



Universidad Politécnica
de Madrid



**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

**Comportamientos de Robots Aéreos
para Operación Coordinada en Misiones
Multi-agente**

Autor: Rafael Martín Lesmes

Tutor: Martín Molina González

Madrid, Junio 2021

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Título: Comportamientos de Robots Aéreos para Operación de Coordinada en
Misiones Multi-agente

Junio 2021

Autor: Rafael Martín Lesmes

Tutor: Martín Molina González

Departamento de Inteligencia Artificial

ETSI Informáticos

Universidad Politécnica de Madrid

Agradecimientos

A Martín Molina y a Pablo Santamaría por proponerme este trabajo y orientarme en su realización.

A mi familia y amigos, y especialmente a Adrián Salas por tener que soportarme durante toda la carrera.

Resumen

El propósito de este Trabajo de Fin de Grado es el de desarrollar comportamientos de robots aéreos que deben coordinarse para realizar misiones de forma conjunta (multi-agente). El trabajo incluye el diseño, programación y evaluación de dichos comportamientos mediante misiones visualizadas empleando el simulador de robots Gazebo. El diseño y la implementación se han realizado con la ayuda del entorno de desarrollo software de robótica aérea Aerostack.

Abstract

The purpose of this Final Degree Project is to develop behaviors of aerial robots that must be coordinated to perform missions jointly (multi-agent). The paper includes the design, programming and evaluation of these behaviors through visualised missions using Gazebo, a robot simulator. The design and implementation has been carried out with the help of Aerostack, a software development environment oriented to aerial robotics.

Índice

1	Introducción.....	1
1.1	Objetivos del trabajo	1
1.2	Organización de la memoria	1
2	Antecedentes del trabajo.....	3
2.1	Robot Operating System (ROS).....	3
2.1.1	El sistema de ficheros de ROS	3
2.1.2	El grafo computacional.....	4
2.1.3	Comunicación entre nodos	5
2.2	Gazebo.....	6
2.3	Aerostack.....	7
2.3.1	Arquitectura.....	7
2.3.2	Behaviors	8
2.3.3	Características	9
2.3.4	Intérpretes de misiones	9
2.4	Quadrotor PID Controller	10
3	Diseño y programación.....	11
3.1	Visión general	11
3.2	Follow Target With PID Control	14
3.3	Move Away From Robot With PID Control.....	17
3.4	Reach Target Altitude With PID Control.....	20
3.5	Take A Look At Target With PID Control.....	23
3.6	Get Close To Target With PID Control.....	26
3.7	Keep Look At Target With PID Control.....	29
3.8	Keep Target Altitude With PID Control.....	31
4	Pruebas	34
4.1	Pruebas de ejecución individual	34
4.1.1	Pruebas de ejecución individual: Follow Target With PID Control	34
4.1.2	Pruebas de ejecución individual: Move Away From Robot With PID Control.....	39
4.1.3	Pruebas de ejecución individual: Reach Target Altitude With PID Control	44
4.1.4	Pruebas de ejecución individual: Take A Look At Target With PID Control.....	49
4.1.5	Pruebas de ejecución individual: Get Close To Target With PID Control.....	52
4.1.6	Pruebas de ejecución individual: Keep Look At Target With PID Control.....	58
4.1.7	Pruebas de ejecución individual: Keep Target Altitude With PID Control	63
4.2	Prueba de ejecución integrada en misiones aéreas	68
4.3	Pruebas de rendimiento	69
4.4	Dimensión del software desarrollado	74
5	Conclusiones	76
6	Bibliografía	77
7	Anexos	78
7.1	Anexo 1	78

7.2	Anexo 2	79
7.3	Anexo 3	81
7.4	Anexo 4	82
7.5	Anexo 5	84
7.6	Anexo 6	85
7.7	Anexo 7	86
7.8	Anexo 8	88
7.9	Anexo 9	89
7.10	Anexo 10.....	92
7.11	Anexo 11.....	96
7.12	Anexo 12.....	98

1 Introducción

Hoy en día, la industria de los vehículos aéreos no tripulados o drones está experimentando un gran auge debido al uso de este tipo de dispositivos en el ámbito civil debido al fácil acceso que tienen los particulares y empresas para adquirir una de estas aeronaves. Además, el desarrollo tecnológico, la fabricación barata de componentes y la facilidad de operar los drones también están contribuyendo a su expansión.

La gran importancia que están adquiriendo los robots aéreos reside en la capacidad de estas aeronaves en realizar tareas que suponen un gran coste económico o que ponen en riesgo las vidas de personas, como puede ser el caso de evaluar la situación de un incendio, inspeccionar una planta de energía térmica o manipular materiales nocivos.

Debido a la necesidad de que los drones realicen misiones cada vez más complejas, con la menor intervención humana posible y que conlleven la interacción entre múltiples robots resulta conveniente e interesante disponer de software para realizar este tipo de tareas, y precisamente, el presente trabajo se centra en este campo.

Este trabajo se ha planteado como parte de los objetivos del grupo de investigación Computer Vision and Aerial Robotics (CVAR) con el fin de incrementar la autonomía de los robots aéreos haciendo uso de técnicas de inteligencia artificial.

1.1 Objetivos del trabajo

El objetivo general del trabajo es el de construir y evaluar arquitecturas de control de robots aéreos, los cuales deben coordinarse para realizar misiones de forma conjunta.

Los objetivos de este trabajo son:

- Estudio de las herramientas software, especialmente ROS, Aerostack y el simulador Gazebo.
- Análisis del problema de coordinación de robots aéreos.
- Diseño y programación de los comportamientos haciendo uso de la forma de programación empleada en el entorno software Aerostack.
- Evaluación de los comportamientos mediante la ejecución de misiones en escenarios realistas haciendo uso del simulador de robots Gazebo

1.2 Organización de la memoria

La memoria está dividida en siete secciones principales.

La primera sección contiene la introducción del proyecto y la organización de la memoria.

El segundo capítulo titulado “Antecedentes del trabajo” se realiza un análisis del software, sistemas y técnicas de las que se ha hecho uso durante el desarrollo de este trabajo.

La sección “Diseño e implementación de los behaviors”, hace hincapié en el desarrollo de los distintos behaviors realizados, así como de los topics y servicios empleados en su programación.

En la cuarta sección se presentan los distintos casos de prueba por los que han pasado cada uno de los behaviors desarrollados.

El capítulo cinco concluye el trabajo analizando la cumplimentación de los objetivos del trabajo.

Finalmente, en las secciones seis y siete se presentan las referencias usadas de las que se ha hecho uso y los anexos a los que se les irá haciendo mención a lo largo de la memoria.

2 Antecedentes del trabajo

En este apartado, se describen las principales herramientas software con las que se ha trabajado, así como el contexto general en el que se ha realizado el trabajo.

2.1 Robot Operating System (ROS)

Robot Operating System (ROS) es un middleware de código abierto que fue diseñado y desarrollado por el Laboratorio de Inteligencia Artificial de la Universidad de Stanford en el año 2007 y que, hoy en día, se ha convertido en uno de los software más usados en la industria de la robótica.

Este framework fue diseñado con el objetivo de simplificar el proceso de desarrollo de software, para lo que dispone de una serie de servicios, como puede ser la transmisión de mensajes entre diferentes procesos, la gestión de paquetes o el manejo de dispositivos a bajo nivel.

A lo largo de los años, los desarrolladores de ROS han ido creando diferentes distribuciones de ROS con el propósito de permitir que los programadores que emplean este middleware trabajen sobre una base de código lo más estable posible. Cabe mencionar que, para la elaboración de este trabajo, se ha empleado la distribución de ROS “Melodic Morenia”.

A la hora de desarrollar código empleando ROS, se podría decir que es muy versátil pues permite el uso de varios lenguajes de programación como Python o C++. Pero, a pesar de su volubilidad respecto al uso de lenguajes de programación, ROS sólo puede ser ejecutado en plataformas basadas en UNIX.

Para comprender el funcionamiento de ROS, hay que conocer tres aspectos fundamentales: el sistema de ficheros de ROS, el grafo computacional sobre el que se basa este entorno y cómo se comunican los distintos elementos que lo conforman.

2.1.1 El sistema de ficheros de ROS

El sistema de ficheros es el modo en el que los ficheros de ROS se estructuran en el disco duro. Tal y como se puede apreciar en la figura 2.1, los ficheros en ROS se organizan principalmente en metapackages, packages, package manifest, mensajes, servicios y código. A continuación, se muestra una breve descripción de cada uno de los componentes:

- Los metapackages, son packages que realizan una función concreta. Sirven para representar un grupo de otros paquetes relacionados.
- ROS emplea los packages o paquetes para organizar sus programas y consisten en nodos, procesos y archivos de configuración.
- Los mensajes de ROS, que permiten a los nodos de ROS comunicarse entre ellos. Se declaran dentro de un fichero con la terminación .msg. Se suelen utilizar cuando se emplea el modelo publicador/suscriptor.
- En cada paquete hay un package manifest que recibe el nombre de package.xml. En este archivo se puede encontrar información sobre la

versión del paquete que se está usando, su autor, la licencia que tiene y las dependencias que requiere.

- Los servicios ROS son similares a los mensajes, pero se definen en un fichero con extensión .srv y se emplean cuando se hace uso del modelo cliente/servidor.

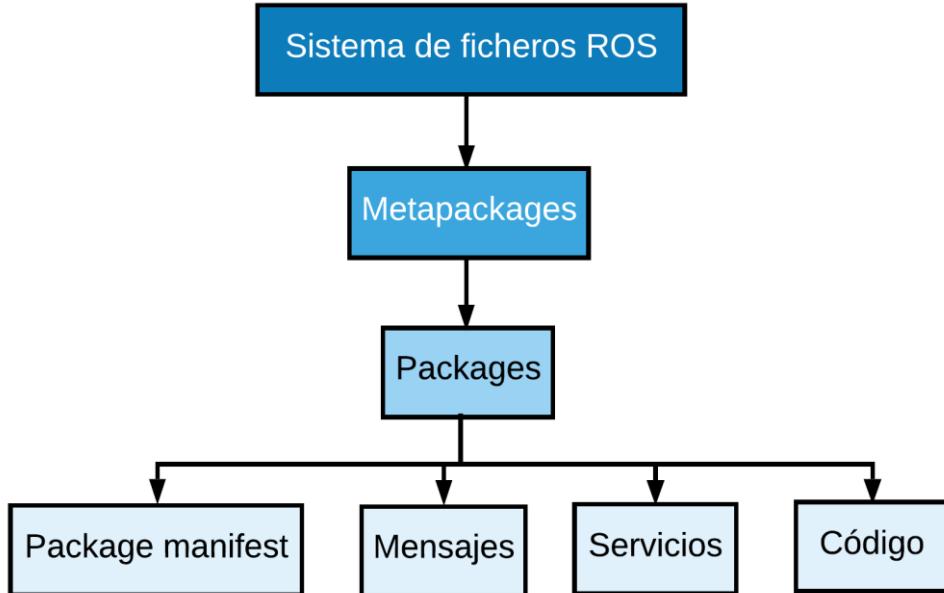


Fig. 2.1 Organización del sistema de ficheros de ROS.

2.1.2 El grafo computacional

El grafo computacional de ROS es una red peer-to-peer (P2P) que procesa toda la información en conjunto.

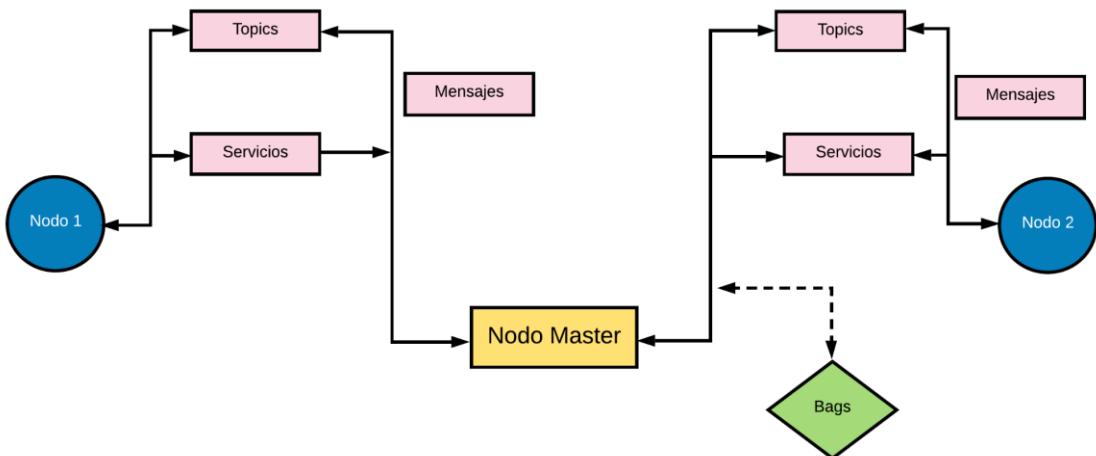


Fig. 2.2. Grafo computacional de ROS.

La figura 2.2 muestra las distintas partes de las que se compone el grafo de organización de los elementos que conforman ROS. Dichas partes son descritas a continuación:

- Los nodos de ROS son programas implementados haciendo uso de ROS y que se comunican entre sí.
- El nodo master funciona como intermediario entre nodos, ya que ayuda a que distintos nodos ROS se comuniquen entre sí. Posee toda la información sobre todos los nodos que existen en el entorno ROS.
- Los mensajes son la forma en la que los nodos ROS envían y reciben información de otros nodos.
- Los topics son una de las maneras que tienen los nodos de comunicarse entre sí. Para ello, un nodo tiene la capacidad de publicar o suscribirse a un topic.
- Los servicios son un tipo de comunicación entre nodos parecida a los topics. En los servicios, un nodo será el que provea el servicio y otro nodo actuará como cliente. El nodo cliente tendrá que esperar a que el nodo servidor responda con los resultados.
- Los bags actúan como registro de los datos procedentes de los topics. Estos bags pueden ser reproducidos en el caso de que sean necesarios.

2.1.3 Comunicación entre nodos

En el apartado anterior se expone que, los nodos, se comunican entre sí empleando los topics o servicios. Esto es un aspecto clave de la programación con ROS por lo que conviene conocer cómo se realiza esta comunicación. Por simplicidad, ya que el objetivo de este apartado es mostrar cómo funciona la comunicación entre nodos, a continuación (figura 2.3), se explicará cómo sucede esta comunicación empleando una arquitectura de tipo editor/suscriptor, que es la que más se hace uso en este trabajo:

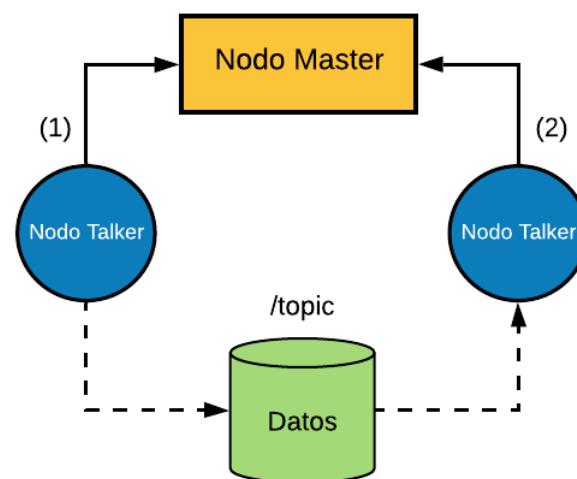


Fig. 2.3. Comunicación por topics.

Tal y como se puede observar en la figura 2.3, hay tres nodos, el nodo “talker”, el nodo “listener” y el nodo “master”.

Antes de ejecutar cualquier nodo en ROS, se debe ejecutar el nodo master y una vez este se inicia, esperará al resto de nodos. Cuando el nodo “talker” (editor) comience a ejecutarse, se conectará al nodo master e intercambiará con él los detalles de lo que quiere publicar (1).

Al ejecutarse el nodo “listener” (suscriptor), este contactará con el nodo master y empezará a intercambiar la información del nodo al que se quiere suscribir (tema al que se va a suscribir, tipo de mensaje, ...) (2).

Una vez que los nodos editor y suscriptor se han conectado, el nodo maestro deja de ser útil en la comunicación y los nodos se intercambiarán mensajes entre ellos. El nodo “talker” publica datos en el topic de nombre “/topic” y mientras, el nodo “listener” recibe esos datos siempre y cuando esté suscrito al topic.

2.2 Gazebo

Gazebo es un simulador 3D de código abierto orientado a la robótica y que fue desarrollado por la Open Source Robotics Foundation (OSRF).

Posee un sistema de generación de imágenes en 3D basado en OGRE, un motor de renderizado 3D orientado a escenas y de software libre. También hace uso de los motores de físicas ODE y Bullet [1], que se encargan de simular ciertas propiedades físicas como pueden ser el movimiento o la elasticidad de objetos o la fluidez de líquidos.

Gazebo permite recrear entornos 3D dando la posibilidad de modificar el ambiente en el que se va a realizar la simulación. Se pueden definir parámetros como la gravedad, luminosidad o la atmósfera del entorno, haciendo posible la evaluación del comportamiento de robots en entornos extremos sin necesidad de arriesgarse a hacer comprobaciones en un entorno real, en el que el robot pudiera verse dañado.

Finalmente, indicar que Gazebo es el simulador que se ha empleado para evaluar el funcionamiento del software desarrollado y es el más usado en el desarrollo de Aerostack.

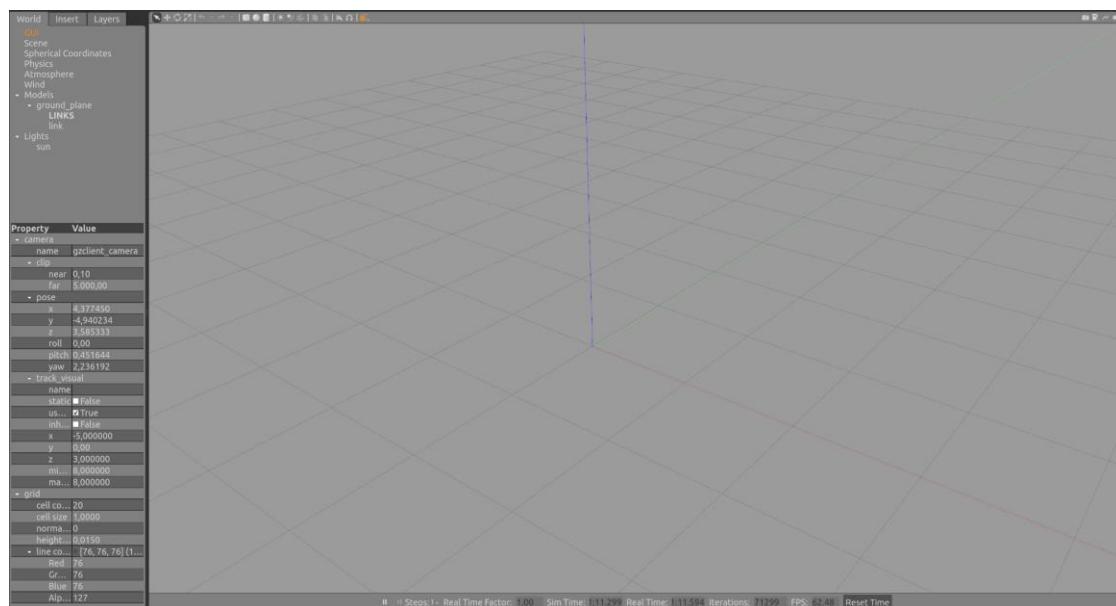


Fig. 2.4. Imagen de la interfaz gráfica del simulador Gazebo.

2.3 Aerostack

Aerostack es un entorno desarrollado por el grupo de investigación CVAR cuyo objetivo es ayudar a los desarrolladores a diseñar y construir arquitecturas de robots aéreos [2] (www.aerostack.org). Aerostack está basado en ROS y permite simular sistemas aéreos autónomos para probar algoritmos o estructuras.

2.3.1 Arquitectura

La arquitectura de Aerostack se basa en una serie de componentes principales los cuales a su vez están conformados por una serie de procesos o nodos ROS. En la figura 2.5 se pueden observar los elementos principales que componen Aerostack y en su interior una serie de cajas, que representan los nodos ROS que los forman. A continuación, se muestra una descripción más amplia de los componentes de Aerostack:

- Interfaces, que pueden ser del tipo sensor-actuador (en la figura 2.5 “Sensor-actuator Interface”) y se encarga de recibir datos de los sensores y enviar órdenes a los actuadores del robot o del tipo interacción (en la figura 2.5 “Interaction Interface”) que se encarga de comunicar el robot con operadores humanos o con otros robots.
- Un canal de comunicación (en la figura 2.5 “Communication channel”), que contiene información dinámica que se comparte entre distintos procesos, lo que facilita la interoperabilidad de estos. Además, ayuda a reutilizar componentes en diferentes tipos de plataformas aéreas.
- Comportamientos o behaviors de los robots, que se encargan de implementar diversas capacidades funcionales como pueden ser la estimación del estado, la extracción de características o el control y planificación del movimiento de un dron. Los comportamientos son controlados por el componente “Behavior Management”, que se encarga de controlar la ejecución paralela de múltiples behaviors del robot para evitar conflictos de consistencia.
- Belief memory, que es un sistema de generación de beliefs o creencias encargado de abstraer y almacenar los estados más relevantes del entorno del robot. Las creencias se emplean para tomar decisiones acerca de cuál es la siguiente tarea a realizar durante la ejecución de una misión.
- Controlador de misión (“Mission Execution Control” en la figura 2.5), que ejecuta un plan de misión previamente especificado en un árbol de comportamiento o en Python. Incluye procesos de supervisión y recuperación de la seguridad con el fin de proporcionar tolerancia a fallos.

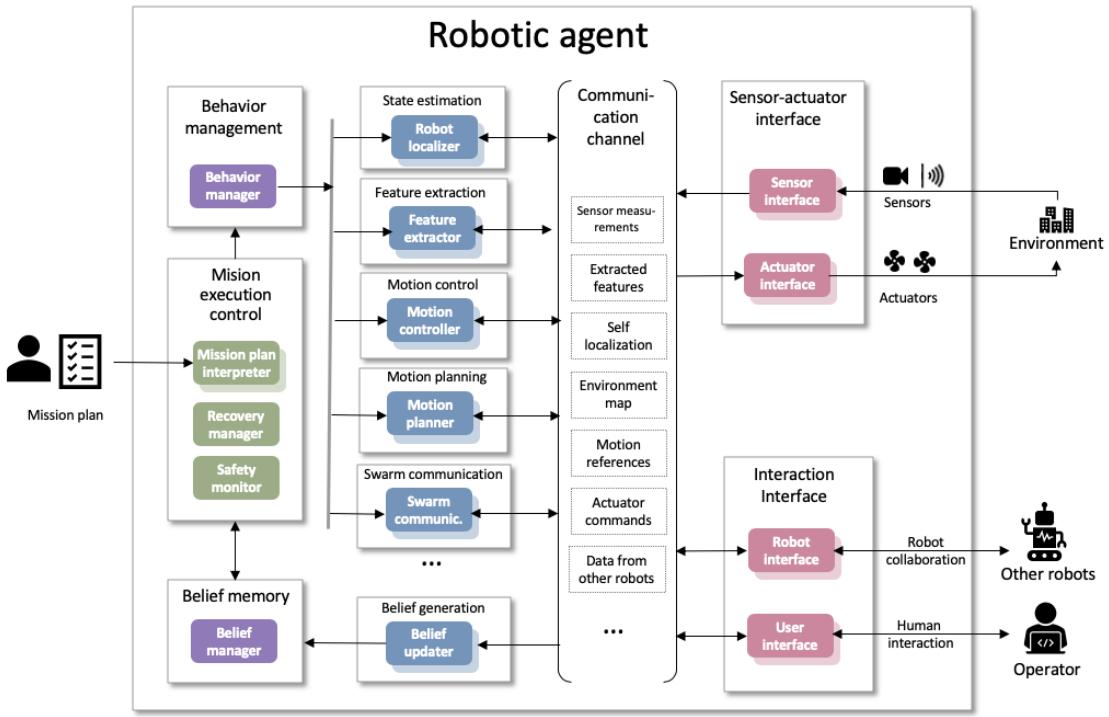


Fig. 2.5. Diferentes componentes de la arquitectura de Aerostack.

2.3.2 Behaviors

Los behaviors o comportamientos representan una acción que un dron es capaz de hacer. Son capaces de realizar tareas tan diversas como la extracción de características del entorno, el control de movimiento o la estimación del estado de un dron. También pueden permitir a un robot interactuar con otros robots. Son implementados como nodos ROS.

Aerostack, proporciona una biblioteca software para implementar behaviors de robots que puedan ser empleados por los desarrolladores para construir una arquitectura aérea en particular. Algunos ejemplos de los componentes que ofrece Aerostack son:

- Extractores de características que leen los estados de los sensores e implementan algoritmos de reconocimiento de visión.
- Controladores de movimientos como el controlador PID.
- Behaviors de autolocalización o de mapeo.
- Planificadores de movimiento que generan trayectorias libres de obstáculos para alcanzar un punto destino.
- Métodos de comunicación con otros agentes como robots u operadores humanos.

Todos los behaviors de Aerostack se caracterizan por:

- Poseer una comunicación común por la cual cada comportamiento envía y/o recibe mensajes a través del canal de comunicación presente en la arquitectura de Aerostack. En el caso de que varios behaviors publiquen datos en el mismo tema ROS, se deben emplear mecanismos de control para evitar conflictos.

- Tener una interfaz uniforme y única encargada de controlar la ejecución (activación, desactivación, ...).

2.3.3 Características

Las características de Aerostack se pueden abordar desde dos puntos de vista: desde el punto de vista de los comportamientos de los drones o desde el punto de vista de la robótica.

Desde el punto de vista de los comportamientos encontramos las siguientes características:

- Un modelo cognitivo capaz de soportar el comportamiento autónomo. El modelo multicapa de Aerostack es capaz de identificar y organizar los procesos necesarios para permitir el comportamiento autónomo, integrando los procesos de percepción, control reactivo, deliberativo, supervisión, etc.
- La organización, al estar separada en capas, se puede obtener múltiples representaciones en varios niveles, lo que permite ofrecer una interacción más dinámica con los operadores.
- División funcional, que permite una ejecución eficaz. La arquitectura, se divide en varios bloques funcionales permite que se comporte como una red distribuida, haciendo que el sistema sea más eficiente y que, por tanto, que los vuelos de los drones sean lo más realistas posibles.

Desde el punto de vista de la robótica las características son las siguientes:

- Los elementos que componen la arquitectura hacen uso de términos comunes en el mundo de la robótica autónoma, lo que facilita a los desarrolladores el mantenimiento de Aerostack.
- Organización homogénea de la arquitectura. La arquitectura está organizada en tres niveles: capa, sistema y proceso, facilitando la comprensión del sistema.
- Implementación independiente, que hace que sea más sencilla la reutilización para distintas plataformas.

2.3.4 Intérpretes de misiones

Aerostack actualmente dispone de dos maneras de elaborar y ejecutar misiones [3]:

- Intérprete Python: para programar una misión en Python es necesario usar una API, llamada Execution Engine API. Presenta dos limitaciones principales: es difícil verificar la viabilidad de la misión de antemano y es difícil de usar por usuarios no familiarizados con lenguajes informáticos.
- Intérprete basado en árboles de comportamiento: el árbol de comportamiento presenta los behaviors de manera jerárquica. Durante la ejecución de la misión se muestra la operación que se está ejecutando del árbol.

```

import mission_execution_control as mxc
import rospy

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print('Mission completed.')

```

Fig. 2.6. Ejemplo de intérprete Python.

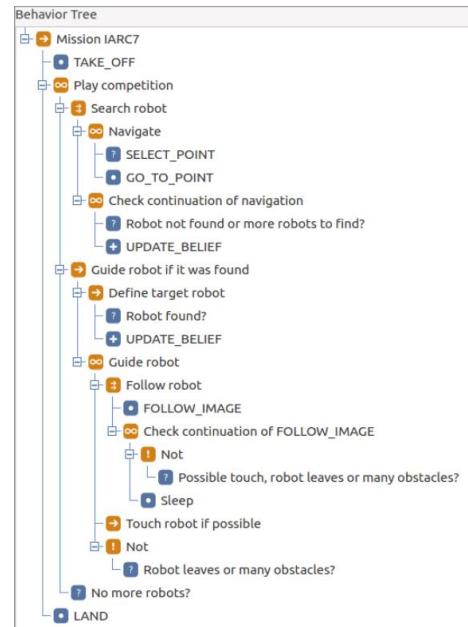


Fig. 2.7. Ejemplo de intérprete basado en árboles de comportamiento.

2.4 Quadrotor PID Controller

Ciertos comportamientos de los robots pueden programarse mediante técnicas de teoría de control y, en particular, la técnica PID de controladores, la cual consiste en regular el valor de una variable haciendo uso de otras.

Es por esto por lo que todos los behaviors desarrollados en este trabajo se sustentan gracias a un nodo ROS que forma parte de Aerostack llamado “Quadrotor PID Controller”. Este nodo implementa un controlador de movimiento para drones y opera de la siguiente manera:

- El nodo recibe como datos de entrada la posición y la velocidad deseadas para un dron. Estos datos se emplean dependiendo del modo de control del controlador.
- El controlador genera una salida de alta frecuencia para los actuadores de los motores del dron.

La configuración de este nodo se lleva a cabo a través de un archivo en formato YAML llamado quadrotor_pid_config.yaml y que posee parámetros para cuatro controladores PID: posición, altitud, velocidad y guiñada. Cada uno de estos controladores posee tres ganancias:

- Ganancia proporcional (P): se encarga de regular el tamaño de las oscilaciones de un robot sobre una determinada posición, cuanto mayor ganancia proporcional mayor serán las oscilaciones.
- Ganancia integral (I): borra el error estacionario del dron, haciendo que este mantenga la posición deseada.
- Ganancia derivativa (D): influye sobre la velocidad a la que un robot alcanza el valor deseado.

3 Diseño y programación

En este capítulo, se explican el diseño y programación de los distintos comportamientos desarrollados.

3.1 Visión general

Tal y como se ha dicho anteriormente, los behaviors representan acciones que un robot puede realizar y, en el caso de los comportamientos desarrollados en este trabajo, se tratan de acciones orientadas a que múltiples robots sean capaces de realizar acciones de persecución, huida, imitación y observación al operar en misiones de inspección o vigilancia.

Además, debido a la necesidad de desarrollar estos behaviors empleando un controlador PID, los comportamientos, han sido ubicados dentro del paquete ROS quadrotor_motion_with_pid_control de Aerostack, en el que ya existían otros behaviors pero que carecían de capacidad multi-dron.

Es importante resaltar que el nombre dado a los behaviors representa la acción que realizan al ser activados junto con el tipo de controlador que emplean en su ejecución y es por esta razón por la que la terminación del nombre de los comportamientos desarrollados es “With PID Control”

Respecto a cómo están diseñados los behaviors decir que, de una manera general, todos los comportamientos desarrollados heredan de la clase Behavior Execution Manager, que está implementada como un nodo ROS y provee de una interfaz a cada behavior. Esta interfaz provee al comportamiento de una serie de servicios que hacen que dicho behavior pueda, por ejemplo, activarse, desactivarse o notificar cuando termina su ejecución. La clase Behavior Execution Manager [4] también incluye funciones generales que son compartidas por cada subclase e implementan los procedimientos específicos para cada comportamiento.

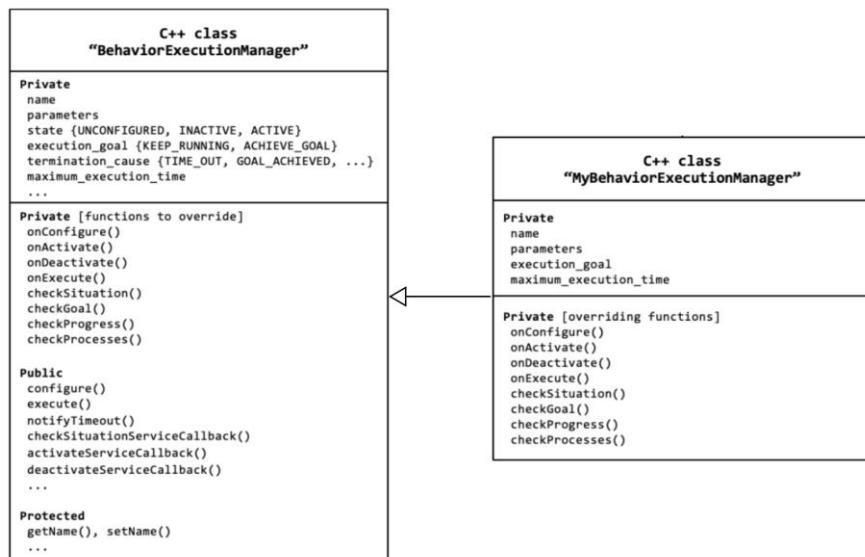


Fig. 3.1. Diagrama UML que muestra como un behavior hereda de la clase Behavior Execution Manager.

Cada una de las funciones heredadas de la clase Behavior Execution Manager realizan una tarea determinada. Dependiendo de la tarea específica realizada por cada función, estas se pueden agrupar en dos grandes bloques: funciones encargadas del control y ejecución del behavior y funciones encargadas de la gestión de la activación o desactivación del behavior. Las siguientes tablas muestran la función específica que realiza cada función:

TABLA I
FUNCIONES ENCARGADAS DEL CONTROL Y LA EJECUCIÓN.

Función	Descripción
checkSituation ()	Esta función verifica si el behavior puede ser ejecutado en el estado actual del dron. Devuelve 1 (true) si el comportamiento puede ser ejecutado o 0 (false) si no se puede ejecutar.
checkGoal ()	Verifica que el behavior haya alcanzado su objetivo (goal).
checkProgress ()	Verifica que el robot progrese adecuadamente de acuerdo con la ejecución del behavior.
checkProcesses ()	Chequea que los procesos empleados para controlar el behavior se ejecuten correctamente.

TABLA II
FUNCIONES ENCARGADAS DE LA ACTIVACIÓN/DESACTIVACIÓN.

Función	Descripción
onConfigure ()	Esta función prepara la ejecución del behavior. En ella se leen los parámetros y/o ficheros.
onActivate ()	Se encarga de activar el comportamiento y en ella se leen los argumentos del behavior y se crean los publicadores y los suscriptores.
onExecute ()	En ella se desarrolla la ejecución del behavior. Se ejecuta en un bucle infinito.
onDeactivate ()	Desactiva el comportamiento. En ella se “apagan” (shutdown) los publicadores y suscriptores.

En cuanto a la programación de los comportamientos, decir que se ha realizado empleando C++ para programar las funciones que intervienen en la ejecución de los algoritmos empleados en los behaviors y XML para añadir cada comportamiento al fichero quadrotor_motion_with_pid_control.launch, que se encarga de especificar los parámetros que se establecen y los nodos que inician cada behavior. También se hizo uso de la herramienta CMake para incluir en el fichero CMakeList.txt los archivos .h de cada comportamiento, que se encargan de definir las cabeceras de las funciones, las variables que se van a usar y las librerías empleadas y los archivos .cpp de cada behavior en los que se programa el algoritmo de cada comportamiento. Además de los archivos .h y .cpp de cada behavior, el fichero CMakeList.txt también alberga las dependencias y los componentes requeridos con el fin de decir al sistema cómo compilar el código y donde instalarlo.

Respecto a los aspectos más básicos de la programación, hay que destacar que se ha hecho uso del IDE Visual Studio Code en la implementación de los algoritmos y se ha utilizado las guías de estilo de programación recomendadas por los desarrolladores de ROS [5]. Para el control de versiones se ha empleado GitHub (<https://github.com/lesmesrafa/Multi-drone-behaviors.git>).

