INGENIERÍA DE TELECOMUNICACIÓN - INGIENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Escuela Técnica Superior de Ingeniería de Telecomunicación

Año Académico 2016-2017

Proyecto Fin de Carrera

Seguimiento de rutas 3D por un drone con autolocalización visual con balizas

Autor: Manuel Zafra Villar

Tutor: Jose María Cañas Plaza

Real stupidity beats artificial intelligence every time.

- Terry Pratchett

Agradecimientos

En primer lugar quiero agredecer a mi tutor Jose María su dedicación, apoyo y motivación. Gracias por confiar en que puedo llevar este proyecto más allá de su objetivo principal.

A mi familia por su total apoyo no sólo a lo largo de este proyecto, sino de todo el viaje que ha supuesto realizar la carrera.

A Clara y Cristina, por poder contar con su hacha.

Y a Dani, por sacarme una sonrisa cuando más lo necesitaba.

Resumen

El objetivo de este Proyecto de Fin de Carrera es el de diseñar e implementar un sistema de vuelo autónomo para drones en espacios interiores. Debido a las características del medio el sistema de control debe ser muy preciso, por lo que no se podrán usar técnicas de localización como GPS. Para ello se hace uso de la visión artificial, mediante la cual se obtienen resultados de gran precisión.

El sistema de control desarrollado se basa en el control del giro del drone para seguir una ruta definida. El algoritmo de control predice la posición del vehículo y ajusta la velocidad angular del mismo en base al futuro error con respecto a la ruta. Además, el sistema incluye una interfaz gráfica con la que monitorizar el movimiento del drone en la que se muestra un mundo en 3D en el que se dibujan la ruta y la posición del drone, las imágenes captadas por la cámara y una serie de botones que permiten la teleoperación del mismo. El sistema se ha validado experimentalmente realizando estudios del comportamiento en entornos simulados. Adicionalmente, se han llevado a cabo experimentos en entornos reales para dar nuevas perspectivas sobre las futuras versiones del sistema.

Para el desarrollo del proyecto se ha adaptado un componente de autolocalización basado en un sistema de visión artifical fiducial. Se ha refactorizado el software para ajustarlo al propósito del proyecto y se ha validado experimentalmente.

El presente trabajo se ha desarrollado en el sistema operativo Ubuntu 16.04, en la plataforma software de robótica JdeRobot. El componente se ha implementado con el lenguaje de programación Python. El simulador en el que se han realizado los experimentos ha sido Gazebo.

Contenidos

Figuras

Capítulo 1

Introducción

La ciencia ficción y la robótica siempre han ido de la mano y gracias a esto se ha conseguido interesar a cada vez un mayor número de personas. El avance actual de esta tecnología dista mucho de haber evolucionado hasta el punto de los relatos narrados en libros y películas, sin embargo, se están alcanzando logros que años atrás se habrían antojado imposibles. Estamos viviendo un momento en el que la robótica comienza a arraigar en nuestra cultura, estando presente en prácticamente todos los ámbitos de nuestra vida. El amplio crecimiento de las nuevas tecnologías está cambiando nuestras estructuras sociales y económicas, modificando nuestros hábitos y obligando una adaptación al rápido avance de la tecnología.

El abaratamiento de los componentes hardware y el desarrollo de las tecnologías asociados a estos está permitiendo que cada vez más personas puedan tener acceso a dispositivos robóticos con mayor facilidad. Ya sea sean pequeñas empresas que adquieren un robot con funciones de vigilancia o producción, como cualquier particular que lo obtiene con un mero objetivo de ocio y entretenimiento. En concreto la robótica aérea está teniendo gran repercusión en los últimos años gozando de una gran divulgación debido a los numerosos usos a los que se puede aplicar.

Otro de los factores que ha popularizado la robótica y ha otorgado a esta un gran abanico de posibilidades ha sido el avance en la visión por computador. El desarrollo de

nuevos tipos de cámaras digitales y procesadores más potentes ha facilitado el estudio de técnicas innovadoras que permiten un mayor rango de precisión y utilidad. Esta evolución permite a los robots percibir y analizar su entorno de una forma cada vez más rigurosa.

El trabajo realizado y definido en esta memoria está incluido en los campos de la robótica aérea y la visión artificial. En este primer capítulo se dará contexto a este ámbito de la tecnología, necesario para entender los objetivos y el desarrollo del presente proyecto, definiendo los conceptos más importantes y explicando el desarrollo y estado actual de estos campos científicos.

1.1 Robótica

La robótica es una rama de la tecnología que se dedica al diseño y a la creación de artefactos y máquinas que despeñen trabajos y operaciones llevados a cabo por seres humanos. Esta tecnología ha permitido grandes avances en no sólo tareas que antes realizaban los humanos, sino en tareas que hasta ahora no era posible llevarlas a cabo. Como ejemplo tenemos la exploración de Marte llevada a cabo por el astromóvil *Curiosity*, enviado al planeta en una misión dirigida por la NASA y mediante la cual se está pudiendo estudiar las características del planeta de una forma que hasta ahora habría resultado imposible. Otro ejemplo lo encontramos en el robot *DaVinci*, una plataforma quirúrgica con la que se consiguen realizar operaciones con gran precisión y mucho menos invasivas que con la cirugía clásica.

El término robot nació en 1923 en la obra de Karel Čapek, un escritor checo que jugó con el significado de "robota", cuyo significado en su lengua materna es el de trabajo forzado o servidumbre. Más tarde Isaac Asimov acuñó este término definiendo la robótica como la ciencia que estudia a los robots y creando así las famosas *Tres Leyes de la Robótica*. Un robot es esencialmente una computadora provista de sensores y actuadores, programada para reaccionar e interaccionar con el medio que lo rodea a partir de la información obtenida por sus sensores.

El origen de esta ciencia lo podemos situar en el siglo XVIII cuando Joseph Jacquard inventa una máquina textil programable mediante tarjetas perforadas. Durante la Revolución Industrial se impulsó este tipo de instrumentos mecánicos, divulgando su uso y estableciendo así las bases de los modelos de producción industriales actuales. A pesar de esto, los primeros robots complejos de comenzaron a producir en la decada de los 50, donde la investigación en inteligencia artifical desarrolló maneras de emular el procesamiento de información humana con computadoras electrónicas e inventó una variedad de mecanismos para probar sus teorías.

Actualmente el incremento en la potencia de los procesadores y los grandes avances en los distintos tipos de infraestructuras físicas, así como en dispositivos hardware y desarrollo de software, ha permitido un uso muy diverso de la robótica:

- **Industria:** Se utilizan tanto para realizar trabajos peligrosos o de gran dificultad para un humano como puede ser la aplicación de sustancias nocivas, el moldeado de materiales o el transporte pesado; como para tareas de inspección y control de calidad mediante visión artificial y sistemas mecánicos. Además, el uso de robots conlleva una mejora de calidad y un gran aumento de la productividad.

- **Medicina:** Se han desarrollado dispositivos que permiten realizar desde trabajos quirúrgicos guiados por imágenes hasta cirugía mínimamente invasiva realizada mecánicamente por un robot. También podemos encontrar robots asistenciales para personas que necesitan una supervisión y cuidado continuo, prótesis robóticas que van desde la sustitución parcial de alguna parte dañada del cuerpo hasta exoesqueletos. Otro ejemplo estaría en la robótica terapéutica, utilizada como medio de rehabilitación fisiológica y en tratamientos para enfermedades como el Alzheimer.

- **Investigación:** En los laboratorios se utilizan para realizar tareas repetitivas de medición y control de calidad y desempeñar trabajos peligrosos para los humanos como puede ser la manipulación de sustancias dañinas. También en la investigación espacial se hace un gran uso de la robótica, lo que permite investigar entornos que para un ser humano serían prácticamente imposibles, como el caso del robot *Curiosity* mencionado anteriormente.

(a) Robot Curiosity

(b) Robot DaVinci

Figura 1.1: Aplicaciones de Robótica en Investigación y Medicina

- **Ocio:** En los últimos años se ha producido una integración de esta tecnología en eventos de cultura, deporte y ocio. Podemos ver ejemplos de esto en eventos deportivos en los que se aplica la realidad aumentada, en el desarrollo de efectos especiales de la industria audiovisual o en las nuevas generaciones de consolas y videojuegos, en los que se hace amplio uso de la visión por computador.

- **Seguridad:** La robótica ha dado al mundo de la seguridad y la vigilancia una nueva perspectiva, siendo la visión artificial el eje en torno al que giran estas aplicaciones. La automatización de estas tareas permite una mayor facilidad y eficiencia a la hora de ejecutar esta labor, podemos encontrar drones que vigilan grandes concentraciones de personas, cámaras en seguridad vial que analizan el tráfico o el uso doméstico de estas para la prevención de accidentes.

- **Educación:** La robótica ha surgido como un recurso didáctico innovador que favorece la construcción de conceptos y conocimientos de distintas disciplinas, no únicamente las tecnológicas o científicas. Utilizando esta tecnología como factor de motivación a partir del interés para llevar al alumno al desarrollo de su propio conocimiento.

- **Agricultura:** Todavía no son muy comunes los robots que trabajan en agricultura, pero a medida que pasa el tiempo se vuelven más y más populares. La tecnología

robótica aplicada al sector agrícola se encuentra en un estado de desarrollo avanzado debido a la necesidad de aumentar la producción sin aumentar los recursos al mismo tiempo que se minimiza el impacto ambiental.

(a) Robot en agricultura

(b) Brazo robótico industrial

Figura 1.2: Aplicaciones de Robótica en Industria

- **Industria militar:** En muchas ocasiones los mayores avances en materia de tecnología se han realizado durante periodos de guerra. Actualmente existe una gran gama de vehículos terrestres sin piloto humano con funciones de reconocimiento e incluso algunos vehículos armados. El mayor desarrollo en cuanto a robótica militar está siendo llevada a cabo en la robótica aérea, donde estos sistemas han pasado de ser unidades de apoyo a unidades primarias de ataque.

1.2 Drones

El comúnmente llamado dron es un *Vehículo Aéreo No Tripulado* (VANT), en inglés Unmaned Aerial Vehicle (UAV), estos vehículos no cuentan con ningún tipo de tripulación y son capaces de mantener de forma autónoma el vuelo controlado. El pilotaje puede realizarse mediante un operador humano con control remoto o controlado de forma autónoma por software. Hasta hace poco, los drones eran de uso exclusivo en el ámbito militar y en centros de investigación. En los últimos años, la miniaturización de sensores y su electrónica, han permitido la comercialización, diversificación y popularización de numerosos

modelos a precio asequible. Su pequeño tamaño, la gran estabilidad de algunos modelos y el tiempo de autonomía, los hacen ser idóneos para ciertas tareas hasta ahora impensables.

Los drones se comenzaron a desarrollar con fines militares después de la primera guerra mundial, y se emplearon como objetivos de práctica para los cañones antiaéreos en la segunda guerra mundial. A pesar de esto, hasta finales del siglo XX no se comenzaron a usar UAVs totalmente autónomos debido a los obstáculos existentes en cuanto a complejidad de software y hardware. En los años 90 gracias a la disponibilidad del sistema de posicionamiento global (GPS) y de las comunicaciones satélite se liberó a los UAVs de operar dentro del alcance de las señales de radio y de los sistemas de navegación basados en giroscopios y datos de aire. Esto propició el avance en tecnologías que garantizaban la autonomía de los vehículos. Aunque el uso militar de los drones lleva teniendo lugar desde principios de siglo, no ha sido hasta esta última década que se ha extendido el uso de drones civiles

Existe una gran variedad en cuanto a tipos de drones, pudiendo clasificarse en tres grandes grupos: los que operan de forma similar a un avión mediante propulsión y superficies sustentadoras; aerostatos, que poseen uno o más recipientes llenos de un gas más ligero que el aire y son propulsados por motores; y los que tienen un funcionamiento similar al de los helicópteros siendo propulsados por uno o más rotores. En este caso nos centraremos en los llamados cuadricópteros que son propulsados por cuatro rotores y son el modelo que más se ha popularizado.

(a) Aerodino Nimbus

(b) Aeronave Predator

Figura 1.3: Ejemplo tipos de drones

La fisonomía típica de estos drones consta de un cuerpo situado en el centro rodeado de cuatro hélices, dos delanteras a derecha e izquierda y otras tantas en la parte trasera. La propulsión se produce mediante el giro de estas hélices que generan una fuerza de empuje vertical, así, según la diferencia entre la fuerza aplicada por los rotores y el peso del cuadricóptero se consigue aumentar, reducir o mantener la altura (Fig. 1.5 : e y f). Para el movimiento horizontal, ya sea hacia delante/atrás o derecha/izquierda, se reduce la fuerza del par de rotores en la dirección hacia la que se quiere dirigir y se aumenta en el par opuesto (Fig. 1.5 : a, b, c y d). Además, estas diferencias de potencia deben ser proporcionales para poder así mantener la altura de vuelo. De esta forma el movimiento se consigue alterando la dirección de la fuerza de sustentación con respecto a la fuerza de gravedad.

Figura 1.4: Cuadricóptero

El movimiento de giro se consigue de forma distinta a los helicópteros tradicionales, que al solo poseer una hélice principal necesitan una hélice de apoyo perpendicular a ésta para contrarrestar la torsión generada por el rotor principal. En el caso de los drones el problema de la torsión se solventa asignando el sentido de giro de las hélices de forma opuesta entre rotores que están situados de forma contigua. Así, si se aumenta la potencia de los rotores con un sentido de giro y se disminuye en los rotores con el sentido contrario, se consigue el movimiento de giro del cuadricóptero (Fig 1.5 : g y h).

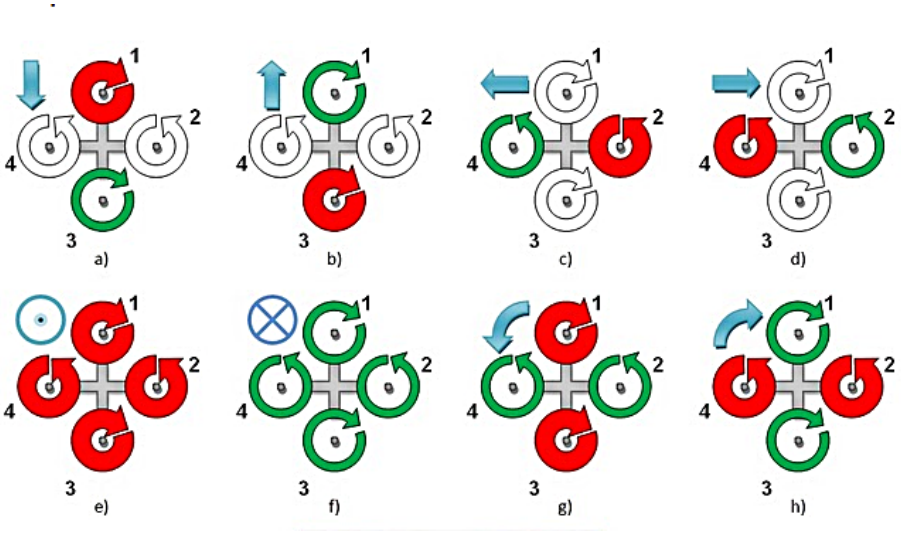


Figura 1.5: Movimiento de un cuadricóptero

Los UAVs se sirven, al igual que cualquier otro tipo de robot, de diversos sensores y actuadores para poder realizar su propósito. Los sensores más comunes son:

- **Altímetro:** este sensor mide los cambios en la presión atmosférica de forma que puede establecer la altura a la que se encuentra el dron.
- **Magnetómetro:** es un dispositivo que mide la dirección del campo gravitacional de la tierra y de esta forma calcula la orientación del dispositivo con respecto a ésta.
- **Receptor GPS:** (Global Positioning System), funciona mediante una red de satélites que orbitan la Tierra de forma estacionaria triangulando la posición del receptor a partir de los tiempos de respuesta con dichos satélites, ya que como mínimo siempre podrá establecer contacto con tres de estos.
- **Unidad de medición inercial o IMU (Inertial Measurement Unit):** es un dispositivo electrónico que mediante una combinación de acelerómetrosy giróscopos determina la velocidad, orientación y las fuerzas gravitacionales del vehículo

- **Cámara:** que recopila vídeo o imagen del entorno en el que se sitúa el vehículo. Es el sensor más importante para este trabajo, ya que será la única fuente de información que determine la posición del dron.

La navegación autónoma de UAVs normalmente se basa en el uso de los receptores GPS que tienen una precisión de unos pocos metros. Se han desarrollado numerosos proyectos de vuelo autónomo de drones en exteriores, como el caso de la policía británica que ha comenzado a usarlos para realizar patrullas por rutas predefinidas, o el proyecto de *Amazon* con el que se pretende realizar entregas de paquetes de forma automatizada empleando drones. Sin embargo, para la navegación en interiores la localización vía GPS no da resultados fiables debido al margen de error que este método tiene. La cantidad de obstáculos y el escaso espacio de vuelo del que se dispone hacen que un margen de un par de metros sea inviable, para ello se debe hacer empleo de otras técnicas más complejas como la visión artificial [9][4][3].

