Ingeniería de Telecomunicaciones

Curso académico 2014/2015

Proyecto Fin de Carrera

Cuadricóptero AR.Drone en Gazebo y JdeRobot

Autor:
Daniel Yagüe Sánchez
Tutor:
José María Cañas Plaza

Una copia de este proyecto, las fuentes del programa y vídeos de los experimentos están disponibles en la siguiente dirección:

`http://jderobot.org/Daniyague-pfc`

Resumen

La robótica aérea ha experimentado un auge sin precedentes en los últimos años. Los robots aéreos, comúnmente conocidos como *drones*, suelen incluir a bordo sensores como el resto de robots y sus actuadores les permiten movimientos versátiles. Cada vez son más las aplicaciones en las que destacan como el transporte de mercancías, inspección de instalaciones en mal estado, entretenimiento y localización de supervivientes en desastres naturales, entre muchas otras. Muchas empresas (Google, Amazon, Parrot...) apuestan por esta tecnología y abren con ellas nuevas líneas de investigación.

Programar a un robot aéreo para realizar un trabajo reduce costes, tiempo y riesgos para las personas. Para alcanzar la madurez del *software* que proporcione a un *drone* la capacidad de realizar una tarea, los simuladores robóticos son una potente herramienta. Permiten recrear mundos virtuales sobre un ordenador que simulan robots y emulan el comportamiento que estos tendrían en la realidad al integrar en él el programa en cuestión. Muchos cuentan con motores de físicas, que simulan las fuerzas y reacciones ante determinados eventos, como la gravedad, la repulsión de dos cuerpos tras un choque o el viento. Esto supone un gran método de depuración y facilita el avance del *software* antes de emplear los robots reales.

Este Proyecto Fin de Carrera abarca ambos campos. El objetivo es proporcionar un soporte para robots aéreos en el entorno JdeRobot dentro del simulador Gazebo, de manera que estos respondan de manera realista al ser programados por cualquier aplicación del mismo entorno para *drones*. Junto con este soporte se diseñaron varias aplicaciones de navegación para validarlo experimentalmente. En concreto aplicaciones de seguimiento de balizas valiéndose de la información de posición, seguimiento de una carretera por medio de la cámara ventral, localización de objetos en un área de búsqueda y localización y seguimiento de otro *drone*.

Como plataforma robótica se ha empleado JdeRobot 5.1. El lenguaje de programación ha sido C++. La biblioteca empleada para tratamiento de imágenes ha sido OpenCV 2.4. El simulador que se ha utilizado es Gazebo, en concreto las versiones 1.8 y 5.0.

Índice general

Capítulo 1

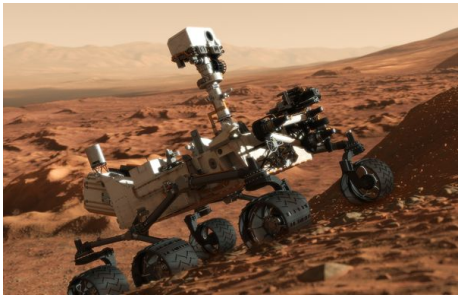
Introducción

El proyecto que explica esta memoria se encuadra dentro del campo de la robótica aérea y de los simuladores. Es por tanto preciso, en este primer capítulo, dar un contexto en relación a estos campos. En las siguientes páginas se dará una visión general acerca de qué es la robótica, un breve repaso acerca de su desarrollo a lo largo de la historia y su situación actual. También se hará énfasis en los campos de la simulación y de la robótica aérea. De igual manera, algunos de los conceptos necesarios para la comprensión de este trabajo serán expuestos en las siguientes secciones.

1.1. Robótica

La **robótica** es la "técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales"[1]. Es una rama tecnológica que ha permitido, mediante el diseño y creación de robots, grandes avances no solo en tareas que antes desarrollaban personas, sino además en otras que suponen gran dificultad para ser realizadas por estas o se realizan en entornos peligrosos. Un ejemplo es el robot *rover* Curiosity (o MSL por sus siglas en inglés Mars Science Laboratory) que explora Marte, robots industriales que trabajan en hornos a altas temperaturas, o robots para uso doméstico, como la aspiradora Roomba, y robots usados en desastres como el ocurrido en la central nuclear de Fukushima[3].

(a) Brazo robótico industrial.



(b) *Rover* robótico Curiosity.

(c) Aspiradora Roomba

Figura 1.1: Ejemplos de robots

Morfología

Entre los componentes *hardware* de un robot pueden distinguirse fuentes de alimentación, sensores, actuadores, controladores, memoria y dispositivos de comunicación. Los sensores y actuadores permiten al robot interactuar con el entorno. Los primeros son el equivalente en la máquina de los órganos sensoriales en nosotros, ya que obtienen medidas de distintas magnitudes como pueden ser temperatura, presión, distancia de objetos cercanos, o la batería que le queda. Los segundos permiten la manipulación de objetos o la locomoción del robot, igual que a las personas nos lo permiten piernas, brazos, cuello, etc. Es el software del robot el que le da la "inteligencia", es decir, la capacidad de llevar a cabo con autonomía sus tareas.

Clasificación

Existen distintos criterios[2] de clasificacion de robots. Uno de ellos es la aplicación:

- **Industriales**: Suelen ser brazos articulados empleados para sujetar o manejar piezas, pintar, soldar...
- **Domésticos**: Llevan a cabo tareas en el hogar. Desde limpiar suelos o piscinas hasta vigilar.
- **Uso médico**: Desde ayudar en operaciones quirúrgicas hasta apoyar en el tratamiento de enfermos de Alzheimer.
- **Militares**: *Drones* de reconocimiento, tareas de rescate, colocar/desactivar bombas...
- **Espaciales**: Sondas y *rovers* enviados sin compañía humana, o bien robots para apoyo en misiones tripuladas.
- **Entretenimiento**: Juguetes, educación...

También es posible agruparlos según su modo de locomoción:

- **Estacionarios**: Aunque puedan mover articulaciones (como los brazos industriales), no están diseñados para cambiar de posición.
- **Ruedas**: Una (en forma de bola), dos (*segway*), tres o más ruedas.
- **Piernas**: También pueden tener una o varias piernas.
- **Aéreos**: Utilizan alas, hélices o propulsores.
- **Acuáticos**: Pueden moverse con aletas o hélices igual que los anteriores.

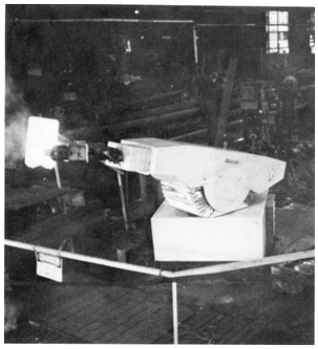
Historia

El término *robot* fue publicado por primera vez por el checo Karel Capek en *Rossum's Universal Robots* (1921), mientras que la palabra *robótica* fue usada por primera vez en la obra *¡Embustero!* (*Liar!*), de Isaac Asimov, en 1941.

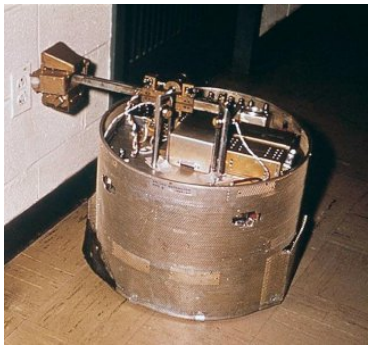
Los precursores de esta disciplina son los autómatas. Son máquinas con apariencia de seres animados que realizan movimientos propios de estos. Su origen se remonta a la prehistoria, con máquinas como la

estatua de Memón en Etiopía, que emitía sonidos al recibir luz. Leonardo da Vinci (1452-1519) diseñó un león mecánico y consiguió que anduviese por una habitación. Es famoso el autómata de Henri Maillardet, creado alrededor del 1800, que realizaba varios dibujos y escribía poemas, pero siempre los mismos. Estos son solo algunos de los muchos ejemplos de autómatas en la historia.

Hasta la segunda mitad del siglo XX no aparecieron los primeros robots reales, fuera de la ciencia ficción. Los primeros modelos surgen en los años 50 y funcionaban en entornos muy controlados. Unimate (figura 1.2a) fue el primer robot comercial y se usó en la fabricación de automóviles. En los 60 fue desarrollada *la bestia* (figura 1.2b), un pequeño robot con capacidad para explorar paredes en busca de enchufes para recargarse. Otros ejemplos posteriores son *Shakey* y su sucesor *Flakey* (figura 1.2c), cuyo propósito era desplazarse y evitar obstáculos. Ya en los 90 se desarrollaron las técnicas de creación de mapas y de navegación en entornos no estructurados. *Xavier*[6] fue diseñado con este propósito.



(a) Brazo robótico Unimate.



(b) "La bestia".

(c) Flakey.

Figura 1.2: Algunos de los primeros robots.

A finales del siglo XX aparecen los primeros humanoides, robots con apariencia humana. Algunas funciones de los humanoides son realizar tareas de asistencia a enfermos o personas mayores, investigación para mejorar extremidades ortopédicas, entretenimiento, algunos trabajos como recepcionista o en el sector industrial, así como probar vehículos o herramientas diseñados para la forma humana. *ASIMO* es uno de estos humanoides, y es capaz de andar, correr, reconocer objetos en movimiento, gestos y posturas, y apartarse al encontrarse con gente, entre otros.

Actualidad

En la actualidad hay numerosos ejemplos que plasman la integración de la robótica en distintas tareas y campos. KIVA Systems es una empresa que desarrolló robots para conducir estanterías móviles que permiten almacenar rápidamente productos y gestionar automáticamente la localización y ordenación de estos [22]. La empresa Amazon compró KIVA Systems en 2012 dada la gran utilidad que sus robots tendrían en los almacenes de la primera.

Como robots domésticos podemos destacar a la aspiradora Roomba o al friegasuelos Scooba, de la compañía iRobot. Estos son pequeños robots que recorren salas del hogar realizando su tarea para evitar el uso de fregonas o aspiradoras convencionales. Cuentan con sistemas de sensores para encontrar caminos en casas de cualquier forma, cubriendo todas las zonas del suelo [23].

Hoy día también existen usos en el campo de la cirugía. El sistema quirúrgico Da Vinci es un robot

Figura 1.3: Humanoide Asimo.

no autónomo que sirve de ayuda al cirujano en una operación. Este controla al robot mientras opera al paciente, de forma que puede realizar la intervención sentado. Da Vinci aporta una mayor precisión reduciendo el posible temblor de los moviemientos humanos. Cuenta con una cámara endoscópica de alta definición que elimina la necesidad de que un auxiliar sostenga la cámara en las laparoscopias.

En cuanto al ámbito militar, el robot PackBot (iRobot) es usado por el ejército estadounidense para inspeccionar explosivos, localizar agentes químicos o radiactivos en el entorno y localizar la procedencia de disparos o explosiones, entre otros. Este modelo fue usado en el accidente en la central nuclear de Fukushima para inspeccionar las plantas inferiores, aportar información de lo que ocurre y de los niveles de radiación y retirar escombros.

(a) Robots KIVA bajo estanterías.

(b) Sistema quirúrgico Da Vinci.

(c) PackBot.

Figura 1.4: Robots en la actualidad.

1.2. Robótica aérea

Una rama de la robótica en auge y con creciente interés en la actualidad es la robótica aérea. Los robots aéreos son llamados UAV (Unmanned Aerial Vehicle, Vehículo Aéreo No Tripulado), comúnmente denominados *drones*. Se trata de aeronaves capaces de volar con autonomía sin la presencia de un piloto o personal que lo controle remotamente, aunque también se permita esta función.

Historia

Sus orígenes se remontan a la Primera Guerra Mundial, cuando aparecieron los primeros blancos aéreos (1916). También aparecen entonces los precursores de los misiles (Aeroplano Automático de Hewitt-Sperry). Después de la guerra, estos objetivos pasan a volar por control remoto (*Fairey Queen*, 1931). El *Queen Bee*, en 1935, fue el primero de estos objetivos con capacidad de reutilización. En esta década el veterano británico de la Primera Guerra Mundial Reginald Denny trabajó en EEUU en la creación de varios de estos aeroplanos, desde el *RP-1* hasta el *RP-4*. También se investigó el uso de *drones* de ataque. Fueron usados, si bien de forma limitada, durante la Segunda Guerra Mundial (TDN-1, Project Fox...).

Los avances continúan durante la Guerra Fría. El *RP-71*, posteriormente conocido como *MQM-57 Falconer*, fue el primer BTT (*Basic Training Targets*) que evolucionó para ser utilizado con funciones de reconocimiento de terreno, y tuvo su primer vuelo en 1955. Drones de reconocimiento fueron usados también en la guerra de Vietnam, en China y Corea.

La modificación del modelo *B-17 Flying Fortress* permitió utilizar varios como *drones* para reunir datos radiactivos en pruebas nucleares siendo enviados cerca de la nube provocada por la explosión. Esto ocurrió en las pruebas realizadas en el Atolón Bikini.

En la década de los 70 fueron desarrollados en Israel varios modelos importantes. El *Firebee 1241* fue diseñado a partir del *Firebee* estadounidense, y se usó con funciones de reconocimiento y como señuelo. El *Scout* fue un modelo ligero y pequeño difícil de detectar y derribar que transmitía datos procedentes de su radar, que realizaba barridos de 360º. Ya en los 80, Israel y EEUU desarrollaron el *Pioneer*. Este UAV podía volar por trayectorias preprogramadas, con piloto automático o controlado desde tierra. Era necesario monitorizar la posición del Pioneer con un enlace radio. Contaba con una autonomía de 5'5 horas. El ejército estadounidense lo utilizó en la Guerra del Golfo.

Clasificación y usos

Podemos encontrar varios tipos de *drones* atendiendo a su forma y componentes materiales. Los UAV de ala fija son similares a pequeños aviones y despegan y aterrizan del mismo modo. Estos son capaces de alcanzar altas velocidades. Un ejemplo es el UAV MAVinci. Otro tipo es el de fuselaje sustentador, el cual carece de alas y se sirve del propio cuerpo para producir la fuerza de sustentación que le permite volar. Los de ala rotatoria se asemejan a los helicópteros. Suelen tener cuatro o más motores (cuadricópteros, octocópteros, etc). Tienen la ventaja de poder permanecer cernidos en un punto fijo. Ejemplos de cuadricópteros son el X5C de Syma y el ArDrone de Parrot.

En la actualidad los UAV tienen utilidad en múltiples campos. Algunos de ellos son los siguientes:

- **Militares.** Blancos móviles aéreos, reconocimiento de terreno y combate, entre otras tareas.

- **Vigilancia.** Seguridad en hogares, vigilancia de autopistas, costas, etc.
- **Inspección y reparaciones.** Fotografiar torres eléctricas, oleoductos, presas, gaseoductos, molinos eólicos, puentes, plataformas petrolíferas, etc, con el objetivo de vigilar o buscar daños que deban repararse.
- **Filmación.** Grabación de video para retransmisiones deportivas, anuncios o escenas de cine difíciles de grabar con cámaras convencionales.
- **Sondas de investigación.** Es posible enviar UAV para obtener datos a partir de sus sensores o tomar muestras de partículas, microorganismos, etc. Por ejemplo, se han realizado estudios de huracanes por medio de medidas de presión y temperatura tomadas por UAV enviados al huracán.
- **Rescates.** Es más eficiente para rescatar personas que hayan sufrido accidentes en el mar, montañas, u otras zonas de difícil acceso, o bien víctimas de desastres naturales, contar con la ayuda de UAV que faciliten la localización de supervivientes.
- **Detección de incendios.** Otra utilidad es la detección de focos de fuego, por ejemplo en incendios forestales. En general son útiles para la conservación de reservas naturales o zonas protegidas.

Actualidad

Como ya hemos comentado, existen muchas líneas de investigación con los UAV en la actualidad. Podemos destacar el prototipo de Google de su programa Project Wing [24] (figura 1.5a), que consiste en un híbrido entre motores de hélices y ala fija. Combina las ventajas de ambos, pudiendo despegar y aterrizar como hace un helicóptero y desplazarse con mayor velocidad gracias a sus alas. Una de sus funciones es llevar a cabo envíos de mercancías. El UAV podrá disminuir su velocidad hasta permanecer cernido sobre el punto de entrega par hacer descender el producto y acelerar una vez se ha entregado.

También en la línea de transporte de productos está el *drone* diseñado por Amazon en su sistema de entregas a domicilio Prime Air [26] (figura 1.5b). La empresa tiene previsto utilizar cuadricópteros para enviar las compras a sus clientes en pocas horas o minutos.

Parrot es una empresa orientada al desarrollo y a la comercialización de dispositivos inalámbricos y de manos libres para uso doméstico. En 2010 introdujo el ArDrone: un cuadricóptero con fines de entretenimiento junto con una API de código abierto. Permite a los programadores conectar sus aplicaciones al UAV mediante WiFi. En la última versión, la 2.0, puede controlarse desde 50 metros de distancia, cuenta con cámaras HD y con sensor GPS. Otro *drone* de la misma compañía es el Bebop, más pequeño que el ArDrone y orientado a las fotografías [25] (figura 1.5c).

Cuadricópteros

Dentro de este proyecto prestaremos atención a un tipo contreto de UAV, los **cuadricópteros**. Será también necesario para la comprensión del trabajo expuesto el conocer los principios físicos que rigen el vuelo de estos robots.

Un cuadricóptero es básicamente un helicóptero con cuatro rotores en los extremos de unas extremidades que forman una cruz. La figura 1.6 muestra el alzado del modelo ArDrone, de la empresa Parrot,

(a) Project Wing de Google. (b) Prime Air de Amazon. (c) Bebop de Parrot.

Figura 1.5: Ejemplos actuales de UAVs.

para diferenciar los rotores frontal-derecho, frontal- izquierdo, trasero-derecho y trasero-izquierdo. Esta configuración puede variar y tener un único motor frontal y un único motor trasero, al igual que con el derecho y el izquierdo, formando un "+" en lugar de una "x".

Figura 1.6: ArDrone en modo interiores (izq) y en modo exteriores (dcha). Los rotores de la parte superior de la imagen corresponden a los frontales.

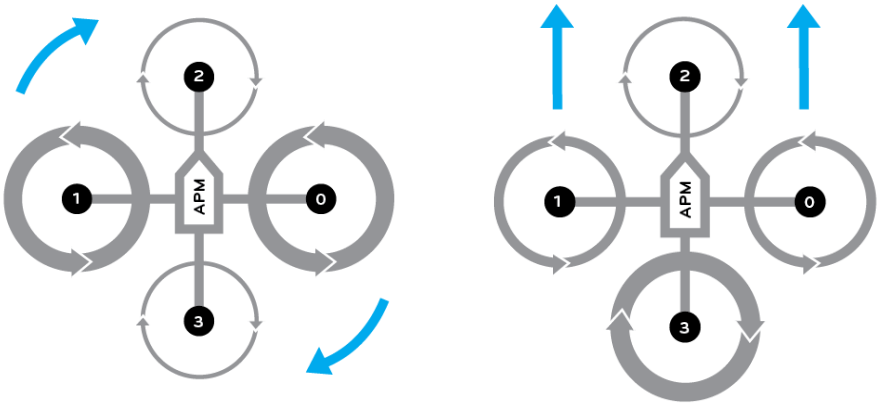
Las hélices de los rotores, al girar, producen una fuerza de empuje hacia arriba llamada sustentación, o *lift* en inglés, y que es la misma que hace elevarse a los helicópteros normales y a los aviones. Esta fuerza es perpendicular a la velocidad del fluido relativa a la hélice y está contenida en el plano definido por la misma velocidad y la normal a la superficie de la hélice, como puede verse en la figura 1.7. Para que el *drone* despegue, la suma de fuerzas provocadas por cada rotor debe superar su peso. Una vez en el aire, si la suma de fuerzas es igual al peso, el *drone* permanecerá en una altitud fija o cernido (*hovering*). Para aterrizar, o desplazarse hacia abajo, es necesario hacer que la fuerza resultante sea algo menor que la del peso.

Un problema que tienen que afrontar los helicópteros de un solo rotor, es que este produce una fuerza de torsión en el sentido de giro, por lo que es necesario otro rotor más pequeño perpendicular al principal para producir otra fuerza de sustentación que se oponga a la torsión y que el helicóptero no esté continuamente dando vueltas en torno a su eje vertical. En el caso de los cuadricópteros, al disponer de varios rotores, la solución es que el giro de las hélices de una misma extremidad sea opuesto al giro de las de la otra extremidad, de forma que las torsiones se anulen. Este concepto se tiene en cuenta a la hora de que el robot gire sobre su eje z. Para provocar un giro en sentido horario será preciso aumentar la potencia en los rotores con el sentido contrario, al mismo tiempo que se reduce proporcionalmente la potencia de los otros

Figura 1.7: Fuerza de sustentación.

dos para que la fuerza de sustentación siga constante, ya que en caso contrario el robot se desplazaría en su eje z.

El movimiento hacia delante / atrás o hacia la derecha / izquierda se consigue disminuyendo la potencia de los rotores que estén en el lado hacia el cual se deba desplazar y aumentando los del lado contrario en igual proporción si el *drone* debe permanecer a una altura fija. Es decir, para movernos hacia la derecha habrá que disminuir la potencia de los rotores derechos y aumentar la de los izquierdos, de forma que el *drone* se incline hacia la derecha y la fuerza de sustentación tenga una componente horizontal no nula, aunque la dirección de esta fuerza tenga una dirección ligeramente inclinada con respecto a la de la fuerza de gravedad.



(a) Giro a la derecha. (b) Movimiento hacia delante.

Figura 1.8: Distintas configuraciones de los motores del cuadricóptero para desplazarse.

1.3. Robótica aérea en la Universidad Rey Juan Carlos

Actualmente existen varios proyectos de robótica aérea en la URJC en ámbitos como visión, autolocalización y navegación, los cuales son el antecedente directo de este proyecto de fin de carrera. Se han diseñado numerosos componentes robóticos como drivers, algoritmos de comportamiento y soporte para simulaciones, trabajando con robots como NAO, ArDrone, FX-61 Phantom, Pioneer y Kobuki, entre otros. También se han construido robots propios, sin centrarse solo en la parte software.

Entre estos trabajos encontramos el Proyecto Fin de Carrera de Óscar Higuera [14], en el cual se desarrolló tanto el hardware como el software del cuadricóptero X-Pider[1], que cuenta con las caracterísiticas más comunes de los cuadricópteros. Es capaz de llevar a cabo vuelos estables, interpretar los datos de sus sensores, procesarlos y mantener su rumbo, así como comunicarse con una estación de monitorización y conrol.

Más recientemente, cabe destacar el trabajo de fin de grado de Alberto Martín [13], que consistió en crear el componente `ardrone_server`. En él se dio soporte JdeRobot a la versión 1.0 del Parrot ArDrone (figura 1.6) permitiendo el acceso a sus sensores y actuadores a aplicaciones JdeRobot. Dentro de este trabajo se incluye también una herramienta para controlar al UAV remotamente y leer la información de sus sensores, el componente `uav_viewer`.

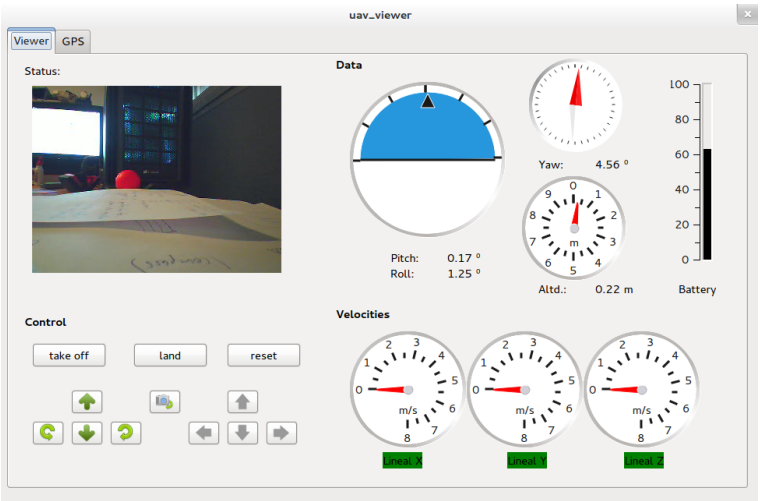


Figura 1.9: Interfaz gráfica del componente `uav_viewer`

El proyecto de Alberto Martín incluía también un algoritmo de visión y navegación autónoma dentro del componente `object_tracking`. Este algoritmo permite al cuadricóptero seguir objetos en tres dimensiones utilizando como fuente de información las cámaras frontal y ventral. Si se usa la cámara frontal el *drone* actúa moviéndose adelante, atrás, hacia los lados, arriba o abajo en función del movimiento del objeto al que sigue, procurando mantenerlo siempre delante y a un rango determinado de distancias. En el caso de la cámara ventral utiliza el sensor de altitud para mantener la altura y posicionarse sobre el cuerpo. En este caso puede ayudarse de otro objeto cercano al principal que le indique la orientación. Requiere para funcionar la ayuda de ficheros de configuración que indican el color sobre el que aplicar el filtro para detectar el objeto a seguir. Los movimientos se efectúan en función de la desviación del objeto detectado con respecto al centro de la imagen y del área de la detección en el caso del seguimiento frontal.

1.4. Simuladores en robótica

Los simuladores en robótica son usados para crear mundos virtuales y observar cómo un robot simulado actúa en dicho entorno. De esta forma pueden probarse aplicaciones robóticas sin depender de un robot físico, haciendo que las pruebas sean más baratas. Otra ventaja es que si el robot se choca o tiene un

[1]Xpider: Design and Development of a Low Cost VTOL UAV Platform

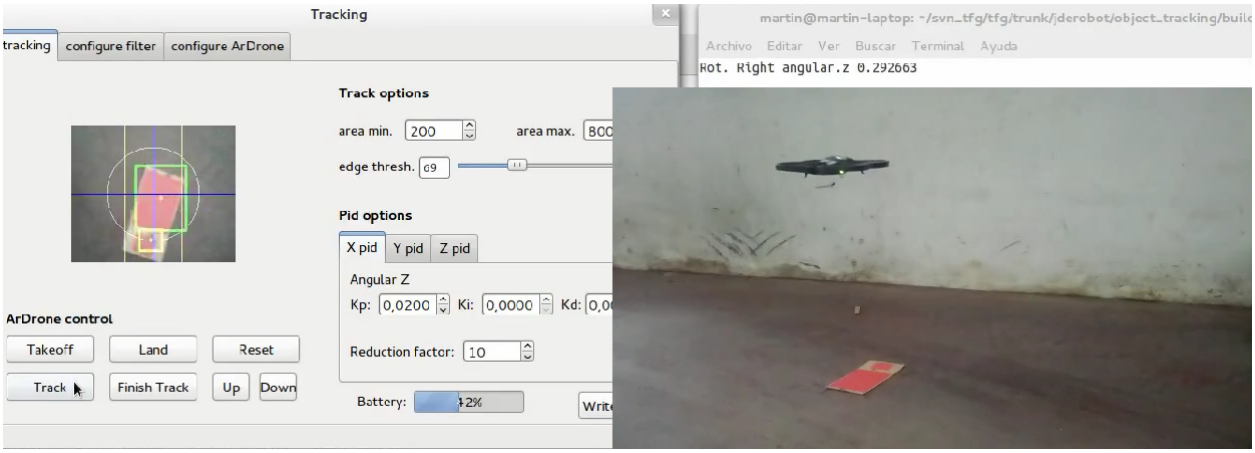


Figura 1.10: Componente `object_tracking` funcionando en un ArDrone.

comportamiento extraño que no se había previsto es posible reiniciar la simulación sin que el robot real (o las personas cercanas) haya sufrido daños. Tampoco es necesario instalar el programa en el robot real indefinidas veces, llevándose a cabo todo el proceso en un ordenador. La depuración es más cómoda y rápida con la simulación, puesto que muchos robots requieren cargar sus baterías para funcionar, lo cual puede llevar bastante tiempo.

Existen, no obstante, algunos inconvenientes, como el hecho de que los datos que aportan los sensores y las interacciones con el mundo virtual no son idénticos a sus equivalentes en el mundo real. Así mismo, es posible que en el mundo real existan factores que no estén presentes en la simulación, provocando un comportamiento inesperado. Aún así es muy recomendable valerse de simulaciones en las primeras fases del desarrollo de aplicaciones robóticas. Una vez que se alcanzan resultados virtualmente satisfactorios se puede trabajar con robots reales y perfeccionar los algoritmos con un menor riesgo.

Algunos de estos simuladores representan los mundos en 3D y recrean la física de estos (gravedad, colisiones...) permitendo visualizar el movimiento del robot en escenarios muy realistas. Motores de física comunes son ODE (*Open Dynamics Engine*) y PhysX.

Un ejemplo es *Microsoft Visual Simulation Environment*, incluido en el entorno de desarrollo *Microsoft Robotics Developer Studio*[2]. Se trata de un simulador basado en Windows y que utiliza tecnología NVIDIA y un motor PhysX. El entorno proporciona otras herramientas como un entorno de programación visual o un servicio de monitorización de sensores utilizando un navegador web.

Otro ejemplo es Gazebo[3], un simulador muy completo que se distribuye como software libre. Cuenta con modelos de robots que pueden usarse directamente, además de incluir la posibilidad de que el usuario cree su propio robot o entornos, como pueden ser un campo o el interior de un edificio, incluyendo texturas, luces y sombras. Simula la física de los cuerpos rígidos: choques, empujes, gravedad, etc. Dispone de una amplia gama de sensores como cámaras, láseres, sensores de contacto, IMU, etc. Los algoritmos pueden aplicarse a los modelos cargando *plugins*, o librerías dinámicas. Muchos de estos modelos y plugins han sido creados por la organización OSRF (*Open Source Robotics Foundation*), que desarrolla y distribuye software libre para su uso en investigación robótica.

[2] `https://msdn.microsoft.com/en-us/library/dd939239.aspx`
[3] `www.gazebosim.org`

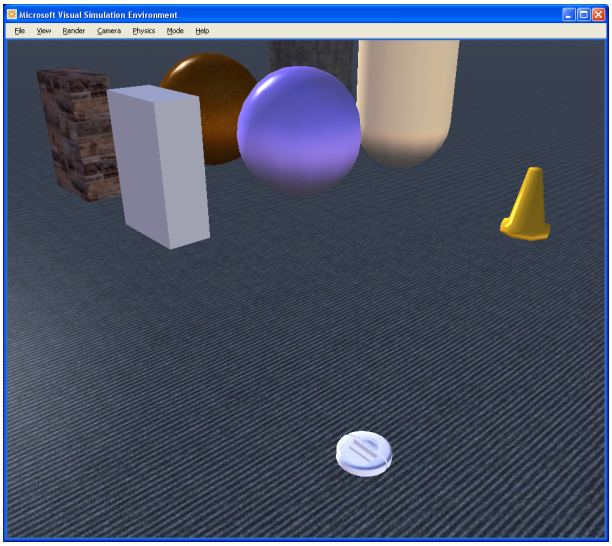


Figura 1.11: Simulador *Microsoft Visual Simulation Environment* con un robot *Roomba*.

Gazebo fue diseñado por Andrew Howard y Nate Koenig en 2002 [9]. Empezó como parte del proyecto Player/Stage. Stage es otro simulador orientado a altas poblaciones de robots pero con menor fidelidad, mientras que Gazebo está orientado a simular pocos robots con gran fidelidad [8]. Gazebo se usa en la competición DRC (*DARPA Robotics Challenge*), un torneo organizado por la agencia del Departamento de Defensa de Estados Unidos DARPA (*Defense Advanced Research Project Agency*) en el que los participantes deben desarrollar programas que permitan a determinados robots superar las pruebas. Esta agencia invirtió diez millones de dólares en la OSRF para el mantenimiento y desarrollo de este simulador de forma que se han mejorado sus capacidades en gran medida en los últimos años.

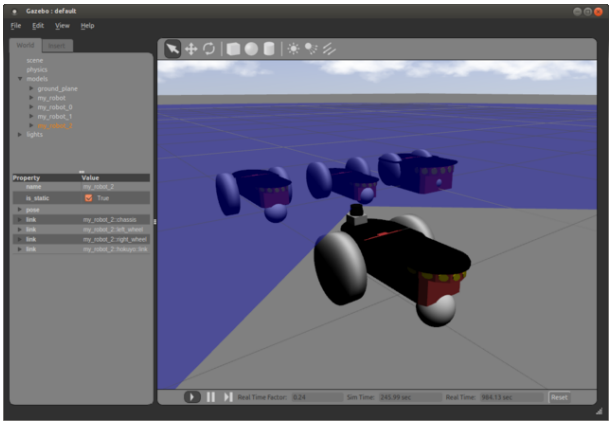


Figura 1.12: Robots *Pioneer* simulados en Gazebo.