Finalmente, decir que para conseguir programar cada uno de los behaviors y que funcionen correctamente, ha sido necesario el uso de diversos topics y servicios:

- set_control_mode, un servicio que establece el modo de control como puede ser POSE, SPEED, o ATTITUDE, GROUND_SPEED o SPEED_3D. Cada uno de estos valores que puede tomar este servicio junto con el uso del nodo de Aerostack “Quadrotor PID Controller” hacen que ciertas variables presentes en los topics motion_reference/speed y motion_reference/pose no sean tenidas en cuenta por el controlador. En este trabajo se han empleado durante el desarrollo de los behaviors los modos de control GROUND_SPEED y SPEED_3D.
- self_localization/pose, un topic que da la posición actual del robot.
- self_localization/flight_state, un topic que comprueba que el estado del dron sea HOVERING si este está flotando en el aire; FLYING si está volando, LANDING si está aterrizando, LANDED si ha aterrizado o TAKING_OFF si está despegando. Dependiendo de que valor de los anteriores mencionados tenga el estado del robot se ejecutará un behavior o no lo hará.
- actuator_command/flight_action, un topic que especifica la acción que ejecuta el robot, que pueden ser LAND para ordenar que aterrice; HOVER para que flote en el aire; TAKE_OFF para que aterrice o MOVE para que se mueva).
- motion_reference/speed, un topic que controla la velocidad del robot que ejecuta el behavior. Si se hace uso del modo de control GROUND_SPEED, el controlador solo emplea las velocidades en los ejes “x” e “y”, descartando el resto de las velocidades. Si se emplea el modo de control SPEED_3D, el controlador solo hará uso de las velocidades en los ejes “x”, “y”, “z”, ignorando el resto de las velocidades.
- motion_reference/pose, un topic que controla a qué posición debe ir el dron que ejecuta el behavior. Si se utiliza el modo de control GROUND_SPEED, el controlador solo emplea como posición deseada la de altitud (z) y el ángulo yaw, descartando el resto de las posiciones. Si se emplea el modo de control SPEED_3D, el controlador tomará como posición deseada el ángulo yaw, ignorando el resto de las posiciones.
- shared_robot_positions_channel, un topic que permite obtener la posición de cada robot en un espacio tridimensional.

TABLA III
TOPICS Y SERVICIOS EN CADA BEHAVIOR I.

Behaviors	Topics/Servicios			
	set_control_mode	self_localization/pose	self_localization/flight_state	shared_robot_position_channel
FOLLOW_TARGET	Si (cliente)	Si (suscriptor)	Si (publicador)	Si (suscriptor)
GET_CLOSE_TO_TARGET	Si (cliente)	Si (suscriptor)	Si (publicador)	Si (suscriptor)
MOVE_AWAY_FROM_TARGET	Si (cliente)	Si (suscriptor)	Si (publicador)	Si (suscriptor)
REACH_TARGET_ALTITUDE	Si (cliente)	Si (suscriptor)	Si (publicador)	Si (suscriptor)
KEEP_TARGET_ALTITUDE	Si (cliente)	Si (suscriptor)	Si (publicador)	Si (suscriptor)
TAKE_A_LOOK_AT_TARGET	Si (cliente)	Si (suscriptor)	Si (publicador)	Si (suscriptor)
KEEP_LOOK_AT_TARGET	Si (cliente)	Si (suscriptor)	Si (publicador)	Si (suscriptor)

TABLA IV
TOPICS Y SERVICIOS EN CADA BEHAVIOR II.

Behaviors	Topics/Servicios		
	actuator_command/flight_action	motion_reference/speed	motion_reference/pose
FOLLOW_TARGET	Si (publicador)	Si (publicador)	Si (publicador)
GET_CLOSE_TO_TARGET	Si (publicador)	Si (publicador)	Si (publicador)
MOVE_AWAY_FROM_TARGET	Si (publicador)	Si (publicador)	Si (publicador)
REACH_TARGET_ALTITUDE	Si (publicador)	Si (publicador)	Si (publicador)
KEEP_TARGET_ALTITUDE	Si (publicador)	Si (publicador)	Si (publicador)
TAKE_A_LOOK_AT_TARGET	Si (publicador)	No	Si (publicador)
KEEP_LOOK_AT_TARGET	Si (publicador)	No	Si (publicador)

3.2 Follow Target With PID Control

Este behavior permite al ser activado, que un dron persiga a otro. Además, posee tres parámetros que permiten personalizar algunos aspectos del comportamiento, haciendo que el dron actúe de cierta manera según el valor que tomen dichos parámetros, los cuales son:

- `target_name`, de tipo `string` y que representa la identificación del dron a perseguir. Es un parámetro obligatorio.
- `safety_distance`, indica la distancia mínima que debe mantener el dron que activa el behavior con el dron al que persigue. Es un parámetro opcional con un valor por defecto de 2.0 metros.
- `maximum_speed`, un número real que indica la velocidad máxima a la que debe ir el dron en metros por segundo. Es un parámetro no obligatorio con un valor por defecto de 0.3 metros por segundo.
- `yaw`, un parámetro que puede tener dos valores: `constant`, si no se desea que el robot persiga mirando al objetivo o `target_facing` para lo contrario. Por defecto, el valor de este parámetro es `constant`.

Para diseñar este behavior, se ha tenido en cuenta que el dron que se quiere perseguir envía su posición al dron que activa este comportamiento a través de un topic. Una vez obtenida dicha posición y dependiendo del valor del parámetro `yaw`, o bien el robot acelera sin modificar su eje de guiñada o, por el contrario, empleando los ángulos de Euler, se hace rotar al dron sobre el eje normal o eje de guiñada, de manera que la parte frontal del mismo se orienta hacia el objetivo a perseguir y, una vez que esto ocurre se modifica la velocidad del robot de manera que este sea capaz de simular la acción esperada que, en este caso es la de perseguir al objetivo deseado.

Otra forma mucho más fácil de diseñar este behavior hubiera sido emplear directamente las coordenadas que se reciben del dron a perseguir para que el robot que active el behavior vaya directamente a esa posición. Sin embargo, de esa manera, el controlador PID no permite regular la velocidad a la que se persigue.

Este behavior se ejecutará de forma recurrente hasta que se desactive externamente, momento en el que el robot se queda flotando en el aire. Decir también que este comportamiento solo puede ser ejecutado en el caso de que el estado del dron que lo invoque sea `HOVERING` o `FLYING`.

Además, el espacio de seguridad que mantienen los drones se calcula mediante la distancia euclídea de un dron al otro. Esta distancia también se emplea para autorregular la velocidad del dispositivo que ejecute este behavior.

La figura 3.2 muestra cómo interactúa este behavior con los distintos topics y servicios de los que hace uso. Como se puede ver, este comportamiento publica información en los topics `actuator_command/flight_action`, `motion_reference/speed` y `motion_reference/pose` y se suscribe a los topics `self_localization/flight_state`, `self_localization/pose` y `/shared_robot_position_channel`. También, este behavior actúa como cliente de `trajectory_controller/set_control_mode` y de `behavior_activation_finished`. Respecto a `check_activation`, `check_activation_conditions`, `activate_behaviors` y `deactivate_behaviors` mencionar que es el propio comportamiento el que actúa como servicio para estos cuatro clientes.

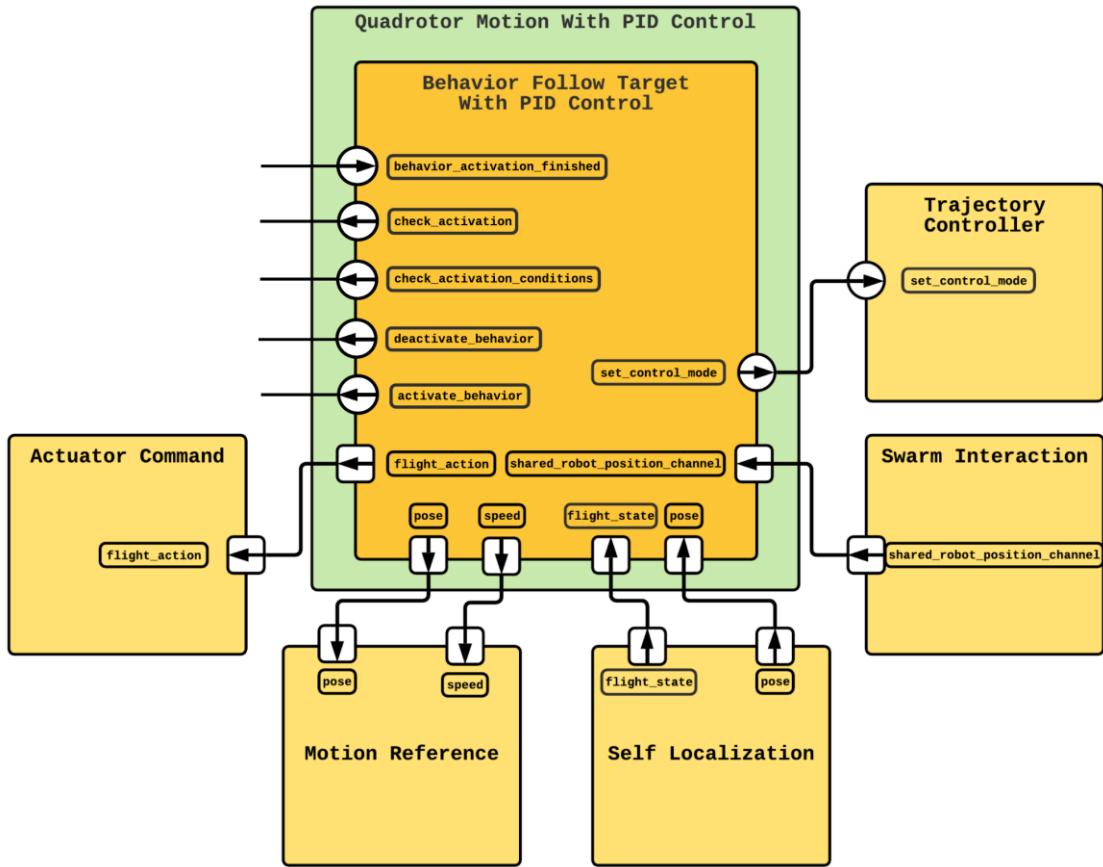


Fig. 3.2. Interacciones con topics y servicios de Follow Target With Pid Control.

Por último, y para mostrar de una manera más estructurada el funcionamiento de este behavior, en el algoritmo 1 se puede observar los pasos principales de la ejecución del comportamiento expresado en pseudocódigo.

Algoritmo 1. Follow Target

```

1: Obtener la posición del objetivo a perseguir  $x_{target}$ ,  $y_{target}$ ,  $z_{target}$ 
2: Obtener la posición del robot que llama a Follow Target  $x_{est}$ ,  $y_{est}$ ,  $z_{est}$ 
3: Mientras  $\neg$ inhibido:
4:   Calcular distancia euclídea  $d$ , al objetivo y a los ejes "x" e "y",  $d_{eje}$ 
5:   Si  $Yaw = "target_facing"$  Entonces calcular el ángulo al que hay que
       orientar el robot  $Yaw_{sp}$ ,  $Yaw_{sp} = atan2(y_{target} - y_{est}, x_{target} - x_{est})$ 
6:   Cambiar  $set\_control\_mode$  a GROUND_SPEED
7:   Publicar  $Yaw_{sp}$  y  $z_{est}$ 
8:   Por cada eje  $x$ ,  $y$  :
9:     Si  $d > SAFETY_DISTANCE$  Entonces Calcular  $speed_{eje} = (speed \cdot d_{eje}) / d$ 
10:    Sino  $speed_{eje} = 0$ 
11:   Publicar  $speed_{eje}$ 
12:   Cambiar a MOVE  $flight\_action$ 
14:    $speed_{eje} = 0$ 
15:   Cambiar a HOVER  $flight\_action$ 

```

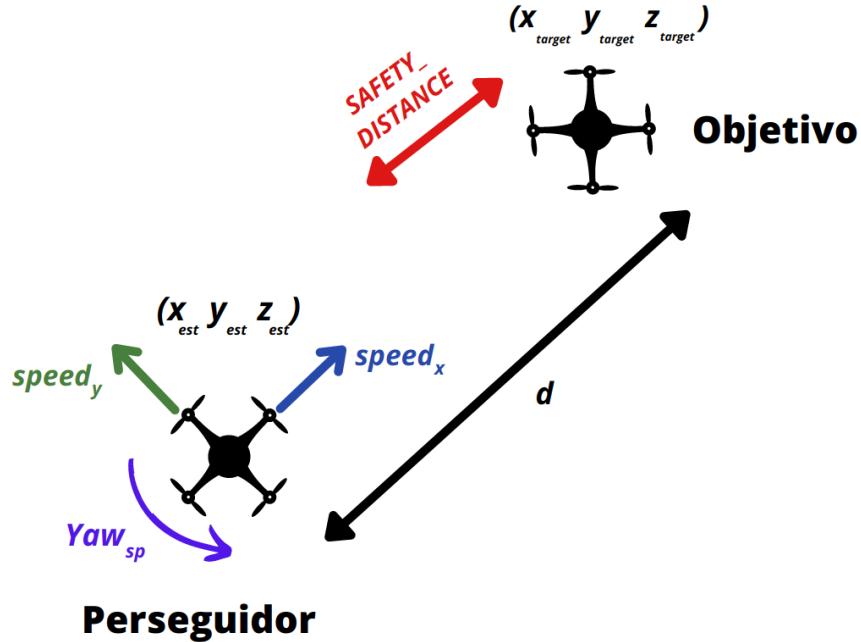


Fig. 3.3. Ilustración que muestra algunos de los elementos clave del algoritmo 1.

Como se puede observar en el algoritmo 1, una vez que se obtienen las posiciones de ambos drones se ejecuta un bucle en el que se calcula el ángulo al que se debe orientar el robot, Yaw_{sp} (si se desea, ya que depende del parámetro yaw). También se regula la altitud tal y como puede observarse en las líneas 6 y 7 y se regula la velocidad [6] en los ejes “x” e “y” (líneas 8-11), siendo la variable “speed” del dividendo de la fórmula para ajustar la velocidad, la velocidad máxima a la que se desea perseguir.

Para una visión más detallada de cómo se programaría este algoritmo, el [Anexo 1](#) muestra su implementación en pseudocódigo similar a C++.

3.3 Move Away From Robot With PID Control

Al activar este behavior, se hace que un robot huya de otro robot una vez que el dron que lo ejecuta está en el aire. En total este comportamiento posee tres parámetros de los cuales uno de ellos es obligatorio. Estos parámetros permiten personalizar aspectos secundarios de este behavior como la velocidad o la orientación. Estos parámetros son los siguientes:

- `follower_name`, de tipo string y que indica la identificación del robot del que hay que huir. Es un parámetro obligatorio sin valor por defecto.
- `separation_distance`, indica la distancia mínima que debe mantener el dron que activa el behavior con el dron del que huye. Es un parámetro no obligatorio con un valor por defecto de 3.0 metros.
- `maximum_speed`, un número real que indica la velocidad máxima a la que debe ir el dron en metros por segundo. Es un parámetro opcional con valor por defecto de 0.3 metros por segundo.
- `yaw`, un parámetro que puede tener tres valores: `constant`, si se desea mantener constante el valor del ángulo; `path_facing` si se desea mirar al lado contrario en el que está el dron del que se huye y `follower_facing` para lo contrario. Por defecto, el valor de este parámetro es `constant`.

Para la correcta ejecución de este behavior, el robot del que se desea huir debe de publicar su posición en un topic. Una vez que se obtiene la posición del objetivo a huir y, dependiendo del parámetro yaw, el robot comienza a girar sobre su eje normal de manera que queda mirando al lado contrario al que se sitúa el dron perseguidor o bien gira hasta mirar al frontalmente al a este dron o directamente no gira sobre ningún eje.

Después, basta con aumentar la velocidad en los ejes “x” e “y” para conseguir huir. En el caso de la altura, se ha considerado que el dron que invoca a este behavior mantenga la altura constante tomando como referencia la altura con la que se activó el behavior.

La distancia a la que se empieza a huir se calcula mediante la distancia euclídea de un dron al otro. Esta distancia también se emplea para autorregular la velocidad del dispositivo que ejecute este behavior.

Además, este comportamiento se ejecutará de forma recurrente hasta que la persona que opere el robot decida que se termine la ejecución del behavior, momento en el que el dron se queda flotando en el aire y a la misma altura con la que se activó el comportamiento. Es importante destacar que, este behavior, solo puede ser activado en el caso de que el estado del dron que lo vaya a utilizar sea HOVERING o FLYING.

La figura 3.4 muestra cómo interactúa este behavior con los distintos topics y servicios de los que hace uso. Como se puede ver, este comportamiento actúa como cliente de trajectory_controller/set_control_mode y de behavior_activation_finished. También, publica información en los topics actuator_command/flight_action, motion_reference/speed y motion_reference/pose y se suscribe a los topics self_localization/flight_state, self_localization/pose y /shared_robot_position_channel. Respecto a check_activation, check_activation_conditions, activate_behaviors y deactivate_behaviors mencionar que es el propio comportamiento el que actúa como servicio para estos cuatro clientes.

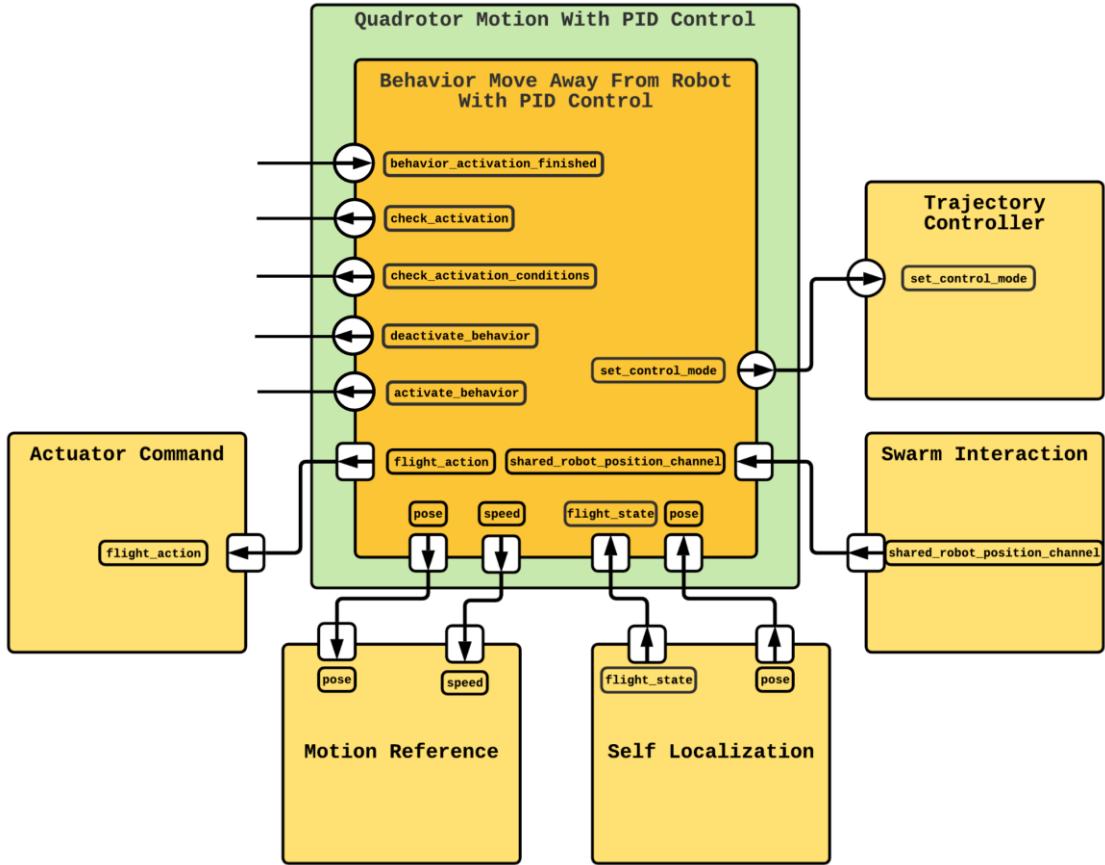


Fig. 3.4. Interacciones con topics y servicios de Move Away From Robot With Pid Control.

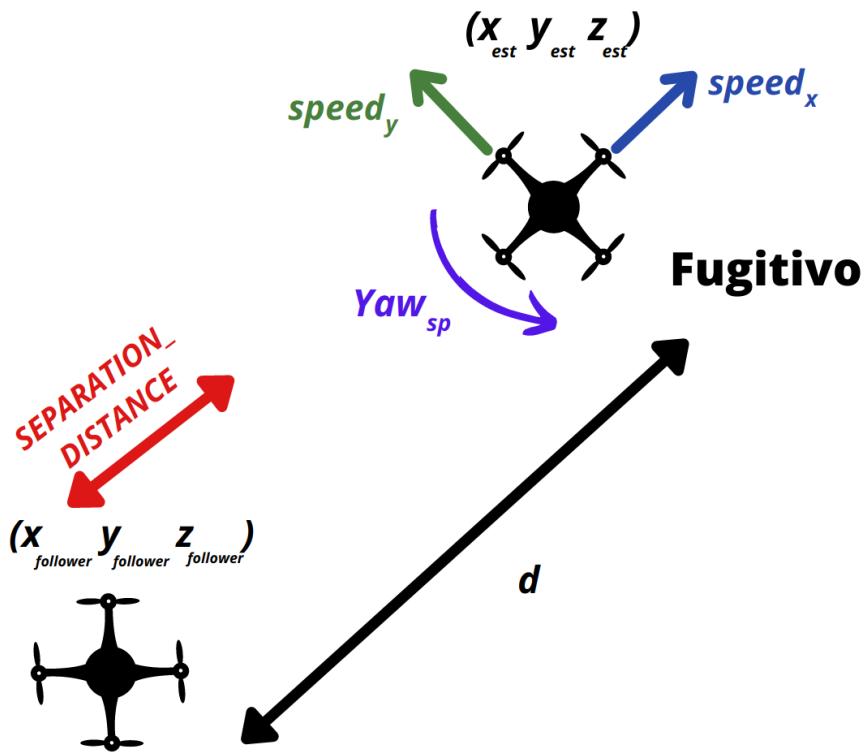
Por último, el algoritmo 2, muestra expresado en pseudocódigo, los pasos seguidos en la programación de la función de ejecución del behavior.

Algoritmo 2. Move Away From Robot

```

1: Obtener la posición del objetivo del que huir  $x_{follower}$ ,  $y_{follower}$ ,  $z_{follower}$ 
2: Obtener la posición del robot que llama al behavior  $x_{est}$ ,  $y_{est}$ ,  $z_{est}$ 
3: Mientras  $\neg$  inhibido:
4:   Calcular distancia euclídea  $d$  , al objetivo y a los ejes “x” e “y”,  $d_{eje}$ 
5:   Si Yaw = “follower_facing” Entonces calcular el ángulo al que hay que
orientar el robot  $Yaw_{sp}$  ,  $Yaw_{sp} = atan2(-(y_{follower} - y_{est}), -(x_{follower} - x_{est}))$ 
6:   Si Yaw = “path_facing” Entonces calcular el ángulo al que hay que
orientar el robot  $Yaw_{sp}$  ,  $Yaw_{sp} = atan2(y_{target} - y_{est}, x_{target} - x_{est})$ 
7:   Cambiar set_control_mode a GROUND_SPEED
8:   Publicar  $Yaw_{sp}$  y  $z_{est}$ 
9:   Por cada eje  $x$  ,  $y$  :
10:    Si  $d < SEPARATION_DISTANCE$  Entonces Calcular  $speed_{eje} = (speed \cdot d_{eje}) / d$ 
11:    Sino  $speed_{eje} = 0$ 
12:    Publicar  $speed_{eje}$ 
13:    Cambiar a MOVE flight_action
14:     $speed_{eje} = 0$ 
15:    Cambiar a HOVER flight_action

```



Perseguidor

Fig. 3.5. Ilustración que muestra algunos de los elementos clave del algoritmo 2.

Tal y como se puede ver en el algoritmo 2, una vez que se obtienen las posiciones de ambos drones, se ejecuta a partir de la línea 3 un bucle en el que se calcula el ángulo al que se debe orientar el robot, Yaw_{sp} dependiendo del valor del parámetro yaw. En este algoritmo se muestra el caso en el que se ejecute el behavior con un parámetro yaw=follower_facing. También se regula la altitud tal y como puede observarse en las líneas 6 y 7 y se regula la velocidad [6] en los ejes "x" e "y" (líneas 8-11). La variable "speed" presente en el dividendo de la fórmula para calcular la velocidad representa la velocidad máxima a la que debe ir el dron.

En el [Anexo 2](#) se muestra la programación del algoritmo 2 en pseudocódigo similar a C++.

3.4 Reach Target Altitude With PID Control

Cuando este behavior es activado, hace que el robot que lo ejecuta se sitúe a la misma altura que otro robot (target) o a una altura personalizada en función de la altura del target. Los argumentos de los que hace uso este behavior son los siguientes:

- target_name, es un parámetro obligatorio de tipo string que indica la identificación del dron del que quiero tomar la altitud como referencia.
- shift, es un parámetro opcional cuyo valor por defecto es 0.0 metros. Sirve para aumentar (o disminuir) la altitud a la que quiero situar el robot respecto al target.

- `maximum_speed`, un número real que indica la velocidad máxima a la que debe ascender el robot en metros por segundo. Es un parámetro opcional, que no tiene un valor por defecto.

Para conseguir el propósito de alcanzar la misma altura (o escoger una altura) respecto a la de otro robot (`target`), este, debe de publicar su posición a través de un topic. El robot que active este behavior por lo tanto debe suscribirse a ese topic para recibir la posición que envía el `target`.

Una vez que el robot que activa el behavior posee las coordenadas del `target`, comienza a ascender (o descender) a una velocidad constante hasta alcanzar la altura deseada, momento en el cual se quedará flotando a dicha altitud.

Cabe destacar que, este behavior solo se podrá activar si el dron que lo ejecuta está en estado de `HOVERING` o de `FLYING`.

Respecto al parámetro `maximum_speed`, comentar que carece de valor por defecto ya que se hace uso del modo de control `GROUND_SPEED` en el caso de que no se establezca una velocidad.

La figura 3.6 muestra cómo interactúa este behavior con los distintos topics y servicios de los que hace uso. Como se puede ver, este comportamiento publica información en los topics `actuator_command/flight_action`, `motion_reference/speed` y `motion_reference/pose` y se suscribe a los topics `self_localization/flight_state`, `self_localization/pose` y `/shared_robot_position_channel`. Respecto a `check_activation`, `check_activation_conditions`, `activate_behaviors` y `deactivate_behaviors` mencionar que es el propio comportamiento el que actúa como servicio para estos cuatro clientes. También, este behavior actúa como cliente de `trajectory_controller/set_control_mode` y de `behavior_activation_finished`.

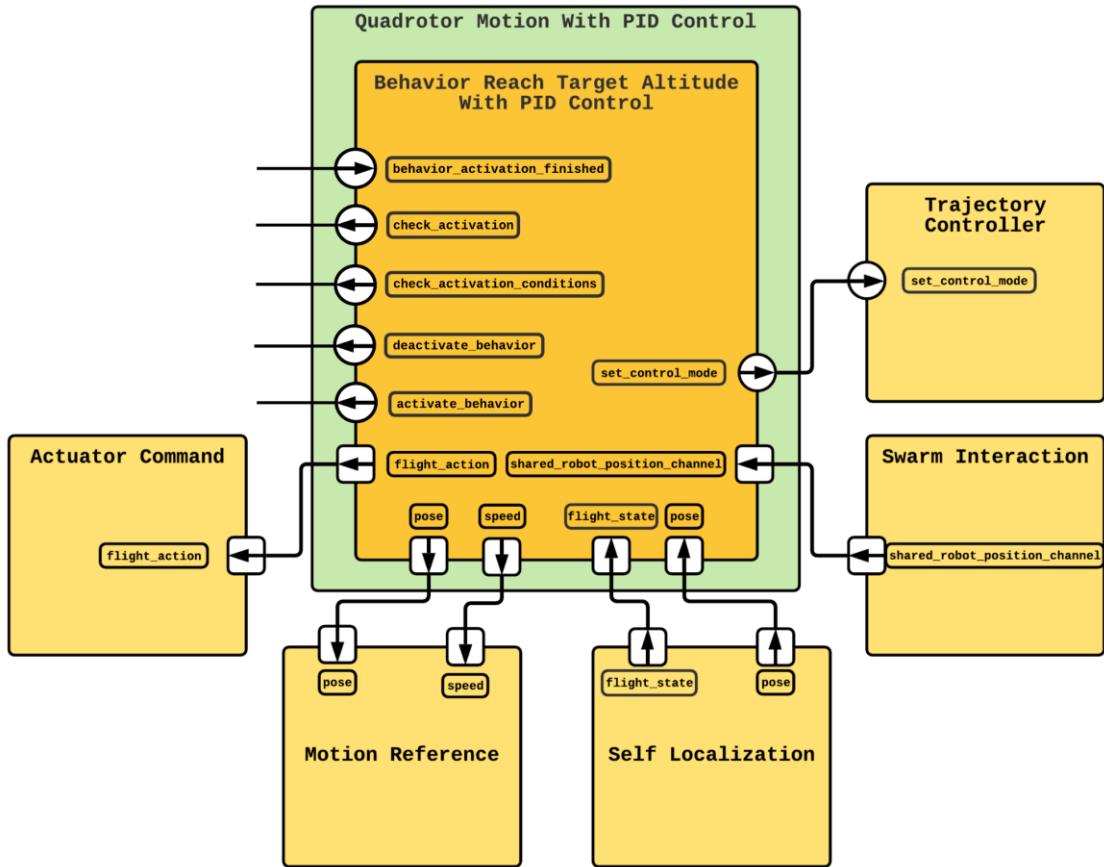


Fig. 3.6. Interacciones con topics y servicios de Reach Target Altitude With Pid Control.

Con el fin de mostrar de una manera más técnica la ejecución de este comportamiento, en el algoritmo 3 se puede observar los pasos principales seguidos en el desarrollo del comportamiento expresado en pseudocódigo.

Algoritmo 3. Reach Target Altitude

- 1: Obtener la altitud del objetivo a igualar la altitud z_{target}
 - 2: Obtener la altitud del robot que llama a Reach Target Altitude z_{est}
 - 3: **Mientras** \neg ALTITUD_ALCANZADA:
 - 4: **Si** \neg VELOCIDAD **Entonces** cambiar `set_control_mode` a GROUND_SPEED
 - 5: **Si** VELOCIDAD **Entonces** cambiar `set_control_mode` a SPEED_3D
 - 6: Publicar altitud_final si `set_control_mode` = GROUND_SPEED
 - 7: Publicar $speed_z = (speed \cdot d_z) / d$ si `set_control_mode` = SPEED_3D
 - 8: Cambiar a MOVE `flight_action`
 - 9: $speed_z = 0$
 - 10: Cambiar a HOVER `flight_action`
-

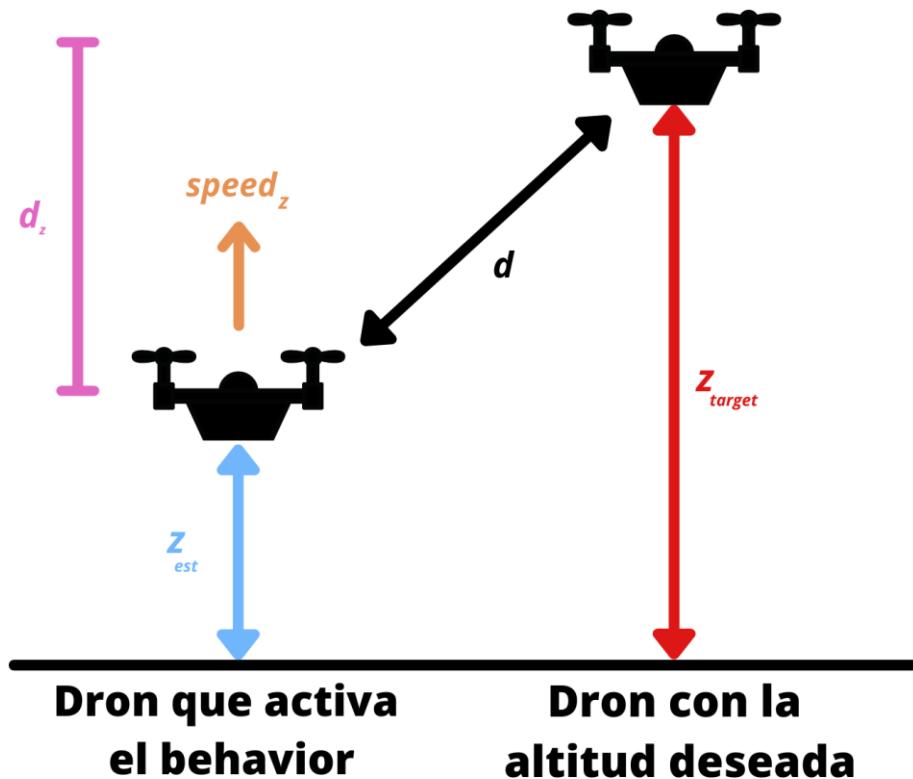


Fig. 3.7. Ilustración que muestra algunos de los elementos clave del algoritmo 3.

Como se puede observar en el algoritmo 3, una vez que se obtienen las alturas de ambos drones y dependiendo del modo de control empleado se publicará la velocidad [6] a la que debe ascender el dron (líneas 4-8). Una vez se alcanza la altitud deseada, finalizará la iteración del bucle y el dron quedará flotando en el aire. En el [Anexo 3](#) se muestra en pseudocódigo cómo sería una posible implementación de este algoritmo.

3.5 Take A Look At Target With PID Control

Cuando un robot activa este behavior, hace que este comience a rotar hasta situarse frontalmente con el robot u objetivo al que se quiere mirar. Debido a la sencillez de este behavior, posee un solo parámetro, `target_name`, que es un parámetro sin el cual el behavior no podría ejecutarse y que indica el identificador del objetivo a mirar.

Respecto al diseño, decir que, un factor clave de este behavior es que depende de que el objetivo a mirar publique sus coordenadas en un topic al cual estará suscrito este behavior. Una vez que se publican las coordenadas, se hace rotar al robot empleando los ángulos de Euler hasta que queda situado frontalmente con el objetivo, momento en el que finalizará la ejecución del behavior dejando al robot en estado de HOVERING.

Al igual que en los casos anteriores, este behavior solo se podrá activar si el dron que lo ejecuta está previamente en estado de HOVERING o de FLYING.

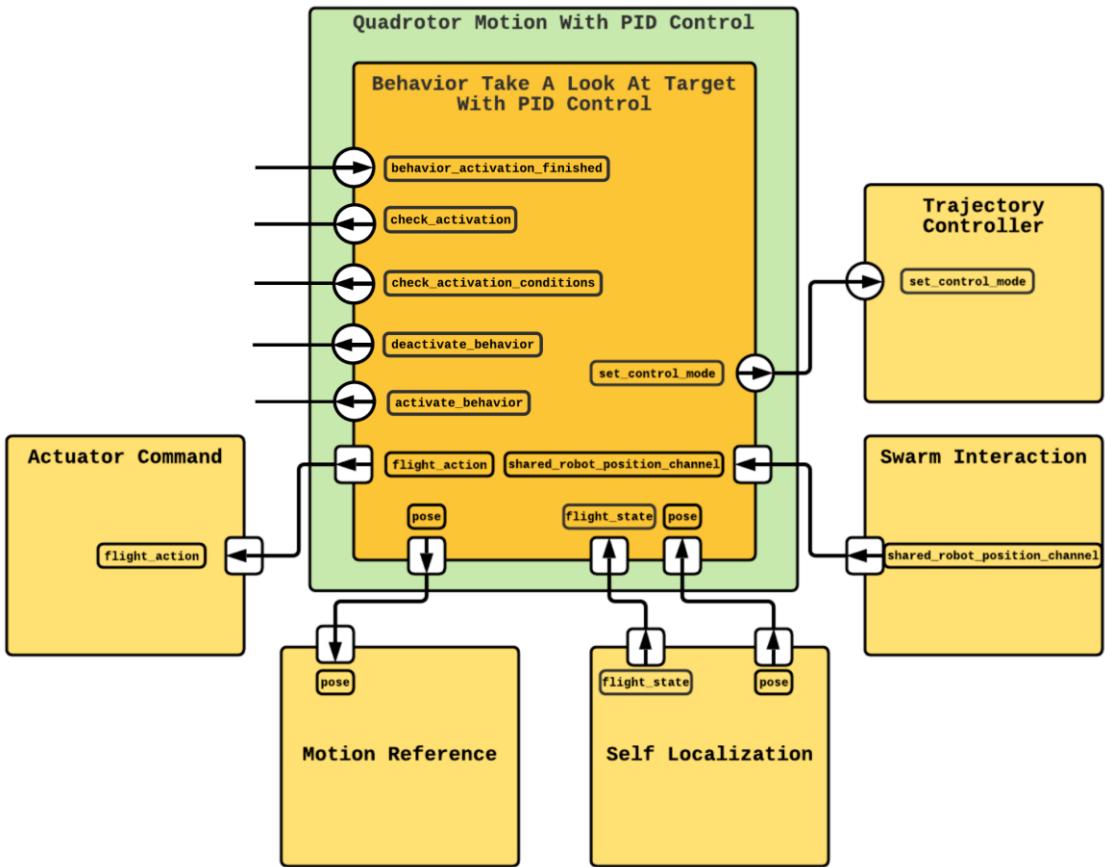


Fig. 3.8. Interacciones con topics y servicios de Take A Look At Target With Pid Control.

