceeded, vehicle is going
                         to return home')
                    self.common_data.set_rth_mode(self.common_data.rth_modes.
                        HOVER)
                    return 'rth'

            rospy.sleep(0.1)
        return 'shutdown'
```

Código 3.6 Clase FAILSAFE.

Con todo, finalmente la máquina de estados de Main State Machine queda tal y como se puede observar en la Figura 3.4.

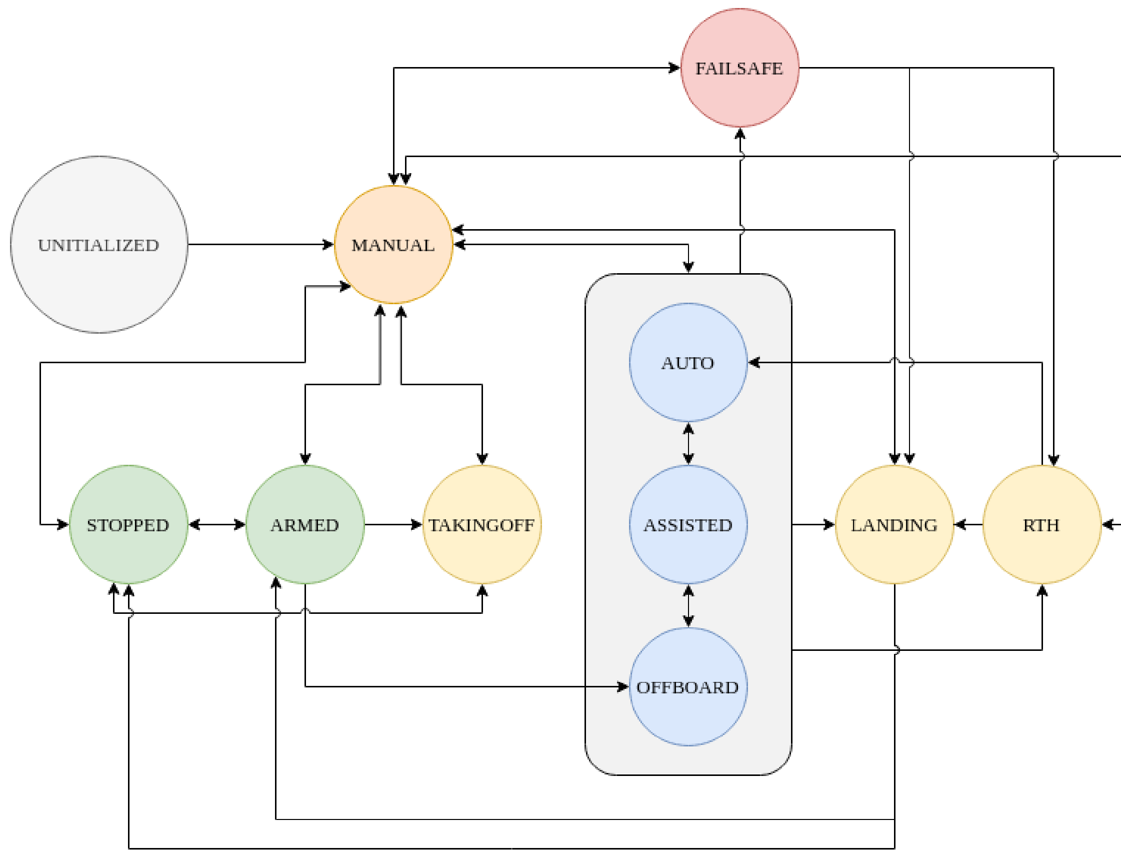

Figura 3.4 Main State Machine con nuevos estados.

4 Adaptación a Pixhawk

Una vez finalizada la etapa de ampliación de las funcionalidades del framework, da comienzo el siguiente apartado del proyecto, la adaptación de las interfaces para añadir compatibilidad con los autopilotos de arquitectura Pixhawk.

El objetivo es realizar esta adaptación realizando los mínimos cambios posibles en el código, de modo que una vez se tenga compatibilidad con ambos autopilotos, exista un *framework* único en el que se seleccione el hardware a utilizar.

El primer paso consistirá por tanto en detectar qué partes del código son reutilizables y cuáles habrá que adecuar para la filosofía de diseño de Pixhawk en cada una de las máquinas de estado.

Tabla 4.1 Incompatibilidades.

Elementos no compatibles	**Motivo**
Librerías	Librería mavros en vez de dji_sdk
Autoridad	No existe el concepto de autoridad en Pixhawk, se alternará entre los distintos modos de vuelo.
RC	La información del mando se recibe de otro modo. No se informa del estado de la conexión.
Estado de vuelo	Diferente forma de detectar si el dron está armado/desarmado y en tierra o en el aire.

Una vez identificados los elementos no compatibles, se procederá a realizar los cambios necesarios, los cuales se han implementado en una copia del *framework*.

4.1 Cambios en Safety Manager

- **Cambios de autoridad**: en Pixhawk, no existe el concepto de autoridad tal y como se encuentra implementado en DJI. Para poder entonces indicar que el control residirá en el *framework*, habrá que cambiar el modo de vuelo a OFFBOARD, en el cual, el dron aceptará el envío de comandos de velocidad. Por tanto, se recodificará el método **tryRequestAuthority** para que realice una llamada al servicio "/mavros/set_mode", solicitando el cambio a modo OFFBOARD para obtener la autoridad, y a MANUAL para ceder el control al piloto.
- **Consulta de DeviceStatus**: en DJI, existían tres posibles estados: MANUAL, APP y SERIAL, que se podían conocer a través de un topic publicado por el autopiloto. Para Pixhawk, se mantendrán estos estados, pero la consulta deberá hacerse a través de otro topic, "/mavros/state", donde se podrá leer el modo de vuelo de la aeronave[11]. Si este es OFFBOARD, se activará el modo SERIAL, en caso contrario, se supondrá MANUAL.
- **Información del mando radiocontrol**: la señal recibida del RC, se publica por otro topic, con un tipo de mensaje distinto, mavros_msgs/RCIn, donde en el miembro *channels* se podrá leer los valores de las distintas entradas.

Surge también el inconveniente de que en Pixhawk no se proporciona información sobre el estado de conexión de la emisora, lo que era necesario para poder activar el *failsafe*. Como alternativa, se implementa un temporizador de tiempo configurable, el cual establecerá la variable **rc_received** a RcConnection::DISCONNECTED. Cuando llega un nuevo mensaje, en el *callback* se le da el valor de RcConnection::CONNECTED. Si esto no hubiera pasado, la próxima vez que se ejecute el *callback* del temporizador, se determinaría que se ha perdido la conexión del mando.

```
_connection_timer = nh.createTimer(ros::Duration(1), &RcTopicChecker::
    conection_check, this);

void RcTopicChecker::conection_check(const ros::TimerEvent&)
{
   std::lock_guard<std::mutex> lock(_mutex);

   if (rc_received == RcConnection::DISCONNECTED && _current_rc_status !=
       RcStatus::LOST)
   {
      ROS_ERROR("[rc_status] RC lost");
      _current_rc_status = RcStatus::LOST;
   }
   else if (rc_received == RcConnection::CONNECTED && _current_rc_status
       == RcStatus::LOST)
   {
      ROS_WARN("[rc_status] RC connection restored");
      _current_rc_status = RcStatus::MANUAL; // Always return to Manual
          after reconnecion
   }

   rc_received = RcConnection::DISCONNECTED;
}
```

Código 4.1 Temporizador para comprobación de desconexión.

4.2 Cambios en Main State Machine

- Detección de Flight Status: en DJI, la aeronave informaba de su estado de vuelo, existiendo tres posibles: STOPPED, ON_GROUND e IN_AIR. En Pixhawk no se nos proporciona esta información de forma explícita, sin embargo, se puede deducir el estado a partir de los mensajes obtenidos por:
  - /mavros/state: informa entre otras cosas del estado de los motores (armado/desarmado), de forma que se puede alternar entre STOPPED y ON_GROUND.
  - /mavros/extended_state: proporciona información del estado del dron en un mensaje de tipo **ExtendedState**[12]. Cuando el campo *landed_state* tenga un valor distinto de 1 (equivalente a LANDED_STATE_ON_GROUND), se considerará estado IN_AIR.

Se realizará la suscripción a estos dos topics mediante la librería **message_filters**, que permite la recepción y sincronización de mensajes recibidos por varios topics, de forma que el *callback* se ejecutará una vez se hayan recibido los datos de ambos. La lógica por tanto será:

```
if not armado:
    flight_status = STOPPED
else:
    if LANDED_STATE_ON_GROUND:
        flight_status = ON_GROUND
    else:
        flight_status = IN_AIR
```

4.3 Cambios en Auto State Machine

Para este código no será necesario realizar ninguna modificación, pues la única comunicación que realiza es con el control de MATLAB, que no ha sufrido cambios.

4.4 Adaptación de la simulación

Al igual que en DJI, se ve necesario poder realizar pruebas en simulación ante la dificultad de volar aeronaves reales. Para lograr esto, se hace uso de las herramientas de simulación ofrecidas por PX4 para el entorno Gazebo.

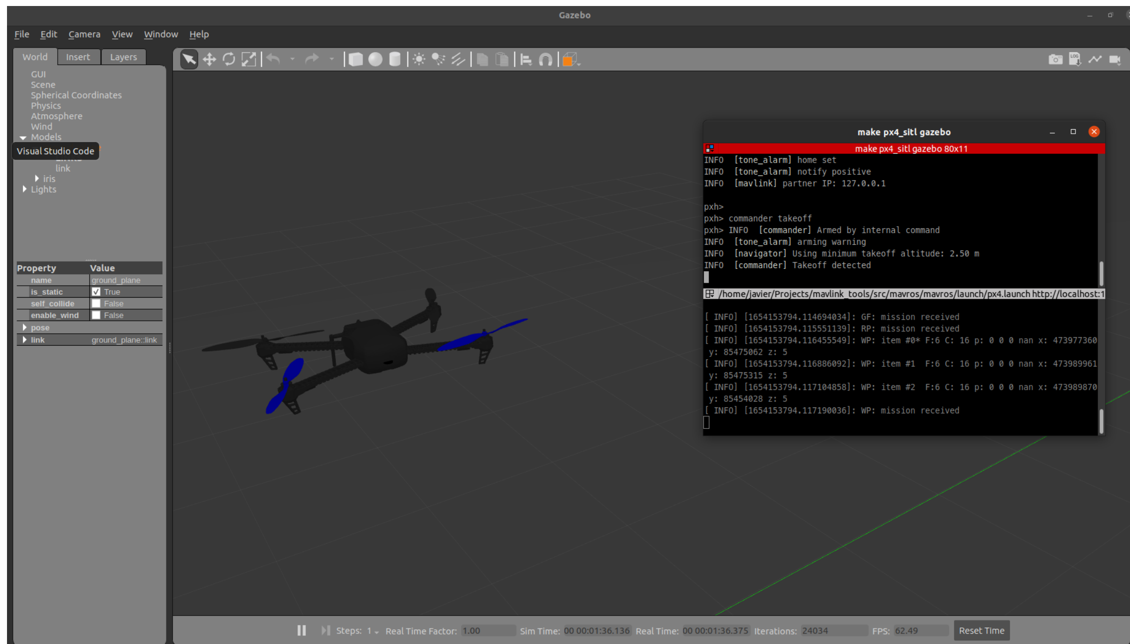

Figura 4.1 Simulación PX4 en Gazebo.

Será preciso nuevamente hacer uso de una interfaz intermedia que haga de puente entre el *framework* y el entorno de simulación, para suplir la falta del control. Se realizaran algunas modificaciones al código ya desarrollado para DJI:

- Cambiar topics y servicios de DJI por los de Mavros y el procesado de los mensajes recibidos.
- Publicar constantemente una referencia de posición, puesto que para poder mantener el autopiloto en modo OFFBOARD, se debe estar indicando periódicamente, aunque esta no cambie.

Teniendo ya disponible por tanto la simulación, se es capaz de realizar distintas pruebas que permiten entender mejor el funcionamiento de Pixhawk y corregir algunos errores en el código.

5 Unificación de códigos

Como se comentó anteriormente, el objetivo final es disponer de un único *framework* que soporte tanto autopilotos de DJI como de Pixhawk, pudiendo indicar cual de ellos se va a utilizar para la aplicación y que dentro del programa se ajuste automáticamente la compatibilidad, cargando las librerías y funciones necesarias.

La ventaja de haber utilizado el mismo código para Pixhawk realizando las mínimas modificaciones posibles es que no hace falta tener ficheros específicos para cada uno de los sistemas, sino que en los archivos de la versión de DJI, se añadirán las modificaciones hechas para Pixhawk y, mediante el uso de directivas, se seleccionará las funcionalidades adecuadas, como en el siguiente ejemplo:

```
#if defined(PIXHAWK)
  _authority_srv = nh.serviceClient<mavros_msgs::SetMode>("/mavros/set_mode");
#elif defined(DJI)
  _authority_srv = nh.serviceClient<dji_sdk::SDKControlAuthority>("/dji_sdk/
      sdk_control_authority");
#endif
```

Código 5.1 Configuración de servicio con macros.

PIXHAWK y DJI, son dos flags de precompilación que se han añadido al archivo CMakeList.txt con la intención de poder indicar el autopiloto a utilizar, comentando aquel que no se vaya a usar, de forma que no se defina en el momento de compilación, y por tanto, consiguiendo que únicamente compile la sentencia incluida en la comprobación satisfecha. En el ejemplo, lo que se puede observar es que se configura el servicio del cambio de autoridad de la aeronave en función de la aeronave utilizada cambiando el tipo de mensaje utilizado y el nombre del servicio.