Una vez situados en el contexto del proyecto explicaremos cuál es el problema y cómo se ha abordado. En los siguientes capítulos se listarán los objetivos propuestos, se analizarán las herramientas en las que nos hemos apoyado y explicaremos el desarrollo de la solución y por qué se ha diseñado de esa manera. Finalmente se presentarán las conclusiones a las que hemos llegado, las líneas de investigación que se abren y si los resultados obtenidos son satisfactorios.

Capítulo 2

Objetivos

2.1. Problema a abordar

El objetivo general a tratar en este Proyecto Fin de Carrera es dar un *soporte en la plataforma software JdeRobot para cuadricópteros en Gazebo*. Para ello hemos definido tres subobjetivos que explicamos a continuación.

1. Desarrollo de *drivers*: Este subobjetivo consta a su vez de tres puntos diferenciados: (a) Materializar en Gazebo la simulación de un cuadricóptero similar al ArDrone de Parrot, (b) proporcionar acceso a sensores simulados y (c) proporcionar acceso a actuadores simulados.

   Para ello, se diseñarán los programas, clases y funciones necesarios para encapsular la interfaz de Gazebo en componentes JdeRobot que ofrezcan una interfaz de más alto nivel a otros componentes. Deben ser compatibles con los *drivers* para el cuadricóptero real. Los componentes que se encarguen de la parte lógica del robot, es decir, de darle una funcionalidad particular, tendrán la capacidad de conectarse a estos *drivers* de simulación para enviar las instrucciones oportunas con total transparencia a si el modelo que las reciba es real o simulado, es decir, empleando la misma interfaz.

2. Desarrollo de componentes de navegación: En total se programarán cinco aplicaciones de navegación autónoma para *drones*. Los objetivos que cada una persigue son:

   - Seguimiento de balizas por posición: Navegación utilizando la información de los sensores de posición y orientación para alcanzar un conjunto ordenado de balizas dadas sus coordenadas en el espacio.
   - Seguimiento de carretera: Programar al cuadricóptero para seguir una carretera utilizando la información obtenida de la cámara ventral.
   - Búsqueda de objetos: Exploración de un área de búsqueda dada y localizar por medio de la cámara ventral un tipo concreto de objetos.
   - Seguimiento de otro *drone*: Localizar por medio de la cámara frontal a otro cuadricóptero y seguirlo durante un tiempo determinado. Este objetivo requiere dos aplicaciones: La que permitirá a un cuadricóptero desplazarse por un camino conocido y la que proporcionará al otro la capacidad de localizar al primero y seguirlo.

3. Validación experimental: Para comprobar el buen funcionamiento de esta infraestructura se probarán algoritmos previamente diseñados en JdeRobot y que han controlado modelos reales con éxito, así como otros nuevos desarrollados en este proyecto. En concreto los componentes `uav_viewer` y `object_tracking`, que permiten teleoperar al *drone* y hacer que siga a un objeto, respectivamente. Estos deberán recibir la misma información de los sensores, dar las mismas órdenes al robot simulado y hacer que se comporte de manera similar al robot real.

   Será necesario también validar experimentalmente en el simulador las aplicaciones desarrolladas. Se realizarán los ajustes precisos para que los componentes provean al cuadricóptero simulado de la inteligencia deseada en la mayor cantidad de casos posibles y considerando los factores oportunos.

2.2. Requisitos

Además de alcanzar los objetivos planteados las soluciones desarrolladas deben satisfacer estos requisitos:

1. El entorno de desarrollo para los drivers será JdeRobot. Se hará uso de los componentes existentes en JdeRobot que puedan ser de utilidad, como por ejemplo el componente `camera_dump`.
2. Compatibilidad con las interfaces que hayan sido definidas con anterioridad, como las del componente `ardrone_server`, y con el robot real.
3. El comportamiento de los modelos simulados deberá ser lo más parecido posible sus homólogos reales al conectarlos con otros componentes, como `uav_viewer` u `object_tacking`. Por tanto, la física de vuelo deberá estar simulada de manera realista.
4. Los componentes desarrollados deberán ser computacionalmente eficientes.

2.3. Metodología y plan de trabajo

La realización de un proyecto requiere de una metodología que establezca las pautas a seguir y la planificación de las tareas que se deben llevar a cabo para cumplir los objetivos del mismo. Hemos escogido el modelo de *desarrollo en espiral*, ya que es un modelo ampliamente usado en la ingeniería de *software*. Este modelo define una serie de ciclos que se repiten en un bucle hasta el final del proyecto, dividiéndolo en varias subtareas más sencillas y estableciendo puntos de control al final de cada iteración en los que se evalúa el trabajo realizado y se enfocan las nuevas tareas para continuar.

Esta metodología recibe su nombre por la forma de espiral que tiene su representación gráfica o diagrama de flujo, que podemos ver en la figura 2.1. En cada iteración se llevan a cabo las siguientes actividades:

- **Determinar los objetivos**, dividir en subobjetivos y fijar requisitos.
- **Analizar los riesgos** y factores que impidan o dificulten el trabajo y las consecuencias negativas que este pueda ocasionar.
- **Desarrollar** las tareas para lograr los objetivos según los requisitos especificados.

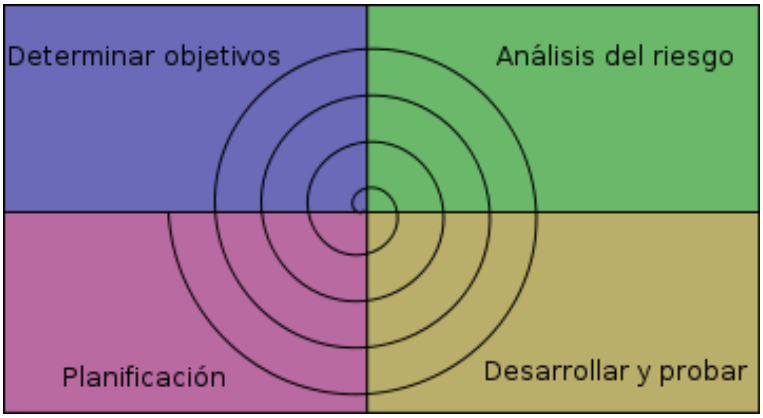


Figura 2.1: Representación gráfica del desarrollo en espiral.

- **Planificar** las próximas fases tras evaluar el transcurso del proyecto.

Durante el ciclo de vida del proyecto se han llevado a cabo reuniones semanales con el tutor. En ellas se evaluaban las tareas realizadas y se marcaba qué dirección tomar para el siguiente desarrollo. Si los puntos marcados en la anterior reunión no se habían alcanzado se ampliaba el plazo o se discutían otras vías para avanzar. En caso contrario se proponían nuevos subobjetivos.

Para facilitar el seguimiento nos hemos valido de Subversion[1], una herramienta de control de versiones de código[2], así como del mediawiki de JdeRobot[3], en el que hemos actualizado periódicamente nuestra página con el trabajo desarrollado acompañando las explicaciones con vídeos e imágenes.

El plan de trabajo de este proyecto puede dividirse en varias etapas:

- Familiarización con el entorno Jderobot. Aquí se ha aprendido a programar con la metodología propia de la plataforma, utilizando sus bibliotecas y componentes en aplicaciones simples, que han permitido también adquirir nociones sobre interfaces gráficas y controles periódicos. También ha sido importante el estudio del lenguaje C++ y de la herramienta de comunicación ICE.
- Estudio de Gazebo y sus *plugins*. Se han estudiado algunos *plugins* existentes en JdeRobot y se ha aprendido a manejar la API de Gazebo diseñando *plugins* sencillos.
- Estudio de los *drivers* TUM. Se ha estudiado el código desarrollado por la Universidad Técnica de Munich, el entorno de programación ROS y la física que rige la movilidad de los cuadricópteros. Cobra importancia también el curso en la plataforma EDX[4] de programación de cuadricópteros simulados.
- Desarrollo de *drivers*. Esta etapa abarca la programación de controladores para los sensores y la integración de los *drivers* para cuadricópteros de la Universidad de Munich con la plataforma JdeRobot y las interfaces ICE de los *drivers* para el ArDrone real.
- Desarrollo de aplicaciones JdeRobot que hagan uso de los *drivers* diseñados y proporcionen inteligencia al UAV simulado. En este punto se han construído también varios mundos virtuales valiéndonos

[1] http://subversion.apache.org
[2] https://svn.jderobot.org/users/daniyague/pfc
[3] http://jderobot.org/Daniyague-pfc
[4] http://www.edx.org

de los modelos de objetos, como casas o coches, que proporciona la comunidad de Gazebo y de otros, como carreteras o colinas, diseñados por nosotros.

Capítulo 3

Infraestructura

En este capítulo se mostrarán las bibliotecas y los componentes software de los que se ha hecho uso para la realización de este proyecto. Estos abarcan bibliotecas para elaborar interfaces gráficas, plataformas de programación de robots, *middlewares* para facilitar la comunicación entre procesos, etc. También explicaremos con detalle los *drivers* para el ArDrone diseñados por la Universidad Técnica de Munich, cuya funcionalidad se ha estudiado y extraído para el desarrollo de este proyecto.

3.1. Herramientas para interfaces gráficas

En esta sección se describirán algunas bibliotecas para el desarrollo de interfaces gráficas de usuario, concretamente GTK+ y Qt. Estas herramientas proveen varios elementos con los que representar visualmente la información que maneja nuestro programa e interactuar con él, por ejemplo, mediante botones, imágenes, *checkboxes* y etiquetas, entre otros componentes, comunes a prácticamente todos los programas con interfaz gráfica.

3.1.1. GTK+

GTK+[1] (GIMP Tool Kit) es un *toolkit* multiplataforma que ofrece un amplio número de los elementos antes mencionados en forma de *widgets*. Aunque está escrito en C, las últimas versiones son compatibles con muchos lenguajes como Java, Perl, Ruby, Python o Lua. Principalmente es usado por el entorno de escritorio GNOME propio de GNU/Linux y Unix, pero también por el entorno Xface, así como por los navegadores Firefox y Chromium.

Aunque las interfaces gráficas de las aplicaciones usadas han sido desarrolladas principalmente en Qt, al familiarizarnos con el entorno *JdeRobot* hemos hecho uso de GTK, ya que componentes como `introrob` o `naooperator` se han programado de esta manera. Los componentes de aprendizaje diseñados en la primera fase de este proyecto también tienen interfaces GTK. La versión utilizada fue la 2.6. La figura 3.1 muestra un ejemplo de aplicación GTK.

[1] `www.gtk.org`

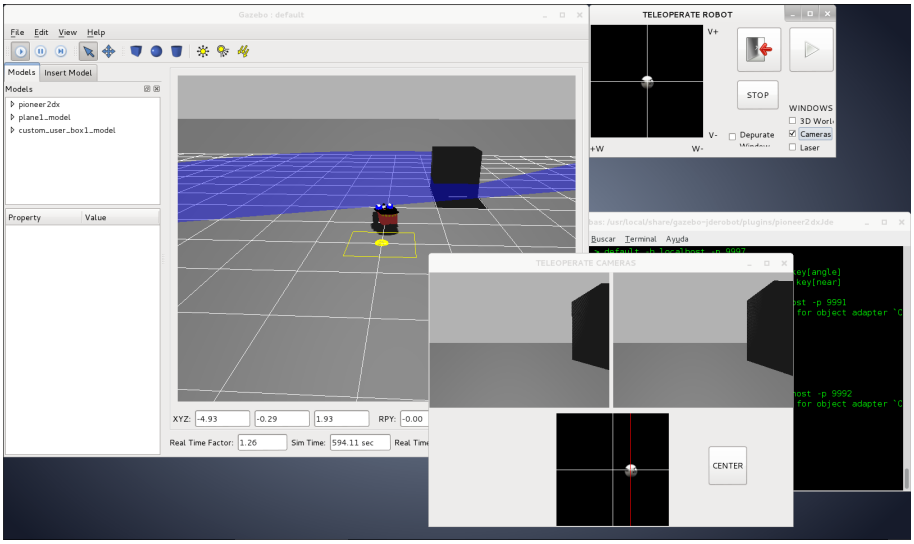


Figura 3.1: Interfaz GTK+ de la aplicación `introrob`.

3.1.2. Qt

Qt[2] es otra biblioteca multiplataforma para la creación de GUIs escrita en C++, que tiene igualmente compatibilidad con variedad de lenguajes. Está desarrollada como software libre y de código abierto, bajo licencia GPL v3 y LGPL v2.1. Fue desarrollada originalmente por la empresa Trolltech, que fue adquirida por Nokia en 2008 [18].

Con Qt es posible definir la interfaz gráfica con un fichero XML y generar a partir de este código C++ para su uso en la aplicación. Dispone de *bindings* para múltiples lenguajes de programación como Python, Java, Perl, Ruby, Lua, etc. También proporciona acceso a bases de datos SQL, análisis de ficheros XML, gestión de hilos, soporte para redes, soporte para la internacionalización y una API unificada para el manejo de ficheros multiplataforma.

Para este proyecto se ha utilizado la versión 4.8, y se ha empleado en el desarrollo de la aplicación de navegación del cuadricóptero. Además, forma parte de otros trabajos útiles desarrollados anteriormente como `uav_viewer` y `object_tracking`. Podemos observar un ejemplo en la figura 3.2.

3.2. ICE

ZeroC ICE[3] (Internet Communications Engine) es un *middleware* de *software* libre orientado al desarrollo de aplicaciones distribuidas que permite a los programadores centrarse en la lógica de la aplicación, haciendo transparentes detalles como abrir conexiones, retransmitir paquetes por la red, serialización, etc. Es compatible con lenguajes como C++, Java, Python, PHP o C#, haciendo posible que dos máquinas con procesos escritos en lenguajes distintos puedan comunicarse.

ICE ofrece mecanismos de llamadas a procedimientos remotos, tanto asíncronas como síncronas, sin necesidad de que los tipos de datos sean conocidos en tiempo de compilación, control de hilos sin necesidad

[2] `qt-project.org`
[3] `www.zeroc.com`

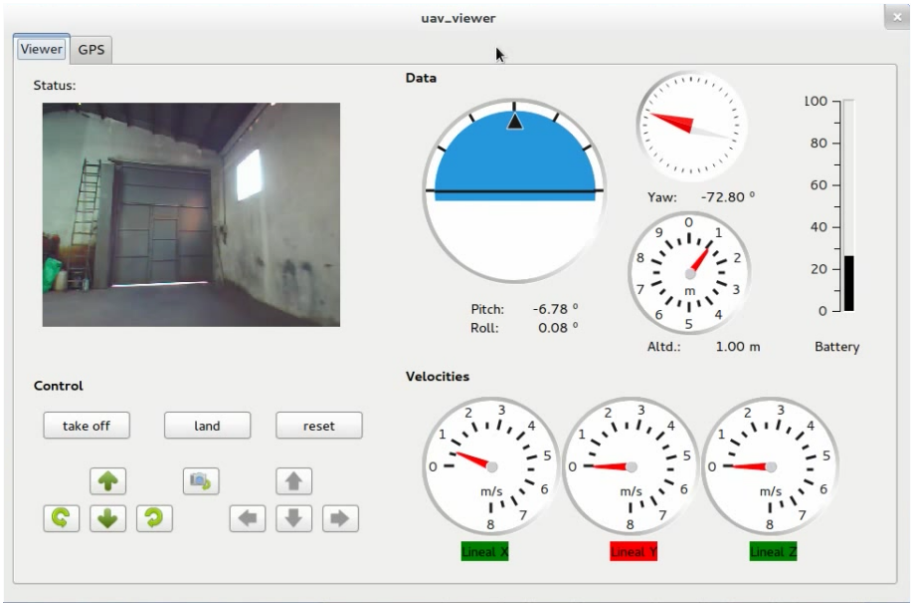


Figura 3.2: Interfaz Qt de la aplicación `uav_viewer`.

de preocuparnos por regiones críticas en cuanto a accesos a memoria, posibilidad de elegir entre TCP, UDP o SSL como protocolos de nivel de transporte, y un lenguaje propio llamado *slice* para establecer interfaces de comunicación. Con *slice* se pueden definir clases, métodos, tipos definidos por el usuario como diccionarios, secuencias o enumeraciones, herencias, excepciones, etc. Tras definir una interfaz, es necesario un "mapping" de *slice* al lenguaje propio de nuestra aplicación, que será independiente del lenguaje en el que estén escritas las aplicaciones con las que esta se vaya a comunicar.

Es interesante en el marco de este proyecto ya que es el mecanismo que utilizan algunas de las aplicaciones que lo conforman, además de que permitirá la conexión de componentes que se desarrollen en el futuro. JdeRobot utiliza ICE como infraestructura de comunicación entre sus componentes. Algunas compañías que también lo utilizan son Skype, Indra, A+W o Hewlett-Packard. La versión utilizada en este proyecto es la 3.4.

3.3. Entorno de programación JdeRobot

Prácticamente todo el desarrollo de software para robots se lleva a cabo dentro de entornos (*frameworks*), o marcos de trabajo. Estos aportan un conjunto de bibliotecas, programas y una metodología a seguir que facilita las tareas del programador. Un entorno de programación puede ser muy simple: El patrón MVC (Modelo-Vista-Controlador), a rasgos generales, no hace más que separar el acceso a los datos de la lógica de la aplicación y de la interfaz gráfica con la que interactúa el usuario. Aparte de aportar pautas de este tipo, también se pueden encargar de estructurar y ordenar la información (bibliotecas, ficheros de configuración...). ROS y *JdeRobot* son ejemplos de entornos de programación robóticos.

La plataforma dentro del cual se elabora este proyecto es *JdeRobot*[4]. Es una plataforma de código libre escrita fundamentalmente en C++ orientada a software de robótica, visión artificial y domótica, cuyo

[4] `www.jderobot.org`

entorno proporciona un conjunto de procesos independientes distribuidos llamados *componentes* que se comunican entre ellos a través del *middleware* ICE.

Una de sus funcionalidades es simplificar el acceso al hardware con una interfaz sencilla. Los componentes que se encargan de esto son los *drivers*, que proporcionan una capa por encima del software de los fabricantes, por ejemplo NAOqi en el caso del Nao, o Ar.Drone SDK en el del ArDrone. Esto hace que aplicaciones robóticas en otros componentes, posiblemente ejecutando en ordenadores o tabletas, puedan realizar lecturas de sensores con una función local, al igual que dar órdenes a los actuadores. Además, distintos modelos de un mismo tipo de robot, por ejemplo un robot aéreo, pueden incorporar esta capa de abstracción de forma que los componentes externos al robot no necesiten saber si se trata de una marca u otra, como podrían ser ArDrone y Phantom. Al ser más sencillas las llamadas a sensores y a actuadores, es más sencillo también usar *drivers* para robots simulados de modo idéntico a los *drivers* para robots reales.

JdeRobot proporciona herramientas como *drivers* para acceder a datos proporcionados por sensores (kinect, cámaras, láseres, etc), teleoperadores para varios modelos de robots, como Kobuki (TurtleBot), Pioneer y Nao, calibradores y sintonizadores para filtros y la aplicación VisualHFSM, que permite programar robots con máquinas de estado finito. Además ofrece bibliotecas como `fuzzylib`, `visionlib` y `progeo`.

Dentro de este proyecto cobran relevancia varios componentes específicos de *JdeRobot* relacionados con la robótica aérea como `ardrone_server`, `uav_viewer` y `object_tracking`, creados por Alberto Martín Florido[5].

3.3.1. Teleoperador de cuadricópteros: `uav_viewer`

Este componente proporciona una manera sencilla de controlar los actuadores del ArDrone real y de obtener datos de sus sensores más importantes, como altitud, orientación, velocidad, flujo de imágenes, etc, por medio de una interfaz gráfica que podemos observar en la figura 3.2. Cobra importancia dentro de este proyecto porque nos permitirá verificar que el cuadricóptero simulado responde a órdenes simples y que los datos sensoriales mostrados en la interfaz gráfica se ajustan a las observaciones hechas en las simulaciones.

3.3.2. Controlador del cuadricóptero real: `ardrone_server`

Como hemos dicho, `ardrone_server` es el *driver* JdeRobot del ArDrone real. Encapsula las bibliotecas ArDrone SDK ofrecidas por Parrot para controlarlo. Permite que otros componentes se conecten al robot gracias a las interfaces ICE que ofrece.

Es importante un análisis de este componente ya que una parte de este proyecto es su equivalente dentro del mundo simulado y se ha desarrollado respondiendo a sus interfaces y funcionalidades. Con tan solo cambiar la configuración de las conexiones ICE en los componentes, estos deben ser capaces de funcionar correctamente y producir en el robot simulado un efecto similar al del robot real. Para cumplir nuestros objetivos debemos prestar especial atención a las interfaces de `ardrone_server`, de modo que las implementaciones que realicemos se ajusten a las de este.

[5] `http://jderobot.org/Amartinflorido-tfg`

El soporte para el cuadricóptero real ofrece seis interfaces (figura 3.3): Tres de información sensorial, dos de actuación y una de configuración. A continuación se describirá el propósito de cada una de ellas.

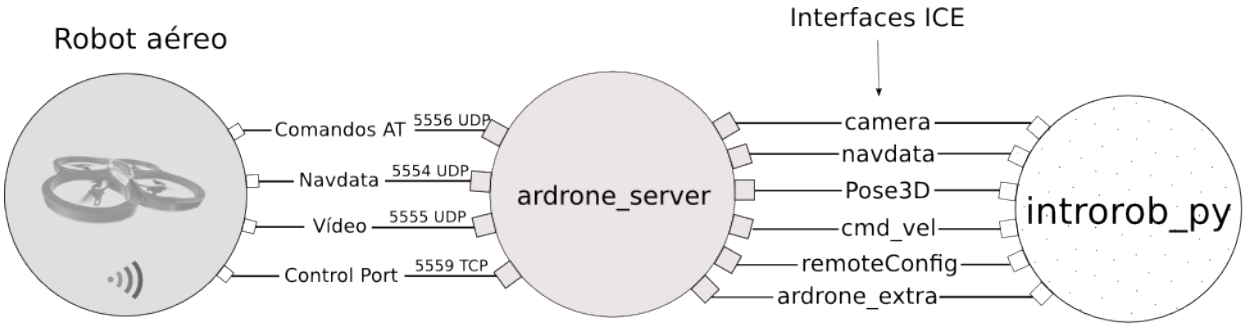


Figura 3.3: Conexiones entre los componentes `ardrone_server` e `introrob_py` mediante las interfaces ICE

Interfaz `navdata`: Sensores variados

Esta interfaz tiene la finalidad de ofrecer información de varios sensores. Permite a otras aplicaciones obtener una estructura de datos que engloba esta información. Los datos que pueden obtenerse son, entre otros, el estado del cuadricóptero, el campo magnético detectado, la velocidad del viento, los ángulos de rotación del *drone*, su velocidad y su aceleración. En el código 3.1 se encuentra la interfaz ICE `navdata`. En la estructura `NavdataData` se almacena la información de la que hemos hablado. El método `getNavdata()` permite obtener estructuras de este tipo.

```
module jderobot{
    sequence<int> arrayInt;
    sequence<float> arrayFloat;

    class NavdataData
    {
        //0-> ArDrone1, 1-> ArDrone2
        int vehicle;
        int state;
        //The remaing charge of baterry %
        float batteryPercent;
        //Magnetometer ArDrone 2.0
        int magX;
        int magY;
        int magZ;
        //Barometer ArDrone 2.0
        int pressure;
        //Temperature sensor ArDrone 2.0
        int temp;
        //Estimated wind speed ArDrone 2.0
        float windSpeed;
        float windAngle;
```

```
        float windCompAngle;
        //rotation about the X axis
        float rotX;
        //rotation about the Y axis
        float rotY;
        //rotation about the Z axis
        float rotZ;
        //Estimated altitude (mm)
        int altd;
        //linear velocity (mm/sec)
        float vx;
        //linear velocity (mm/sec)
        float vy;
        //linear velocity (mm/sec)
        float vz;
        //linear accelerations (unit: g)
        float ax;
        float ay;
        float az;
        //Tags in Vision Detectoion
        //Should be unsigned
        int tagsCount;
        arrayInt tagsType;
        arrayInt tagsXc;
        arrayInt tagsYc;
        arrayInt tagsWidth;
        arrayInt tagsHeight;
        arrayFloat tagsOrientation;
        arrayFloat tagsDistance;
        //time stamp
        float tm;
    };

    interface Navdata
    {
        idempotent NavdataData getNavdata();
    };
};
```

Código 3.1: Interfaz *slice* `navdata`.

Interfaz `pose3d`: Sensores IMU y GPS

Esta interfaz, estandarizada en JdeRobot, se corresponde con un sensor IMU y un GPS. Un sensor IMU (*Inertial Measurement Unit*, Unidad de Medición Inercial) sirve para obtener la velocidad, la orientación y las fuerzas gravitacionales de un objeto, utilizando para ello acelerómetros y giróscopos. El sensor GPS (*Global Positioning System*, Sistema de Posicionamiento Global) aporta geolocalización, es decir, latitud

y longitud. En esta interfaz, se incluye también la componente z de las coordenadas espaciales, es decir, la altitud.

La estructura de datos en la que se engloba esta información se llama `Pose3DData`. Dentro de esta podemos encontrar un vector con las coordenadas tridimensionales en las que se encuentra nuestro robot y la orientación del mismo en forma de cuaternión. Los métodos para obtener y establecer estos datos son, respectivamente, `getPose3DData()` y `setPose3DData(Pose3DData)`. Su definición está en el código 3.2:

```
module jderobot{
    /**
    * Pose3D data information
    */
  class Pose3DData
  {
        float x;
        float y;
        float z;
    float h;
        float q0;
        float q1;
        float q2;
        float q3;
  };

  /**
   * Interface to the Pose3D.
   */
  interface Pose3D
  {
    idempotent Pose3DData getPose3DData();
    int setPose3DData(Pose3DData data);
  };

}; //module
```

Código 3.2: Interfaz *slice* `pose3d`.

Interfaz `camera`: Sensor cámara

Otra interfaz implementada es `camera`, la cual sirve para ofrecer a otras aplicaciones las imágenes de las cámaras del cuadricóptero. Esta está estandarizada en JdeRobot para enviar y recibir flujos de imágenes, fotograma a fotograma. Como puede verse en el código 3.3, aquí se define una clase que representa la cámara, con atributos como el nombre, la URI desde la que se transmiten las imágenes, una pequeña descripción o las distancias focales en los ejes X e Y.

Esta interfaz implementa otra llamada `ImageProvider` dentro del fichero `image.ice`, el cual contiene otra clase que representa una imagen (código 3.4), con parámetros como la anchura, la altura, el formato,

etc. Es en esta última interfaz donde se define el método `getImageData()`, que ofrece un fotograma concreto en cada instante de tiempo.

```
module jderobot{
  /**
   * Static description of a camera
   */
  class CameraDescription
  {
    string name;
    string shortDescription;
    string streamingUri;
    float fdistx;
    float fdisty;
    float u0;
    float v0;
    float skew;
    float posx;
    float posy;
    float posz;
    float foax;
    float foay;
    float foaz;
    float roll;
  };

  /**
   * Camera interface
   */
  interface Camera extends ImageProvider
  {
    idempotent CameraDescription getCameraDescription();
    int setCameraDescription(CameraDescription description);
    string startCameraStreaming();
    void stopCameraStreaming();
  };

}; /*module*/
```

Código 3.3: Interfaz *slice* `camera`.

```
module jderobot{

  /**
   *  Static description of the image source.
   */
  class ImageDescription
  {
```

```
    int width; /**< Image width [pixels] */
    int height;/**< Image height [pixels] */
    int size;/**< Image size [bytes] */
    string format; /**< Image format string */
  };

  /**
   * A single image served as a sequence of bytes
   */
  class ImageData
  {
    Time timeStamp; /**< TimeStamp of Data */
    ImageDescription description; /**< ImageDescription of Data, for convienence purposes */
    ByteSeq pixelData; /**< The image data itself. The structure of this byte sequence
              depends on the image format and compression. */
  };

  //! Interface to the image consumer.
  interface ImageConsumer
  {
    //! Transmits the data to the consumer.
    void report( ImageData obj );
  };

  /**
   * Interface to the image provider.
   */
  interface ImageProvider
  {
    /**
     * Returns the image source description.
     */
    idempotent ImageDescription getImageDescription();

    /**
     * Returns the latest data.
     */
    ["amd"] idempotent ImageData getImageData()
      throws DataNotExistException, HardwareFailedException;
  };
}; //module
```

Código 3.4: Interfaz *slice* `image`.

Interfaz `cmdVel`: Movimiento del *drone*

Esta es una interfaz de actuación. Su objetivo es dar a otros programas la capacidad de ordenar velocidades al cuadricóptero, tanto lineales como angulares. En otras palabras, esta interfaz permite mover al *drone*.

En su definición podemos encontrar la clase que define las velocidades lineal y angular: `CMDVelData`. El método que permite ordenar velocidades al *drone* es `setCMDVelData`, cuyo parámetro es un objeto de la clase anterior. Aunque están definidas, en muchos casos las velocidades angulares con respecto a los ejes X e Y son ignoradas por los cuadricópteros, ya que estos robots no son concebidos para realizar grandes giros hacia delante o lateralmente, lo cual los desestabilizaría. No están pensados para dar "volteretas" en el aire, por ejemplo. En el código 3.5 podemos ver la definición de esta interfaz.

```
module jderobot{
    class CMDVelData
    {
        float linearX;
        float linearY;
        float linearZ;
        float angularX;
        float angularY;
        float angularZ;
    };

    interface CMDVel{
        int setCMDVelData(CMDVelData data);
    };
};
```

Código 3.5: Interfaz *slice* `cmdVel`.

Interfaz `ardrone_extra`: Maniobras básicas

En esta interfaz de actuación se definen métodos para realizar acciones más complejas como aterrizar y despegar (métodos `land` y `takeOff`), cambiar la cámara de la que se reciben las imágenes (`toggleCam`) y aportar funciones extra al *drone*, como recalibrar la estimación de la orientación (`flatTrim`) o grabar vídeos en un dispositivo extraíble USB (`recordOnUsb`). Los métodos `ledAnimation` y `flightAnimation` no están implementados en el momento de elaboración de este proyecto. La definición de la interfaz puede verse en el código 3.6.

```
module jderobot{
    interface ArDroneExtra{
        void recordOnUsb(bool record);
        void ledAnimation(int type,float duration, float req);
        void flightAnimation(int type, float duration);
        void flatTrim();
        void toggleCam();
        void land();
```

```
        void takeoff();
        void reset();
    };
};
```

Código 3.6: Interfaz *slice* `ardrone_extra`.

Interfaz `remoteConfig`: Configuración en tiempo de ejecución

La interfaz `remoteConfig` ofrece comandos para configurar "en caliente" al cuadricóptero. Por ejemplo, en un instante determinado nos puede interesar que no se sobrepase una altura determinada por peligro a chocar contra un techo, por lo que usaríamos esta interfaz para configurar el parámetro que limita la altura, y al salir a una zona despejada aumentar dicho valor puesto que no hay peligro de choque con el techo.

Con el comando `write(string, int)` se pueden añadir sucesivas líneas de texto en un fichero especificado por el parámetro *id*. Con `setConfiguration(int)` la información de dicho fichero es leída e interpretada para realizar la configuración. Entre otros, se encuentran los siguientes parámetros configurables:

- **default_camera**: Indica qué cámara esta enviando el flujo de imágenes actual: 0 para la frontal y 1 para la ventral.
- **outdoor**: Indica el dron está en modo interiores con un 0, o en modo exteriores con un 1.
- **altitude_max**: Máxima altitud a la que puede llegar el dron.
- **altitude_min**: Mínima altitud a la que puede llegar el dron.
- **euler_angle_max**: Máximo ángulo del plano del cuadricóptero con respecto al plano horizontal (roll o pitch), en radianes.
- **control_vz_max**: Velocidad máxima en el eje z, en mm/s.
- **control_yaw**: Velocidad angular máxima en torno al eje z, en rad/s.

En el código 3.7 podemos ver la definición de esta interfaz.

```
module jderobot{
    interface remoteConfig{
    int initConfiguration();
    string read(int id);
    int write (string data, int id);
    int setConfiguration(int id);
    };
};
```

Código 3.7: Interfaz *slice* `remoteConfig`.

3.4. Gazebo

Como ya se ha explicado en la introducción, Gazebo[6] es un simulador ampliamente reconocido y empleado a escala mundial. Cuenta con el respaldo de la OSRF (Open Source Robotic Foundation) y una amplia comunidad de programadores que contribuyen con su desarrollo y que lo han impulsado para su uso en la competición DARPA Robotics Challenge desde 2012[20]. En esta participan varios equipos que deben superar una serie de tareas robóticas[19]. En su sección Virtual Robotics Challenge, estos retos se realizan de manera simulada. DARPA ha invertido más de 10 millones de dólares en la OSRF para mejorar el simulador y mantenerlo.

Las simulaciones de este proyecto se han llevado a cabo en Gazebo, concretamente con la versión 1.8.1 y la 5.0.1. La API de Gazebo ofrece distintos módulos con clases que representan sensores, actuadores, robots, físicas, mensajes, modelos físicos, funciones, propiedades y útiles matemáticos como vectores o cuaterniones para desarrollar *plugins* que arranquen junto con los objetos simulados y controlen su comportamiento.