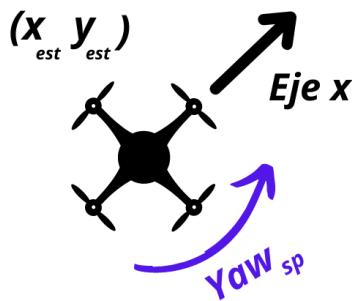
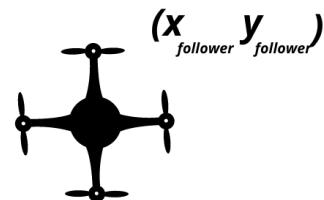
En la figura 3.8 se puede observar cómo el comportamiento hace uso de servicios y de topics. Como se puede ver, este comportamiento actúa como cliente de trajectory_controller/set_control_mode y de behavior_activation_finished. También, publica información en los topics actuator_command/flight_action y motion_reference/pose y se suscribe a los topics self_localization/flight_state, self_localization/pose y /shared_robot_position_channel. Respecto a check_activation, check_activation_conditions, activate_behaviors y deactivate_behaviors mencionar que es el propio comportamiento el que actúa como servicio para estos cuatro clientes, los cuales están involucrados en la activación y desactivación del comportamiento.

Por último, y para mostrar de una manera más estructurada el funcionamiento de este behavior, en el algoritmo 4 se puede observar los pasos principales seguidos para programar la ejecución del comportamiento expresado en pseudocódigo.

Algoritmo 4. Take A Look At Target

```
1: Obtener la posición del objetivo al que mirar  $x_{target}$ ,  $y_{target}$ 
2: Obtener la posición del robot que llama al behavior  $x_{est}$ ,  $y_{est}$ 
3: Mientras  $\neg$ MIRANDO:
4:     Calcular el ángulo al que hay que orientar el robot  $Yaw_{sp}$  ,
 $Yaw_{sp} = atan2(y_{target} - y_{est}, x_{target} - x_{est})$ 
5:     Cambiar set_control_mode a SPEED_3D
6:     Publicar  $Yaw_{sp}$ 
7:     Cambiar a MOVE flight_action
8:     Cambiar a HOVER flight_action
```

Dron observado



Observador

Fig. 3.9. Ilustración que muestra algunos de los elementos clave del algoritmo 4.

Como se puede observar en el algoritmo 4, una vez que se obtienen las posiciones de ambos drones en los planos x e y, se ejecuta un bucle (línea 3) en el que por cada iteración se calcula y se publica el ángulo Yaw (líneas 4 a 7). Este bucle finalizará cuando el eje “x” del dron que ejecuta el behavior se oriente hacia el robot deseado. En el [Anexo 4](#) se muestra en pseudocódigo cómo sería una posible implementación de este algoritmo.

3.6 Get Close To Target With PID Control

Este behavior permite al robot que lo activa perseguir a otro robot el cual debe de publicar su posición en un topic previamente. La diferencia con el behavior Follow Target With Pid Control es que en este caso en cuanto se está a una distancia determinada del robot a perseguir la ejecución de este behavior termina. Este behavior posee tres parámetros, los cuales son:

- target_name, de tipo string y que representa el identificador del dron a perseguir. Es un parámetro obligatorio.
- target_distance, indica la distancia a la que se deja de perseguir. Es un parámetro opcional con un valor por defecto de 2.0 metros.
- maximum_speed, un número real que indica la velocidad máxima a la que debe ir el dron en metros por segundo. Es un parámetro no obligatorio con un valor por defecto de 0.3 metros por segundo.
- yaw, un parámetro que puede tener dos valores: constant, si no se quiere que el robot persiga mirando al objetivo o target_facing para lo contrario. Por defecto, el valor de este parámetro es constant.

Para diseñar este behavior, el dron que se quiere perseguir debe enviar su posición al dron que activa este behavior a través de un topic. Una vez obtenida esa posición y dependiendo del valor del parámetro yaw, el robot no modifica su eje normal o bien, empleando los ángulos de Euler, se hace rotar al dron sobre el eje de guiñada, de manera que la parte frontal se orienta hacia el objetivo a perseguir, haciendo que, con tan solo aumentar la velocidad del robot, este sea capaz de perseguir al otro robot hasta alcanzar una distancia deseada (target_distance) con el robot perseguido, momento en el que finalizará la ejecución.

Al igual que en el caso de Follow Target With Pid Control se podría haber diseñado este behavior empleando un modo de control que hiciera más sencillo el diseño del behavior, sin embargo hacerlo así supondría no poder controlar la velocidad del robot en la ejecución de este behavior y puesto que la velocidad es importante para evitar posibles colisiones o para evitar que el dron que persigue se quede rezagado, este behavior se ha diseñado de manera que se tenga pleno control sobre la velocidad máxima.

La distancia de seguridad que mantienen los drones se calcula mediante la distancia euclídea de un dron al otro. Esta distancia también se emplea para autorregular la velocidad del dispositivo que ejecute este behavior.

Este behavior solo puede ser funcionar en el caso de que el estado del dron que lo ejecute sea HOVERING o FLYING.

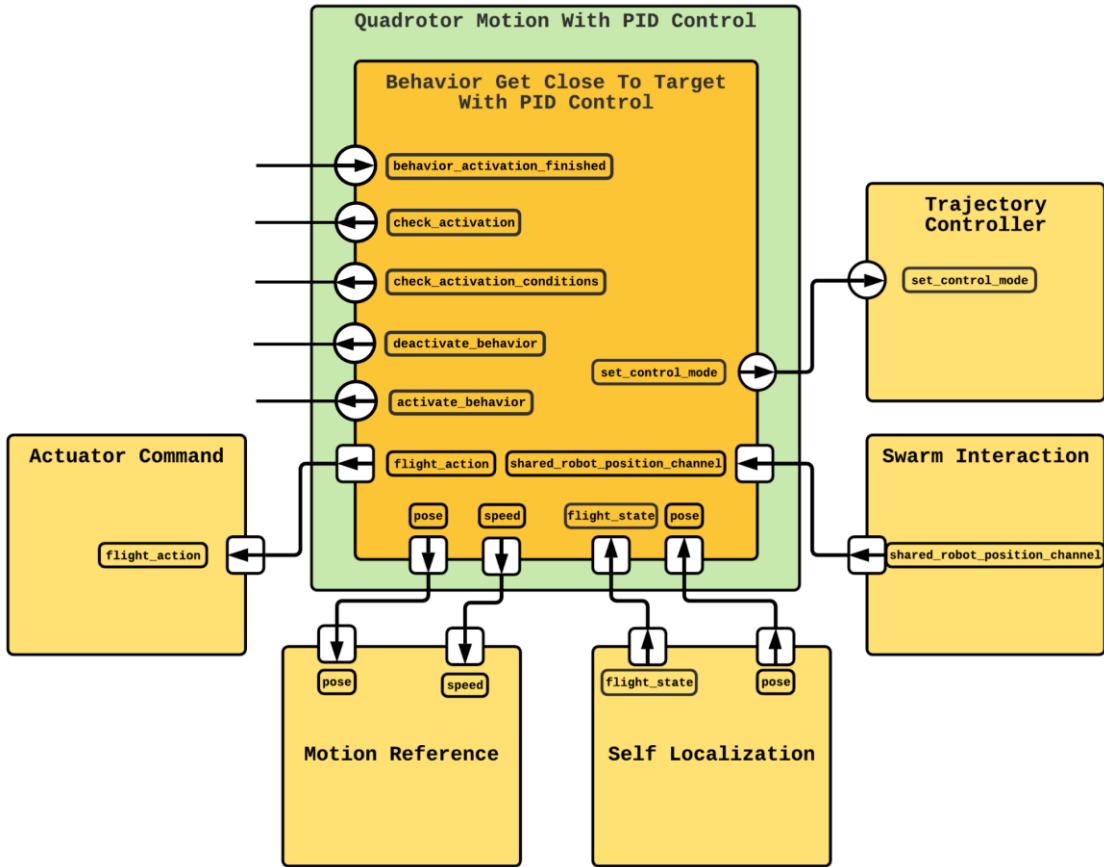


Fig. 3.10. Interacciones con topics y servicios de Get Close To Target With Pid Control.

En la figura 3.10 se puede observar cómo el comportamiento hace uso de servicios y de topics. Como se puede ver, este comportamiento actúa como cliente de trajectory_controller/set_control_mode y de behavior_activation_finished. También, publica información en los topics actuator_command/flight_action, motion_reference/pose y motion_reference/speed. También se suscribe a los topics self_localization/flight_state, self_localization/pose y /shared_robot_position_channel. Respecto a check_activation, check_activation_conditions, activate_behaviors y deactivate_behaviors mencionar que es el propio comportamiento el que actúa como servicio para estos cuatro clientes, los cuales están involucrados en la activación y desactivación del comportamiento.

Con el fin de mostrar cómo funciona internamente este behavior, en el algoritmo 5 se puede observar los pasos principales seguidos para programar la ejecución del comportamiento expresado en pseudocódigo.

Algoritmo 5. Get Close To Target

```

1: Obtener la posición del objetivo a perseguir  $x_{target}$ ,  $y_{target}$ ,  $z_{target}$ 
2: Obtener la posición del robot que llama al behavior  $x_{est}$ ,  $y_{est}$ ,  $z_{est}$ 
3: Mientras  $\neg d$ :
4:     Calcular distancia euclídea  $d$ , al objetivo y a los ejes "x" e "y",  $d_{eje}$ 
5:     Si Yaw = "target_facing" Entonces calcular el ángulo al que hay que
       orientar el robot  $Yaw_{sp}$  ,  $Yaw_{sp} = atan2(y_{target} - y_{est}, x_{target} - x_{est})$ 
6:     Cambiar set_control_mode a GROUND_SPEED
7:     Publicar  $Yaw_{sp}$  y  $z_{est}$ 
8:     Por cada eje  $x$  ,  $y$  :
9:         Si  $d > TARGET_DISTANCE$  Entonces calcular  $speed_{eje} = (speed \cdot d_{eje}) / d$ 
10:        Sino  $speed_{eje} = 0$ 
11:        Publicar  $speed_{eje}$ 
12:        Cambiar a MOVE flight_action
14:     $speed_{eje} = 0$ 
15:    Cambiar a HOVER flight_action

```

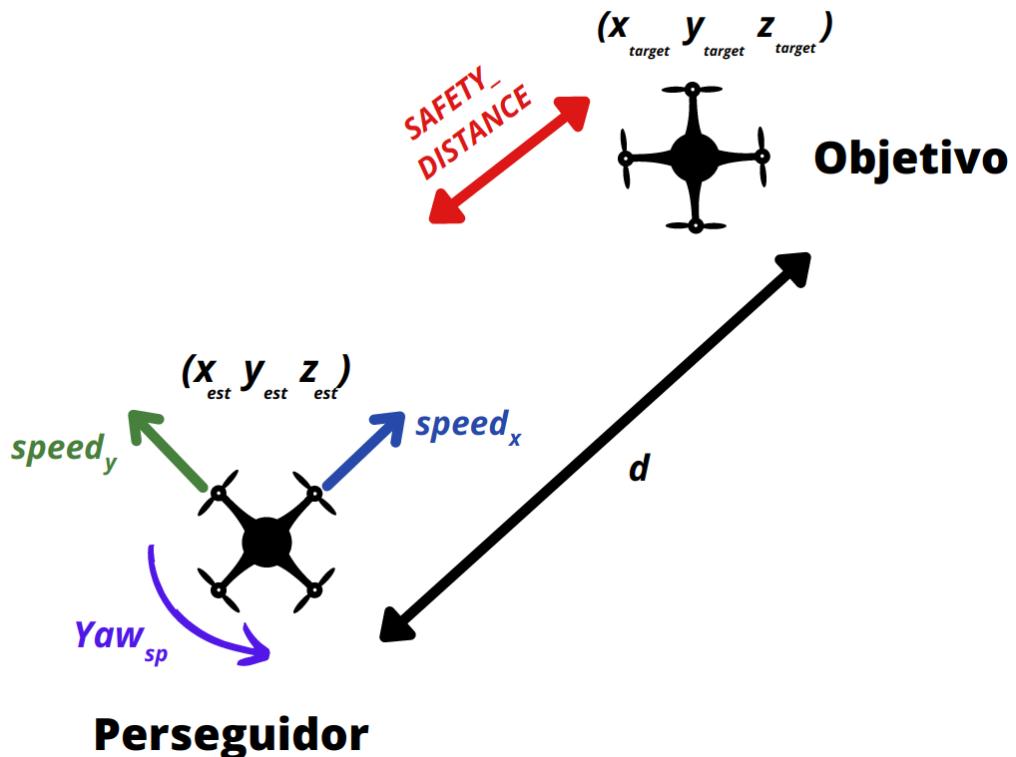


Fig. 3.11. Ilustración que muestra algunos de los elementos clave del algoritmo 5.

Como se puede observar en el algoritmo 4, una vez que se obtienen las posiciones de ambos drones en los planos x e y, se ejecuta un bucle (línea 3) en el que por cada iteración se calcula y se publica el ángulo Yaw (líneas 4 a 7) y la velocidad [6] (líneas 8 a 11). Este bucle finalizará cuando el eje "x" del dron que ejecuta el behavior se oriente hacia el robot deseado.

Para una visión más detallada de cómo se programaría este algoritmo, el [Anexo 5](#) muestra su implementación.

3.7 Keep Look At Target With PID Control

Cuando un robot activa este behavior, hace que este comience a rotar hasta situarse frontalmente con el robot u objetivo al que se quiere mirar hasta que un agente externo finaliza la ejecución de este behavior. Debido a su sencillez, este behavior posee un solo parámetro, target_name, que es un parámetro sin el cual el behavior no podría funcionar y que indica el identificador del objetivo al que mirar.

Un factor clave de este behavior es que depende de que el objetivo a mirar publique sus coordenadas en un topic al cual estará suscrito este behavior. Una vez que se publican las coordenadas, se hace rotar al robot empleando los ángulos de Euler hasta que queda situado frontalmente con el objetivo. Aunque el objetivo a mirar cambie de posición, el robot que active este behavior actualizará de nuevo su ángulo de guiñada para seguir mirando al objetivo hasta que se desactive este behavior externamente.

Este behavior solo se podrá activar si el dron que lo ejecuta se encuentra en estado de HOVERING o de FLYING.

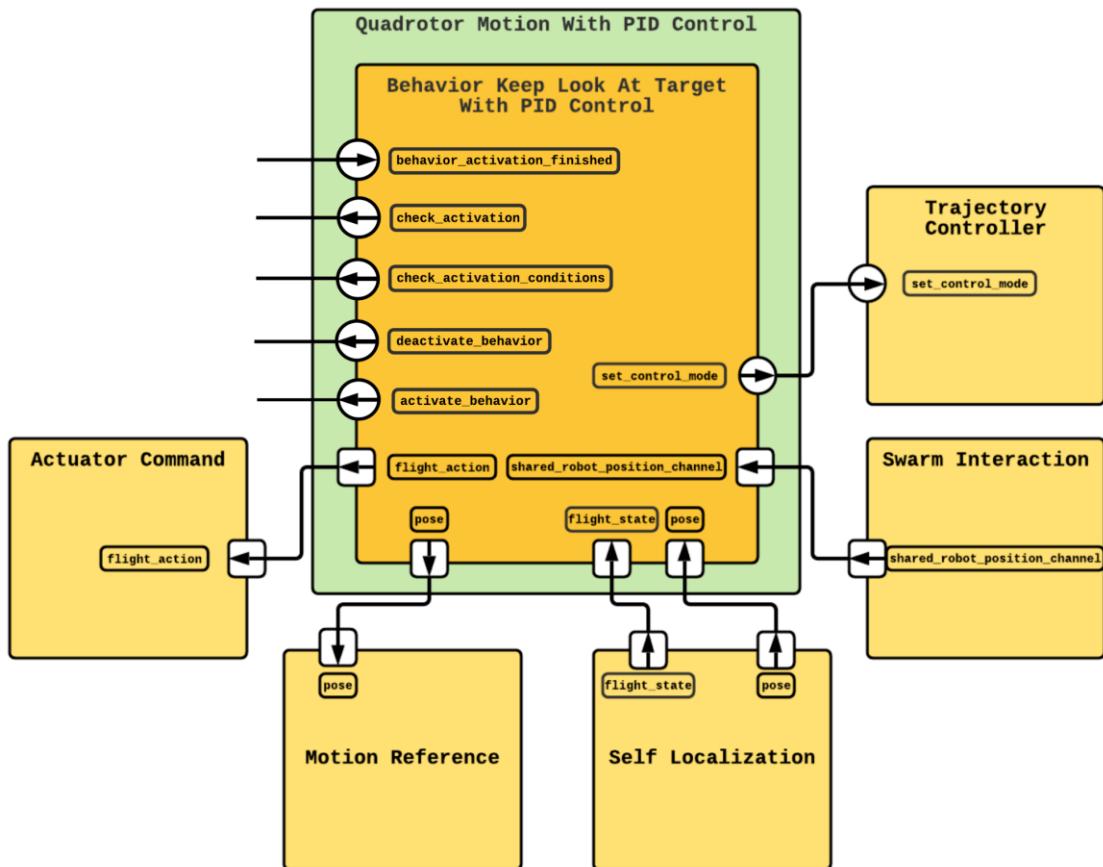


Fig. 3.12. Interacciones con topics y servicios de Keep Look At Target With Pid Control.

La figura 3.12 muestra cómo interactúa este behavior con los distintos topics y servicios de los que hace uso. Como se puede ver, este comportamiento publica información en los topics actuator_command/flight_action, y motion_reference/pose y se suscribe a los topics self_localization/flight_state,

`self_localization/pose` y `/shared_robot_position_channel`. También, este behavior actúa como cliente de `trajectory_controller/set_control_mode` y de `behavior_activation_finished`. Respecto a `check_activation`, `check_activation_conditions`, `activate_behaviors` y `deactivate_behaviors` mencionar que es el propio comportamiento el que actúa como servicio para estos cuatro clientes.

Con el objetivo de mostrar el funcionamiento de este behavior, en el algoritmo 6 se puede observar los pasos principales seguidos para programar la ejecución del comportamiento expresado en pseudocódigo.

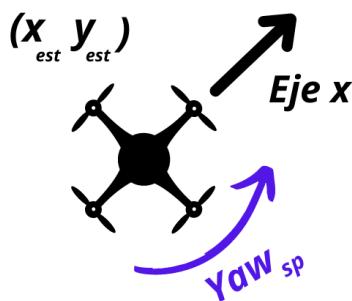
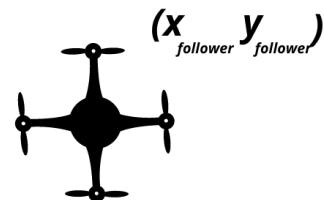
Algoritmo 6. Keep Look At Target

```

1: Obtener la posición del objetivo al que mirar  $x_{target}$ ,  $y_{target}$ 
2: Obtener la posición del robot que llama al behavior  $x_{est}$ ,  $y_{est}$ 
3: Mientras  $\neg$  inhibido:
4:     Calcular el ángulo al que hay que orientar el robot  $Yaw_{sp}$  ,
 $Yaw_{sp} = atan2(y_{target} - y_{est}, x_{target} - x_{est})$ 
5:     Cambiar set_control_mode a SPEED_3D
6:     Publicar  $Yaw_{sp}$ 
7:     Cambiar a MOVE flight_action
8:     Cambiar a HOVER flight_action

```

Dron observado



Observador

Fig. 3.13. Ilustración que muestra algunos de los elementos clave del algoritmo 6.

Como se puede observar en el algoritmo 6, una vez que se obtienen las posiciones de ambos drones en los planos x e y, se ejecuta un bucle (línea 3)

en el que por cada iteración se calcula y se publica el ángulo Yaw (líneas 4 a 7). Este bucle terminará de iterar cuando un agente externo, como una orden de finalizar el behavior se active.

Para una visión más detallada de cómo se programaría este algoritmo, el [Anexo 6](#) muestra su implementación en pseudocódigo.

3.8 Keep Target Altitude With PID Control

Al activarse este behavior, hace que el robot que lo ejecuta se mantenga a la misma altura que otro robot (target) o a una altura personalizada en función de la altura del target hasta que un agente externo desactive la ejecución del behavior. Los argumentos de los que hace uso este behavior son los siguientes:

- target_name, es un parámetro obligatorio de tipo string que indica la identificación del dron del que quiero tomar la altitud como referencia.
- altitude_regulation, es un parámetro opcional cuyo valor por defecto es 0.0 metros. Sirve para aumentar (o disminuir) la altitud a la que quiero situar el robot respecto al target.
- maximum_speed, un número real que indica la velocidad máxima a la que debe ascender el robot en metros por segundo. Es un parámetro opcional, que no tiene un valor por defecto.

Para conseguir el propósito de mantener la misma altura respecto a la de otro robot (target), este, debe de publicar su posición a través de un topic. El robot que active este behavior por lo tanto debe suscribirse a este topic y para recibir la posición que envía el target.

Una vez que el robot que activa el behavior posee las coordenadas del target, comienza a ascender (o descender) a una velocidad constante hasta alcanzar la altura deseada. Este behavior se desactivará cuando un agente externo lo ordene y, por lo tanto, la altitud del robot seguirá ajustándose hasta que se desactive.

Cabe destacar que, este behavior solo se podrá activar si el dron que lo ejecuta está en estado de HOVERING o de FLYING.

Respecto al parámetro maximum_speed, comentar que carece de valor por defecto ya que se hace uso del modo de control GROUND_SPEED en el caso de que no se establezca una velocidad.

La figura 3.14 muestra cómo interactúa este behavior con los distintos topics y servicios de los que hace uso. Como se puede ver, este comportamiento publica información en los topics actuator_command/flight_action, motion_reference/speed y motion_reference/pose y se suscribe a los topics self_localization/flight_state, self_localization/pose y /shared_robot_position_channel. Respecto a check_activation, check_activation_conditions, activate_behaviors y deactivate_behaviors mencionar que es el propio comportamiento el que actúa como servicio para estos cuatro clientes. También, este behavior actúa como cliente de trajectory_controller/set_control_mode y de behavior_activation_finished.

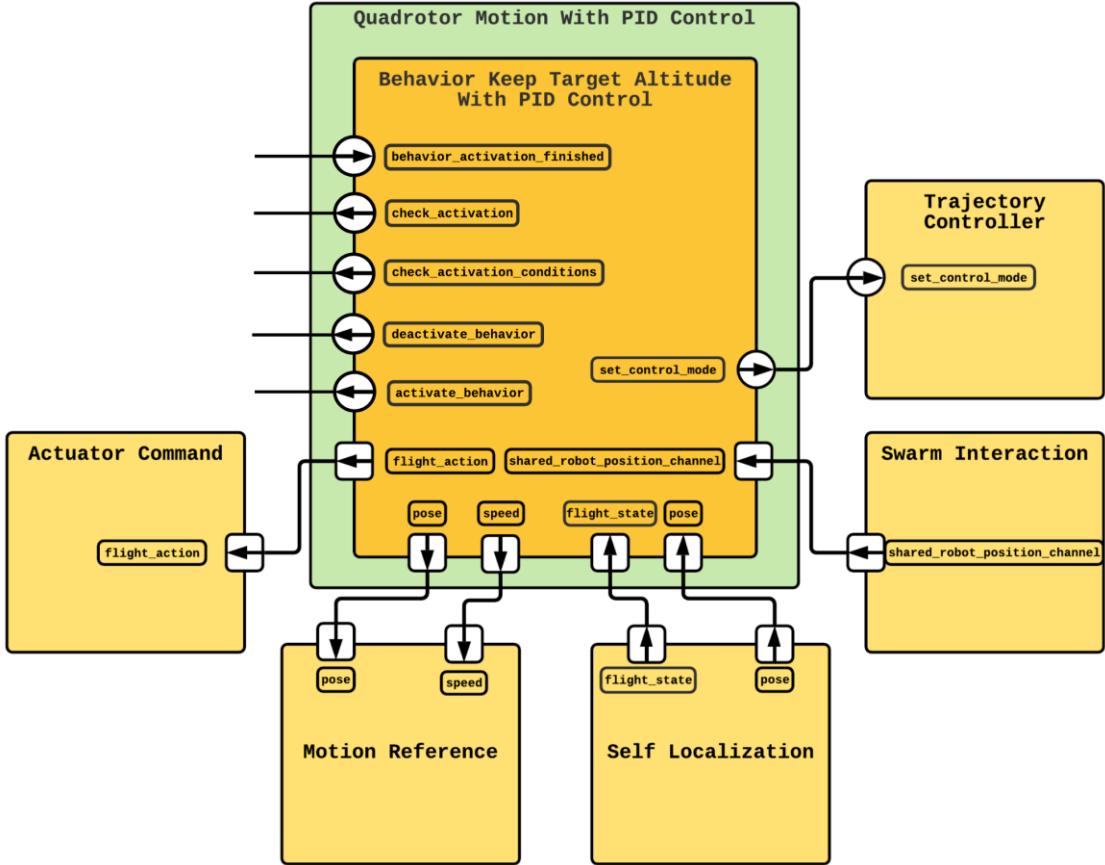


Fig. 3.14. Interacciones con topics y servicios de Keep Target Altitude With Pid Control.

Con el fin de mostrar de una manera más técnica la ejecución de este comportamiento, en el algoritmo 7 se puede observar los pasos principales seguidos en el desarrollo del comportamiento expresado en pseudocódigo.

Algoritmo 7. Keep Target Altitude

```

1: Obtener la altitud del objetivo a igualar la altitud  $z_{target}$ 
2: Obtener la altitud del robot que llama a keep Target Altitude  $z_{est}$ 
3: Mientras  $\neg$ inhibido:
4:   Si  $\neg$ VELOCIDAD Entonces cambiar set_control_mode a GROUND_SPEED
5:   Si VELOCIDAD y  $\neg$ ALTITUD_ALCANZADA Entonces cambiar set_control_mode a SPEED_3D
6:   Publicar altitud_final si set_control_mode = GROUND_SPEED
7:   Publicar  $speed_z = (speed \cdot d_z) / d$  si set_control_mode = SPEED_3D
8:   Cambiar a MOVE flight_action
9:    $speed_z = 0$ 
10:  Cambiar a HOVER flight_action

```

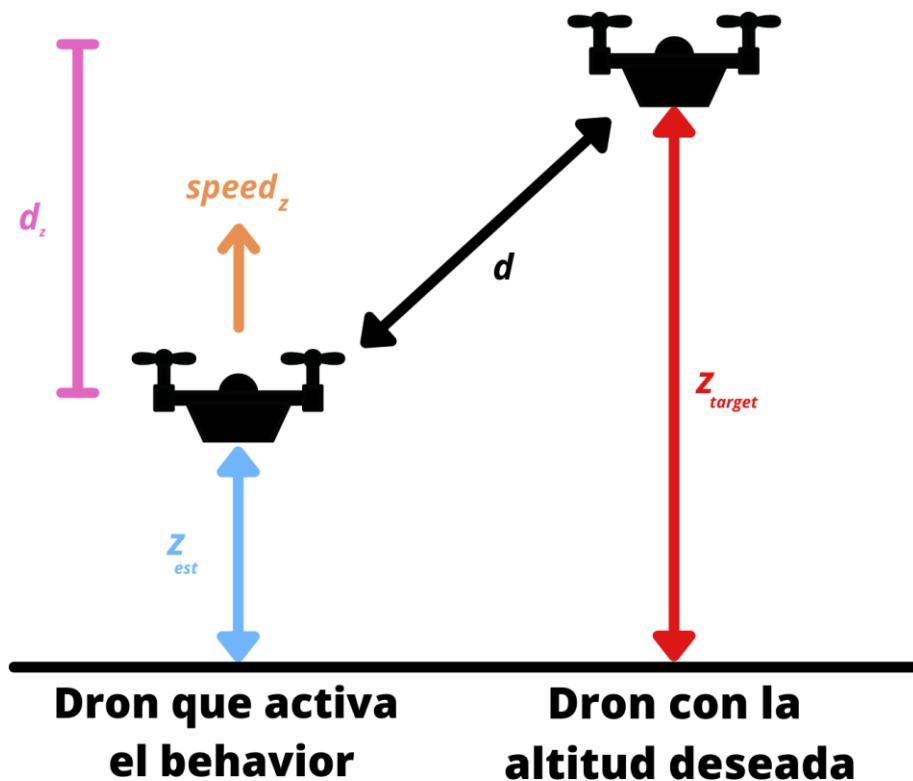


Fig. 3.15. Ilustración que muestra algunos de los elementos clave del algoritmo 7.

Como se puede observar en el algoritmo 7, una vez que se obtienen las alturas de ambos drones y dependiendo del modo de control empleado se publicará la velocidad [6] a la que debe ascender el dron (líneas 4-8). Una vez se alcanza la altitud deseada, finalizará la iteración del bucle y el dron quedará flotando en el aire. En el [Anexo 7](#) se muestra en pseudocódigo cómo se ha realizado la implementación de este algoritmo.

4 Pruebas

En este capítulo, se muestran las distintas pruebas de verificación que se han llevado a cabo para comprobar que el software desarrollado realiza las acciones que se espera que haga. Para comprobar todo esto, se ha hecho uso del simulador Gazebo y de los bags de ROS, con los que se elaborarán gráficas empleando el programa PlotJuggler3.

4.1 Pruebas de ejecución individual

En este apartado, se realizan diversas pruebas de ejecución sobre cada behavior con el fin de verificar que cada una de las partes que lo componen funcionen correctamente. Para este fin, se medirán una serie de métricas y se probará cada behavior mediante la realización de misiones.

4.1.1 Pruebas de ejecución individual: Follow Target With PID Control

A continuación, se muestra una tabla mostrando las pruebas de ejecución individual realizadas sobre este behavior:

TABLA V

PRUEBAS DE EJECUCIÓN REALIZADAS PARA FOLLOW TARGET WITH PID CONTROL

Prueba	Resultado esperado
Se llama al behavior cuando el estado del dron es distinto a FLYING o HOVERING	El behavior no se activa
Se llama al behavior cuando el estado del dron es FLYING o HOVERING	El behavior se activa
El parámetro target_name se omite	El behavior no se activa
El parámetro target_name es incorrecto	El behavior no funciona correctamente
El parámetro safety_distance se omite	El behavior se activa con una distancia por defecto de 2.0 metros
El parámetro safety_distance es incorrecto	El behavior no se activa
El parámetro maximum_speed se omite	El behavior se activa con una velocidad por defecto de 0.3 m/s
El parámetro maximum_speed es incorrecto	El behavior no se activa
El parámetro yaw se omite	El behavior se activa con yaw = constant por defecto
El parámetro yaw es incorrecto	Error YAML
Ejecución la función stopTask sobre este behavior	El behavior se desactiva

A continuación, se muestran una serie de pruebas de ejecución que, al ser más complejas a la hora de verificarlas, es más fácil probar su resultado midiendo una serie de métricas que ilustren cada caso. Decir que en este caso

el dron con identificador “drone112” es el que ejecuta Follow Target With PID Control:

- El dron va como máximo a la velocidad que se le indica. En esta prueba se ejecutó el behavior con distintas velocidades para comprobar que la velocidad máxima de persecución que recibía el behavior en el parámetro `maximum_speed` no era sobrepasada en ningún momento de la ejecución. Con este fin, se han realizado pruebas con dos velocidades distintas, 0.4 m/s (figura 4.1) y 0.8 m/s (figura 4.2) y se ha empleado el topic `self_localization/speed`, que mide la velocidad real a la que se mueve el robot. Se han medido la velocidad de los ejes “x” e “y”, ya que la velocidad en el eje “z” (velocidad a la que se aumenta o reduce la altitud) no contribuye de manera directa en la persecución, sólo sitúa al dron que activa el behavior a la misma altitud del dron perseguido.

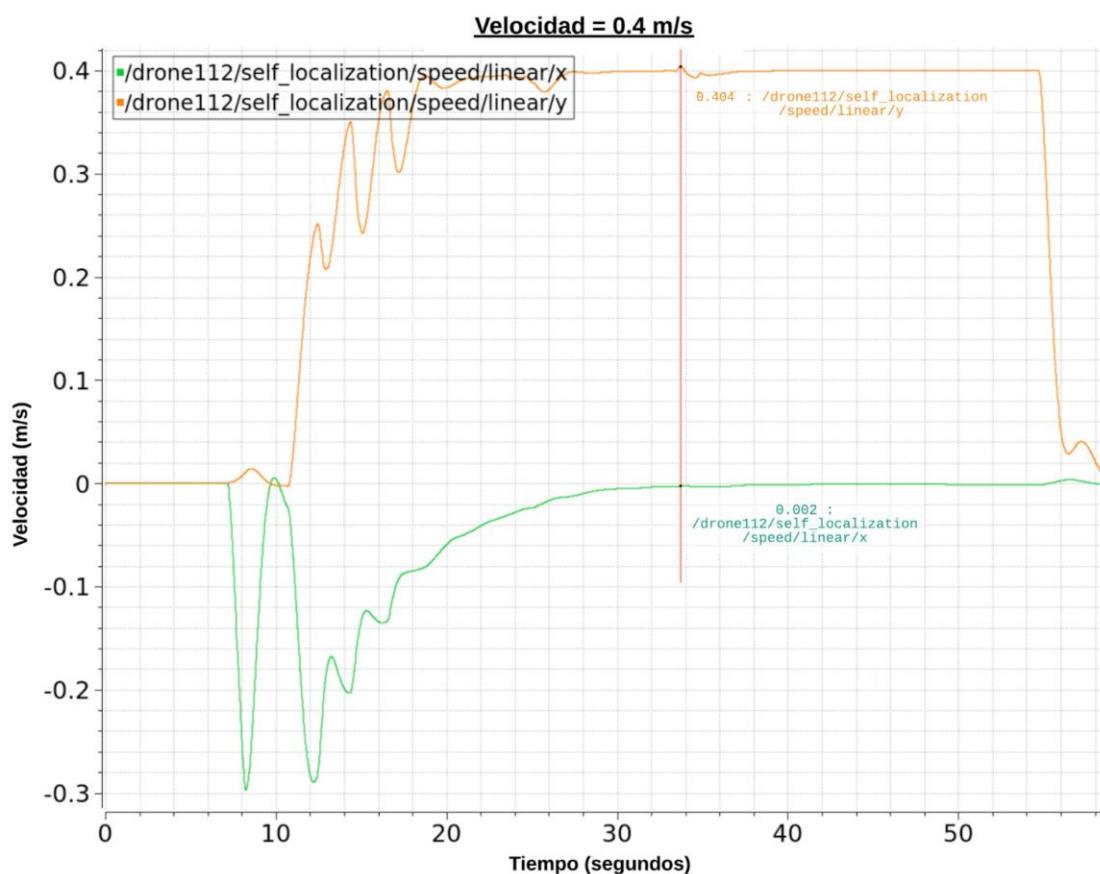


Fig. 4.1. Gráfica de velocidad en los ejes “x” e “y” del topic `self_localization/speed` con velocidad = 0.4 m/s.