```
# Uncomment for PIXHAWK
# add_definitions(-DPIXHAWK)
# Uncomment for DJI
add_definitions(-DDJI)
```

Código 5.2 Extracto de CMakeList.txt para selección de flag.

Aquellos puntos donde se ha hecho uso de la implementación de esta lógica para adaptar el código a cada autoipiloto han sido:

- **Inclusión de mensajes de librerías**: los mensajes utilizados para los topics y servicios propios de los autopilotos se encuentran en "dji_sdk" para DJI y en "mavros_msgs" para Pixhawk. Gracias a que no se compilará aquella parte que no se encuentre dentro de la sentencia utilizada, no será necesario tener instaladas las librerías de ambos.

  ```
  #if defined(PIXHAWK)
  #include <mavros_msgs/RCIn.h>
  ```

```
#elif defined(DJI)
#include <dji_sdk/UInt8Stamped.h>
#endif
```

Código 5.3 Ejemplo de elección de mensajes a incluir.

- **Suscripción a topics**: los topics por donde se recibe la información de la batería, el mando y el estado varía en función del autopiloto.

```
 #if defined(PIXHAWK)
_rc_topic = std::make_shared<RcTopicChecker>(_nh, "/mavros/rc/in", 10,
    parameters.safety_margin, parameters.p_mode_selector_min,
    parameters.p_mode_selector_max, parameters.set_authority_min,
    parameters.set_authority_max);
_device_topic = std::make_shared<DeviceTopicChecker> (_nh, "/mavros/
    state", 1, parameters.safety_margin);
_battery_topic = std::make_shared<BatteryTopicChecker>(_nh, "/mavros/
    battery", 0.5, parameters.safety_margin, parameters.
    critic_battery_threshold);
 #elif defined(DJI)
 _rc_topic = std::make_shared<RcTopicChecker>(_nh, "/dji_sdk/rc", 50,
     parameters.safety_margin, parameters.p_mode_selector_min,
     parameters.p_mode_selector_max, parameters.set_authority_min,
     parameters.set_authority_max);
_device_topic = std::make_shared<DeviceTopicChecker> (_nh, "/dji_sdk/
    device_status", 50, parameters.safety_margin);
_battery_topic = std::make_shared<BatteryTopicChecker>(_nh, "/dji_sdk/
    battery_state", 5, parameters.safety_margin, parameters.
    critic_battery_threshold);
 #endif
```

Código 5.4 Ejemplo de suscripciones.

- **Configuración de *callback***: asociado al punto anterior, los *callbacks* de cada suscriptor deberán estar adecuados al tipo de mensaje y a la acción que tendrán realizar.

```
#if defined(PIXHAWK)
void RcTopicChecker::callback(const mavros_msgs::RCIn::ConstPtr& msg)
{
   authority_reader(msg->channels[4], msg->channels[5]);
}
#elif defined(DJI)
void RcTopicChecker::callback(const sensor_msgs::Joy::ConstPtr& msg)
{
   authority_reader(msg->axes[4], msg->axes[5]);
}
#endif
```

Código 5.5 Ejemplo de elección de *callback*.

- **Detección de desconexión del mando radio control**: mientras que en DJI se disponía del topic del estado de la conexión, en Pixhawk era necesario instanciar un temporizador.

```
#if defined(PIXHAWK)
_connection_timer = nh.createTimer(ros::Duration(1), &RcTopicChecker::
    conection_check, this);
```