1.3 Visión artificial

La visión artificial, también conocida como visión por computador, es una rama tecnológica, subcampo de la inteligencia artificial en la que mediante el procesado y análisis de imágenes se trata de extraer información para que sea tratada por un computador valiéndose de la estadística, la geometría, la óptica y otras disciplinas científicas. La entrada a un sistema de visión artificial es una imagen, y la salida producida es la información de la realidad contenida en la escena. De esta forma el sentido de la vista se implementa en un robot, permitiendo realizar tareas interactivas con el medio en el que se encuentra.

Esta ciencia surgió con la idea de conectar una cámara de vídeo a un computador y que éste comprendiera lo que las imágenes representaban. El inicio se puede marcar en 1960, cuando Larry Roberts, creador de *ARPAnet*, estudió mediante su tesis doctoral en el MIT la posibilidad de extraer información en tres dimensiones a partir de imágenes en dos dimesiones tomadas desde diferentes perspectivas. De esta forma creó un programa

en el que un robot podía analizar imágenes de una estructura de bloques sobre una mesa, analizando su contenido y pudiendo reproducir la misma estructura desde otra perspectiva.

Los métodos utilizados han mejorado de forma considerable en las últimas décadas. Los primeros sistemas actuaban sobre imágenes binarias que se procesaban en pequeños bloques, estando muy limitados en cuanto al tipo de imagen sobre la que se operaba. Con el desarrollo de los algoritmos y las cámaras, se pudo crear sistemas que analizaran imágenes en escala de grises y posteriormente imágenes en color, aumentando notablemente las capacidades de la visión por computador.

Las bases de un sistema de visión artificial son el sistema de formación de imágenes y el sistema de procesamiento. La formación de imágenes está constituida por los procesos de captación de imagen y la aplicación de técnicas ópticas que permiten distinguir las particularidades visuales más importantes para el procesado. Una vez obtenida la señal, ésta es procesada mediante algoritmos para obtener la información de alto nivel deseada. En primer lugar, se lleva a cabo un preprocesado cuya función es reducir el ruido y realzar las características de la imagen, a continuación se segmenta la imagen en regiones de interés, se extrae la información mediante el análisis de estas regiones y finalmente se clasifica e interpreta la información obtenida.

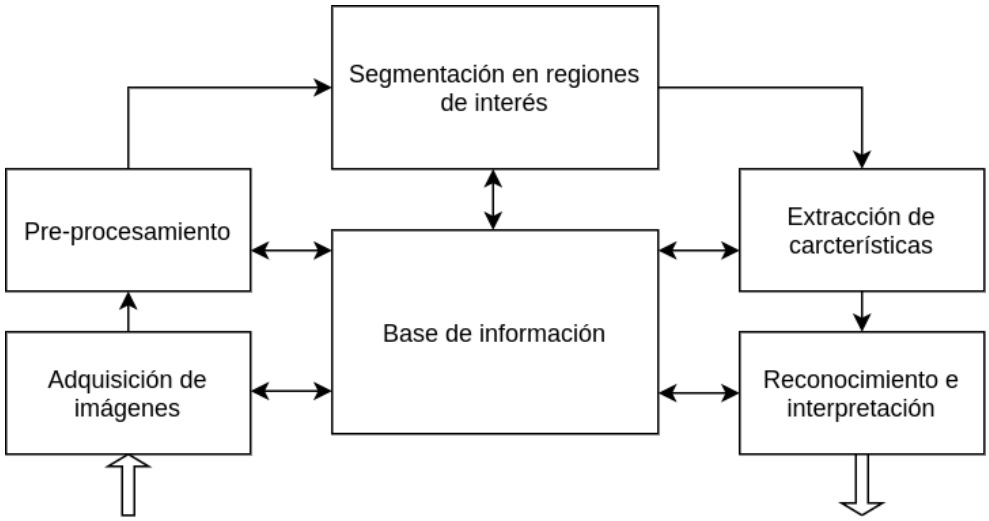


Figura 1.6: Sistema de visión artificial

Dentro de las aplicaciones de la visión por computador en la robótica podemos encontrar la autolocalización visual, mediante la que se consigue determinar la posición y orientación de la cámara con respecto al medio que la rodea sirviéndose de las imágenes capturadas como única fuente de información. Existen diversas técnicas que podemos clasificar en dos grupos, las que miden la posición relativa de la cámara y las que miden la posición absoluta.

En primer lugar, las técnicas que miden la posición relativa basan su funcionamiento en el movimiento que realiza el dispositivo. Es el caso de la odometría visual, que estima la posición relativa de la cámara con respecto a su posición inicial a partir del análisis de la secuencia de imágenes recibidas, sin previo conocimiento del entorno. Aunque esta técnica proporciona buena precisión a corto plazo la acumulación de errores aumenta notablemente con la distancia recorrida.

Por otro lado, las técnicas de estimación de posición absoluta se sirven de puntos de referencia para calcular la posición de la cámara. Esto se pude conseguir mediante comparación de las imágenes captadas con el aspecto de ciertos elementos presentes en el medio que rodea el robot, ya sean balizas artificiales u otro tipo de distintivo característico del medio.

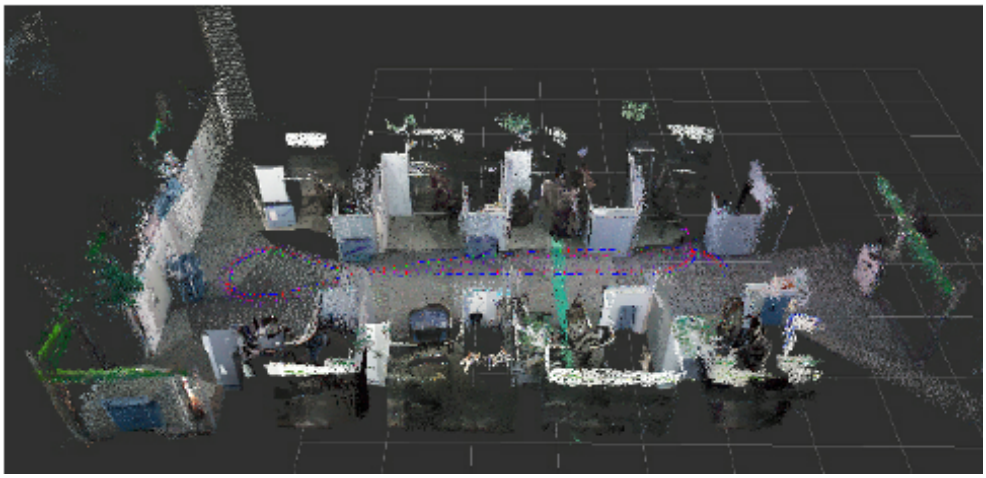


Figura 1.7: VisualSLAM

Una de las técnicas más importantes de este grupo es *Visual SLAM* (Simultaneous Localization and Mapping) con la que se consigue construir el mapa de un entorno a la vez que se obtiene la posición de la cámara en éste. Un ejemplo es la técnica *MonoSLAM*, que se fundamenta en el uso de una cámara monocular. Para su funcionamiento, se calcula la posición inicial de la cámara dotando a un *Filtro de Kalman Extendido* de la posición de al menos tres puntos. A partir de ahí, se puede estimar la posición y orientación de la cámara y generar nuevos puntos de referencia en el mapa.

Otra técnica de autolocalización es la visión por balizas visuales. Se basa en el conocimiento previo del entorno, en el que se sitúan una serie de marcadores cuya posición es conocida. Con la detección de estos marcadores se calcula la distancia y orientación relativa al marcador y, ya que la posición con respecto al entorno de la baliza es conocida, se obtiene la posición absoluta de la cámara. Los marcadores deben tener una serie de características para poder estimar la posición con el menor grado de error, esto son un alto contraste, de forma que no se pueda confundir con el medio en el que se encuentre, y la forma, los marcadores deben ser muy distintos entre sí. Este método es barato y fácil de instalar, aunque es muy costoso computacionalmente debido a la complejidad de los algoritmos necesarios. En este trabajo los marcadores visuales son los utilizados por la librería de detección de balizas *AprilTags*.

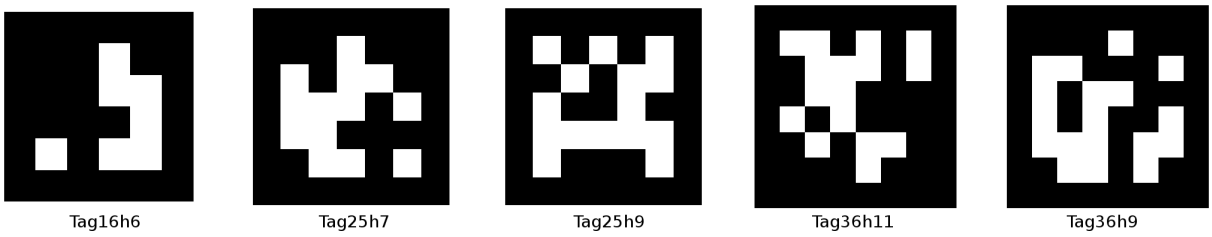


Figura 1.8: Balizas visuales AprilTags

1.4 Antecedentes

Se han realizado diversos trabajos en el Grupo de Robótica de la Universidad Rey Juan Carlos que sirven de antecedente al presente Proyecto de Fin de Carrera. En concreto algunos proyectos relacionados con la visión artificial y el soporte software para cuadricópteros sirven de precedente directo a este trabajo, además de JdeRobot, que es la plataforma base ha permitido desarrollar estos trabajos.

En 2015, Alberto López Cerón escribe 'Autolocalización visual robusta basada en marcadores' [6] como su Trabajo de Fin de Máster. En este trabajo expone varias técnicas de autolocalización visual y desarrolla una aplicación que resuelve el problema de la estimación de posición. Mediante el uso de una cámara a color y los marcadores AprilTags, consigue calcular de forma precisa la posición de la cámara con respecto a estos marcadores. Para esto se sirve del procesado de imagen con OpenCV y la librería de geometría proyectiva Aruco.

También en 2015 Daniel Yagüe Sánchez da soporte software para la simulación con drones en su Trabajo Fin de Carrera 'Cuadricóptero AR.Drone en Gazebo y JdeRobot' [7]. Aquí implementa los drivers necesarios integrar la simulación de cuadricópteros en JdeRobot, así como varios componentes de navegación autónoma para drones.

Más tarde, en 2016, Samuel Matín Martínez desarrolla su Trabajo Fin de Grado titulado 'Mapas 3D de Parches Planos en Entornos Reales utilizando Sensores RGB-D' [5], donde se sirve del trabajo previo realizado por Alberto López para estimar la posición de la cámara. Aquí, Samuel Martín modifica el código añadiendo una capa de comunicaciones necesaria para poder integrar la aplicación en su trabajo y refactorizando los tipos de datos según los estándares de JdeRobot.

Capítulo 2

Objetivos

En este capítulo se explicarán los objetivos que se quieren alcanzar con el presente trabajo, así como los requisitos que se deben cumplir y la metodología y plan de trabajo utilizados para el desarrollo del mismo.

2.1 Descripción del problema

El objetivo de este proyecto es el desarrollo de un sistema de navegación para drones que permita un vuelo autónomo en interiores mediante el seguimiento fino de una ruta establecida a priori. Para ello, el drone ha de conocer en todo momento su posición en el entorno mediante técnicas de visión por computador basadas en marcadores visuales artificiales.

En cuanto a la autolocalización visual, y debido a la complejidad del sistema, se parte de un componente 'cam_autoloc' desarrollado previamente por Alberto López [6] y mejorado más tarde por Samuel Martín [5] en su Proyecto Fin de Máster y Trabajo Fin de Grado respectivamente. Este componente calcula la posición y orientación 3D de una cámara a partir de las imágenes capturadas mediante la detección de las balizas visuales AprilTags.

Conociendo el objetivo principal, éste se ha desglosado en varios subobjetivos que se definen a continución:

1. Refactorización e integración del componente 'cam_autoloc'. Debido a que las versiones de la infraestructura utilizada en el componente han quedado obsoletas en comparación a los estándares de JdeRobot, se refactorizará el compontente. Además se harán las modificaciones necesarias para adaptarlo a las necesidades del sistema de navegación desarrollado.

2. Desarrollo de un componente de navegación basado en la posición absoluta del drone. Para ello también se desarrollará una interfaz gráfica que muestre las imágenes obtenidas por la cámara del drone y un mundo en 3D que muestre tanto el movimiento del drone como la ruta a seguir. Además, desde dicha interfaz se permitirá arrancar y parar el drone.

3. Validación experimental en entorno simulado. Para lo que se creará un mundo tridimensional en el que se realizarán las pruebas pertinentes para determinar la robustez del sistema en conjunto. Se examinará el error de la estimación de posición y cómo este puede afectar al sistema de pilotaje del cuadricóptero.

2.2 Requisitos

Los requisitos a satisfacer por la aplicación desarrollada en este proyecto son los siguientes:

- La aplicación hará uso de la plataforma de desarrollo JdeRobot, utilizando los componentes y herramientas contenidos que sean de utilidad.

- La aplicación debe estar desarrollada modularmente con los estándares de JdeRobot para de esta forma poder integrarse en la plataforma.

- El sistema operativo para el desarrollo y ejecución deberá ser Ubuntu 16.04.

- El sistema de pilotaje será desarrollado utilizando Python como lenguaje de programación.
- Para la autolocalización, el sistema sólo puede depender de las imagenes servidas por la cámara del drone.
- La estimación de la posición deberá ser los suficientemente robusta para permitir una navegación fina del drone en espacios interiores.
- La aplicación desarrollada debe ser capaz de ejectuarse en tiempo real, debiendo tener la eficiencia computacional necesaria para permitir una navegación fiable.
- El sistema debe ser exportable a cualquier escenario que sea un espacio de interiores y contenga marcadores AprilTags con posiciones conocidas a priori.

2.3 Metodología de desarrollo

Para establecer las pautas a seguir en las tareas de un proyecto es necesario definir una metodología. Para este proyecto se ha elegido un modelo de ciclo de vida de desarrollo en espiral basado en prototipos. Esta metodología es un modelo ampliamente utilizado en ingeniería de software ya que facilita un desarrollo rápido de versiones sin tener que basarse en fases estrictamente definidas y separadas. El modelo permite trabajar en el proyecto de forma incremental, aumentando la complejidad del prototipo generado en cada iteración. El ciclo de vida permite obtener productos parciales que pueden ser evaluados permitiendo una adaptación sencilla a los cambios requeridos.

La metodología consta de una serie de conjuntos de actividades agrupadas en cuatro regiones de tareas. De esta forma el modelo itera sobre estas actividades en un número de ciclos necesarios hasta obtener el producto final. Estos conjuntos de tareas son los siguientes:

- **Determinar objetivos.** En esta fase se establecen los hitos a cumplir en la iteración teniendo en cuenta los objetivos finales.

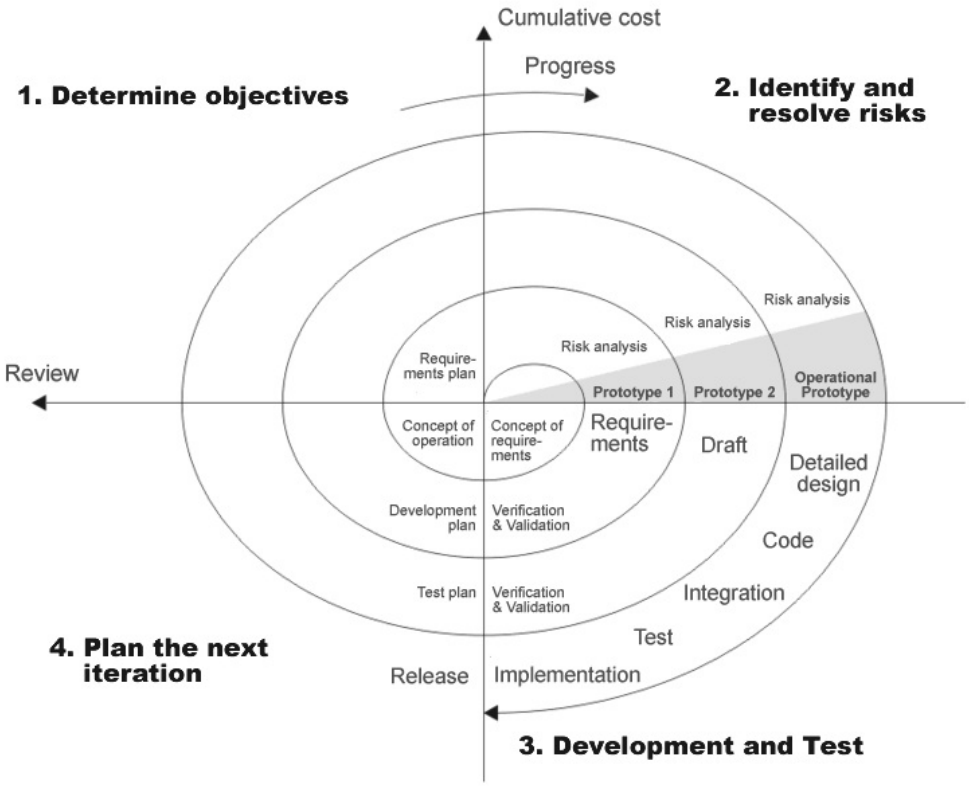


Figura 2.1: Esquema de desarrollo en espiral

- **Análisis del riesgo y evaluación de alternativas.** Se estudian los factores de riesgo y el impacto que éstos puedan ocasionar. Además, se evalúan las distintas formas de alcanzar los hitos fijados considerando los pros y contras desde diferentes puntos de vista.