Tal y como se puede observar en la figura 4.1 el dron no sobrepasa la velocidad deseada salvo en un breve lapso de medio segundo en el que se sobrepasa la velocidad en +0.004 m/s en el eje “y”, un valor bastante pequeño y que no supone ningún problema en la ejecución del comportamiento.

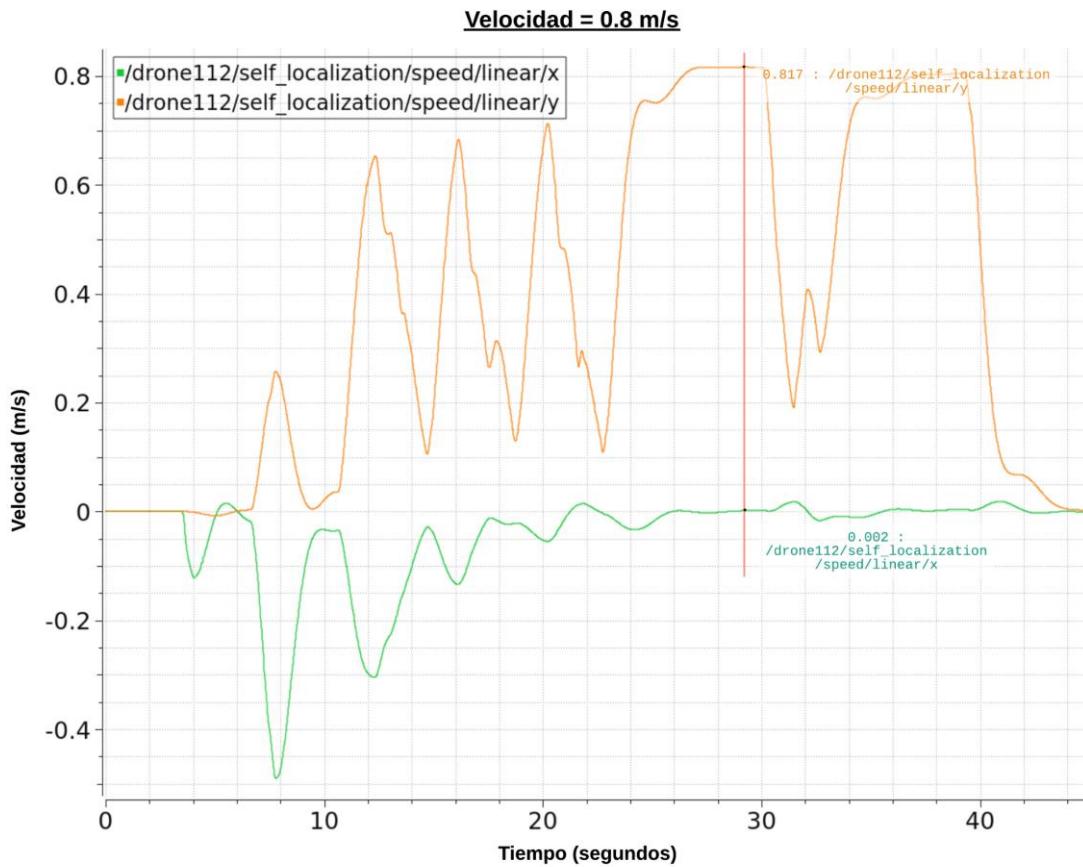


Fig. 4.2. Gráfica de velocidad en los ejes “x” e “y” del topic self_localization/speed con velocidad = 0.8 m/s.

En la figura 4.2, y al igual que en el caso representado por la figura 4.1, sigue habiendo un desfase de velocidad, aunque esta vez más acentuado, llegando a sobrepasar la velocidad deseada en un máximo de +0.017 m/s. Sin embargo, sigue siendo un valor muy bajo como para tenerlo en cuenta.

- Prueba de altitud. Esta prueba consiste en observar si la altitud del dron que ejecuta el behavior es similar a la del dron que se persigue. Esta prueba consiste en verificar que el robot que active este behavior alcanza una altura muy similar (± 0.1 metros) a la del robot perseguido. Para ello, se empleará el topic self_localization/pose que da la posición real del dron en un momento concreto.

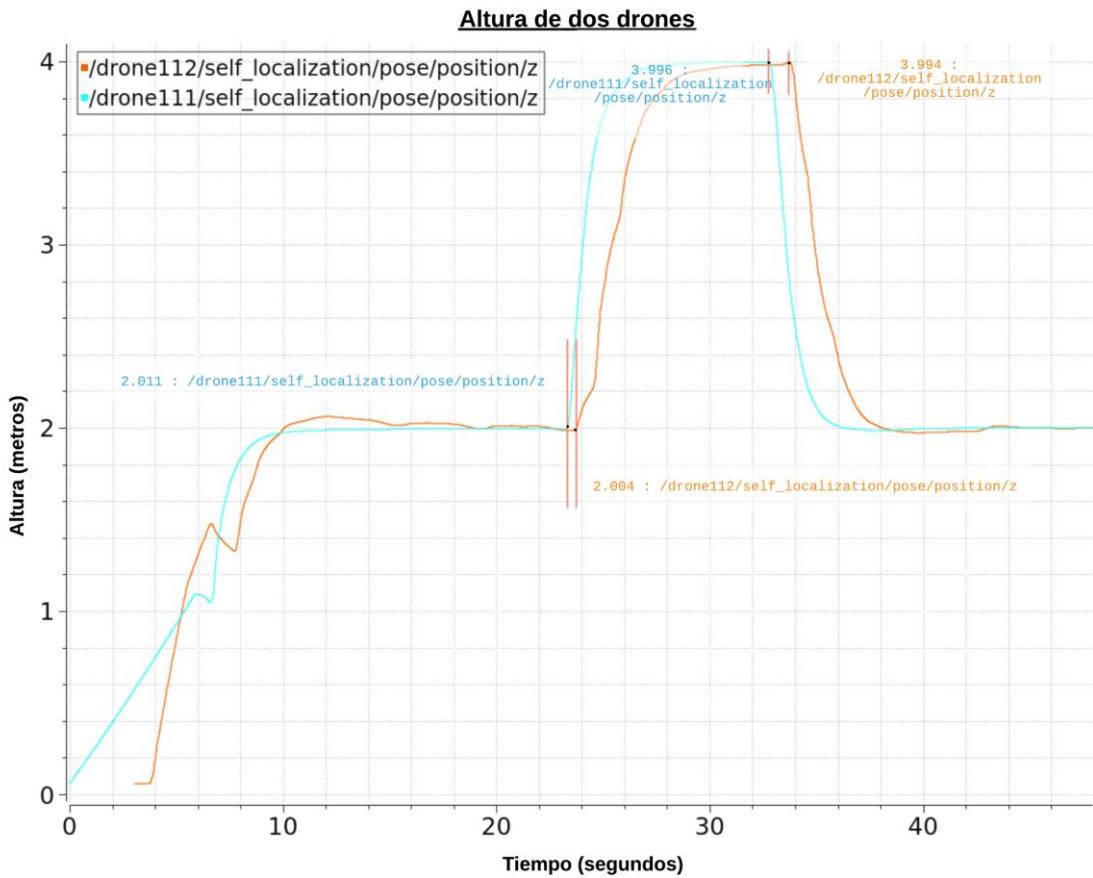


Fig. 4.3. Gráfica de altitud alcanzada por los drones 111 y 112.

Como se puede observar en la figura 4.3, para esta prueba se han medido las altitudes alcanzadas por los drones con identificador “drone111” (que es el dron que se desea perseguir) y “drone112” (que es el dron que activa el behavior teniendo en cuenta el retraso en la activación del behavior por parte de este último dron). Hay que recordar que el dron que activa este behavior recibe con un ligero retraso la altitud a la que debe colocarse y es por este motivo que la gráfica marrón está ligeramente más atrasada en el tiempo que la azul, y por lo tanto los puntos escogidos para medir la altitud también están más atrasados. Tal y como se puede ver, esta prueba se supera con creces pues las diferencias de altura son del orden de centésimas de metro o incluso milésimas de metro, siendo estas métricas muy inferiores al objetivo con el que considerar esta prueba como exitosa (± 0.1 metros).

- Prueba de distancia. En esta prueba se observará si el dron que ejecuta el behavior mantiene una distancia mínima que se le indica por parámetro si se acerca mucho al dron perseguido. Para realizar esta prueba se ha tomado como referencia la distancia que mantienen los drones en el eje “y”. La distancia mínima que deben mantener es de 2.0 metros.

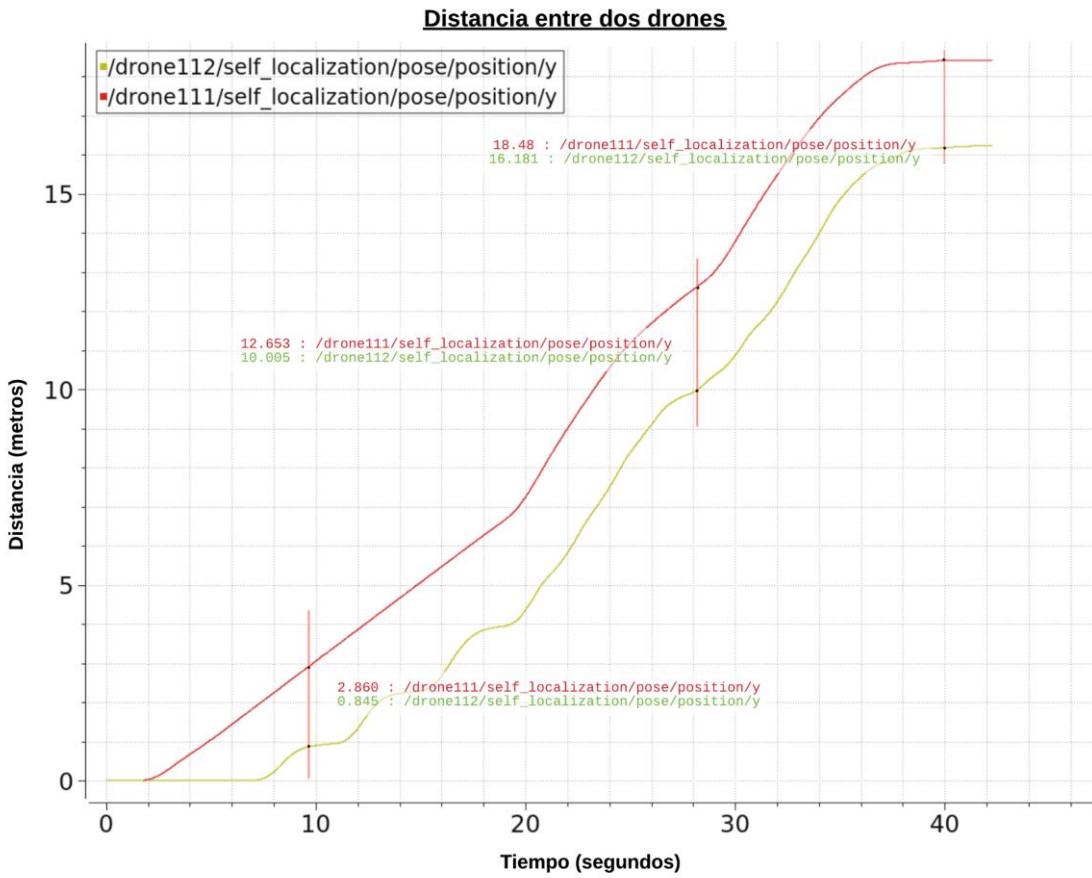


Fig. 4.4. Gráfica que muestra la distancia entre los drones 111 y 112.

En la figura 4.4, se ve como se han cogido tres muestras de la distancia que mantienen los drones y en los tres casos se mantiene la condición de que esta distancia sea como mínimo de 2.0 metros.

Hasta ahora, se han realizado pruebas sobre ciertos aspectos del comportamiento, pero el éxito de estas pruebas no determina claramente si el behavior realiza la acción que debe hacer. Es por este motivo que, para verificar el correcto funcionamiento del comportamiento se ha realizado una prueba de dicho comportamiento junto con behaviors preexistentes en Aerostack. Esto se puede observar en los ficheros de misión de los drones (figuras 4.5 y 4.6).

```
def mission():
    print("Starting mission...")
    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')
    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')
    print("Taking off...")
    mxc.executeTask('TAKE_OFF')
    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 6, 2], [2, 12, 4], [2, 18, 2]], yaw="path_facing")
    print('Mission completed.')
```

Fig. 4.5. Fichero mission1.py, ejecutado por el dron con identificador drone111.

```

def mission():
    print("Starting mission...")
    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following drone...")
    mxc.executeTask('FOLLOW_TARGET_WITH_PID_CONTROL', target_name = "drone111", safety_distance = 2.0,
                    maximum_speed = 0.4)

    print('Mission completed.')

```

Fig. 4.6. Fichero mission2.py, ejecutado por el dron con identificador drone112.

El funcionamiento de este comportamiento se puede considerar correcto si el dron que activa el behavior es capaz de perseguir a otro dron sin impactar con él.

En el siguiente enlace se puede observar el resultado de la ejecución de la prueba: https://youtu.be/VDuiEkjg2_A

4.1.2 Pruebas de ejecución individual: Move Away From Robot With PID Control

A continuación, se muestra una tabla mostrando las pruebas de ejecución realizadas sobre este behavior.

TABLA VI

PRUEBAS DE EJECUCIÓN REALIZADAS PARA MOVE AWAY FROM ROBOT WITH PID CONTROL

Prueba	Resultado esperado
Se llama al behavior cuando el estado del dron es distinto a FLYING o HOVERING	El behavior no se activa
Se llama al behavior cuando el estado del dron es FLYING o HOVERING	El behavior se activa
El parámetro follower_name se omite	El behavior no se activa
El parámetro follower_name es incorrecto	El behavior no funciona correctamente
El parámetro separation_distance se omite	El behavior se activa con una distancia por defecto de 3.0 metros
El parámetro separation_distance es incorrecto	El behavior no se activa
El parámetro maximum_speed se omite	El behavior se activa con una velocidad por defecto de 0.3 m/s
El parámetro maximum_speed es incorrecto	El behavior no se activa
El parámetro yaw se omite	El behavior se activa con yaw = constant por defecto
El parámetro yaw es incorrecto	Error YAML
Ejecución de la función stopTask sobre este behavior	El behavior se desactiva

A continuación, se muestran una serie de pruebas de ejecución que, al ser más complejas de verificar, se ha decidido emplear gráficas para su comprobación. En este caso el dron con identificador “drone111” ejecutará Move Away From Robot With PID Control:

- Prueba de velocidad. El objetivo de esta prueba es comprobar que el dron que ejecuta el behavior va como máximo a la velocidad que se le indica el parámetro maximum_speed. En esta prueba se ha ejecutado el comportamiento empleando dos velocidades, 0.3 m/s (figura 4.7) y 0.6 m/s (figura 4.8). Se ha empleado el topic self_localization/speed, que mide la velocidad real a la que se mueve el robot.

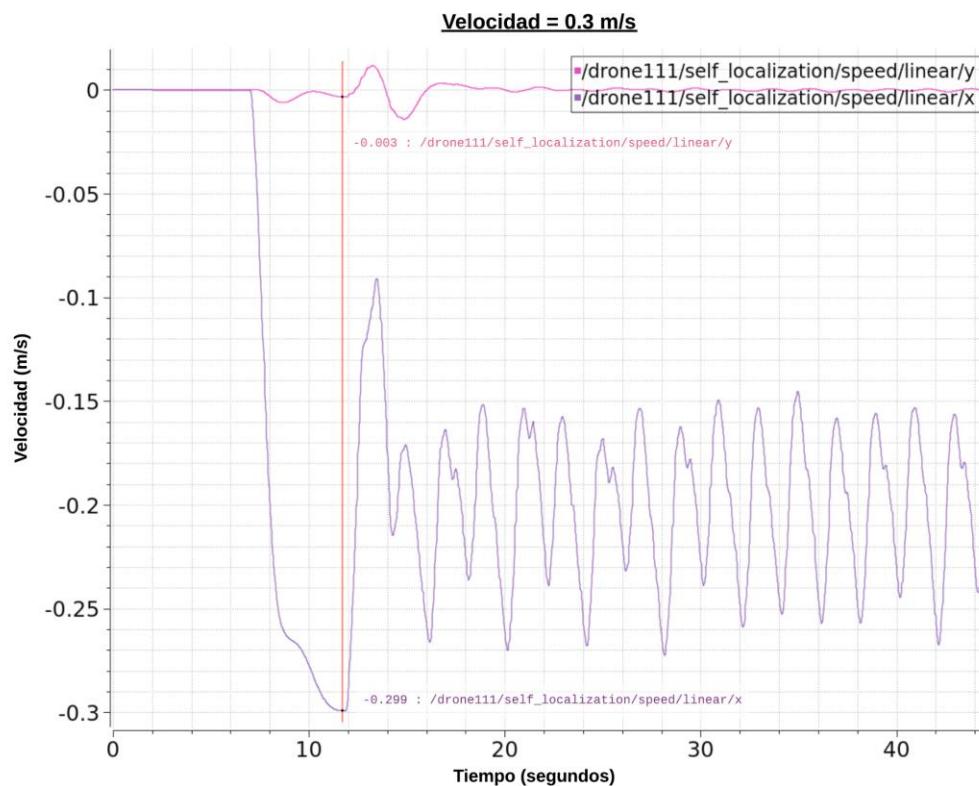


Fig. 4.7. Gráfica de velocidad en los ejes “x” e “y” del topic self_localization/speed con velocidad = 0.3 m/s.

Como se puede ver en la figura 4.7 el dron no sobrepasa la velocidad deseada en ningún momento de la ejecución del behavior. Que el robot aéreo no alcance la velocidad máxima a partir del segundo 12 se debe a que el dron debe frenar en el caso de que la distancia de separación representada por el parámetro separation_distance ha sido alcanzada.

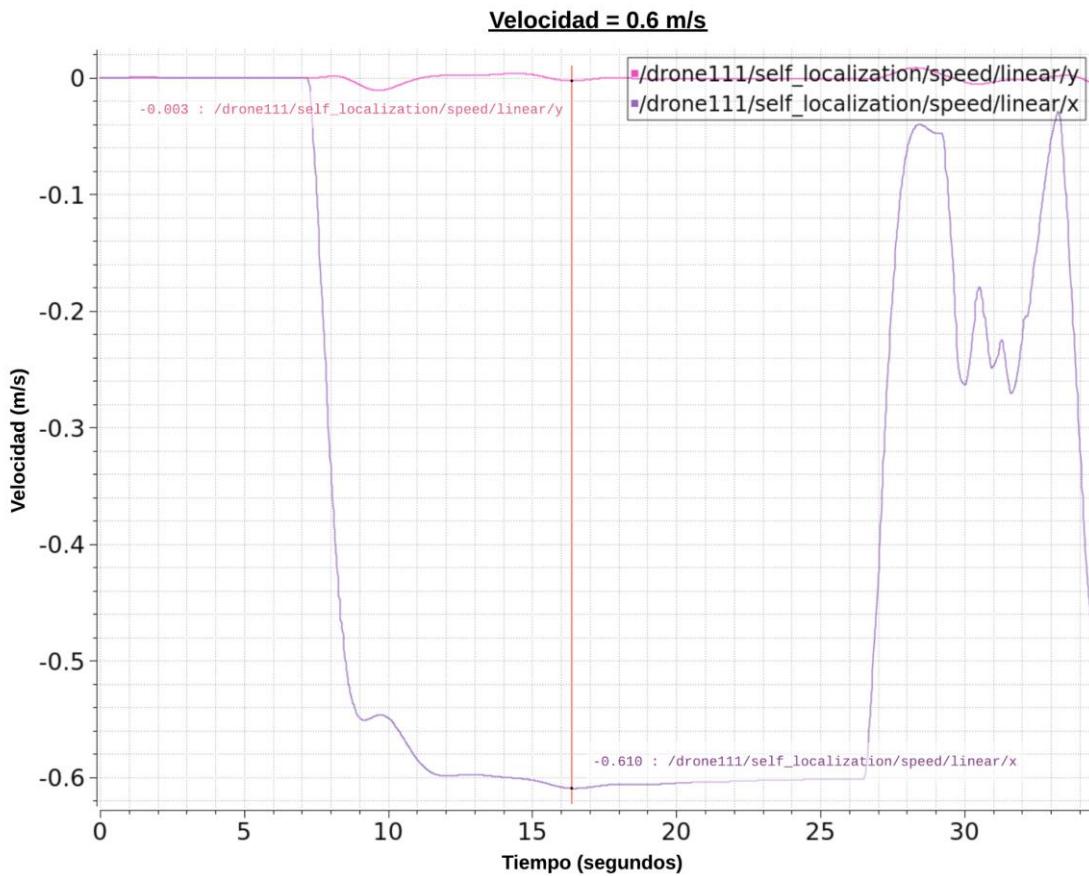


Fig. 4.8. Gráfica de velocidad en los ejes “x” e “y” del topic self_localization/speed con velocidad = 0.6 m/s.

En el caso de gráfica la figura 4.8 si que hay un ligero desfase de velocidad de +0.01 m/s aunque unos segundos después la velocidad se estabiliza en 0.6 m/s. Al ser el desfase pequeño y que después se estabiliza la velocidad, se puede dar como válidos los resultados obtenidos.

- Prueba de altitud. Esta prueba consiste en ver si la altitud del dron que ejecuta el behavior es similar a la del dron del que huye. Esta prueba consiste en verificar que el robot que active este behavior alcanza una altura muy similar (± 0.1 metros) a la del robot que persigue. Para ello se empleará el topic self_localization/position.

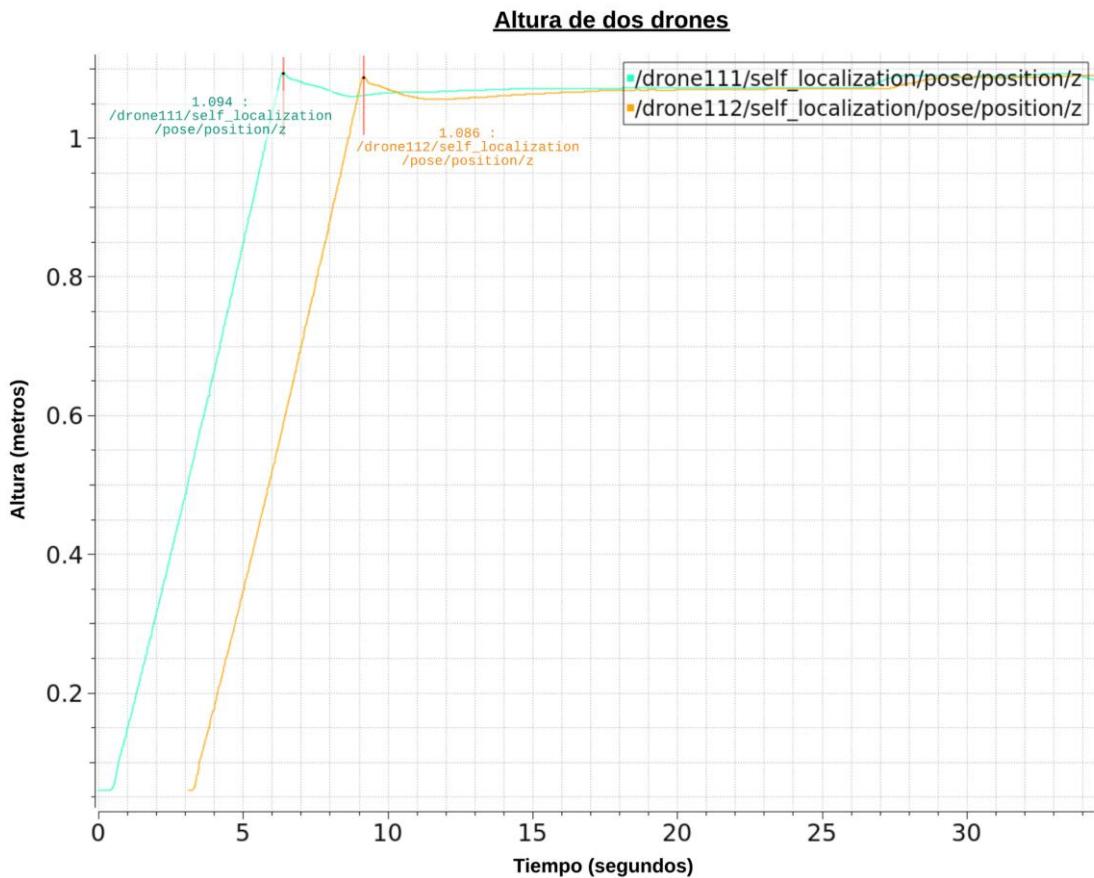


Fig. 4.9. Gráfica de altitud alcanzada por los drones 111 y 112.

En la figura 4.9, se han medido las altitudes alcanzadas teniendo en cuenta el retraso en la activación del behavior por parte del dron con identificador “drone112” y, tal y como se puede observar, las diferencias de altura son del orden de centésimas de metro siendo estas métricas muy inferiores al objetivo con el que considerar esta prueba como exitosa (diferencia de ± 0.1 metros).

- Prueba de distancia. En esta prueba se comprueba si el dron que ejecuta el behavior mantiene una distancia que se le indica por el parámetro separation_distance si se acerca mucho dron que le persigue, es decir, siempre se debe mantener una distancia de un dron a otro de 3 metros como mínimo. Para realizar esta prueba se ha tomado como referencia la distancia que mantienen los drones en el eje “x”. La distancia mínima que deben mantener es de 3.0 metros.

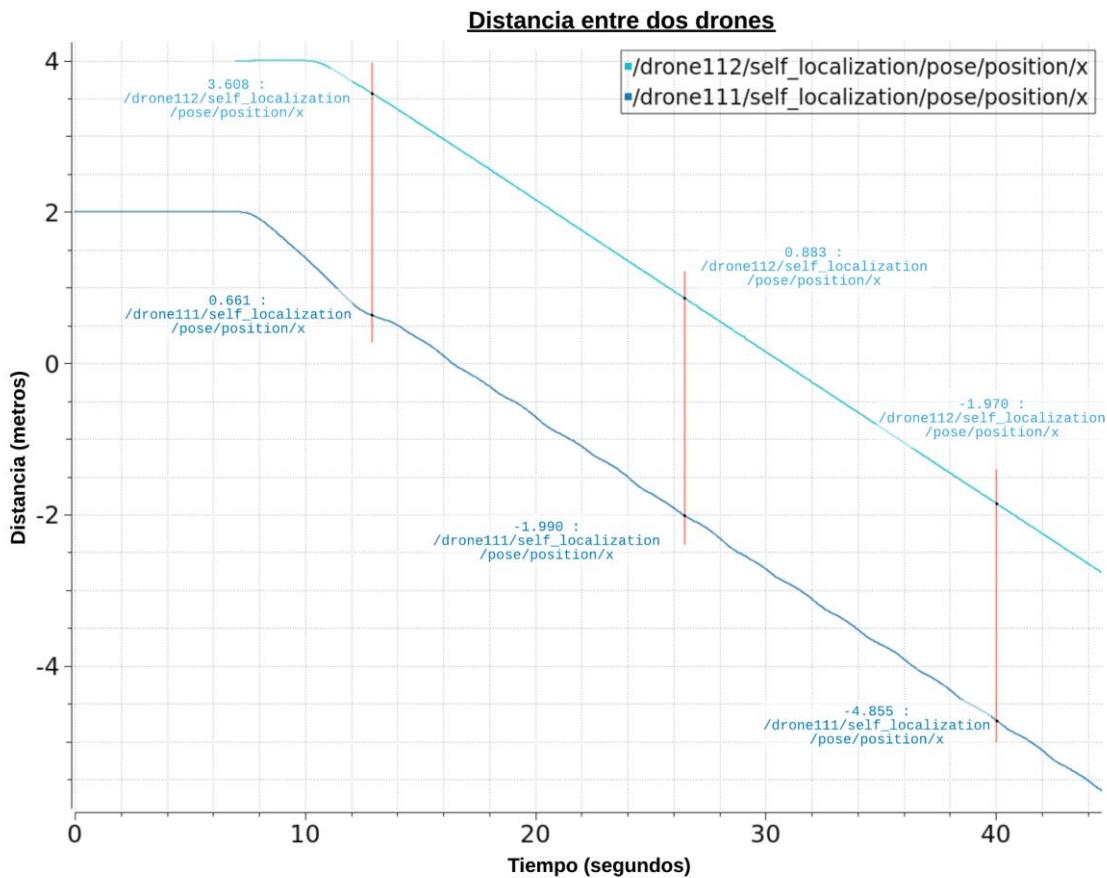


Fig. 4.10. Gráfica que muestra la distancia entre los drones 111 y 112.

La figura 4.10 muestra que la distancia de 3.0 metros se mantiene entre los dos drones salvo en algunos casos en los que por unas décimas de metro esto no ocurre. Esto sucede debido a que cuando el dron que activa este behavior (drone111) alcanza la distancia de 3.0 metros, frena a la espera de que el otro dron se acerque más y así acelerar otra vez para seguir manteniendo la distancia. También decir que en los primeros 8 segundos de la gráfica la distancia es de 2.0 metros pues en ese momento el behavior no se había activado. Por lo tanto, se puede dar esta prueba como correcta.

Al igual que en el behavior anterior, las pruebas exitosas sobre distintos aspectos del comportamiento no garantiza que el behavior realice la acción que debe hacer. Es por este motivo que, para verificar el correcto funcionamiento del comportamiento se ha realizado una prueba en la que se emplean behaviors preexistentes en Aerostack junto con Move Away From Robot tal y como se ve en los ficheros de misión (figuras 4.11 y 4.12).

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Moving away from follower...")
    mxc.executeTask('MOVE_AWAY_FROM_ROBOT_WITH_PID_CONTROL', follower_name = "drone112",
                    separation_distance = 3.0, maximum_speed = 0.6)

    print('Mission completed.')

```

Fig. 4.11. Fichero mission1.py, ejecutado por el dron con identificador drone111.

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following drone...")
    mxc.executeTask('FOLLOW_TARGET_WITH_PID_CONTROL', target_name = "drone111", safety_distance = 2.0,
                    maximum_speed = 0.4)

    print('Mission completed.')

```

Fig. 4.12. Fichero mission2.py, ejecutado por el dron con identificador drone112.

El funcionamiento de este comportamiento se puede considerar correcto si el dron que activa el behavior es capaz de huir de otro dron si este se sitúa muy cerca de él.

En el siguiente enlace se puede observar el resultado de la ejecución de la prueba: https://youtu.be/Dv_oxvNIUXM

4.1.3 Pruebas de ejecución individual: Reach Target Altitude With PID Control

A continuación, se muestra una tabla mostrando las pruebas de ejecución realizadas sobre este behavior.

TABLA VII

PRUEBAS DE EJECUCIÓN REALIZADAS PARA REACH TARGET ALTITUDE WITH PID CONTROL

Prueba	Resultado esperado
Se llama al behavior cuando el estado del dron es distinto a FLYING o HOVERING	El behavior no se activa
Se llama al behavior cuando el estado del dron es FLYING o HOVERING	El behavior se activa
El parámetro target_name se omite	El behavior no se activa
El parámetro target_name es incorrecto	El behavior no funciona correctamente
El parámetro shift se omite	El behavior se activa con una variación por defecto de 0.0 metros
El parámetro shift es incorrecto	El behavior no se activa
El parámetro maximum_speed se omite	El behavior se activa con controlMode= GROUND_SPEED
El parámetro maximum_speed es incorrecto	El behavior no se activa
Ejecución de la función stopTask sobre este behavior	El behavior se desactiva

A continuación, se muestran una serie de pruebas de ejecución complejas de verificar, para las que se ha decidido emplear gráficas para su comprobación. En este caso el dron con identificador “drone112” ejecutará Reach Target Altitude With PID Control:

- Prueba de altitud. Esta prueba consiste en ver si la altitud del dron que ejecuta el behavior es similar a la del dron al que se quiere igualar la altitud. Esta prueba consiste en verificar que el robot que activa este behavior alcanza una altura muy similar (± 0.1 metros) a la del robot a igualar. Se empleará el topic self_localization/position y se expondrán dos casos, uno con valor del parámetro shift = 0.0 y otro valor -1.0.

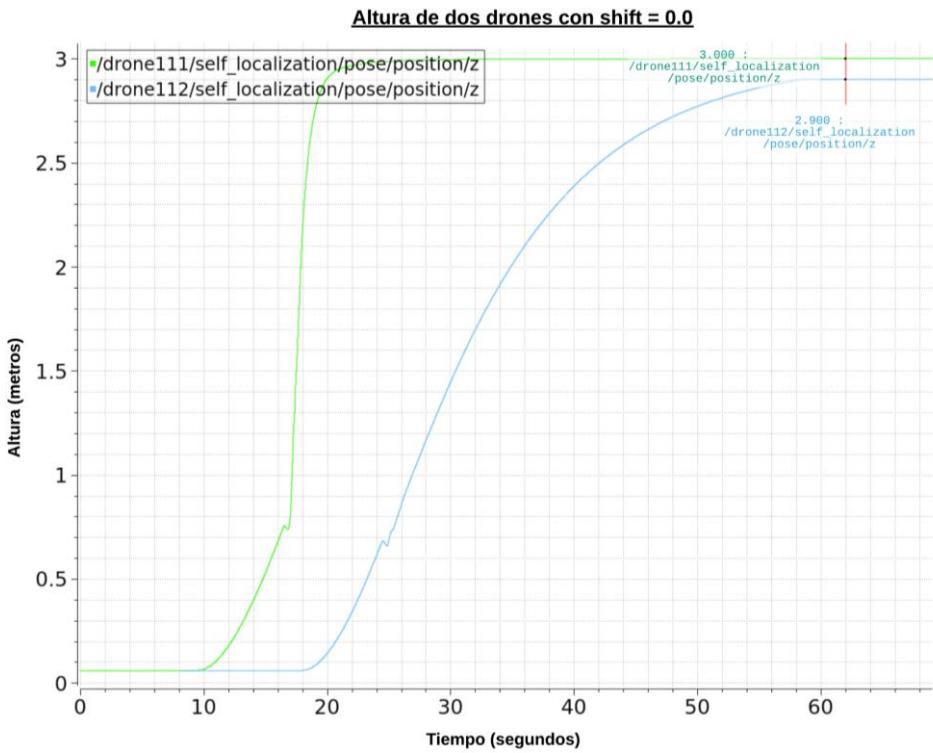


Fig. 4.13. Gráfica de altitud alcanzada por los drones 111 y 112 sin shift.

En la figura 4.13, se han medido las altitudes alcanzadas con una variación (parámetro shift) igual a 0.0 metros. Tal y como se puede ver, esta prueba se supera pues el error está en el intervalo de ± 0.1 metros.