```
#elif defined(DJI)
_rc_status_sub = nh.subscribe("/dji_sdk/rc_connection_status", 1, &
    RcTopicChecker::rc_status_callback, this);
#endif
```

Código 5.6 Selección de tipo de detección de desconexión del mando.

La principal ventaja de tener un único *framework* para ambos reside en que en el caso de que en el futuro se añadan funcionalidades o mejoras, no sea necesario modificar el código de la misma manera en dos repositorios distintos. Además, en el caso de que el cambio sea específico únicamente para uno de los autopilotos, no interferirá en nada en el funcionamiento del otro.

Aplicando este método para todas las modificaciones, nace el *framework* final, renombrando las máquinas de estados a uav_safety_manager, uav_main_state_machine y uav_auto_state_machine y renombrando los topics y servicios con el prefijo "uav" también, puesto que ahora el sistema no es específico para un tipo de autopiloto.

6 Resultados experimentales

Finalizado el desarrollo, el último paso será realizar pruebas de vuelo para confirmar el correcto funcionamiento de las funcionalidades originales del *framework*, de las nuevas y de la adaptación a Pixhawk. En empresas de tecnología, es de gran importancia el diseño y ejecución de tests de validación, para poder demostrar el correcto funcionamiento del sistema antes de ser implementado en futuros proyectos o ser vendido a un cliente. A pesar de que durante todo el desarrollo se han ido haciendo pruebas frecuentes del sistema, estas se realizaron de forma improvisada y sin guardar evidencias de los resultados. Por tanto, se decide diseñar un plan de vuelo con distintas pruebas a realizar, con el fin de poder ver la respuesta de las nuevas funcionalidades en distintos escenarios. Se han dividido los tests en cuatro categorías:

- RTH: pruebas relacionadas con la la llamada al servicio de Return to Home. Se comprobará que se puede solicitar la maniobra desde los estados permitidos, en los dos modos disponibles (HOVER y LAND) y que es posible detener la maniobra a voluntad.
- RC: pone a prueba la detección de desconexión y reconexión del mando a distancia, con su rutina asociada.
- GEO: permite comprobar que la detección de geofences funciona correctamente.
- FAIL: consistirán en pruebas adicionales del estado de failsafe.

Prueba	Descripción
RTH1	Estando el dron en modo AUTO/HOVERING, a cierta distancia del home, se llamará al servicio de RTH en modo HOVER. El dron volverá a la posición X e Y del home y se mantendrá hovering, no aterrizará.
RTH2	Estando el dron en modo AUTO/HOVERING, a cierta distancia del home, se llamará al servicio de RTH en modo LAND. El dron volverá a la posición X e Y del home y aterrizará.
RTH3	Estando el dron en modo AUTO/HOVERING, a cierta distancia del home, se llamará al servicio de RTH en modo LAND. Antes de la llegada del dron al home, se llamará al servicio de ABORT RTH. El dron iniciará el regreso al home pero se detendrá y entrará en HOVERING al abortar.
RTH4	Estando el dron en modo AUTO/GOTOWP, a cierta distancia del home, se llamará al servicio de RTH en modo LAND. El dron volverá a la posición X e Y del home y aterrizará.
RTH5	Llamar al servicio /dji_main_state_machine/rth en todos los estados, y comprobar que se cumple la tabla.
RTH6	Llamar al servicio /dji_main_state_machine/abort_rth en todos los estados, y comprobar que se cumple la tabla.
RC1	Desconectar RC estando en AUTO/HOVERING. El dron deberá volver al home después del timeout y aterrizar. Una vez en el suelo se quedará en estado FAILSAFE.
RC2	Desconectar RC estando en AUTO/HOVERING y reconectar antes del timeout. El dron deberá haber saltado a modo FAILSAFE y cambiar a MANUAL tras reconexión.
RC3	Desconectar RC estando en AUTO/HOVERING, esperar a timeout y reconectar a mitad de RTH. El dron deberá empezar la vuelta al home pero se detendrá y volverá a MANUAL tras la reconexión.
RC4	Desconectar RC estando en AUTO/GOTOWP. El dron deberá detener la misión, volver al home después del timeout y aterrizar. Una vez en el suelo se quedará en estado FAILSAFE.
GEO1	Se definirá una geocage poligonal de tamaño inferior al espacio de vuelo. En modo MANUAL, se pilotará al dron fuera del límite. El dron deberá detectar el exceso del geocage, mostrar una advertencia, pero seguir en modo MANUAL.
GEO2	Se definirá una geocage poligonal de tamaño inferior al espacio de vuelo. En modo AUTO, se establecerá un waypoint fuera del límite y se iniciará la misión. El dron deberá detectar el exceso del geocage, tomar la autoridad e iniciar el RTH. Se realizará varias veces y con distintos límites para comprobar su robusto funcionamiento.
GEO3	Se definirá una geocage no válida (vacía, con menos de 3 puntos o con size(x) != size(y)). Se comprobará que se ha dado como inválida y que nunca se sobrepasará por tanto un geocage.
GEO4	Se definirá una geocage circular. En modo AUTO, se establecerá un waypoint fuera del límite y se iniciará la misión. El dron deberá detectar el exceso del geocage, tomar la autoridad e iniciar el RTH.
FAIL1	Forzar un FAILSAFE en todos los estados y comprobar que solo se accede a FAILSAFE desde los estados estipulados.
FAIL2	Forzar un FAILSAFE/GEOCAGE, y durante el RTH, forzar FAILSAFE/RC_LOST. El dron deberá detenerse, entrar en modo FAILSAFE y tras el timeout, iniciar RTH de nuevo en LAND.
FAIL3	Forzar un FAILSAFE/BATTERY y comprobar que el dron entra en LAND inmediatamente.

Figura 6.1 Plan de vuelo.

La tabla a la que se hace referencia en RTH5 y RTH6, es la nueva tabla de compatibilidades estado-servicio, que incluye ahora los nuevos estados RTH y FAILSAFE, y los nuevos servicios "/rth" y "/abort_rth", como se puede ver en la Tabla 6.1.

Tabla 6.1 Tabla de compatibilidad estado-servicio.

	/set_mode	/takeoff	/land	/set_control_ref	/rth	/abort_rth
UNINITIALIZED	OK	ERROR	ERROR	ERROR	ERROR	ERROR
STOP	OK	ERROR	ERROR	ERROR	ERROR	ERROR
ARMED	OK	OK	ERROR	ERROR	ERROR	ERROR
TAKEOFF	OK	ERROR	ERROR	ERROR	ERROR	ERROR
AUTO	OK	ERROR	OK	OK	OK	ERROR
ASSISTED	OK	ERROR	OK	ERROR	ERROR	ERROR
OFFBOARD	OK	ERROR	OK	ERROR	ERROR	ERROR
LAND	OK	ERROR	ERROR	ERROR	ERROR	ERROR
MANUAL	OK	ERROR	ERROR	ERROR	ERROR	ERROR
RTH	OK	ERROR	ERROR	ERROR	ERROR	OK
FAILSAFE	OK	ERROR	ERROR	ERROR	ERROR	ERROR

Antes de realizar las pruebas con un dron real, se llevan a cabo en los entornos de simulación de DJI y Pixhawk, para detectar posibles fallos antes del vuelo con la aeronave física.

Una vez completado con éxito el plan de vuelo en simulación, podrá comenzar la preparación del vuelo real. El primer paso será elegir el dron a utilizar, aunque la elección era dependiente de la disponibilidad de las aeronaves de CATEC. Finalmente, se pudo disponer de un hexacóptero de DJI (Figura 6.2), compuesto por el frame Flame Wheel[13], un autopiloto DJI A3 y una NVIDIA Jetson TX2 montada sobre la Orbitty Carrier[14].

El *framework* de navegación, será ejecutado en el ordenador a bordo, ya que este posee conexión directa con el autopiloto. Este ordenador tiene como sistema operativo Ubuntu 18.04, que se trata del mismo con el que se ha desarrollado todo el proyecto, por lo que la configuración resulta sencilla.

Los vuelos se realizarán en el *Testbed*, una zona en el interior de la nave de FADA-CATEC preparada para la realización de vuelos de prueba. Este recinto está equipado con una instalación de Vicon, un sistema compuesto por multitud de cámaras repartidas por todo el recinto que permiten una localización de alta precisión de un objeto equipado con una serie de marcadores especiales. Esta herramienta será indispensable para las pruebas, ya que el dron utilizado no dispone de ningún sistema de localización mediante LiDAR o cámaras, pero sí está equipado con marcadores Vicon.

Figura 6.2 Dron utilizado en las pruebas de vuelo.

Figura 6.3 Cámaras Vicon en Testbed.