- **Desarollo y validación.** Se ejecutan las tareas definidas para lograr los objetivos del ciclo y una vez desarrollado se realizan las tareas de pruebas necesarias para comprobar el buen funcionamiento del producto.

- **Planificación.** Una vez analizados los resultados, se planea la siguiente iteración teniendo en cuenta los problemas y errores encontrados durante esta iteración.

2.4 Plan de Trabajo

A lo largo del desarrollo de este trabajo se han realizado reuniones con el tutor de forma periódica en las que se comentaban los problemas encontrados y se establecía la ruta a seguir. Durante todo el proceso he creado una página mediawiki a modo de cuaderno de bitácora en la que se comentan todos los avances así como se muestra contenido audiovisual de los mismos[1]. Además, el código está disponible en un repositorio GIT[2].

Mediante el seguimiento de la metodología de trabajo expuesta anteriormente, en el desarrollo del trabajo se pueden diferenciar distintas fases:

1. **Familiarización con JdeRobot.** El objetivo principal de esta fase es comprender el funcionamiento de la plataforma y aprender a utilizar sus herramientas. También se prepara el entorno de desarrollo con la instalación del software necesario y sus dependencias. Para ello se elaboró un componente básico que consistía en una interfaz gráfica que mostraba las imágenes servidas por una cámara y a la que se podían aplicar varios filtros mediante OpenCV.

2. **Familiarización con Gazebo.** Para entender la herramienta de simulación se elaboró un componente básico de navegación para el robot terrestre Pioneer. Además, se creó el entorno de simulación para el vuelo de un drone en interiores utilizado en las siguientes fases.

3. **Desarrollo de aplicación de monitorización para drones.** En esta etapa se desarrolló un componente prototipo que constaba de una interfaz gráfica en la que se mostraba un mundo en 3D construido mediante *OpenGL*. El propósito de este componente era monitorizar la posición del drone con respecto a un eje de coordenadas y mostrar su estela de movimiento además de una ruta definida.

4. **Desarrollo de componente de navegación.** Durante esta etapa se añadieron nuevas funcionalidades a la aplicación de monitorización anteriormente implemen-

[1] http://jderobot.org/Mazafrav-pfc

[2] https://github.com/RoboticsURJC-students/2016-pfc-Manuel_Zafra

tada. Se desarrolló un sistema de navegación para el cuadricóptero consistente en el seguimiento de una ruta en 3D.

5. **Refactorización e integración de componente de autolocalización.** Para poder integrar el componente fue necesario realizar diversas modificaciones y ajustes tales como la actualización del código a las nuevas versiones de sus dependencias, resolución de errores de compilación y adaptación de algoritmos. También se integró el componente en aplicación desarrollada y se estudió el comportamiento de los algoritmos de visión artificial mediante pruebas en simulación y se realizaron las modificaciones pertinentes hasta obtener una autolocalización robusta.

6. **Validación experimental en simulación.** Se realizaron los experimentos pertinentes en entornos simulados para comprobar la estabilidad del sistema y su viabilidad en un entorno real.

Capítulo 3

Infraestructura

En el presente capítulo se describen las distintas elecciones de infraestructura tanto hardware como software en la que nos hemos apoyado en la realización de este Proyecto Fin de Carrera. Además, se explica el funcionamiento de los distintos componentes que la forman. Los componentes software abarcan desde middleware, plataformas de desarrollo y plugins, hasta bibliotecas de álgebra e interfaces gráficas.

3.1 Cuadricóptero Parrot ArDrone2

La plataforma física utilizada es el *ArDrone2.0* de *Parrot*. El cuadricóptero fue desarrollado en 2012 como una versión mejorada de su antecesor *ArDrone*. El procesador a bordo utiliza un sistema operativo *Linux* y se comunica con el piloto a través de una punto Wi-Fi autogenerado. Los sensores de los que dispone incluyen un altímetro ultrasónico, así como un giroscopio, acelerómetro y magnetómetro de tres ejes, que son utilizados para la estabilización del vehículo. También dispone de una cámara frontal de 720p y de una cámara ventral QVGA.

Tabla 3.1: Características ArDrone2.0

Característica	Valor
Tamaño	58.4 x 58.4 cm.
Peso	436 g.
Alcance Wi-fi	50 m.
Velocidad máx.	18 km/h
Camara frontal	1280 x 720 pixels
Altura máx.	100 m.

Figura 3.1: ArDrone 2

3.2 Entorno JdeRobot

Es un entorno de desarrollo (*framework*) multiplataforma de código libre utilizado en elaborado a partir de la tesis doctoral de Jose María Cañas Plaza [1] y mantenido por el Grupo de Robótica de la Universidad Rey Juan Carlos. Está orientado a la implementación de software en los ámbitos de la robótica, la visión artificial y la domótica. La función de esta plataforma es facilitar la integración y el desarrollo de este tipo de aplicaciones permi-

tiendo el acceso a hardware con interfaces sencillas, creando una capa de comunicaciones entre componentes y aportando numerosas bibliotecas y drivers.

JdeRobot[1] usa ICE para comunicar los diferentes componentes a través de conexiones TCP/IP, permitiendo el flujo de información entre aplicaciones independientemente del lenguaje de programación usado o de la arquitectura interna de estos. Así se facilita la modularidad de los componentes desarrollados a la vez que se simplifica la comunicación entre los mismos.

Por otra parte, dentro de *JdeRobot* se puede encontrar una gran variedad de drivers desarrollados para dar soporte a los distintos sensores y actuadores físicos como pueden ser cámaras, drones o robots terrestres. Esto facilita el acceso a hardware con una interfaz sencilla, proporcionando una capa por encima del software de los fabricantes como por ejemplo *Ar.Drone SDK* en el caso del *ArDrone*. Esto proporciona una capa de abstracción de forma que los componentes externos al robot funcionen de forma independiente de los distintos tipos de modelos y marcas utilizados.

Además, la plataforma también dispone de una serie de plugins que dan soporte a disintots robots dentro del simulador *Gazebo*, un entorno de simulación orientado a la robótica. De esta forma se pueden realizar pruebas sobre los componentes desarrollados sin necesidad de adaptar el código, ya que la comunicación se sigue realizando a través de ICE y los plugins se encargan de reproducir el comportamiento real de los dispositivos.

3.2.1 Interfaz Pose3D

Pose3D es una interfaz empleada en *JdeRobot* con la que se implementa el tipo de datos `Pose3Ddata` cuya función es definir la posición y orientación de un objeto en el espacio 3D. Está compuesta por un punto en 3D que indica la situación del objeto en coordenadas homogéneas y por un cuaternión que marca la orientación del mismo. Para este objeto se tienen los métodos set y get.

[1]http://jderobot.org/Main_Page

```
 Pose3DData
{
 float x; /* x coord */
 float y; /* y coord */
 float z; /* z coord */
 float h; /* */
 float q0; /* qw */
 float q1; /* qx */
 float q2; /* qy */
 float q3; /* qz */
};
```

3.2.2 Herramienta UAV_Viewer

Este componente se encuentra dentro de *JdeRobot* y permite la teleoperación del cuadricóptero y la visualización de los sensores más importantes como son la cámara, el velocímetro y el altímetro mediante una interfaz gráfica. Originalmente fue creado por Alberto Martín [8] en 2014 como herramienta de visualización y de teleoperación del dispositivo.

En este trabajo el componente ha servido para teloperar el drone en las fases tempranas de desarrollo y para la depuración de errores y pruebas en cuanto a la autolocalización visual.

3.2.3 Herramienta CameraCalibrator

Este componente es una herramienta de *JdeRobot* que permite obtener los parámetros extrínsecos e intrínsecos de una cámara. *CameraCalibrator* recibe las imágenes captadas por la camara mediante una interfa ICE y ofrece una sencilla interfaz de usuario que facilita el proceso de calibración. En su archivo de configuración se puede modificar parámetros como el número de tomas que se van a realizar, el retardo entre ellas o el patrón de calibración. El componente utiliza las herramientas de calibración de *OpenCV* y escribe los datos de salida en un fichero `.yml`. Esta herramienta fue utilizada para calibrar tanto la cámara del drone simulado (Figura 3.2) como la cámara del cuadricóptero real ArDrone2.

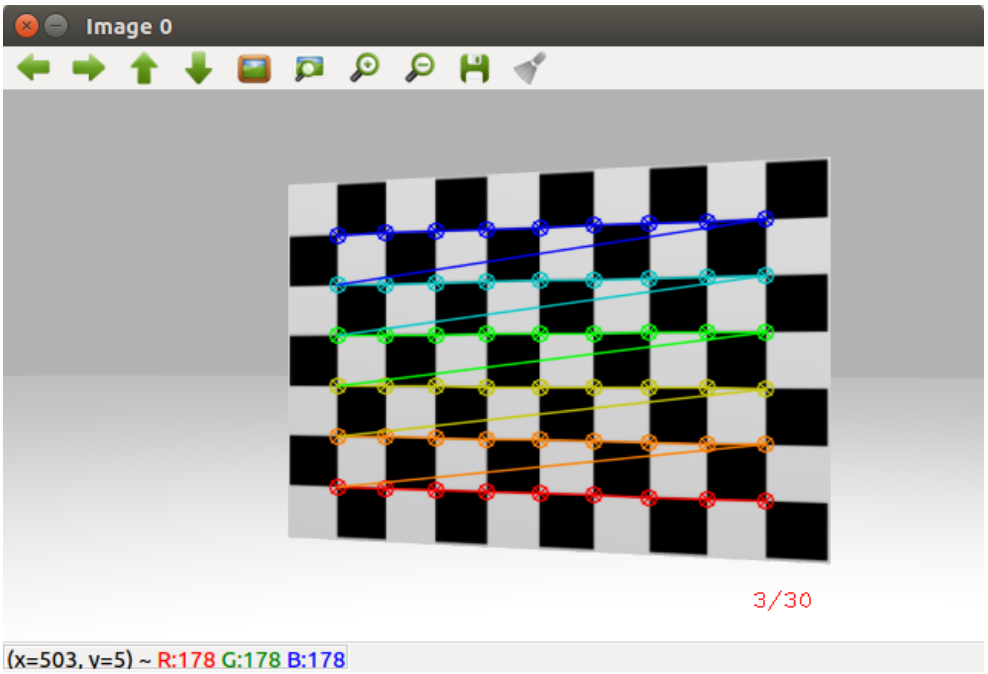


Figura 3.2: Proceso de calibración en Gazebo

3.2.4 Herramientas Recorder/Replayer

Estos componentes pertenecientes a la plataforma *JdeRobot* ofrecen la capacidad de captar y reproducir flujos de datos servidos en las interfaces ICE. Recorder se ata a un servidor ICE y almacena en ficheros los datos servidos por éste. De forma análoga, Replayer reproduce un flujo de datos almacenado Recorder mediante una interfaz ICE.

Las ventajas ofrecidas son numerosas, como la capacidad de grabar los datos recibidos a frecuencia definidas por el usuario permitiendo almacenar datos sensoriales mientras se teleopera, de forma que los datos recogidos se puedan estudiar detenidamente y usar repeditadmente después. Estos componentes han sido la herramienta utilizada para almacenar las rutas de navegación del drone.

3.2.5 Driver ArDrone_Server

ArDrone_Server es el driver incluido en *JdeRobot* que permite conectarse al ArDrone Parrot mediante intefaces ICE. El driver encapsula las bibliotekas ArDrone SDK ofrecidas por el fabricante para controlarlo.

Son seis las interfaces ofrecidas por el driver. Tres de ellas están destinadas a servir la información obtenida por los sensores: 'camera', donde se sirven las imágenes captadas por las cámaras; 'pose3d', correspondiente a los sensores IMU y GPS; y 'navdata', que ofrece información de diversos sensores como el estado del drone, su aceleración y la velocidad del viento entre otros.

Las otras tres interfaces están orientadas a la actuación y la configuración del drone. Tenemos la interfaz 'cmdVel' cuyo objetivo es enviar los comandos de velocidades tanto lineales como angulares para controlar el movimiento del drone. Por otro lado está 'ardrone_extra', en la que se definen métodos para realizar maniobras más complejas como aterrizar o despegar. También incluye otros métodos para cambiar la cámara desde la que se reciben imágenes y aportar otras funciones extra al drone.

Finalmente, existe una interfaz de configuración llamada 'remoteConfig' que permite configurar el drone en tiempo real mediante la escritura de parámetros en un fichero.

3.2.6 Progeo

Progeo (Projective Geometry) es una biblioteca incluida en JdeRobot que proporciona una serie de funciones para la resolución de problemas de geometría proyectiva. Estas funciones relacionan los puntos en 2D de la imagen con puntos 3D del entorno real. Las funciones principales son 'project', que proporciona el punto de la imagen donde proyecta un punto del mundo dado y 'backproject', que calcula un rayo de retroproyección correspondiente a un punto dado de la imagen. Para el funcionamiento de esta librería es necesario calibrar la cámara previamente para obtener los parámetros intrínsecos de la misma.

3.2.7 ArDrone2 Plugin

Los plugins son bibliotecas dinámicas que se cargan en tiempo de ejecución y que otorgan de funcionalidad a los distintos objetos del mundo simulado en *Gazebo*, comportándose de forma similar a drivers. En 2015 Daniel Yagüe [7] desarrolla un plugin para la simulacion de cuadricópteros en *Gazebo* con el que implementa un control realista optimizado para el ArDrone 2.0. Este plugin está integrado actualmente en la plataforma *JdeRobot* y sobre el mismo se ha realizado la simulación del cuadricóptero de este proyecto.

3.3 Componente Cam_Autoloc

Cam_autoloc es aplicación desarrollada por Alberto López Cerón [6] en 2015 para su Trabajo Fin de Máster 'Autolocalización visual robusta basada en marcadores'. El componente fue desarrollado dentro de la plataforma *JdeRobot* y está programada en C++. Su función es estimar la posición y orientación 3D de una cámara, ya sea en un entorno real o simulado, a partir de los marcadores visuales *AprilTags*.

El algoritmo se sirve de las imágenes recibidas a través de una interfaz creada con ICE y devuelve la posición estimada mediante un objeto `Pose3D`. Además del flujo de imágenes, necesita de información conocida a priori suministrada a través de dos ficheros. El primero es un fichero de texto que contiene una lista donde figuran el identificador, posición y orientación 3D de cada baliza visual ubicada en el entorno. El segundo fichero contiene toda la información de los parámetros intrínsecos de la cámara utilizada, por lo que es necesario realizar un análisis previo de la cámara para extraer dichos parámetros.

Para estimar la posición, el algoritmo comienza analizando la imagen recibida mediante las librerías *OpenCV* y *AprilTags* para explorar la imagen en 2D en busca de las balizas. Una vez localizadas localizadas, se hace uso de la librería *Progeo* para calcular la posición y orientación en tres dimensiones de la cámara con respecto a cada marcador. Finalmente, se realiza un proceso de fusión temporal y fusión espacial de las estimaciones obtenidas

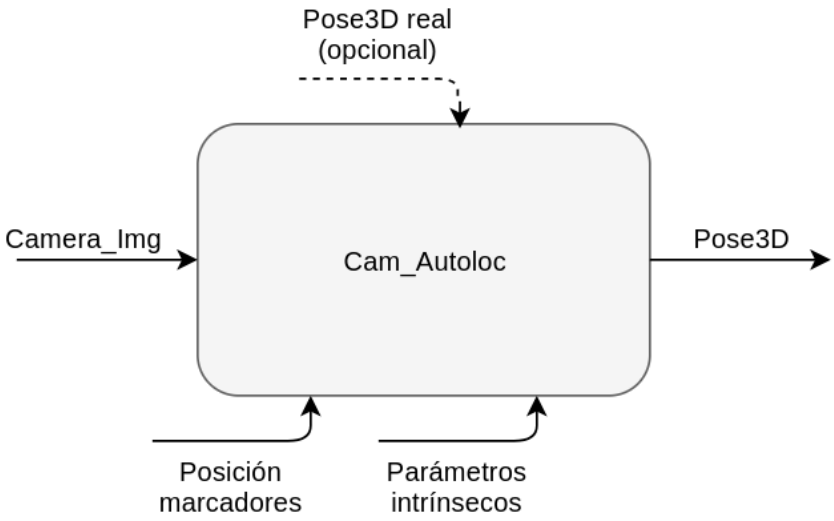


Figura 3.3: Interfaces Cam_autoloc

a partir de cada baliza. La aplicación permite elegir qué clase de filtro temporal utilizar, pudiendo escoger entre un filtro por pesos o un *filtro Kalman*.