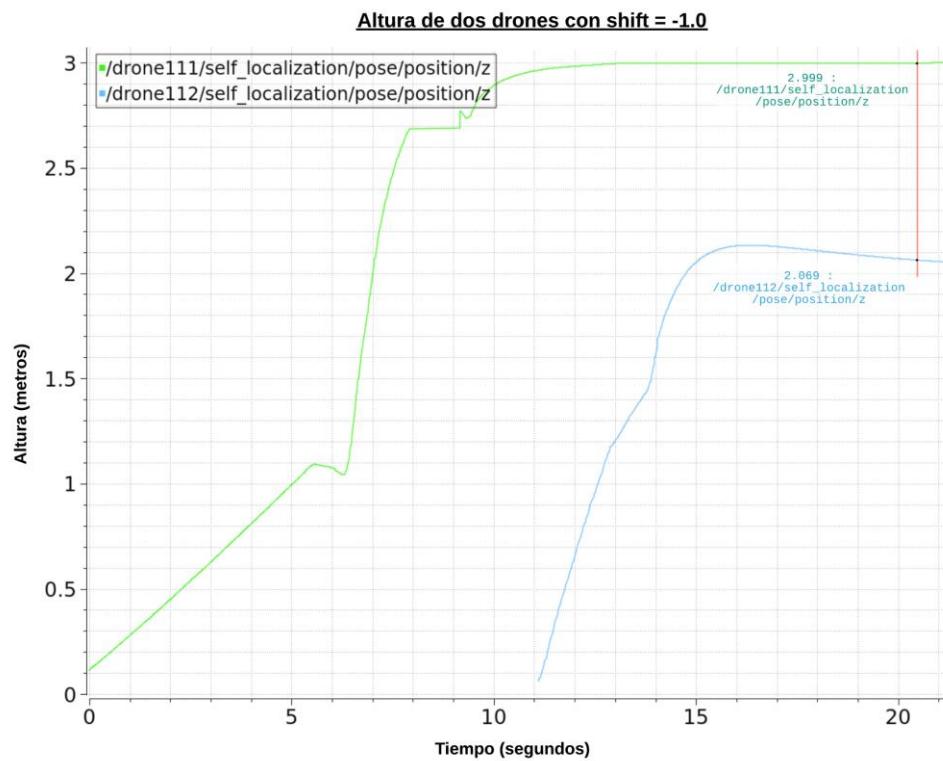


Fig. 4.14. Gráfica de altitud alcanzada por los drones 111 y 112 con shift.

En el caso de la figura 4.14 la altitud alcanzada debía ser de 1.999 metros y se alcanza una altura de 2.069 metros, por lo tanto, la diferencia de altitud es de 0.07 metros, un valor tan pequeño, que se puede dar como válida la prueba.

- Prueba de velocidad. En esta prueba se observa si el dron que ejecuta el behavior va como máximo a la velocidad que se le indica el parámetro `maximum_speed`. En esta prueba se ha ejecutado el behavior empleando la velocidad, 0.3 m/s (figura 4.15). Se ha empleado el topic `self_localization/speed`.

Velocidad = 0.3 m/s

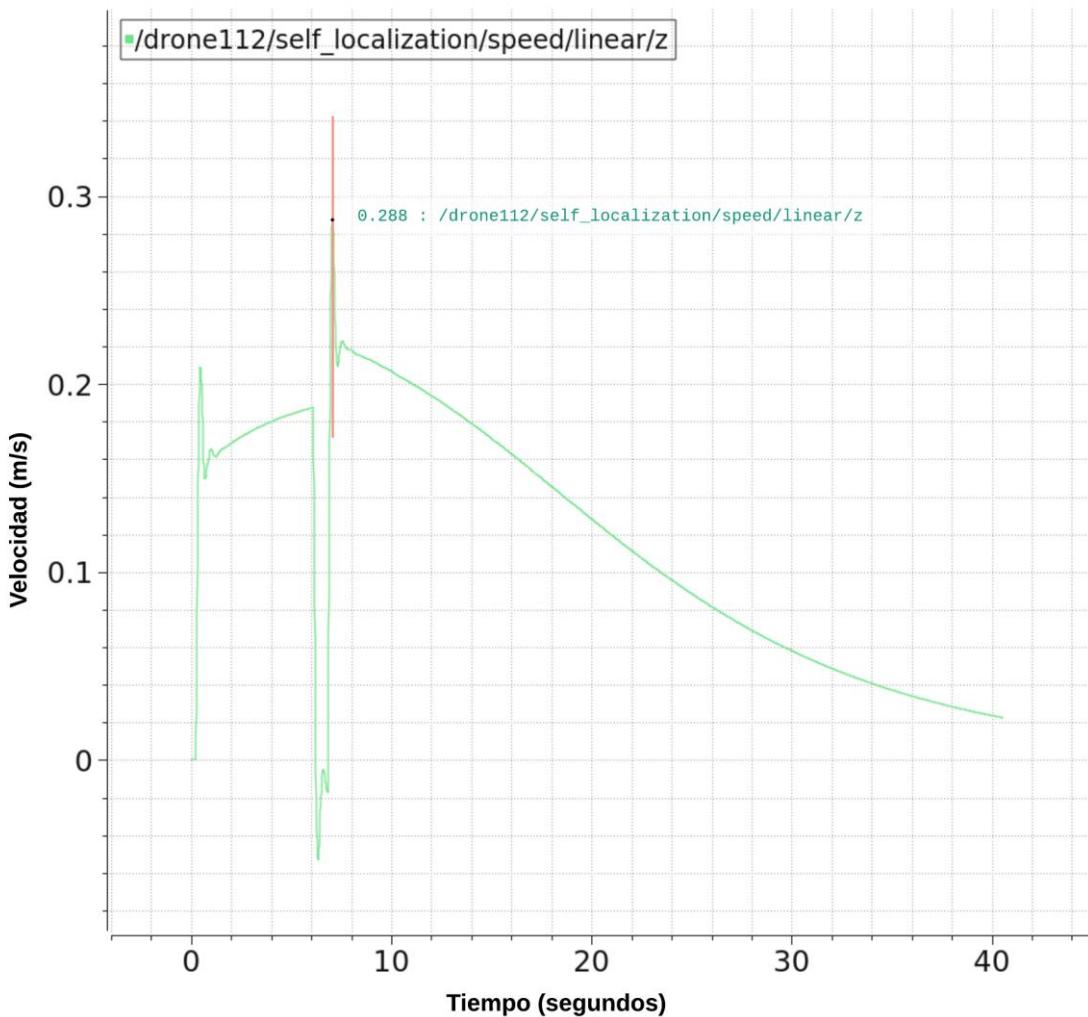


Fig. 4.15. Gráfica de velocidad en el eje “z” del topic `self_localization/speed` con velocidad = 0.3 m/s.

En el caso de la figura 4.15 se puede observar que nunca se rebasa la velocidad máxima de 0.3 m/s en el eje “z” (velocidad a la que se regula la altitud).

El haber realizado las pruebas anteriores sobre este behavior no garantiza que realice la acción que debe hacer. Por ello, para verificar el funcionamiento del comportamiento se le ha realizado una prueba junto con otros behaviors de Aerostack (figuras 4.16 y 4.17).

```
def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 0, 3]], yaw='path_facing')

    time.sleep(20.0)

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 0, 1]], yaw='path_facing')

    print('Mission completed.')
```

Fig. 4.16. Fichero mission1.py, ejecutado por el drone con identificador drone111.

```
def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Reach target altitude...")
    mxc.executeTask('REACH_TARGET_ALTITUDE_WITH_PID_CONTROL', target_name = "drone111", shift = 0.0)

    print('Mission completed.')
```

Fig. 4.17. Fichero mission2.py, ejecutado por el drone con identificador drone112.

El funcionamiento de este comportamiento se puede considerar correcto si el dron que active el behavior alcanza la altitud del dron deseado (o la altitud con una variación). También hay que tener en cuenta que el dron que active el comportamiento, una vez alcance la altura deseada, debe permanecer a esa misma altura.

En el siguiente enlace se puede observar el resultado de la ejecución de la prueba: <https://youtu.be/PP5ZaIhtDuo>

4.1.4 Pruebas de ejecución individual: Take A Look At Target With PID Control

A continuación, se muestra una tabla mostrando las pruebas de ejecución realizadas para este behavior.

TABLA VIII

PRUEBAS DE EJECUCIÓN REALIZADAS PARA TAKE A LOOK AT TARGET WITH PID CONTROL

Prueba	Resultado esperado
Se llama al behavior cuando el estado del dron es distinto a FLYING o HOVERING	El behavior no se activa
Se llama al behavior cuando el estado del dron es FLYING o HOVERING	El behavior se activa
El parámetro target_name se omite	El behavior no se activa
El parámetro target_name es incorrecto	El behavior no funciona correctamente
Ejecución de la función stopTask sobre este behavior	El behavior se desactiva

En las siguientes líneas, se muestra una prueba de ejecución para cuya verificación se empleará una gráfica. En este caso el dron con identificador “drone112” ejecutará Take A Look At Target With PID Control:

- Prueba de orientación. En esta prueba se observará si la orientación del dron en el eje de giñada (yaw), se orienta hacia el dron que se quiere mirar. Para este fin se empleará el topic self_localization/pose de dos drones y también se hará uso de la herramienta rviz.

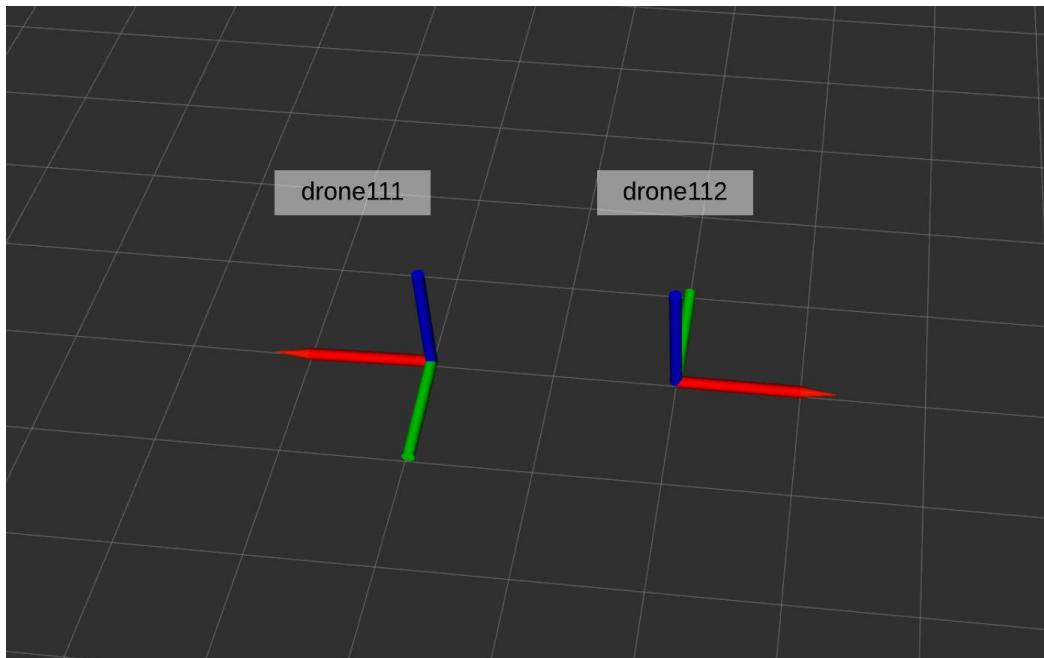


Fig. 4.18. Visualización en rviz de los ejes de cada dron antes de ejecutar el behavior.

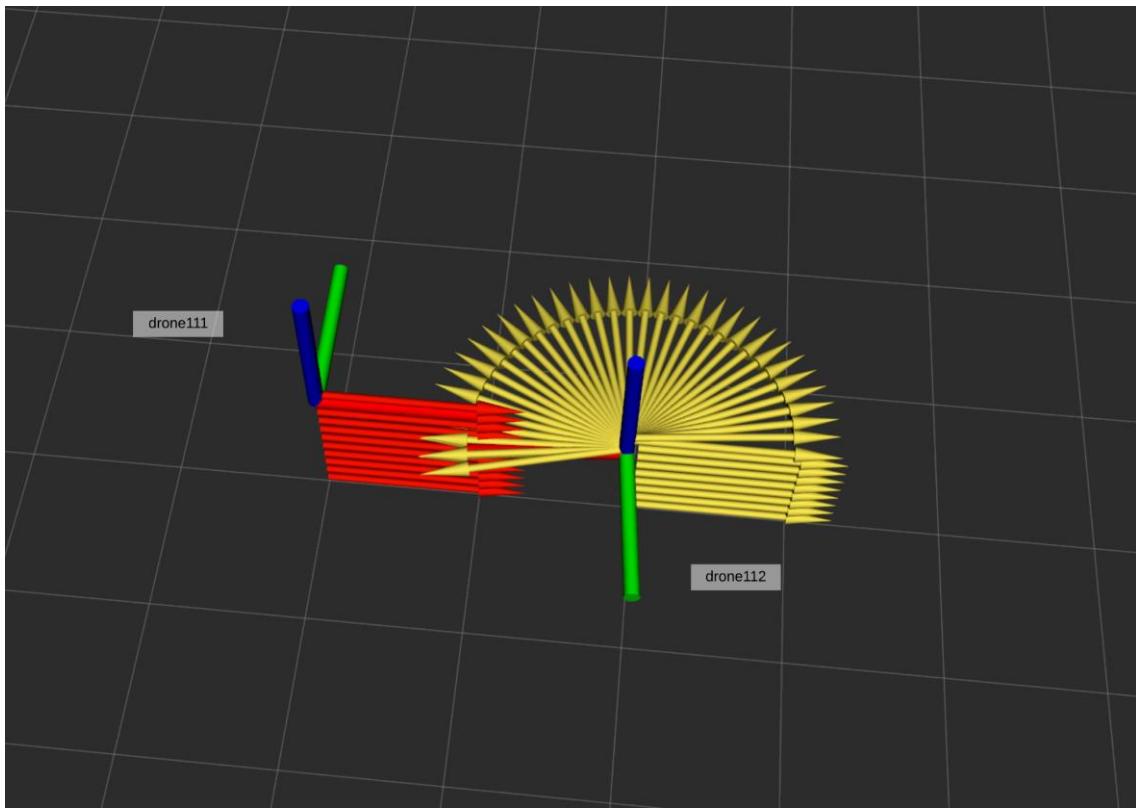


Fig. 4.19. Visualización en rviz del último instante en el que se ejecuta el behavior.

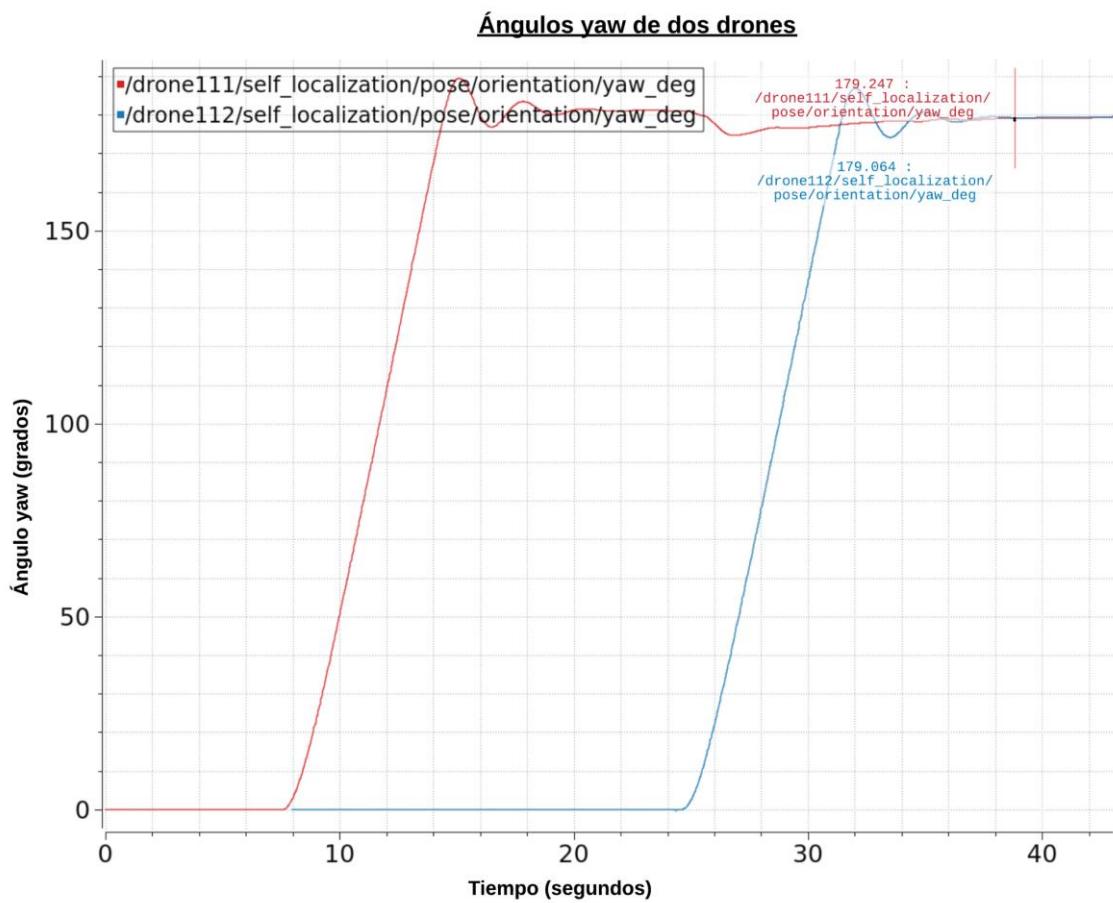


Fig. 4.20. Gráfica de los ángulos yaw de los drones 111 y 112.

Tal y como se puede observar en la gráfica de la figura 4.20, a partir del segundo 25, el dron que ejecuta el behavior comienza a orientarse hacia el dron objetivo hasta que se estabiliza a unos 179° sobre el segundo 34. Existe una pequeña diferencia, pero, tal y como se puede observar en la figura 4.19, que muestra el camino dejado por el eje “x” de los drones durante la ejecución del behavior y en la figura 4.21, en la que se puede ver la imagen de la cámara frontal del dron al final de la ejecución del comportamiento, eso no supone ningún problema pues se visualiza correctamente el robot objetivo.

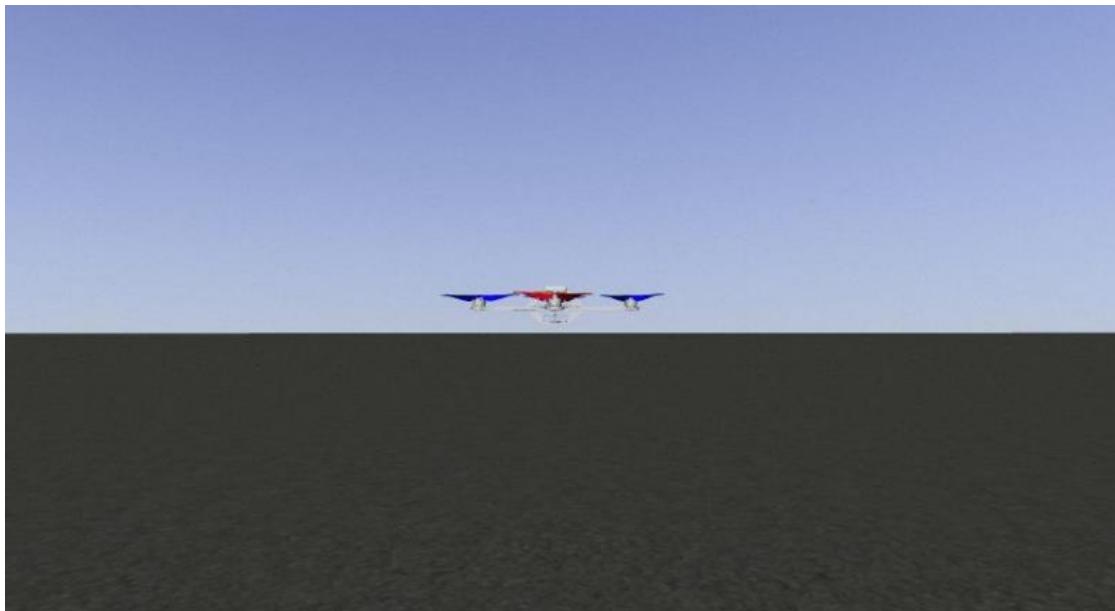


Fig. 4.21. Imagen correspondiente a lo que observa la cámara frontal del dron que ejecuta el comportamiento al finalizar la ejecución del mismo.

Hasta el momento, se han realizado comprobaciones sobre ciertos módulos del behavior, pero el éxito de estas pruebas no deja muy claro si el comportamiento realiza la acción que debe hacer. Es por este motivo que, para verificar que todo funciona correctamente, se ha realizado una prueba de dicho comportamiento junto con behaviors preexistentes en Aerostack tal y como se puede observar en los ficheros de misión de los drones (figuras 4.22 y 4.23).

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 0, 1]], yaw='path_facing')

    time.sleep(20.0)

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 6, 2]], yaw='path_facing')

    print('Mission completed.')

```

Fig. 4.22. Fichero mission1.py, ejecutado por el dron con identificador drone111.

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[4, 0, 1]], yaw='path_facing')

    print("Taking a look at target...")
    mxc.executeTask('TAKE_A_LOOK_AT_TARGET_WITH_PID_CONTROL', target_name = "drone111")

    print('Mission completed.')

```

Fig. 4.23. Fichero mission2.py, ejecutado por el dron con identificador drone112.

El funcionamiento de este comportamiento se puede considerar válido si el dron que activa el behavior se orienta frontalmente al dron deseado y finaliza su ejecución una vez está orientado.

En el siguiente enlace se puede observar el resultado de la ejecución de la prueba: <https://youtu.be/yEBchL54JGg>

4.1.5 Pruebas de ejecución individual: Get Close To Target With PID Control

A continuación, se muestra una tabla mostrando las pruebas de ejecución realizadas sobre este behavior:

TABLA IX

PRUEBAS DE EJECUCIÓN REALIZADAS PARA GET CLOSE TO TARGET WITH PID CONTROL

Prueba	Resultado esperado
Se llama al behavior cuando el estado del dron es distinto a FLYING o HOVERING	El behavior no se activa
Se llama al behavior cuando el estado del dron es FLYING o HOVERING	El behavior se activa
El parámetro target_name se omite	El behavior no se activa
El parámetro target_name es incorrecto	El behavior no funciona correctamente
El parámetro target_distance se omite	El behavior se activa con una distancia por defecto de 2.0 metros
El parámetro target_distance es incorrecto	El behavior no se activa
El parámetro maximum_speed se omite	El behavior se activa con una velocidad por defecto de 0.3 m/s
El parámetro maximum_speed es incorrecto	El behavior no se activa
El parámetro yaw se omite	El behavior se activa con yaw = constant por defecto
El parámetro yaw es incorrecto	Error YAML
Ejecución de la función stopTask sobre este behavior	El behavior se desactiva

A continuación, se muestran una serie de pruebas de ejecución que, al ser más complejas a la hora de verificarlas, es más fácil probar su resultado midiendo una serie de métricas que ilustren cada caso. Decir que en este caso el dron con identificador “drone112” es el que ejecuta Get Close To Target With PID Control:

- El dron va como máximo a la velocidad que se le indica. En esta prueba se ejecutó el behavior con distintas velocidades para comprobar que la velocidad máxima de persecución que recibía el behavior en el parámetro maximum_speed no era sobrepasada en ningún momento de la ejecución. Con este fin, se han realizado pruebas con dos velocidades distintas, 0.3 m/s (figura 4.24) y 0.9 m/s (figura 4.25) y se ha empleado el topic self_localization/speed, que mide la velocidad real a la que se mueve el robot. Se han medido la velocidad de los ejes “x” e “y” ya que la velocidad en el eje “z” (velocidad a la que se aumenta o reduce la altitud) no contribuye de manera directa en la persecución, sólo sitúa al dron que activa el behavior a la misma altitud del dron perseguido.

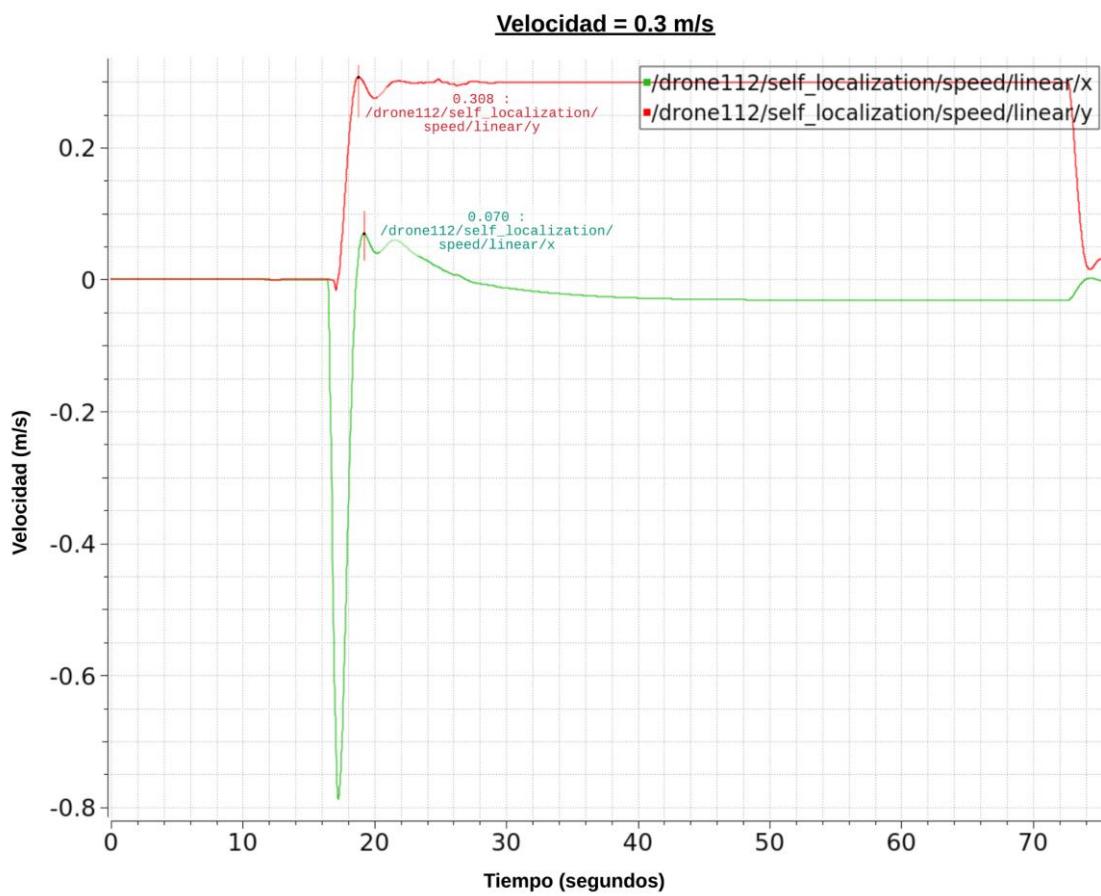


Fig. 4.24. Gráfica de velocidad en los ejes “x” e “y” del topic self_localization/speed con velocidad = 0.3 m/s.

Tal y como se puede observar en la figura 4.24 el dron sobrepasa la velocidad deseada en un breve lapso. Más concretamente hay un desfase de velocidad de +0.008 m/s en el eje “y”, un valor tan irrisorio que no merece la pena tenerlo en cuenta.

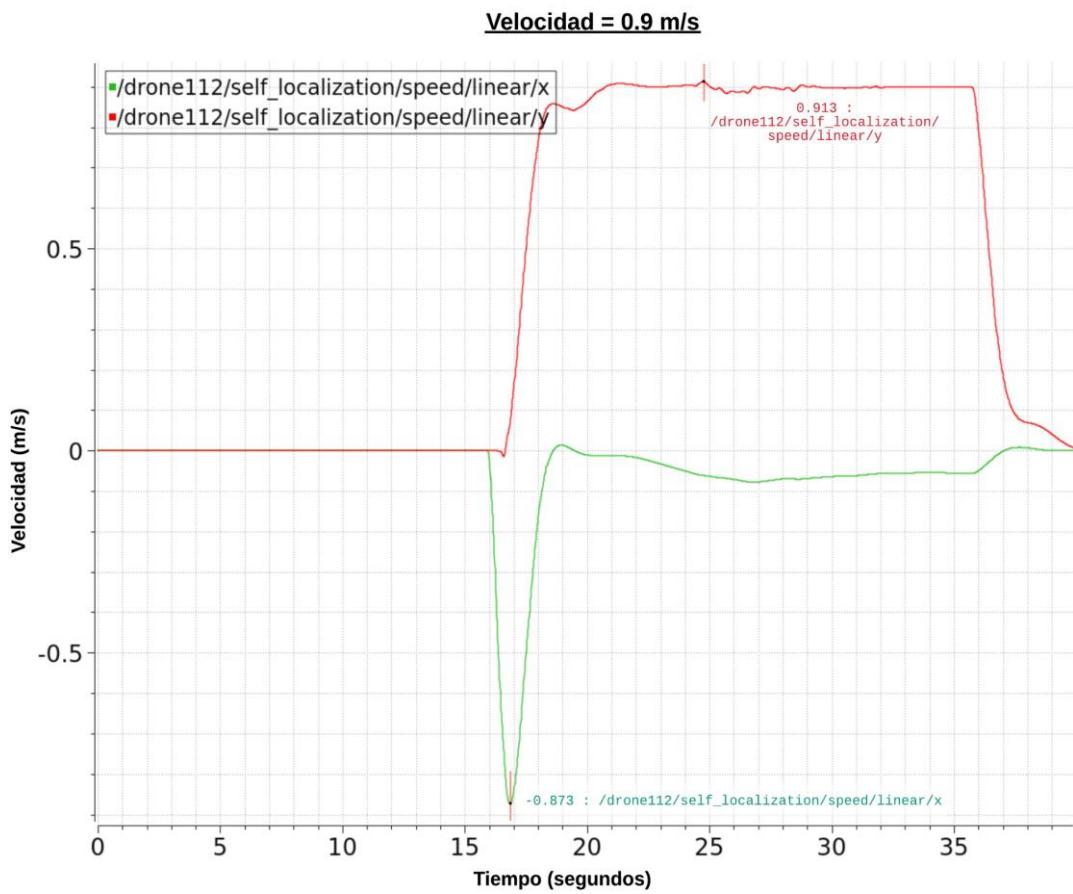


Fig. 4.25. Gráfica de velocidad en los ejes “x” e “y” del topic self_localization/speed con velocidad = 0.9 m/s.

En la figura 4.25 el dron sobrepasa la velocidad deseada, aunque en este caso en +0.013 m/s en el eje “y”. Es un error algo mayor que cuando la velocidad tenía un valor de 0.3 m/s pero aun así es muy bajo.

- Prueba de altitud. Esta prueba consiste en ver si la altitud del dron que ejecuta el behavior es similar a la del dron que se persigue. Esta prueba consiste en verificar que el robot que activa este behavior alcanza una altura muy similar (± 0.1 metros) a la del robot perseguido. Para ello, se empleará el topic self_localization/pose que da la posición real del dron en un momento concreto.

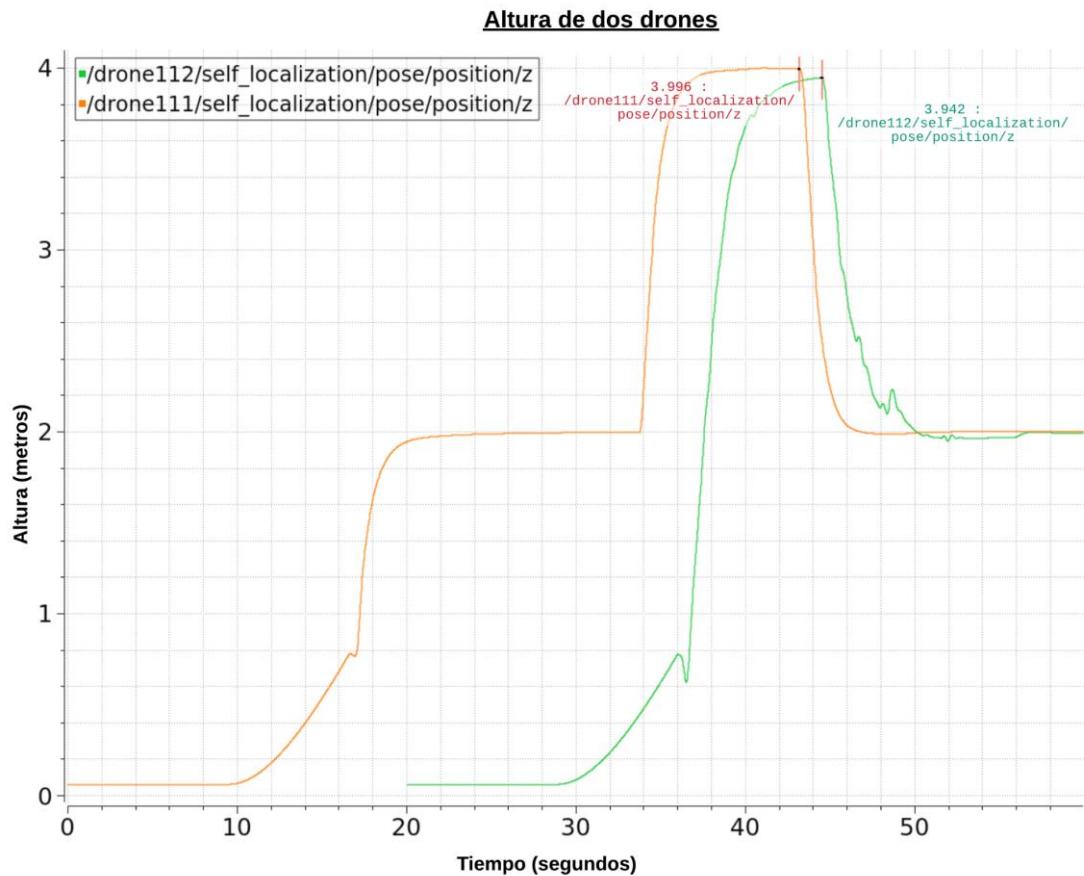


Fig. 4.26. Gráfica de altitud alcanzada por los drones 111 y 112.

Como se puede observar en la figura 4.26, para esta prueba se han medido las altitudes alcanzadas por los drones involucrados en esta prueba teniendo en cuenta el retraso en la activación del behavior por parte del dron con identificador “drone112”. Tal y como se puede ver, esta prueba se supera suficientemente pues las diferencias de altura son del orden de centésimas de metro, siendo estas métricas muy inferiores al objetivo con el que considerar esta prueba como exitosa (± 0.1 metros).

- Prueba de distancia. En esta prueba se observará si el dron que llama al behavior finaliza la ejecutar dicho comportamiento cuando se encuentra a determinada distancia del dron perseguido. Esta distancia es la indicada por el parámetro `target_distance`. La distancia mínima que deben mantener es de 2.0 metros.

Distancia entre dos drones

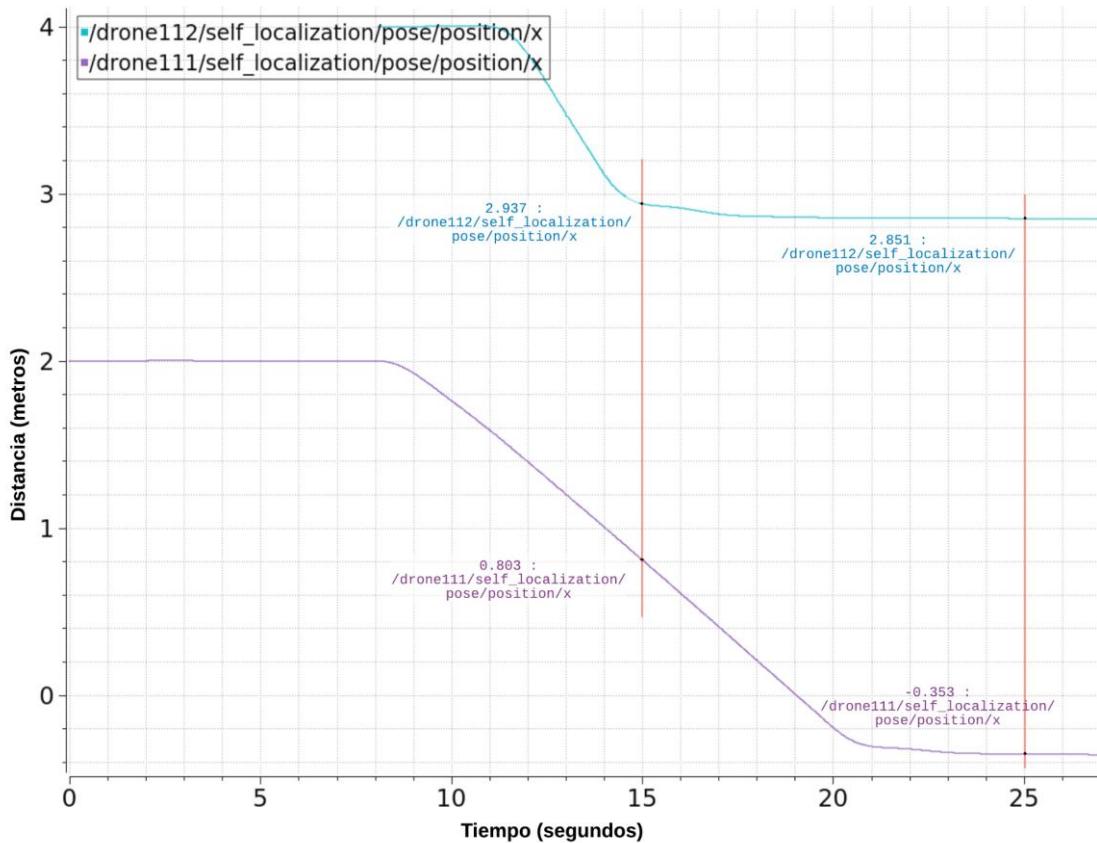


Fig. 4.27. Gráfica que muestra la distancia entre los drones 111 y 112.