3.4.1. Arquitectura

Las bibliotecas de Gazebo, cuyo esquema puede verse en la figura 3.4, se dividen en:

- **Physics**: Ejecuta un bucle en el que se actualizan las propiedades físicas. Utiliza las bibliotecas de terceros ODE y Bullet. Permite simular choques, fricción, gravedad, inercias, etc.
- **Rendering**: Permite a la interfaz gráfica representar visualmente los componentes del mundo. Depende a su vez de las librerías de **OGRE**.
- **Sensors**: Se encarga de implementar los sensores y de la generación de sus datos.
- **Transport**: Permite la comunicación entre los componentes de Gazebo por medio de *sockets* a través de la dependencia `boost::asio`.
- **GUI**: Graphical User Interface. Utiliza Qt para crear una interfaz gráfica que permite al usuario visualizar la simulación e interactuar con ella.

3.4.2. GUI

La figura 3.5 muestra la interfaz gráfica con un mundo en el que hay tres robots. Encima de la ventana de visualización del mundo hay varios botones que permiten mover objetos, girarlos e introducir cuerpos simples como cubos, esferas y cilindros. Abajo de la ventana se muestran botones para pausar, renaudar, adelantar, retroceder o reiniciar la simulación. A la izquierda podemos ver un listado con el mundo y los modelos que contiene, en el que se pueden visualizar y cambiar sus parámetros.

[6] http://gazebosim.org/

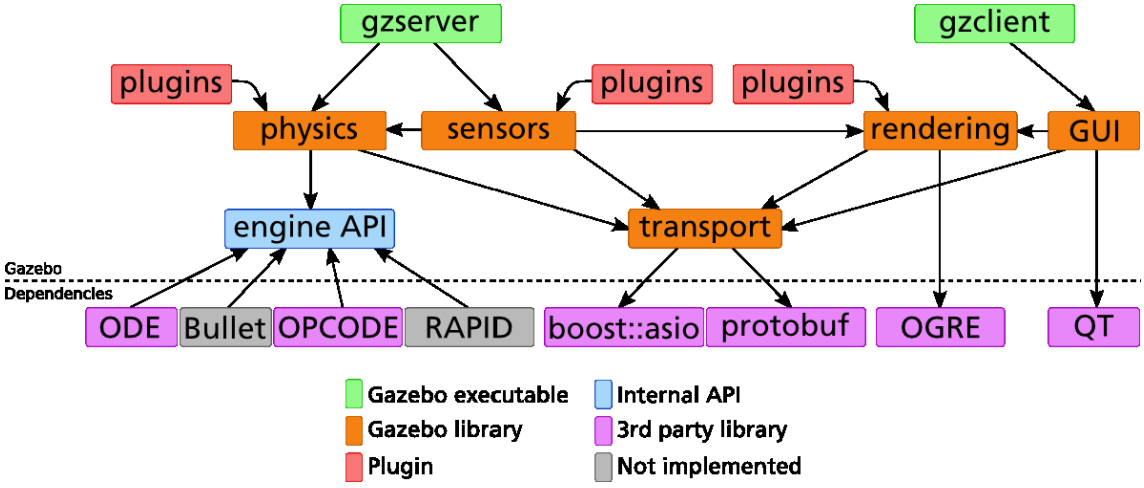


Figura 3.4: Esquema de bibliotecas y dependencias de Gazebo.

3.4.3. SDF

Los mundos simulados con Gazebo son mundos 3D. Estos se cargan a partir de ficheros con extensión ".world", que son ficheros XML definidos en lenguaje *SDF*, o Simulation Description Format. Este lenguaje, además de mundos, permite almacenar información relativa a cualquier aspecto de la simulación, por lo que lo hemos usado para describir la escena y cualquier patrón que queremos simular. La versión que hemos utilizado es la 1.4. Las *etiquetas* representan componentes de la simulación. Las principales son:

- **World**: Representa al mundo como un conjunto de propiedades físicas, modelos y *plugins*.
- **Model**: Guarda información y elementos de un modelo, que puede representar un robot u objetos del entorno. Dentro pueden encontrarse las etiquetas `link`, `joint` y `sensor`, entre otras.
- **Physics**: Permite indicar el tipo de motor físico, intervalo de tiempo al que se actualiza la simulación, la gravedad, etc.
- **Scene**: Describe efectos como luz ambiental, nubes, sombras, etc.
- **Plugin**: Indica un *plugin* para ejecutar. Esta etiqueta puede ser hija de otras como `link` o `sensor`. Deberá contener como atributo la ruta de la biblioteca dinámica deseada. Permite materializar el comportamiento del elemento al que pertenece.

3.4.4. Plugins

Esta sección tiene especial importancia, ya que gran parte de este proyecto consiste en el desarrollo de un soporte para robots aéreos en Gazebo en forma de *plugins*. Estos son bibliotecas dinámicas que se cargan en tiempo de ejecución. Los *plugins* dan una funcionalidad a los objetos del mundo simulado. Son necesarios para dar inteligencia a robots simulados, ya sea por medio de ellos mismos u ofreciendo la información de sus sensores a aplicaciones externas y recibiendo de ellas comandos para los actuadores simulados. Se comportan en este sentido como *drivers*. Tienen acceso a toda la funcionalidad de Gazebo a partir de las clases C++ que este ofrece.

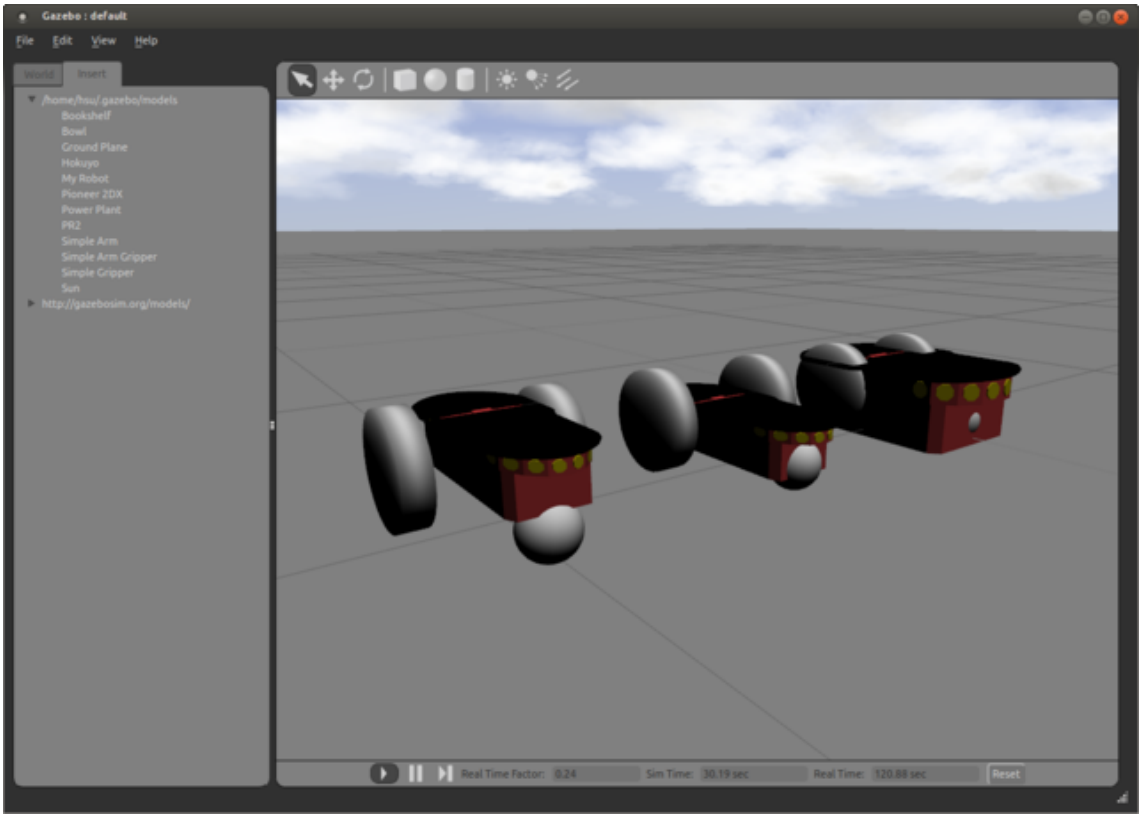


Figura 3.5: Interfaz gráfica de Gazebo.

Existen cuatro tipos: World Plugins, Model Plugins y Sensor Plugins, que controlan, respectivamente, un mundo, un modelo o un sensor, junto con el tipo System Plugin, que se indica por medio de la línea de comandos y permite controlar el proceso de inicio de Gazebo. Para programar un *plugin*, es necesario que las clases creadas hereden de `WorldPlugin`, `ModelPlugin` o `SensorPlugin` respectivamente[15].

El único método obligatorio es `Load`, que es el primero que Gazebo ejecuta al cargarse la librería dinámica. En él debe indicarse el código que queremos que se ejecute iterativamente hasta el final de la simulación por medio del método `ConnectWorldUpdateBegin`. A esta función periódica la llamaremos `OnUpdate`. Otro método útil que definir en un *plugin* es `Init`. Este se ejecutará despues que `Load` y antes que la primera iteración de `OnUpdate` y tiene la función de inicializar los campos que necesiten un valor inicial.

El *plugin* debe registrarse en el simulador por medio de una *macro*. La instrucción necesaria es distinta para cada tipo: `GZ_REGISTER_WORLD_PLUGIN`, `GZ_REGISTER_MODEL_PLUGIN`, `GZ_REGISTER_SENSOR_PLUGIN` y `GZ_REGISTER_SYSTEM_PLUGIN`. El parámetro necesario para la macro es el nombre de la clase que implemente el *plugin*.

3.4.5. Drivers TUM Simulator para el cuadricóptero

Para la realización de parte de este proyecto nos hemos apoyado en los *drivers* para cuadricópteros en Gazebo diseñados en la Universidad Técnica de Munich[17], que implementan un control realista optimizado para la versión 2.0 del ArDrone, aunque puede simular la 1.0 también, junto con un controlador que

cuenta con la posibilidad de manejar el cuadricóptero simulado con un *joystick*. Hemos extraído la funcionalidad de este *software* cambiando la arquitectura de ROS por la del entorno *JdeRobot*, añadiendo las interfaces ICE necesarias para comunicarse con otros componentes y modificando las partes convenientes.

Estos *plugins* permiten dar órdenes sencillas al *drone*, como girar hacia los lados, desplazarse hacia arriba, abajo, derecha e izquierda, despegar y aterrizar. También implementan la parte sensorial, como las cámaras, barómetro, GPS, IMU, sónar y magnetómetro. Vamos a explicar el mecanismo de estos controladores y las funciones *callback* más importantes, pero antes vamos a explicar una serie de conceptos que permitan comprender totalmente las posteriores secciones.

Conceptos: Sistemas de referencia, cuaterniones y controladores PID

Es importante hacer hincapié en algunos aspectos referentes a los *sistemas de coordenadas* en una simulación. Gazebo tiene su propio sistema cartesiano, el cual no es el mismo que el de los modelos cargados. Cuando un objeto experimenta un cambio en su orientación, este seguirá "viendo" sus ejes de igual manera. Por ejemplo, nuestra derecha sigue en la misma dirección relativa por mucho que nos movamos, sin embargo es probable que el norte o el oeste hayan cambiado de dirección en referencia a nuestro sistema delante-atrás-derecha-izquierda. En Gazebo pasa lo mismo. Un modelo interpreta su movimiento como si fuera del resto del mundo. Habrá que tener en cuenta qué sistema de referencia utilizar al trabajar con vectores en una simulación. En la figura 3.6 se pueden ver las diferencias entre estos dos sistemas.

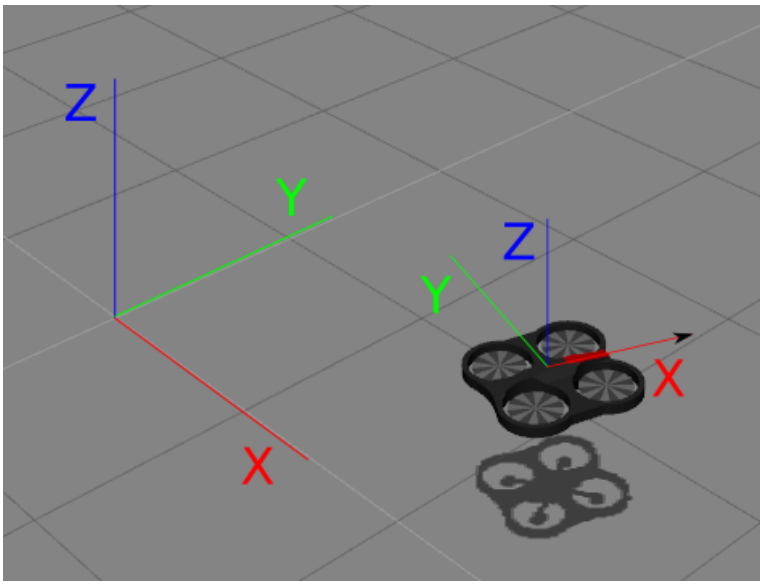


Figura 3.6: Izquierda: Sistema de coordenadas de Gazebo. Derecha: Sistema de coordenadas del robot simulado.

También es conveniente hablar de los *cuaterniones*. Consisten una extensión de los números complejos, a los que añaden otras dos componentes imaginarias. Tienen definidas una serie de operaciones y propiedades cuya explicación no entra en los objetivos de este trabajo. Bastará con exponer brevemente los detalles que nos interesan. En términos generales, pueden verse como un conjunto de cuatro componentes que representan un eje de rotación y un ángulo de giro sobre dicho eje, como puede verse en la ecuación 3.1 y en la figura 3.7, donde α es el ángulo y β_x, β_y y β_z son los ángulos del eje de rotación con respecto a los

ejes x, y y z cartesianos. Nos serán de gran utilidad cuando trabajemos con vectores a la hora de calcular fuerzas, así como para representar orientaciones de objetos en 3D con bajo coste computacional.

$$\begin{aligned} q_0 &= \cos(\alpha/2) \\ q_1 &= \sin(\alpha/2)\cos(\beta_x) \\ q_2 &= \sin(\alpha/2)\cos(\beta_y) \\ q_3 &= \sin(\alpha/2)\cos(\beta_z) \end{aligned} \tag{3.1}$$

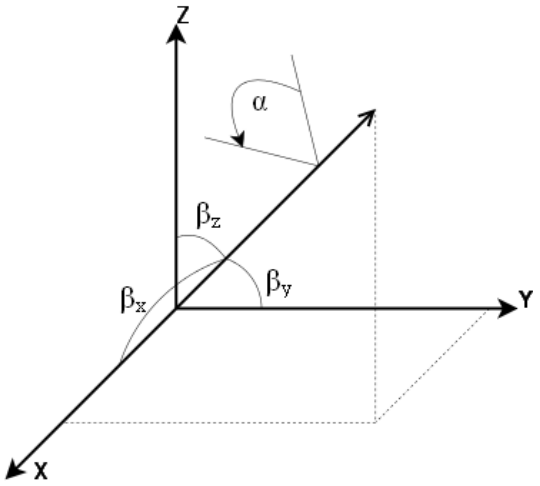


Figura 3.7: Ángulos que definen el eje del cuaternión (β_x, β_y y β_z) y el ángulo de rotación en torno a él (α).

Ahora vamos a explicar en qué consiste un *controlador PID*, o controlador Proporcional Integral Derivativo. Se trata un sistema de control que calcula el error de una magnitud a la entrada con respecto a un valor deseado y da un valor de salida en función de ese error. Consta de tres subsistemas cuyas salidas se suman para dar un resultado final. El control proporcional calcula el error y lo multiplica por una constante, ejerciendo una acción mayor cuanto más grande sea el error. El derivativo devuelve el producto de otra constante por la derivada del error, suavizando la respuesta si el error está disminuyendo. Es posible que haya un pequeño *offset* entre la magnitud de referencia y la de salida y que el control proporcional derivativo apenas ejerza control, pudiendo ocasionar un mal funcionamiento del sistema si ese error se mantiene durante bastante tiempo. Es por esto que se utiliza el control integral multiplicando la suma del error acumulado por una constante. La ecuación 3.2 muestra la salida general *s(t)* de un controlador PID, siendo *e(t)* el error y K_p, K_i y K_d las constantes proporcional, integral y derivativa, respectivamente.

$$s(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{3.2}$$

Para finalizar este apartado, vamos a introducir algunas definiciones. Hablaremos del *plano del cuadricóptero* para referirnos al plano definido por las dos extremidades del *drone*. Este plano es normalmente horizontal cuando el robot está en reposo o cernido en un punto, y se inclina unos grados cada vez que se mueve en direcciones horizontales. No debe confundirse pues con un plano horizontal. El *eje de inclinación* hará referencia al eje cuya dirección y sentido son los del vector normal al plano del cuadricóptero. Teniendo en cuenta lo explicado, se puede intuir que este eje corresponderá con un eje vertical (eje z en Gazebo) salvo en caso de los movimientos comentados antes.

Estructura y funcionamiento de los controladores

Estos *plugins* proporcionan una capa de abstracción de alto nivel. Han sido programados para aceptar órdenes como despegar o aterrizar y comandos de velocidad, y aportan mecanismos para transformar dichas órdenes en fuerzas y pares motor que el motor de físicas de Gazebo es capaz de asimilar para producir su efecto en el UAV. Este paso de órdenes a magnitudes físicas vectoriales se realiza mediante controladores PID implementados dentro de los *plugins*.

Están estructurados en el entorno ROS[7] (por sus siglas en inglés *Robot Operating System*). ROS proporciona funcionalidades similares a un sistema operativo, como son abstracciones de hardware o el intercambio de mensajes entre procesos. La organización en ROS se lleva a cabo mediante *paquetes*, los cuales contienen procesos, bibliotecas, archivos de configuración y demás recursos que proporcionen una utilidad común. Los sistemas de control se basan en una serie de procesos llamados nodos, que están distribuidos en varias máquinas y que interactúan mediante *topics*, servicios o *parameter servers*. La red del sistema se llama *grafo*.

En este caso, para comunicar cada nodo se emplea el sistema de *topics*. En un `topic`, o tema, se publica un tipo determinado de estructura de datos, o mensaje. Un nodo que envía mensajes se denomina `publisher`, siendo dichos mensajes recibidos por todos los nodos que estén subscritos al topic en cuestión. Los nodos receptores se denominan `subscribers`, y son susceptibles de ejecutar una función para cada `topic` al que están subscritos una vez que se publica un mensaje en ellos. En la figura 3.8 puede verse una representación de relaciones via *topics*, representados por flechas, que parten de los *publishers* y llegan a los *subscribers*.

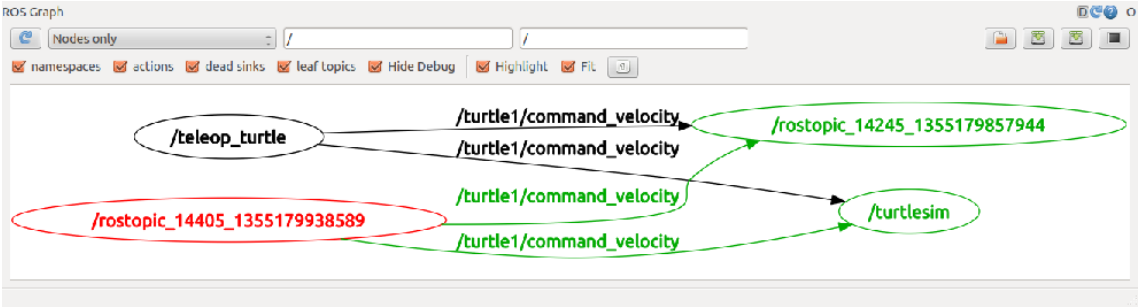


Figura 3.8: Interacciones entre nodos usando topics.

Aquí se corresponde cada nodo con un sensor, con el controlador de los motores y con el controlador del estado, así como con el controlador *joystick* y la parte que atiende sus comandos. En función de los mensajes que envíen los sensores o el usuario y del estado del robot (vuelo, aterrizando, inicializando...), los actuadores llevarán a cabo una u otra acción.

La clase `GazeboQuadrotorSimpleController` se encarga de actualizar las fuerzas y torsiones del cuadricóptero. La función `VelocityCallback`, en el código 3.8, se encarga de modificar la velocidad objetivo cada vez que un *publisher* la modifica en el *topic* correspondiente. A esta variable se le añade ruido aleatorio para dar más realismo a la simulación. Para cambiar el estado del *drone* (despegando, vuelo, aterrizado...) se utiliza la función `NavdataCallback`, que recibe como parámetro una estructura de datos que contiene el estado.

[7] `www.ros.org`

```
void GazeboQuadrotorSimpleController::VelocityCallback(
        const geometry_msgs::TwistConstPtr& velocity) {

  velocity_command_ = *velocity;

  static common::Time last_sim_time = world->GetSimTime();
  static double time_counter_for_drift_noise = 0;
  static double drift_noise[4] = {0.0, 0.0, 0.0, 0.0};
  // Get simulator time
  common::Time cur_sim_time = world->GetSimTime();
  double dt = (cur_sim_time - last_sim_time).Double();
  // save last time stamp
  last_sim_time = cur_sim_time;

  // generate noise
  if(time_counter_for_drift_noise > motion_drift_noise_time_) {
    drift_noise[0] = 2*motion_drift_noise_*(drand48()-0.5);
    drift_noise[1] = 2*motion_drift_noise_*(drand48()-0.5);
    drift_noise[2] = 2*motion_drift_noise_*(drand48()-0.5);
    drift_noise[3] = 2*motion_drift_noise_*(drand48()-0.5);
    time_counter_for_drift_noise = 0.0;
  }
  time_counter_for_drift_noise += dt;

  velocity_command_.linear.x += drift_noise[0] + 2*motion_small_noise_*(drand48()-0.5);
  velocity_command_.linear.y += drift_noise[1] + 2*motion_small_noise_*(drand48()-0.5);
  velocity_command_.linear.z += drift_noise[2] + 2*motion_small_noise_*(drand48()-0.5);
  velocity_command_.angular.z += drift_noise[3] + 2*motion_small_noise_*(drand48()-0.5);
}
```

Código 3.8: Método `VelocityCallBack` de `GazeboQuadrotorSimpleController`.

En el método `Update` de esta misma clase se aplican las fuerzas al robot según el estado y la velocidad ordenada. Este método será ejecutado periódicamente por Gazebo en cada ciclo de la simulación mientras el robot permanezca en ella. Después de calcular estas fuerzas, se invocará a los métodos `AddRelativeForce` y `AddRelativeTorque` sobre el miembro dato `link`, tomando como parámetro la fuerza y la torsión que se desea aplicar, respectivamente. El comienzo de esta función se muestra en el código 3.9.

```
void GazeboQuadrotorSimpleController::Update() {

  math::Vector3 force, torque;

  // Get new commands/state
  callback_queue_.callAvailable();

  // Get simulator time
  common::Time sim_time = world->GetSimTime();
```

```
  double dt = (sim_time - last_time).Double();
  if (dt == 0.0) return;

  // Get Pose/Orientation from Gazebo (if no state subscriber is active)
  if (imu_topic_.empty()) {
    pose = link->GetWorldPose();
    angular_velocity = link->GetWorldAngularVel();
    euler = pose.rot.GetAsEuler();
  }
  if (state_topic_.empty()) {
    acceleration = (link->GetWorldLinearVel() - velocity) / dt;
    velocity = link->GetWorldLinearVel();
  }
    ...
  // Get gravity
  math::Vector3 gravity_body = pose.rot.RotateVector(
    world->GetPhysicsEngine()->GetGravity());
  double gravity = gravity_body.GetLength();
  double load_factor = gravity * gravity /
    world->GetPhysicsEngine()->GetGravity().GetDotProd(gravity_body);
    ...
```

Código 3.9: Comienzo del método `Update` de `GazeboQuadrotorSimpleController`.

Para calcular las fuerzas que se aplicarán al *drone*, primero se obtienen a partir de `link` las velocidades lineal y angular, aceleración, posición y orientación. A continuación se llama al método `GetPhysicsEngine` sobre `world` para obtener el motor físico, y sobre este, se llama a `GetGravity` para obtener el vector de la fuerza de gravedad y, con este, el coeficiente `load_factor`, que indicará la inclinación del robot con respecto al plano *xy*, tomando el valor 1 cuando este y el plano del cuadricóptero sean paralelos y aumentando hasta infinito cuando el plano del cuadricóptero esté inclinado 90 grados.

También necesitaremos varios controladores PID. En el código 3.10 se implementa clase `PIDController`, que es una subclase de `GazeboQuadrotorSimpleController`. El método `Load` de esta clase tiene como objetivo leer del fichero SDF las constantes correspondientes. Estas pueden ser diferentes para cada controlador, pudiendo valer cero para alguno de los parámetros si no es necesario. Para calcular la salida de los PID se usará el método `update`, que recibe como argumentos la salida deseada, la magnitud actual, la derivada de dicha magnitud y el intervalo de tiempo transcurrido. Más adelante explicaremos qué entradas se proporcionan a cada controlador y qué significan sus salidas.

```
void GazeboQuadrotorSimpleController::PIDController::Load(
          sdf::ElementPtr _sdf, const std::string& prefix)
{
  gain_p = 0.0;
  gain_d = 0.0;
  gain_i = 0.0;
  time_constant = 0.0;
  limit = -1.0;
```

```
  if (!_sdf) return;
  // _sdf->PrintDescription(_sdf->GetName());
  if (_sdf->HasElement(prefix + "ProportionalGain"))
    gain_p = _sdf->GetElement(prefix + "ProportionalGain")->GetValueDouble();
  if (_sdf->HasElement(prefix + "DifferentialGain"))
    gain_d = _sdf->GetElement(prefix + "DifferentialGain")->GetValueDouble();
  if (_sdf->HasElement(prefix + "IntegralGain"))
    gain_i = _sdf->GetElement(prefix + "IntegralGain")->GetValueDouble();
  if (_sdf->HasElement(prefix + "TimeConstant"))
    time_constant = _sdf->GetElement(prefix + "TimeConstant")->GetValueDouble();
  if (_sdf->HasElement(prefix + "Limit"))
    limit = _sdf->GetElement(prefix + "Limit")->GetValueDouble();

}

double GazeboQuadrotorSimpleController::PIDController::update(
          double new_input, double x, double dx, double dt)
{
  // limit command
  if (limit > 0.0 && fabs(new_input) > limit)
    new_input = (new_input < 0 ? -1.0 : 1.0) * limit;

  // filter command
  if (dt + time_constant > 0.0) {
    dinput = (new_input - input) / (dt + time_constant);
    input  = (dt * new_input + time_constant * input) / (dt + time_constant);
  }

  // update proportional, differential and integral errors
  p = input - x;
  d = dinput - dx;
  i = i + dt * p;

  // update control output
  output = gain_p * p + gain_d * d + gain_i * i;
  return output;
}

void GazeboQuadrotorSimpleController::PIDController::reset()
{
  input = dinput = 0;
  p = i = d = output = 0;
}
```

Código 3.10: Implementación de la clase `PIDController`.

Con esta información estamos listos para obtener los dos vectores que provocarán los movimientos en el cuadricóptero: las variables `force` y `torque`. A partir de ahora se explicarán por separado los algoritmos

para realizar cada tipo de movimiento.

Movimiento rotatorio

El movimiento en torno al eje de inclinación dependerá únicamente de la componente z del vector `torque`, una torsión en torno al eje z del cuadricóptero. Para llegar a este valor es preciso rotar el vector velocidad angular según la orientación del robot, ya que Gazebo nos proporciona datos en su sistema de referencia, con los ejes cartesianos fijos e independientes a la situación de los modelos.

Gazebo proporciona la clase `math::Quaternion`. Como hemos dicho, el miembro `pose` contiene un dato de este tipo, que se utiliza para guardar el eje de inclinación del cuadricóptero y el ángulo de rotación en torno a este. Utilizando el método `RotateVectorReverse` sobre este cuaternión y pasando como parámetro cualquiera de los vectores se consigue rotarlos sobre este eje un ángulo opuesto al del robot, que es equivalente a pasar del sistema de coordenadas de Gazebo al del *drone*. El resultado de rotar de esta manera el vector `angular_velocity` se guarda en el vector `angular_velocity_body`.

Ahora se llama al método `update` del controlador PID encargado de ajusar la velocidad angular en torno al eje z (`controllers_.yaw`). Los parámetros utilizados en este caso son la velocidad angular deseada (`velocity_command.angular.z`), la que el *drone* tiene (`angular_velocity_body`) y el intervalo de tiempo transcurrido desde la anterior llamada a `OnUpdate`. La variable retornada es la aceleración angular. Multiplicando este valor por el momento de inercia correspondiente se obtiene el momento de fuerza necesario para realizar este movimiento. Un esquema de este controlador puede verse en la figura 3.9. La parte del método `Update` que realiza esta función está incluída en el código 3.11.

```
...
math::Vector3 angular_velocity_body =
    pose.rot.RotateVectorReverse(angular_velocity);
...
torque.z = inertia.z * controllers_.yaw.update(
    velocity_command_.angular.z, angular_velocity.z, 0, dt);
...
```

Código 3.11: Continuación de `Update`. Obtención de la torsión en torno a z.

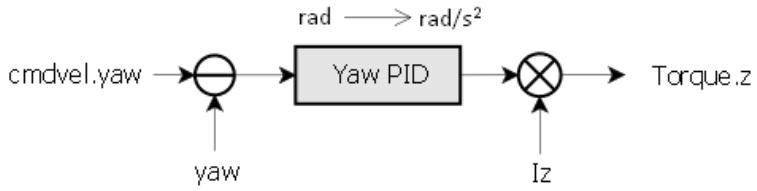


Figura 3.9: Diagrama de bloques del controlador para la velocidad angular en torno al eje z.

Movimiento horizontal

Este movimiento es el que se produce en el plano horizontal, en las direcciones adelante / atrás y derecha / izquierda. En este caso, las direcciones de movimiento están referidas al cuadricóptero sin inclinar, con su eje de inclinación paralelo al eje z de Gazebo. Es por esto que no se puede utilizar el mismo cuaternión

que el calculado para el movimiento rotatorio anterior, ya que si el *roll* o el *pitch* del *drone* son distintos de cero, los vectores rotados no serán paralelos al plano horizontal. Se necesita definir otro cuaternión que solo tenga en cuenta el ángulo *yaw* del robot, llamado `heading_quaternion`. Debido a que el eje del nuevo cuaternión será el eje z, los ángulos con respecto a x e y son rectos y su coseno es cero, por lo que solo habrá componentes no nulas en q_0 y q_3, como podemos ver en la ecuación 3.3.

$$\begin{aligned} q_0 &= \cos(yaw/2) \\ q_1 &= 0 \\ q_2 &= 0 \\ q_3 &= \sin(yaw/2) \end{aligned} \tag{3.3}$$

Las variables que interesan aquí son la velocidad y la aceleración lineal. Una vez rotados, solo son necesarias las componentes x e y de dichos vectores.

Para que el desplazamiento sea realista, no basta con que el robot se mueva hacia un lado u otro. Este movimiento se consigue en el *drone* real reduciendo la velocidad de los motores correspondientes y aumentando la de los opuestos, como hemos visto en la sección 1.2, provocando un ligero balanceo.

Es aquí donde intervienen las componentes x e y de la fuerza de torsión. Para hallarlas con los controladores hace falta conocer los ángulos *roll* y *pitch* actuales y los que se quieren alcanzar. Para estos últimos se utilizan las variables `roll_command` y `pitch_command`, a las que se asignará la salida de los controladores `velocity_x` y `velocity_y` respectivamente, ya que hará falta más inclinación cuanta más velocidad se desee alcanzar. Una vez hallados estos ángulos se obtienen las salidas de los controladores `roll` y `pitch`, se multiplican por los momentos de inercia correspondientes y se asignan los resultados a las componentes `torque.x` y `torque.y`. En la figura 3.10 se encuentra un diagrama de bloques de estos controladores. La parte del método `Update` que realiza esta función puede observarse en el código 3.12.