3.4 Biblioteca OpenCV

OpenCV[2] responde a las siglas de Open Source Computer Vision Library. Es una librería de software libre con licencia BSD orientada al desarrollo de aplicaciones de visión artificial y *machine learning*. Fue desarrollada originalmente por Intel a principios de 1990 y desde entonces se ha utilizado en infinidad de aplicaciones por grupos de investigación, empresas privadas u organizaciones gubernamentales gracias a que su publicación bajo licencia BSD permite que sea libremente utilizada para propósitos comerciales y de investigación.

OpenCV es multiplataforma, existiendo versiones para la mayoría de los sistemas operativos más comunes. Está escrita en C++ aunque existen 'bindings' en otros lenguajes como *Python*, *Java* o *MATLAB*. La librería contiene más de 2500 algoritmos que permiten

[2]http://opencv.org/

identificar rasgos faciales, hacer tracking de objetos, encontrar imágenes similares o extraer el modelo en 3D de objetos.

3.5 Balizas visuales AprilTags

AprilTags es un sistema de visión por computador que permite detectar balizas visuales contenidas en una imagen. Fue creado en 2011 por Ed Olson [10] y es empleado en una gran variedad de tareas relacionadas con la visión artifical, la robótica y la calibración de cámaras.

Las balizas se basan en el concepto de los códigos QR, aunque éstas están diseñadas para contener muchos menos bits de información. Además, presenta un nuevo sistema de codificación que aborda problemas específicos de los códigos de barras 2D como es la robustez frente a la rotación y a los falsos positivos que pueden dar las imágenes naturales.

La librería detecta cualquier baliza presente en una imagen y proporciona la posición de la baliza en dicha imagen además de su correspondiente ID. Eso lo realiza mediante un algoritmo de segmentación basado en gradientes locales que consigue que las líneas se estimen con precisión. Esto provoca una tasa de falsos negativos muy bajos, aunque aumenta la probabilidad de falsos positivos. Sin embargo, gracias a la codificación de las balizas esta probabilidad es reducida hasta niveles aceptables.

3.6 Biblioteca ICE

ICE[3] (Internet Comunication Engine) es un middleware desarrollado por *ZeroC*, que basa su funcionamiento en RPC (Remote Procedure Call). Utiliza la arquitectura cliente-servidor y permite que un software ejecute código de forma remota sin tener en cuenta la capa de comunicaciones. ICE permite mecanismos de llamadas tanto asíncronas como

[3]https://zeroc.com/products/ice

síncronas, ejecución sobre TCP, TLS, UDP o WebSocket como protocolos a nivel de transporte, y ofrece control de hilos sin necesidad de preocuparse por regiones críticas.

Las operaciones, tipos de datos e interfaces usados en ICE están definidas mediante SLICE (Specification Language for ICE) que es independiente del lenguaje en el que estén implementados los programas. Esta capa de abstracción permite separar las implementaciones de los objetos de sus interfaces, por tanto, también acepta la comunicación entre programas aunque estén escritos en distintos lenguajes. Actualmente tiene soporte para C++, Java, C#, Python, Objective-C, Ruby y PHP.

En este trabajo, ICE es el mecanismo que permite la comunicación entre las distintas aplicaciones y el drone, de forma que cada componente pueda estar trabajando a frecuencias distintas sin que haya problemas de sincronismo. Otra de las razones para el uso de ICE en este proyecto es la facilidad de integración de las aplicaciones en JdeRobot gracias la modularidad que aporta esta biblioteca.

3.7 Simulador Gazebo

Gazebo[4] es un simulador de entornos 3D que permite evaluar el comportamiento de robots de forma virtual. Es una plataforma de software libre bajo licencia GPL mediante la cual se pueden crear mundos y modelos de robots para probar el sistema desarrollado de forma segura. Cuenta con el respaldo de la OSRF (Open Source Robotic Foundation) y una amplia comunidad de programadores que contribuyen a su desarrollo.

Entre las características de *Gazebo* destacan sus múltiples motores de físicas (*Bullet* y *Open Dynamics Engine* entre otros), soporte para plugins, motor de renderizado avanzado y un gran repositorio con la mayoría de robots comerciales, además de un extenso número de sensores.

Los mundos creados para su uso en *Gazebo* se definen en ficheros con la extensión '.world'. Estos ficheros están escritos mediante SDF (Simulation Description Format), que

[4]http://gazebosim.org/

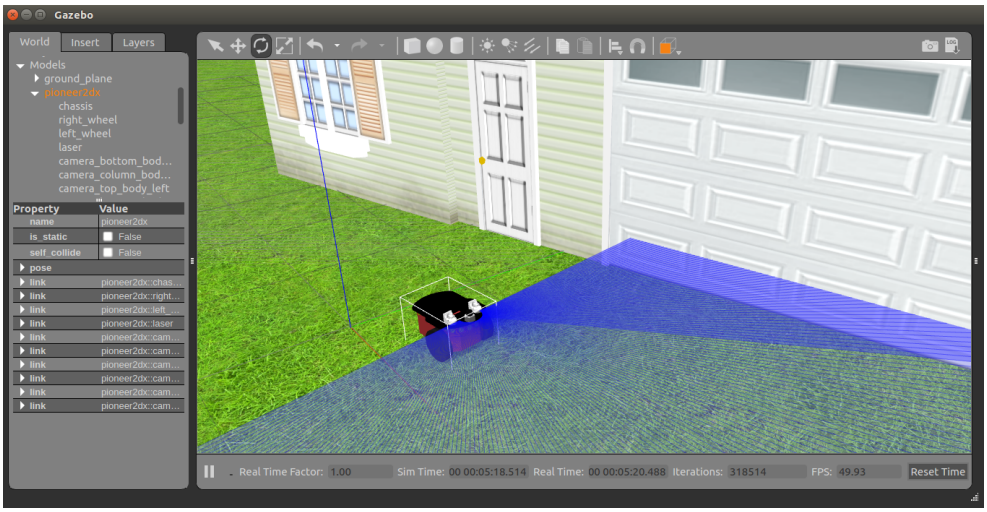


Figura 3.4: Interfaz gráfica de Gazebo

es un formato de XML. Originalmente, SDF fue creado de forma expresa para su uso en *Gazebo*, sin embargo, ha evolucionado hasta convertirse en un formato propio para la descripción de entornos de simulación en robótica. En los ficheros se detallan las características de los distintos modelos que forman parte de la escena, el motor de físicas a utilizar, y otros parámetros como los efectos de luz y las sombras. Para este trabajo he creado un nuevo entorno a partir de un mundo previamente definido, 'GrannyAnnie', al que se han añadido diversas balizas *AprilTags* además de un cuadricóptero.

3.8 Python

Python[5] es un lenguaje de programación desarrollado a finales de los ochenta por Guido van Rossum como sucesor del lenguaje de programación ABC. Es un lenguaje interpretado; multiparadigma, ya que soporta programación orientada a objetos e imperativa; multiplataforma y multipropósito. Se trata de un lenguaje potente, flexible y con una sintaxis clara y concisa. La versión utilizada para el desarrollo este proyecto fue inicialmente Python 2.7, aunque más tarde se refactorizó a Python 3.5.

[5]https://www.python.org/

3.8.1 PyQt

PyQt[6] es un *binding* de la biblioteca gráfica Qt, que es una herramienta multiplataforma cuyo fin es el desarrollo de aplicaciones con interfaza gráfica de usuario. *PyQt* está desarrollado por Riverbank Computing Limited, aunque existe otro *binding* llamado *PySide* con licencia LPGL liberado por Nokia. Qt es desarrollado como un software libre mediante *Qt Project* donde participan tanto desarrolladores de algunas empresas como Nokia como la comunidad *open source* .

Qt utiliza C++ de forma nativa y funciona en las principales plataformas. Su API dispone de métodos para el uso de XML, gestión de hilos, soporte de red, acceso a ficheros y permite acceder a bases de datos mediante SQL.

Su arquitectura se basa en el uso de *signals* y *slots* para la comunciación entre objetos que permite implantar un patrón de diseño de observador de forma fácil sin tener que recurrir a código repetitivo. Los *widgets* de la GUI pueden enviar señales que contienen información de eventos que son recibidos por otros objetos mediante funciones especiales llamadas *slots*.

La interfaz gráfica del componente desarrollado en este trabajo está realizada con *PyQt*, mediante la cual se muestran las imágenes obtenidas por la cámara del cuadricóptero y el mundo en 3D creado con OpenGL mediante *PyOpenGL*, biblioteca para la cual *PyQt* ofrece soporte. Además, en la interfaz se incluyen dos gráficas que muestran los errores de navegación en tiempo real mediante *PyQtGraph*. También se han incluido diversos botones para controlar el comportamiento del drone y el sistema de navegación.

3.8.2 NumPy

Numpy[7] es un módulo de Python que permite a éste funcionar como un lenguaje de alto nivel para la manipulación de datos numéricos, admitiendo un mayor soporte para vectores y matrices y agregando un gran número de funciones matemáticas. Incorpora

[6] https://riverbankcomputing.com/software/pyqt/intro
[7] http://www.numpy.org/

operaciones tan básicas como la suma o la multiplicación u otras mucho más complejas como la transformada de Fourier o el álgebra lineal. Además, incluye herramientas que permiten el uso de código fuente de otros lenguajes de programación como C#, C++ o Fortran.

La biblioteca basa su funcionamiento en la estructura de datos 'ndarray' del inglés N-dimensional array. Al contrario que con los arrays dinámicos definidos en Python, ndarray sólo permite que el tipo de elmentos que lo forman sean homogéneos, es decir, deben ser el mismo tipo de objeto.

3.8.3 PyQtGraph

PyQtGraph[8] es una librería gráfica para Python cuyo objetivo es dotar de ciertas funcionalidades requeridas en aplicaciones científicas y técnicas. Mediante esta librería se pueden obtener gráficos interactivos de forma rápida y fluida gracias a que está implementada sobre *PyQt* y *NumPy*. *PyQtGraph* usa la estructura GraphicsView de *Qt* que es de por sí un sistema gráfico de gran capacidad consiguiendo además una fácil integración con las interfaces de Qt.

En la interfaz gráfica de la aplicación desarrollada se encuentran dos gráficas creadas con esta librería en las que se muestra en tiempo real la magnitud de los errores de posición y orientación del drone con respecto a la ruta que debe seguir.

3.9 OpenGL

OpenGL (Open Graphics Library)[9] es una especificación estándar que define una API multilenguaje y multiplataforma para el desarrollo de aplicaciones gráficas. Está diseñado para su uso en juegos, animación, CAD/CAM, visualización médica y otras aplicaciones que

[8] http://www.pyqtgraph.org/
[9] https://www.opengl.org/

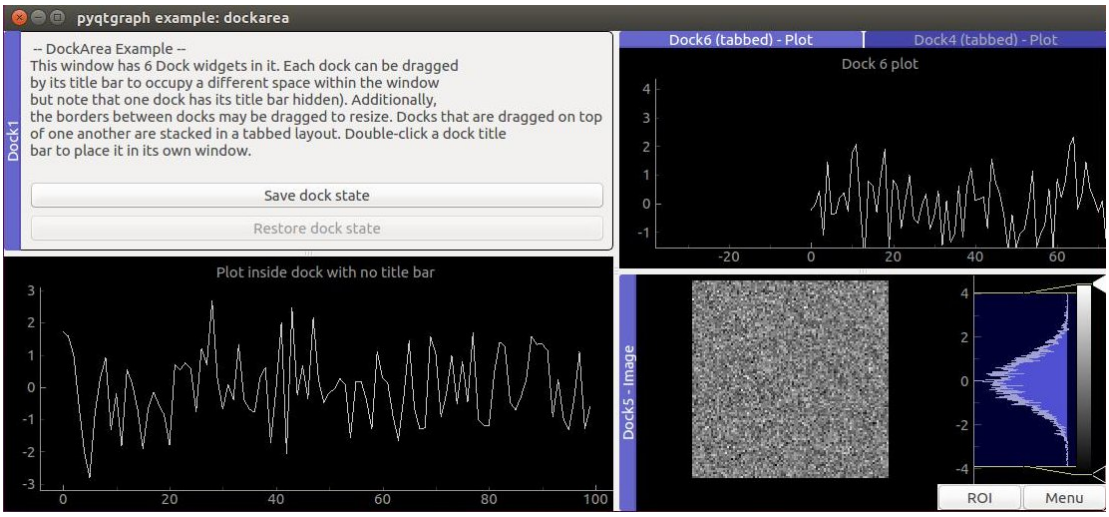


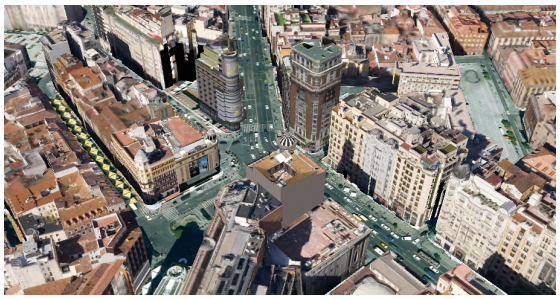
Figura 3.5: PyQtGraph con PyQt

necesiten de un entorno robusto para la visualización de formas en dos y tres dimensiones. El API de *OpenGL* es uno de los estándares gráficos más adoptados gracias a las cualidades de portabilidad e independencia de plataforma que posee. Fue desarrollada originalmente por Silicon Graphics Inc. (SGI) en 1992 y ha seguido desarrollándose y extendiéndose desde entonces.

OpenGL describe un conjunto de funciones y el comportamiento de éstas. A partir de esto, los fabricantes de hardware desarrollan implementaciones en forma de funciones y bibliotecas que se ajustan a los requisitos definidos. No es por lo tanto ningún lenguaje de programación, sino un conjunto de librerías que son utilizadas a través de lenguajes de programación para conseguir un interfaz software entre las aplicaciones y el hardware gráfico.

La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples como puntos, líneas y triángulos. El funcionamiento básico de *OpenGL* consiste en aceptar tales primitivas y convertirlas en píxeles mediante una 'pipeline' gráfica. De esta forma la API se basa en procedimientos de bajo nivel que requieren que el programador dicte los pasos exactos necesarios para renderizar una escena. Este diseño requiere un conocimiento en

(a) Portal 2



(b) Google Earth

Figura 3.6: Aplicaciones gráficas con OpenGL

profundidad de la 'pipeline' gráfica por parte del programador a cambio de un gran grado de libertad para la implementación de algoritmos gráficos.

PyOpenGL[10] es el *binding* multiplataforma para Python utilizado en este trabajo. Está creado mediante SWIG, liberado mediante una licencia similar a las licencias BSD (Berkeley Software Distribution). Entre sus características se encuentra un buen soporte e interoperabilidad con la mayoría de las librerías externas para interfaces gráficas de Python como *PyQt*.

A través de *PyOpenGL* se ha desarrollado un mundo en tres dimensiones en el que se dibuja un eje de coordenadas y el modelo tridimensional de un drone cargado a partir de un fichero. En este mundo se muestra el movimiento del drone, la estela del mismo y la ruta a seguir.

[10]http://pyopengl.sourceforge.net/

Capítulo 4

Desarrollo Software

A lo largo de este capítulo se describe todo el diseño de la solución y la implementación realizada. Se dará una visión global del sistema diseñado donde se explica cómo se comunican los distintos componentes y las funciones globales que estos realizan. A continuación se detallará el funcionamiento y diseño de cada uno de los componentes, además de los problemas encontrados y las soluciones propuestas. Finalmente se describirá el proceso de integración del sistema así como las modificaciones realizadas para optimizar el funcionamiento.

4.1 Diseño global

El objetivo de este trabajo es crear una aplicación que permita el vuelo autónomo de un drone mediante visión artificial y un control por posición. En la aplicación desarrollada se pueden diferenciar dos partes principales: el componente que se encarga de estimar la posición mediante algoritmos de visión por computador y el componente que toma las decisiones sobre el movimiento del mismo. Una vez diferenciados los módulos, el sistema puede abstraerse como un conjunto de cajas negras que intactúan entre sí en el que cada caja recibe y genera distintos flujos de información.

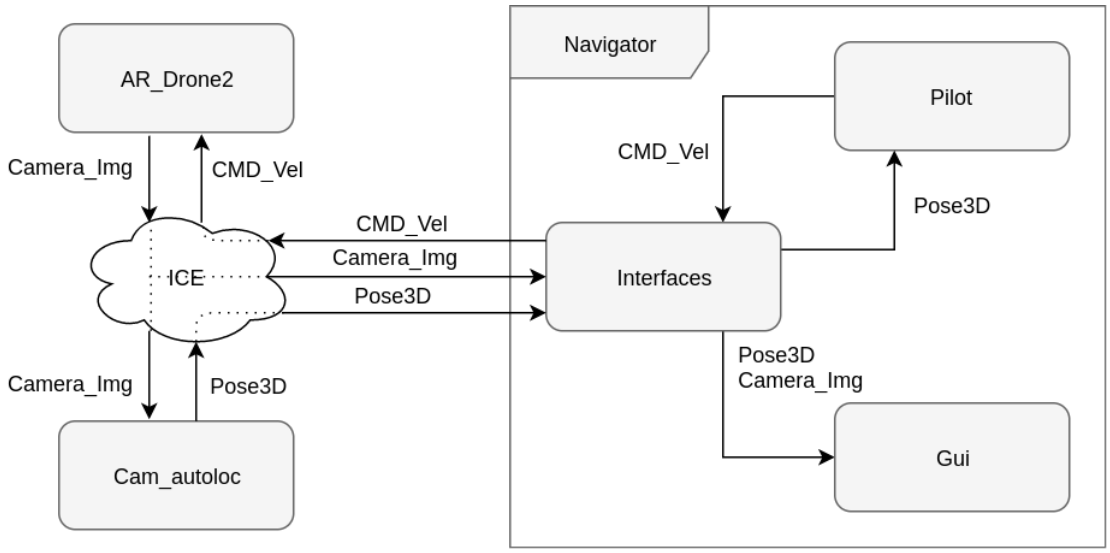


Figura 4.1: Diagrama de caja negra

En la figura 4.1 podemos ver esta composición en cajas negras con todas las entradas y salidas de flujos de información. Se ha omitido la entrada de ficheros de configuración de cada módulo ya que no es relevante para el funcionamiento general y se detallará más adelante. Podemos observar es que toda la comunicación entre procesos se lleva a cabo mediante ICE. Cada módulo tiene al menos una interfaz en modo servidor y una o más interfaces en modo cliente.