Tal y como se puede observar, en la figura 4.27 el dron que ejecuta el behavior (drone112), en torno al segundo 15, alcanza la distancia mínima de 2 metros y, una vez que eso ocurre se finaliza la ejecución del behavior.

El éxito de estas pruebas no aclara si el comportamiento realiza la tarea que debe realizar. Por ello, para verificar que todo funciona correctamente, se ha realizado una prueba de dicho comportamiento junto con behaviors preexistentes en Aerostack (figuras 4.28 y 4.29).

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 3, 2]], yaw='path_facing')

    time.sleep(15.0)

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 6, 2]], yaw='path_facing')

    print('Mission completed.')

```

Fig. 4.28. Fichero mission1.py, ejecutado por el dron con identificador drone111.

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following drone...")
    mxc.executeTask('GET_CLOSE_TO_TARGET_WITH_PID_CONTROL', target_name = "drone111", target_distance = 2.0,
                    maximum_speed = 0.4)

    print('Mission completed.')

```

Fig. 4.29. Fichero mission2.py, ejecutado por el dron con identificador drone112.

El funcionamiento de este comportamiento se puede considerar válido si el dron que activa el behavior persigue al robot objetivo hasta alcanzar una distancia determinada, momento en el que el behavior termina su ejecución.

En el siguiente enlace se puede observar el resultado de la ejecución de la prueba: <https://youtu.be/5Z3-UVB-Cbc>

4.1.6 Pruebas de ejecución individual: Keep Look At Target With PID Control

A continuación, se muestra una tabla mostrando las pruebas de ejecución realizadas para este behavior.

TABLA X

PRUEBAS DE EJECUCIÓN REALIZADAS PARA KEEP LOOK AT TARGET WITH PID CONTROL

Prueba	Resultado esperado
Se llama al behavior cuando el estado del dron es distinto a FLYING o HOVERING	El behavior no se activa
Se llama al behavior cuando el estado del dron es FLYING o HOVERING	El behavior se activa
El parámetro target_name se omite	El behavior no se activa
El parámetro target_name es incorrecto	El behavior no funciona correctamente
Ejecución de la función stopTask sobre este behavior	El behavior se desactiva

A continuación, se muestra una prueba de ejecución para cuya verificación se empleará una gráfica. En este caso el dron con identificador “drone112” ejecutará Keep Look At Target With PID Control:

- Prueba de orientación. En esta prueba se comprobará si la orientación del dron en el eje de guiñada (yaw), se orienta hacia el dron que se quiere observar. Si el dron que se desea mirar modifica su posición se debe actualizar la orientación. Para este fin se empleará el topic self_localization/pose de dos drones y también se hará uso de la herramienta rviz.

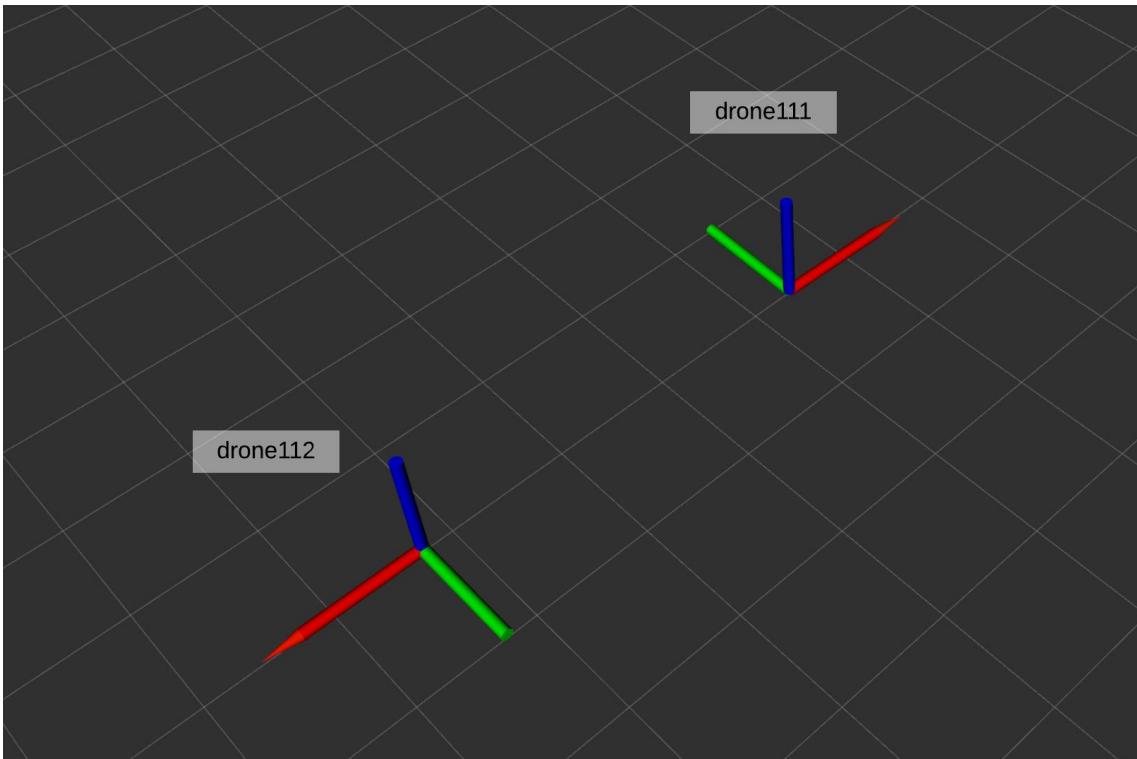


Fig. 4.30. Visualización en rviz de los ejes de cada dron antes de ejecutar el behavior.

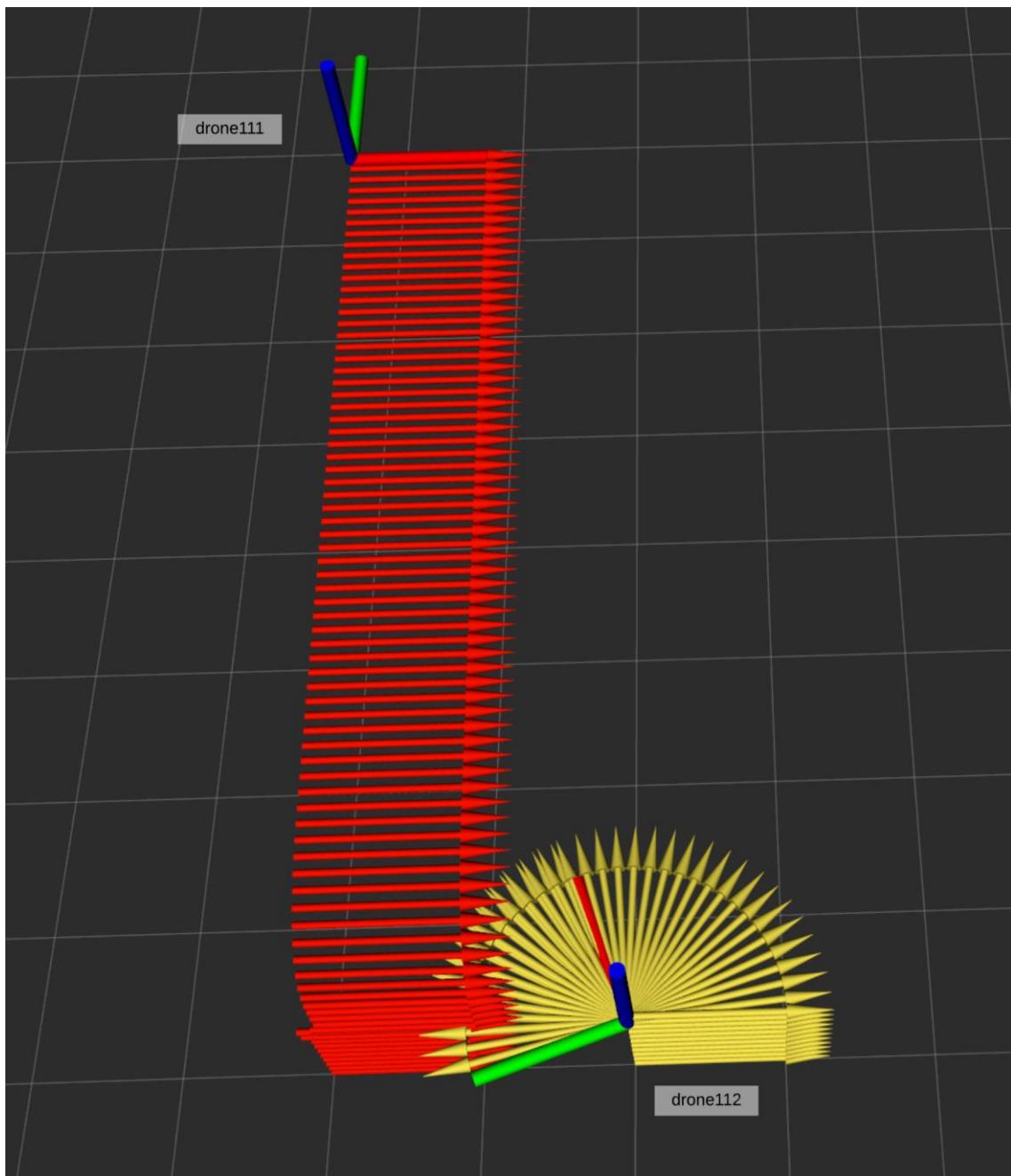


Fig. 4.31. Visualización en rviz de la ejecución del behavior.

En la figura 4.31, se puede observar cómo el dron que activa el behavior sigue observando al dron objetivo (eje “x” orientado al objetivo deseado) a pesar de que este altere su posición.

Ángulos yaw de dos drones

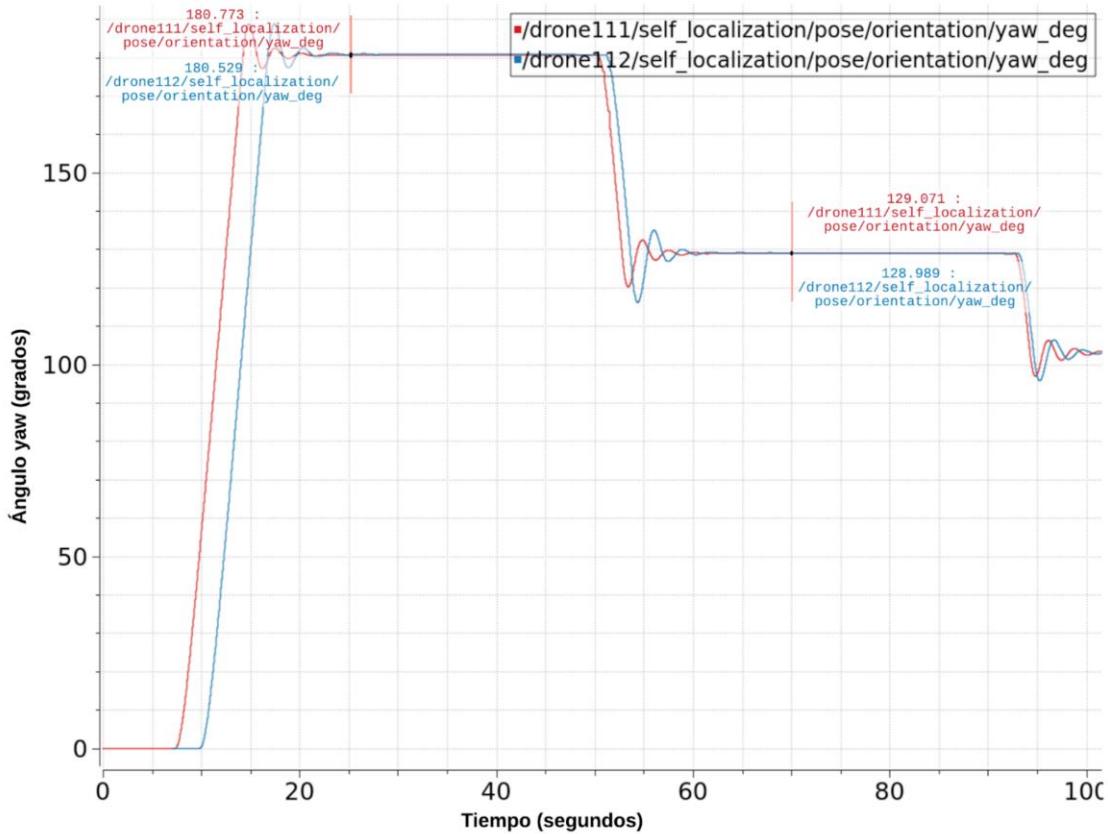


Fig. 4.33. Gráfica de los ángulos yaw de los drones 111 y 112.

Tal y como se puede observar en la gráfica de la figura 4.33, a partir del segundo 10, el dron que ejecuta el behavior comienza a orientarse hacia el dron objetivo hasta que se estabiliza a unos 180° . A partir de los segundos 44 y 70 el dron al que se desea mirar (drone111) cambia de posición y el dron que ejecuta el behavior se orienta hacia él. El error en todos los casos es minúsculo así que se podría dar como pasada esta prueba.

Hasta el momento, se han realizado comprobaciones sobre ciertos módulos del behavior, pero el éxito de estas pruebas no deja muy claro si el comportamiento realiza la acción que debe hacer. Es por este motivo que, para verificar que todo funciona correctamente, se ha realizado una prueba de dicho comportamiento junto con behaviors preexistentes en Aerostack tal y como se puede observar en los ficheros de misión de los drones (figuras 4.34 y 4.35).

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 0, 1]], yaw='path_facing')
    time.sleep(20.0)

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 6, 2]], yaw='path_facing')

    print('Mission completed.')

```

Fig. 4.34. Fichero mission1.py, ejecutado por el dron con identificador drone111.

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[4, 0, 1]], yaw='path_facing')

    print("Taking a look at target...")
    mxc.executeTask('KEEP_LOOK_AT_TARGET_WITH_PID_CONTROL', target_name = "drone111")

    print('Mission completed.')

```

Fig. 4.35. Fichero mission2.py, ejecutado por el dron con identificador drone112.

El funcionamiento de este comportamiento se puede considerar válido si el dron que activa el behavior se orienta frontalmente hacia el dron deseado, aunque este modifique su posición.

En el siguiente enlace se puede observar el resultado de la ejecución de la prueba: https://youtu.be/ahiJ2kL_QzA

4.1.7 Pruebas de ejecución individual: Keep Target Altitude With PID Control

A continuación, se muestra una tabla mostrando las pruebas de ejecución realizadas sobre este behavior.

TABLA XI

PRUEBAS DE EJECUCIÓN REALIZADAS PARA KEEP TARGET ALTITUDE WITH PID CONTROL

Prueba	Resultado esperado
Se llama al behavior cuando el estado del dron es distinto a FLYING o HOVERING	El behavior no se activa
Se llama al behavior cuando el estado del dron es FLYING o HOVERING	El behavior se activa
El parámetro target_name se omite	El behavior no se activa
El parámetro target_name es incorrecto	El behavior no funciona correctamente
El parámetro shift se omite	El behavior se activa con una variación por defecto de 0.0 metros
El parámetro shift es incorrecto	El behavior no se activa
El parámetro maximum_speed se omite	El behavior se activa con controlMode= GROUND_SPEED
El parámetro maximum_speed es incorrecto	El behavior no se activa
Ejecución de la función stopTask sobre este behavior	El behavior se desactiva

A continuación, se muestran una serie de pruebas de ejecución complejas de verificar, para las que se ha decidido emplear gráficas para su comprobación. En este caso el dron con identificador “drone112” ejecutará Keep Target Altitude With PID Control:

- Prueba de altitud. Esta prueba consiste en ver si la altitud del dron que ejecuta el behavior es similar (± 0.1 metros) a la del dron al que se quiere igualar la altitud. Se empleará el topic self_localization/position y se expondrán dos casos, uno con shift = 0.0 y otro con shift = 2.0.

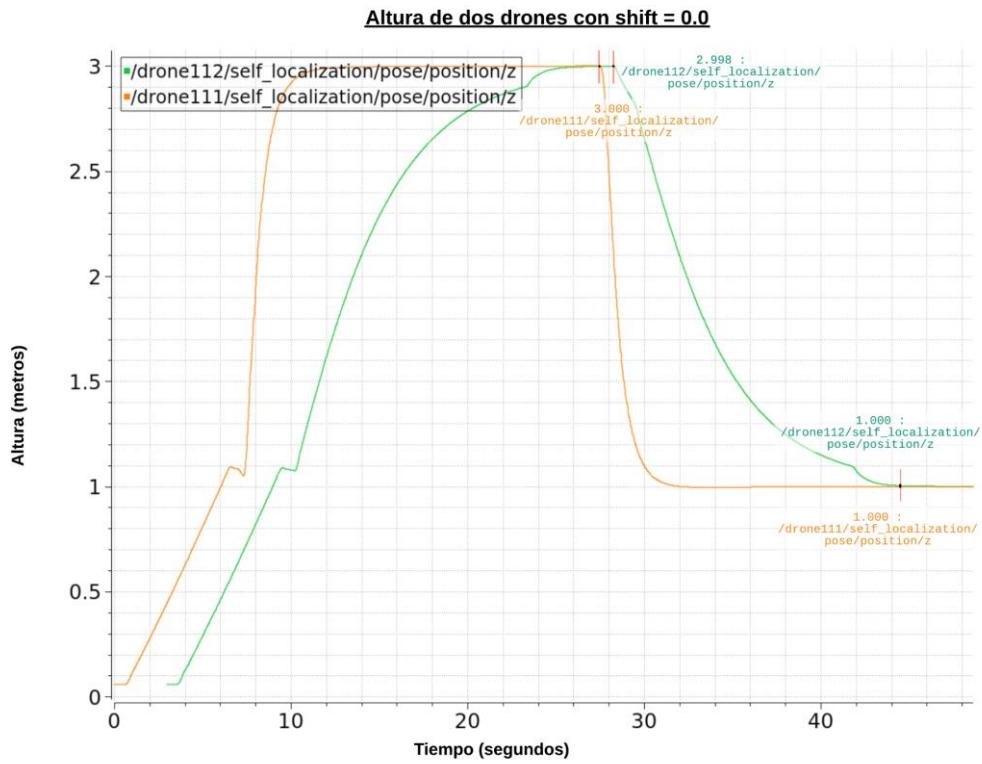


Fig. 4.36. Gráfica de altitud alcanzada por los drones 111 y 112 sin shift.

En la figura 4.36, se han medido las altitudes alcanzadas teniendo en cuenta el retraso en la activación del behavior por parte del dron con identificador “drone112”. Tal y como se puede ver, la variación es de 0.0 metros.

Altura de dos drones con shift = 2.0

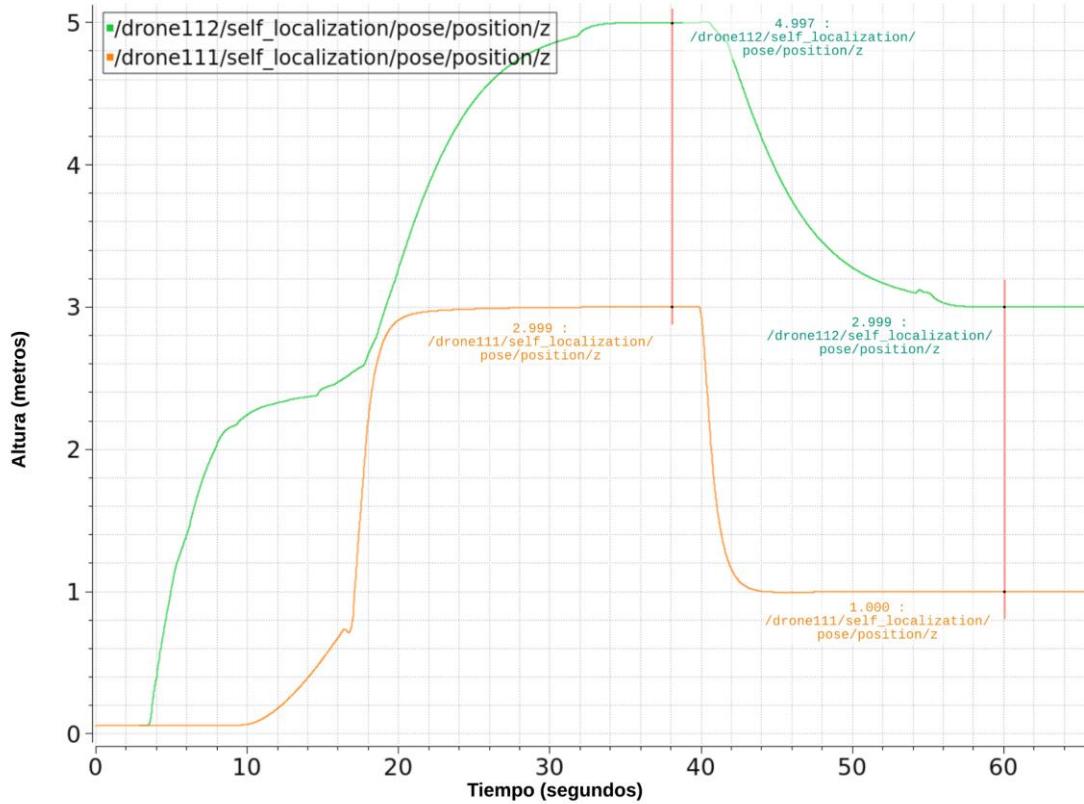


Fig. 4.37. Gráfica de altitud alcanzada por los drones 111 y 112 con shift.

En el caso de la figura 4.37 la altitud alcanzada debía ser de 4.999 metros y se alcanza una altura de 4.997 metros por lo tanto se puede dar como válida esta prueba.

- Prueba de velocidad. En esta prueba se observará si el dron que ejecuta el behavior va como máximo a la velocidad que se le indica el parámetro `maximum_speed`. En esta prueba se ha ejecutado el behavior empleando la velocidad, 0.5 m/s (figura 4.38). Se ha empleado el topic `self_localization/speed`.

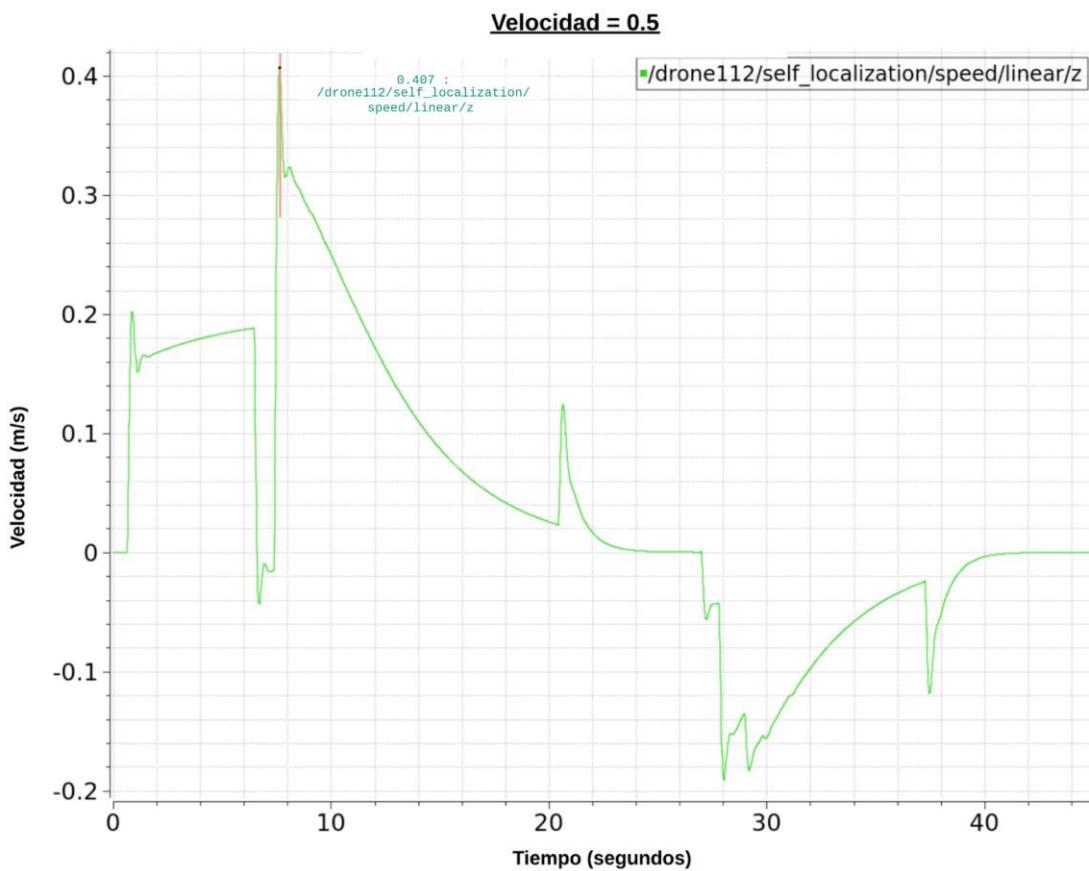


Fig. 4.38. Gráfica de velocidad en el eje “z” del topic self_localization/speed con velocidad = 0.5 m/s.

En el caso de la figura 4.38 se puede observar que nunca se sobrepasa la velocidad máxima de 0.5 m/s en el eje “z” (velocidad a la que se regula la altitud) y, por lo tanto, se supera la prueba en este caso.

Al igual que en el behavior anterior, las pruebas exitosas sobre distintos aspectos del comportamiento no garantizan que el behavior realice la acción que debe hacer. Es por este motivo que, para verificar el correcto funcionamiento del comportamiento se ha realizado una prueba en la que se emplean behaviors preexistentes en Aerostack junto con Keep Target Altitude tal y como se ve en los ficheros de misión (figuras 4.39 y 4.40).

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 0, 3]], yaw='path_facing')

    time.sleep(20.0)

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[2, 0, 5]], yaw='path_facing')

    print('Mission completed.')

```

Fig. 4.39. Fichero mission1.py, ejecutado por el dron con identificador drone111.

```

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
    mxc.startTask('INFORM_POSITION_TO_ROBOTS')

    print("Taking off...")
    mxc.executeTask('TAKE_OFF')

    print("Following path...")
    mxc.executeTask('FOLLOW_PATH', path=[[4, 0, 1]], yaw='path_facing')

    print("Keep target altitude...")
    mxc.executeTask('KEEP_TARGET_ALTITUDE_WITH_PID_CONTROL', target_name = "drone111", shift = 0.0)