Será necesario instalar y configurar en el ordenador a bordo el paquete de ROS *vicon_bridge*[15], que permite obtener la información de Vicon y hacerla compatible con ROS publicando la posición como un mensaje **PoseStamped**. Además, será necesario hacer una transformación de los sistemas de referencia, pues el origen de coordenadas de Vicon puede no coincidir con el *home*. Se desarrolla un script que obtiene la transformación entre los frame_ids "vicon" y "map", asumiendo que la posición en "map" es el origen de coordenadas y, por tanto, tomando como traslación y rotación la pose publicada por Vicon; y luego, publica la posición leída, aplicando dicha transformación, al topic "/uav_main_control/pose", que es el consultado tanto por las máquinas de estados como por el control. Se resuelve así el problema de localización del dron.

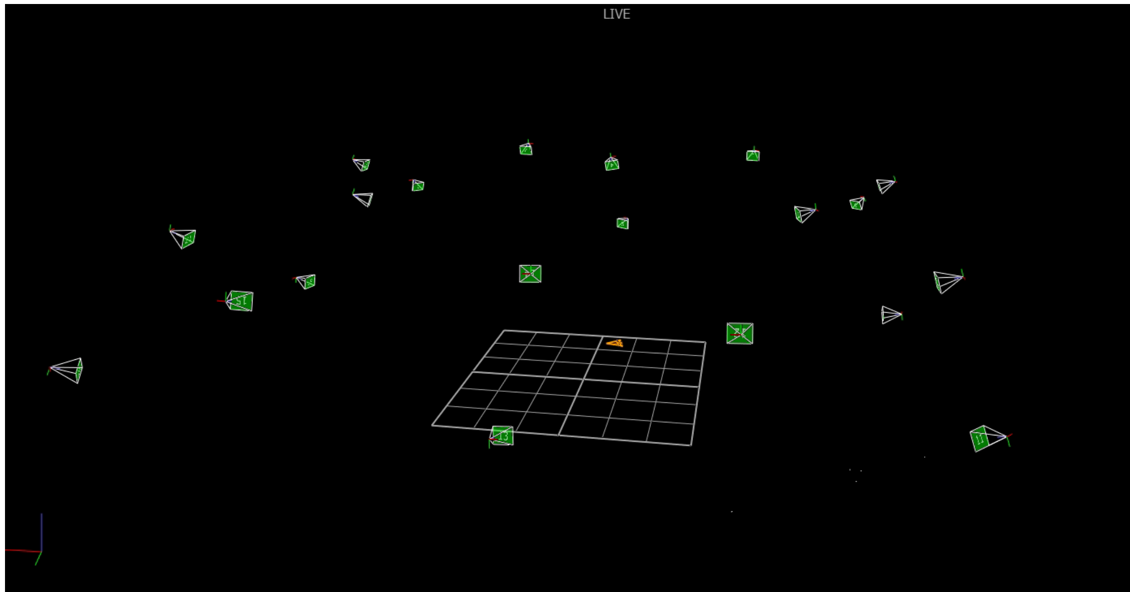

Figura 6.4 Visualización de la posición del dron con el software de Vicon.

Por último, para terminar los preparativos previos al vuelo, será necesario la instalación del control de la aeronave. Este fue desarrollado en MATLAB por la USA (Unidad de Sistemas Autónomos) para poder adaptarse a distintas aeronaves. Como este modelo de UAV ya se había utilizado previamente, ya existía una configuración, por tanto, solo fue necesario exportarlo para ROS como un paquete más, el cual se instalará en el *workspace* del ordenador a bordo y será ejecutado junto al resto de módulos.

Una vez terminado el proceso de preparación, se está listo para comenzar las pruebas de vuelo.

6.1 Pruebas de vuelo

6.1.1 Día 1

El primer día de vuelo se centró en probar que la localización, el control y las funcionalidades originales de *framework* estuvieran funcionando correctamente. Para ello, se programó desde la GCS una misión sencilla, en la que el dron realizaría una trayectoria rectangular, habiendo despegado anteriormente de forma manual.

El primer fallo detectado fue que el control no actuaba al cambiar la referencia. El problema provenía de que como el control se había diseñado para el *framework* original, los topics a los que estaba suscrito tenían como nombre "/dji_...", y en el versión nueva, se habían renombrado a "/uav_...", por lo que no estaba recibiendo ninguna información. Como modificar el script de MATLAB no entra en el alcance de este proyecto, se optó por remapear el nombre de los topics a los originales.

Una vez realizado este cambio, se comprobó que las funcionalidades originales seguían funcionando correctamente.

6.1.2 Día 2

Para el segundo día se pretendía realizar las pruebas RTH y GEO.

Se detecta un nuevo fallo durante el test: al entrar en el estado RTH, la aeronave no volvía al *home* y derivaba. El motivo era nuevamente el control de MATLAB, pues este está configurado para que el seguimiento de referencias únicamente se active cuando el estado de Main State Machine sea AUTO, por lo que al entrar en modo RTH, el control no se encontraba en funcionamiento.

Como solución provisional, se decidió "engañar" al control, publicando por el topic del estado de Main State Machine (/dji_main_state_machine/status) que el dron se encuentra en AUTO cuando realmente está en RTH. De esta forma, el control se activará y la maniobra de vuelta a casa ya funciona.

Ya solucionado, la jornada puede continuar sin más inconvenientes y las pruebas de vuelo RTH y GEO son completadas con éxito.

6.1.3 Día 3

En el tercer y último día, se planea finalizar el plan completando las pruebas RC y FAIL.

Debido a las medidas de seguridad establecidas por CATEC, las pruebas de desconexión del mando no podrán realizarse desconectando realmente el radio control, pues en caso de que el dron se descontrolara, podría causar daños. Por tanto, como alternativa, se remapeará el topic de conexión del mando a uno que no será leído por el sistema, y se progamará un *script* que publicará al topic original, pudiendo cambiar a voluntad el mensaje a 1 (conectado) o 0 (desconectado). Aunque no se trate del mismo procedimiento, el resultado debería ser el mismo.

Para la prueba de la batería, se remapeará también el topic original y se hará un *script* para simular la descarga de esta.

Realizadas estas adaptaciones, se finaliza el plan de vuelo con nuevamente con éxito.

En las siguientes figuras, se podrá observar como ante los eventos de pérdida de la conexión con el radio control y al descender el voltaje de la batería por debajo del umbral, el estado cambia a failsafe, para luego ejecutar la acción correspondiente.

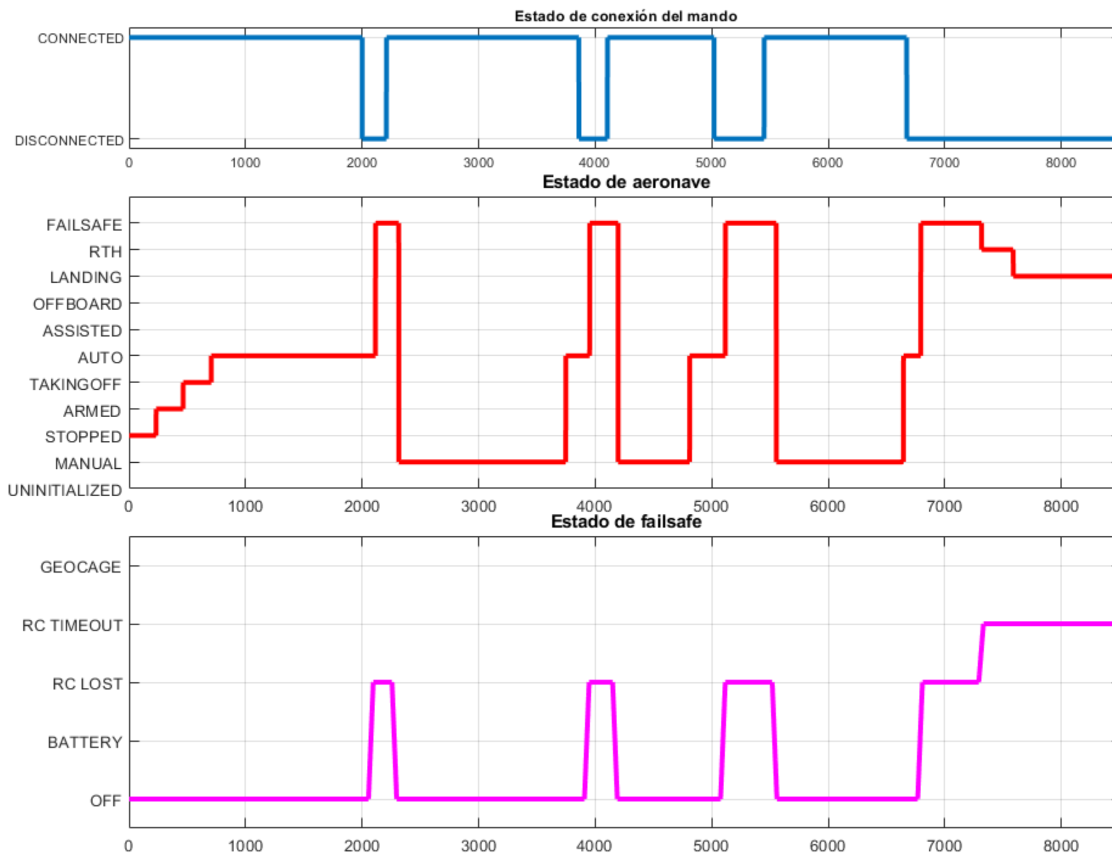

Figura 6.5 Prueba de vuelo RC1 y RC2
Se puede observar como primero se realizan tres primeras desconexiones y reconexiones en un tiempo menor al *timeout*, lo que hace que el estado cambie primero a FAILSAFE, para luego pasar a MANUAL tras la reconexión (RC2). En la cuarta y última desconexión, se supera el tiempo límite y la aeronave comienza la maniobra de RTH y posterior aterrizaje (RC1).

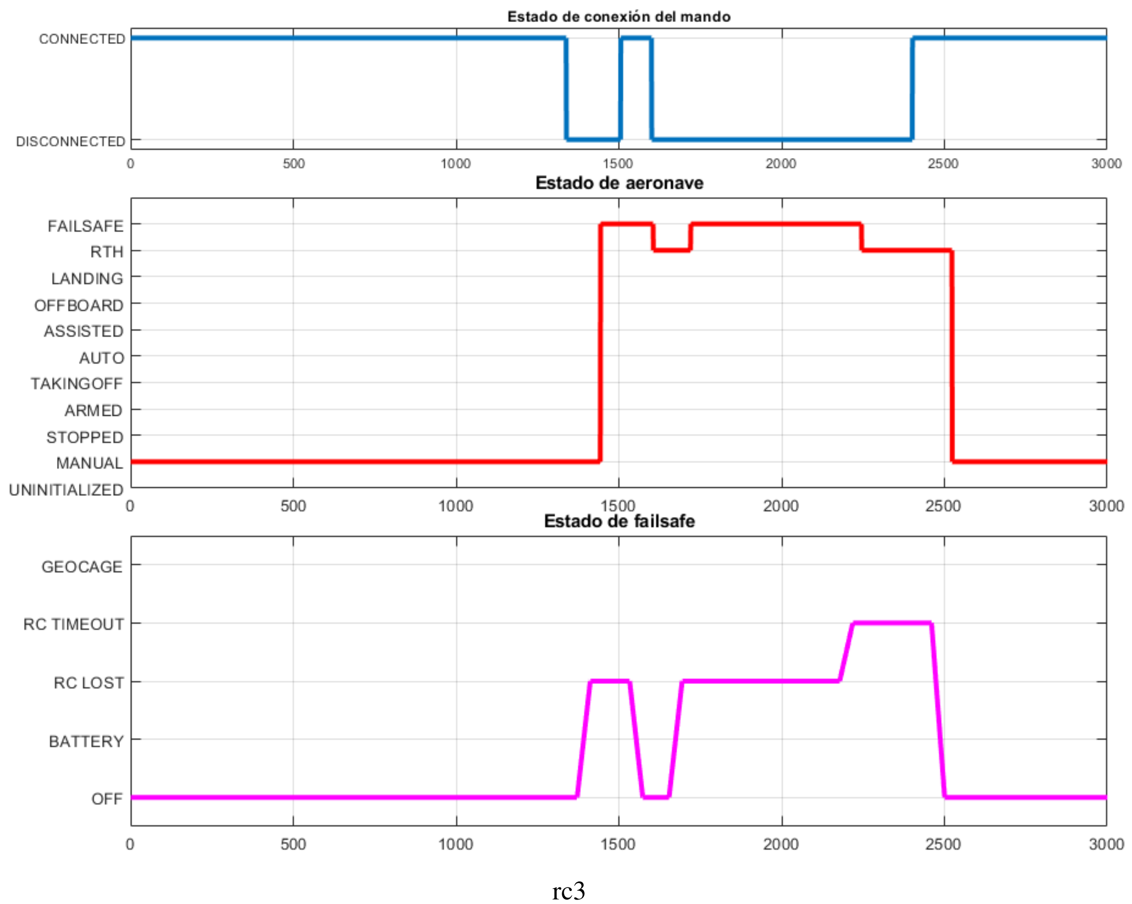

Figura 6.6 Prueba de vuelo RC3
En este caso, se desconecta el mando y se espera al *timeout*, comenzando así la maniobra de RTH. Mientras esta está en ejecución, se recupera la conexión, desactivando el failsafe y pasando a modo manual, tal y como se esperaba.

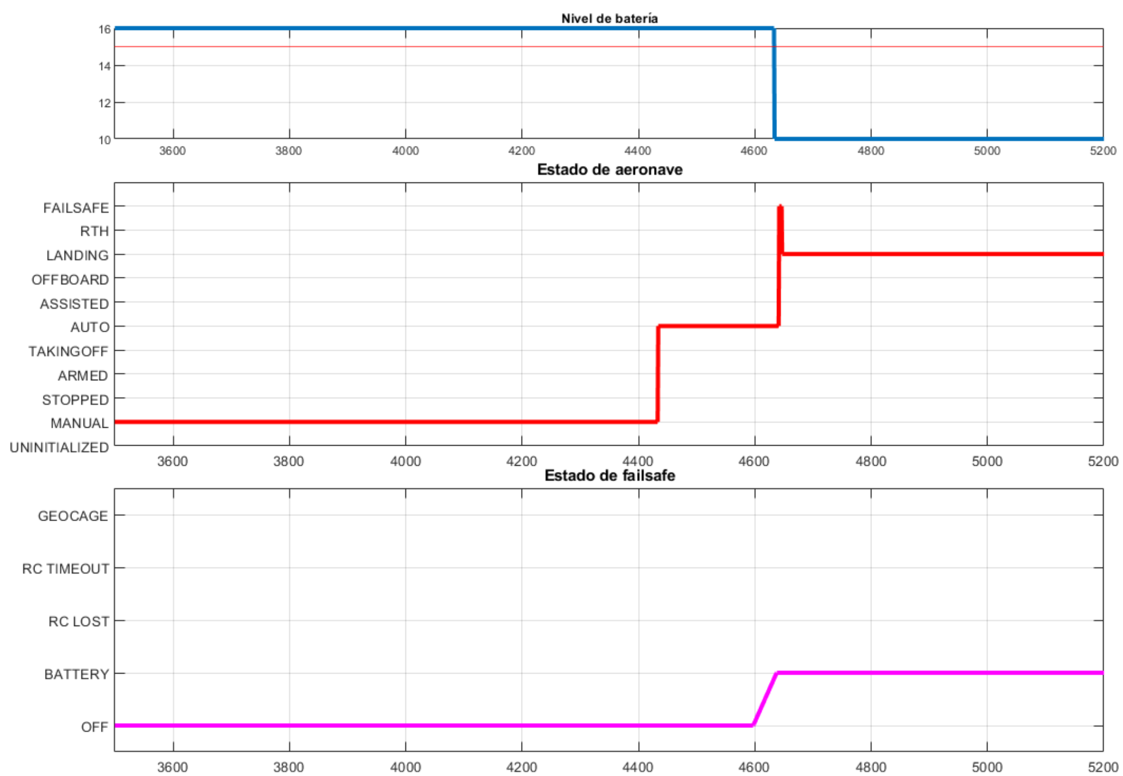

Figura 6.7 Prueba de vuelo FAIL3
Se puede comprobar en esta prueba como al descender el voltaje de la batería por debaje del límite establecido (línea roja), comienza inmediatamente la maniobra de aterrizaje.

7 Conclusiones

Una vez finalizadas las pruebas de vuelo, se puede dar por concluido el desarrollo del proyecto, y por tanto, es momento analizar el trabajo realizado, los resultados obtenidos y comprobar si se han cumplido los objetivos planteados al inicio.

En cuanto al código, se ha obtenido una estructura robusta y acorde con la filosofía de diseño original, siendo este fácil de entender para el usuario y a prueba de fallos. Como no se han añadido nuevas dependencias y el funcionamiento de los elementos de la primera versión no han sufrido cambios, la actualización del *framework* en aquellos proyectos que lo utilicen será sencilla.

La detección de los eventos de desconexión del mando radio control, intrusión en zonas de vuelo restringida y batería crítica y las maniobras asociadas a cada uno, permitirán que a partir de ahora, los proyectos que incluyan el *framework* dispongan de un añadido de seguridad, lo que reducirá en gran medida el riesgo de accidentes que puedan dar lugar a: retrasos debidos al tiempo empleado en reparar drones que hayan sufrido algún accidente durante las pruebas, gastos por el reemplazo de nuevas piezas y daños a personas e infraestructuras.

Las pruebas de vuelo han permitido comprobar el correcto funcionamiento de la lógica implementada, demostrando su robustez en varios escenarios, así como ha permitido plantear la futura línea de trabajo para la actualización del control de MATLAB, añadiendo soporte para las nuevas funcionalidades.

Sin embargo, solo se han podido realizar las pruebas para el autopiloto de DJI, del cual se esperaba que no resultara en demasiados conflictos, pues el diseño original ya probó su compatibilidad con esta arquitectura. La parte de Pixhawk, para la cual se ha tenido que adaptar el sistema, no ha podido ser probada en un autopiloto real, y, aunque se ha probado en simulación, en un futuro sería conveniente llevar a cabo el plan de vuelo tal y como se realizó con DJI.

En conclusión, revisando los objetivos planteados al inicio del proyecto, los cuales eran:

- Estudio y comprensión del código ya existente: se realizó un análisis detallado del código existente, el cual se puede consultar en el Apéndice C. Este estudio ha sido fundamental para el desarrollo del trabajo.
- Implementación de nuevas funcionalidades: se han implementado con éxito las siguientes funcionalidades:
  - Detección de eventos:
    - Nivel de batería crítica, siendo este configurable al voltaje deseado.
    - Pérdida de señal RC.
    - Exceso de límites mediante la definición de geocages.
  - Actuación ante los eventos anteriormente descritos, habiendo sido necesario diseñar la rutina de RTH.

- Adaptación para autopilotos Pixhawk: se modificaron aquellas funcionalidades que requerían un cambio para poder comunicarse con este tipo de sistemas.
- Creación de *framework* único para ambos entornos: mediante el uso de los flags de precompilación, se obtuvo un único repositorio con soporte para ambos autopilotos.