En primer lugar, el componente Cam_autoloc recibe las imágenes servidas por el drone y envía datos de tipo `Pose3D` a *Navigator*. Este componente comienza analizando la imagen recibida en busca de la presencia de marcadores *AprilTags*. Una vez localizados, si los hay, aplica cálculos de geometría proyectiva para estimar la posición de la cámara con respecto a cada una de las balizas. Una vez calculadas las posiciones se realiza una fusión espacial mediante un filtro basado en pesos. A continuación se efectúa una fusión temporal aplicando un filtro de Kalman, con el que se realiza una estimación óptima estadísticamente. Finalmente, la estimación calculada se envía al componente de navegación mediante una interfaz ICE.

El componente de navegación *Navigator* recibe las posiciones estimadas y a partir de éstas genera una serie de órdenes de velocidades que envía al drone vía interfaz ICE

para realizar el seguimiento de la ruta predefinida. Dentro de este componente se pueden diferenciar tres módulos, el primero de ellos es Interfaces, que se ocupa de manejar los flujos de información entrantes y salientes del componente. Por otro lado el módulo *GUI* es el responsable de la interfaz gráfica, de la renderización del mundo 3D y la monitorización de errores. Finalmente tenemos Pilot, que es donde se aplican los algoritmos necesarios para la navegación y es el que envía las instrucciones de movimiento al drone. Este módulo recibe únicamente los datos `Pose3D` servidos por *Cam_ autoloc* y la información de la ruta a seguir que es cargada desde un fichero. El algoritmo de navegación se basa en la predicción del error de giro, que se adapta en cada iteración en base la velocidad angular y la posición del drone con respecto a la ruta.

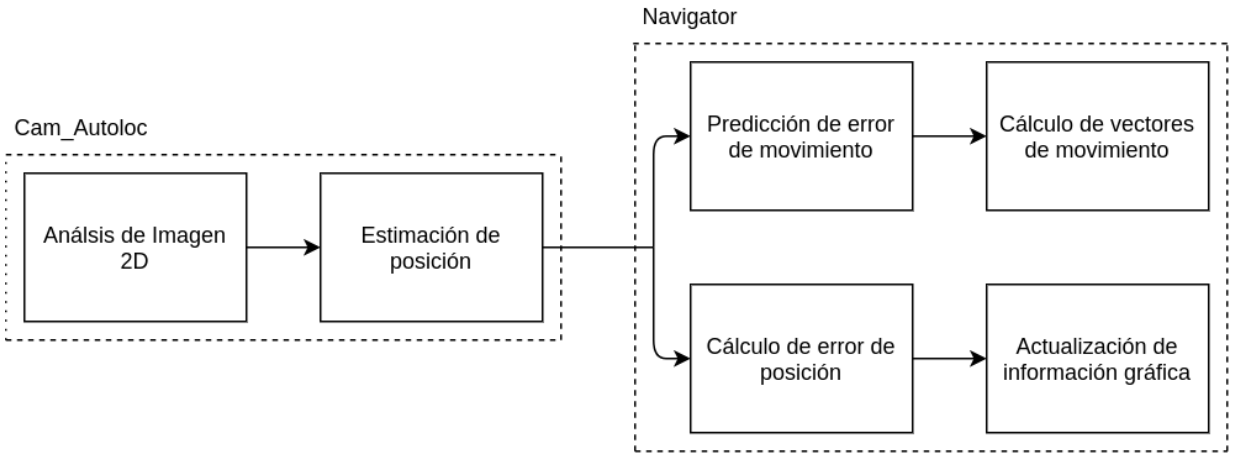


Figura 4.2: Diagrama de bloques

Los componentes están estructurados de forma modular, teniendo una clase principal que crea cada uno de los objetos y a continuación lanza sus correspondientes hilos. En el caso de *Navigator*, cada hilo definido hace una llamada al método `update()` de los objetos, método mediante el cual éstos realizan sus tareas principales. Estos hilos se ejecutan de forma asíncrona con distintas periodicidades predefinidas. El módulo encargado de la gestión de recursos es Interfaces, en el que se definen las conexiones con ICE y se gestionan los recursos. Los problemas de acceso asíncrono a los recursos se gestionan en *Interfaces* mediante el uso de la clase Lock definida en la librería Threading de *Python*. El componente

Cam_ autoloc tiene una estructura muy similar a *Navigator*, teniendo la clase Sensor que actúa de forma análoga a *Interfaces* y la clase *GUI*, donde se implementa la interfaz gráfica y se ejecutan los algoritmos de visión artificial.

4.2 Componente de autolocalización visual, Cam_Autoloc

Este componente fue desarrollado originalmente por Alberto López Cerón[6] como su Proyecto Fin de Máster[1]. Más adelante, Samuel Martín[5] refactorizó parte del código para integrar una capa de comunicaciones mediante interfaces ICE y se añadió un módulo para gestionar la memoria compartida[2]. Además, se añadieron métodos para la conversión entre cuaterniones y ángulos de Euler, debido a que la aplicación original utilizaba los ángulos de Euler y la interfaz Pose3D, cuaterniones.

La arquitectura software del componente está formada por un módulo principal *Main-Window* que interconecta el resto de módulos e implementa una intefaz gráfica de usuario. Uno de esos módulos es *Sensors*, que se encarga de implementar y gestionar las interfaces ICE y los recursos compartidos entre el resto de módulos. Otro de los módulos es *Camera-Manager*, donde se procesan las imágenes y se ejecutan los algoritmos de visión artificial. Finalmente, *World* implementa un mundo en 3D creado mediante OpenGL en el que se muestran las posiciones de las balizas con respecto a un eje de coordenadas y las posiciones estimadas. Siguiendo los estándares de JdeRobot, cada uno de los módulos es ejecutado mediante un hilo que realiza la llamada al correspondiente método `update()`, mediante el cual llevan a cabo sus tareas en cada iteración.

Dentro del componente se ha definido una clase *GeometryUtils* en la que se implementan diversos métodos para el cálculo geométrico. Estos métodos incluyen operaciones tales como el cálculo de la intereseción entre rectas y planos, la generación de matrices de rotación o la conversión entre ángulos de Euler y cuaterniones. La estructura de datos que implementa la cámara es `TPinHoleCamera`, un modelo que se encuentra definido en la

[1] http://jderobot.org/Alopezceron-tfm
[2] http://jderobot.org/Samartin-tfg

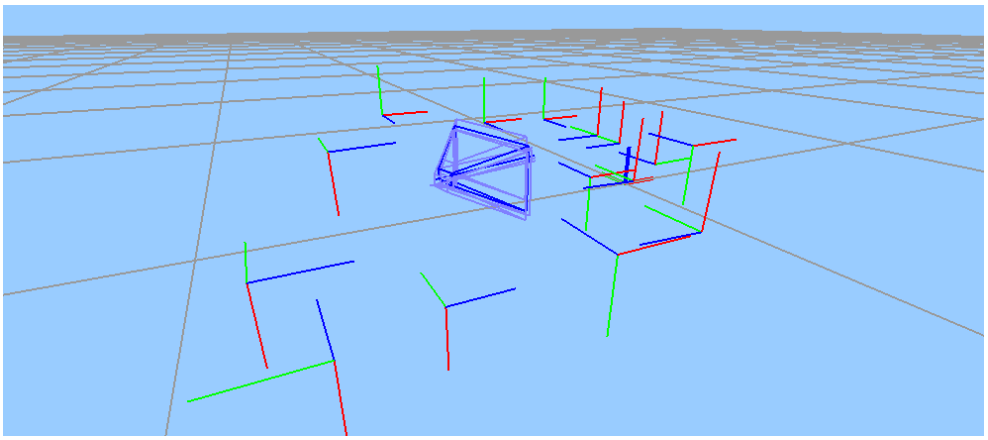


Figura 4.3: Mundo 3D generado por el módulo *World*

librería *Progeo* de *JdeRobot*. Este modelo se basa en un modelado de cámara en el que la proyección es cónica, por lo que todos los rayos de luz eventualmente pasan por un mismo punto que es el foco de la cámara. El modelo Pin Hole es de bastante utilidad gracias a su sencillez y precisión. Para poder instanciar un objeto `TPinHoleCamera` es necesario obtener previamente los parámetros de la cámara mediante calibración.

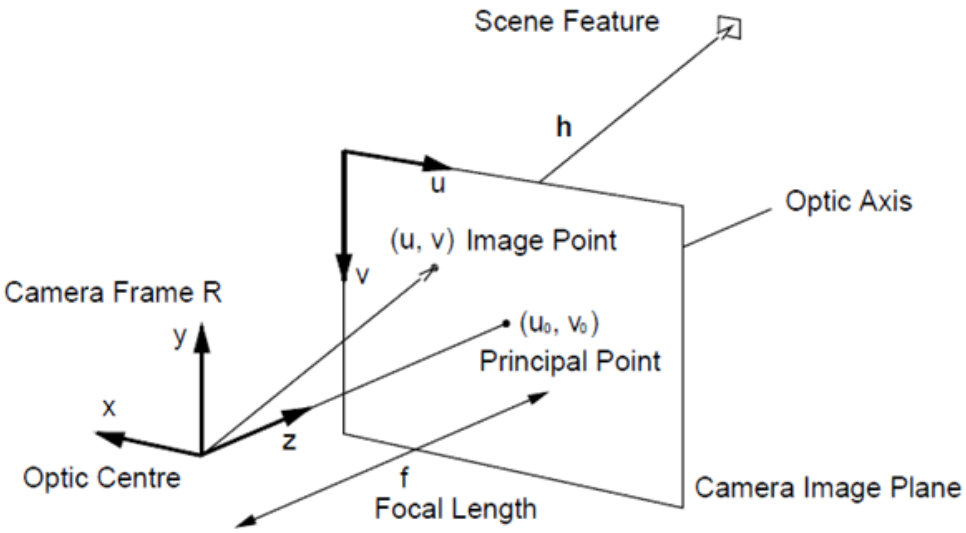


Figura 4.4: Modelo Pin Hole

El método *ProcessImage* contenido en *CameraManager* se ocupa de procesar la imagen en 2D capturada por la cámara y buscar en ésta la presencia de marcadores además de estimar la posición 3D con respecto a éstos. Los marcadores encontrados son instanciados por la clase *MarkerInfo*, que contiene toda la informacion necesaria de cada marcador: id, tamaño y posición. La posición se encuentra almacenada en dos matrices distintas, una que contiene la posición del marcador con respecto al mundo, y la otra la posición del mundo con respecto al marcador. Para localizar los marcadores en la imagen se hace uso del método de detección ofrecido por la biblioteca *AprilTags*. Este método se aplica a una versión en escala de grises de la imagen obtenida y tiene como salida un array en el que figuran todos los marcadores encontrados.

Una vez el array de marcadores detectados es generado, cada uno de ellos es localizado en la imagen original y se aplican una serie de operaciones geométricas. Estas operaciones comienzan con la aplicación del método `SolvePnP()` de *OpenCV*, que devuelve la posición relativa de una cámara dado un sistema de referencia compuesto por la corespondencia entre los puntos 2D de la imagen y los correspondientes puntos 3D del mundo. De esta forma se obtienen los vectores de translación y rotación del marcador con respecto a la cámara. Para obtener la matriz RT (Rotation-Translation) completa, se hace uso de la funcion `Rodrigues()` de *OpenCV*. Finalmente, la matriz que contiene la posición de la cámara con respecto al mundo se obtiene multiplicando la matriz calculada, posición del marcador con respecto a la cámara, y la matriz de posición del mundo con respecto al marcador.

Las posiciones estimadas para cada marcador son almacenadas en un array al que es aplicada una fusión espacial mediante un filtro por pesos. Este filtro asigna diferentes pesos a cada estimación basándose en la distancia a la cámara, de forma que los marcadores más cercanos son asignados con un peso mayuor. Los valores de los pesos fueron determinados de forma experimental, analizando las distancias a las que el sistema pierde precisión. El filtro comienza extrayendo el peso total de la estimaciones y calcula un ratio para cada estimación (4.1). El valor final se obtiene multiplicando cada estimación por su correspondiente ratio y sumando todos los resultados (4.2). En cuanto a los ángulos de

rotación hay una operación especial, ya que no pueden ser sumados de la misma forma que las coordenadas lineales (4.3).

$$ratio_i = \frac{peso_i}{peso_{total}} \tag{4.1}$$

$$[x, y, z]_{fusion} = \sum_{i=1}^{n}([x_i, y_i, z_i] \cdot ratio_i) \tag{4.2}$$

$$\alpha_{fusion} = \arctan\left(\frac{\sum(\sin\alpha_i \cdot ratio_i)}{\sum(\cos\alpha_i \cdot ratio_i)}\right) \tag{4.3}$$

Después de aplicar la fusión espacial, la aplicación original daba opción a aplicar una fusión temporal. Esta fusión puede llevarse a cabo mediante un filtro por pesos como en el caso de la fusión espacial, o mediante la aplicación de un *Filtro de Kalman*. En este caso el código se modificó para que siempre se aplicara un *Filtro de Kalman*. Este filtro fue desarrollado por Rudolf E. Kalman [11] y su función es estimar el estado de un sistema dinámico usando una serie de medidas observadas en el tiempo que poseen ruido estadístico. Tiene numerosas aplicaciones en tecnología, siendo ampliamente utilizado en los campos de la robótica como optimizador de trayectorias y en control de movimiento y navegación. Si el ruido es Gaussiano, el filtro funciona de forma óptima minimizando el error cuadrático.

El filtro estima el estado de un proceso, x, en un tiempo dado k (4.4), conocida la medida z (4.5). Donde w_k representa el ruido del proceso y v_k el ruido de la medida.

$$x_k = A \cdot x_{k-1} + w_{k-1} \tag{4.4}$$

$$z_k = H \cdot x_k + v_k \tag{4.5}$$

La matriz A relaciona el estado actual con el anterior, mientras que la matriz H relaciona el estado con la medida. De esta forma, el algoritmo estima el estado actual del

proceso basándose en el estado anterior mientras se retroalimenta de medidas ruidosas. Las ecuaciones del filtro se pueden dividir en dos grupos, las de predicción de estado y las de actualización de la medida.

Las ecuaciones de predicción son las mostradas a continuación (4.6)(4.7). Donde $\hat{x}_k^-$ es el estado estimado a priori; $\hat{x}_{k-1}$ el estado estimado a posteriori; P_k^- la covarianza de error a priori; and P_{k-1} la covarianza de error a posteriori.

$$\hat{x}_k^- = A \cdot \hat{x}_{k-1} \tag{4.6}$$

$$P_k^- = A \cdot P_{k-1} \cdot A^T + Q \tag{4.7}$$

Las ecuaciones de actualización de la medida (4.8)(4.9)(4.10), se encuentran a continuación, donde $(z_k - H \cdot \hat{x}_k^-)$ es el residuo y K_k es la ganancia de Kalman que minimiza el error a posteriori.

$$K_k = \frac{P_k^- \cdot H^T}{H \cdot P_k^- \cdot H^T + R} \tag{4.8}$$

$$\hat{x}_k = \hat{x}_k^- + K_k \cdot (z_k - H \cdot \hat{x}_k^-) \tag{4.9}$$

$$P_k = (I - K_k \cdot H) \cdot P_k^- \tag{4.10}$$

Con este método obtenemos resultados suavizados de las medidas a la vez que eliminamos los errores de pico. En el sistema de autolocalización, la variación de un solo píxel puede provocar un cambio repentino en la estimación, en ese caso el *Filtro de Kalman* reduce estas variaciones bruscas. Los valores de las matrices de covarianza de error que modelan los errores del sistema y de las medidas deben ser obtenidos de forma experimental para poder conseguir resultados fiables. La estimación de posición final obtenida con el filtro es finalmente enviada al componente *Navigator* mediante ICE.

4.3 Componente de control por posición, Navigator

Como se ha explicado anteriormente, este es el componente responsable de la navegación del drone. Consta de tres módulos principales que se han desarrollado en distintas iteraciones del ciclo de vida, siendo Interfaces el primero en ser desarrollado y Pilot el último y más complejo.