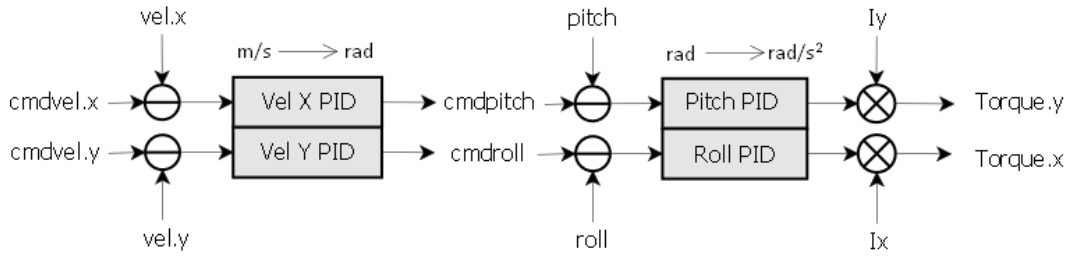


Figura 3.10: Diagrama de bloques de los controladores para las componentes x e y de la velocidad lineal.

```
...
math::Quaternion heading_quaternion(cos(euler.z/2),0,0,sin(euler.z/2));
math::Vector3 velocity_xy =
    heading_quaternion.RotateVectorReverse(velocity);
math::Vector3 acceleration_xy =
    heading_quaternion.RotateVectorReverse(acceleration);
...
double pitch_command =  controllers_.velocity_x.update(
    velocity_command_.linear.x,
    velocity_xy.x,
    acceleration_xy.x, dt) / gravity;
```

```
double roll_command  = -controllers_.velocity_y.update(
    velocity_command_.linear.y,
    velocity_xy.y,
    acceleration_xy.y, dt) / gravity;
torque.x = inertia.x * controllers_.roll.update(
    roll_command, euler.x, angular_velocity_body.x, dt);
torque.y = inertia.y * controllers_.pitch.update(
    pitch_command, euler.y, angular_velocity_body.y, dt);
...
```

Código 3.12: Método `GazeboQuadrotorSimpleController::Update`. Obtención de las torsiones en torno a x e y.

No es necesario calcular otras fuerzas distintas a las de torsión aunque queramos un movimiento lineal. La fuerza necesaria procederá de la que se aplica en sentido perpendicular al plano del cuadricóptero para controlar la altura, puesto que al inclinarse, dicha fuerza no será del todo vertical. Tendrá una componente horizontal que permitirá a Gazebo mover al robot, como se puede distinguir en la figura 3.11.

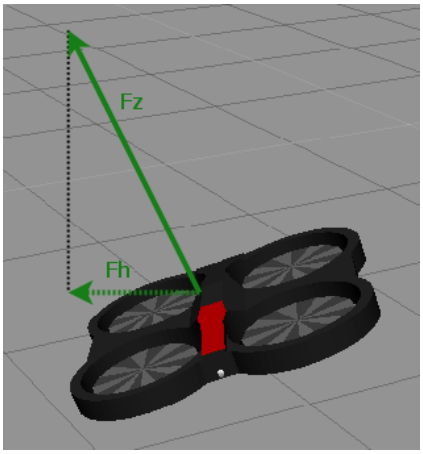


Figura 3.11: Al inclinarse el cuadricóptero la fuerza perpendicular F_z tiene una componente horizontal F_h que hace que se produzca movimiento lateral.

Movimiento vertical

El último movimiento lineal es el movimiento vertical. Se logra de manera muy similar al movimiento anterior. En lugar de utilizar las componentes x e y de los vectores de velocidad y aceleración, se utiliza la componente z y el controlador PID correspondiente (`controllers_.velocity_z`). La salida de este controlador, sin embargo, no se considera un ángulo, sino una aceleración directamente, por lo que habrá que multiplicarla por la masa del robot para hallar la fuerza. Para este movimiento el cuadricóptero no necesita cambiar su inclinación, así que no hace falta controlarla como en el caso anterior. Podemos ver un diagrama para este controlador en la figura 3.12.

Una vez realizados estos cálculos falta añadir al resultado una cantidad proporcional a la gravedad, asignada a la variable `gravity`. Esto se debe a que los rotores del cuadricóptero tienen que provocar una fuerza compensatoria a la gravedad para mantenerse en el aire aunque no se ordene ningún movimiento. La

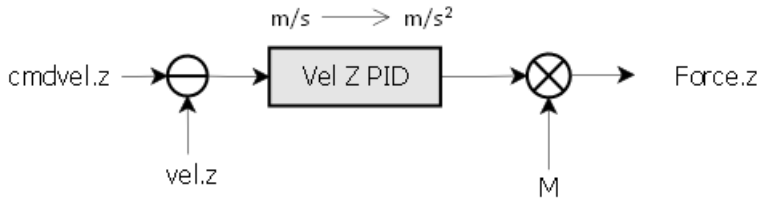


Figura 3.12: Diagrama de bloques del controlador para la componente z de velocidad lineal.

cantidad que se añade, aun así, no es siempre igual a la gravedad. Como hemos visto en la sección anterior, los movimientos horizontales se producen cuando el robot se inclina. Esta torsión hace que la fuerza calculada para el movimiento vertical deje de ser paralela al eje z y tenga una componente horizontal, permitiendo el movimiento en esas direcciones, pero reduciendo la componente vertical de la fuerza, lo cual haría que el robot cayese poco a poco. Aquí es donde se utiliza la variable `load_factor`, que presentamos con anterioridad. Esta variable mide el nivel de inclinación y toma valores que van de 1 (nada de inclinación) hasta infinito (inclinación de 90 grados). La cantidad que añadiremos entonces a la fuerza en cuestión será el producto de la gravedad por `load_factor`, de manera que la fuerza será más grande cuanto más inclinado esté el *drone*, evitando la pérdida de altitud (ver figura 3.13). La parte del método `Update` que realiza esta función se encuentra en el código 3.13.

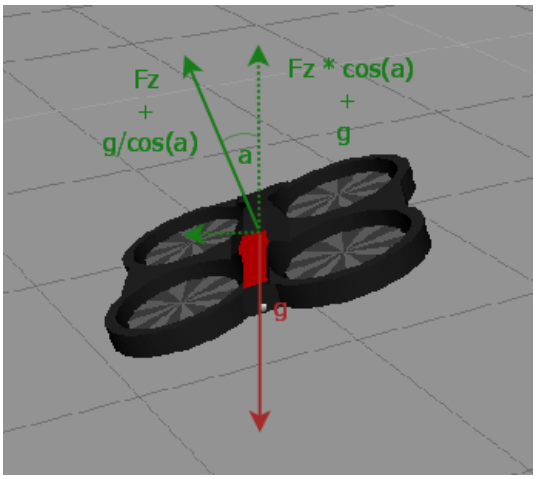


Figura 3.13: Efecto compensatorio de `load_factor`, cuyo valor es $1/\cos(a)$. La componente vertical compensa la gravedad durante los movimientos horizontales aunque F_z sea nula.

```
...
force.z = mass * (controllers_.velocity_z.update(
    velocity_command_.linear.z,
    velocity.z, acceleration.z, dt) +
    load_factor * gravity);
if (max_force_ > 0.0 && force.z > max_force_) force.z = max_force_;
if (force.z < 0.0) force.z = 0.0;
...
```

Código 3.13: Método `GazeboQuadrotorSimpleController::Update`. Obtención de la fuerza en z.

Aplicación de fuerzas

A la hora de añadir los vectores `force` y `torque` al modelo simulado se han distinguido tres estados en los que la manera de aplicarlos cambia ligeramente. Su magnitud variará dependiendo de si el cuadricóptero está despegando, aterrizando o en modo vuelo. En cualquier caso, los métodos utilizados sobre la variable `link` serán `AddRelativeForce`, para aplicar fuerzas, y `AddRelativeTorque`, para torsiones.

El cálculo de la fuerza de los cuatro rotores se hace siempre de la misma forma. Lo único que hay que hacer es comprobar cuál es el estado del cuadricóptero, almacenado en el miembro dato `navi_state`. Cuando el *drone* está despegando, la fuerza necesaria debe superar su peso. En este caso la fuerza se multiplica por un factor de 1'5. Si el estado es el de aterrizar, dicho valor escalar es de 0'8. En modo vuelo la fuerza se aplica sin multiplicar por ningún escalar. La programación de esta sección está en el código 3.14.

```
  ...
// process robot state information
if(navi_state == LANDED_MODEL)
{
}
else if((navi_state == FLYING_MODEL)||(navi_state == TO_FIX_POINT_MODEL))
{
  link->AddRelativeForce(force);
  link->AddRelativeTorque(torque);
}
else if(navi_state == TAKINGOFF_MODEL)
{
  link->AddRelativeForce(force*1.5);
  link->AddRelativeTorque(torque*1.5);
}
else if(navi_state == LANDING_MODEL)
{
  link->AddRelativeForce(force*0.8);
  link->AddRelativeTorque(torque*0.8);
}
...
```

Código 3.14: Método `GazeboQuadrotorSimpleController::Update`. Aplicación de fuerzas.

Control del estado

La clase que desempeña la función de una máquina de estados se llama `GazeboQuadrotorStateController`. Funciona como un *plugin* separado del anterior. Tiene varias funciones *callback* que se ejecutan al recibir órdenes como despegar o aterrizar (códigos 3.15 y 3.16). Las variables `m_isFlying`, `m_takeoff` y `m_batteryPercentage` son usadas por el método `Update` para permanecer en un estado o cambiar a otro. La variable `robot_current_state` almacena el estado actual del *drone*, cuyos valores más relevantes son:

- `UNKNOWN_MODEL`: Estado desconocido.

- `INITIALIZE_MODEL`: Mientras inicia.
- `LANDED_MODEL`: Cuando el *drone* está en tierra.
- `FLYING_MODEL`: Cuando el *drone* está en el aire.
- `HOVERING_MODEL`: Cernido en un punto.
- `TAKINGOFF_MODEL`: Mientras está despegando.
- `TO_FIX_POINT_MODEL`: Hacia un punto fijo.
- `LANDING_MODEL`: Mientras está aterrizando.

```
void GazeboQuadrotorStateController::TakeoffCallback(const std_msgs::EmptyConstPtr& msg)
{
  if(robot_current_state == LANDED_MODEL)
  {
    m_isFlying = true;
    m_takeoff = true;
    m_batteryPercentage = 100.;
    ROS_INFO("%s","\nQuadrotor takes off!!");
  }
  else if(robot_current_state == LANDING_MODEL)
  {
    m_isFlying = true;
    m_takeoff = true;
    ROS_INFO("%s","\nQuadrotor takes off!!");
  }
}
```

Código 3.15: Método `GazeboQuadrotorStateController::TakeoffCallBack`.

```
void GazeboQuadrotorStateController::LandCallback(const std_msgs::EmptyConstPtr& msg)
{
  if((robot_current_state == FLYING_MODEL)||
      (robot_current_state == TO_FIX_POINT_MODEL)||
      (robot_current_state == TAKINGOFF_MODEL))
  {
    m_isFlying = false;
    m_takeoff = false;
    ROS_INFO("%s","\nQuadrotor lands!!");
  }
}
```

Código 3.16: Método `GazeboQuadrotorStateController::LandCallBack`.

El método `Update` de esta clase hace que se comprueben estas variables para pasar de un estado a otro, como se muestra en el código 3.17.

```
void GazeboQuadrotorStateController::Update() {
  ...
  // process robot operation information
  if ((m_takeoff) && (robot_current_state == LANDED_MODEL)) {
    m_timeAfterTakeOff = 0;
    m_takeoff = false;
    robot_current_state = TAKINGOFF_MODEL;
  } else if (robot_current_state == TAKINGOFF_MODEL) {
    // take off phase need more power
    if (!sonar_topic_.empty()) {
      if (robot_altitude > 0.5) {
        robot_current_state = FLYING_MODEL;
      }
    } else {
      m_timeAfterTakeOff += dt;
      if (m_timeAfterTakeOff > 0.5) {
        robot_current_state = FLYING_MODEL;
      }
    }
    if (m_isFlying == false) {
      m_timeAfterTakeOff = 0;
      robot_current_state = LANDING_MODEL;
    }
  } else if ((robot_current_state == FLYING_MODEL)||
      (robot_current_state == TO_FIX_POINT_MODEL)) {
    if (m_isFlying == false) {
      m_timeAfterTakeOff = 0;
      robot_current_state = LANDING_MODEL;
    }
  } else if (robot_current_state == LANDING_MODEL) {
    if (!sonar_topic_.empty()) {
      m_timeAfterTakeOff += dt;
      if ((robot_altitude < 0.2) || (m_timeAfterTakeOff > 5.0)) {
        robot_current_state = LANDED_MODEL;
      }
    } else {
      m_timeAfterTakeOff += dt;
      if (m_timeAfterTakeOff > 1.0) {
        robot_current_state = LANDED_MODEL;
      }
    }
    if (m_isFlying == true) {
      m_timeAfterTakeOff = 0;
      m_takeoff = false;
      robot_current_state = TAKINGOFF_MODEL;
    }
```

```
    }
    ...
}
```

Código 3.17: Método `GazeboQuadrotorStateController::Update`.

Podemos ver un diagrama, en la figura 3.14, de las principales clases que nos han sido de utilidad. Las funciones con el sufijo *Callback* son las ejecutadas al recibir cierto tipo de mensaje.

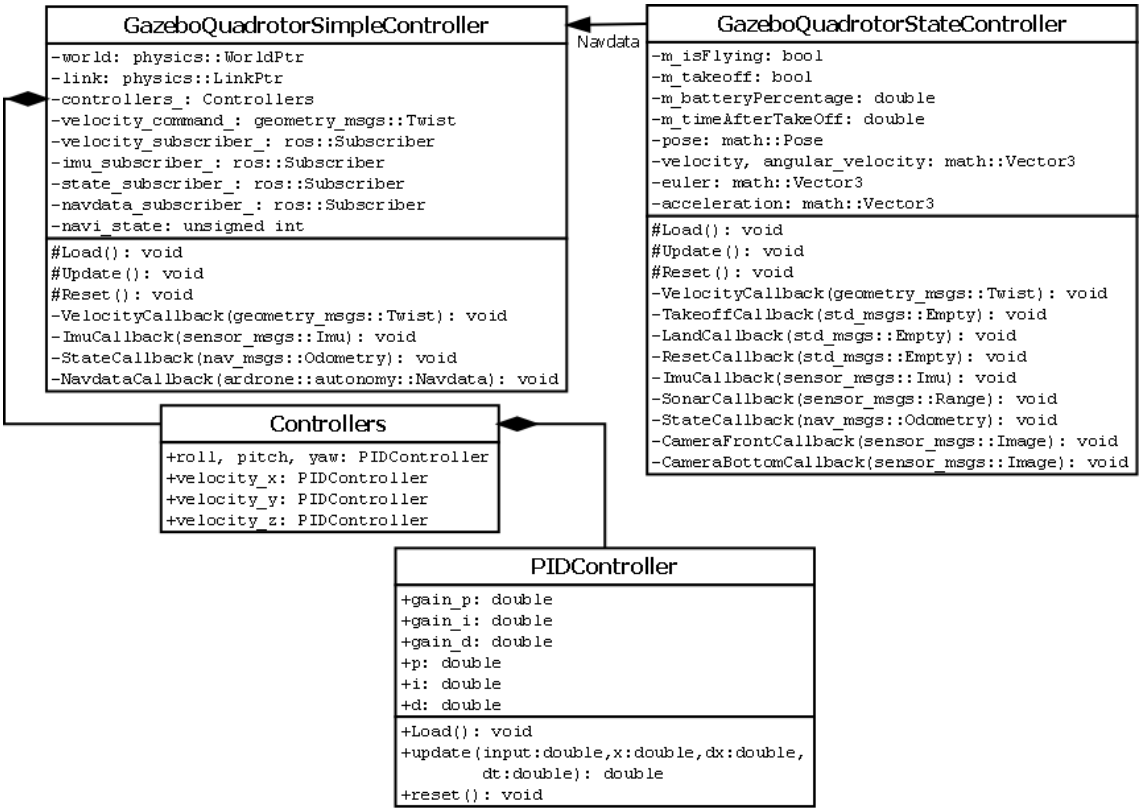


Figura 3.14: Diagrama de clases de TUM_simulator.

3.5. AprilTags

AprilTags son unas bibliotecas para el procesamiento de imágenes con implementación en Java y en C. Son útiles para tareas como realidad aumentada, robótica o calibración de cámaras. Proporciona un mecanismo de detección de figuras o etiquetas dentro de una imagen, similar al funcionamiento de los códigos QR. Permite obtener las coordenadas dentro de la imagen y la orientación de la etiqueta con respecto al plano de la imagen en 3D.

Estas bibliotecas nos han sido útiles para la aplicación de navegación desarrollada en este trabajo, concretamente en el barrido de un área en busca de determinados objetos. En Gazebo hemos diseñado estos objetos como figuras cuya textura es una imagen que contiene una de estas etiquetas. Nos hemos valido de este algoritmo de detección para localizarlos y guardar información como las coordenadas y la imagen.

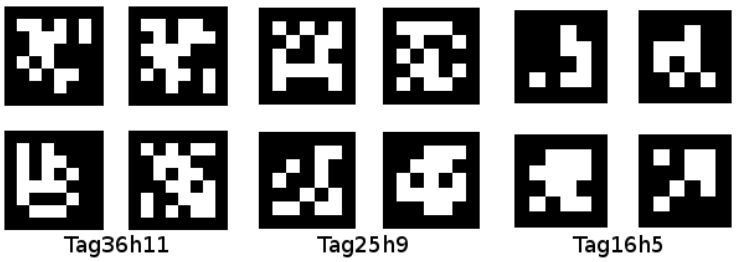


Figura 3.15: Etiquetas para detección en AprilTags.

Capítulo 4

Desarrollo de soporte para UAVs simulados

En este capítulo se describe el soporte JdeRobot para los cuadricópteros en el simulador Gazebo, que consiste en el desarrollo de *plugins* para Gazebo, Estos hacen el papel de *drivers* y proporcionan a las aplicaciones robóticas acceso a sensores y actuadores del cuadricóptero. Este soporte está basado en los *drivers* que ha desarrollado el Departamento de Ciencias de la Computación de la Universidad Técnica de Munich[17] utilizando el entorno ROS, de igual forma que el modelo SDF del ArDrone está basado en el modelo URDF de dicha universidad.

Los *plugins* desarrollados se han usado en el curso superior universitario en programación de *drones*[27] llevado a cabo en la Universidad Rey Juan Carlos, en el que alrededor de treinta participantes tuvieron que programar a un cuadricóptero para desplazarse mediante control por posición a través de una serie de balizas, control visual siguiendo a un robot terrestre y control visual siguiendo una carretera.

A continuación se explicará la estructura y el funcionamiento de los *drivers* para el UAV simulado indicando en cada apartado las clases más relevantes y sus funciones.

4.1. Diseño global

Como hemos dicho, los *drivers* desarrollados son un conjunto de *plugins*, o bibliotecas dinámicas, que ofrecen las interfaces ICE permitiendo que otras aplicaciones JdeRobot accedan a los sensores y actuadores del cuadricóptero simulado conectándose con puertos UDP o TCP. Al iniciar una simulación, Gazebo se encargará de arrancar las bibliotecas dinámicas requeridas para el *drone* virtual y estas alterarán periódicamente sus variables internas, produciendo movimiento, fuerzas, cambios de estado, etc. Los *drivers* se sirven de las bibliotecas de Gazebo y materializan interfaces ICE con las que permitir el acceso a otros componentes JdeRobot. La figura 4.1 esquematiza las conexiones entre distintos componentes mediante estas interfaces. El componente de la figura solo necesita que se establezcan los puertos con los que conectarse al UAV simulado o al real. Una vez conectado a uno de los dos, podrá darle instrucciones o leer sus datos.

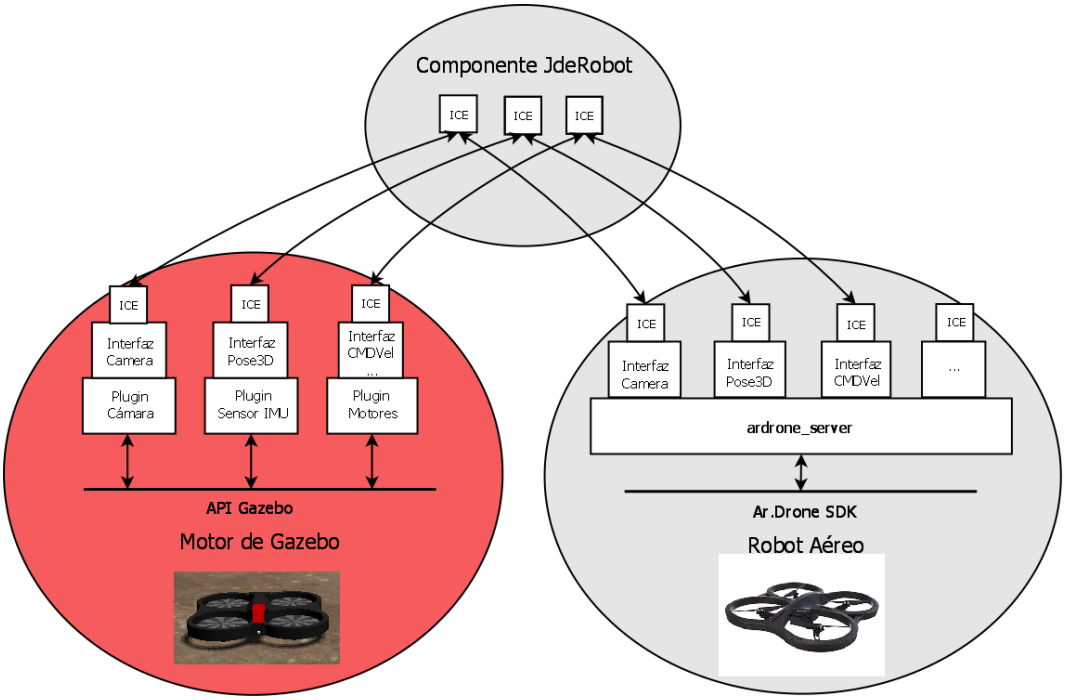


Figura 4.1: Esquema de conexiones de los *drivers* para cuadricópteros en Gazebo. Las aplicaciones JdeRobot externas pueden conectarse tanto al robot simulado (izquierda) como al real (derecha) especificando los puertos y la máquina para cada interfaz.

El modelo del ArDrone para Gazebo lo hemos obtenido a partir del proporcionado por los *drivers* `TUM_simulator`, cambiando su formato URDF (*Universal Robotic Description Format*), utilizado normalmente por el entorno ROS, por el SDF (*Simulation Description Format*). Este modelo aparece en un mundo de Gazebo en la figura 4.2.

Las interfaces ICE utilizadas son las mismas que implementan los *drivers* JdeRobot para el cuadricóptero real (componente `ardrone_server`[13]), explicadas en la sección 3.3.2:

- `navdata`: para obtener estructuras de tipo `Navdata` con información sensorial del *drone*, como batería, velocidad, etc.
- `pose3d`: para acceder a estructuras `Pose3D` con coordenadas espaciales y rotaciones.
- `camera`: para obtener fotogramas de los vídeos proporcionados por las cámaras del *drone*.
- `remoteConfig`: para configurar el *drone* "en caliente".
- `cmdVel`: para comandar velocidades específicas.
- `ardrone_extra`: para acceder a funcionalidades extra del cuadricóptero y ordenar maniobras básicas.

Las tres primeras corresponden a la parte sensorial del robot, con datos como aceleración, orientación, batería restante, flujos de vídeo... Este cuadricóptero tiene una amplia variedad de sensores a bordo, pero nosotros nos hemos centrado en dar acceso a las cámaras, al sensor IMU y al GPS. Las tres últimas interfaces aportan mecanismos para controlar y configurar los actuadores, que consisten en cuatro motores que permiten desplazamientos a lo largo de los ejes x, y y z cartesianos y giros en torno al eje z.

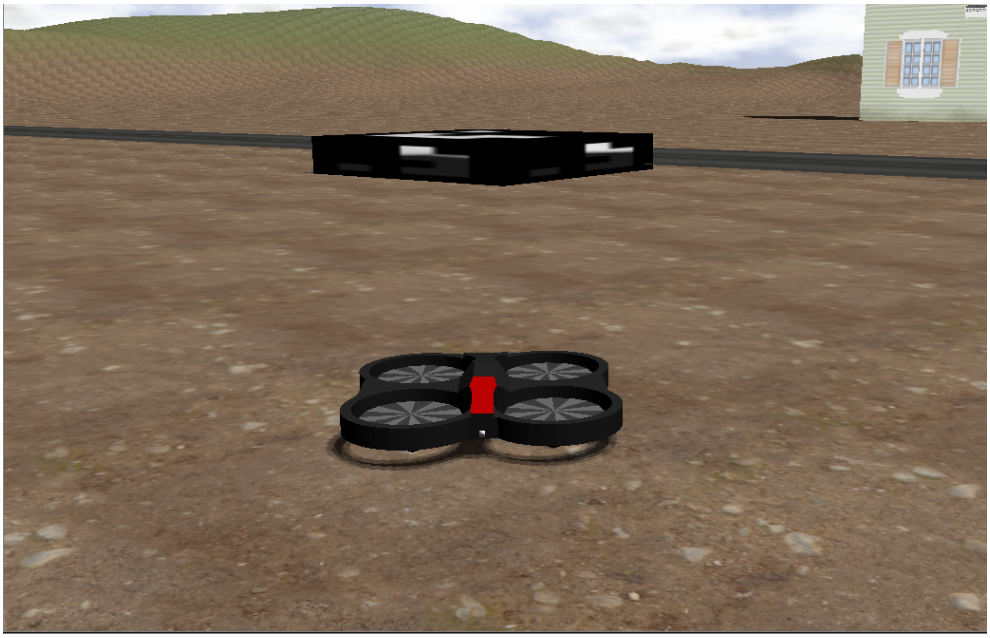


Figura 4.2: Modelo del ArDrone en Gazebo.

Los mecanismos internos de las bibliotecas dinámicas explicadas en este capítulo se basan en la ejecución de una serie de funciones y un registro de la biblioteca como aparece en la figura 4.3. Tras registrar el *plugin*, seguirá el método `Load` y después el `Init`. En el primero se cargarán distintos parámetros del fichero SDF, como son en nuestro caso el fichero de configuración con los puertos para las interfaces ICE, el nombre del modelo que el *plugin* controlará o las constantes para los controladores PID internos. En `Init` se dan valores iniciales a algunas de las variables con las que se llevarán a cabo los cálculos, como son la velocidad objetivo o el estado, y se construyen las estructuras que lo necesiten, como la imagen inicial que ofrece la cámara. Si se especifica en alguno de estos dos métodos, Gazebo ejecutará periódicamente una función, en este caso `OnUpdate`, la cual interactuará con el robot simulado modificando o manteniendo alguna de sus características. A lo largo de la simulación se recibirán peticiones ICE asíncronas que podrán provocar cambios en las variables de los controladores.

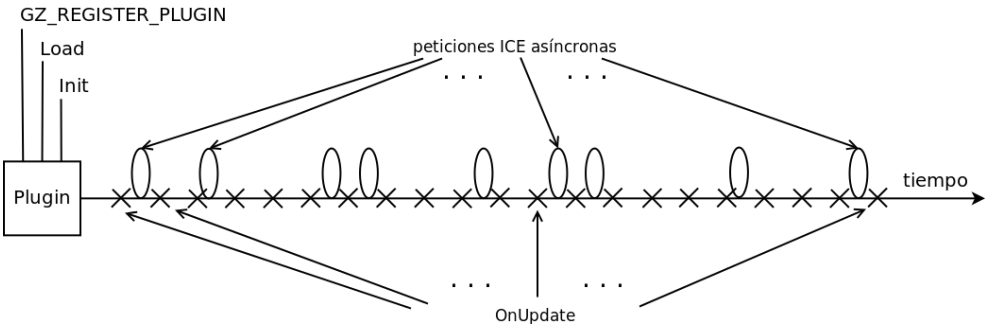


Figura 4.3: Ejecución típica de un *plugin* de JdeRobot.

Para una mejor comprensión del trabajo vamos a separar el desarrollo en función de con cuál de las interfaces se relaciona. Vamos a comenzar con la parte sensorial, seguida de la parte de actuadores y terminaremos con la de configuración.

4.2. Sensores

Los principales sensores en los que se ha enfocado este proyecto son un sensor IMU, o *Inertial Measurement Unit*, y dos cámaras, una frontal y otra ventral, como en el caso del ArDrone.

4.2.1. Sensores IMU y GPS

Vamos a detallar los mecanismos desarrollados para simular los sensores cuya información permite formar un dato `Pose3D`, y, por tanto, implementar la interfaz con el mismo nombre. Se trata de un sensor IMU para obtener la rotación y un sensor GPS para la posición 3D. Es importante observar que un sensor GPS no da información acerca de la altitud a la que se encuentra el cuadricóptero. Esta se obtendría a partir de un sensor de altitud, ultrasonidos, infrarrojos, etc. La longitud y la latitud, que sí proporciona un GPS, pueden pasarse de grados a metros, pero como Gazebo permite obtener directamente las coordenadas en metros podemos saltarnos estos pasos y simplificar los *plugins*. También hay que tener en cuenta que, para el correcto funcionamiento de estos controladores y de las aplicaciones que reciban sus datos, el cuadricóptero debe estar sobre una superficie horizontal en el momento de inicio de la simulación. Si no, los ángulos *roll* y *pitch* no serán correctos.

Se ha separado el código de cada parte en dos *plugins* distintos: `ImuPlugin` y `PosePlugin`. `ImuPlugin` se encarga de simular el funcionamiento del sensor IMU. Al arrancar, en su método `Load`, obtiene una referencia de un objeto sensor, de tipo `sensors::Sensor`, de las bibliotecas de Gazebo, sobre la cual llamar a funciones para calibrar al sensor u obtener datos relativos a la orientación del UAV. `PosePlugin` controla un modelo virtual, un objeto simulado. La referencia que recibe en su método `Load` no es la de un sensor, sino la del modelo que contiene a este *plugin*. El objeto proporcionado por Gazebo que representa a un modelo es del tipo `physics::Model`, y aunque permite invocar métodos para dar movimiento al robot, nosotros lo utilizamos solo para leer sus coordenadas tridimensionales, por lo que sirve como implementación de un sensor GPS más uno de altitud.

Los datos ofrecidos por estos *drivers* adoptan la forma de `Pose3D`, estándar de JdeRobot para expresar posición y orientación, representadas por un vector y un cuaternión, respectivamente. Para conformarlo, las coordenadas tridimensionales que obtiene `PosePlugin` son enviadas a `ImuPlugin`, que las combina con el cuaternión que ha obtenido y almacena la estructura final. Los dos *plugins* actualizan sus variables al ritmo al que itera Gazebo. `PosePlugin` envía al mismo ritmo la posición a `ImuPlugin`, que es el que implementa la interfaz ICE `pose3d` y el que dispone de los datos actualizados para que accedan a ellos otros procesos. Podemos ver un diagrama en la figura 4.4.

A continuación se muestra el código 4.1, perteneciente a `ImuPlugin`. La primera función, `PoseCallback`, es la encargada de recibir de Gazebo los datos referentes a las coordenadas x, y y z y el ángulo *yaw*. Esta se ejecuta de manera similar a los *callbacks* de ROS en el sistema de *topics*. Cada vez que el GPS escriba sus datos en el *topic* `pose_topic`, al cual el sensor IMU se subscribe al ser arrancado por el simulador, se invocará a `PoseCallback`. La variable global `math::Pose pose` es importante ya que es en la que se guarda el vector de posición y del ángulo *yaw*.

La segunda función que aparece es `OnUpdate`, la cual es llamada por Gazebo a una determinada frecuencia. Aquí se lee de la variable que representa al sensor (`parentSensor`) la orientación con el método `GetOrientation` y se almacena en `orien`.

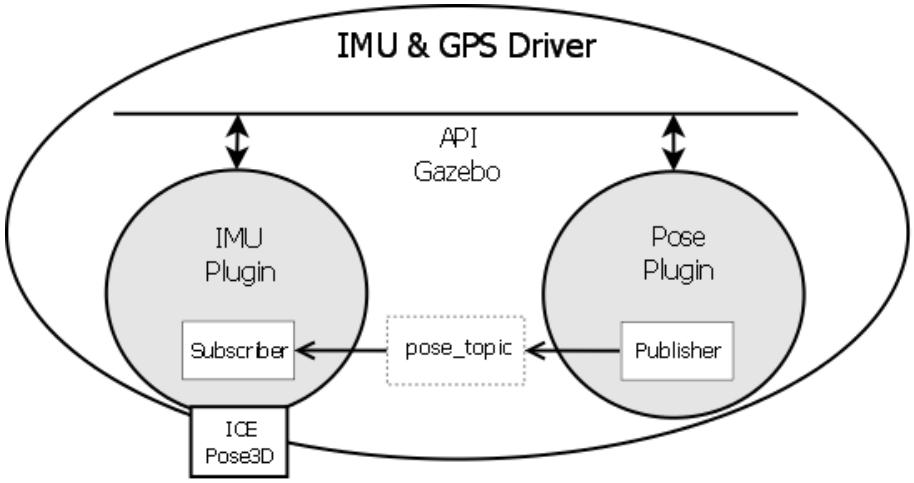


Figura 4.4: Esquema de conexiones de *drivers* IMU y GPS.

```
void PoseCallback(const boost::shared_ptr<const msgs::Pose> &_msg) {
    pthread_mutex_lock(&pose_mutex);
    pose = msgs::Convert(*_msg);
    pthread_mutex_unlock(&pose_mutex);
}

void ImuPlugin::OnUpdate() {
    pthread_mutex_lock(&this->mutex_imuplugin);
    imuplugin.orien = this->parentSensor->GetOrientation();
    pthread_mutex_unlock(&this->mutex_imuplugin);
    pthread_mutex_lock(&pose_mutex);
    imuplugin.pose.pos = pose.pos;
    pthread_mutex_unlock(&pose_mutex);
}
```

Código 4.1: imuplugin.cc. Funciones `PoseCallback` y `OnUpdate`.