- Realización de experimentos para comprobar el correcto funcionamiento del código desarrollado: se diseñó y puso en práctica un plan de vuelo para poner a prueba las mejoras implementadas.

Otro aspecto a destacar, es que el desarrollo del puente entre el *framework* y la simulación, aunque no estaba previsto al comienzo, ha resultado ser una parte vital para el avance del proyecto, pues ha permitido realizar pruebas continuamente, lo que ayudaba a depurar el código. Además, esta herramienta ha sido útil para CATEC, puesto que se ha usado para hacer simulaciones previas a los vuelos reales en otros proyectos que utilizaban el *framework*.

Como futuras líneas de desarrollo, se plantean distintas opciones:

- Sugerir RTH cuando se calcule que la batería no va a ser suficiente para volver al *home* si el vuelo se extiende durante más tiempo.
- Soporte para múltiples geofences.
- Avisos cuando la aeronave se aproxime a una geofence o si la ruta establecida entra en conflicto con estas.
- Evitación de obstáculos.
- Adaptación de la GCS para soportar los nuevos estados, las llamadas a los nuevos servicios de solicitud y parada de maniobra de RTH y la visualización de la geofences.

A nivel personal, este proyecto ha sido una gran oportunidad para aprender cómo desarrollar software a un nivel profesional, entorno en el que existe una gran exigencia de fidelidad del sistema, pues el objetivo final será la creación de un producto comercial que deberá funcionar correctamente. Me ha permitido también afianzar y ampliar mis conocimientos en C++, Python y ROS, herramientas fundamentales en la industria de la automatización y robótica, así como también me ha permitido trabajar con dos de los autopilotos más populares para el desarrollo, como son DJI y Pixhawk.

Apéndice A
Hardware

El principal componente hardware usado en este proyecto, se trata del dron. El tipo de dron con el que se trabajará será el multirrotor. Un multirrotor se compone de varias hélices situadas por lo general en el extremo de unos brazos que lo unen a un cuerpo. Las configuraciones más comunes se componen de 4, 6 u 8 hélices, cada una de ellas con un controlador electrónico de velocidad ESC (*Electronic Speed Controller*) que permite variar las revoluciones del motor, de forma que cada uno se puede hacer girar a una velocidad determinada, consiguiendo así que la aeronave pueda desplazarse en todas las direcciones, girar sobre sí misma y mantener una posición fija en el espacio.

Uno de los componentes principales de un multirrotor y que más importancia tendrá en este proyecto, es el controlador de vuelo o autopiloto. Este elemento puede considerarse el "cerebro" de la aeronave, siendo el encargado de múltiples tareas, tales como:

- Recibir, interpretar y transmitir la información de los sensores incorporados en el dron: acelerómetro, giroscopio, GPS, barómetro...
- Gestionar la señal de control enviada a los ESC de cada una de las hélices.
- Recibir las órdenes enviadas por el radiocontrol o a través del ordenador a bordo.
- Interactuar con el *payload* a bordo del vehículo (cámaras, LiDAR, gimbal...).

En este proyecto se trabajará sobre dos arquitecturas de autopiloto: DJI y Pixhawk.

A.1 DJI

La empresa de tecnología china DJI es la líder de ventas de UAVs a nivel mundial, con una cuota de mercado cercana al 80 % [16]. DJI oferta una amplia gama de drones tanto para uso particular como industrial de gran calidad y fácil uso que le han llevado a la posición que ocupan en su campo.

Para este proyecto, se trabaja con el controlador de vuelo A3, diseñado para aplicaciones industriales y que ofrece un control de vuelo preciso para tetra, hexa y octacópteros. Este autopiloto es compatible con el entorno de desarrollo *DJI Onboard SDK*, que se comentará más adelante en el capítulo de Software.

Adicionalmente, para el control manual de la aeronave, se dispone del sistema radio control DJI Lightbridge 2. Este será un elemento indispensable puesto que en todos los vuelos, aunque sean autónomos, se requerirá la presencia de un piloto por motivos de seguridad.

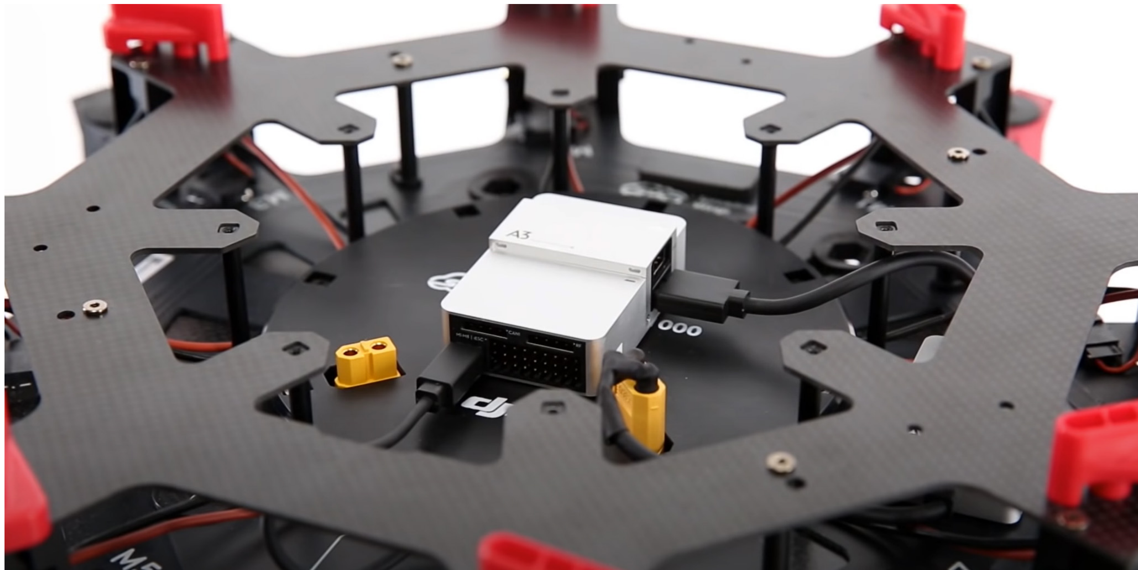

Figura A.1 Autopiloto A3.

A.2 Pixhawk

Pixhawk [17] es un proyecto open-source independiente que ofrece diseños hardware de autopilotos de bajo coste y alta calidad para su uso a nivel académico, industrial y de entretenimiento [18].

Pixhawk establece un estándar para el diseño del controlador de vuelo, definiendo un conjunto de componentes (CPU, sensores, etc) y sus conexiones. Existen distintos diseños, nombrados FMUvX (FMUv1, FMUv2, FMUv3, ...), indicando X la versión. A partir de estos, distintos fabricantes han desarrollado distintos dispositivos, cada uno de ellos con unas características que lo hacen más apropiado para según qué aplicaciones.

Figura A.2 Autopiloto Pixhawk 4 basado en FMUv5.

Este tipo de autopiloto es objeto de interés para este proyecto pues, al ser open-source, dispone de una capacidad de personalización mucho mayor que los equivalentes de DJI, lo que resulta esencial en aplicaciones de I+D+i, ya que permite el desarrollo sin restricciones además de una amplia documentación y una comunidad activa de desarrolladores.

Apéndice B
Software

A continuación, se detallará el software utilizado para el desarrollo del proyecto. Gran parte de las herramientas elegidas vienen condicionadas por el hardware utilizado, como pueden ser los SDKs (*Software Development Kit*) tanto de DJI como de Pixhawk. También, al partir el proyecto de un *framework* preexistente, se mantendrán, por ejemplo, los lenguajes de programación utilizados.

B.1 ROS

ROS (*Robot Operating System*) es un software de código abierto para el desarrollo de aplicaciones robóticas. Es usado en áreas tales como la robótica móvil, percepción, mapeo, localización o control. Dispone de una comunidad de millones de desarrolladores y usuarios que contribuyen a la creación de un ecosistema, teniendo acceso por tanto a infinidad de herramientas de libre uso.

En este proyecto, se utilizará la distribución ROS Melodic, que es la compatible con el sistema operativo Ubuntu 18.04, en el que se realizará todo el desarrollo.

Para una mejor comprensión del proyecto, conviene definir ciertos conceptos de ROS que serán nombrados en múltiples ocasiones a lo largo de este documento:

- **Nodo**: Los nodos son procesos que realizan acciones. Puede haber varios nodos en ejecución al mismo tiempo, y estos, son capaces de interaccionar entre ellos mediante topics y servicios. Aunque un mismo nodo puede realizar múltiples acciones, es recomendable distribuir las tareas entre varios de ellos para reducir la complejidad del código.
- **Topic**: Un topic es un canal de comunicación unidireccional sobre el que los nodos intercambian mensajes. Cada topic se caracteriza por un nombre (por ejemplo "/uav/speed") y el tipo de mensaje que se publica, que deberá ser definido en su creación y no podrá cambiar. Existen dos tipos de usuarios sobre un topic, los publicadores, que son los que envían mensajes a través del canal, y los suscriptores, que los reciben. Puede haber múltiples publicadores y suscriptores sobre un mismo topic, tantos topics como sea necesario y un mismo nodo podrá disponer de tantos publicadores y suscriptores como se requiera.
- **Servicio**: Un servicio es otro tipo de comunicación enfocado a la realización de peticiones. Un nodo podrá ofrecer un servicio que quedará definido por un nombre (por ejemplo "/uav/takeoff") y un tipo de mensaje con el que el nodo cliente deberá realizar la solicitud. Los servicios, a diferencia de los topics, suponen una comunicación bidireccional, pues el proveedor del servicio responderá a la solicitud una vez finalizada la tarea ejecutada asociada al servicio.
- **Mensajes**: Son las estructuras de datos utilizadas para el intercambio de información mediante topics y servicios. Un mensaje puede componerse de varios elementos de distintos tipos (cadenas de caracteres, enteros, booleanos, otros mensajes...). Existen una gran variedad de mensajes predefinidos en ROS, aunque pueden definirse mensajes personalizados atendiendo a las necesidades del usuario.
  Para los topics, se utilizan mensajes que se definen en ficheros con la extensión .msg.