Existen tres flujos de entrada de información al sistema, siendo uno de éstos opcional. Estos datos son: Pose3D, la posición estimada por el componente *Cam_autoloc*; *Camera_Img*, las imágenes captadas por la cámara del drone; y Pose3DSim, que sólo se utiliza en simulación de forma opcional y corresponde a la posición verdadera enviada por el plugin del simulador. Para los datos de salida tenemos: CMDVel, correspondiente a los comandos de velocidad enviados al drone; y Extra, con los que se envían instrucciones específicas al drone como el despegue o aterrizaje.

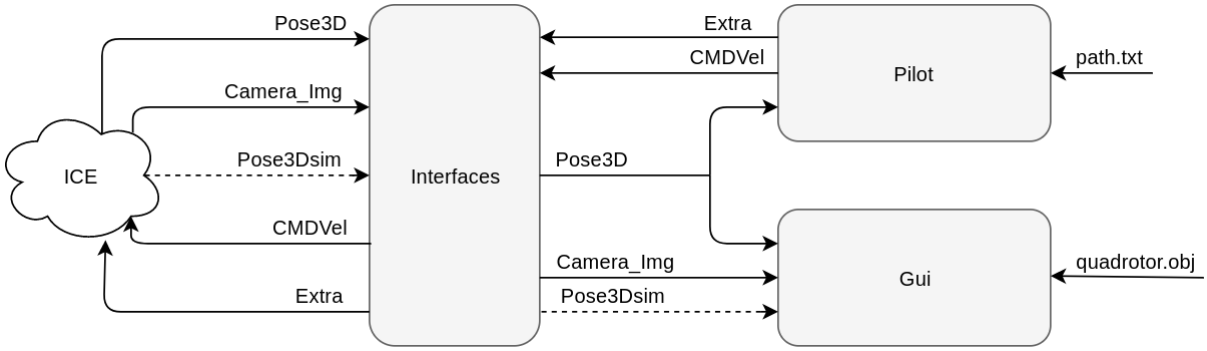


Figura 4.5: Diagrama Navigator

Además, el sistema hace uso de tres ficheros. El primero de éstos es `path.txt`, en el que figura una lista con todos los puntos del espacio que forman la ruta a seguir. El segundo es `quadrotor.obj`, un fichero que contiene la información de la estructura en 3D y las texturas de un drone para su renderización en la interfaz gráfica. Finalmente tenemos `nav.cfg`, en el que se definen los parámetros necesarios para establecer las interfaces ICE.

Para cada uno de los módulos creados se lanza un hilo de ejecución asociado, que se encarga de llamar a la función principal de cada objeto de forma periódica. Los hilos se

ejecutan en intervalos de tiempo distintos y de forma asíncrona, por lo que es necesaria una gestión de las regiones críticas en el uso de los recursos compartidos, siendo Interfaces el responsable de esta gestión.

El componente se puede lanzar de forma configurable según sea un entorno real o simulado. En simulación se añade una interfaz ICE para la recepción de los datos Pose3D correspondientes a la posición verdadera del drone dentro del entorno simulado. Estos datos son enviados por el plugin del quadrotor en Gazebo y son utilizados para calcular y mostrar en gráfica el error entre la posición real y la estimada. Para entornos reales no se dispone de la posición absoluta por lo que únicamente se mostrará una gráfica del error de la posición del drone con respecto a la ruta.

4.3.1 Módulo Interfaces

Interfaces fue el primer módulo desarrollado debido a que es el esqueleto necesario para toda la comunicación del componente. En un principio se desarrolló una aplicación simple con la función recibir imágenes de vídeo y mostrarlas mediante una interfaz gráfica básica. A partir de este punto se añadieron nuevas interfaces ICE para un drone simulado en *Gazebo*.

La clase comienza creando una interfaz ICE a partir de la información proporcionada por el fichero `nav.cfg`, en el que se detallan las direcciones IP y los puertos de las distintas conexiones de las que va a hacer uso. A continuación se crean los servidores *proxy* por los que se van a recibir los flujos de información, como se detalla en la siguiente traza de código:

```
ic = EasyIce.initialize(sys.argv)
properties = ic.getProperties()

basecamera = ic.propertyToProxy("Navigator.Camera.Proxy")
self.cameraProxy = jderobot.CameraPrx.checkedCast(basecamera)

if not self.cameraProxy:
   print ('Interface camera not connected')
```

El módulo posee una variable por cada interfaz en la que se almacena el dato recibido mediante llamadas periódicas al método `update()` a través del hilo correspondiente. Para cada llamada a `update()` se hace uso de un mutex creado con la clase `Lock` de forma que no haya problemas en el acceso de los recursos compartidos. También se han implementado los correspondientes métodos `get()` para el acceso a los datos, en los que se vuelve a utilizar el mismo mutex para evitar conflictos en las regiones críticas.

```
def update(self):
     self.lock.acquire()
     self.updateCamera()
     self.updateNavdata()
     self.updatePose()
     self.lock.release()

 def updateCamera(self):
     if self.cameraProxy:
         self.image = self.cameraProxy.getImageData("RGB8")
         self.height= self.image.description.height
         self.width = self.image.description.width

def getImage(self):
     if self.cameraProxy:
         self.lock.acquire()
         img = np.zeros((self.height, self.width, 3), np.uint8)
         img = np.frombuffer(self.image.pixelData, dtype=np.uint8)
         img.shape = self.height, self.width, 3
         self.lock.release()
         return img;
     else:
         return None
```

4.3.2 Módulo GUI

Este módulo implementa la clase Gui, responsable de la interfaz gráfica del componente 4.6. Son distintas funciones las que realiza este módulo, comenzando por la representación de un mundo en 3D implementado con OpenGL en el que se muestra la posición del drone con respecto a un eje de coordenadas, la ruta de navegación establecida, la estela del movimiento del cuadricóptero y, en caso de simulación, la estela de la posición verdadera. También muestra las imágenes captadas por la cámara del dispositivo y permite la teleoperación del mismo mediante una serie de botones con los que se puede pausar y reanudar el movimiento, así como enviar la orden de aterrizaje o despegue. Finalmente, la interfaz dispone de una gráfica para la ejecución en entorno real y dos para simulación para la monitorización del error.

La interfaz está implementada con *PyQt*, el binding para *Python* de *Qt*, con el que se crea una ventana principal a la que se añaden los distintos *widgets*. Al igual que el resto de módulos, tras la creacion del objeto Gui se asocia a un hilo que realiza llamadas al método `update()`, mediante el cual se actualizan los valores de los recursos compartidos y se refresca la información mostrada en la interfaz.

El mundo en 3D consta de elementos estáticos (el eje de coordenadas y la ruta a seguir) y dinámicos (la figura del cuadricóptero y las estelas de movimiento). Para las estelas de movimiento se añaden los datos de posición recibidos en cada iteración a un buffer circular. Originalmente las estelas se almacenaron en un array, aunque finalmente se optó por la implementación de un buffer circular con motivo de aliviar la carga computacional y permitir su reutilización.

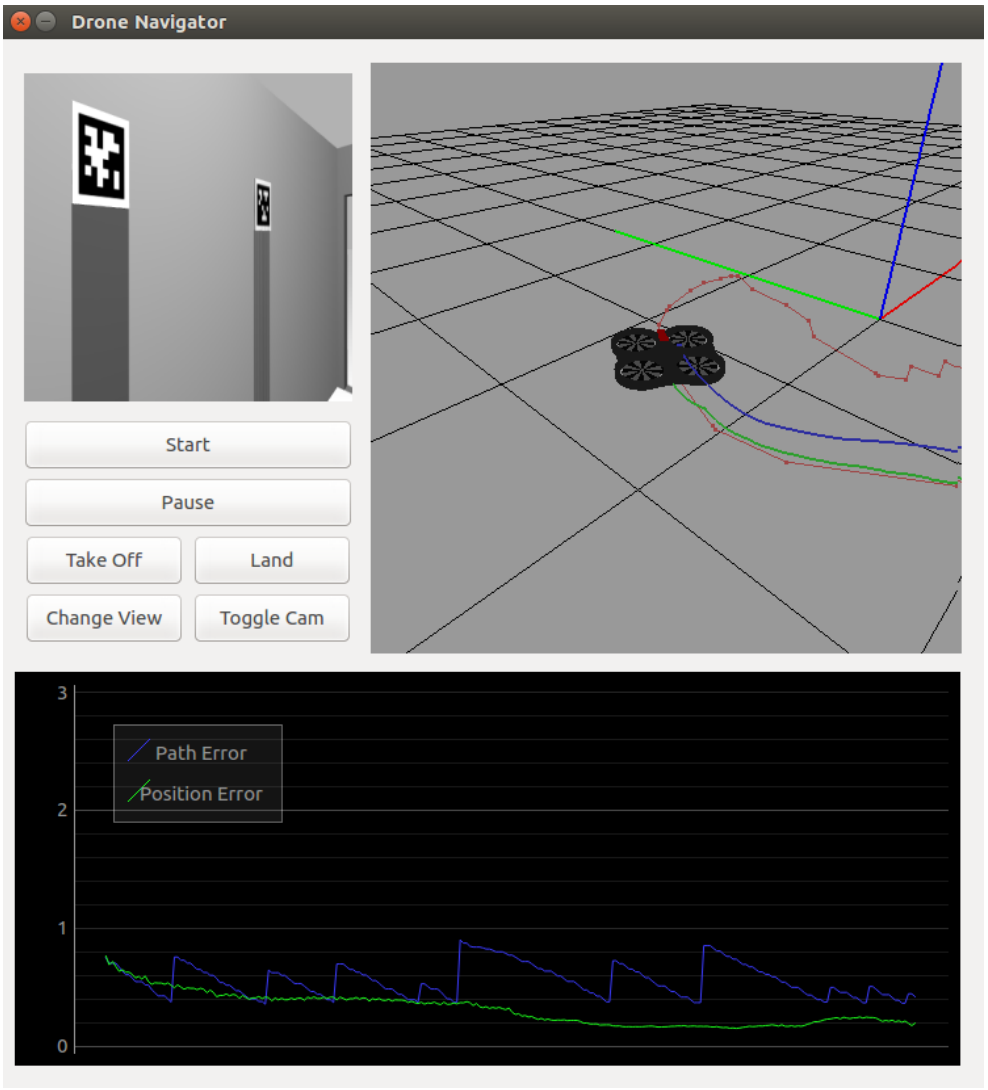


Figura 4.6: Interfaz gráfica del componente Navigator

La clase `RingBuffer` implementada parte de la idea de que el buffer funciona de forma distinta según esté o no lleno. De esta manera se define una segunda clase `Full`, a la que cambia el objeto instanciado una vez esté lleno. La clase posee los métodos `append()`, con el que se añade un nuevo dato al buffer, `get(x)`, mediante el que se obtiene el objeto en la posición dada, y `getlen()`, que devuelve el número de espacios ocupados en el buffer. La diferencia entre las dos clases radica en la adición de la variable `cur` que representa un cursor responsable de marcar la posición del último dato insertado.

```
class RingBuffer:
   #Clase que implementa un buffer sin llenar
   def __init__(self,size_max):
      self.max = size_max
      self.data = []

   class __Full:
      #Clase que implementa un buffer lleno
      def append(self, x):
          self.data[self.cur] = x
            self.cur = (self.cur+1) % self.max
      def get(self,x):
            return self.data[(self.cur+x) % self.max]
      def getlen(self):
            return self.max

   def append(self,x):
      self.data.append(x)
      if len(self.data) == self.max:
          self.cur = 0
          #Cambia su clase de non-Full a Full
          self.__class__ = self.__Full

    def get(self,x):
        #Devuelve el dato en la posicion x
        return self.data[x]

    def getlen(self):
        return len(self.data)
```

Para el renderizado del cuadricóptero se ha hecho uso de los archivos *Collada* disponibles en *JdeRobot*, que son los utilizados en la simulación con *Gazebo*. Inicialmente se intentó renderizar a partir de los archivos `.dae` de Collada mediante la librería *PyCollada* de *Python* cuya función es la de importar este tipo de archivos. Sin embargo, a pesar de que el renderizado de este formato con *OpenGL* daba muy buen resultado a nivel de calidad gráfica y coste computacional, su implementación resultaba compleja.

Finalmente se optó por exportar los archivos a un formato `.obj`, que resulta mucho más sencillo de renderizar y, aunque no tenga el mismo potencial de calidad que el formato anterior, los resultados son más que suficientes para la función que debe desempeñar. La importación del fichero se lleva a cabo mediante la clase `OBJFile` encargada de traducir el

contenido del fichero al tipo de datos manejable por *OpenGL*. Esta clase implementa un objeto donde se almacenan los datos del modelo 3D y que dispone de los métodos necesarios para su renderizado. La clase fue desarrollada con fines académicos y está liberada bajo una licencia MIT[3].

Otra de las funcionalidades de la interfaz es la de permitir la teleoperación del drone y el comportamiento de las cámaras. Para ello se dispone de una serie de botones implementados mediante PyQt. En primer lugar tenemos los botones `Start` y `Pause` mediante los cuales se puede pausar o reanudar el vuelo del drone. Esto se realiza modificando una variable booleana de Interfaces `self.pause` que en caso de ser positiva las velocidades enviadas al drone mediante `CMDVel()` son nulas. A parte de estos dos botones, tenemos `TakeOff` y `Land`, que llaman a la correspondientes funciones de Interfaces para enviar las órdenes de maniobra de despegue o aterrizaje al drone a través de la interfaz `ArDroneExtra`. Los botones restantes corresponden a `CameraView`, mediante el cual se modifica el tipo de vista del mundo 3D pudiendo alternar entre una vista general enfocada al eje de coordenadas o una vista subjetiva enfocada al drone; y `ToggleCam`, con el que alternamos entre las distintas cámaras que tenga el drone.

Por último en la interfaz se encuentra una gráfica a tiempo real en la que se indica el error entre la posición estimada y la ruta especificada en metros, y en caso de entorno simulado, también el error entre la posición estimada y la posición real del drone expresado en metros. Estos errores se calculan en cada iteración de la clase Gui y se almacenan en un array de datos que es mostrado en la gráfica mediante la librería PyQtGraph. El error entre posición estimada y real es calculado como la distancia entre esos dos puntos, mientras que el error entre posición y ruta es más complejo de calcular debido a que esta útlima está formada por una serie de puntos.

[3]https://opensource.org/licenses/MIT

4.3.3 Módulo Pilot

Pilot es el módulo clave para todo el funcionamiento del componente, ya que es el que realiza todas las tareas de pilotaje del drone. Como se ha explicado previamente, *Pilot* sólamente recibe una estimación de posición calculada en Cam_autoloc y a partir de estos datos genera una serie de velocidades que va enviando al cuadricóptero mediante la función `CMDVel()`. La navegación del cuadricóptero se realiza en función de una ruta definida a partir de una serie de puntos en coordenadas cartesianas.

El primer paso para desarrollar el sistema de navegación fue el de creación de una ruta. Para ello, en un primer momento, se introdujeron las coordenadas de dicha ruta de forma manual. Sin embargo, a medida que el sistema de navegación fue adquiriendo complejidad se hizo necesaria la búsqueda de un nuevo método para crear rutas más complejas y reales.

La solución la ofrecían los componentes *Recorder* y *Replayer*, que forman parte de las herramientas ofrecidas por *JdeRobot*. Sin embargo, el tipo de dato usado por este componente era `Pose3DEncoders`, un tipo con funciones similares a las de `Pose3D` pero que había quedado obsoleto en las nuevas versiones de *JdeRobot*. Debido a esto se tuvo que realizar un proceso de refactorización de ambos componentes, para su posterior uso e integración en los repositorios oficiales de *JdeRobot*. De forma adicional, se implementó una pequeña herramienta en *Python* que recibía los datos proporcionados por *Replayer* y los almacenaba en un fichero de texto. Esta herramienta traduce el tipo de datos `Pose3D` a un formato de texto legible y solo escribe en el fichero los datos de posición sin incluir los ángulos de orientación.

Inicialmente se desarrollaron varios sistemas simples de navegación que se basaban en los vectores de dirección entre la posición del drone y el punto de ruta correspondiente. Esto provocaba cambios bruscos en los controles de dirección y por consecuencia, la desestabilización del sistema. Como solución se propuso el desarrollo de un sistema de navegación predictivo enfocado al control del ángulo de giro.

De esta forma se partió de la premisa de que la velocidad lineal del drone iba a ser constante y la única velocidad variable sería la velocidad angular en torno al eje Z. De la

misma forma que las velocidades lineales a tener en cuenta iban a ser las correspondientes a los ejes X y Z del drone, descartando la velocidad en Y debido a que con la predicción de giro no sería necesario el movimiento lateral. Así, el control por predicción se realizará únicamente en la componente de velocidad horizontal y en la velocidad angular de giro, mientras que las componentes verticales se tratarán de forma reactiva. Teniendo estos precedentes en cuenta, el algoritmo se explica a continuación.