El código 4.2 muestra el método `OnUpdate` de `PosePlugin`. En este se lee la posición utilizando de la API de Gazebo el método `GetWorldPose` sobre la variable que representa el modelo del cuadricóptero (`model`) y la publica para que sea recibida por el *plugin* `imuplugin`. De esta manera se actualizan la posición y la orientación a la misma frecuencia.

```
void PosePlugin::OnUpdate() {
    math::Pose pose = model->GetWorldPose();
    msgs::Pose msg;
    msgs::Set(&msg, pose);
    pub_->Publish(msg);
}
```

Código 4.2: poseplugin.cc: Método `OnUpdate`

Hace falta una tercera clase que implemente la interfaz ICE y con la cual otras aplicaciones puedan

leer el `Pose3DData`. Esta clase se llama `Pose3DI`, y el hilo de ejecución que la contiene arranca con el *plugin* del sensor IMU, por lo que la interfaz estará activa mientras esta biblioteca dinámica se ejecute aunque Gazebo no haya cargado el *plugin* de posición `PosePlugin`. Sin embargo la ausencia de dicho *plugin* hará que la posición no se actualice. En el código 4.3 podemos ver los métodos más importantes de la interfaz `Pose3D`, que son `setPose3DData` y `getPose3DData`. Estos son los métodos que otros procesos podrán invocar remotamente vía ICE. Su objetivo es recibir y ofrecer el dato `Pose3DData` del sensor IMU virtual del cual `Pose3DI` posee una referencia, llamada en el código `imu_p`.

```
virtual jderobot::Pose3DDataPtr getPose3DData ( const Ice::Current& ) {
    pthread_mutex_lock(&imu_p->mutex_imuplugin);

    pose3DData->x = imu_p->imuplugin.pose.pos.x;
    pose3DData->y = imu_p->imuplugin.pose.pos.y;
    pose3DData->z = imu_p->imuplugin.pose.pos.z;
    pose3DData->h = 0;
    pose3DData->q0 = imu_p->imuplugin.orien.w;
    pose3DData->q1 = imu_p->imuplugin.orien.x;
    pose3DData->q2 = imu_p->imuplugin.orien.y;
    pose3DData->q3 = imu_p->imuplugin.orien.z;

    pthread_mutex_unlock(&imu_p->mutex_imuplugin);

    return pose3DData;
}

virtual Ice::Int setPose3DData ( const jderobot::Pose3DDataPtr & data,
                                 const Ice::Current& ) {
    pthread_mutex_lock(&imu_p->mutex_imuplugin);

    imu_p->imuplugin.pose.pos.x = data->x;
    imu_p->imuplugin.pose.pos.y = data->y;
    imu_p->imuplugin.pose.pos.z = data->z;
    imu_p->imuplugin.orien.w = data->q0;
    imu_p->imuplugin.orien.x = data->q1;
    imu_p->imuplugin.orien.y = data->q2;
    imu_p->imuplugin.orien.z = data->q3;

    pthread_mutex_unlock(&imu_p->mutex_imuplugin);
}
```

Código 4.3: imuplugin.cc. Clase `Pose3DI`

Por último podemos observar un diagrama *software* global de estos controladores en la figura 4.5.

4.2.2. Cámaras

Para simular las cámaras se ha desarrollado el *plugin* `ToggleCamPlugin`, que permite entregar los fotogramas a otras aplicaciones y, cuando estas lo soliciten, cambiar la cámara de la que se recibe el

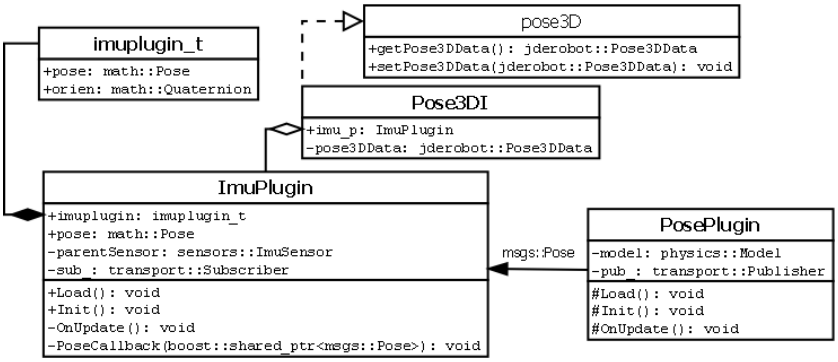


Figura 4.5: Diagrama de *drivers* IMU y GPS.

flujo de imágenes. En nuestro modelo de cuadricóptero se cargan dos *plugins* de este tipo, uno por cada cámara. Cada uno lee del fichero SDF del cuadricóptero una etiqueta en la cual se especifica si la cámara correspondiente es la que envía imágenes al iniciar la simulación. Por lo tanto, una etiqueta deberá indicar lo contrario que la otra.

La clase `ToggleCamPlugin` se comunica con `QuadrotorPlugin`, que es el *plugin* principal y el que implementa la interfaz ICE `camera`, además de las interfaces de actuación. La comunicación transcurre por dos vías: En un sentido, las cámaras reciben la orden de apagarse (dejar de enviar imágenes) o encenderse de este *plugin* principal, y en el otro le envía imágenes al ritmo que Gazebo las actualiza cuando está encendida. `QuadrotorPlugin`, por lo tanto, está constantemente recibiendo imágenes, o bien de la cámara frontal, o bien de la ventral. Las comunicaciones se llevan a cabo mediante el sistema de *topics*, al igual que en el apartado anterior. Un esquema de los componentes que intervienen en este proceso y sus conexiones puede verse en la figura 4.6.

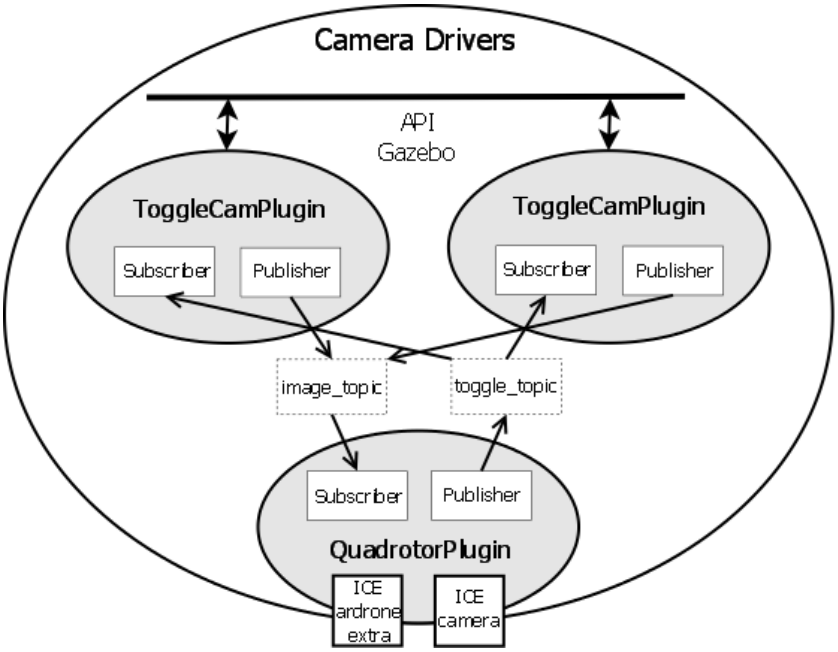


Figura 4.6: Esquema de conexiones de `ToggleCamPlugin`.

El método `OnNewFrame` es el que Gazebo ejecuta periódicamente. Podemos verlo en el código 4.4. En

él, primero se comprueba si la variable `on_` vale `false`, en cuyo caso se sale del método. Si no, se almacena la imagen obtenida por el sensor (recibida como parámetro) en una variable de tipo `common::Image`. Esta variable se introduce en un mensaje `msgs::Image` que será enviado a un *topic* cuyo nombre se establece en el modelo SDF del cuadricóptero. El mensaje lo recibirá el *plugin* principal, que guardará la imagen para que la interfaz ICE disponga de ella.

```
void ToggleCamPlugin::OnNewFrame(const unsigned char *_image,
    unsigned int _width, unsigned int _height, unsigned int _depth,
    const std::string &_format) {
    pthread_mutex_lock (&mutex);
    if (!on_) {
        pthread_mutex_unlock (&mutex);
        return;
    }
    if(count==0){
    image.create(_height, _width, CV_8UC3);
    count++;
    }
    memcpy((unsigned char *) image.data, &(_image[0]), _width*_height * 3);
    common::Image msg_image;
    msg_image.SetFromData((unsigned char *)image.data,
                                    _width,
                                    _height,
                                    common::Image::RGB_INT8);
    msgs::Image msg;
    msgs::Set(&msg, msg_image);
    pub_->Publish(msg);
    pthread_mutex_unlock (&mutex);
}
```

Código 4.4: Método `OnNewFrame` de `ToggleCamPlugin`

El método que se ejecuta al recibir la orden de alternar cámara es `ToggleCallback`, el cual simplemente cambia la variable `on_` al valor contrario. Podemos verlo en el código 4.5.

```
void ToggleCamPlugin::ToggleCallback(
        const boost::shared_ptr<const msgs::Vector2d> &_msg) {
    pthread_mutex_lock(&mutex);
    on_ = !on_;
    pthread_mutex_unlock(&mutex);
}
```

Código 4.5: Método `ToggleCallback` de `ToggleCamPlugin`

En el otro extremo, el método de `QuadrotorPlugin` que se ejecuta al recibir imágenes es `ImageCallback`, y extrae del mensaje la imagen y la copia en su variable `image`, del tipo de `opencv cv::Mat`. Este método puede verse en el código 4.6

```
void QuadrotorPlugin::ImageCallback(const boost::shared_ptr<const msgs::Image> &_msg) {
```

```
    common::Image c_image;
    msgs::Set(c_image, *_msg);
    pthread_mutex_lock(&img_mutex);
    //delete image;
    image = new cv::Mat();
    image->create(c_image.GetHeight(), c_image.GetWidth(), CV_8UC3);
    unsigned int size = c_image.GetHeight()*c_image.GetWidth()*3;
    unsigned char * char_arr[size];
    c_image.GetData(char_arr, size);
    memcpy((unsigned char *)image->data, *char_arr, size);
    pthread_mutex_unlock(&img_mutex);
}
```

Código 4.6: Método `ImageCallback` de `QuadrotorPlugin`

La clase `CameraI` de la biblioteca dinámica `QuadrotorPlugin` implementa la interfaz ICE `camera` y proporciona las imágenes almacenadas en este *plugin* extrayéndolas mediante el método `GetImage`.

Por último, la clase `ArDroneExtraI` de `QuadrotorPlugin` implementa la interfaz ICE `ardrone_extra`, cuyo propósito es dar acceso a funcionalidad extra del *drone*, una de las cuales es la de alternar las cámaras a través del método `toggleCam`. Este es el que realiza el envío del mensaje a los componentes `ToggleCamPlugin` para que se active o desactive el flujo de imágenes. Se ha utilizado un mensaje de tipo `msgs::Vector2d`, pero se podría haber utilizado cualquier otro, ya que no es necesario transmitir información. Vale con que `ToggleCallback` se ejecute y cambie al estado contrario de la cámara. Podemos ver un diagrama *software* de estos *drivers* en la figura 4.7

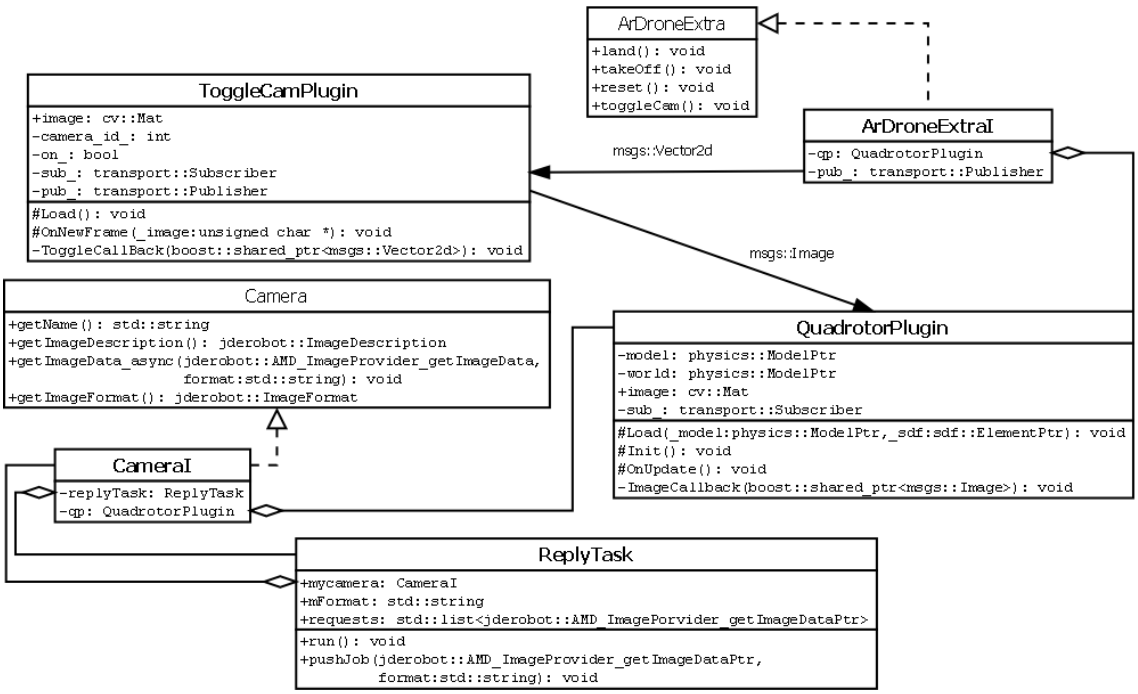


Figura 4.7: Diagrama del *driver* para las cámaras.

4.3. Actuadores

Vamos a explicar ahora cómo se llevan a cabo los movimientos en el UAV. Los actuadores del cuadricóptero son sus cuatro rotores. Para controlarlos, se hace uso de un único *plugin*: `QuadrotorPlugin`. Para ponerlos en marcha, este *plugin* implementa las interfaces `CMDVel` y `ardrone_extra`, cuyos métodos deben hacer al *drone* despegar, aterrizar, establecer una velocidad objetivo y resetear su estado.

No se ofrece un acceso de bajo nivel a cada motor, sino un acceso más abstracto asociado a todo el *drone*. Lo que se simula es la suma de los efectos de cada hélice, que en conjunto derivan en fuerzas para los movimientos lineales, y momentos de fuerza para los movimientos rotatorios. Gazebo proporciona mecanismos en forma de funciones y variables para aplicar dichos campos al objeto en cuestión. También se simula ruido en los comandos de velocidad, de manera que estos sufren variaciones aleatorias a lo largo del tiempo, provocando movimientos en el cuadricóptero que no son deterministas. Esto hace que el comportamiento se asemeje más al cuadricóptero real, ya que es prácticamente imposible mantenerlo cernido a un punto sin leves movimientos. La configuración del ruido se establece en el fichero SDF que describe al cuadricóptero simulado. Es posible eliminarlo si se desea.

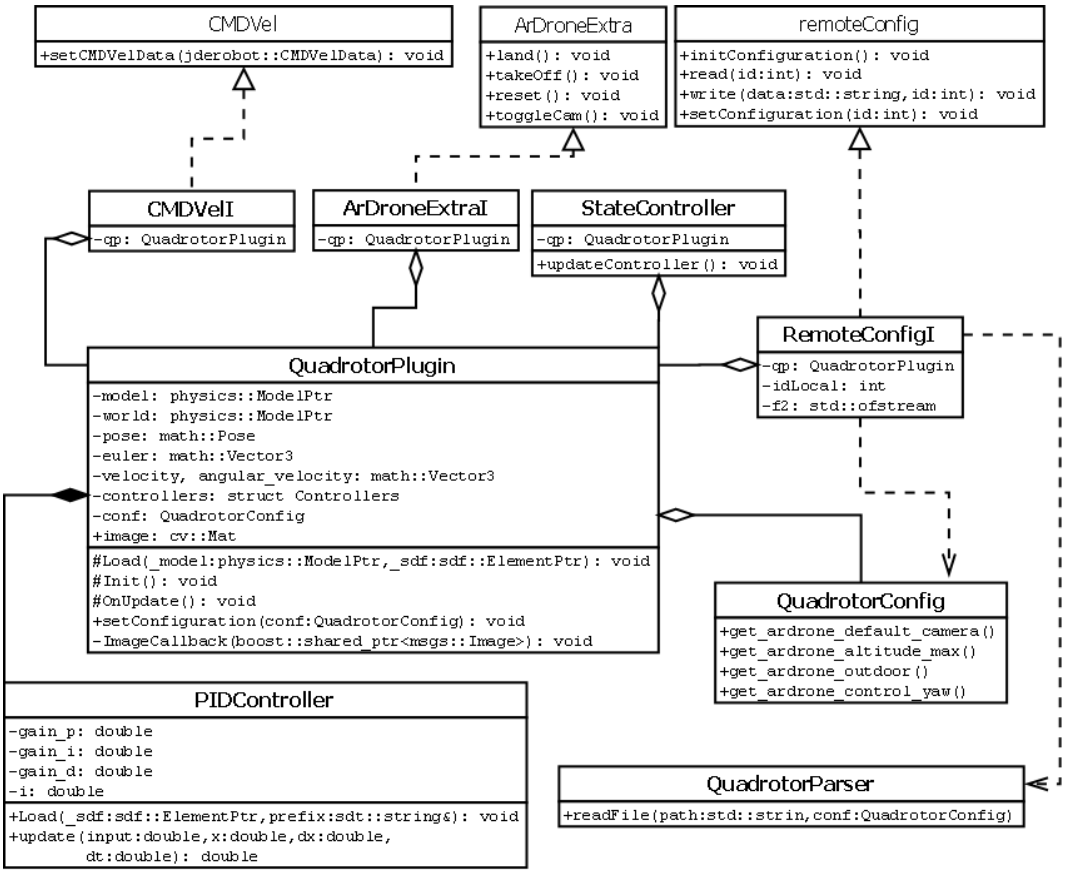


Figura 4.8: Diagrama de `QuadrotorPlugin`.

La figura 4.8 muestra el esquema *software* de los drivers para los actuadores de los cuadricópteros. El plugin que se carga es la clase `QuadrotorPlugin`, que contiene una serie de controladores PID de la clase `PIDController` para calcular las fuerzas requeridas. Las transformaciones entre las entradas y las salidas de los PID se explicó en la sección 3.4.5. Tenemos un diagrama en la figura 4.9. Las variables en rojo

hacen referencia a las órdenes de velocidad que llegan. Las variables en azul indican el resultado que se aplicará al robot simulado. Los controladores PID efectúan operaciones teniendo en cuenta el error entre una magnitud objetivo (velocidades, ángulos...) y la magnitud actual, así como la derivada temporal del error y el error acumulado. Los comandos de velocidad son traducidos a fuerzas y pares motor que Gazebo es capaz de interpretar en una siulación. La componente z de la velocidad producirá una variación en la fuerza perpendicular al plano del cuadricóptero provocando un movimiento vertical. Las componentes x e y de la velocidad se traducen a pares motor que cambian la dirección de esta fuerza perpendicular, teniendo como resultado una componente horizontal de la fuerza y, por tanto, un desplazamiento horizontal. El par motor en el eje z, responsable de los cambios de dirección del cuadricóptero, se obtiene a la salida del PID correspondiente al introducir la componente z de la velocidad angular comandada.

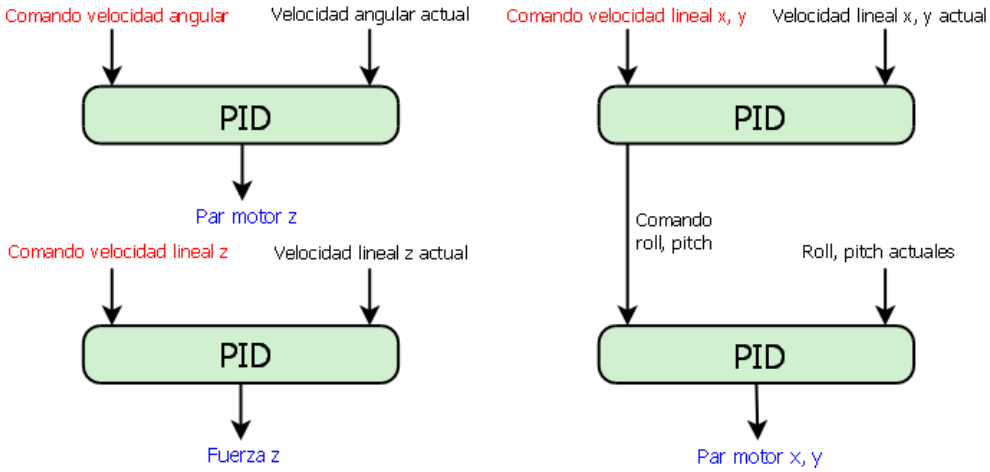


Figura 4.9: Diagrama de entradas y salidas de los PID de `QuadrotorPlugin`.

Salvo la interfaz `Pose3D`, con esta biblioteca dinámica se cargan todas las implementaciones de las interfaces ICE del cuadricóptero simulado. Al arrancar el *plugin* se crea un hilo que gestiona los objetos referentes a cada interfaz y utiliza el motor de ICE para mantener a cada uno a la escucha de llamadas remotas. Las clases `CMDVelI`, `ArDroneExtraI` y `RemoteConfigI` implementan las interfaces `CMDVel`, `ArDroneExtra` y `RemoteConfig`, respectivamente. Es necesaria también la clase `StateController`, que sirve de máquina de estados para el *drone*.

`QuadrotorPlugin` es la clase principal. Es un plugin de Gazebo que hereda de `ModelPlugin`, es decir, un *plugin* que controla la posición y el movimiento de un modelo simulado. Se encarga de mantener un control sobre la velocidad, aceleración, vectores de posición y orientación y sobre el ruido introducido a los motores para que el robot sea más similar al real, así como de realizar los cálculos necesarios para estabilizarlo y aplicar los fundamentos físicos en los que se basan los cuadricópteros. Al igual que el resto de plugins, las clases que hereden de ModelPlugin deben implementar los métodos `Load`, `Init` y `OnUpdate`. Para entender cómo actúan, es necesario introducir algunos miembros dato relevantes:

- `pose`: Miembro del tipo `math::Pose` de Gazebo en el que se guardará la posición y la orientación del *drone* simulado. Contiene a su vez dos datos: Un vector para la posición y un cuaternión para la orientación.
- `euler`: Miembro del tipo `math::Vector` de Gazebo en el que se guardará la orientación del robot en forma de ángulos de Euler, es decir, los ángulos *roll*, *pitch* y *yaw*.

- `velocity`: Del mismo tipo que el anterior. En este vector se almacenará la velocidad del robot en coordenadas cartesianas x, y y z.
- `angular_velocity`: Este otro vector contendrá las componentes cartesianas de la velocidad angular del *drone*.
- `controllers_`: Estructura de datos con seis objetos de la clase `PIDController`, uno por cada coordenada de la velocidad angular y de la velocidad lineal.
- `velocity_command`: Del tipo `jderobot::Velocities`. Representa las velocidades, tanto lineal como angular, que el *drone* debe alcanzar. Será modificada por la clase `CMDVelI` en función de los datos que lleguen de aplicaciones JdeRobot remotas a través de interfaces ICE.

La ejecución es similar a la mostrada en la sección 3.4.5, sustituyendo ROS por JdeRobot y añadiendo instrucciones que regulan el comportamiento del robot en función de las configuraciones establecidas por medio de la interfaz `RemoteConfig`, como los ángulos *roll* y *pitch* máximos o las velocidades lineal y angular máximas.

El método `Load` recibe como parámetros un puntero al modelo que ha cargado el *plugin* y otro puntero a todo el SDF desde el que ha sido cargado. Aquí se leen del fichero distintos parámetros de configuración y se guardan como variables:

- `link`: Miembro del tipo `physics::LinkPtr` de Gazebo que representa el *link* de la simulación correspondiente al cuadricóptero. Permite obtener información acerca de su estado así como modificarla.
- `world`: Miembro del tipo `physics::WorldPtr` de Gazebo que representa al mundo simulado. Será útil para obtener parámetros como el tiempo de simulación o la gravedad.
- `BodyName`: Nombre del elemento que representa al cuerpo del *drone* en Gazebo.
- `MaxForce`: Máxima fuerza (y torsión) que se puede aplicar al *drone*.
- `MotionDriftNoise`: Parámetro para calcular aleatoriamente ruido para introducirlo en la velocidad deseada. Este ruido permanecerá constante durante el tiempo indicado en `MotionDriftNoiseTime`.
- `MotionSmallNoise`: Parámetro para calcular muestras de ruido aleatorio que permanece durante menos tiempo que el anterior. Este ruido se añade a la velocidad junto con el anterior.
- `MotionDriftNoiseTime`: Indica el tiempo que permanece constante el ruido de tipo *Motion Drift*.
- `Inertia`: Vector con los momentos inerciales principales del *drone*.
- `Mass`: Masa del *drone*.

En este método `Load` se establece el estado *Initialize*, se construye la máquina de estados y se asignan las constantes a los controladores PID. También se crean y se inician el hilo que configura las interfaces ICE del *plugin* y el que lleva la máquina de estados del *drone*. Por último se configuran las comunicaciones por *topics*, creando un nodo subscriptor para recibir los mensajes con las imágenes de los *plugins* de las cámaras.

El siguiente método al cual llama Gazebo es `Init`. En él se establece la función *OnUpdate* como aquella a la que Gazebo llamará periódicamente. También se da un valor inicial a todas las variables que lo necesiten, entre ellas la velocidad objetivo (`velocity_command_`), la altitud, velocidad e inclinación máximas y el estado *Landed*, en la variable `navi_state`.

`OnUpdate` contiene el código que Gazebo ejecutará periódicamente cada ciclo de la simulación mientras el robot permanezca en ella. Es necesario llamar a este método con `ConnectWorldUpdateBegin` desde `Load` o `Init`. En `OnUpdate` se mantiene el control de las fuerzas que actuan sobre el robot, manteniéndolo en una posición fija o moviéndolo de manera realista según los principios físicos que se han explicado en las secciones 1.2 y 3.4.5. Después de calcular estas fuerzas, para aplicarlas al *drone* se invocará a los métodos `AddRelativeForce` y `AddRelativeTorque` sobre el miembro dato `link`, el cual representa dentro del código a nuestro UAV. Estas funciones toman como parámetro la fuerza y el momento que se desea aplicar, respectivamente. Esto se realiza ponderando estas variables en función del estado del *drone*, el cual es controlado por la máquina de estados implementada por la clase `StateController`. El desarrollo de `OnUpdate` es muy similar al de los *drivers* TUM_Simulator de la universidad de Munich, con la salvedad de que variables como `velocity_command_` son de tipos compatibles con JdeRobot y de que se tiene en cuenta la configuración externa por `RemoteConfig`, como se ha dicho con anterioridad. En la figura 4.10 se encuentra un diagrama de flujo de la ejecución del *plugin*.

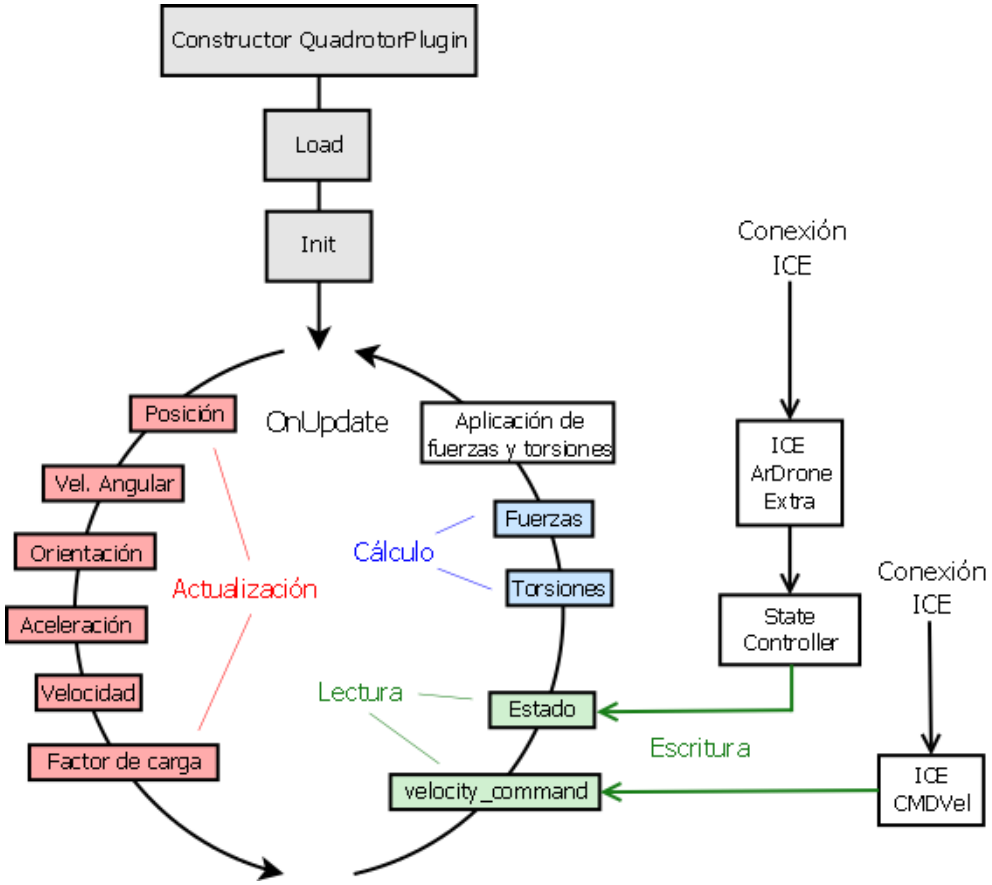


Figura 4.10: Diagrama de flujo de QuadrotorPlugin.

Ahora vamos a describir las clases que implementan las interfaces ICE necesarias. Estas clases deberán tener como atributo la misma instancia de la clase principal `QuadrotorPlugin` para modificar u obtener

alguno de sus parámetros y que el cuadricóptero reaccione.

4.3.1. Establecimiento de velocidades: `CMDVelI`

Esta clase es la que implementa el método de la interfaz `CMDVel`, que es `setCMDVelData`. Es gracias a esta clase que otras aplicaciones remotas puedan conectarse al cuadricóptero y cambiar la velocidad del *drone* sin necesidad de conocer su desarrollo.

`CMDVelI` requiere interactuar con una instancia de `QuadrotorPlugin` que será guardada con la finalidad de modificar sus datos referentes a la velocidad comandada, de tipo `CMDVelData`, utilizando para ello el método `setVelocityCommand` del *plugin*.

```
class CMDVelI: virtual public jderobot::CMDVel {
public:
    CMDVelI(QuadrotorPlugin *qp) {
        this->qp=qp;
    }
    virtual ~CMDVelI() {
        delete qp;
    }
    virtual Ice::Int setCMDVelData(const jderobot::CMDVelDataPtr& vel,
                                   const Ice::Current&) {
        qp->setVelocityCommand(vel);
    }
private:
    QuadrotorPlugin *qp;
};
```

Código 4.7: Clase CMDVelI

4.3.2. Maniobras básicas: `ArDroneExtraI`

Con esta clase se implementa la interfaz `ArDroneExtra`, la cual declara métodos útiles para despegar y aterrizar. Otros métodos dan funcionalidades adicionales al cuadricóptero, como grabar en USB el vídeo de las cámaras, pero se han dejado sin definir para la simulación ya que el objetivo es dar soporte a los actuadores y sensores básicos. Otros de los métodos extra están en fase experimental en el momento de la elaboración de esta memoria, por lo que tampoco se han definido.

En el código 4.8 podemos encontrar métodos de esta clase. Los principales son `land`, para aterrizar, y `takeoff`, para despegar, como se ha dicho anteriormente. Estos se encargan simplemente de cambiar ciertas variables de control en el objeto `QuadrotorPlugin` que contienen. El efecto de estos cambios se materializará por medio de la clase que controla el estado del cuadricóptero, llamada `StateController`, la cual está pendiente continuamente del valor de estas variables de control.