```
# Single temperature reading.

Header header      # timestamp is the time the temperature was measured
                   # frame_id is the location of the temperature reading

float64 temperature # Measurement of the Temperature in Degrees Celsius

float64 variance   # 0 is interpreted as variance unknown
```

Código B.1 Estructura de mensaje sensor_msgs/Temperature.

Para los servicios, la extensión de los mensajes será .srv, y se dividirá en dos campos: *Request*, que serán los campos usados por el cliente al realizar la petición, y *Response*, que será la respuesta del proveedor. Estos dos campos se separan en el fichero mediante "- - -".

```
bool data # e.g. for hardware enabling / disabling
---
bool success  # indicate successful run of triggered service
string message # informational, e.g. for error messages
```

Código B.2 Estructura de mensaje std_srvs/SetBool.

- **Callback**: un callback es una función que se ejecuta al ocurrir un evento. En el caso de ROS habrá dos tipos de callback:
  - Un nodo suscriptor ejecutará la función de callback asociada a un topic cada vez que se reciba un nuevo mensaje por este. De este modo, se podrá procesar el mensaje realizando la acción conveniente, como por ejemplo, actualizar el valor de una variable con el nuevo dato obtenido.
  - Un nodo que provee un servicio, ejecutará el callback asociado a este cuando se le realice una llamada. Por ejemplo, se ofrece un servicio que recibe dos enteros, en el callback se suman y se envía como respuesta el resultado.
- **Paquete**: Los paquetes son la forma en que se organiza el software de ROS. Un paquete consiste, por lo general, en un conjuntos de nodos que cumplen una función. Tienen la ventaja de que son fáciles de reutilizar en cualquier proyecto. Por ejemplo, si se quieren implementar algoritmos de SLAM, sería relativamente sencillo descargar un paquete ya creado por la comunidad e instalarlo en nuestro *workspace*.

B.2 Entornos de desarrollo y comunicación

B.2.1 DJI Onboard SDK

DJI Onboard SDK (OSDK) es un kit de desarrollo de software oficial de DJI que permite conectar el ordenador a bordo a un vehículo o autopiloto DJI usando el puerto serie. Se tratará de un componente esencial para poder comunicar el *framework* con la aeronave, y, de esta forma, habilitar el intercambio de información.

B.2.2 DJI Onboard SDK ROS

DJI Onboard SDK ROS (OSDK-ROS) se trata de un paquete que actúa de interfaz entre ROS y el OSDK. Su ejecución pone a disposición una serie de topics y servicios que permitirán la comunicación con la aeronave a través de ROS, lo que facilita el desarrollo en gran medida.

B.2.3 Mavlink

Mavlink es un protocolo de comunicación ligero para drones basado en el método publicador/suscriptor a través de topics. Es el protocolo utilizado en Pixhawk.

B.2.4 Mavros

Mavros es un paquete que proporciona un driver de comunicación entre ROS y Mavlink. De forma similar al OSDK-ROS, publicará una serie de topics y servicios que permitirá la conexión de ROS con el autopiloto de forma sencilla.

B.3 CATEC GCS

Una *Ground Control Station* es un sistema que permite el control de un UAV desde un ordenador en tierra, permitiendo mandar comandos de movimiento, la creación y ejecución de misiones, la consulta de la telemetría y muchas otras funciones que pueden variar en función de las capacidades del dispositivo.

En CATEC, se desarrolló el software de una GCS para ser usada en los proyectos, permitiendo así una personalización total. Para este proyecto, se utilizará una versión que permite visualizar los estados de las máquinas de estado del *framework* y las llamadas a los servicios ofrecidos por estas, tales como el despegue/aterrizaje, carga/descarga/ejecución de misiones y cambios del modo de control.

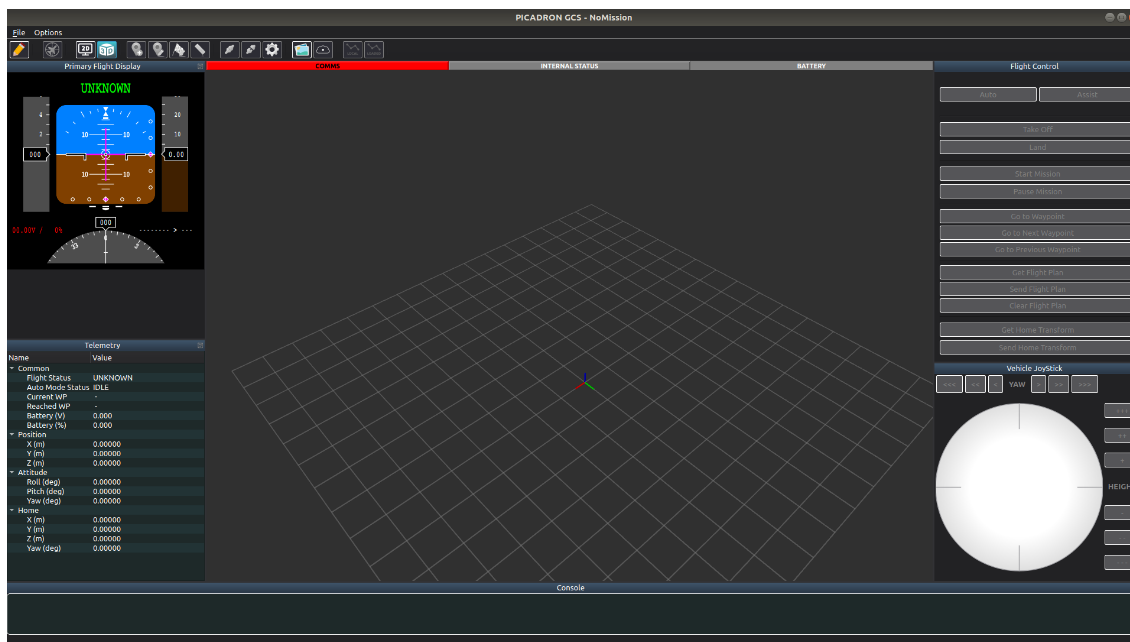

Figura B.1 Interfaz gráfica de la GCS de CATEC.

Apéndice C
Framework preexistente

Este proyecto parte del *framework* para navegación autónoma desarrollado en CATEC. El primer paso será, por tanto, realizar un análisis en profundidad de la estructura del código para entender su funcionamiento, de forma que, más adelante, se pueda ampliar el código sin interferir en las funcionalidades ya existentes, aprovechando aquellas herramientas ya implementadas y adaptándose a la filosofía de programación utilizada.

El objetivo del sistema es poder llevar a cabo misiones con UAVs de forma similar a las herramientas nativas que ofrecen los propios fabricantes de los drones, pero sin depender de la señal GPS. Por tanto, el *framework* que se desarrolló es capaz de controlar el estado de la aeronave, realizar maniobras de despegue/aterrizaje y ofrece la posibilidad de diseñar y ejecutar misiones.

Esto se consigue gracias a tres máquinas de estados, que, ordenadas de mayor a menor prioridad, son:

- **Safety Manager**: permite alternar entre modo manual y automático.
- **Main State Machine**: contempla los distintos estados posibles en los que se puede encontrar la aeronave.
- **Auto State Machine**: permite llevar a cabo las misiones de seguimiento de *waypoints*.

A continuación, se entrará en detalle en el funcionamiento de cada una de ellas, sus diferentes estados, la información que leen y publican a través de los topics y los servicios que ofrecen.

Cabe destacar que el *framework* en sí no envía directamente las órdenes de movimiento al dron, sino que envía las referencias de posición a un control externo, implementado en MATLAB, que es el encargado de calcular las referencias de velocidad necesarias para que el UAV alcance el objetivo.

C.1 Safety Manager

Su función principal es determinar quién tiene la autoridad del UAV. El concepto de autoridad en DJI consiste en alternar entre el modo manual, en el que el dron responde ante las señales recibidas del mando radio control, y el modo automático, en el cual a través de un ordenador se le pueden enviar referencias de posición o velocidad.

Además, como su nombre indica, también realiza funciones de seguridad, que consisten en comprobar que ciertos topics se estén publicando a la frecuencia esperada, con el fin de asegurar que no existen fallos en el dron o en la comunicación.

El lenguaje utilizado en este módulo es C++.

En los siguientes apartados se analizarán los distintos bloques que componen este programa.

C.1.1 Parámetros

Existe un fichero .yaml de configuración, en el cual se definen ciertos parámetros que se cargarán en el servidor de parámetros de ROS al ejecutar la aplicación, de forma que los distintos nodos puedan acceder a ellos. La ventaja principal de utilizar este tipo de archivos reside en que los valores de las variables pueden ser cambiados sin necesidad de recompilar el proyecto.

Los parámetros definidos en el fichero son:

- ***safety_margin***: define una tolerancia a la hora de determinar el tamaño del buffer esperado al hacer la comprobación del número de mensajes recibidos desde un topic.

- ***p_mode_selector_min*** y ***p_mode_selector_max***: determina los valores numéricos de las posiciones del *switch* del radio control asociado al modo de vuelo seleccionado. Este valor dependerá del modelo del mando que se conecte, pues distintos modelos asocian a cada posición un valor distinto.
- ***set_authority_min*** y ***set_authority_max***: indica los valores de las posiciones del *switch* que se utiliza para alternar la autoridad. Este valor depende también del modelo del mando.
- ***topics***: define una lista con los topics a los que el nodo se suscribirá para comprobar que se reciben y la frecuencia a la que se debería recibir.

Estos parámetros serán leídos por la clase **Parameters** y almacenados en variables públicas para que puedan ser accedidas por otros módulos del sistema.

C.1.2 Comprobación de topics

Una de las funciones del Safety Manager es comprobar que en ciertos topics se estén publicando mensajes a la frecuencia esperada, ya que, en caso contrario, puede ser indicativo de la presencia de algún error en el sistema o en las comunicaciones. Estos topics serán los indicados en el archivo de configuración, y pueden ser, por ejemplo, la información de la batería, de la IMU o del LiDAR. Esta comprobación será posible gracias a tres clases: **SubscriberChecker**, **TopicChecker** y **TopicsCheckerManager**.

SubscriberChecker

Esta clase es la encargada de verificar la frecuencia de publicación de los topics. Para ello, dispone de un buffer en el que se van almacenando los mensajes que llegan, y, cuando se llama al método **checkData**, calcula cuál es el tamaño que debería tener el buffer en función del tiempo que ha transcurrido y la frecuencia a la que se debería emitir, añadiendo además cierta tolerancia con el parámeotro *safety_margin*.

$$expected_buffer_length = \Delta t \cdot freq \cdot safety_margin \tag{C.1}$$

Si el buffer está vacío, devolverá un *false* y mostrará un mensaje de error. Si es menor que el esperado, se imprimirá un *warning* y se devolverá *true*. Por último, si es mayor o igual, se devuelve *true*. Luego, independientemente del resultado, se vacía el buffer, permitiendo poder realizar una nueva comprobación en el futuro.

TopicChecker

La función principal de estas clases es añadir al buffer heredado de **SubscriberChecker** los mensajes que lleguen del topic al que se encuentren suscritos.

Existen dos tipos:

- **Generales**: se suscriben al topic que se les indique y van almacenando en el buffer los mensajes recibidos. Se declarará una instancia de la clase **GeneralTopicChecker** por cada topic que se especificó en el fichero de configuración.
- **Específicos**: realizan la misma función que los anteriores, pero además, analizan la información del mensaje que ha llegado. Existen dos clases de este tipo:
  - **RcTopicChecker**: se suscribe al topic del radio control ("/dji_sdk/rc"). El objetivo de leer la señal del mando es poder alternar la autoridad del dron a través de este. Se define una enumeración, **RcStatus**, con los distintos estados en los que puede estar el RC, siendo estos: UNKNOWN, MANUAL y AUTO.
    En dicho topic se reciben mensajes de tipo *Joy*, el cual se compone de dos vectores: *axes* y *buttons*. En el *callback*, se comprueba el valor de los siguientes elementos:
    - ◇ **axes[4]**: se corresponde con el *switch* de selección del modo de la aeronave (ver Figura C.1). Este deberá estar en modo P, ya que es el aquel en el cual la autoridad puede cederse al ordenador. Si este valor no se corresponde con el modo P, el estado será MANUAL. Si el valor leído se corresponde con *p_mode_selector_max*, se considera que el modo P está activo, y por tanto, se podrá activar el modo AUTO.

Figura C.1 Selector de modos de vuelo en Lightbridge.

◇ **axes[5]**: nuevamente, indica el valor de un *switch* (ver Figura C.2). En este caso, la posición en la que se encuentre el interruptor no tiene un efecto directo sobre el dron. Si se cumple la condición del estado P y el modo MANUAL es el activo, se cambiará a modo AUTO cuando se detecte un flanco de subida en el *switch*. Si ya en modo AUTO, se detecta un flanco de bajada, se pasará a MANUAL. Se comparará con los parámetros *set_authority_min* y *set_authority_max* para detectar los flancos de subida y bajada.

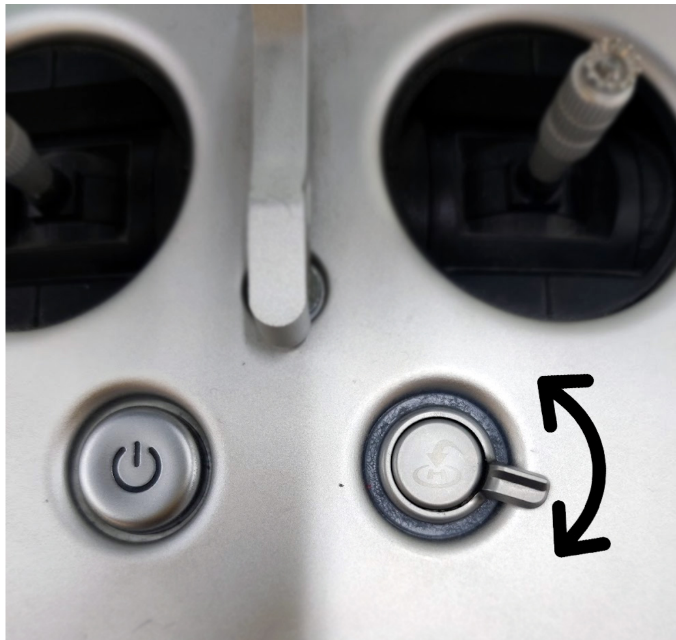

Figura C.2 Selector de autoridad en Lightbridge.

– **DeviceTopicChecker**: se suscribe al topic que indica quién tiene la autoridad del dron ("/dji_-sdk/device_status"). Se define una enumeración, **DeviceStatus**, con los posibles estados, los cuales son: UNKNOWN, MANUAL, APP y SERIAL. De los anteriores estados, los más interesantes serán MANUAL (control con mando) y SERIAL (control con ordenador a bordo).

TopicsCheckerManager

Por último, se define la clase **TopicsCheckerManager**, que recibe un vector de **SubscriberCheckers** y el parámetro *safety_period*. Con el periodo especificado en este último elemento, se configura un temporizador,

que cada vez que se activa, ejecuta un *callback*, donde se llama al método **checkData** de cada **Subscriber-Checker** del vector recibido. En caso de que alguno devuelva *false*, se pondrá la variable **_data_status** a *false* para indicar que existe un error.

C.1.3 Máquina de estados

La máquina de estados que gobierna el control de la autoridad queda definida en la clase **AuthorityStateMachine**. Dispone de tres estados:

- UNINITIALIZED: es el estado inicial. Se esperará a que **_data_status** esté a *true* y que **RcStatus** sea distinto de UNKNOWN. Cuando esto se cumpla, significará que se ha inicializado correctamente el sistema y se procederá a consultar el valor de **DeviceStatus** para determinar donde reside la autoridad.
- MANUAL: **DeviceStatus** está en MANUAL, por tanto, solo se podrá controlar con la emisora. Se comprobará el estado de **RcStatus**, y en caso de que cambie a AUTO, se pedirá la autoridad llamando al servicio "dji_sdk/sdk_control_authority" y en caso de éxito, se cambiará al estado CONTROL.
- CONTROL: el UAV se encuentra en modo SERIAL, en el cual, el ordenador a bordo tiene la autoridad. En cada iteración se realizarán tres comprobaciones: que **_data_status** esté a *true*, **RcStatus** en AUTO y **DeviceStatus** en SERIAL. Si alguna de estas condiciones no se cumple, se liberará la autoridad y se cambiará a MANUAL.

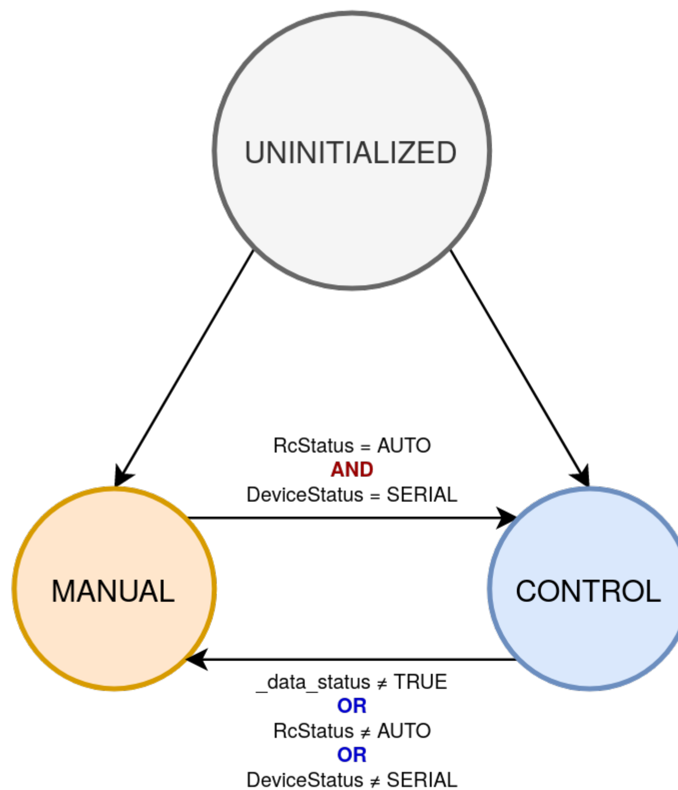

Figura C.3 Diagrama de la máquina de estados de autoridad.

C.1.4 Inicializador del nodo

La clase **DjiSafetyManagerNode** inicializa el nodo e instancia las clases anteriores. Primero, **Parameters** es creada para poder acceder a los parámetros, necesarios para el siguiente paso, que será declarar los siguientes elementos, los distintos **TopicCheckers**, para finalmente, inicializar la máquina de estados, que tendrá acceso a los métodos de todas las clases gracias a que recibirá los punteros a estas como argumento.

C.2 Main State Machine

Esta segunda máquina de estados contempla los distintos estados de vuelo de la aeronave. El lenguaje empleado es Python y se implementa la librería smach[19], que permite de forma sencilla definir una máquina de estados con sus posibles transiciones. Todo esto queda definido en la clase **DjiMainStateMachine**, que se configura como se puede observar la Figura C.4.

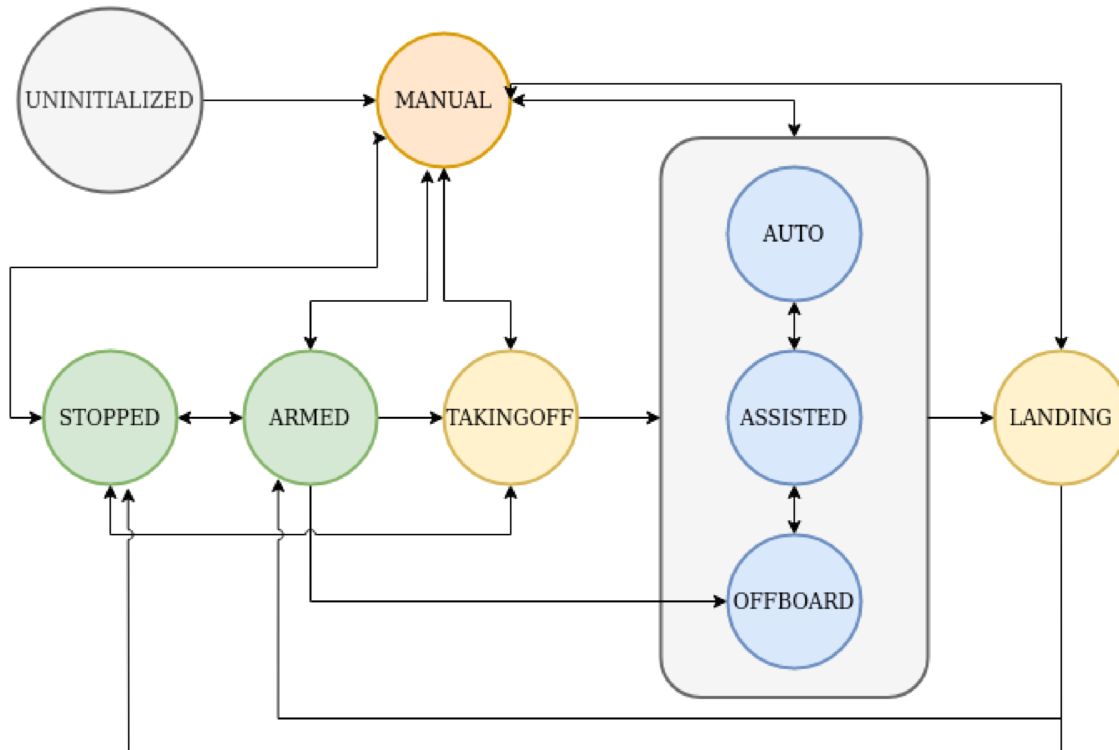

Figura C.4 Diagrama de la máquina de estados Main.

Una de las características de smach, es que en cada estado llama a una clase para su ejecución, lo que nos permite poder realizar distintas acciones según el estado activo. A continuación se detallará la función de cada uno.

C.2.1 Estados

UNINITIALIZED

Es el estado inicial de la máquina. En él, se espera a que el estado de Safety Manager sea MANUAL, y cuando esto ocurra, cambia a MANUAL.

MANUAL

Siempre que la autoridad resida en el radio control, la máquina de estados se encontrará en este estado, en el que únicamente se comprobará si hay un cambio de autoridad para saltar a alguno de los otros estados.

STOPPED

El dron se encuentra en el suelo y los motores no están armados.

ARMED

El dron se encuentra en el suelo y los motores están armados. En este estado, estará disponible la llamada al servicio para solicitar un despegue, que dará lugar a un salto a TAKINGOFF.

TAKINGOFF

Se iniciará la maniobra de despegue. Para ello, se obtiene la posición actual del UAV y se le sumará al eje z la altura de despegue, la cual se especifica en la llamada al servicio. Después, esta posición será publicada como referencia para el control. Cuando se detecte que la altura se ha alcanzado, se saltará a uno de los modos de control: AUTO, ASSISTED u OFFBOARD.

AUTO

En este modo de vuelo, se publicará una referencia de control tanto de posición (xyz) como de orientación (xyzq). Estas poses son determinadas por la máquina de estados Auto State Machine que se explicará más adelante. Este estado solamente se encarga de publicarlas al topic correspondiente.

ASSISTED

El modo asistido permite el control de la aeronave con un joystick, ya sea periférico o el disponible en la GCS. Este tipo de control es similar al manual, pero más sencillo.

El estado en sí no realiza ninguna acción, los comandos de movimiento son gestionados por el control de MATLAB.

OFFBOARD

OFFBOARD es un estado que no realiza ninguna acción específica. El objetivo de este es que, si para alguna aplicación se necesita realizar tareas específicas que no se encuentran contempladas en la máquina de estados, como puede ser, por ejemplo, el seguimiento de un objetivo, estas se ejecuten cuando este estado se encuentre activo.

LANDING

Este estado entrará en acción cuando, estando en uno de los tres modos de vuelo anteriores, se haga una llamada al servicio de aterrizaje. Este estado no comandará directamente al UAV una referencia de posición, si no que el control de MATLAB será el encargado de realizar esta maniobra cuando detecte que está activo.

C.2.2 Servicios

Para poder ofrecer los servicios necesarios, se crea la clase **DjiMainStateMachineSrvs**. Algunos de los servicios implementados no utilizan mensajes propios de ROS, si no que se han tenido que crear atendiendo a las necesidades y están disponibles en el paquete “dji_main_state_machine_msgs”.

La Tabla C.1 resume los servicios ofrecidos.

Tabla C.1 Servicios publicados por Main State Machine.

Nombre del servicio	Mensaje	Descripción
dji_main_state_machine/set_mode	SetMode	Selecciona modo de vuelo.
dji_main_state_machine/takeoff	Takeoff	Solicita un despegue especificando la altura.
dji_main_state_machine/land	Trigger	Solicita un aterrizaje.
dji_main_state_machine/set_control_ref	SetControlRef	Actualiza la referencia de control.

Cada servicio dispondrá de un *callback* donde se gestionará la petición recibida. No todos los servicios se podrán llamar siempre, dependerá del estado en el que se encuentre el sistema. En la Tabla C.2 se refleja la compatibilidad estado-servicio.

Tabla C.2 Tabla de compatibilidad estado-servicio.

	/set_mode	/takeoff	/land	/set_control_ref
UNINITIALIZED	OK	ERROR	ERROR	ERROR
STOP	OK	ERROR	ERROR	ERROR
ARMED	OK	OK	ERROR	ERROR
TAKEOFF	OK	ERROR	ERROR	ERROR
AUTO	OK	ERROR	OK	OK
ASSISTED	OK	ERROR	OK	ERROR
OFFBOARD	OK	ERROR	OK	ERROR
LAND	OK	ERROR	ERROR	ERROR
MANUAL	OK	ERROR	ERROR	ERROR

C.2.3 Topics

A través de la clase **DjiMainStateMachineTopics**, el proceso podrá tanto como publicar como suscribirse a distintos topics, ya sea para enviar o recibir información necesaria para la ejecución.

Tabla C.3 Topics publicados por Main State Machine.

Nombre del topic	**Mensaje**	**Descripción**
dji_main_state_machine/status	UInt8	Publica el estado de la máquina de estados.
dji_main_state_machine/mode	UInt8	Publica el modo de control.
dji_main_state_machine/auto_control_ref	PoseStamped	Publica la referencia de control

Tabla C.4 Topics suscritos por Main State Machine.

Nombre del topic	**Mensaje**	**Descripción**
dji_safety_manager/status	UInt8	Recibe el estado de la maquina de estados de Safety Manager.
dji_sdk/flight_status	UInt8	Lee del topic publicado por el dron sobre el estado del vuelo.
uav_main_control/pose	PoseStamped	Recibe la ubicación del UAV.

C.3 Auto State Machine

Esta última máquina de estados, al igual que la anterior, esta codificada en Python y utiliza smach para su configuración. El objetivo de esta es poder ejecutar misiones de seguimiento de *waypoints* cuando el modo de control de la aeronave es AUTO.

El diagrama de la máquina de estados será el reflejado en la Figura C.5.

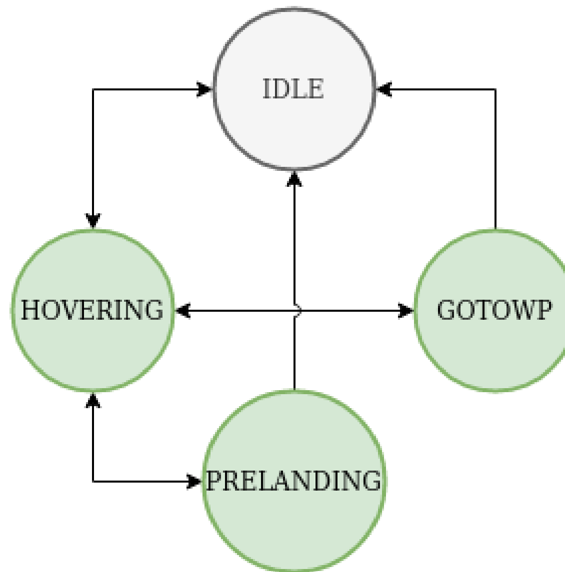

Figura C.5 Diagrama de la máquina de estados Auto.

C.3.1 Estados

IDLE