En primer lugar, se calcula el vector que apunta desde la posición del cuadricóptero hasta el siguiente punto de ruta que se desea alcanzar, obteniendo el vector $\vec{V}$. A continuación se calcula el vector unitario $\vec{u}_v$, que más adelante será utilizado para obtener las componentes horizontal y vertical.4.11

$$\begin{aligned} \vec{V} &= \vec{Path} - \vec{Pose} \\ \vec{u}_v &= \frac{\vec{V}}{|\vec{V}|} \end{aligned} \tag{4.11}$$

Para obtener las magnitudes de las dos velocidades lineales se parte del vector unitario calculado anteriormente. Para ello primero se ha de descomponer en las dos componentes correspondientes a las velocidades lineales, la componente vertical se obtiene directamente del valor de la componente en el eje Z del vector $\vec{u}_v$ y para la horizontal se calcula el módulo de las componentes X e Y de $\vec{u}_v$. Estas dos magnitudes serán multiplicadas por la constante v_k que es la velocidad lineal total a la que viajará el cuadricóptero, obteniendo así v_x como velocidad horizontal y v_z como velocidad vertical.

$$\begin{aligned} v_x &= |\vec{u}_{vxy}| * v_k \\ v_z &= |\vec{u}_{vz}| * v_k \end{aligned} \tag{4.12}$$

El siguiente paso es calcular la distancia horizontal que recorrerá el drone hasta la siguiente iteración del algoritmo para de esta forma poder predecir la posición que tendrá. La distancia d_τ se obtiene multiplicando la velocidad horizontal obtenida anteriormente por el tiempo que transcurre entre dos iteraciones (4.13).

$$d_\tau = v_x * \Delta_t \tag{4.13}$$

Para predecir la posición del cuadricótpero debemos tener en cuenta el ángulo de giro con respecto al eje Z del drone en ese instante. Ya que en el estándar de Pose3D la orientación se indica en cuaterniones, es necesario hacer una conversión a ángulo de Euler (4.14). Una vez obtenido el ángulo θ_z, la posición predicha se obtiene mediante las ecuaciones (4.15). Como se puede observar, no se tiene en cuenta el cambio del altura del drone ya que no influye en el cálculo del ángulo de giro en Z.

$$\theta_z = \arctan^2\left(\frac{2 * (q_0 * q_3 + q_1 * q_2)}{1 - 2 * (q_2^2 + q_3^2)}\right) \tag{4.14}$$

$$\begin{aligned} X_f &= d_\tau * \cos\theta_z + x_{pose} \\ Y_f &= d_\tau * \sin\theta_z + y_{pose} \end{aligned} \tag{4.15}$$

Finalmente, para obtener la compensación de error es necesario calcular el futuro error lateral con respecto a la ruta, L_{fe}, que el drone va a tener si mantiene el ángulo de giro actual. Para ello se calculan las distancias en los ejes X e Y entre el punto de ruta a seguir y la posición predicha y se le aplica el ángulo de giro (4.16).

$$L_{fe} = -\sin\theta_z * (X_{path} - X_f) + \sin\theta_z * (Y_{path} - Y_f) \tag{4.16}$$

También es necesario obtener el error de ángulo, que es la diferencia entre el ángulo actual y el ángulo sobre el eje Z existente entre el drone y el punto de ruta, θ_e(4.17). Teniendo ángulo se calcula una velocidad angular a partir de la distancia horizontal al punto, d_H, y la velocidad lineal actual, v_x (4.18).

$$\begin{aligned} \theta_{path} &= \arctan\left(\frac{V_y}{V_x}\right) \\ \theta_e &= \theta_{path} - \theta_z \end{aligned} \tag{4.17}$$

$$d_H = \sqrt{V_x^2 + V_y^2}$$
$$\delta_e = \theta_e / (\frac{d_H}{v_x}) \tag{4.18}$$

La velocidad angular dependerá entonces de la velocidad angular calculada y de un factor de correción que viene dado por la relación entre el error lateral predicho y la velocidad horizontal. Este factor está multiplicado por una constante K_g, que es la ganancia de correción del giro (4.19). De esta forma obtenemos un control de giro que se adapta al error futuro y cuya ganancia de adaptación podemos modificar experimentalmente mediante la constante K_g.

$$\delta_z = \delta_e + K_g * (L_{fe}/v_x) \tag{4.19}$$

Una vez obtenidas las tres velocidades que se van a enviar al drone, velocidad horizontal v_x, velocidad vertical v_z y velocidad angular δ_z, se aplica una fusión temporal sobre ésta última. Esta fusión consiste en un filtro por pesos en el que se almacenan las 4 últimas velocidades angulares y se asigna un mayor peso a las velocidades más recientes. De esta forma se asegura una mayor robustez ante los fallos de orientación provenientes de *Cam_ autoloc*. Finalmente las órdenes se envían al drone mediante una llamada al método `SendCMDVel(vx,vy,vz,wz)` de Interfaces, que se ocupa de mandar las órdenes decididas al cuadricóptero.

4.4 Integración del sistema y ajustes

El primer paso para la integración del sistema fue la refactorización del componente *Cam_ autoloc*. Muchas de las versiones de librerías y herramientas que el componente utilizaba habían quedado obsoletas con respecto a las del resto de bibliotecas utilizadas en *JdeRobot*. Hubo que modificar llamadas a métodos y tipos de datos usados por varias de estas librerías, concretamente *OpenCV*, *Qt* y *OpenGL*. Finalmente, se adaptó el código

a los cambios introducidos en las últimas versiones, sin embargo, todavía no era posible compilar el componente.

Para la elaboración del componente *Cam_ autoloc* se había utilizado la herramienta de código libre *Qt Creator* utilizando un compilador *qmake*. Esto interfería con el estándar de *JdeRobot*, que usa el sistema *cmake*. Tras varios intentos de compilar el componente mediante *cmake*, se optó por usar el compilador original. Hubo que revisar el archivo de compilación generado por *Qt Creator* y modificar diversos parámetros hasta conseguir la correcta compilación del componente.

Por otra parte, tambien fue necesaria una refactorización del componente *Navigator*. Durante el desarrollo, *JdeRobot* se actualizó a su versión 5.5 en la que se pasó de usar la versión 2.7 de *Python* a la versión 3.0. Debido a esto se tuvo que portar el dódigo de *Python2* a *Python3*, modificando llamadas a funciones que habían quedado obsoletas y el acceso a algunos tipos de datos.

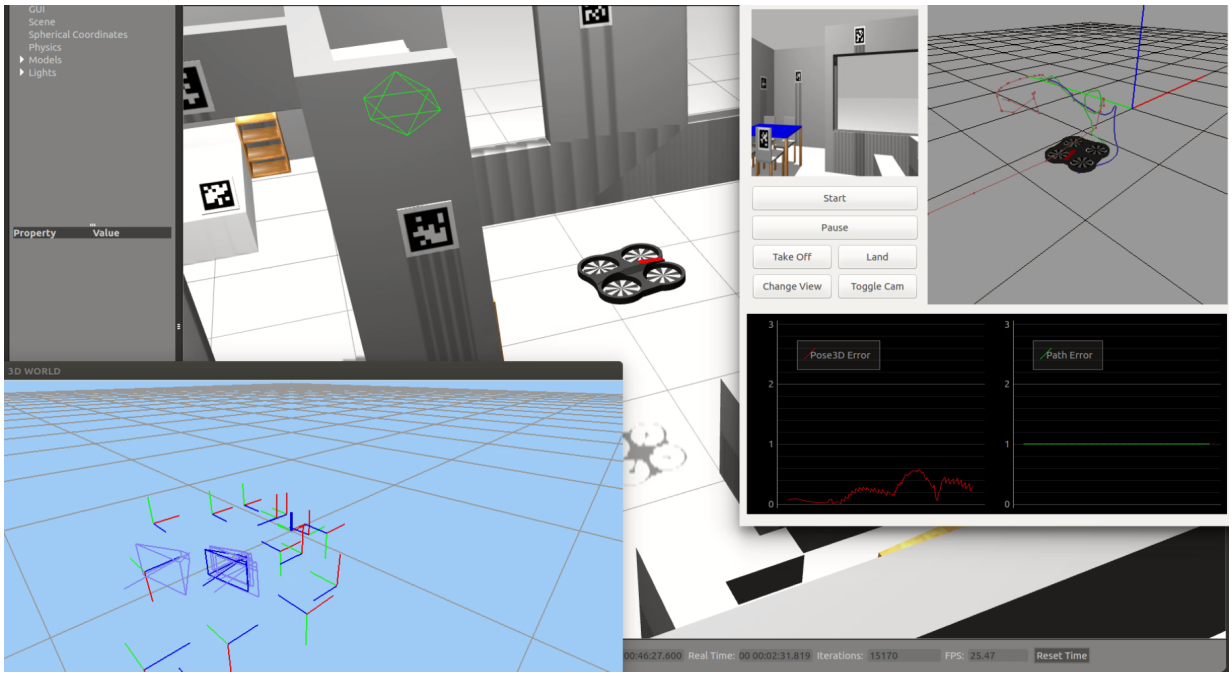


Figura 4.7: Sistema completo en funcionamiento

Para poder integrar las dos herramientas fue necesario realizar varias modificaciones a *Navigator*. Para empezar, hubo que añadir las correspondientes interfaces ICE para así acceder a la posición estimada por el compontente *Cam_ autoloc*. También se añadió una nueva gráfica que mostraba el error de posición estimada, calculado como la distancia entre el punto de posición absoluta del cuadricóptero y el punto de posición estimado. Análogamente, en el mundo 3D se añadió un punto que marcaba la posición verdadera además de su correspondiente estela.

En *Cam_ autoloc* también se realizaron varias modificaciones, añadiendo nuevas funcionalidades. El componente original sólo mostraba en su mundo 3D la posición final estimada, por lo que se consideró conveniente mostrar además las estimaciones para cada baliza, para así poder analizar en profundidad el comportamiento del sistema. Para ello se instancia un nuevo objeto `TPinHoleCamera` en el momento en el que se calcula la posición de la cámara para cada marcador. Estos objetos son almacenados en un array que posteriormente son renderizados en el módulo *World*. Además, se añadió un nuevo criterio para la asignación de pesos en el filtro de fusión espacial. Este criterio consistía en calcular la posición media de las posiciones estimadas por las balizas y asignar mayor peso según la cercanía a esta media. Sin embargo, el critero demostró no tener apenas impacto en la fase de fusión espacial debido a que el número de balizas medio detectado por la cámara era de 2 o 3 balizas. Finalmente se optó por descartar este método para aliviar la carga computacional del componente.

Capítulo 5

Experimentos

En este capítulo se describen los experimentos realizados durante las fases de desarrollo y los realizados para validar el prototipo final. En primer lugar ser realizaron diversos experimentos en simulación de ambos componentes por separado a través de *Gazebo*. Estos experimentos servían para probar y ajustar tanto el algoritmo de pilotaje como el algoritmo de autolocalización visual. Una vez validado el comportamiento unitario de cada componente se ejecutaron experimentos sobre el sistema completo. Finalmente, se evaluó el comportamiento del sistema en un entorno real.

5.1 Experimentos en simulación

Para realizar los experimentos se ha hecho uso del entorno de simulación *Gazebo*. El plugin para la simulación del comportamiento del drone así como el modelo necesario para la renderización del vehículo y sus sensores están incluidos en JdeRobot. Sin embargo, fue necesaria la elaboración de un mundo en 3D que reprodujera un entorno de espacios cerrados. Para ello se sirvió del mundo `GrannyAnnie.world` que modela un apartamento. Además, fue necesaria la creación de los modelos de las balizas AprilTags y sus correspondientes texturas. El mundo creado se integró en los respositorios oficiales de *JdeRobot* bajo el nombre `AprilTagsFlat` y se muestra en la Figura 5.1.

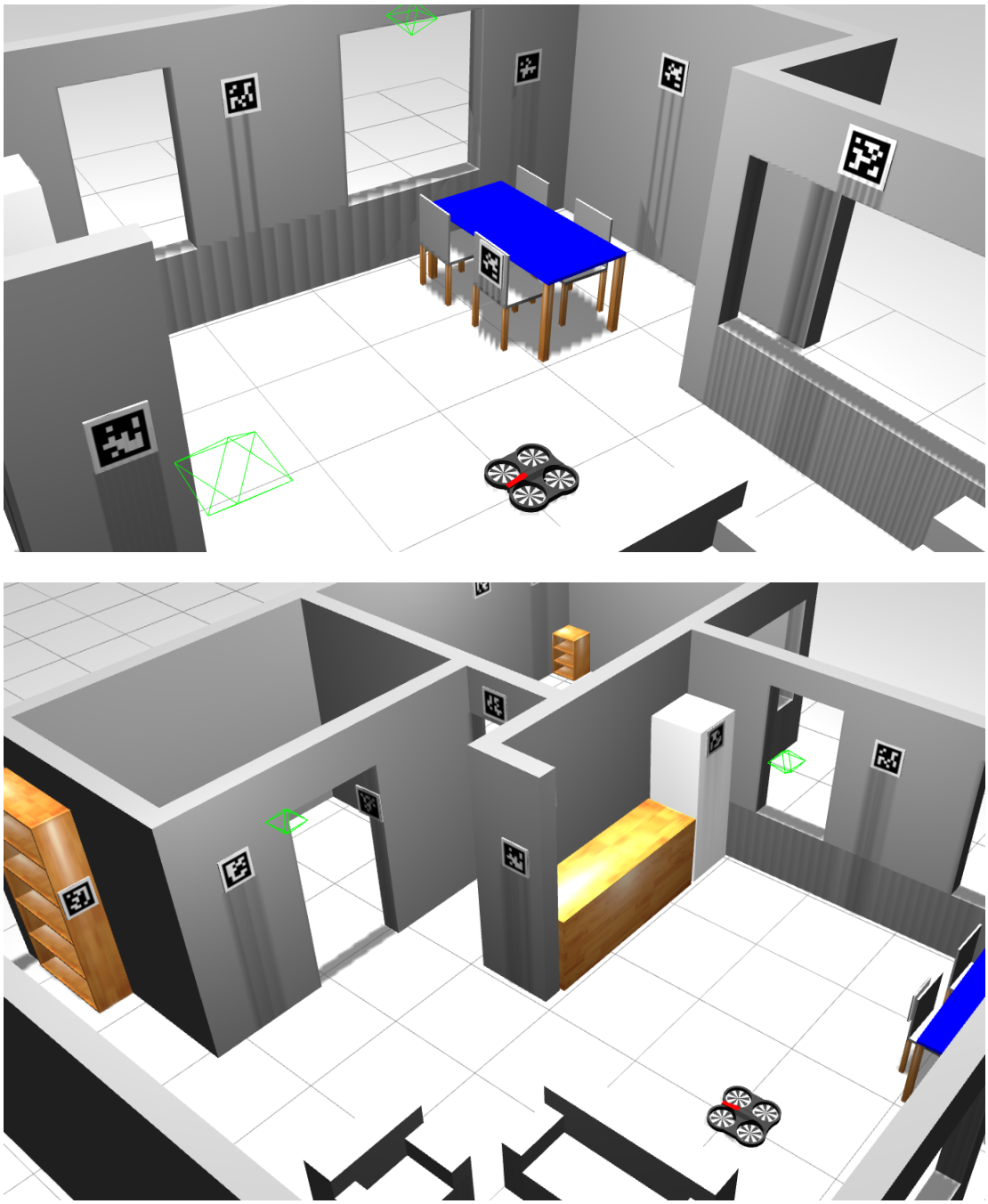


Figura 5.1: Escenario de simulación en Gazebo

5.1.1 Experimentos de control basado en posición

Los experimentos sobre el componente *Navigator* se realizaron obteniendo la posición verdadera suministrada por el plugin de *Gazebo*. La respuesta a trayectorias formadas por líneas rectas, tanto con variaciones de altura como sin ellas, dieron resultados óptimos. Sin embargo, al introducir giros el sistema presentaba comportamientos muy inestables. Tras el análsis del código se descubrió que el origen de este comportamiento estaba en las operaciones realizadas con ángulos, donde no se calculaban correctamente las diferencias entre vehículo y ruta.

Una vez solucionado, el sistema seguía teniendo un comportamiento errático, por lo que fue necesario un estudio de la relación entre la velocidad del cuadricóptero y la ganancia de giro K_g. Finalmente se llegó a la conclusión de que el rango de velocidades para el funcionamiento estable del sistema de navegación era de [0.5-2.5]m/s. Del mismo modo, los valores de ganancia de giro debían estar comprendidos entre el rango [0.1-0.3]. Se observó que valores de ganancia que sobrepasaban este rango provocaban un comportamiento muy inestable, haciendo oscilar al vehículo, mientras que valores inferiores no permitían ajustar el giro de forma correcta, por lo que el error se acumulaba progresivamente. Este valor de ganancia debe ser proporcional a la velocidad lineal del vehículo de forma que cuanto mayor sea la velocidad, mayor correción de giro habrá que introducir.

Finalmente, tras ajustar los parámetros necesarios, se pudo validar la estabilidad de la navegación del cuadricóptero. Aún así, la aplicación presenta algunas limitaciones en cuanto a la velocidad y la complejidad de la ruta definida. Si la ruta posee giros muy cerrados o movimientos bruscos, la navegación no es fiable a más de 1.5m/s.