Para despegar es necesario dar el valor *true* a la variable `m_takeOff`, haciendo que la clase que se encarga de los cambios de estado provoque el cambio a *TakingOff*, cuyo efecto es un aumento en la fuerza provocada por el giro de las hélices. Transcurrido un tiempo, el estado pasará a *Flying*, momento en el

cual esta fuerza se reducirá hasta ser equivalente a la gravedad en sentido opuesto. Al despegar, la variable `m_isFlying` también debe ser establecida como *true*, puesto que es la que indica que el UAV está en el aire. Esta será modificada a *false* para aterrizar, provocando el cambio de estado a *Landing*, que producirá una reducción por debajo de la gravedad en la fuerza de los motores.

```
virtual void takeoff(const Ice::Current&) {
    pthread_mutex_lock(&qp->quadrotor_mtx);
    if (qp->navi_state == Landed ||
        qp->navi_state == Landing)
    {
        qp->m_isFlying = true;
        qp->m_takeOff = true;
    }
    pthread_mutex_unlock(&qp->quadrotor_mtx);
}

virtual void land(const Ice::Current&) {
    pthread_mutex_lock(&qp->quadrotor_mtx);
    if (qp->navi_state == Flying ||
        qp->navi_state == ToFixPoint ||
        qp->navi_state == TakingOff)
    {
        qp->m_isFlying = false;
        qp->m_takeOff = false;
    }
    pthread_mutex_unlock(&qp->quadrotor_mtx);
}

virtual void toggleCam(const Ice::Current&) {
    msgs::Vector2d msg;
    math::Vector2d v2d(0, 0);
    msgs::Set(&msg, v2d);
    pub_->Publish(msg);
}
```

Código 4.8: Clase ArDroneExtraI

4.3.3. Configuración remota: `RemoteConfigI`

Esta clase se encarga de implementar la interfaz ICE `RemoteConfig`. Mediante el método `write`, que recibe como parámetro una cadena de caracteres y un identificador de controlador, se guarda el texto enviado por la aplicación remota en un fichero en forma de XML. Una vez enviada toda la información relevante mediante las invocaciones a este método que se precisen, se llama a `setConfiguration`, cuyo parámetro es un identificador y que lee del fichero correspondiente los parámetros de configuración y los asigna a campos de la clase `QuadrotorPlugin`.

No se han tenido en cuenta todos los parámetros de configuración que existen en los controladores del

ArDrone real, debido a que los componentes JdeRobot en el momento de la elaboración de este proyecto y los que se han programado dentro de él no requieren de ellos. Estos pueden verse en un ejemplo de fichero XML de configuración mostrado en el código 4.9.

```
<?xml version="1.0"?>
<ArDroneServer>
    <Configuration>
        <ardrone>
            <default_camera>1</default_camera>
            <outdoor>0</outdoor>
            <max_bitrate>4000</max_bitrate>
            <bitrate>1500</bitrate>
            <navdata_demo>0</navdata_demo>
            <flight_without_shell>0</flight_without_shell>
            <altitude_max>3000</altitude_max>
            <altitude_min>50</altitude_min>
            <euler_angle_max>0.21</euler_angle_max>
            <control_vz_max>700</control_vz_max>
            <control_yaw>1.75</control_yaw>
        </ardrone>
    </Configuration>
</ArDroneServer>
```

Código 4.9: Ejemplo de fichero XML de configuración

El código que implementa algunas de las clases referentes a la configuración remota en este proyecto es el mismo que compone `ardrone_server`, pero no todos los atributos que permiten ser configurados en ellas influyen en la simulación. Por ejemplo, el parámetro `ardrone_outdoor` indica si el robot está volando dentro de una habitación o entorno cerrado (*false*) o se encuentra al aire libre (*true*). Otro parámetro, `ardrone_max_bitrate`, indica la tasa de bit máxima para enviar datos de vídeo. Estos atributos, entre otros, no tienen efecto en la implementación de los actuadores en simulación.

Las clases `QuadrotorParser` y `QuadrotorConfig` son el equivalente para simulación de `ArDroneParser` y `ArDroneConfig`, para el cuadricóptero real. Las primeras se diferencian de las últimas en sus nombres y en los de algunas de las varibles o métodos que las conforman, para poder compatibilizarlas con el resto de los *drivers*, pero en esencia son las mismas. El método `read` de `QuadrotorParser` se encarga de leer de un fichero XML especificado como parámetro una serie de etiquetas concretas y almacenar su valor en un objeto de tipo `QuadrotorConfig`, pasado también como parámetro a la función. La implementación de `remoteConfig` hace uso de estos dos objetos en su método `setConfiguration` (código 4.10), pasando los datos almacenados al objeto `QuadrotorPlugin` e invocando al método `configureDrone` (código 4.11), que hará que sustituya sus valores por los últimos que se le han establecido.

```
Ice::Int RemoteConfigI::setConfiguration(Ice::Int id, const Ice::Current&)
{
    if (id == idLocal){
        id=0;
        idLocal=0;
        f2.close();
```

```
        std::cout << "file completed" << std::endl;
        // aqui tienes que llamar a tu parser de xml.
        QuadrotorParser parser=QuadrotorParser(5);
        QuadrotorConfig *conf=new QuadrotorConfig();
        parser.readFile(path,conf);
        qp->setConfiguration(conf);
        qp->configureDrone();

        if(remove(path.c_str())!=0){
            std::cout << "Error deleting file" << std::endl;
        }
        return 1;
    }
    return 0;
}
```

Código 4.10: Método `setConfiguration` de `RemoteConfigI`.

```
void QuadrotorPlugin::configureDrone() {
    pthread_mutex_lock(&quadrotor_mtx);

    eulerMax=conf->get_quadrotor_euler_angle_max();
    cam_state=conf->get_quadrotor_default_camera();
    altMax=conf->get_quadrotor_altitude_max();
    altMin=conf->get_quadrotor_altitude_min();
    vzMax=conf->get_quadrotor_control_vz_max();
    yawSpeed=conf->get_quadrotor_control_yaw();
    isOutdoor=conf->get_quadrotor_outdoor();
    withoutShell=conf->get_quadrotor_flight_without_shell();
    frameSize=conf->get_quadrotor_bitrate();

    pthread_mutex_unlock(&quadrotor_mtx);
}
```

Código 4.11: Método `ConfigureDrone` de `QuadrotorPlugin`.

Los parámetros de configuración que se han tenido en cuenta afectan a los actuadores del robot limitando sus movimientos. Son los siguientes:

- `altitude_max`: Máxima altitud a la que puede llegar el *drone* en milímetros.
- `altitude_min`: Mínima altitud a la que puede llegar el *drone* en milímetros.
- `euler_angle_max`: Máximo ángulo del plano del cuadricóptero con respecto al plano horizontal (roll o pitch), en radianes.
- `control_vz_max`: Velocidad máxima en el eje z, en mm/s.
- `control_yaw`: Velocidad angular máxima en torno al eje z, en rad/s.

4.3.4. Máquina de estados: `StateController`

Como hemos visto, además de un algoritmo para obtener la fuerza y el momento necesario, es necesario otro que lleve un control sobre el estado del *drone*, ya que estas dependen del modo en el que se encuentre. Esta es la misión de la clase `StateController`, que actualiza la situación del *drone* simulado a una frecuencia determinada. El código esta extraído de los *drivers* de Munich, concretamente de la clase `GazeboQuadrotorStateController`, explicado en la sección 3.4.5, código 3.17.

El estado del *drone* se encuentra en el miembro `navi_state`, de `QuadrotorPlugin`. `StateController` cuenta con una referencia a la anterior clase, además de ciertos datos que conviene exponer antes de analizar el diagrama de estados:

- `m_takeOff`: Variable de tipo `bool` que indica si se debe despegar (*true*) o no (*false*).
- `m_isFlying`: Variable de tipo `bool` que indica si se debe permanecer en el aire (*true*) o aterrizar (*false*).
- `m_time`: Variable de tipo `double` que se usará para controlar el tiempo que se tarda en despegar o aterrizar.

De estas variables dependerá permanecer en un estado o cambiar a otro. El diagrama de estados puede verse en la figura 4.11.

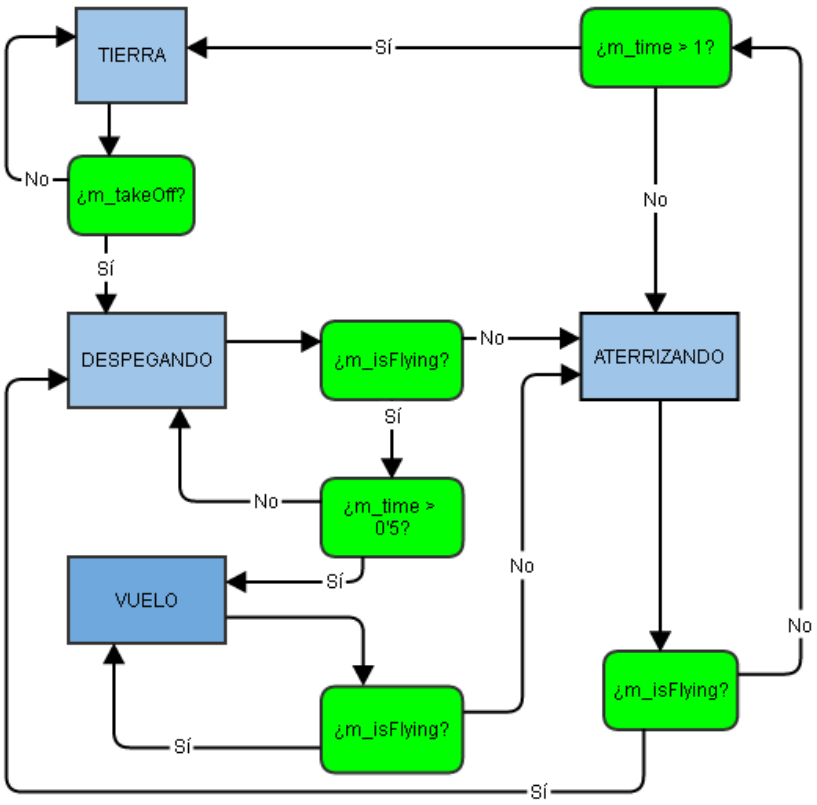


Figura 4.11: Diagrama de estados de `QuadrotorPlugin`.

El cambio del estado y la comprobación de las variables se lleva a cabo en el método `updateController`, que deberá ejecutarse periódicamente. Será tarea de la clase `QuadrotorPlugin` llamar dentro de un bucle a este método, así como modificar las anteriores variables en función de los comandos que se ejecuten remotamente. Lo primero que hace es hallar el tiempo transcurrido desde la anterior invocación. Como puede verse en la figura, tenemos cuatro estados. Partiendo del estado inicial *Tierra*, se comprueba el dato `m_takeOff` para permanecer igual o cambiar al estado *Despegando*, en el que se reinicia el contador de tiempo y se da el valor *false* a `m_takeOff`. Si al comprobar `m_time` no ha transcurrido el tiempo necesario para el despegue, se suma el intervalo de tiempo transcurrido y se permanece en *Despegando*, mientras que en caso contrario se pasa al estado *Vuelo*. Ahora se comprueba `m_isFlying` para, en caso de ser *false*, pasar a *Aterrizando*. Es posible también llegar a este estado desde otros distintos al de vuelo, comprobando siempre `m_isFlying`. Aquí, al igual que en el despegue, se reinicia el contador de tiempo y se añade un incremento de tiempo cada iteración en la que se permanezca en este estado. También se puede pasar de *Aterrizando* a *Despegando* si se da la orden de despegar antes de aterrizar completamente. Una vez que el tiempo suficiente ha transcurrido, el aterrizaje ha finalizado y volvemos al estado inicial.

Capítulo 5

Aplicaciones de navegación de *drones*

En este proyecto hemos programado varias aplicaciones que hacen uso de los *drivers* del cuadricóptero para validar la utilidad de estos. Este capítulo explica cómo se han llevado a cabo dichas validaciones.

Los comportamientos desarrollados en este trabajo son cinco: El primero consiste en que el cuadricóptero siga una serie de balizas en posiciones conocidas; el segundo, en que siga una carretera hasta el final de esta; el tercero, en que realice un barrido por un área de búsqueda y que detecte por medio de la cámara determinados objetos; el cuarto, en desplazarse por un camino conocido modificando su velocidad; el quinto, en que siga a otro cuadricóptero, que será el que adopte el comportamiento anterior. Cada comportamiento corresponde a un programa distinto y tiene sus propias clases, algunas de las cuales son comunes a varias de las aplicaciones o heredan de las de otro comportamiento. En este capítulo vamos a explicar el diseño de cada uno atendiendo a la estructura *software*, la fase de percepción sensorial y la de control.

5.1. Recorrido de secuencia de balizas de posición conocida

Aquí, el cuadricóptero deberá recorrer un camino marcado por una serie de puntos, configurados previamente en un orden determinado. La clase principal que controla las tareas generales de este apartado se llama `Travel`.

5.1.1. Arquitectura *software*

La lógica de esta aplicación se desarrolla mediante un único hilo que repite iteraciones a un ritmo controlado. Este cuenta con una fase de percepción en la que conecta con los sensores para obtener los datos que necesite, en particular, la posición absoluta del cuadricóptero en el mundo 3D. En la fase de control, calcula órdenes para los actuadores en función de los datos que ha obtenido de los sensores. Las órdenes consistirán en velocidades lineales y angulares. El bucle con estas dos fases se repite hasta que se ha alcanzado el último objetivo. La figura 5.1 ilustra este bucle de control.

Configuración

Esta aplicación, al igual que las demás, necesita ciertos parámetros que leerá de un fichero de configuración. Estos datos se corresponden, por un lado, con información necesaria para esta aplicación concreta,

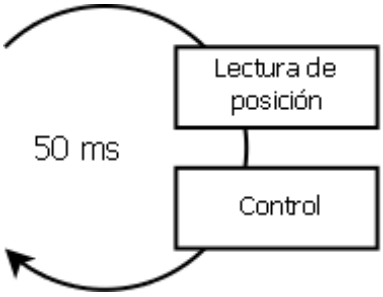


Figura 5.1: Bucle de control del recorrido de balizas con una frecuencia de 20 Hz.

como son las coordenadas de las balizas, y por otro, con información relativa al comportamiento general del *drone*, como la velocidad o la inclinación máxima, que serán necesarios en cada una de las aplicaciones de navegación de este proyecto.

Los datos dentro del fichero de configuración se representan como un árbol XML, en el que aparecen una serie de etiquetas que indicarán el ámbito de las etiquetas hijas y a qué aplicación corresponden.

Para almacenar los parámetros y hacer más fácil su acceso se han diseñado clases específicas para cada funcionalidad. La clase principal es `DroneConfig`, con el fin de que las demás hereden de ella y tengan que implementar su único método `AddInfo`, cuyo parámetro es un *array* o vector de cadenas de caracteres que contendrá los parámetros. Es importante tener en cuenta que estas clases suponen que los datos vienen en un orden concreto, por lo que será necesario respetar ese orden a la hora de escribir las etiquetas en el árbol XML. La figura 5.2 muestra el esquema general de estas clases de configuración.

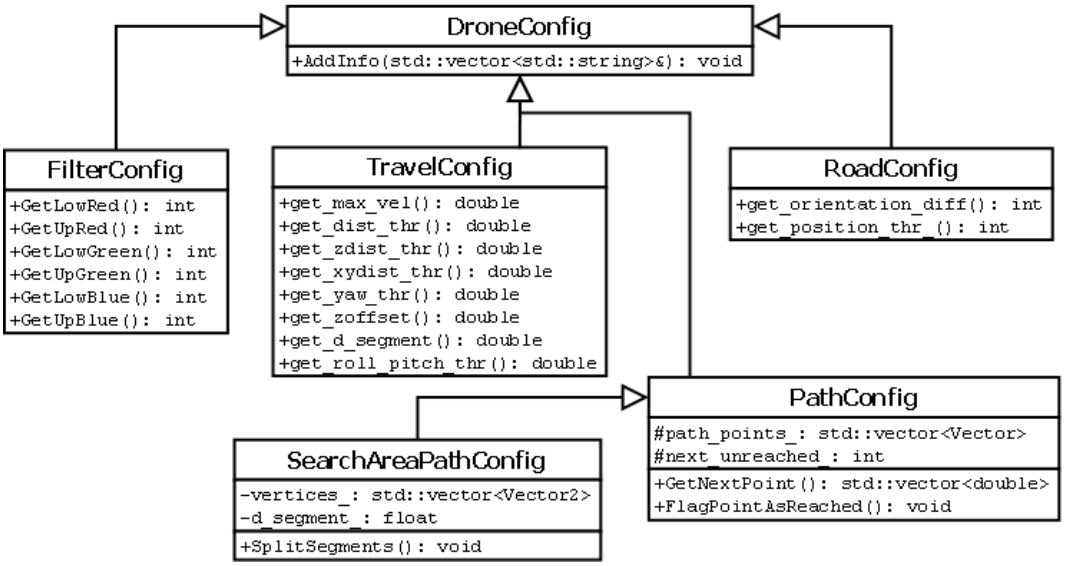


Figura 5.2: Esquema de las clases de configuración en `drone_search`.

El mecanismo para leer documentos XML lo aporta la clase `Parser`, la cual es prácticamente igual a la empleada para leer los parámetros de configuración en los *drivers* para cuadricópteros explicados en el capítulo anterior. Cada objeto de este tipo necesitará conocer la etiqueta dentro de la que buscará los datos. Si esta etiqueta no contiene etiquetas hijas, se leerá el dato que contiene; si no, se leerán los datos de las etiquetas hijas, las cuales no deberán tener a su vez otras etiquetas hijas. Para leer el documento

hay que invocar al método `ReadFile`, al cual se le pasan como parámetros la ruta del fichero que contiene la información y el `DroneConfig` en el que se desee almacenarla.

La clase que se encarga de almacenar los parámetros comunes a todas las aplicaciones es `TravelConfig`. Los elementos que contiene son los siguientes:

- `max_vel_`: Velocidad máxima en m/s que será enviada al cuadricóptero.
- `dist_thr_`: Umbral de distancia mínima expresada en metros a un punto específico. Si la distancia entre el robot y el punto que se desea alcanzar es igual o inferior a este atributo, se considera que se ha alcanzado dicho punto.
- `zdist_thr_`: Umbral parecido al anterior, pero teniendo en cuenta solo la coordenada z (altura).
- `xydist_thr_`: Umbral parecido a los anteriores en el que solo se tienen en cuenta las coordenadas x e y.
- `yaw_thr_`: Umbral de ángulo mínimo en radianes. Si se desea orientar al drone con un ángulo *yaw* concreto, se considerará que se ha alcanzado esta orientación si la diferencia entre el ángulo del drone y el ángulo objetivo es menor a este umbral.
- `zoffset_`: Valor adicional que se puede añadir a la coordenada z de los objetivos. La causa de introducir esta variable es que las coordenadas dadas por el simulador no tienen en cuenta la elevación del terreno. Por ejemplo, si queremos desplazarnos a cuatro metros de altitud pero el terreno tiene una elevación de diez metros sobre la referencia cero de Gazebo, tendremos que sumar diez metros a la coordenada que queremos alcanzar, que pasará a ser catorce sobre el punto de referencia del simulador.
- `roll_pitch_thr_`: Umbral que indica la inclinación máxima en radianes con respecto al plano horizontal que puede tener el plano del cuadricóptero para obtener datos de la cámara. La existencia de este parámetro se debe a que, si se pretende detectar la posición de objetos, la información de las cámaras no será fiable si el UAV está muy inclinado.

Todas estas variables son accesibles mediante correspondientes métodos *get*, que aparecen en la declaración de la clase en el código 5.1. Las demás clases de configuración que se describirán en las siguientes secciones tienen una definición similar.

```
class TravelConfig : public DroneConfig {
public:
    TravelConfig();
    ~TravelConfig();
    virtual void AddInfo(const std::vector<std::string>&);
    double get_max_vel();
    double get_dist_thr();
    double get_zdist_thr();
    double get_xydist_thr();
    double get_yaw_thr();
    double get_zoffset();
    double get_d_segment();
```

```
    double get_roll_pitch_thr();

private:
    double max_vel_;
    double dist_thr_;
    double zdist_thr_;
    double xydist_thr_;
    double yaw_thr_;
    double zoffset_;
    double d_segment_;
    double roll_pitch_thr_;
};
```

Código 5.1: Declaración de la clase `DroneSearch::TravelConfig`.

Para guardar las balizas y sus coordenadas hemos diseñado la clase `PathConfig`, que es hija de `DroneConfig`, por lo que cuenta con un método `AddInfo`. Cada *string* que recibe como parámetro deberá contener tres números separados por espacios que corresponden a las coordenadas cartesianas de la baliza. Las balizas se guardan en un *array* que contiene elementos de clase `Vector`. Esta clase guarda una colección de datos decimales y redefine operaciones útiles para realizar con vectores como suma, resta, multiplicar y dividir por un escalar y comparar vectores. En esta sección se ha diseñado también una clase hija de la anterior llamada `Vector3`, especial para vectores de tres coordenadas.

5.1.2. Sistema de percepción

La fase de percepción sensorial se encarga de obtener y guardar los datos necesarios para la fase de control. En este caso, las principales fuentes de información son el sensor IMU y el sensor GPS. Es necesaria la estructura `Pose3DData` para conocer la posición, la orientación actual y la distancia que separa al cuadricóptero del siguiente objetivo en 3D. Conocidos estos datos podemos calcular las velocidades que enviar al *drone*.

La posición y la orientación se extraen con `GetPose3DData`, de la interfaz ICE `pose3D`, que es invocado dentro de `GoTo`, método cuya función es dirigir al *drone* a las coordenadas 3D que se le proporcionan como parámetros. La invocación de `GetPose3DData` tiene lugar al comienzo de dicho método.

5.1.3. Sistema de control

El algoritmo para dirigir al UAV comienza consultando si quedan balizas sin alcanzar. Esto se consigue ejecutando el método `NextMovement`, que devuelve *true* si quedan y *false* si no. En este último caso habremos terminado el trayecto.

Si quedan balizas sin alcanzar se procede a calcular la distancia a la siguiente y el ángulo de giro para situarla frente al cuadricóptero. Si la distancia es menor a un umbral, se considera que se ha alcanzado la baliza y haremos uso de `PathConfig` para marcar la baliza como alcanzada con `FlagPointAsReached` y obtener la siguiente con `GetNextPoint`, que aparecen en el código 5.2.

```
std::vector<double> PathConfig::GetNextPoint() const{
```

```
    Vector beacon(0);
    if (next_unreached_ >= path_points_.size())
        return beacon.GetCoordinatesAsDouble();
    beacon = path_points_[next_unreached_];
    return beacon.GetCoordinatesAsDouble();
}

void PathConfig::FlagPointAsReached() {
    next_unreached_++;
}
```

Código 5.2: Métodos `GetNextPoint` y `FlagPointAsReached` de la clase `DroneSearch::PathConfig`.

Estas operaciones se llevan a cabo dentro de un método con sobrecarga llamado `GoTo`. Tiene dos versiones: La principal, que toma como parámetros tres coordenadas cartesianas, y otra que solo toma dos (x e y) de forma que dé igual la altura del cuadricóptero.

La distancia que nos separa del objetivo se calcula para cada una de las coordenadas y después se aplica el teorema de Pitágoras. Para calcular el ángulo de giro es necesario hallar el ángulo que forman la línea que une a la baliza con el cuadricóptero y el eje x del sistema de coordenadas de Gazebo. Una vez hallado, se resta al ángulo *yaw* del cuadricóptero. En la imagen 5.3 puede verse una representación de las variables que queremos obtener. Las coordenadas x' e y' del *drone* se restan a las coordenadas x e y del punto al que se dirige. Para calcular la componente z de la distancia el procedimiento es similar. El ángulo que hay que restar al *yaw* del cuadricóptero se obtiene aplicando trigonometría a las distancias en x e y.

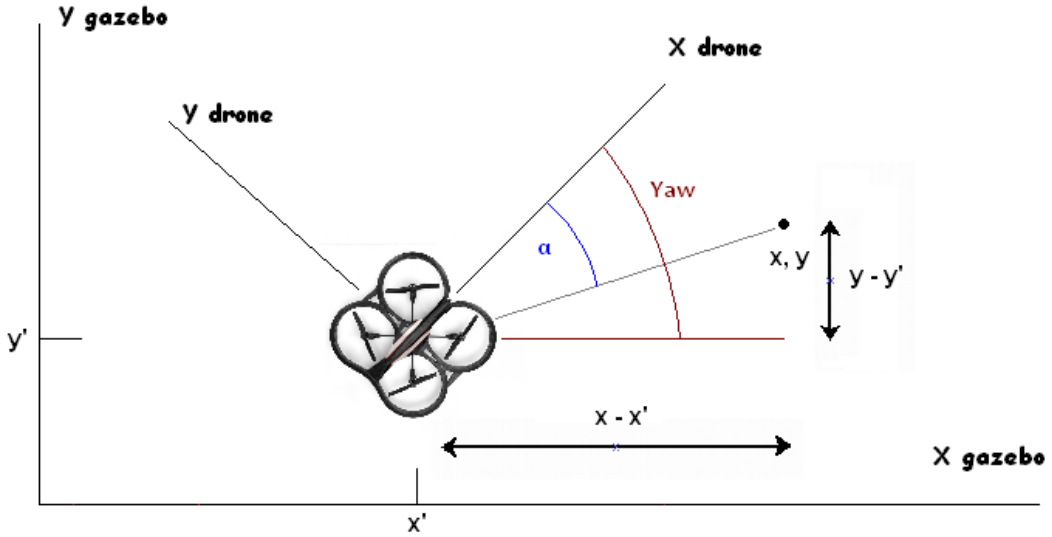


Figura 5.3: `Travel`: Distancias y ángulos para alcanzar una baliza.

La ecuación 5.1 muestra cómo se calcula la distancia en cada uno de los ejes y la distancia total. Δx, Δy y Δz son las proyecciones de la distancia sobre cada eje, d_{xy} es la proyección en el plano xy de la distancia y d es la distancia total.

$$
\begin{aligned}
\Delta x &= x - x' \\
\Delta y &= y - y' \\
\Delta z &= z - z' \\
d_{xy} &= \sqrt{(\Delta x)^2 + (\Delta y)^2} \\
d &= \sqrt{d_{xy}^2 + (\Delta z)^2}
\end{aligned}
\tag{5.1}
$$

El cálculo del ángulo de giro α se muestra en la ecuación 5.2. y_b es el ángulo que forman la recta que une el cuadricóptero con la baliza y el eje x de Gazebo, y y_d es el ángulo *yaw* del cuadricóptero.

$$
\begin{aligned}
\mathrm{y}_b &= \arctan(\tfrac{\Delta x}{\Delta y}) \\
\alpha &= \mathrm{y}_d - \mathrm{y}_b
\end{aligned}
\tag{5.2}
$$

Se aplicará una velocidad angular sobre el eje z hasta que el ángulo α sea inferior al umbral configurado. Las velocidades lineales se aplican en cada coordenada. La componente x se modula con el coseno del ángulo α y la y con el seno, de forma que el cuadricóptero se aproxime al objetivo al mismo tiempo que gira sin que la velocidad cambie de dirección en el plano *xy*. El conjunto de velocidades que se comandan al *drone* mediante la función `setCMDVelData` de la interfaz ICE `CMDVel` se muestran en la ecuación 5.3. v_x, v_y y v_z son las velocidades lineales para cada eje, ω_x, ω_y y ω_z son las velocidades angulares, V es la velocidad lineal máxima en un eje y ω es la velocidad angular máxima.

$$
\begin{aligned}
\mathrm{v}_x &= \mathrm{V} \times \cos(\alpha) \\
\mathrm{v}_y &= \mathrm{V} \times \sin(\alpha) \\
\mathrm{v}_z &= \mathrm{V} \\
\omega_x &= 0 \\
\omega_y &= 0 \\
\omega_z &= \omega
\end{aligned}
\tag{5.3}
$$

Aquí intervienen dos parámetros que se han presentado al explicar la clase `TravelConfig`: `xydist_thr_` y `zdist_thr_`. Estos umbrales se tienen en cuenta para dejar de aplicar velocidades en las componentes que ya se hayan alcanzado, evitando que se oscile si queda alguna por alcanzar. Por ejemplo, si se llega a la misma altura que la baliza siguiente pero aún queda una distancia que avanzar horizontalmente, la componente z de la velocidad será cero hasta alcanzar la baliza completamente. Las operaciones tanto para dar velocidad (`GoTo`) como para cambiar a la baliza siguiente se llevan a cabo con el método `MoveToNextPoint`.

La siguiente figura (5.4) muestra un diagrama de flujo de la aplicación de seguimiento de balizas:

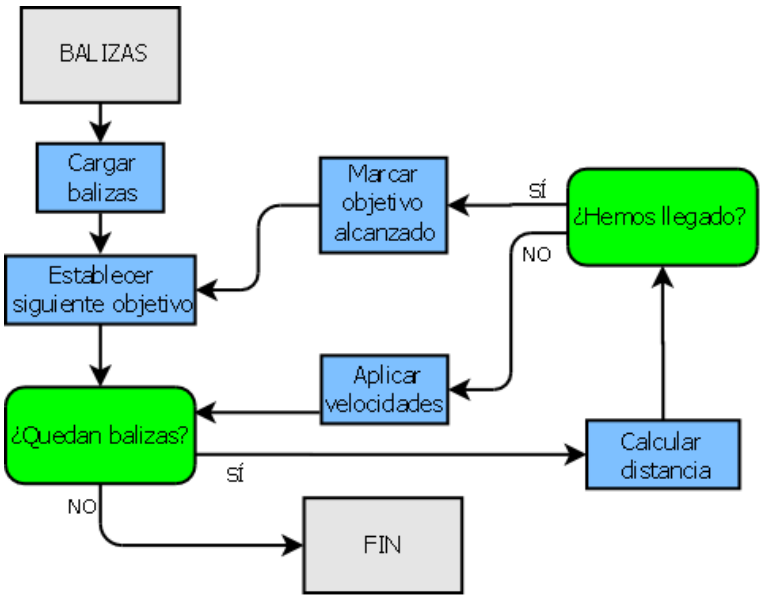


Figura 5.4: Diagrama de flujo de `Travel`.

Las siguientes imágenes, en la figura 5.5, muestran una secuencia de fotogramas de una simulación con este comportamiento funcionando satisfactoriamente. Las balizas están representadas por pequeñas bolas rojas que se han introducido en el mundo virtual por medio de un *plugin* llamado `PathPlugin`. Este carga del fichero XML las posiciones e introduce esferas con centro en cada baliza, con el objetivo poder visualizar el conjunto de balizas en la interfaz gráfica de Gazebo. El cuadricóptero aparece en algunas imágenes dentro de un círculo rojo para distinguirlo cuando se confunda con el fondo y cuando esté alejado y se vea pequeño.

(a) Recorrido de secuencia de balizas 1.

(b) Recorrido de secuencia de balizas 2.

(c) Recorrido de secuencia de balizas 3.

(d) Recorrido de secuencia de balizas 4.

Figura 5.5: Distintos fotogramas de una simulación con un ArDrone recorriendo balizas.

5.2. Navegación siguiendo una carretera

A continuación se describe el componente que tiene como objetivo que el *drone* siga una carretera hasta el final. Debe ser detectada por medio de la cámara ventral, aplicando un filtro de color a las imágenes que ofrece. La clase encargada de esta tarea se llama `RoadTravel`.

5.2.1. Arquitectura *software*

Esta aplicación tiene una arquitectura distinta a la anterior. Existen dos hilos que comparten recursos. El hilo de control tiene una función similar al del componente de seguimiento de balizas, es decir, controlar la lógica de la aplicación. Pero en este caso hemos añadido una pequeña interfaz gráfica para visualizar las imágenes percibidas por la cámara y para facilitar la depuración del código, por lo que es necesario un segundo hilo que actualice dicha interfaz. Podemos ver un esquema de esta estructura en la figura 5.6.

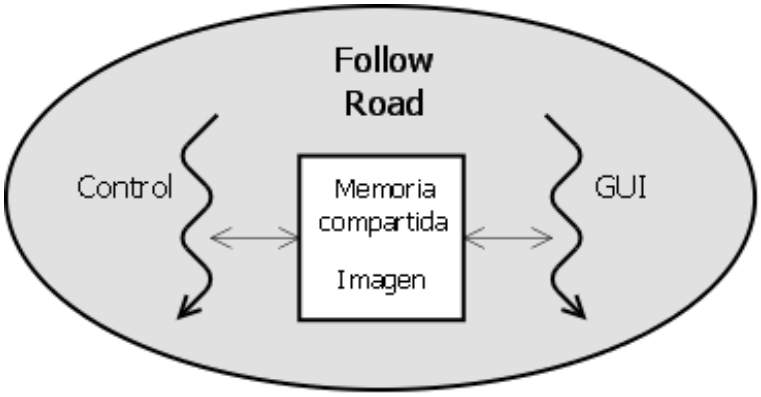


Figura 5.6: Arquitectura *software* de dos hilos: Contro e interfaz gráfica.