    print('Mission completed.')

```

Fig. 4.40. Fichero mission2.py, ejecutado por el dron con identificador drone112.

El funcionamiento de este comportamiento se puede considerar correcto si el dron que activa el behavior mantiene la altura deseada respecto al otro dron a pesar de alcanzar dicha altitud y de que este modifique su posición.

En el siguiente enlace se puede observar el resultado de la ejecución de la prueba: <https://youtu.be/7ONGlJ5vFKg>

4.2 Prueba de ejecución integrada en misiones aéreas

Con el fin de mostrar la utilidad de los behaviors diseñados a lo largo de este trabajo, se ha desarrollado un escenario especial en el que un dron (drone112) se encarga de interceptar a un dron intruso (drone111) si este último sobrepasa un perímetro cuadrangular. Ambos drones se mueven de manera aleatoria al inicio de la misión, por lo que nunca se sabe en qué momento se sobrepondrá el perímetro. El dron encargado de interceptar patrullará de manera aleatoria el interior del perímetro hasta que detecte que el robot invasor lo ha sobrepondido. Una vez el dron sobrepondrá el perímetro es perseguido hasta que le llega un mensaje de stop por parte del robot que patrullaba el interior del cuadrado. Una vez interceptado el robot invasor, el dron interceptor aterrizará en el mismo sitio donde despegó.

Cabe mencionar que, el dron que sobrepondrá el perímetro sabe que lo ha invadido porque se conoce de antemano los cuatro puntos que conforman el cuadrado del área a invadir. Un cuadrado, al estar formado por dos triángulos, se puede saber fácilmente si el dron está dentro o fuera de ese perímetro ya que si la distancia de su posición a cada vértice es menor que las longitudes de los lados de los triángulos entonces el punto en el que se localiza el robot invasor está dentro del perímetro.

Para conseguir ejecutar con éxito esta misión, se ha hecho uso de cinco ficheros principales:

- Un fichero launcher_gazebo.sh, en el que se especifican el número de robots presentes en la misión y su posición en el mundo. También se encarga de cargar y ejecutar el mundo empleando el simulador de robots Gazebo. La implementación de este fichero se puede encontrar en el [Anexo 8](#).
- Dos ficheros main_launcher.sh, que se encargan de desplegar los paquetes ROS de Aerostack necesarios para la correcta ejecución de la misión. Estos dos ficheros pueden ser observados en el [Anexo 9](#) y [Anexo 10](#), los cuales corresponden a los robots con identificador “drone111” y “drone112” respectivamente.
- Dos ficheros mission.py, en los que se establecen los comportamientos que debe ejecutar cada dron. Estos ficheros se pueden encontrar en el [Anexo 11](#) y [Anexo 12](#), los cuales corresponden a los robots con identificador “drone111” y “drone112”.

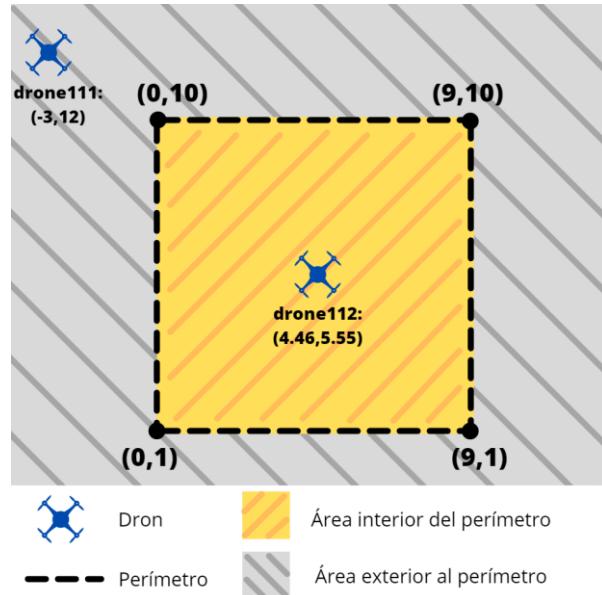


Fig. 4.41. Plano 2D del mundo en el que se lleva a cabo la prueba.

En el siguiente enlace se muestra un video resultado de la ejecución de esta prueba: <https://youtu.be/latH07DfMv8>

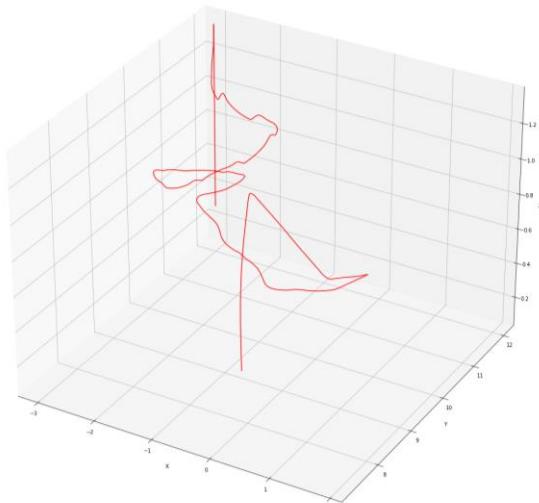


Fig. 4.42. Trayectoria seguida por el dron drone111 durante la ejecución de la prueba.

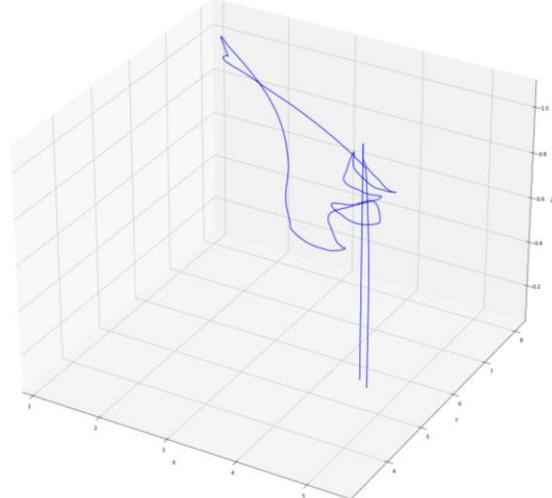


Fig. 4.43. Trayectoria seguida por el dron drone112 durante la ejecución de la prueba.

4.3 Pruebas de rendimiento

Durante la realización de las pruebas de ejecución, aparte de tomar medidas para comprobar el éxito de las pruebas también se han tomado medidas para medir el rendimiento del PC sobre el que han sido ejecutadas las pruebas. El objetivo de estas pruebas es dar una visión general de los recursos consumidos en la ejecución de cada behavior.

TABLA XII
ESPECIFICACIONES DEL PC CON EL QUE SE HAN REALIZADO LAS PRUEBAS

Especificaciones PC	
Modelo	P65 Creator 9SE (16Q4.3)
Fabricante	Micro-Star International Co., Ltd.
CPU	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz x 12
GPU	GeForce RTX 2060/PCIe/SSE2
Resolución de pantalla	3840x2160 pixeles
Tamaño de disco	299.9 GiB SSD
Memoria RAM usable	31.2 GiB
Sistema Operativo	Ubuntu 18.04.5 LTS
Tipo de Sistema Operativo	64-bit

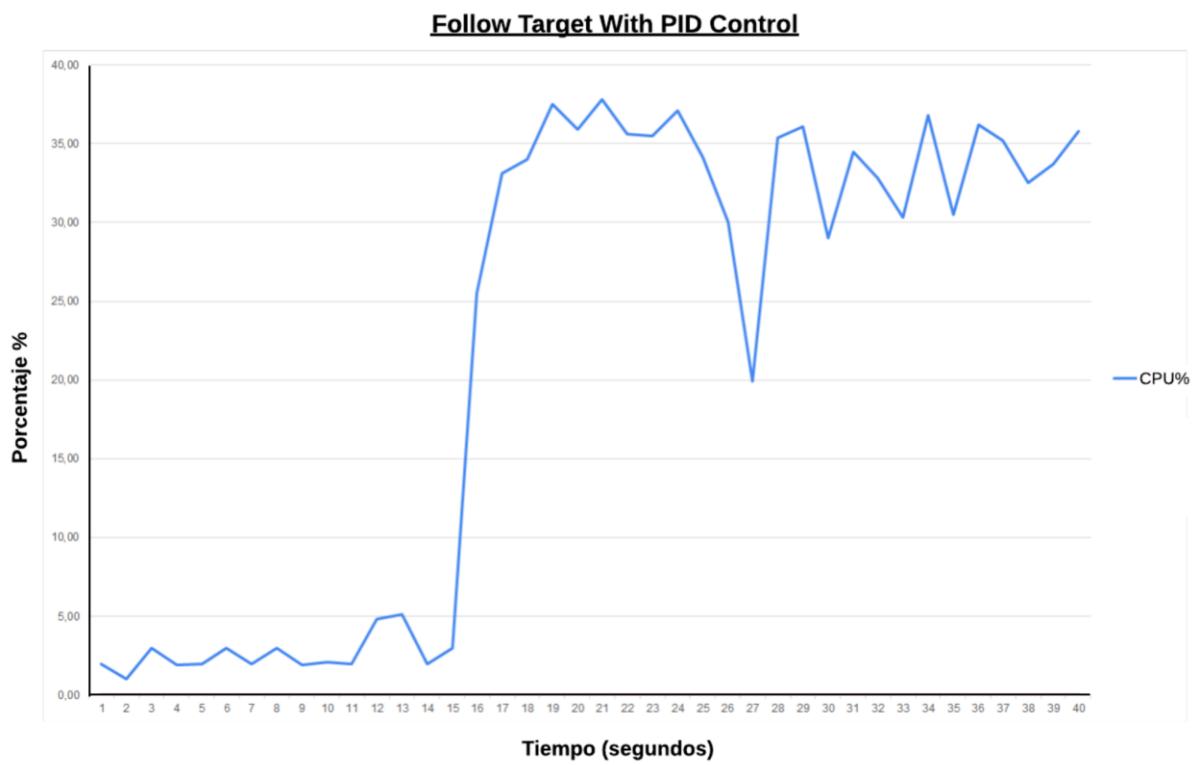


Fig. 4.44. Porcentajes de CPU y memoria consumidos durante la ejecución de Follow Target With PID Control.

Move Away From Robot With PID Control

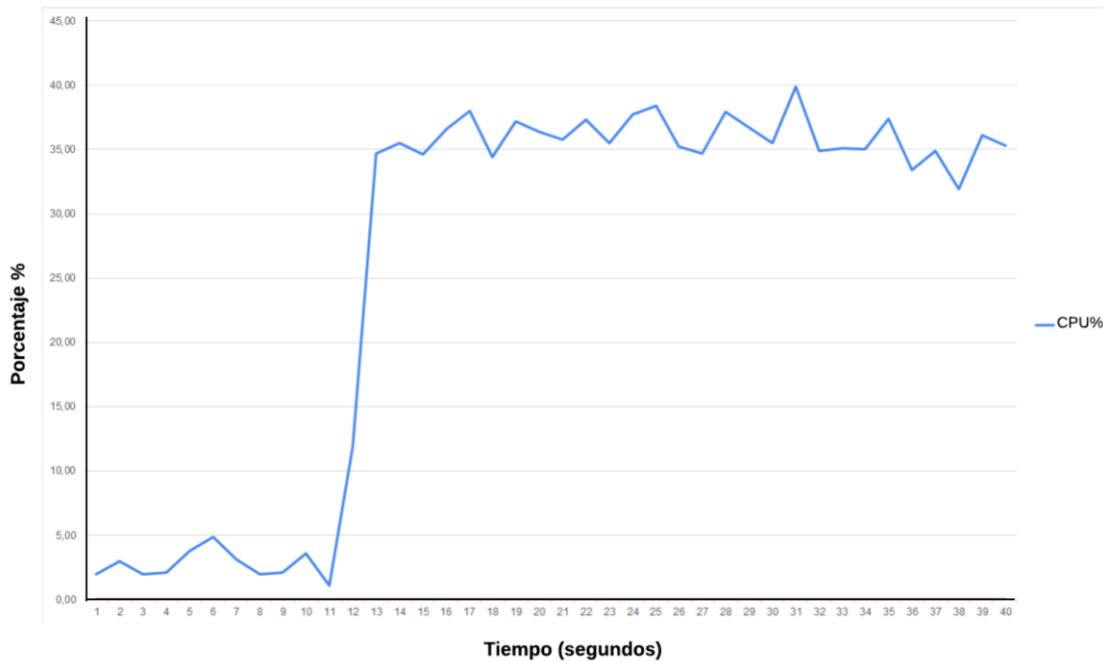


Fig. 4.45. Porcentajes de CPU y memoria consumidos durante la ejecución de Move Away From Robot With PID Control.

Reach Target Altitude With PID Control

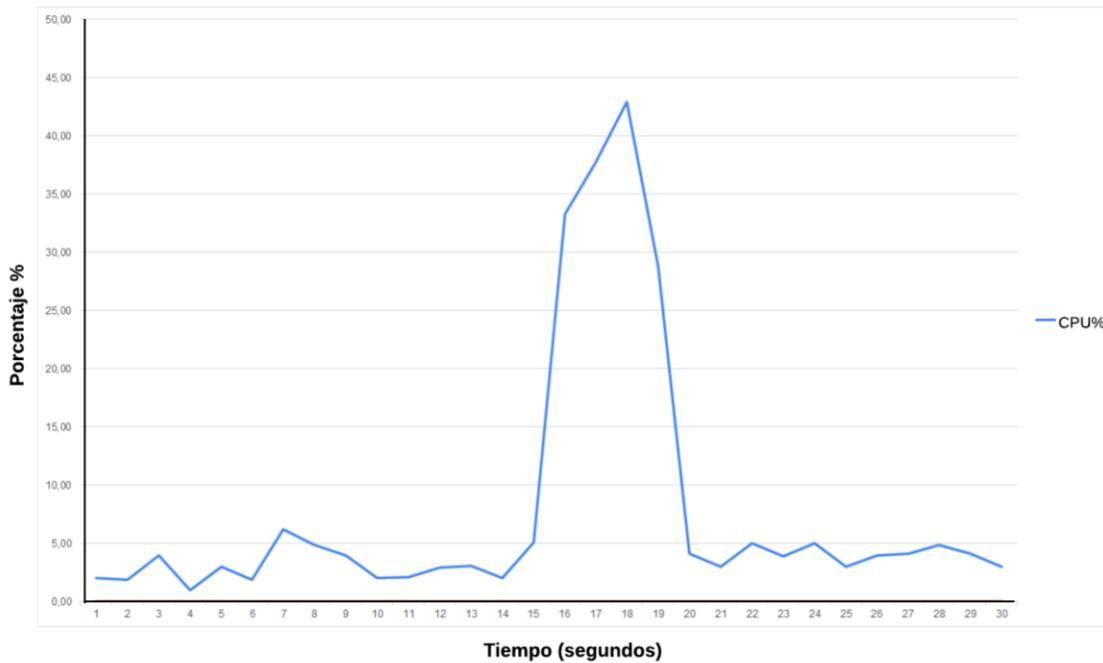


Fig. 4.46. Porcentajes de CPU y memoria consumidos durante la ejecución de Reach Target Altitude With PID Control.

Take A Look At Target With PID Control

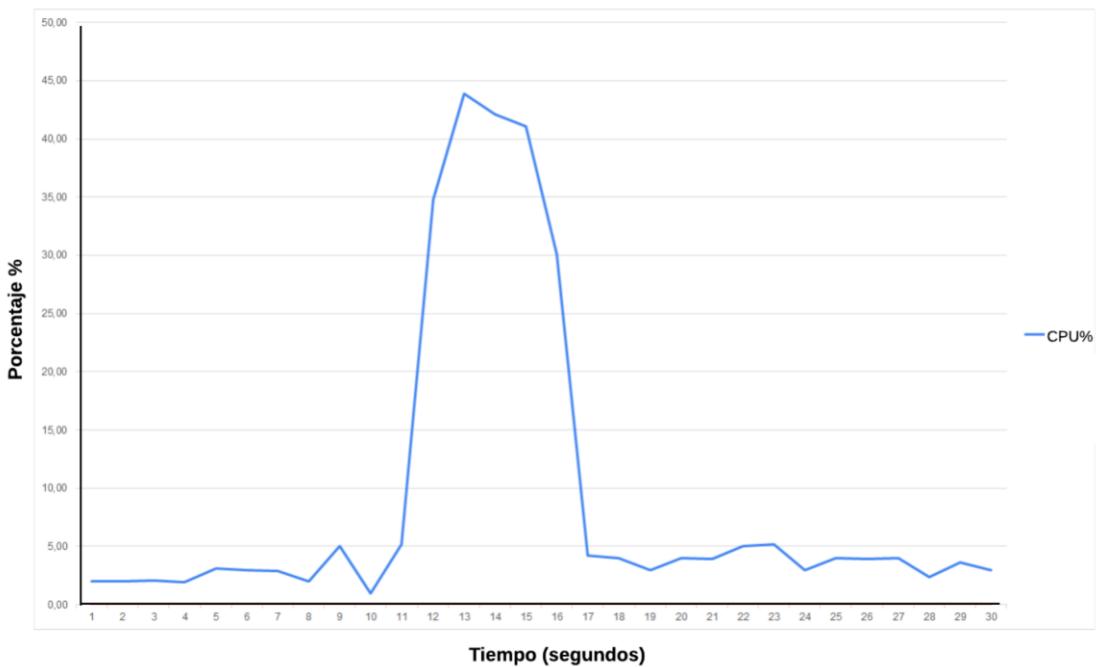


Fig. 4.47. Porcentajes de CPU y memoria consumidos durante la ejecución de Take A Look At Target With PID Control.

Get Close To Target With PID Control

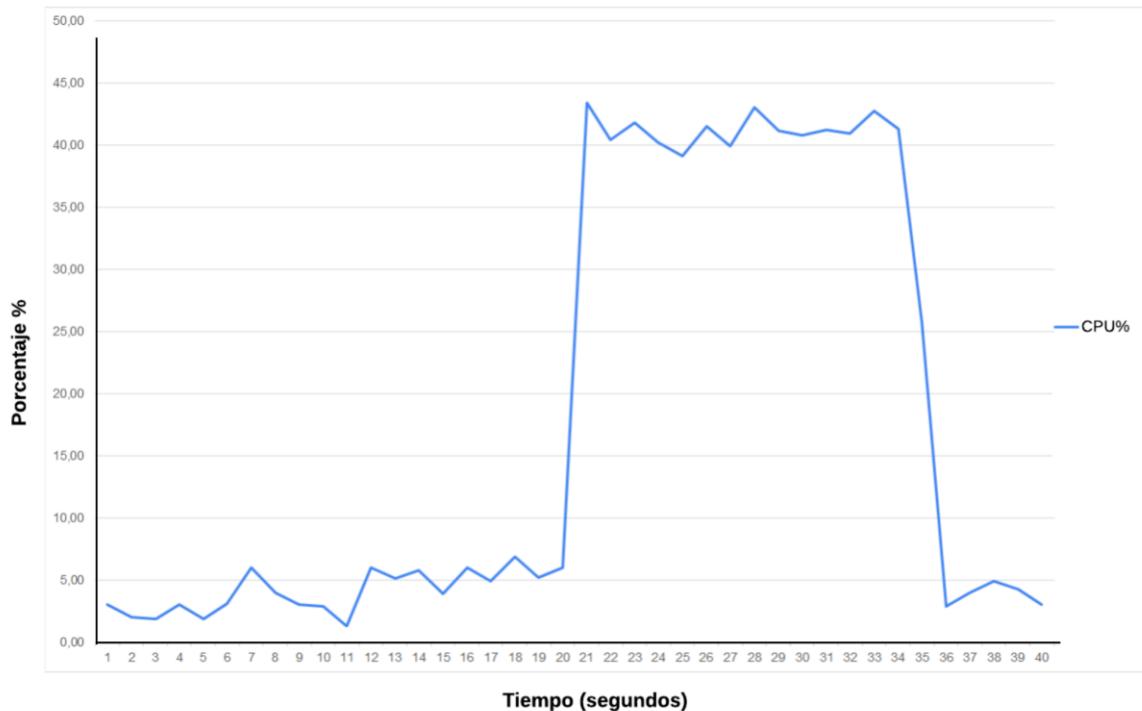


Fig. 4.48. Porcentajes de CPU y memoria consumidos durante la ejecución de Get Close To Target With PID Control.

Keep Look At Target With PID Control

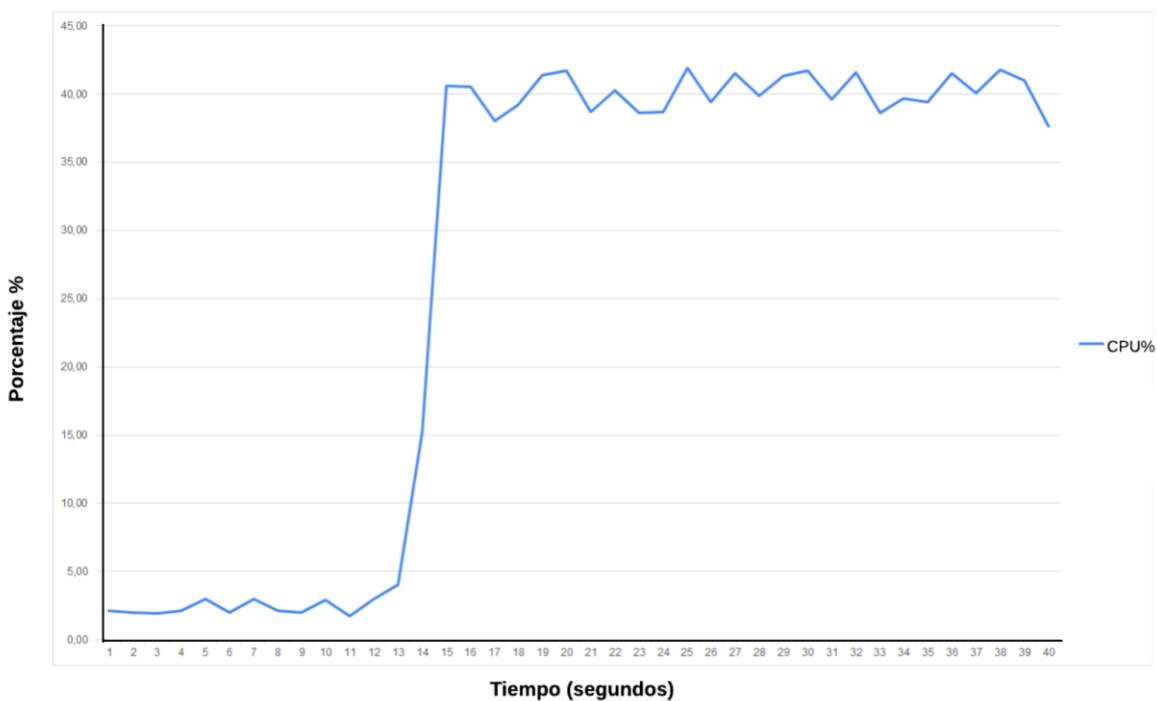


Fig. 4.49. Porcentajes de CPU y memoria consumidos durante la ejecución de Keep Look At Target With PID Control.

Keep Target Altitude With PID Control

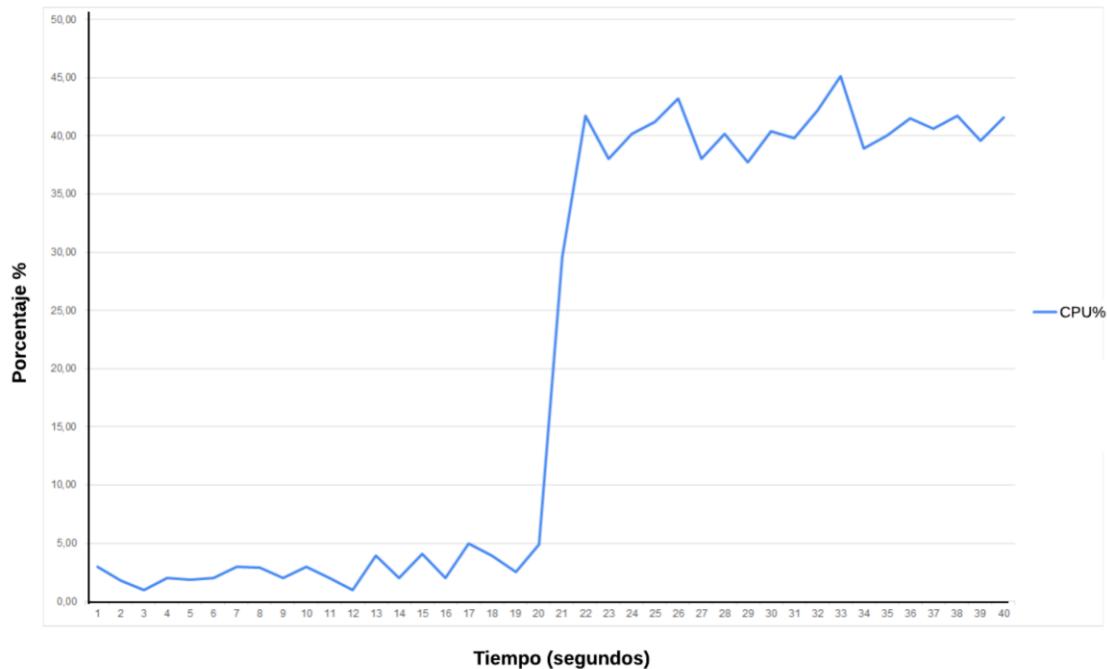


Fig. 4.50. Porcentajes de CPU y memoria consumidos durante la ejecución de Keep Target Altitude With PID Control.

Los gráficos que van desde las figuras 4.44 hasta la figura 4.50, muestran el porcentaje de CPU empleado durante la ejecución de cada uno de los servicios/behaviors.

Tal y como se observa, el porcentaje de CPU anterior a la ejecución de los behaviors (o cuando se desactivan) tiene un valor muy bajo que ronda generalmente el 2.5% de media. Esto es así hasta que los behaviors se activan, y cuando eso sucede se alcanzan valores entre el 30% y el 45% de consumo de CPU. Estos porcentajes se pueden considerar normales en comparación con otros behaviors preexistentes en la plataforma Aerostack como Take Off With PID Control o Land With PID Control que también alcanzan valores similares.

4.4 Dimensión del software desarrollado

En este apartado se exponen ciertas métricas acerca del código implementado en la realización de este proyecto con el fin de dar una visión global del tamaño del software implementado.

TABLA XIII
DIMENSIONES DEL SOFTWARE IMPLEMENTADO I

Fichero	Lenguaje	Líneas de código funcional	Líneas comentadas	Líneas en blanco	Número de clases	Líneas totales
behavior_follow_target_with_pid_control.h	C++	80	44	10	0	134
behavior_follow_target_with_pid_control.cpp	C++	171	46	50	1	267
behavior_move_away_from_robot_with_pid_control.h	C++	81	44	11	0	136
behavior_move_away_from_robot_with_pid_control.cpp	C++	186	46	47	1	276
behavior_reach_target_altitude_with_pid_control.h	C++	73	44	11	0	128
behavior_reach_target_altitude_with_pid_control.cpp	C++	167	40	43	1	250
behavior_take_a_look_at_target_with_pid_control.h	C++	70	44	12	0	126
behavior_take_a_look_at_target_with_pid_control.cpp	C++	136	43	44	1	223
behavior_get_close_to_target_with_pid_control.h	C++	79	44	10	0	133
behavior_get_close_to_target_with_pid_control.cpp	C++	170	46	48	1	264
behavior_keep_look_at_target_with_pid_control.h	C++	70	44	11	0	125
behavior_keep_look_at_target_with_pid_control.cpp	C++	133	43	45	1	221

TABLA XIV
DIMENSIONES DEL SOFTWARE IMPLEMENTADO II

Fichero	Lenguaje	Líneas de código funcional	Líneas comentadas	Líneas en blanco	Número de clases	Líneas totales
behavior_keep_target_altitude_with_pid_control.h	C++	73	44	11	0	128
behavior_keep_target_altitude_with_pid_control.cpp	C++	161	40	45	1	246
quadrotor_motion_with_pid_control.launch	XML	82	0	11	0	93
behavior_catalog.yaml	YAML	238	55	44	0	337
belief_manager_config.yaml	YAML	8	0	3	0	11
mission1.py	Python	65	5	19	0	89
mission2.py	Python	66	3	25	0	94
drone_interception.launch	XML	14	3	3	0	20
launcher_gazebo.sh	Shell Script	29	2	5	0	36
main_launcher.sh	Shell Script	2	0	2	0	4
main_launcher1.sh	Shell Script	130	9	7	0	146
main_launcher2.sh	Shell Script	132	7	6	0	145
mission_launcher.sh	Shell Script	5	1	4	0	10

TABLA XV
DIMENSIONES TOTALES DEL SOFTWARE IMPLEMENTADO

Líneas de código funcional	Líneas comentadas	Líneas en blanco	Número de clases	Líneas totales
2421	697	527	7	3642

5 Conclusiones

Examinando los objetivos de este trabajo expuestos en el [primer capítulo](#) del presente trabajo, en este capítulo se presenta de forma minuciosa los distintos objetivos que se han logrado cumplir junto con las partes más relevantes que se han extraído de cada uno durante su realización:

- Estudio de las herramientas software. Se han estudiado las tecnologías necesarias para lograr diseñar comportamientos de robots sobre el entorno de Aerostack, haciendo hincapié en ROS y Gazebo.
- Análisis del problema de coordinación de robots aéreos. Se analizaron los comportamientos de robots aéreos existentes en Aerostack y se estudiaron las diferentes formas de extensión de comportamientos para operar en misiones coordinadas.
- Diseño y programación de los comportamientos haciendo uso de la forma de programación empleada en el entorno software Aerostack. Se han diseñado siete algoritmos orientados a realizar acciones en las que múltiples robots sean capaces de realizar acciones de persecución, huida, imitación y observación al operar en misiones de inspección o vigilancia. Estos algoritmos han sido implementados empleando el entorno software Aerostack.
- Evaluación de los comportamientos mediante la ejecución de misiones en escenarios realistas haciendo uso del simulador de robots Gazebo. Empleando el simulador de robots Gazebo, se ha evaluado cada behavior tanto de manera individual, midiendo diversos aspectos de cada comportamiento como la velocidad o la altitud como de forma conjunta, con una prueba de ejecución integrada en la que se emplearon alguno behaviors desarrollados en este trabajo. La evaluación realizada mostró que el diseño e implementación de los comportamientos fue satisfactoria para operar en misiones de robots aéreos.

6 Bibliografía

- [1] Gazebo simulator. (2020, December 12). In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Gazebo_simulator
- [2] Sanchez-Lopez, J. L., Molina, M., Bavle, H., Sampedro, C., Fernández, R. A. S., & Campoy, P. (2017). A multi-layered component-based approach for the development of aerial robotic systems: The Aerostack framework. *Journal of Intelligent & Robotic Systems*, 88(2), 683-709.
- [3] Molina, M. (2017). An execution engine for aerial robot mission plans. Technical University of Madrid, Tech. Rep.
- [4] Molina, M., Santamaria, P., & Carrera, A. (2021). Programming Robot Behaviors with Execution Management Functions. arXiv preprint arXiv:2103.06545.
- [5] Hepeyiler, M. (2021, January 5). ROS C++ Style Guide. In CppStyleGuide. Retrieved from http://wiki.ros.org/CppStyleGuide#Preprocessor_directives_.28.23if_vs_.23ifdef.29
- [6] Giernacki, W., Cieślak, J., Molina, M., & Campoy, P. (2020, September). Extensions of the open-source framework Aerostack 3.0 for the development of more interactive flights between UAVs. In 2020 International Conference on Unmanned Aircraft Systems (ICUAS) (pp. 10-16). IEEE.

7 Anexos

En este capítulo se muestran los distintos anexos que son mencionados a lo largo de esta memoria.

7.1 Anexo 1

Pseudocódigo 1. Follow Target

```
1:Funcion void <- FollowTargetWithPidControl::onConfigure()
2:    getPrivateParam("~estimated_pose_topic")
3:    getPrivateParam("~controllers_topic")
4:    getPrivateParam("~motion_reference_speed_topic")
5:    getPrivateParam("~motion_reference_pose_topic")
6:    getPrivateParam("~set_control_mode_service_name")
7:    getPrivateParam("~status_topic")
8:    getPrivateParam("~shared_robot_position_channel_topic")
9:    status_topic.subscribe <- status_sub
10:FinFuncion
11:
12:Funcion rsp.situation_occurs <- FollowTargetWithPidControl::checkSituation()
13:    Si status_msg.state == FLYING o status_msg.state == HOVERING Entonces
14:        rsp.situation_occurs <- true
15:    SiNo
16:        rsp.situation_occurs <- false
17:    FinSi
18:FinFuncion
19:
20:Funcion void <- FollowTargetWithPidControl::onActivate()
21:    self_localization_pose_sub <- estimated_pose_topic.subscribe
22:    shared_robot_pos_sub <- shared_robot_position_channel_topic.subscribe
23:    motion_reference_speed_topic <- motion_reference_speed_pub.advertise
24:    command_high_level_pub <- controllers_topic.advertise
25:    motion_reference_pose_pub <- motion_reference_pose_topic.advertise
26:    set_control_mode_client_srv <- set_control_mode_service_name.serviceClient
27:    arguments <- getArguments()
28:    config_file <- YAML::Load(arguments)
29:    Si config_file["target_name"].IsDefined() Entonces
30:        TARGET_NAME <- config_file["target_name"].asString
31:    SiNo
32:        setTerminationCause(PROCESS_FAILURE)
33:    FinSi
34:    Si config_file["safety_distance"].IsDefined() Entonces
35:        SAFETY_DISTANCE <- config_file["safety_distance"].asDouble
36:    SiNo
37:        SAFETY_DISTANCE <- 2.0
38:    FinSi
39:    Si config_file["maximum_speed"].IsDefined() Entonces
40:        MAXIMUM_SPEED <- config_file["maximum_speed"].asDouble
41:    SiNo
42:        MAXIMUM_SPEED <- 0.3
43:    FinSi
44:    Si config_file["yaw"].IsDefined() Entonces
45:        YAW <- config_file["yaw"].asString
46:    SiNo
47:        YAW <- "constant"
48:    FinSi
49:FinFuncion
50:
```

```

51:Funcion void <- FollowTargetWithPidControl::onExecute()
52:    distx <- shared_coord.position.x - estimated_pose.pose.position.x
53:    disty <- shared_coord.position.y - estimated_pose.pose.position.y
54:    distz <- shared_coord.position.z - estimated_pose.pose.position.z
55:    distxyz <- sqrt(pow(distx, 2) + pow(disty, 2) + pow(distz, 2));
56:    path_point.pose.position.z <- shared_coord.position.z
57:    Si YAW == "path_facing" Entonces
58:        orientation_quaternion.setRPY(0, 0, atan2((disty), (distx)))
59:        path_point.pose.orientation.w <- orientation_quaternion.getW()
60:        path_point.pose.orientation.x <- orientation_quaternion.getX()
61:        path_point.pose.orientation.y <- orientation_quaternion.getY()
62:        path_point.pose.orientation.z <- orientation_quaternion.getZ()
63:    FinSi
64:    setControlMode(GROUND_SPEED)
65:    motion_reference_pose_pub.publish(path_point);
66:    speedSelfRegulation()
67:    motion_reference_speed_pub.publish(motion_reference_speed)
68:    command_high_level_pub.publish(MOVE)
69:FinFuncion
70:
71:Funcion void <- FollowTargetWithPidControl::onDeactivate()
72:    motion_reference_speed.twist.linear.x <- 0.0
73:    motion_reference_speed.twist.linear.y <- 0.0
74:    motion_reference_speed.twist.angular.x <- 0.0
75:    motion_reference_speed.twist.angular.y <- 0.0
76:    motion_reference_speed_pub.publish(motion_reference_speed)
77:    command_high_level_pub.publish(HOVER)
78:    self_localization_pose_sub.shutdown()
79:    motion_reference_speed_pub.shutdown()
80:    command_high_level_pub.shutdown()
81:    motion_reference_pose_pub.shutdown()
82:    set_control_mode_client_srv.shutdown()
83:FinFuncion
84:
85:Funcion void <- FollowTargetWithPidControl::speedSelfRegulation()
86:    Si distxyz <= SAFETY_DISTANCE Entonces
87:        motion_reference_speed.twist.linear.x <- 0.0
88:        motion_reference_speed.twist.linear.y <- 0.0
89:    SiNo
90:        motion_reference_speed.twist.linear.x <- (MAXIMUM_SPEED * distx) / distxyz
91:        motion_reference_speed.twist.linear.y <- (MAXIMUM_SPEED * disty) / distxyz
92:    FinSi
93:    motion_reference_speed_pub.publish(motion_reference_speed)
94:FinFuncion

```

7.2 Anexo 2

Pseudocódigo 2. Move Away From Robot

```

1:Funcion void <- MoveAwayFromRobotWithPidControl::onConfigure()
2:    getPrivateParam("~estimated_pose_topic")
3:    getPrivateParam("~controllers_topic")
4:    getPrivateParam("~motion_reference_speed_topic")
5:    getPrivateParam("~motion_reference_pose_topic")
6:    getPrivateParam("~set_control_mode_service_name")
7:    getPrivateParam("~status_topic")
8:    getPrivateParam("~shared_robot_position_channel_topic")
9:    status_topic.subscribe <- status_sub
10:FinFuncion
11:
12:Funcion rsp.situation_occurs <- MoveAwayFromRobotWithPidControl::checkSituation()
13:    Si status_msg.state == FLYING o status_msg.state == HOVERING Entonces

```

```

14:     rsp.situation_occurs <- true
15:     SiNo
16:         rsp.situation_occurs <- false
17:     FinSi
18:FinFuncion
19:
20:Funcion void <- MoveAwayFromRobotWithPidControl::onActivate()
21:     self_localization_pose_sub <- estimated_pose_topic.subscribe
22:     shared_robot_pos_sub <- shared_robot_position_channel_topic.subscribe
23:     motion_reference_speed_topic <- motion_reference_speed_pub.advertise
24:     command_high_level_pub <- controllers_topic.advertise
25:     motion_reference_pose_pub <- motion_reference_pose_topic.advertise
26:     set_control_mode_client_srv <- set_control_mode_service_name.serviceClient
27:     arguments <- getParameters()
28:     config_file <- YAML::Load(arguments)
29:     Si config_file[“follower_name”].IsDefined() Entonces
30:         FOLLOWER_NAME <- config_file[“follower_name”].asString
31:     SiNo
32:         setTerminationCause(PROCESS_FAILURE)
33:     FinSi
34:     Si config_file[“separation_distance”].IsDefined() Entonces
35:         SEPARATION_DISTANCE <- config_file[“separation_distance”].asDouble
36:     SiNo
37:         SEPARATION_DISTANCE <- 3.0
38:     FinSi
39:     Si config_file[“maximum_speed”].IsDefined() Entonces
40:         MAXIMUM_SPEED <- config_file[“maximum_speed”].asDouble
41:     SiNo
42:         MAXIMUM_SPEED <- 0.3
43:     FinSi
44:     Si config_file[“yaw”].IsDefined() Entonces
45:         YAW <- config_file[“yaw”].asDouble
46:     SiNo
47:         YAW <- “constant”
48:     FinSi
49:FinFuncion
50:
51:Funcion void <- MoveAwayFromRobotWithPidControl::onExecute()
52:     distx <- shared_coord.position.x - estimated_pose.pose.position.x
53:     disty <- shared_coord.position.y - estimated_pose.pose.position.y
54:     distz <- shared_coord.position.z - estimated_pose.pose.position.z
55:     distxyz <- sqrt(pow(distx, 2) + pow(disty, 2) + pow(distz, 2));
56:     Si estimated_pose.pose.position.z > pose_z_ini Entonces
57:         pose_z_ini <- estimated_pose.pose.position.z
58:     SiNo
59:         path_point.pose.position.z <- pose_z_ini
60:     FinSi
61:     Si YAW == “path_facing” Entonces
62:         orientation_quaternion.setRPY(0, 0, atan2(-(disty), -(distx)))
63:         path_point.pose.orientation.w <- orientation_quaternion.getW()
64:         path_point.pose.orientation.x <- orientation_quaternion.getX()
65:         path_point.pose.orientation.y <- orientation_quaternion.getY()
66:         path_point.pose.orientation.z <- orientation_quaternion.getZ()
67:     FinSi
68:     Si YAW == “path_facing” Entonces
69:         orientation_quaternion.setRPY(0, 0, atan2((disty), (distx)))
70:         path_point.pose.orientation.w <- orientation_quaternion.getW()
71:         path_point.pose.orientation.x <- orientation_quaternion.getX()
72:         path_point.pose.orientation.y <- orientation_quaternion.getY()
73:         path_point.pose.orientation.z <- orientation_quaternion.getZ()
74:     FinSi
75:     setControlMode(GROUND_SPEED)
76:     motion_reference_pose_pub.publish(path_point);
77:     speedSelfRegulationTrue()
78:     motion_reference_speed_pub.publish(motion_reference_speed)

```

```

79:     command_high_level_pub.publish(MOVE)
80:FinFuncion
81:
82:Funcion void <- MoveAwayFromRobotWithPidControl::onDeactivate()
83:     motion_reference_speed.twist.linear.x <- 0.0
84:     motion_reference_speed.twist.linear.y <- 0.0
85:     motion_reference_speed.twist.angular.x <- 0.0
86:     motion_reference_speed.twist.angular.y <- 0.0
87:     motion_reference_speed_pub.publish(motion_reference_speed)
88:     command_high_level_pub.publish(HOVER)
89:     self_localization_pose_sub.shutdown()
90:     motion_reference_speed_pub.shutdown()
91:     command_high_level_pub.shutdown()
92:     motion_reference_pose_pub.shutdown()
93:     set_control_mode_client_srv.shutdown()
94:FinFuncion
95:
96:Funcion void <- MoveAwayFromRobotWithPidControl::speedSelfRegulation()
97:     Si distxyz >= 3 * SEPARATION_DISTANCE Entonces
98:         motion_reference_speed.twist.linear.x <- 0.0
99:         motion_reference_speed.twist.linear.y <- 0.0
100:    SiNo
101:        motion_reference_speed.twist.linear.x <- (-MAXIMUM_SPEED * distx) / distxyz
102:        motion_reference_speed.twist.linear.y <- (-MAXIMUM_SPEED * disty) / distxyz
103:    FinSi
104:    motion_reference_speed_pub.publish(motion_reference_speed)
105:FinFuncion

```

7.3 Anexo 3

Pseudocódigo 3. Reach Target Altitude

```

1:Funcion void <- ReachTargetAltitudeWithPidControl::onConfigure()
2:     getPrivateParam("~estimated_pose_topic")
3:     getPrivateParam("~controllers_topic")
4:     getPrivateParam("~motion_reference_speed_topic")
5:     getPrivateParam("~motion_reference_pose_topic")
6:     getPrivateParam("~set_control_mode_service_name")
7:     getPrivateParam("~status_topic")
8:     getPrivateParam("~shared_robot_position_channel_topic")
9:     status_topic.subscribe <- status_sub
10:FinFuncion
11:
12:Funcion rsp.situation_occurs <- ReachTargetAltitudeWithPidControl::checkSituation()
13:     Si status_msg.state == FLYING o status_msg.state == HOVERING Entonces
14:         rsp.situation_occurs <- true
15:     SiNo
16:         rsp.situation_occurs <- false
17:     FinSi
18:FinFuncion
19:
20:Funcion void <- ReachTargetAltitudeWithPidControl::onActivate()
21:     self_localization_pose_sub <- estimated_pose_topic.subscribe
22:     shared_robot_pos_sub <- shared_robot_position_channel_topic.subscribe
23:     motion_reference_speed_topic <- motion_reference_speed_pub.advertise
24:     command_high_level_pub <- controllers_topic.advertise
25:     motion_reference_pose_pub <- motion_reference_pose_topic.advertise
26:     set_control_mode_client_srv <- set_control_mode_service_name.serviceClient
27:     arguments <- getParameters()
28:     config_file <- YAML::Load(arguments)
29:     Si config_file["target_name"].IsDefined() Entonces

```