Como su propio nombre indica (inactivo), mientras este estado esté activo, no se realizará ninguna acción. Esto ocurrirá cuando Main State Machine esté en un estado distinto de AUTO. Cuando cambie a AUTO, se saltará a HOVERING.

HOVERING

Hovering se puede traducir como "permanecer inmóvil en el aire". En este estado, se establecerá como referencia de control la pose actual del dron para que este se mantenga quieto en esa posición.

GOTOWP

En este estado, se llevará a cabo la tarea de seguimiento de *waypoints*. El proceso podrá recibir a través de un servicio una lista de *waypoints* a los que navegar. Al dron se lo podrá indicar que vaya a uno de los puntos en dos modos:

- STOP: se desplaza hasta la posición indicada y pasa a modo HOVERING una vez alcanzado.
- CONTINUE: se desplaza hasta la posición indicada y una vez alcanzada, navega hasta el siguiente *waypoint* en la lista, hasta llegar al último.

El propio estado es el encargado de establecer la referencia llamando al servicio "dji_main_state_machine/set_control_ref" y de comprobar cuándo se ha alcanzado esta.

PRELANDING

Prelanding ofrece un paso previo al aterrizaje en sí, dejando la aeronave a una altura cercana al suelo. Para ello, primero mide la distancia al suelo mediante un altímetro láser, que normalmente se sitúa en el tren de aterrizaje, luego, calcula la distancia que hay que descender para situarse a la altura deseada (1 metro del suelo por defecto) y finalmente obtiene la posición actual del dron, le resta esta distancia en el eje z y establece como referencia de control ese punto.

Una vez alcanzado, se llama al servicio "dji_main_state_machine/land", que finalizará la maniobra de aterrizaje.

C.3.2 Servicios

Los servicios disponibles son declarados en la clase **DjiAutoStateMachineSrvs**. De igual manera que en Main State Machine, se han creado mensajes personalizados que están contenidos en “dji_auto_state_-machine_msgs”.

Tabla C.5 Servicios publicados por Auto State Machine.

Nombre del servicio	**Mensaje**	**Descripción**
dji_auto_state_machine/set_flight_plan	SetFlightPlan	Permite cargar un plan de vuelo.
dji_auto_state_machine/get_flight_plan	GetFlightPlan	Solicita que se envíe el plan de vuelo cargado.
dji_auto_state_machine/go_to_waypoint	GoToWaypoint	Indica el *waypoint* al que se quiere que navegue y el modo (STOP o CONTINUE).
dji_auto_state_machine/stop_mission	Trigger	Detiene la misión que se esté ejecutando.
dji_auto_state_machine/set_home_tf	SetHomeTf	Envía la posición del *home* establecido.
dji_auto_state_machine/get_home_tf	GetHomeTf	Solicita que se envíe el *home* establecido.
dji_auto_state_machine/preland	Trigger	Solicita que se ejecute el prelanding.

La disponibilidad de estos servicios estará sujeta al estado de la máquina de estados.

Tabla C.6 Tabla de compatibilidad servicio-estado.

	IDLE	HOVERING	GOTOWP	PRELANDING
/set_flight_plan	OK	OK	ERROR	OK
/get_flight_plan	OK	OK	OK	OK
/go_to_waypoint	ERROR	OK	OK	ERROR
/stop_mission	ERROR	ERROR	OK	ERROR
/set_home_tf	OK	OK	ERROR	OK
/get_home_tf	OK	OK	OK	OK
/preland	OK	OK	OK	ERROR

C.3.3 Topics

La clase **DjiAutoStateMachineTopics** contendrá los suscriptores y publicadores de topics.

Tabla C.7 Topics publicados por Auto State Machine.

Nombre del topic	Mensaje	Descripción
dji_auto_state_machine/current_wp_index	UInt16	Publica el índice del *waypoint* al que se está navegando.
dji_auto_state_machine/reached_wp_index	UInt16	Publica el índice del último *waypoint* alcanzado.
dji_auto_state_machine/status	UInt8	Publica el estado de la máquina de estados.

Tabla C.8 Topics suscritos por Auto State Machine.

Nombre del topic	Mensaje	Descripción
dji_main_state_machine/status	UInt8	Lee el estado de de Main State Machine.
lightware_range	Range	Lee la medida del altímetro.

C.3.4 Concepto de *home* y transformación de coordenadas

En el apartado de servicios, se han comentado dos servicios relacionados con el *home*. El *home* es el origen del sistema de coordenadas de la aeronave, "/odom_r". Normalmente este punto coincide con el punto de despegue, aunque puede ser modificado llamando al servicio "/set_home_tf".

Al enviar el plan de vuelo desde la GCS, los *waypoints* están referenciados al sistema de coordenadas del mapa ("/map"). Para el control de la aeronave, se necesitará que el sistema de coordenadas sea "/odom_r", por tanto, al conocer la posición del *home* en el mapa, se pueden transformar los *waypoints* a este sistema.

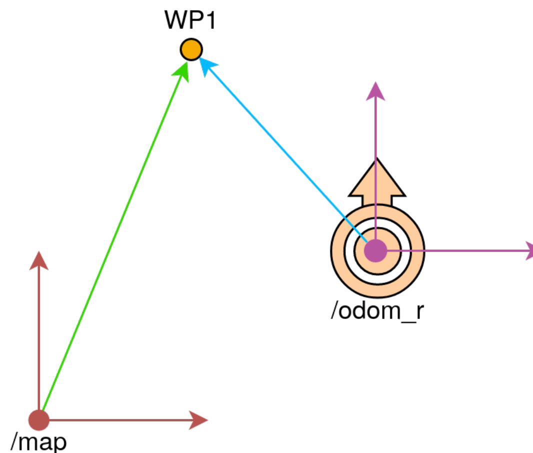

Figura C.6 *Waypoint* en sistema de coordenadas /map y /odom_r.

Así pues, cada vez que se actualice el plan de vuelo, internamente el proceso realizará la transformación entre ambos sistemas.

Índice de Figuras

1.1	Visualización de *waypoints* en la app de DJI	2
1.3	DJI Matrice 600 equipado con LiDAR Velodyne VLP 16	2
1.2	Dron para inspección de superficies por contacto, proyecto AEROARMS H2020	3
1.4	CATEC	3
2.1	Conexionado del simulador	5
2.2	Clases presentes en dji_framework_simulator	6
2.3	Interfaz de usuario	7
2.4	Composición de mensaje PoseStamped con PointStamped y QuaternionStamped	9
2.5	Sistemas de referencia FLU y ENU	9
2.6	Adaptación de la referencia de control	12
2.7	Flags para control	13
2.8	Composición de mensaje Joy para modo ASSISTED	14
3.1	Ejemplo de Ray-Casting	21
3.2	Saltos a/desde FAILSAFE	25
3.3	Interrupción de RTH por failsafe prioritario	25
3.4	Main State Machine con nuevos estados	27
4.1	Simulación PX4 en Gazebo	31
6.1	Plan de vuelo	38
6.2	Dron utilizado en las pruebas de vuelo	40
6.3	Cámaras Vicon en Testbed	40
6.4	Visualización de la posición del dron con el software de Vicon	41
6.5	Prueba de vuelo RC1 y RC2	43
6.6	Prueba de vuelo RC3	44
6.7	Prueba de vuelo FAIL3	45
A.1	Autopiloto A3	50
A.2	Autopiloto Pixhawk 4 basado en FMUv5	50
B.1	Interfaz gráfica de la GCS de CATEC	53
C.1	Selector de modos de vuelo en Lightbridge	57
C.2	Selector de autoridad en Lightbridge	57
C.3	Diagrama de la máquina de estados de autoridad	58
C.4	Diagrama de la máquina de estados Main	59
C.5	Diagrama de la máquina de estados Auto	62
C.6	*Waypoint* en sistema de coordenadas /map y /odom_r	64

Índice de Tablas

2.1 Flags usadas 13
2.2 Servicios llamados por los métodos 15

4.1 Incompatibilidades 29

6.1 Tabla de compatibilidad estado-servicio 39

C.1 Servicios publicados por Main State Machine 60
C.2 Tabla de compatibilidad estado-servicio 61
C.3 Topics publicados por Main State Machine 61
C.4 Topics suscritos por Main State Machine 61
C.5 Servicios publicados por Auto State Machine 63
C.6 Tabla de compatibilidad servicio-estado 63
C.7 Topics publicados por Auto State Machine 64
C.8 Topics suscritos por Auto State Machine 64

Índice de Códigos

2.1 Código de la clase JoystickPublisher 8
2.2 Extracto de la clase PoseConverter 10
2.3 Extracto de la clase RefConverter 12
2.4 Código de la clase GCSJoyConverter 14
2.5 Código de la clase UserInterface 15

3.1 Callback del topic /rc_connection_status 20
3.2 Mensaje SetGeocage.srv 21
3.3 Callback del topic /uav_main_control/pose 21
3.4 Extracto de la clase RTH 23
3.5 Mensaje ReqRTH.srv 24
3.6 Clase FAILSAFE 26

4.1 Temporizador para comprobación de desconexión 30

5.1 Configuración de servicio con macros 33
5.2 Extracto de CMakeList.txt para selección de flag 33
5.3 Ejemplo de elección de mensajes a incluir 33
5.4 Ejemplo de suscripciones 34
5.5 Ejemplo de elección de *callback* 34
5.6 Selección de tipo de detección de desconexión del mando 34

B.1 Estructura de mensaje sensor_msgs/Temperature 52
B.2 Estructura de mensaje std_srvs/SetBool 52

Bibliografía

[1] R. A. Española. (s.f) Diccionario de la lengua española. [Online]. Available: https://dle.rae.es/dron

[2] "Sistemas de aeronaves no tripuladas (uas)," Organización de Aviación Civil Internacional, Circular 328, 2012.

[3] "La demanda de operaciones de drones aumentó en españa en un 370 %," *Actualidad Aeroespacial*. [Online]. Available: https://actualidadaeroespacial.com/la-demanda-de-operaciones-de-drones-aumento-en-espana-en-un-370/

[4] CATEC. Presentación. [Online]. Available: http://www.catec.aero/es/presentación

[5] ROS. dji_sdk. [Online]. Available: http://wiki.ros.org/dji_sdk

[6] Dji - returning home. [Online]. Available: https://developer.dji.com/mobile-sdk/documentation/introduction/component-guide-flightController.html#returning-home

[7] Px4 - safety configuration (failsafes). [Online]. Available: https://docs.px4.io/v1.12/en/config/safety.html

[8] Ardupilot - radio failsafe. [Online]. Available: https://ardupilot.org/copter/docs/radio-failsafe.html

[9] P. Cásek, "Scenario Definition Document," Technische Universität Braunschweig, Tech. Rep., 01 2019.

[10] Point-in-polygon: Jordan curve theorem. [Online]. Available: https://sidvind.com/wiki/Point-in-polygon:_Jordan_Curve_Theorem

[11] mavros_msgs/extendedstate message. [Online]. Available: http://docs.ros.org/en/lunar/api/mavros_msgs/html/msg/State.html

[12] mavros_msgs/extendedstate message. [Online]. Available: http://docs.ros.org/en/api/mavros_msgs/html/msg/ExtendedState.html

[13] Flame wheel arf kit. [Online]. Available: https://www.dji.com/es/flame-wheel-arf

[14] Orbitty carrier for nvidia® jetson™ tx2/tx2i. [Online]. Available: https://connecttech.com/product/orbitty-carrier-for-nvidia-jetson-tx2-tx1/

[15] vicon_bridge - ros wiki. [Online]. Available: http://wiki.ros.org/vicon_bridge

[16] Droneii. (2021) The best drone manufacturers in 2021. [Online]. Available: https://droneii.com/the-best-drone-manufacturers-in-2021

[17] Pixhawk. Pixhawk | the hardware standart for open-source autopilots. [Online]. Available: https://pixhawk.org/

[18] PX4. Pixhawk series. [Online]. Available: https://docs.px4.io/master/en/flight_controller/pixhawk_series.html

[19] Smach. smach. [Online]. Available: http://library.isr.ist.utl.pt/docs/roswiki/smach.html