Periódicamente el hilo gráfico obtendrá de la memoria compartida las imágenes, mientras que el hilo de control las actualiza. La clase que representa la memoria compartida se llama `Common`, y contiene un campo para establecer las imágenes, además de los métodos `GetImage` y `SetImage` para obtenerlas y actualizarlas respectivamente.

Parámetros de configuración

Además de la configuración común a las cuatro aplicaciones, tenemos la configuración característica de esta aplicación. Los parámetros que hacen falta pueden dividirse en dos tipos: Los referentes al color de la carretera y los referentes al posicionamiento, orientación y seguimiento de la carretera. La etiqueta *filter* engloba a los primeros dentro del fichero de configuración y la etiqueta *road* a los segundos.

Hemos diseñado dos clases de configuración llamadas `FilterConfig` y `RoadConfig`. La primera almacena los rangos de color dentro de los que se encuentra el color de la carretera. Los rangos se especifican con un valor mínimo y otro máximo de rojo, verde y azul. La otra clase almacena parámetros que permiten el desplazamiento a lo largo de la carretera. Estos parámetros son los siguientes:

- `orientation_diff_`: Mide el máximo número de píxeles que puede separar en la componente horizontal de la imagen al centro de esta y al punto de referencia sin que el cuadricóptero necesite cambiar de dirección.

- `angular_vel_`: Velocidad angular durante este trayecto. Para controlar mejor la posición y evitar perder de vista la carretera, es posible que la velocidad angular sea más baja que la empleada en las otras secciones.

Para cargar los parámetros de configuración es necesario llamar al método `Configure`, que se encarga de leer los datos relevantes del fichero XML.

Otra configuración necesaria que no requiere leer datos de ningún fichero es el establecimiento de las filas que serán estudiadas dentro de la imagen. Hay tres filas principales que dividen la imagen en cuatro partes iguales. Hay también dos filas complementarias por cada fila mencionada anteriormente, que se sitúan unas pocas filas antes y después que la principal correspondiente. Esta configuración se realiza llamando dentro de `Configure` a otro método llamado `CreateRows`.

5.2.2. Sistema de percepción

La principal fuente de información para esta aplicación es la cámara, aunque también hará falta utilizar la información `Pose3D` del sensor de posicionamiento antes de obtener las imágenes, de manera que si el plano del cuadricóptero está muy inclinado no se estudiará la imagen obtenida debido a que el territorio mostrado no está justo por debajo del *drone*. Los datos `Pose3D` se obtienen de la misma manera que en la aplicación anterior (`GetPose3DData`). Para obtener la imagen hará falta invocar al método `GetImage` de la interfaz `camera`.

Los dos datos se obtienen al comienzo del método `AnalyzeImage`. Aunque en cada iteración del bucle principal se obtiene una imagen entera, para agilizar la ejecución se extrae información solo de algunas líneas configuradas de las cuales se detectarán píxeles pertenecientes a la carretera, y de los que se seleccionará un píxel objetivo o punto de referencia. Este punto servirá de guía al cuadricóptero. Sus coordenadas dentro de la imagen condicionarán las órdenes que se le enviarán.

La carretera se detecta aplicando un filtro de color a las filas de la imagen seleccionadas. Es decir, comprobando que sus valores de rojo, verde y azul se encuentran dentro de un rango especificado. Estas líneas se analizarán en un orden especificado y solamente si es necesario. Es decir, si se encuentra un punto de referencia y quedan filas sin estudiar, las que quedan no se analizan, haciendo el algoritmo más eficiente. En la figura 5.7 se pueden observar algunas líneas en azul y verde que pueden estudiarse. El círculo rojo indica el punto de referencia que se ha seleccionado como próximo objetivo.

Para representar las líneas o filas se ha creado la estructura de datos `ImgRow`, la cual contiene dos enteros que definen la posición de los píxeles de inicio y fin de la fila. Esto es necesario ya que la estructura de la imagen contiene cada fila a continuación de la anterior. `ImgRow` también contiene un identificador de línea, que coincide con su número dentro de la imagen.

Cada sección de carretera detectada en cada fila es representada por otra estructura de datos llamada `RoadSect`. Esta contiene también dos enteros que indican el píxel en el que comienza dentro de la línea y el píxel final. Además cuenta con otros tres enteros que indican el píxel central de la sección, su anchura y la fila a la que pertenece. El tipo `ImgRow` contiene una colección de `RoadSect` para almacenar las secciones que se hayan detectado dentro de una línea. La definición de estas estructuras se encuentra en el código 5.3.

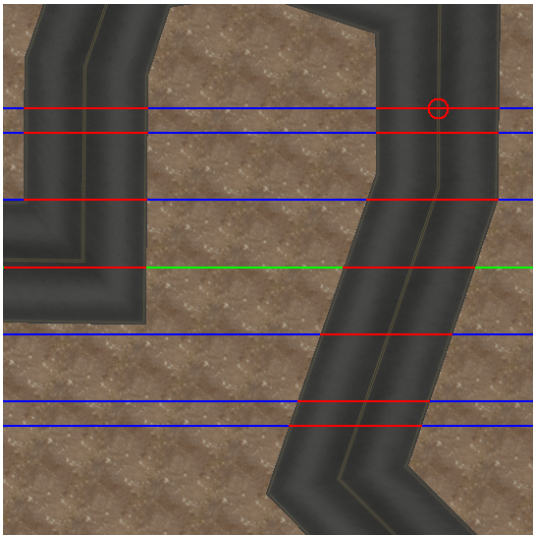


Figura 5.7: Imagen analizada en el seguimiento de carretera.

```
struct RoadSect {
    RoadSect() {
        begin = end = centrum = width = 0;
    }
    int begin, end, centrum, width, parent_row;
};
struct ImgRow {
    int begin;
    int end;
    std::vector<RoadSect> detections;
    int row_id;
};
std::vector<ImgRow> rows_;
```

Código 5.3: Estructuras `ImgRow` y `RoadSect` de la clase `DroneSearch::RoadTravel`.

El orden en que se analizan las filas no tiene por qué coincidir con su orden en la imagen. Viene dado por el orden en el que se guardan en el *array* `rows_` en el código de `CreateRows`. La prioridad de las filas a la hora de escoger el punto de referencia disminuye desde las superiores hasta las inferiores, con excepción de la fila central, que será la última en ser analizada. Esto se debe a que el centro de la imagen muestra la zona justo debajo del *drone*, la cual no nos interesa para avanzar. Aún así, tenemos en cuenta la fila central por si la carretera se desvía hacia los laterales (figura 5.8) o atraviesa la imagen de derecha a iquierda.

En cada iteración se busca línea a línea hasta encontrar un punto de referencia. Una línea se analiza píxel a píxel comprobando los valores de rojo, verde y azul. Cuando se encuentra un píxel que reune las características que nos interesan se inicia una cuenta que se incrementa por cada píxel adyacente similar que encontramos, volviendo a cero si encontramos un píxel que no reune las condiciones. Si no se ha llegado a un límite determinado no se marca el inicio de una sección. Esto permite desechar valores espurios que puedan existir.

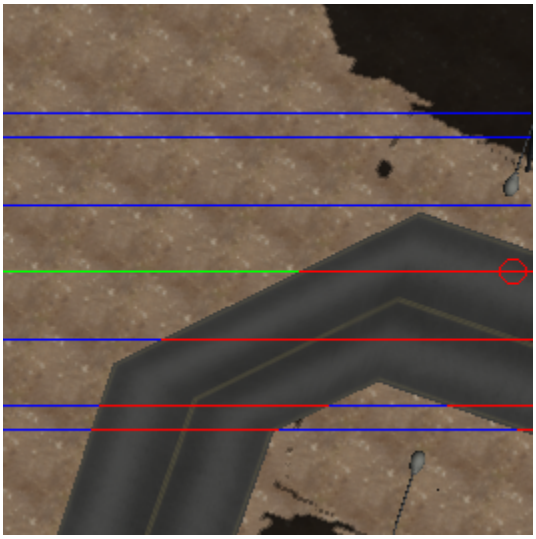


Figura 5.8: Carretera que gira hacia la derecha en la fila central. El punto de referencia se localiza en el borde al que se dirige el camino.

Si se llega al final de la cuenta se crea una sección de carretera (`RoadSect`). A partir de ese momento, se marca el final de la sección cuando se encuentra un número determinado de píxeles consecutivos fuera de los rangos establecidos, para ignorar de igual manera valores espurios dentro de la carretera. Para facilitar los cálculos de las velocidades, tanto el comienzo, el final y el centro de la sección toman un valor según un sistema de referencia cuyo origen es la columna central, es decir, píxeles en la mitad derecha de la imagen tendrán una coordenada x positiva y los que estén en la mitad izquierda tendrán una coordenada negativa. La sección de carretera se guarda en el *array* de la fila a la que pertenece. Si se encuentran más secciones dentro de la misma línea se guardan a continuación.

El método encargado de encontrar secciones en una línea dada es `GetSectionAtRow`. Como parámetros recibe la imagen, el índice de la línea y la coordenada x del anterior punto de referencia. El porqué de este último parámetro será explicado más adelante. Devuelve `NULL` si no hay secciones en la fila, o un puntero a la sección que contiene el punto de referencia escogido de entre todos los posibles.

Existen varias formas en las que la carretera puede estar dispuesta en las imágenes. Hemos tratado de contemplar las más probables e importantes, definiendo criterios a la hora de escoger el punto de referencia en cada situación. También se han definido los estados "inicio" y "en movimiento", ya que los criterios varían en función de si el cuadricóptero ha comenzado a moverse o aún no ha localizado el camino, como se verá a continuación.

Cuando se detecte una sección, se procederá a localizar en ella el punto de referencia. Si la carretera se encuentra horizontal la sección ocupará toda la fila y se asignará el punto de referencia al borde izquierdo de la imagen dentro de la misma línea. Esto no tiene una razón importante y se podría haber dado más prioridad al borde derecho sin ningún problema. Si se trata de una sección con su principio o final dentro de la imagen, el punto de referencia coincide con el centro de la sección.

En el caso de que se localicen varias secciones en la misma fila, el punto de referencia será asignado en función de la coordenada x dentro de la imagen del anterior punto. Si la sección se encuentra en la mitad superior de la imagen, se asignará el punto de referencia al centro de la sección más cercano al anterior. De esta manera no saltaremos a otro tramo o a otro objeto de color similar. Si se trata de una sección situada

en la mitad inferior, la sección con mayor peso será la más alejada al punto anterior. Esto se debe a que, si no se ha encontrado carretera en las filas superiores, es posible que haya una curva pronunciada que cambie el sentido del desplazamiento. De las dos partes del camino querremos dirigirnos a aquella por la que no hemos pasado ya. La razón de pasar como parámetro a la función `GetSectionAtRow` la coordenada x del anterior punto de referencia es que así puede escoger entre un punto u otro dentro de una misma fila.

5.2.3. Sistema de control

Hemos decidido que el cuadricóptero tiene que estar situado sobre algún tramo de carretera para poder iniciar este viaje. Si no es capaz de localizar nada en la primera imágen, esta aplicación finalizará. Los desplazamientos se producirán sin que haya cambios en la altitud del cuadricóptero. Los comandos de velocidad se envían al UAV al final del método `AnalyzeImage`. Este devuelve *false* en caso de no encontrar nada en la imagen, y *true* si queda carretera por recorrer.

En el estado inicial se analizan las imagenes recibidas antes de comenzar a moverse hacia adelante. El único movimiento, pues, que puede realizar en este estado es rotatorio, y se produce cuando el punto de referencia se encuentra en la mitad inferior de la imagen, como ocurre en la figura 5.9. El cuadricóptero girará en el sentido que más le convenga hasta que el punto se localice en la mitad superior, momento en el que comenzará a moverse hacia adelante y pasará al estado "en movimiento". Se han distinguido dos estados porque esta imagen muestra un caso en el que las acciones serán diferentes si se trata del principio del recorrido o del final. En el primer caso el cuadricóptero deberá orientarse y seguir el camino, mientras que en el segundo terminará la aplicación.

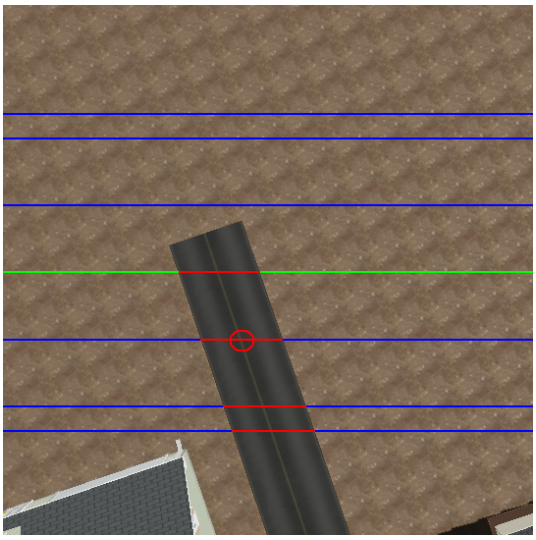


Figura 5.9: Modo inicio: El cuadricóptero deberá girar hasta que el punto con el círculo rojo se encuentre en la mitad superior de la imagen.

Una vez iniciado el viaje, las velocidades que se ordenan al *drone* tendrán valores positivos solo en la componente x de la velocidad lineal y en la z de la angular, es decir, no hay movimientos laterales. Si es necesario un giro brusco, el cuadricóptero se detendrá y girará hasta que el camino a seguir esté delante de él. La velocidad lineal es máxima cuando el punto de referencia esté justo delante, e irá disminuyendo a medida que el ángulo que forman la línea que une el punto con el centro de la imagen y el eje y de esta

aumenta.

Este estado finaliza cuando se cumplen las siguientes condiciones:

1. No hay secciones en las filas superiores.
2. Hay menos de dos secciones en las filas inferiores (solo el camino por el que hemos venido).
3. De haber secciones centrales, ninguna empieza o acaba en un borde de la imagen. Esto querría decir que nos encontramos frente a una curva que hace que la carretera salga por el centro de la imagen hacia la derecha o la izquierda.

La figura 5.10 muestra un diagrama de flujo de las distintas fases por las que pasa este algoritmo.

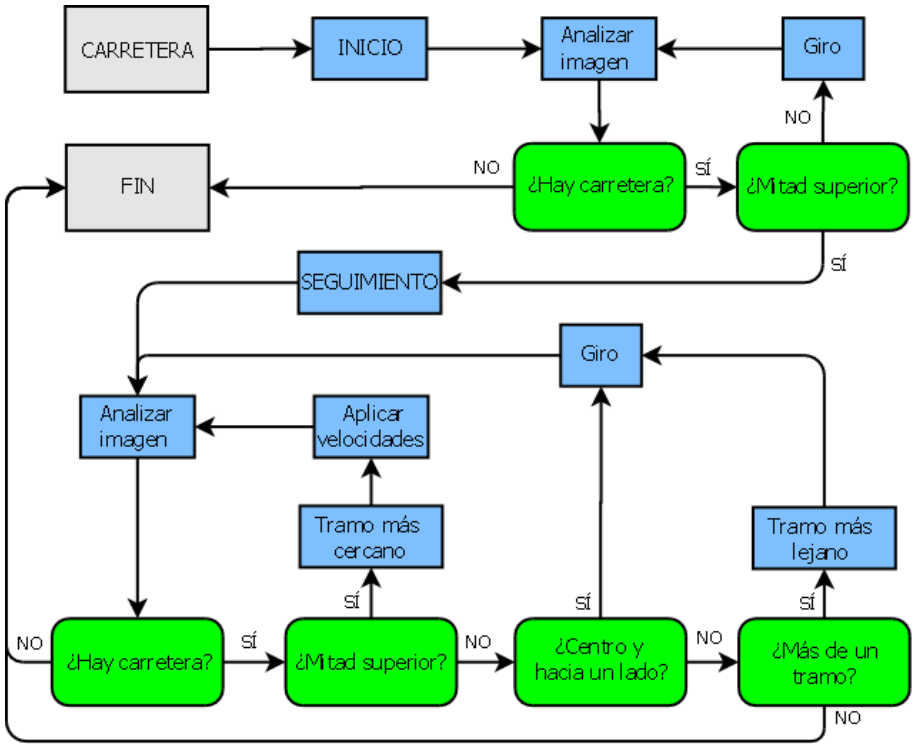
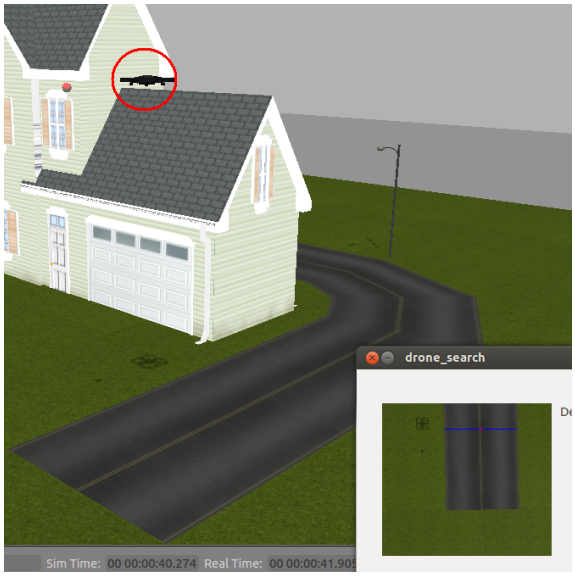
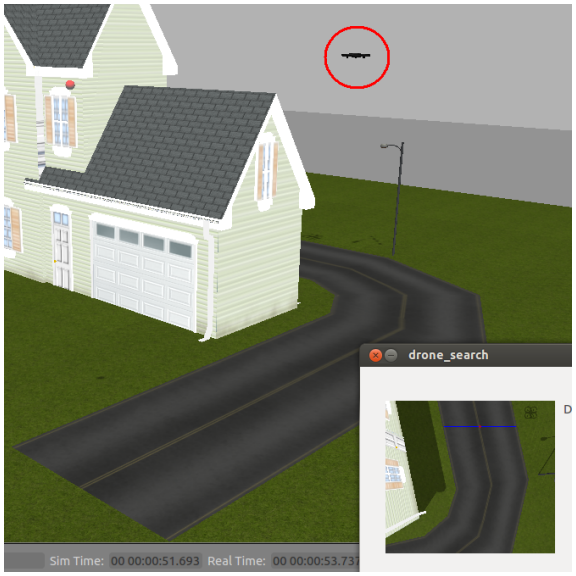


Figura 5.10: Diagrama de flujo de `RoadTravel`.

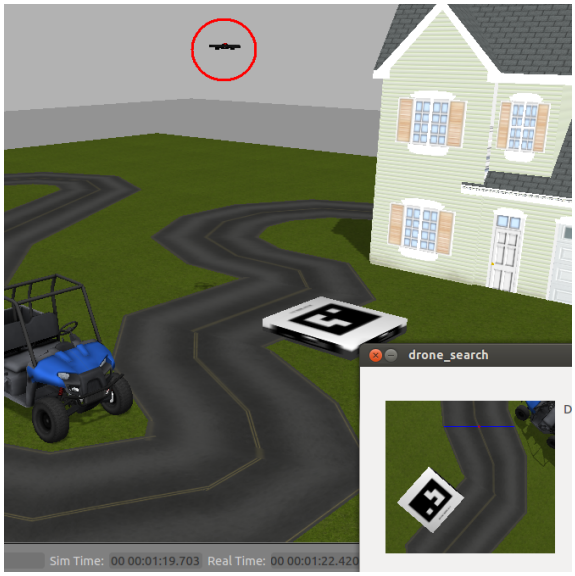
A continuación, en la figura 5.11, podemos ver una secuencia de imágenes que ilustran el trayecto a lo largo de la carretera en diferentes instantes. En la esquina inferior derecha de cada imagen podemos ver la interfaz gráfica con la imagen percibida por el cuadricóptero, la sección de carretera detectada mostrada como una línea azul y el punto de referencia como un punto rojo.



(a) Seguimiento de carretera 1.



(b) Seguimiento de carretera 2.



(c) Seguimiento de carretera 3.

(d) Seguimiento de carretera 4.

Figura 5.11: Distintos fotogramas de una simulación con un ArDrone siguiendo una carretera.

5.3. Barrido y detección de objetos

Esta aplicación tiene como objetivo explorar un área y detectar por medio de la cámara ventral objetos específicos. El cuadricóptero realizará un barrido dentro de una zona poligonal. Mientras se mueve, analizará las imágenes de la cámara ventral en busca de objetos con las características especificadas, guardando sus posiciones y las imágenes que los contienen.

5.3.1. Arquitectura *software*

La arquitectura en esta sección es igual que en el seguimiento de carretera: Existen dos hilos, uno para la interfaz gráfica y otro para el control del algoritmo. Estos hilos comparten recursos, que consisten en las imágenes obtenidas de las cámaras y las fotocapturas de los objetos localizados. El hilo de control escribe estos datos mientras que el hilo de la interfaz los lee y los muestra. La clase empleada para representar la memoria compartida es la misma que en el caso previo.

Configuración

La configuración de este algoritmo requiere especificar los vértices del polígono que define el área de búsqueda. Es igualmente necesario un parámetro de distancia cuya función será dividir las aristas del polígono en varios puntos que constituirán, ordenados de una forma que explicaremos más adelante, la trayectoria del *drone*. La velocidad puede obtenerse de configuraciones globales explicadas anteriormente.

La clase de configuración encargada de almacenar estos valores es `SearchAreaPathConfig`. La etiqueta clave del fichero XML es *vertex*. El método `AddInfo` recibirá por tanto cada uno de estos *vertex* o vértices. Dentro del fichero, dos vértices configurados de manera consecutiva constituyen un lado, de manera que a medida que añadimos nuevas etiquetas de este tipo se configuran aristas consecutivas, por lo que el orden en el que se escriben los vértices es importante.

Hay que tener en cuenta que si se construye un polígono complejo o cruzado, es decir, que contiene aristas no consecutivas que se cortan, el cuadricóptero explorará algunas zonas repetidamente además de pasar por otras que no abarca el área de búsqueda. Otro aspecto a tener en cuenta es que debe ser un polígono convexo, es decir, cuyos ángulos internos sean todos menores de 180º. Si se configura un polígono cóncavo la exploración por todo el área de búsqueda también se llevaría a cabo, pero el UAV exploraría muy probablemente espacios fuera de la zona configurada.

El polígono que abarca el área de búsqueda puede tener tantos vértices como se establezcan en el fichero, y puede ser regular o irregular.

5.3.2. Sistema de percepción

Este algoritmo se vale tanto de los sensores `pose3D` como de las cámaras, dado que esta aplicación combina la parte de navegación por posición con la detección por medio de cámaras. La información sensorial se obtiene en dos etapas. En la primera, se obtiene la posición y la orientación para poder calcular las maniobras que permitan dirigirse de un punto a otro. Una vez realizados los cálculos de movimiento, se obtienen las imágenes de la cámara ventral.

La obtención de las coordenadas 3D tiene lugar al comienzo del método `MoveToNextPoint`. Las imágenes se obtienen al iniciar el método `UpdateCameraImages`. Estas se analizarán para detectar los objetos, que contienen en su textura una etiqueta de AprilTags. Obtenida una imagen de la cámara se ralizarán varias operaciones sobre ella utilizando los métodos proporcionados por esta biblioteca para detectar dichas figuras.

Para elaborar este proceso, primero hay que convertir la imagen RGB que nos da la cámara a una imagen en escala de grises. Esto es posible con la función `cv::cvtColor` que nos proporciona la biblioteca OpenCV. Después debemos usar un `AprilTags::TagDetector` que contenga la etiqueta que queremos localizar dentro de la imagen para poder detectarla. Sobre este detector invocamos al método `extractTags`, al cual hay que pasarle como parámetro la imagen en escala de grises. Este método devuelve una colección de `AprilTags::TagDetection`. Cada uno de estos objetos representa una detección de la etiqueta seleccionada dentro de la imagen.

Para realizar estas tareas se ha creado la clase `Locator`, la cual implementa al detector y contiene métodos que procesan la imagen y las etiquetas detectadas. Hay que tener en cuenta que no se elabora procesamiento alguno en el caso de que el *roll* o el *pitch* del *drone* sea mayor a un umbral establecido para que al detectar una etiqueta en la imagen se tenga la certeza de que está situada en las mismas coordenadas x e y que el cuadricóptero.

Para compartir las imágenes con la interfaz gráfica, se añade un círculo rodeando a los objetos detectados. Este círculo es de color verde la primera vez que se localiza cada objeto. Si se localiza algo pero está lejos o ya ha sido almacenado, el círculo es de color azul.

5.3.3. Sistema de control

Para poder explorar el área especificada es necesario elaborar un camino creando varios puntos. Estos deben dividir cada arista en varios segmentos y deben ordenarse adecuadamente. El método `SplitSegments` se encargará de esta tarea. Un ejemplo del recorrido en un área rectangular puede verse en la figura 5.12. A medida que se dividen los lados en varios puntos se ordenan alternando puntos de cada lado.

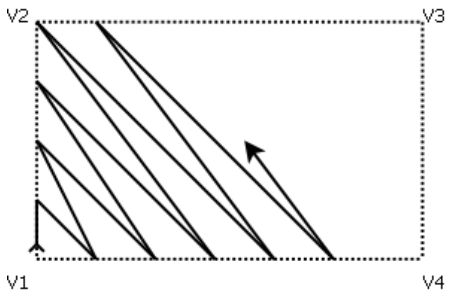


Figura 5.12: Barrido del *drone* en un área rectangular

El resultado es una sucesión de balizas muy similar a la que se usaba en el comportamiento de la sección 5.1, con la diferencia de que no se tiene en cuenta la altitud del drone. Esta se mantendrá constante y será igual a aquella con la que comience el barrido. La clase `SearchAreaPathConfig` es muy similar a `PathConfig`. El método que la diferencia de su clase padre es `SplitSegments`, encargado de dividir cada lado del polígono, intercalar cada punto del espacio y guardarlos en el orden en el que el *drone* los seguirá.

El algoritmo de división de los lados del área transcurre de este modo: Partiendo de un vértice inicial, seleccionamos los dos vértices adyacentes para establecer los dos primeros lados. Uno de esos vértices será el vértice siguiente, y el otro el anterior. Tenemos pues dos direcciones que segmentamos paralelamente. Una vez establecidos los primeros lados que se van a dividir, se entra en el bucle principal. Para crear cada punto se utiliza la siguiente fórmula:

$(x, y) = (x_s, y_s) + u \times k$

(x_s, y_s) es el vértice inicial del lado. Una vez obtenido un nuevo punto, se incrementa `k`. Al inicio de cada iteración se comprueba que `k` no sea mayor que la variable `l`, que indica la longitud del lado. En ese caso, el lado ha qedado completamente dividido y hay que pasar al siguiente, reorganizando los vértices fuente y destino. Una vez obtenido el nuevo punto de cada dirección, se introducen en `path_points_` en el orden correspondiente.

El algoritmo termina cuando, al terminar de dividir un lado, el siguiente está siendo dividido en la otra dirección, es decir, es el último lado que queda por dividir. ¿Qué ocurre entonces si ese último lado no se ha dividido del todo? No hay problema en esa situación. Si siguiéramos dividiendo en puntos el último lado los puntos finales pertenecerían a la misma recta, por lo que el *drone* no exploraría nuevo territorio. Daría media vuelta sucesivas veces hasta acabar en el último punto. En la figura 5.13 puede verse un ejemplo de lo que ocurriría si continuásemos creando puntos. El *drone* se desplazaría varias veces a lo lardo del lado S3 hasta detenerse en un punto entre sus extremos.

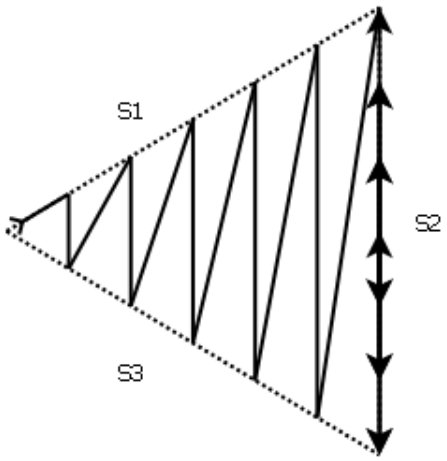


Figura 5.13: Recorrido incorrecto en el último lado.

Una vez establecido el camino que se va a recorrer, el control pasa por varias fases periódicamente: La fase de navegación por posición y la fase de análisis de imágenes. La primera es prácticamente igual que en la aplicación de seguimiento de balizas, por lo que no profundizaremos más en ella.

Tras comandar las velocidades oportunas y extraer información sobre las imágenes, se procesan los datos obtenidos. Si se han detectado objetos se analiza su posición en la imagen. Si esta se encuentra dentro de un radio cuyo centro es el de la imagen, la posición del objeto se guarda junto con una imagen del mismo. Antes de guardar ningún dato se debe comprobar que las coordenadas no coinciden o no están cerca de otras anteriormente guardadas. La imagen de la última detección hecha se muestra en la interfaz gráfica, a la derecha del flujo de vídeo (ver secuencias en la figura 5.15).

En la figura 5.14 podemos observar un diagrama de flujo de las distintas etapas de este algoritmo. Después, en la figura 5.15, tenemos una secuencia de imágenes de una simulación en la que se ha probado esta aplicación.

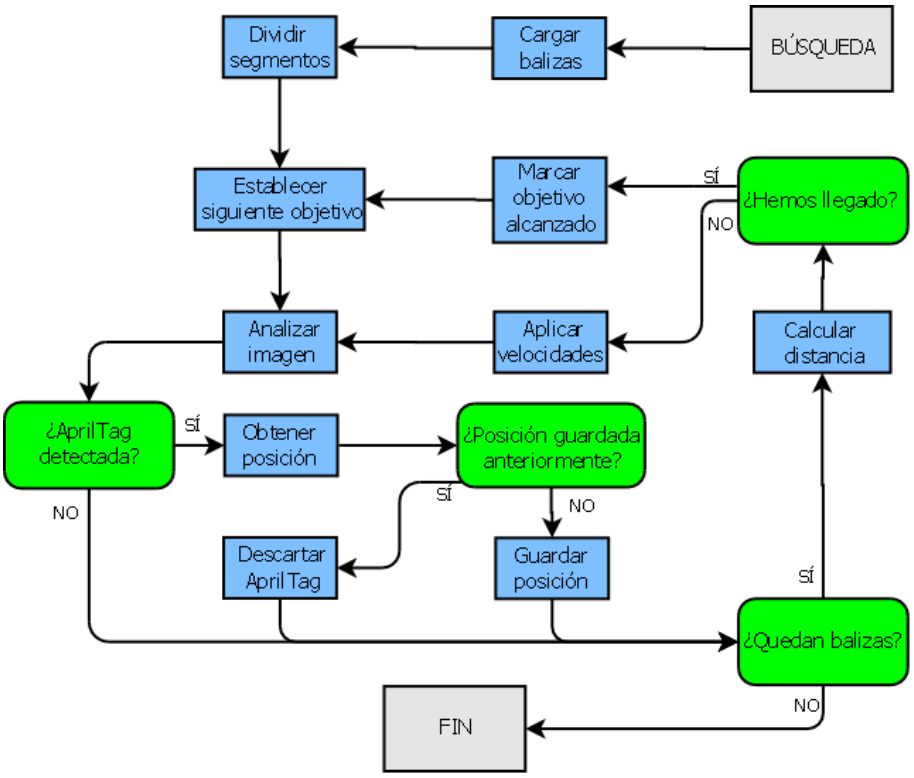
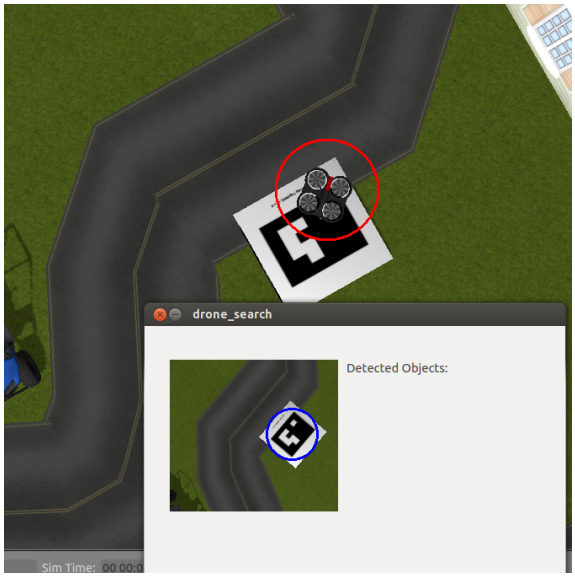


Figura 5.14: Diagrama de flujo de `Search`.



(a) Búsqueda de objetos: El *drone* detecta una etiqueta lejana con la cámara. Al no guardarse se marca con un círculo azul en la imagen del flujo de vídeo (izquierda de la interfaz gráfica).

(b) Búsqueda de objetos: Al pasar por encima del objeto, se guardan la posición y la imagen (bajo el texto "Detected Objects:") y se marca en verde en la imagen del flujo de vídeo.