```

30:     TARGET_NAME <- config_file["target_name"].asString
31:     SiNo
32:         setTerminationCause(PROCESS_FAILURE)
33:     FinSi
34:     Si config_file["shift"].IsDefined() Entonces
35:         SHIFT <- config_file["shift"].asDouble
36:     SiNo
37:         SHIFT <- 0.0
38:     FinSi
39:     Si config_file["maximum_speed"].IsDefined() Entonces
40:         MAXIMUM_SPEED <- config_file["maximum_speed"].asDouble
41:         enable_speed3D <- true
42:     FinSi
43: FinFuncion
44:
45: Funcion void <- ReachTargetAltitudeWithPidControl::onExecute()
46:     Si shared_drone_coord.position.z > 0.5 ó
47:         shared_drone_coord.position.z + SHIFT > 0.5 Entonces
48:             altitude_point_to_achieve <- shared_drone_coord.position.z + SHIFT
49:         SiNo
50:             altitude_point_to_achieve <- 0.5
51:         FinSi
52:         Si enable_speed3D == true Entonces
53:             setControlMode(SPEED_3D)
54:             Si altitude_point_to_achieve < estimated_pose_msg.pose.position.z Entonces
55:                 MAXIMUM_SPEED = -(MAXIMUM_SPEED * distz) / distxyz
56:             FinSi
57:             motion_reference_speed.twist.linear.z = (MAXIMUM_SPEED * distz) / distxyz;
58:             motion_reference_speed_pub.publish(motion_reference_speed);
59:         SiNo
60:             setControlMode(GROUND_SPEED)
61:             path_point.pose.position.z <- altitude_point_to_achieve;
62:             motion_reference_pose_pub.publish(path_point);
63:         FinSi
64:         command_high_level_pub.publish(MOVE)
65: FinFuncion
66:
67: Funcion void <- ReachTargetAltitudeWithPidControl::onDeactivate()
68:     motion_reference_speed.twist.linear.z <- 0.0
69:     motion_reference_speed_pub.publish(motion_reference_speed)
70:     command_high_level_pub.publish(HOVER)
71:     self_localization_pose_sub.shutdown()
72:     motion_reference_speed_pub.shutdown()
73:     command_high_level_pub.shutdown()
74:     motion_reference_pose_pub.shutdown()
75:     set_control_mode_client_srv.shutdown()
76: FinFuncion
77:
78: Funcion void <- ReachTargetAltitudeWithPidControl::checkGoal()
79:     Si altitude_point_to_achieve - estimated_pose_msg.pose.position.z <= 0.1 Entonces
80:         setTerminationCause(GOAL_ACHIEVED)
81: FinFuncion

```

7.4 Anexo 4

Pseudocódigo 4. Take A Look At Target

```

1: Funcion void <- TakeALookAtTargetWithPidControl::onConfigure()
2:     getPrivateParam("~estimated_pose_topic")
3:     getPrivateParam("~controllers_topic")
4:     getPrivateParam("~motion_reference_pose_topic")

```

```

5:     getPrivateParam("~set_control_mode_service_name")
6:     getPrivateParam("~status_topic")
7:     getPrivateParam("~shared_robot_position_channel_topic")
8:     getPrivateParam("~motion_reference_speed_topic")
9:     status_topic.subscribe <- status_sub
10:FinFuncion
11:
12:Funcion rsp.situation_occurs <- TakeALookAtTargetWithPidControl::checkSituation()
13:   Si status_msg.state == FLYING o status_msg.state == HOVERING Entonces
14:     rsp.situation_occurs <- true
15:   SiNo
16:     rsp.situation_occurs <- false
17:   FinSi
18:FinFuncion
19:
20:Funcion void <- TakeALookAtTargetWithPidControl::onActivate()
21:   self_localization_pose_sub <- estimated_pose_topic.subscribe
22:   shared_robot_pos_sub <- shared_robot_position_channel_topic.subscribe
23:   command_high_level_pub <- controllers_topic.advertise
24:   motion_reference_pose_pub <- motion_reference_pose_topic.advertise
25:   set_control_mode_client_srv <- set_control_mode_service_name.serviceClient
26:   arguments <- getParameters()
27:   config_file <- YAML::Load(arguments)
28:   Si config_file["target_name"].IsDefined() Entonces
29:     TARGET_NAME <- config_file["target_name"].asString
30:   SiNo
31:     setTerminationCause(PROCESS_FAILURE)
32:   FinSi
33:FinFuncion
34:
35:Funcion void <- TakeALookAtTargetWithPidControl::onExecute()
36:   orientation_quaternion.setRPY(0, 0, atan2((shared_drone_coord.position.y -
37:     estimated_pose_msg.pose.position.y), (shared_drone_coord.position.x -
38:     estimated_pose_msg.pose.position.x)))
39:   path_point.pose.orientation.w <- orientation_quaternion.getW()
40:   path_point.pose.orientation.x <- orientation_quaternion.getX()
41:   path_point.pose.orientation.y <- orientation_quaternion.getY()
42:   path_point.pose.orientation.z <- orientation_quaternion.getZ()
43:   motion_reference_speed.twist.linear.x = 0.0
44:   motion_reference_speed.twist.linear.y = 0.0
45:   motion_reference_speed.twist.linear.z = 0.0
46:   setControlMode(SPEED_3D)
47:   motion_reference_speed_pub..publish(motion_reference_speed)
48:   motion_reference_pose_pub.publish(path_point);
49:   command_high_level_pub.publish(MOVE)
50:FinFuncion
51:
52:Funcion void <- TakeALookAtTargetWithPidControl::onDeactivate()
53:   command_high_level_pub.publish(HOVER)
54:   motion_reference_speed_pub.shutdown()
55:   self_localization_pose_sub.shutdown()
56:   command_high_level_pub.shutdown()
57:FinFuncion
58:
59:Funcion void <- TakeALookAtTargetWithPidControl::checkGoal()
60:   Si shared_robot_pos_received == true y abs(path_point.pose.orientation.z -
61:     estimated_pose_msg.pose.orientation.z) < 0.1
62:     y abs(path_point.pose.orientation.w - estimated_pose_msg.pose.orientation.w)
63:       < 0.1) Entonces
64:       setTerminationCause(GOAL_ACHIEVED)
65:FinFuncion

```

7.5 Anexo 5

Pseudocódigo 5. Behavior Get Close To Target

```
1:Funcion void <- GetCloseToTargetWithPidControl::onConfigure()
2:    getPrivateParam("~estimated_pose_topic")
3:    getPrivateParam("~controllers_topic")
4:    getPrivateParam("~motion_reference_speed_topic")
5:    getPrivateParam("~motion_reference_pose_topic")
6:    getPrivateParam("~set_control_mode_service_name")
7:    getPrivateParam("~status_topic")
8:    getPrivateParam("~shared_robot_position_channel_topic")
9:    status_topic.subscribe <- status_sub
10:FinFuncion
11:
12:Funcion rsp.situation_occurs <- GetCloseToTargetWithPidControl::checkSituation()
13:    Si status_msg.state == FLYING o status_msg.state == HOVERING Entonces
14:        rsp.situation_occurs <- true
15:    SiNo
16:        rsp.situation_occurs <- false
17:    FinSi
18:FinFuncion
19:
20:Funcion void <- GetCloseToTargetWithPidControl::onActivate()
21:    self_localization_pose_sub <- estimated_pose_topic.subscribe
22:    shared_robot_pos_sub <- shared_robot_position_channel_topic.subscribe
23:    motion_reference_speed_topic <- motion_reference_speed_pub.advertise
24:    command_high_level_pub <- controllers_topic.advertise
25:    motion_reference_pose_pub <- motion_reference_pose_topic.advertise
26:    set_control_mode_client_srv <- set_control_mode_service_name.serviceClient
27:    arguments <- getParameters()
28:    config_file <- YAML::Load(arguments)
29:    Si config_file["target_distance"].IsDefined() Entonces
30:        TARGET_NAME <- config_file["target_distance"].asString
31:    SiNo
32:        setTerminationCause(PROCESS_FAILURE)
33:    FinSi
34:    Si config_file["safety_distance"].IsDefined() Entonces
35:        TARGET_DISTANCE <- config_file["safety_distance"].asDouble
36:    SiNo
37:        TARGET_DISTANCE <- 2.0
38:    FinSi
39:    Si config_file["maximum_speed"].IsDefined() Entonces
40:        MAXIMUM_SPEED <- config_file["maximum_speed"].asDouble
41:    SiNo
42:        MAXIMUM_SPEED <- 0.3
43:    FinSi
44:    Si config_file["yaw"].IsDefined() Entonces
45:        YAW <- config_file["yaw"].asDouble
46:    SiNo
47:        YAW <- "constant"
48:FinFuncion
49:
50:Funcion void <- GetCloseToTargetWithPidControl::onExecute()
51:    distx <- shared_coord.position.x - estimated_pose.pose.position.x
52:    disty <- shared_coord.position.y - estimated_pose.pose.position.y
53:    distz <- shared_coord.position.z - estimated_pose.pose.position.z
54:    distxyz <- sqrt(pow(distx, 2) + pow(disty, 2) + pow(distz, 2));
55:    Si distxyz > TARGET_DISTANCE y distz >= 0.1 Entonces
56:        path_point.pose.position.z <- shared_coord.position.z
57:    FinSi
58:    Si YAW == "path_facing" Entonces
59:        orientation_quaternion.setRPY(0, 0, atan2((disty), (distx)))
```

```

60:     path_point.pose.orientation.w <- orientation_quaternion.getW()
61:     path_point.pose.orientation.x <- orientation_quaternion.getX()
62:     path_point.pose.orientation.y <- orientation_quaternion.getY()
63:     path_point.pose.orientation.z <- orientation_quaternion.getZ()
64: FinSi
65:     setControlMode(GROUND_SPEED)
66:     motion_reference_pose_pub.publish(path_point);
67:     speedSelfRegulation()
68:     motion_reference_speed_pub.publish(motion_reference_speed)
69:     command_high_level_pub.publish(MOVE)
70:FinFuncion
71:
72:Funcion void <- GetCloseToTargetWithPidControl::onDeactivate()
73:     motion_reference_speed.twist.linear.x <- 0.0
74:     motion_reference_speed.twist.linear.y <- 0.0
75:     motion_reference_speed.twist.angular.x <- 0.0
76:     motion_reference_speed.twist.angular.y <- 0.0
77:     motion_reference_speed_pub.publish(motion_reference_speed)
78:     command_high_level_pub.publish(HOVER)
79:     self_localization_pose_sub.shutdown()
80:     motion_reference_speed_pub.shutdown()
81:     command_high_level_pub.shutdown()
82:     motion_reference_pose_pub.shutdown()
83:     set_control_mode_client_srv.shutdown()
84:FinFuncion
85:
86:Funcion void <- GetCloseToTargetWithPidControl::speedSelfRegulation()
87:     Si distxyz <= SAFETY_DISTANCE Entonces
88:         motion_reference_speed.twist.linear.x <- 0.0
89:         motion_reference_speed.twist.linear.y <- 0.0
90:     SiNo
91:         motion_reference_speed.twist.linear.x <- (MAXIMUM_SPEED * distx) / distxyz
92:         motion_reference_speed.twist.linear.y <- (MAXIMUM_SPEED * disty) / distxyz
93:     FinSi
94:     motion_reference_speed_pub.publish(motion_reference_speed)
95:FinFuncion
96:
97:Funcion void <- GetCloseToTargetWithPidControl::checkGoal()
98:     Si estimated_pose_msg_received == true y shared_robot_pos_received == true
99:         y (TARGET_DISTANCE <= distxyz y distxyz < TARGET_DISTANCE + 0.1) Entonces
100:            setTerminationCause(GOAL_ACHIEVED)
100:FinFuncion

```

7.6 Anexo 6

Pseudocódigo 6. Keep Look At Target

```

1:Funcion void <- KeepLookAtTargetWithPidControl::onConfigure()
2:     getPrivateParam("~estimated_pose_topic")
3:     getPrivateParam("~controllers_topic")
4:     getPrivateParam("~motion_reference_pose_topic")
5:     getPrivateParam("~set_control_mode_service_name")
6:     getPrivateParam("~status_topic")
7:     getPrivateParam("~shared_robot_position_channel_topic")
8:     getPrivateParam("~motion_reference_speed_topic")
9:     status_topic.subscribe <- status_sub
10:FinFuncion
11:
12:Funcion rsp.situation_occurs <- KeepLookAtTargetWithPidControl::checkSituation()
13:     Si status_msg.state == FLYING o status_msg.state == HOVERING Entonces
14:         rsp.situation_occurs <- true

```

```

15:  SiNo
16:      rsp.situation_occurs <- false
17:  FinSi
18:FinFuncion
19:
20:Funcion void <- KeepLookAtTargetWithPidControl::onActivate()
21:     self_localization_pose_sub <- estimated_pose_topic.subscribe
22:     shared_robot_pos_sub <- shared_robot_position_channel_topic.subscribe
23:     command_high_level_pub <- controllers_topic.advertise
24:     motion_reference_pose_pub <- motion_reference_pose_topic.advertise
25:     set_control_mode_client_srv <- set_control_mode_service_name.serviceClient
26:     arguments <- getParameters()
27:     config_file <- YAML::Load(arguments)
28:     Si config_file[“target_name”].IsDefined() Entonces
29:         TARGET_NAME <- config_file[“target_name”].asString
30:     SiNo
31:         setTerminationCause(PROCESS_FAILURE)
32:     FinSi
33:FinFuncion
34:
35:Function void <- KeepLookAtTargetWithPidControl::onExecute()
36:     orientation_quaternion.setRPY(0, 0, atan2((shared_drone_coord.position.y -
37:         estimated_pose_msg.pose.position.y), (shared_drone_coord.position.x -
38:         estimated_pose_msg.pose.position.x)))
39:     path_point.pose.orientation.w <- orientation_quaternion.getW()
40:     path_point.pose.orientation.x <- orientation_quaternion.getX()
41:     path_point.pose.orientation.y <- orientation_quaternion.getY()
42:     path_point.pose.orientation.z <- orientation_quaternion.getZ()
43:     motion_reference_speed.twist.linear.x = 0.0
44:     motion_reference_speed.twist.linear.y = 0.0
45:     motion_reference_speed.twist.linear.z = 0.0
46:     setControlMode(SPEED_3D)
47:     motion_reference_pose_pub.publish(path_point);
48:     command_high_level_pub.publish(MOVE)
49:FinFuncion
50:
51:Funcion void <- KeepLookAtTargetWithPidControl::onDeactivate()
52:     command_high_level_pub.publish(HOVER)
53:     self_localization_pose_sub.shutdown()
54:     command_high_level_pub.shutdown()
55:     motion_reference_speed_pub.shutdown()
56:     motion_reference_pose_pub.shutdown()
57:     set_control_mode_client_srv.shutdown()
58:FinFuncion

```

7.7 Anexo 7

Pseudocódigo 7. Keep Target Altitude

```

1:Funcion void <- KeepTargetAltitudeWithPidControl::onConfigure()
2:    getPrivateParam(~estimated_pose_topic")
3:    getPrivateParam(~controllers_topic")
4:    getPrivateParam(~motion_reference_speed_topic")
5:    getPrivateParam(~motion_reference_pose_topic")
6:    getPrivateParam(~set_control_mode_service_name")
7:    getPrivateParam(~status_topic")
8:    getPrivateParam(~shared_robot_position_channel_topic")
9:    status_topic.subscribe <- status_sub
10:FinFuncion
11:
12:Funcion rsp.situation_occurs <- KeepTargetAltitudeWithPidControl::checkSituation()

```

```

13:   Si status_msg.state == FLYING o status_msg.state == HOVERING Entonces
14:     rsp.situation_occurs <- true
15:   SiNo
16:     rsp.situation_occurs <- false
17:   FinSi
18:FinFuncion
19:
20:Function void <- KeepTargetAltitudeWithPidControl::onActivate()
21:   self_localization_pose_sub <- estimated_pose_topic.subscribe
22:   shared_robot_pos_sub <- shared_robot_position_channel_topic.subscribe
23:   motion_reference_speed_topic <- motion_reference_speed_pub.advertise
24:   command_high_level_pub <- controllers_topic.advertise
25:   motion_reference_pose_pub <- motion_reference_pose_topic.advertise
26:   set_control_mode_client_srv <- set_control_mode_service_name.serviceClient
27:   arguments <- getParameters()
28:   config_file <- YAML::Load(arguments)
29:   Si config_file[“target_name”].IsDefined() Entonces
30:     TARGET_NAME <- config_file[“target_name”].asString
31:   SiNo
32:     setTerminationCause(PROCESS_FAILURE)
33:   FinSi
34:   Si config_file[“shift”].IsDefined() Entonces
35:     SHIFT <- config_file[“shift”].asDouble
36:   SiNo
37:     SHIFT <- 0.0
38:   FinSi
39:   Si config_file[“maximum_speed”].IsDefined() Entonces
40:     MAXIMUM_SPEED <- config_file[“maximum_speed”].asDouble
41:     enable_speed3D <- true
42:   FinSi
43:FinFuncion
44:
45:Function void <- KeepTargetAltitudeWithPidControl::onExecute()
46:   Si shared_drone_coord.position.z > 0.5 ó
47:     shared_drone_coord.position.z + SHIFT > 0.5 Entonces
48:       altitude_point_to_achieve <- shared_drone_coord.position.z + SHIFT
49:   SiNo
50:     altitude_point_to_achieve <- 0.5
51:   FinSi
52:   Si enable_speed3D == true y abs(estimated_pose_msg.pose.position.z -
53:     altitude_point_to_achieve) > 0.1 Entonces
54:     setControlMode(SPEED_3D)
55:     Si altitude_point_to_achieve < estimated_pose_msg.pose.position.z Entonces
56:       MAXIMUM_SPEED = -(MAXIMUM_SPEED * distz) / distxyz
57:     FinSi
58:     motion_reference_speed.twist.linear.z = (MAXIMUM_SPEED * distz) / distxyz;
59:     motion_reference_speed_pub.publish(motion_reference_speed);
60:   SiNo
61:     setControlMode(GROUND_SPEED)
62:     path_point.pose.position.z <- altitude_point_to_achieve;
63:     motion_reference_pose_pub.publish(path_point);
64:   FinSi
65:   command_high_level_pub.publish(MOVE)
66:FinFuncion
67:Function void <- KeepTargetAltitudeWithPidControl::onDeactivate()
68:   motion_reference_speed.twist.linear.z <- 0.0
69:   motion_reference_speed_pub.publish(motion_reference_speed)
70:   command_high_level_pub.publish(HOVER)
71:   self_localization_pose_sub.shutdown()
72:   motion_reference_speed_pub.shutdown()
73:   command_high_level_pub.shutdown()
74:   motion_reference_pose_pub.shutdown()
75:   set_control_mode_client_srv.shutdown()
76:FinFuncion

```

7.8 Anexo 8

```
#!/bin/bash

DRONE_SWARM_MEMBERS=3
MAV_NAME="hummingbird_adr"
APPLICATION_PATH=${PWD}

if [ -z $DRONE_SWARM_MEMBERS ] # Check if DRONE_SWARM_MEMBERS is NULL
then
    #Argument 1 empty
    echo "-Setting Swarm Members = 1"
    DRONE_SWARM_MEMBERS=1
else
    echo "-Setting DroneSwarm Members = $1"
fi

gnome-terminal \
--tab --title "bridge" --command "bash -c \"
roslaunch ${APPLICATION_PATH}/configs/gazebo_files/launch/suspension_bridge.launch
project:=${APPLICATION_PATH};"
exec bash\" &

gnome-terminal \
--tab --title "Spawn_mav1" --command "bash -c \"
roslaunch rotors_gazebo spawn_mav.launch --wait \
namespace:=${MAV_NAME}1 \
mav_name:=$MAV_NAME \
x:=0 \
y:=0 \
log_file:=${MAV_NAME}1 ; \
roslaunch rotors_gazebo spawn_mav.launch --wait \
namespace:=${MAV_NAME}2 \
mav_name:=$MAV_NAME \
x:=4 \
y:=0 \
log_file:=${MAV_NAME}2;
roslaunch rotors_gazebo spawn_mav.launch --wait \
namespace:=${MAV_NAME}3 \
mav_name:=$MAV_NAME \
x:=8 \
y:=0 \
log_file:=${MAV_NAME}3;
exec bash\" &
```

7.9 Anexo 9

```
#!/bin/bash
NUMID_DRONE=111
DRONE_SWARM_ID=1
MAV_NAME=hummingbird_addr
export APPLICATION_PATH=${PWD}
#-----
# INTERNAL PROCESSES
#-----
gnome-terminal \
#
#-----
# Basic Behaviors
#-----
--tab --title "Basic Behaviors" --command "bash -c \""
roslaunch basic_quadrotor_behaviors basic_quadrotor_behaviors.launch --wait \
    namespace:=drone$NUMID_DRONE;
exec bash\"";
gnome-terminal \
#
#-----
# Swarm behaviors
#-----
--tab --title "Swarm behaviors" --command "bash -c \""
roslaunch swarm_interaction swarm_interaction.launch --wait \
    namespace:=drone$NUMID_DRONE;
exec bash\"";
gnome-terminal \
#
#-----
# Gazebo motor speed controller
#-----
--tab --title "Gazebo motor speed controller" --command "bash -c \""
roslaunch motor_speed_controller motor_speed_controller.launch --wait \
    namespace:=drone$NUMID_DRONE \
    mav_name:=hummingbird;
exec bash\"";
gnome-terminal \
#
#-----
# Quadrotor Motion With PID Control
#-----
--tab --title "Quadrotor Motion With PID Control" --command "bash -c \""
```

```

roslaunch quadrotor_motion_with_pid_control quadrotor_motion_with_pid_control.launch
--wait \
    namespace:=drone$NUMID_DRONE \
    robot_config_path:=${APPLICATION_PATH}/configs/drone$NUMID_DRONE \
    uav_mass:=0.7;
exec bash"" ;
gnome-terminal \
#-----
# Python Interpreter
#-----
--tab --title "Python Interpreter" --command "bash -c \""
roslaunch python_based_mission_interpreter_process
python_based_mission_interpreter_process.launch --wait \
    drone_id_namespace:=drone$NUMID_DRONE \
    drone_id_int:=$NUMID_DRONE \
    mission:=mission1.py \
    mission_configuration_folder:=${APPLICATION_PATH}/configs/mission;
exec bash"" ;
gnome-terminal \
#-----
# Gazebo Interface
#-----
--tab --title "Gazebo Interface" --command "bash -c \""
roslaunch gazebo_interface gazebo_interface.launch --wait \
    robot_namespace:=drone$NUMID_DRONE \
    drone_id:=$DRONE_SWARM_ID \
    mav_name:=$MAV_NAME;
exec bash"" ;
gnome-terminal \
#-----
# Belief Manager
#-----
--tab --title "Belief Manager" --command "bash -c \""
roslaunch belief_manager_process belief_manager_process.launch --wait \
    drone_id_namespace:=drone$NUMID_DRONE \
    drone_id:=$NUMID_DRONE \
    config_path:=${APPLICATION_PATH}/configs/mission;
exec bash"" ;
gnome-terminal \
#-----
# Common Belief Updater
#-----

```

```

--tab --title "Common Belief Updater" --command "bash -c \"
roslaunch common_belief_updater_process common_belief_updater_process.launch --wait \
    drone_id_namespace:=drone$NUMID_DRONE \
    drone_id:=$NUMID_DRONE;
exec bash\"";
gnome-terminal \
#-----
# Belief Updater
#-----
--tab --title " Swarm Belief Updater" --command "bash -c \"
roslaunch swarm_belief_updater_process swarm_belief_updater_process.launch --wait \
    drone_id_namespace:=drone$NUMID_DRONE \
    drone_id:=$NUMID_DRONE;
exec bash\"";
gnome-terminal \
#-----
# Belief Memory Viewer
#-----
--tab --title "Belief memory Viewer" --command "bash -c \"
roslaunch belief_memory_viewer belief_memory_viewer.launch --wait \
    robot_namespace:=drone$NUMID_DRONE \
    drone_id:=$NUMID_DRONE;
exec bash\"";
gnome-terminal \
#-----
# Recovery Manager
#-----
--tab --title "Recovery Manager" --command "bash -c \"
roslaunch recovery_manager_process recovery_manager_process.launch --wait \
    robot_namespace:=drone$NUMID_DRONE \
    drone_id:=$DRONE_SWARM_ID;
exec bash\"";
gnome-terminal \
#-----
# Safety Monitor
#-----
--tab --title "Safety Monitor" --command "bash -c \"
roslaunch safety_monitor_process safety_monitor_process.launch --wait \
    robot_namespace:=drone$NUMID_DRONE;
exec bash\"";
gnome-terminal \
#-----
```

```

# Behavior Coordinator
#-----
--tab --title "Behavior coordinator" --command "bash -c \" sleep 2;
roslaunch behavior_coordinator behavior_coordinator.launch --wait \
robot_namespace:=drone$NUMID_DRONE \
catalog_path:=${APPLICATION_PATH}/configs/mission/behavior_catalog.yaml;
exec bash\" &
#-----

# SHELL INTERFACE
#-----
gnome-terminal \
#-----
# alphanumeric_viewer
#-----
--tab --title "alphanumeric_viewer" --command "bash -c \"
roslaunch alphanumeric_viewer alphanumeric_viewer.launch --wait \
drone_id_namespace:=drone$NUMID_DRONE;
exec bash\" &
rqt_image_view /hummingbird_adr1/camera_front/image_raw/compressed

```

7.10 Anexo 10

```

#!/bin/bash
NUMID_DRONE=112
DRONE_SWARM_ID=2
MAV_NAME=hummingbird_adr
export APPLICATION_PATH=${PWD}

#-----
# INTERNAL PROCESSES
#-----
gnome-terminal \
#-----
# Basic Behaviors
#-----
--tab --title "Basic Behaviors" --command "bash -c \"
roslaunch basic_quadrotor_behaviors basic_quadrotor_behaviors.launch --wait \
namespace:=drone$NUMID_DRONE;
exec bash\" ;
gnome-terminal \
#-----
```

```

# Swarm behaviors
#-----
--tab --title "Swarm behaviors" --command "bash -c \
roslaunch swarm_interaction swarm_interaction.launch --wait \
namespace:=drone$NUMID_DRONE;
exec bash\"";
gnome-terminal \
#-----


#-----#
# Gazebo motor speed controller
#-----
--tab --title "Gazebo motor speed controller" --command "bash -c \
roslaunch motor_speed_controller motor_speed_controller.launch --wait \
namespace:=drone$NUMID_DRONE \
mav_name:=hummingbird;
exec bash\"";
gnome-terminal \
#-----


#-----#
# Quadrotor Motion With PID Control
#-----
--tab --title "Quadrotor Motion With PID Control" --command "bash -c \
roslaunch quadrotor_motion_with_pid_control quadrotor_motion_with_pid_control.launch \
--wait \
namespace:=drone$NUMID_DRONE \
robot_config_path:=${APPLICATION_PATH}/configs/drone$NUMID_DRONE \
uav_mass:=0.7;
exec bash\"";
gnome-terminal \
#-----


#-----#
# Python Interpreter
#-----
--tab --title "Python Interpreter" --command "bash -c \
roslaunch python_based_mission_interpreter_process \
python_based_mission_interpreter_process.launch --wait \
drone_id_namespace:=drone$NUMID_DRONE \
drone_id_int:=$NUMID_DRONE \
mission:=mission2.py \
mission_configuration_folder:=${APPLICATION_PATH}/configs/mission;
exec bash\"";
gnome-terminal \
#-----


#-----#
# Gazebo Interface

```

```

#-----
--tab --title "Gazebo Interface" --command "bash -c \"
roslaunch gazebo_interface gazebo_interface.launch --wait \
    robot_namespace:=drone$NUMID_DRONE \
    drone_id:=$DRONE_SWARM_ID \
    mav_name:=$MAV_NAME;
exec bash\"";
gnome-terminal \
#-----

# Belief Manager
#-----
--tab --title "Belief Manager" --command "bash -c \"
roslaunch belief_manager_process belief_manager_process.launch --wait \
    drone_id_namespace:=drone$NUMID_DRONE \
    drone_id:=$NUMID_DRONE \
    config_path:=${APPLICATION_PATH}/configs/mission;
exec bash\"";
gnome-terminal \
#-----

# Common Belief Updater
#-----
--tab --title "Common Belief Updater" --command "bash -c \"
roslaunch common_belief_updater_process common_belief_updater_process.launch --wait \
    drone_id_namespace:=drone$NUMID_DRONE \
    drone_id:=$NUMID_DRONE;
exec bash\"";
gnome-terminal \
#-----

# Belief Updater
#-----
--tab --title " Swarm Belief Updater" --command "bash -c \"
roslaunch swarm_belief_updater_process swarm_belief_updater_process.launch --wait \
    drone_id_namespace:=drone$NUMID_DRONE \
    drone_id:=$NUMID_DRONE;
exec bash\"";
gnome-terminal \
#-----

# Belief Memory Viewer
#-----
--tab --title "Belief memory Viewer" --command "bash -c \"
roslaunch belief_memory_viewer belief_memory_viewer.launch --wait \

```

```

robot_namespace:=drone$NUMID_DRONE \
drone_id:=$NUMID_DRONE;
exec bash\" ;
gnome-terminal \
#-----
# Recovery Manager
#-----
--tab --title "Recovery Manager" --command "bash -c \"
roslaunch recovery_manager_process recovery_manager_process.launch --wait \
robot_namespace:=drone$NUMID_DRONE \
drone_id:=$DRONE_SWARM_ID;
exec bash\" ;
gnome-terminal \
#-----
# Safety Monitor
#-----
--tab --title "Safety Monitor" --command "bash -c \"
roslaunch safety_monitor_process safety_monitor_process.launch --wait \
robot_namespace:=drone$NUMID_DRONE;
exec bash\" ;
gnome-terminal \
#-----
# Behavior Coordinator
#-----
--tab --title "Behavior coordinator" --command "bash -c \" sleep 2;
roslaunch behavior_coordinator behavior_coordinator.launch --wait \
robot_namespace:=drone$NUMID_DRONE \
catalog_path:=${APPLICATION_PATH}/configs/mission/behavior_catalog.yaml;
exec bash\" &

#-----
# SHELL INTERFACE
#-----
gnome-terminal \
#-----
# alphanumeric_viewer
#-----
--tab --title "alphanumeric_viewer" --command "bash -c \"
roslaunch alphanumeric_viewer alphanumeric_viewer.launch --wait \
drone_id_namespace:=drone$NUMID_DRONE;
exec bash\" &

```

```
rqt_image_view /hummingbird_ardrone/camera_front/image_raw/compressed
```

7.11 Anexo 11

```
#!/usr/bin/env python3

import mission_execution_control as mxc
import rospy
import time
import math
import random

def isInMatrix(x1, y1, x4, y4, x, y):
    if x <= x1:
        return False
    if x >= x4:
        return False
    if y >= y1:
        return False
    if y <= y4:
        return False
    return True

def isInside(x1, y1, x2, y2, x3, y3, x4, y4, x, y):
    if(y1 == y2):
        return isInMatrix(x1, y1, x4, y4, x, y)
    l = abs(y4 - y3)
    k = abs(x4 - x3)
    s = math.sqrt(k * k + l * l)
    sin = l / s
    cos = k / s
    x1R = cos * x1 + sin * y1
    y1R = -x1 * sin + y1 * cos
    x4R = cos * x4 + sin * x4
    y4R = -x4 * sin + y4 * cos
    xR = cos * x + sin * y
    yR = -x * sin + y * cos
    return isInMatrix(x1R, y1R, x4R, y4R, xR, yR)

def mission():
    print("Starting mission...")
    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')
```

```

print("Informing position to robots...")
mxc.startTask('INFORM_POSITION_TO_ROBOTS')

print("Taking off...")
mxc.executeTask('TAKE_OFF')

# Points to visit
points_to_go = []
for x in range(-5,15):
    for y in range(-5,15):
        points_to_go.append([x, y, 1])

# Obtain position (x, y)
_, droneNumber111 = mxc.queryBelief('name(?N,drone111)')
_, pos = mxc.queryBelief('position(' + str(droneNumber111['N']) + ', (?X,?Y,?))')
inside_perimeter = isInside(x1= 0, y1= 1, x2= 0, y2= 10, x3= 9, y3= 10, x4= 9, y4= 1, x= pos['X'], y= pos['Y'])

# While not inside perimeter visit random points
while inside_perimeter == False:
    _, pos = mxc.queryBelief('position(' + str(droneNumber111['N']) + ', (?X,?Y,?))')
    inside_perimeter = isInside(x1= 0, y1= 1, x2= 0, y2= 10, x3= 9, y3= 10, x4= 9, y4= 1, x= pos['X'], y= pos['Y'])
    x = random.randint(0, 399)
    print("Following path...")
    mxc.startTask('FOLLOW_PATH', path=[points_to_go[x]], yaw='path_facing')
    time.sleep(1)
    print("Stopping follow_path...")
    mxc.stopTask('FOLLOW_PATH')

print("Moving away from robot...")
mxc.startTask('MOVE_AWAY_FROM_ROBOT_WITH_PID_CONTROL', follower_name = "drone112",
separation_distance = 3.0, maximum_speed = 0.4, yaw = "path_acing")

# When message arrives, the drone stops
unset_stop_msg = True
while unset_stop_msg:
    msg_in_memory = mxc.trueBelief('text(?, stop_drone111)')
    if msg_in_memory:
        unset_stop_msg = False
        mxc.stopTask('MOVE_AWAY_FROM_ROBOT_WITH_PID_CONTROL')

print("Landing...")
mxc.executeTask('LAND')
print('Mission completed.')

```

7.12 Anexo 12

```
#!/usr/bin/env python3

import mission_execution_control as mxc
import rospy
import time
import random
import math

def isInMatrix(x1, y1, x4, y4, x, y):
    if x <= x1:
        return False
    if x >= x4:
        return False
    if y >= y1:
        return False
    if y <= y4:
        return False
    return True

def isInside(x1, y1, x2, y2, x3, y3, x4, y4, x, y):
    if(y1 == y2):
        return isInMatrix(x1, y1, x4, y4, x, y)
    l = abs(y4 - y3)
    k = abs(x4 - x3)
    s = math.sqrt(k * k + l * l)
    sin = l / s
    cos = k / s
    x1R = cos * x1 + sin * y1
    y1R = -x1 * sin + y1 * cos
    x4R = cos * x4 + sin * x4
    y4R = -x4 * sin + y4 * cos
    xR = cos * x + sin * y
    yR = -x * sin + y * cos
    return isInMatrix(x1R, y1R, x4R, y4R, xR, yR)

def mission():
    print("Starting mission...")

    print("Paying attention to robots...")
    mxc.startTask('PAY_ATTENTION_TO_ROBOT_MESSAGES')

    print("Informing position to robots...")
```

```

mxc.startTask('INFORM_POSITION_TO_ROBOTS')

print("Taking off...")
mxc.executeTask('TAKE_OFF')
# Points inside the square to patrol it
points_inside_perimeter = []
for x in range(2, 10):
    for y in range(1, 9):
        points_inside_perimeter.append([x, y, 1])

# Extract memory identifier from drones to be neutralised
_, droneNumber111 = mxc.queryBelief('name(?N,drone111)')
_, pos = mxc.queryBelief('position(' + str(droneNumber111['N']) + ', (?X,?Y,?))')
drone111_arrested = False
while drone111_arrested == False:
    _, pos111 = mxc.queryBelief('position(' + str(droneNumber111['N']) + ', (?X,?Y,?))')
    x = random.randint(0, 63)
    print("Following path...")
    mxc.startTask('FOLLOW_PATH', path= [points_inside_perimeter[x]], yaw='path_facing')
    drone111_inside_perimeter = isInside(x1= 0, y1= 1, x2= 0, y2= 10, x3= 9, y3= 10,
                                         x4= 9, y4= 1, x= pos['X'], y= pos['Y'])
    if drone111_inside_perimeter:
        print("Following drone...")
        mxc.executeTask('GET_CLOSE_TO_TARGET_WITH_PID_CONTROL', target_name=
"drone111", target_distance= 1.5, maximum_speed= 0.5, yaw= "target_facing")
        time.sleep(1)
        print("Sending stop_drone111 message...")
        mxc.executeTask('INFORM_ROBOTS', receiver= 'drone111', message= 'stop_drone111')
        drone111_arrested = True
    time.sleep(1)
    print("Stopping follow_path...")
    mxc.stopTask('FOLLOW_PATH')
    print("Going to initial x,y points...")
    mxc.executeTask('FOLLOW_PATH', path= [[4.468, 5.554, 1]], yaw= 'path_facing')
    print("Landing...")
    mxc.executeTask('LAND')
    print('Mission completed.')

```