(c) Búsqueda de objetos: El *drone* vuelve a sobrevolar la zona, pero no guarda nueva información ya que encontró un objeto en una posición cercana a la actual. Se marca en verde, pero la fotocaptura del último objeto detectado no varía.

(d) Búsqueda de objetos: El cuadricóptero prosigue el barrido de la zona.

Figura 5.15: Distintos fotogramas de una simulación con un ArDrone buscando etiquetas de AprilTags.

5.4. Gato persigue ratón: Componente ratón

El propósito del último comportamiento que se ha programado ha sido servir de ejemplo para la prueba que se debe superar en el concurso de programación de *drones* de la Universidad Rey Juan Carlos. En este concurso los participantes deben enviar su código para aplicar un filtro de color a las imágenes de la cámara frontal y detectar a un cuadricóptero rojo, al que llamaremos "ratón", que comenzará a moverse y al que habrá que seguir. Este código se ejecutará en el componente `introrob_py`, escrito en `python`, el cual se conectará al controlador del cuadricóptero que llamaremos "gato". El objetivo es que el gato permanezca cerca del ratón el mayor tiempo posible, teniendo este un recorrido, velocidad, aceleración, posición de inicio, etc, desconocidos para el participante.

El componente C++ diseñado como base para la prueba se llama `MouseDrone` y su código es muy similar al diseñado para el seguimiento de balizas de la sección 5.1. Se distinguen cuatro niveles: En el primero, el ratón ha despegado y se halla cernido en un punto. A medida que sube el nivel, aumenta la velocidad, siendo esta configurable mediante un fichero XML. En el mismo fichero se establece también en qué momento se pasa de un nivel a otro. A continuación describiremos el código empleado para la configuración y para la ejecución del algoritmo.

5.4.1. Arquitectura *software*

La arquitectura en este caso es de un solo hilo que controla la lógica de la aplicación, como en el caso del seguimiento de balizas.

Parámetros de configuración

La clase `PathConfig`, que almacena los puntos que marcan el camino, se ha reutilizado para configurar el movimiento de un punto a otro. Durante el transcurso del recorrido, se aumenta la velocidad que se ordena al ratón en función del tiempo, existiendo cuatro niveles. Se ha diseñado otra clase de configuración llamada `MouseConfig` en la que se guardan las variables referentes al tiempo que dura cada nivel y a las velocidades lineal y angular en cada uno de ellos,.

5.4.2. Sistema de percepción

Este componenete solo utiliza la información `pose3D`, es decir, la posición y la orientación, para desplazarse. Estos datos se obtienen al comienzo del método `MoveToNextPoint`.

5.4.3. Sistema de control

El algoritmo se basa en un bucle principal del que se sale cuando se pasa el nivel final. En el primer nivel, el *drone* ratón está cernido a un punto, con velocidad igual a cero. Dentro del bucle se calcula el tiempo transcurrido desde que inició la aplicación para compararlo con el tiempo en el que inicia el siguiente nivel. El ratón se mueve más deprisa a medida que pasa el tiempo.

Aparte del control de velocidades y de nivel, el funcionamiento es igual que el de la aplicación de balizas. Se ha reutilizado la clase `Travel`. Hemos implementado un nuevo método llamado `CheckTimesAndRunFaster`,

el cual sube al siguiente nivel si ha transcurrido el tiempo necesario y cambia las velocidades. Este método es invocado justo antes de ordenar velocidades al *drone*.

Cuando se pasa el tiempo del último nivel, la aplicación termina y el ratón se detiene.

5.5. Gato persigue ratón: Componente gato

La parte complementaria al componente anterior es la que conecta al UAV negro, es decir, al gato, y le da las órdenes necesarias para que persiga al ratón. Esta aplicación se ha realizado para proporcionar un seguimiento inicial que permita medir la dificultad de la prueba del concurso de programación de *drones*.

El algoritmo deberá localizar al ratón, de color rojo, del cual se desconoce su posición inicial, por lo que probablemente haya que buscarlo. Una vez localizado, mediante un filtro de color a la imagen de la cámara frontal, se extraerá la posición del ratón dentro de la imagen y se estimará la distancia en función del área que ocupe. Con estos datos, habrá que calcular la velocidad con la cual moverse para mantenerse a una distancia prudente, de manera que no perdamos nuestro objetivo si cambia de dirección rápidamente ni nos mantengamos a demasiada distancia si aumenta su velocidad.

5.5.1. Arquitectura *software*

A pesar de emplear la información de la cámara frontal, no se ha utilizado interfaz gráfica para depurar. La arquitectura se basa en un solo hilo de control. Este actualiza periódicamente los datos de la cámara frontal para detectar objetos relevantes, analizar sus características y ordenar una velocidad al *drone* gato en función de la información obtenida.

Parámetros de configuración

Para la configuración del filtro de color se ha creado la clase `CatConfig`, en la cual se almacenan los rangos que utilizará el filtro de color. La diferencia con `FilterConfig` (seguimiento de carretera) es que no se guardan valores mínimos y máximos de RGB, sino de HSV (*Hue*, *Saturation* y *Value* - Matiz, Saturación y Valor). Estos valores mínimos y máximos se leerán de un fichero XML de configuración. Otros parámetros útiles para este componente son:

Además de los valores mínimo y máximo de las componentes de color que queremos detectar, son útiles las siguientes variables:

- `min_area_`: Área mínima que puede tener el ratón detectado en la imagen sin que el gato se mueva hacia adelante.
- `max_area_`: Área máxima que puede tener el ratón detectado en la imagen sin que el gato se mueva hacia atrás.
- `min_area_limit_`: Área mínima bajo la cual se descartarán los objetos detectados en la imagen.
- `max_area_limit_`: Área máxima sobre la cual se descartarán los objetos detectados en la imagen.
- `velX_`: Componente x de la velocidad lineal que aplicaremos al gato cuando haya que moverlo hacia adelante.

- `velY_`: Componente y de la velocidad lineal que aplicaremos al gato cuando haya que moverlo lateralmente.
- `velZ_`: Componente z de la velocidad lineal que aplicaremos al gato cuando haya que moverlo verticalmente.
- `angular_vel_`: Velocidad angular que aplicaremos al gato cuando haga falta girar sobre el eje z.

5.5.2. Sistema de percepción

Como ya se ha comentado, la fuente de información más importante para la tarea de perseguir al ratón es la cámara frontal. Las imágenes se obtienen del mismo modo que en las aplicaciones anteriores. También es necesaria información de orientación para no tener en cuenta las imágenes si la inclinación es muy alta, como en los casos anteriores. La estructura `pose3D` se obtiene antes de obener el fotograma correspondiente para evitar la tarea si se da el caso.

El algoritmo de detección aplica un procesamiento a la imagen consistente en un suavizado para eliminar ruido, la conversión del formato RGB al HSV, un filtro para umbralizar la imagen y extraer el contorno de la parte de la imagen que nos interesa. Después se aproximará a un polígono, del cual se podrán extraer las coordenadas (fila y columna en la imagen) del centro y su área. La figura 5.16 muestra un diagrama de este proceso.

Figura 5.16: Diagrama del procesamiento de imagen en el componente gato.

Las funciones utilizadas para el procesamiento de imagen pertenecen a `opencv`. El código empleado puede verse a continuación (5.4). Antes que nada, se abandona el procesamiento si el plano del cuadricóptero se encuentra muy inclinado. Una vez convertido a un polígono, guardamos sus características en un objeto de tipo `ObjectDetected`. Entre estas se encuentran la posición dentro de la imagen y el área, valores de los cuales nos valdremos para calcular las velocidades. Si se han detectado varios polígonos se selecciona el de mayor área.

```
void Cat::Execute() {
    // Image Processing
    ...

    cv::Mat frame = sensors_->GetImage();
    cv::Mat imgTh,hsv;
    cv::GaussianBlur(frame, frame, cv::Size(3, 3), 0, 0, cv::BORDER_DEFAULT);
    cv::cvtColor(frame, hsv, CV_BGR2HSV);
    cv::inRange(hsv, cv::Scalar(hue_min_, sat_min_, val_min_),
                cv::Scalar(hue_max_, sat_max_, val_max_), imgTh);

    cv::blur(imgTh, imgTh, cv::Size(7, 7));
    cv::Mat threshold_output;
    std::vector<std::vector<cv::Point> > contours;
```

```
std::vector<cv::Vec4i> hierarchy;

/// Detect edges using Threshold
cv::threshold(imgTh, threshold_output, edge_threshold_, 255, cv::THRESH_BINARY);
/// Find contours
cv::findContours(threshold_output, contours, hierarchy, CV_RETR_LIST,
                 CV_CHAIN_APPROX_SIMPLE, cv::Point(0, 0));

/// Approximate contours to polygons + get bounding rects and circles
std::vector<std::vector<cv::Point> > contours_poly(contours.size());
std::vector<cv::Rect> boundRect(contours.size());

...
```

Código 5.4: Comienzo del método `Execute` de `Cat`: Procesamiento de imagen y detección de objetos.

5.5.3. Sistema de control

Para establecer las velocidades hay que tener en cuenta las coordenadas del objeto detectado y su área. A la hora de obtener las coordenadas x (columnas) e y (filas) del polígono se realiza un cambio de sistema de referencia. El que utiliza `opencv` tiene su origen en el vértice superior izquierdo de la imagen y las coordenadas coinciden con el número de columna y de fila. Para facilitar los cálculos, se ha cambiado a un sistema con el origen en el centro de la imagen y cuyas coordenadas x aumentan junto con las columnas pero cuyas coordenadas y aumentan al disminuir las filas. En la imagen 5.17 podemos ver la diferencia entre estos dos sistemas. Es un cambio similar al realizado en el seguimiento de la carretera, pero cambiando también la componente y.

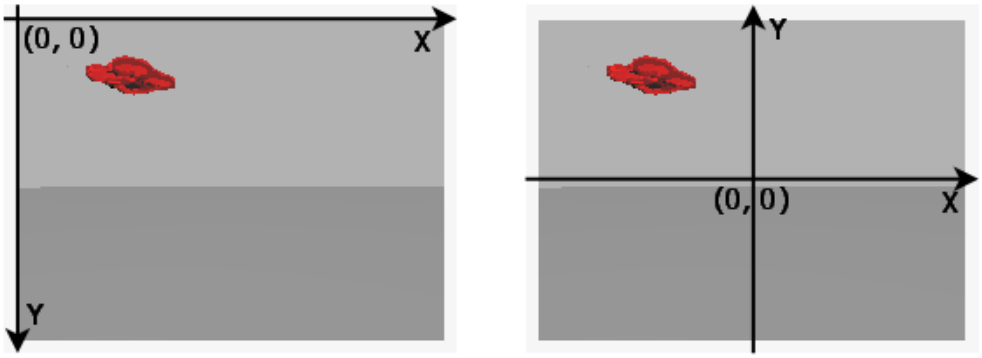


Figura 5.17: Izquierda: Sistema de referencia utilizado en las imagenes `cv::Mat`. Derecha: Sistema de referencia modificado en el componente `CatDrone`.

Las velocidades se comandan al gato si las coordenadas del objeto detectado en la imagen superan unos umbrales llamados bandas muertas, específicos para cada componente de velocidad. De esta manera evitamos que el gato tenga movimientos oscilatorios para centrar a su objetivo cuando este no se ha alejado significativamente.

La velocidad lateral se multiplica por un factor que toma su valor máximo, uno, cuando la coordenada y está muy alejada del origen, haciendo que esta velocidad solo tenga importancia si la desviación del

ratón es alta. Igual comportamiento tiene la velocidad angular en torno al eje z. Podemos ver una gráfica de la componente y de la velocidad lineal y de la componente z de la velocidad angular en función de la posición en el eje x de la imagen en la figura 5.18.

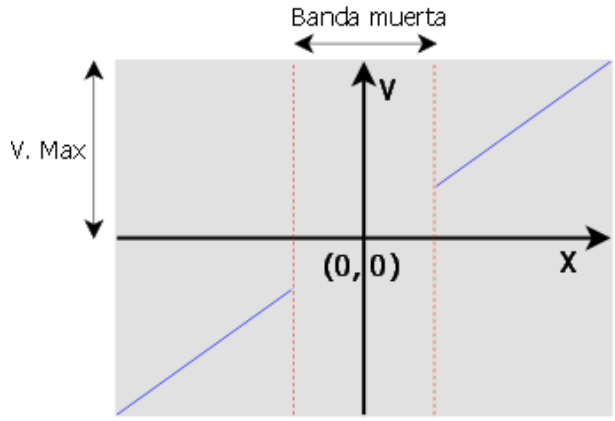


Figura 5.18: Función de la componente y de la velocidad lineal y la z de la angular para el *drone* gato.

La componente x de la velocidad, es decir, la que mueve al *drone* hacia adelante, es constante si el área del polígono es menor a `min_area_`, y toma el valor dado por la variable `velX_`, aunque se aumenta en un 50 % cuando el área detectada es menor que la mitad del área mínima. Si el *drone* gato se encuentra muy cerca del *drone* ratón, se desplazará lentamente hacia atrás. En la figura 5.19 podemos ver una gráfica de esta velocidad en función del área detectada.

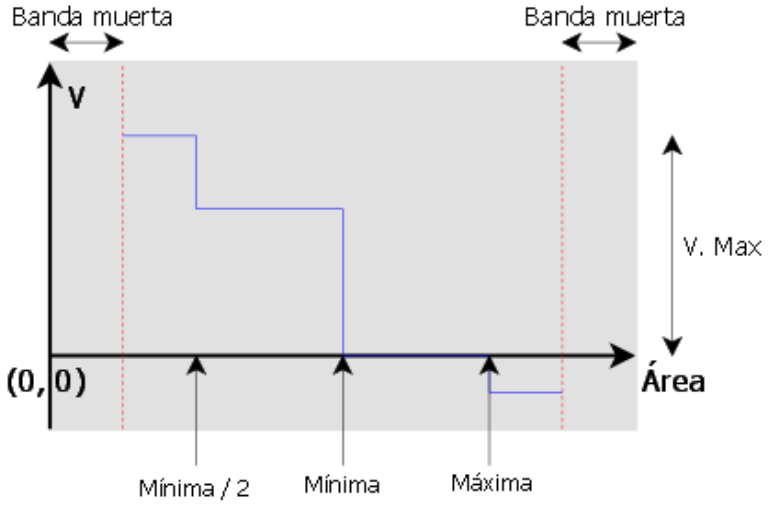


Figura 5.19: Función de la componente x de la velocidad lineal para el *drone* gato. Los valores de área mínima y área máxima definen el intervalo de áreas entre las que debe estar el área detectada para no ordenar velocidad.

La velocidad vertical es constante, cambiando de sentido en función de si el ratón se mueve hacia arriba o hacia abajo. En la figura 5.20 podemos encontrar la gráfica de esta velocidad en función de la coordenada y de la detección en la imagen.

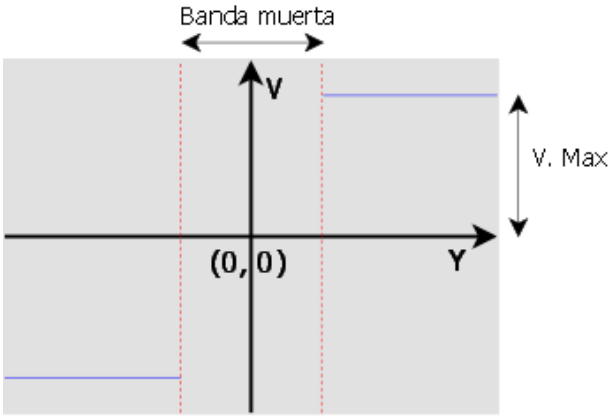
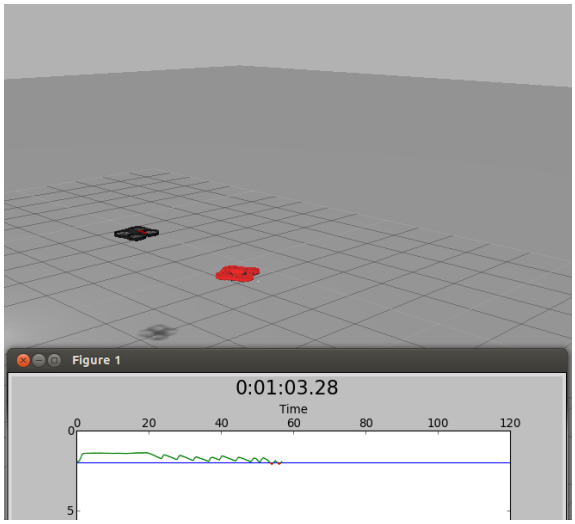
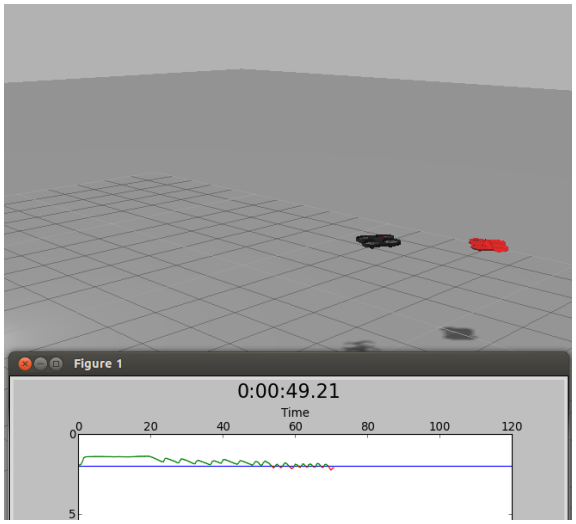


Figura 5.20: Función de la componente z de la velocidad lineal para el *drone* gato.

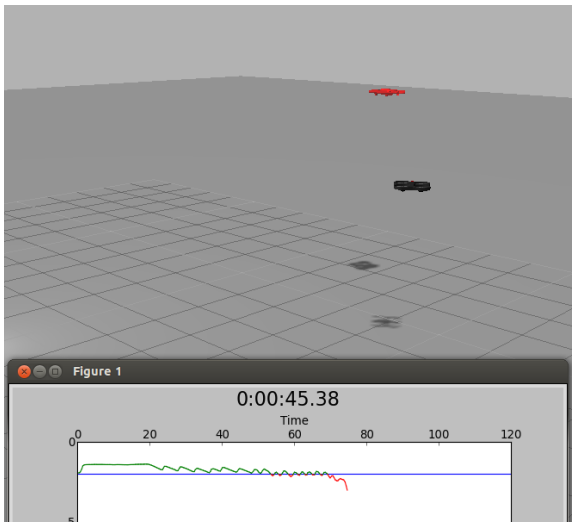
A continuación se muestra una secuencia de fotogramas de una simulación con los dos componentes, gato y ratón, funcionando (figura 5.21). En la gráfica inferior de cada imagen se muestra la distancia entre los dos cuadricópteros. A medida que pasa el tiempo el ratón se mueve más rápido y al gato le cuesta más mantenerse cerca.



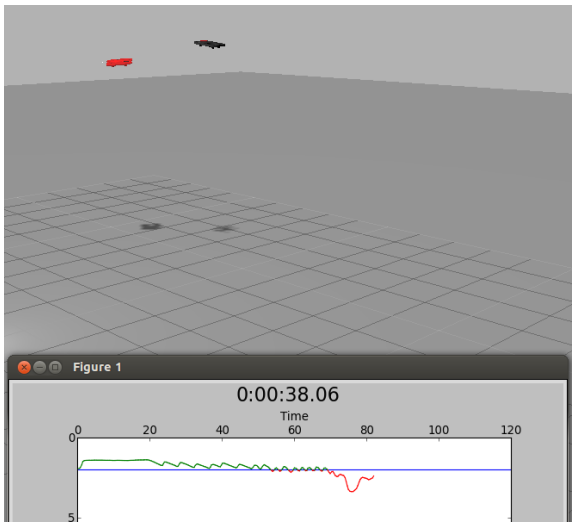
(a) Gato persigue ratón 1.



(b) Gato persigue ratón 2.



(c) Gato persigue ratón 3.



(d) Gato persigue ratón 4.

Figura 5.21: Distintos fotogramas de una simulación con un ArDrone negro (gato) persiguiendo a uno rojo (ratón).

Capítulo 6

Conclusiones y trabajos futuros

En los capítulos previos hemos explicado el desarrollo fundamental de este trabajo. Para finalzar esta memoria, vamos a verificar que se han cumplido los objetivos iniciales y las posibles líneas de trabajo que partan de este proyecto.

6.1. Conclusiones

Después de todo lo expuesto en esta memoria, podemos afirmar que el objetivo principal de dar soporte JdeRobot para cuadricópteros en Gazebo se ha logrado satisfactoriamente. Vamos a extraer conclusiones sobre cada subobjetivo superado:

- Los *drivers* para cuadricópteros simulados han sido elaborados en su totalidad y cumplen las funcionalidades establecidas en cuanto a recubrimiento para sensores y actuadores. Ha sido posible teleoperar al ArDrone con la exactitud y el realismo que puede proporcionar las leyes físicas de Gazebo, así como utilizarlos con componentes diseñados previamente cuyo resultado en simulaciones es comparable al que se obtiene aplicándolos al ArDrone en la realidad, concretamente con `uav_viewer` y `object_tracking`. Los cambios necesarios para alternar el control del *drone* real al simulado son simplemente especificar en qué puertos acepta las conexiones ICE cada uno en ficheros de configuración. El *software* diseñado puede ser empleado por cualquier cuadricóptero con dos cámaras, GPS y sensor IMU.

  El trabajo de este proyecto ha requerido un estudio en profundidad del simulador Gazebo, tanto de su API C++ como de su arquitectura *software*, el conjunto de bibliotecas de las que depende y la construcción de mundos y modelos. Ha sido necesario también el estudio del lenguaje de marcado URDF, necesario para la transformación del modelo ArDrone de la Universidad Técnica de Munich en un modelo SDF que pueda ser útil a cualquier programador independientemente de su conocimiento del entorno ROS. Hemos diseñado mundos con casas, carreteras, desniveles, coches y señales, entre otros objetos, para recrear un entorno más realista en el que controlar a nuestro *drone*.

  Los *drivers* se diseñaron inicialmente para funcionar en la versión 1.8 de Gazebo, pero dada la inversión que recibe la OSRF, el desarrollo y la mejora del simulador se produce rápidamente, dejando

anticuada esta versión. Es por eso que los *drivers* han sido adaptados a las nuevas versiones de Gazebo, funcionando correctamente en la 4.1 y la 5.0.

Ha sido necesario analizar y comprender los *plugins* elaborados por la Universidad Técnica de Munich, los cuales plasman en código C++ el cálculo de las fuerzas generadas por los actuadores del cuadricóptero bajo los principios en los que se basa su movimiento, explicados en la sección 1.2. Para ello ha hecho falta aprender la mecánica de ROS, entorno en el que están desarrollados. Gracias a estas tareas ha sido posible remodelar los programas al entorno JdeRobot, desechar las partes innecesarias en este proyecto y añadir otras como la posibilidad de realizar configuraciones dinámicas, alternar cámaras y emplear la sencilla y escalable estructura de datos `Pose3D`.

Además de la compatibilidad de los controladores desarrollados con otros programas se ha demostrado su capacidad para emplearse en actividades académicas como el curso de programación de *drones*[1] en el que las aplicaciones desarrolladas por los alumnos se conectaban al cuadricóptero en Gazebo mediante el componente JdeRobot `introrob_py`. Son los controladores que se emplean en la nueva edición del curso y en el campeonato de programación de *drones*[2] cuya prueba es diseñar el comportamiento de un cuadricóptero gato que persiga a un cuadricóptero ratón, similar al comportamiento explicado en la sección 5.5.

- Además del trabajo anteriormente mencionado hemos desarrollado cinco aplicaciones de navegación que demuestran que es posible valerse de un simulador para diseñar programas robóticos. Las acciones y movimientos del ArDrone se han ejecutado como se esperaba en cada uno de los componentes. Estos abarcan distintas ramas como navegación visual, navegación por posición y algoritmos de detección visual.

  La aplicación para navegación por seguimiento de balizas conocidas ha demostrado ser una vía adecuada para mover al cuadricóptero a lo largo de puntos 3D fijos automáticamente. Esto puede emplearse en el mundo real conociendo las latitudes y longitudes correctas, útil si no tenemos el alcance suficiente con control remoto para transportar mercancías, dirigir al *drone* a un área donde llevar a cabo otra tarea o incluso como blanco móvil.

  El componente de seguimiento de carreteras incluye un amplio número de posibilidades en cuanto a qué disposiciones de la carretera bajo la cámara que pueden darse. El filtro de color puede ser configurado para seguir otros tipos de asfalto, caminos, etc, y continuar con el trayecto incluso si se perciben materiales cerca de este que sean del mismo color. El hecho de que las carreteras en Gazebo no permitan recrear curvas, sino cambios bruscos con picos cuando se produce un cambio de dirección, y que el *drone* haya podido completar el recorrido, nos permite deducir que ante caminos reales, con giros más suaves y curvas definidas, la tarea podrá llevarse a cabo con mayor facilidad que en las simulaciones de este proyecto.

  La localización de objetos en un área de búsqueda ha mostrado la posibilidad de crear trayectorias complejas dados algunos puntos perimetrales, distinguir información relevante y almacenarla, siendo susceptible de enviarse para reconocimiento por otros robots o usuarios que requieran conocer a qué ubicaciones dirigirse o si existen determinados agentes o características en un territorio, como estructuras dañadas, gente extraviada o supervivientes de desastres.

[1] `http://jderobot.org/Programacion-de-drones`
[2] `http://jderobot.org/Campeonato-de-drones`

Los componentes gato-ratón han demostrado la posibilidad de que varios cuadricópteros interactúen entre sí, aportando otro ejemplo de control visual. Ha quedado verificada la capacidad de localizar no solo objetos fijos, como en la aplicación de detección en un área de búsqueda, sino un objeto móvil al mismo tiempo que el *drone* se mantiene cerca de él. El componente gato ha demostrado reaccionar ante imprevistos como cambios de velocidad y dirección del *drone* ratón.

- La validación experimental de los *drivers* para cuadricópteros y de las aplicaciones de navegación se ha llevado a cabo con éxito tras el ajuste de distintos parámetros de configuración y realizar pruebas para distintos casos.

  Los *drivers* se han amoldado para operar correctamente con `uav_viewer`, el componente sigue-carretera, recorrido del área de búsqueda, etc. Se han reimplementado en diversas ocasiones tras realizar pruebas con distintas posiciones y orientaciones iniciales, modificaciones de trayectorias imprevistas o incluyendo ruido a los motores.

  Las aplicaciones de navegación, por su parte, han requerido de análisis de la actuación del UAV ante situaciones como distintas orientaciones de la carretera, la presencia de obstáculos en ella, o la ampliación de polígonos concretos simples para el área de búsqueda, como cuadrados, a otros más generales, irregulares, de número de aristas indefinido y lados no paralelos.

  Los controladores de este proyecto están siendo empleados en otros como el trabajo de fin de grado de Andrés Hernández [28], en el que una tarea consiste en posar al cuadricóptero sobre vehículos en movimiento. También son usados en el trabajo fin de máster de Alberto López-Cerón Pinilla [29], cuyo propósito es la incorporación de técnicas de construcción de mapas y modelos 3D de los escenarios por los que vuela.

Si atendemos a los requisitos de este proyecto, podemos decir que se han satisfecho todos ellos:

- Todas los programas e implementaciones se han construido dentro del entorno JdeRobot.
- Las interfaces ICE han sido las mismas que tienen los *drivers* JdeRobot para el cuadricóptero real. Además el modelo empleado se asemeja con fidelidad al ArDrone real.
- Visto el funcionamiento realista realizado por el cuadricóptero en simulación y el comportamiento mostrado con varias aplicaciones de JdeRobot, este se asemeja al generado en el robot real.
- Las aplicaciones desarrolladas son computacionalmente eficientes, permitiendo que el ritmo de las simulaciones difiera mínimamente del tiempo real transcurrido.

Finalmente, queda decir que los controladores que hemos elaborado se han incorporado al repositorio GIT oficial de JdeRobot [3] con el objetivo de que cualquiera pueda hacer uso de ellos.

6.2. Trabajos futuros

Es posible implementar más sensores de los que hemos atendido (cámara, IMU y GPS). El cuadricóptero real, cuya versión simulada hemos utilizado, es capaz de aportar mayor información sensorial. El objetivo ha

[3] `https://github.com/RoboticsURJC/JdeRobot`

sido que nuestros controladores puedan funcionar con otros modelos de cuadricóptero, que no tienen todos las mismas capacidades. Sin embargo, se pueden construir otros *plugins* adicionales que envíen información de otros sensores que puedan usar e incluirlos si el modelo con el que se trabaje dispone de dicho sensor. Estos sensores pueden ser sónares, magnetómetros, sensores de infrarojos, de ultrasonidos o medidores de batería restante, entre otros. Los datos recibidos pueden escribirse en una estructura `NavdataData`, la cual se proporciona con nuestros controladores pero con muchos de sus campos vacíos.

La interfaz `remoteConfig` también puede ser implementada de manera que haya más parámetros configurables, como la tasa de bit máxima o las altitudes mínima y máxima, velocidades y aceleraciones diferentes en caso de volar en interiores o exteriores.

Otro punto que mejorar es la forma en la que el *drone* aterriza, ya que actualmente desciende durante unos segundos y después se detiene, no teniendo en cuenta la altitud. Combinando el aterrizaje con un sónar, es posible dar la orden de aterrizar desde cualquier altitud.

Existen algunos métodos en la interfaz `ardrone_extra` no implementados por los controladores para el ArDrone real en el momento en que se ha elaborado este proyecto. Estos tampoco han sido implementados por los controladores del simulado. Otra línea de investigación es la implementación de dichos métodos para simulación, ya sea antes o después de que estén listos para el UAV real.

Las aplicaciones de navegación también abren líneas de investigación, como por ejemplo combinar el seguimiento de una carretera con la localización de objetos. Esto permitiría detectar accidentes de tráfico, obstáculos que entorpezcan la circulación o añadir la funcionalidad al cuadricóptero de radar aéreo. Una línea futura posible es la detección de obstáculos para el seguimiento de balizas, puesto que es posible que haya objetos que se interpongan entre una baliza y otra.

Dados los componentes elaborados, se abren diversas vías en simulación de cuadricópteros y programación de cuadricópteros reales. Se abre la posibilidad de investigar con otros algoritmos para manejar al UAV. Ahora es posible emplear el simulador Gazebo en futuros proyectos de JdeRobot con *drones* como fase previa a incorporar algoritmos a cuadricópteros reales.

Bibliografía

[1] www.rae.es

[2] www.allonrobots.com/types-of-robots

[3] tecnologia.elpais.com/tecnologia/2014/03/07/actualidad/1394190950_809529.html

[4] Asimov, Isaac (1983). “4 The Word I Invented". Counting the Eons.

[5] Robert Kirk. “Robots”.

[6] Carnegie Mellon. The Robotics Institute. www.ri.cmu.edu/index.html

[7] UAV Universe. sites.google.com/site/uavuni/

[8] playerstage.sourceforge.net/gazebo/gazebo.html

[9] Gazebosim - gazebosim.org

[10] www.ros.org

[11] www.gtk.org

[12] www.zeroc.com/ice.html

[13] Alberto Martín Florido. Navegación visual en un cuadricóptero para el seguimiento de objetos. Trabajo Fin de Grado, Grado en Ingeniería de Computadores, Universidad Rey Juan Carlos, 2013-2014.

[14] Óscar Higuera Rincón. Xpider: Desarrollo de un Vehículo Aéreo No Tripulado de Cuatro Motores. Proyecto Fin de Carrera, Ingeniería Informática, Universidad Rey Juan Carlos, 2010-2011.

[15] gazebosim.org/wiki/Tutorials/1.5/plugins/overview

[16] www.jderobot.org

[17] Hongrong Huang, Juergen Sturm. TUM Simulator. Computer Vision Group, Technical University of Munich.

[18] http://es.wikipedia.org/wiki/Qt_%28biblioteca%29

[19] http://spectrum.ieee.org/automaton/robotics/humanoids/darpa-robotics-challenge-here-are-the-official-details

[20] http://spectrum.ieee.org/automaton/robotics/robotics-software/osrf-prepares-for-darpa-virtual-robotics-challenge

[21] Borja Menéndez Moreno. Programación de humanoides con autómatas de estado finito usando JdeRobot. Trabajo Fin de Máster, Máster Universitario Oficial en Sistemas Telemáticos e Informáticos, Universidad Rey Juan Carlos, 2013-2014.

[22] www.kivasystems.com/solutions/

[23] www.irobot.es/robots-domesticos/aspiracion

[24] https://www.youtube.com/watch?v=cRTNvWcx9Oo

[25] www.parrot.com/es/productos/bebop-drone/

[26] www.amazon.com/b?node=8037720011

[27] http://jderobot.org/Programacion-de-drones

[28] Andrés Hernández. Air Project. http://jderobot.org/Andresjhe-tfg

[29] Alberto López-Cerón. VisualSLAM over a drone. http://jderobot.org/Alopezceron-tfm