En las siguientes figuras se pueden observar los experimentos realizados con distintos tipos de rutas. La línea verde corresponde a la estimación de posición del componente *Cam_ autoloc*, que en este caso no está siendo utilizada para la navegación. En azul se muestra la posición verdadera del vehículo, que es la utilizada por *Navigator* para tomar las decisiones de los comandos de velocidad en este caso. Por último, en rojo se muestra la ruta a seguir.

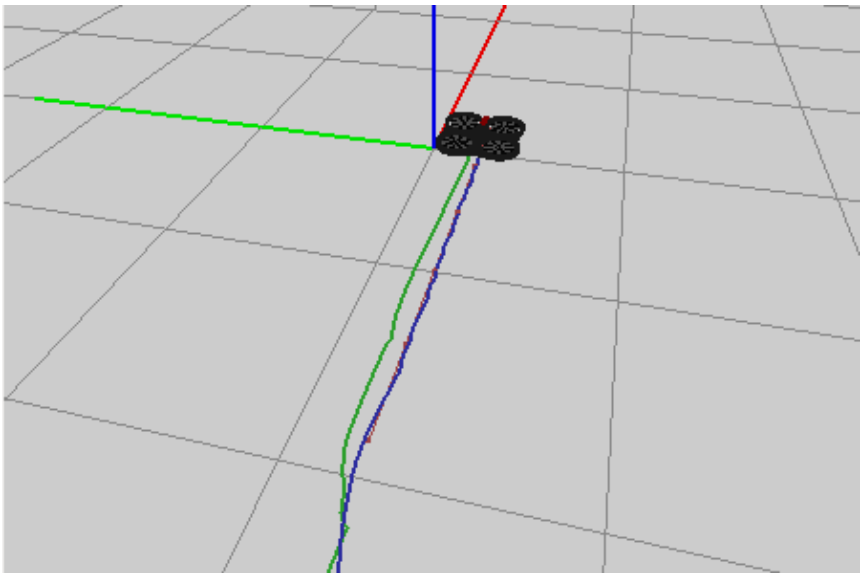


Figura 5.2: Navegación del drone para una ruta rectilínea

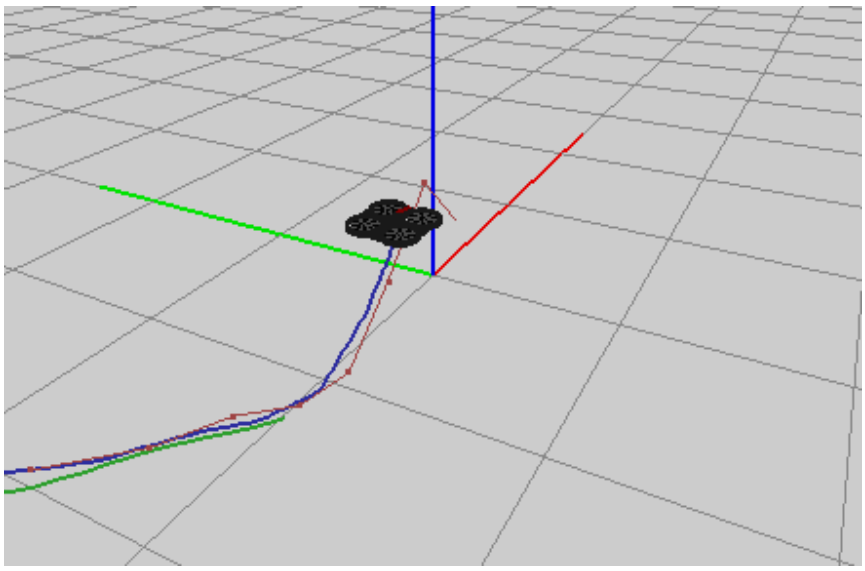


Figura 5.3: Navegación del drone para una ruta recta con cambios de altura

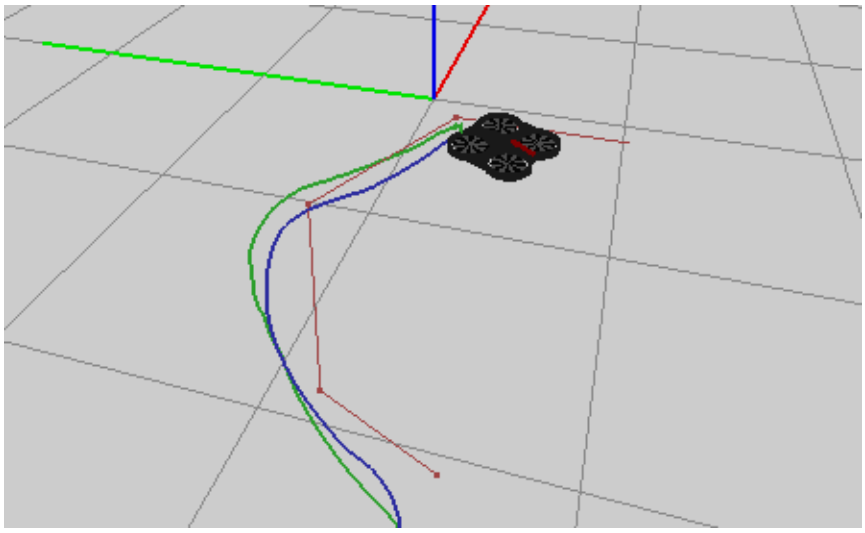


Figura 5.4: Navegación del drone para una ruta con giro

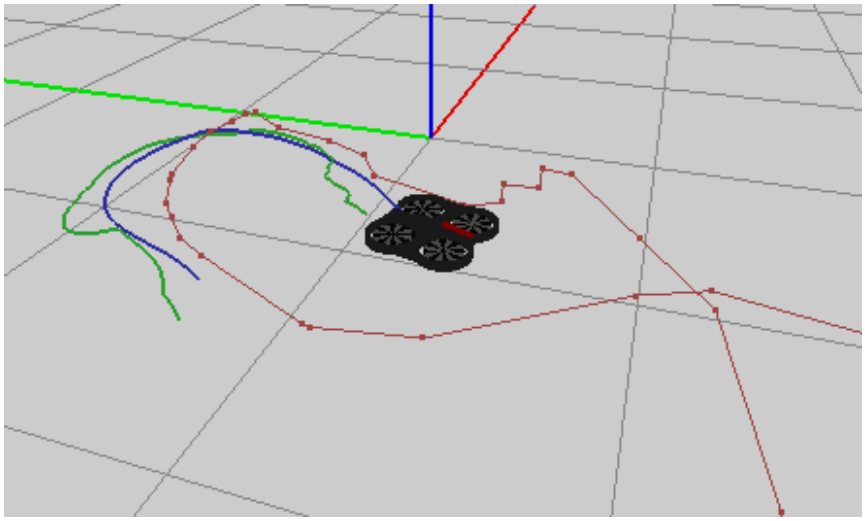


Figura 5.5: Navegación del drone para una ruta compleja

Como se puede observar, la respuesta del vehículo ante rutas rectas es óptima. En la Figura 5.2 el drone muestra un ajuste perfecto a la ruta, en la estela de movimiento también se observa cómo realiza un pequeño giro para situarse de frente a la ruta. Los

resultados obtenidos con una ruta recta con cambios de altura, Figura 5.3, son también óptimos.

En cuanto a las rutas con giros, el vehículo ya no presenta una navegación tan fuertemente ajustada a la ruta. En la Figura 5.4, la estela de movimiento demuestra una distancia mínima con respecto a los puntos de la ruta. También se pueden observar los aumentos de velocidad angular en torno al tercer punto originados por la corrección de giro. Finalmente, en la Figura 5.5 el cuadricóptero presenta un comportamiento menos ajustado que en los casos anteriores aunque igualmente estable. Esto es debido al margen de error de 20cm. establecido, lo que permite al vehículo realizar maniobras más amplias para evitar giros abruptos.

5.1.2 Experimentos de localización

Los primeros experimentos sobre este componente mostraron resultados totalmente erróneos, las estimaciones mostraban distancias y ángulos incorrectos. Tras hacer un examen en profundidad del código del componente se detectaron varios fallos. El primero de ellos fue la extracción de parámetros intrínsecos y extrínsecos del fichero de calibración de la cámara. El componente extraía estos parámetros y los almacenaba en variables, sin embargo, los parámetros que eran utilizados para la implementación del objeto `TPinHoleCamera` eran constantes que ya estaban definidas anteriormente. Una vez hechas las modificaciones pertinentes para que el componente instanciara los objetos con los parámetros correctos, la estimación de la posición mostraba mejores resultados.

A pesar de esto, la detección de algunas balizas seguía siendo inestable. En un principio se achacó el problema a un cálculo de posición defectuoso debido a la simetría presente en algunas balizas. Sin embargo, tras la realización de varios experimentos, se observó que el problema no provenía del algoritmo de autolocalización, sino del simulador. Las texturas asignadas a los modelos de las balizas no estaban orientadas correctamente, por lo que los ángulos de orientación de las balizas eran erróneos. Para solucionar esto hubo que reimplementar los modelos de *Gazebo* para que la orientación de las texturas se mostraran correctamente.

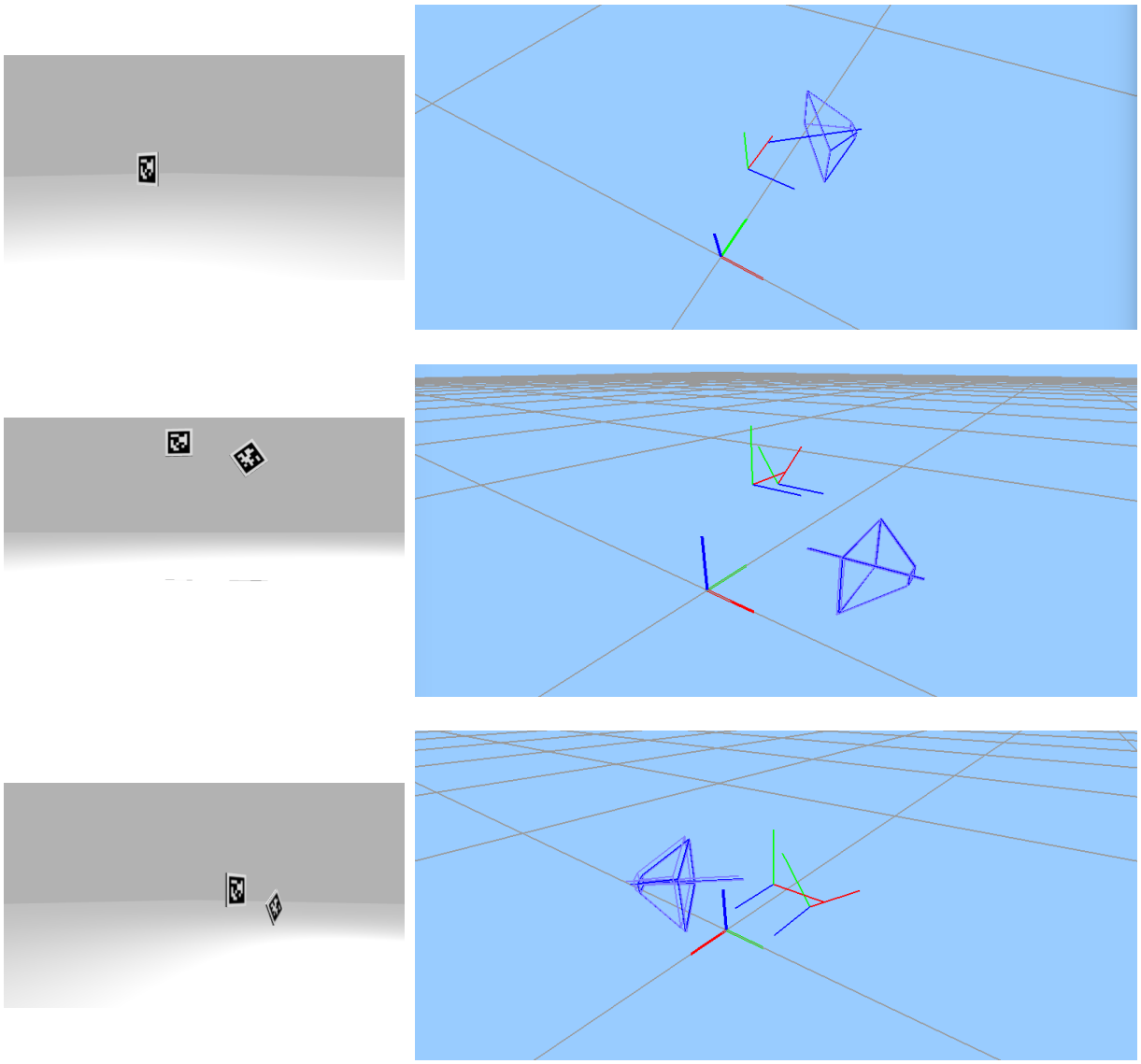


Figura 5.6: Experimentos de estimación de posición en escenario simple

En la Figura 5.6 se puede observar el funcionamiento del componente de autolocalización en escenarios simples. La posición estimada se ajusta perfectamente a la posición verdadera de la cámara.

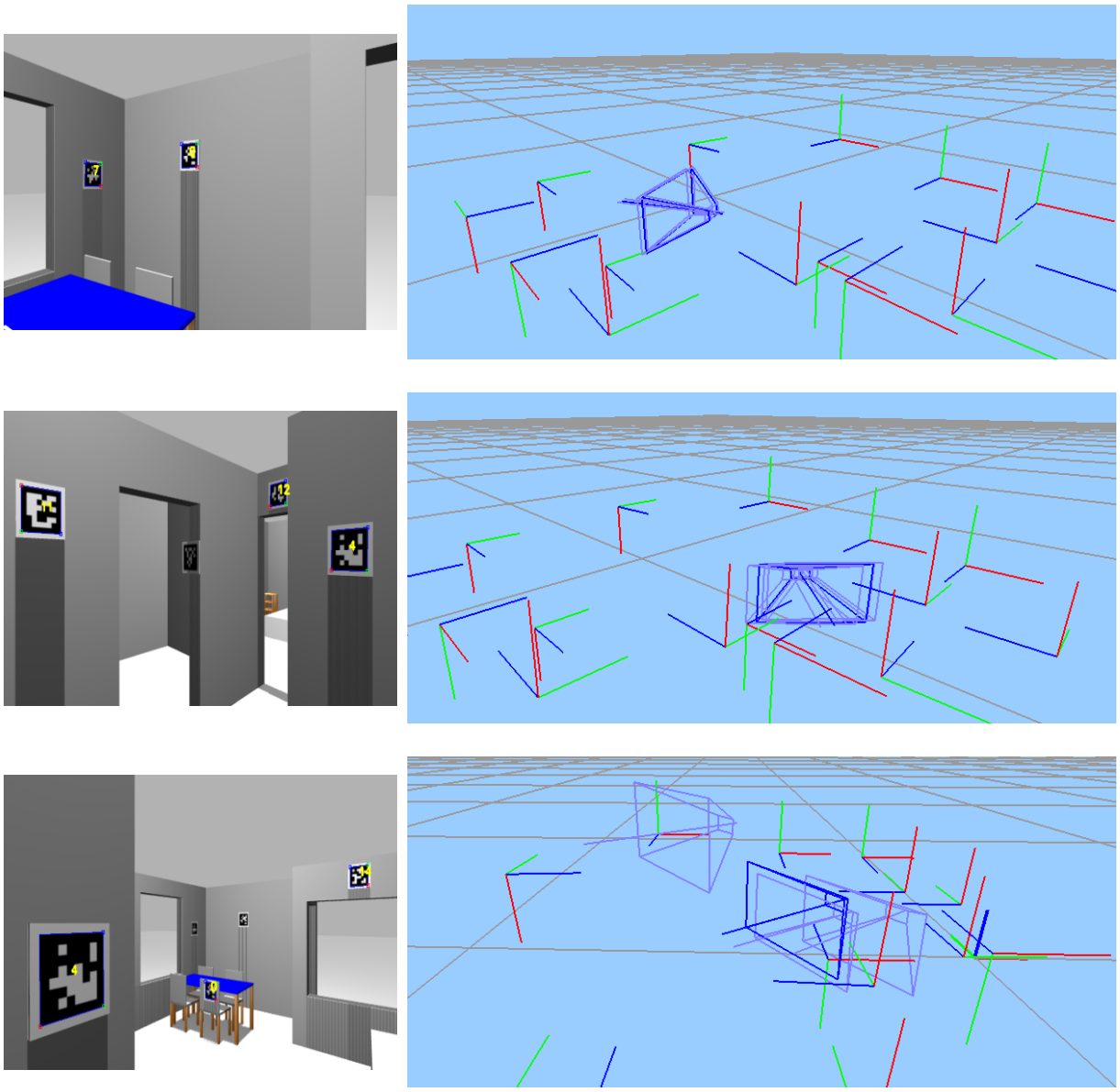


Figura 5.7: Experimentos de estimación de posición en *Gazebo*

En la Figura 5.7 los experimentos han sido realizados en el mundo `AprilTagsFlat`, donde se encuentran repartidas 15 balizas. Para el análisis del comportamiento de la estimación de posición se ha hecho uso del componente *Uav_viewer* mediante el cual se ha teleoperado el vehículo. En la primera figura se puede observar un resultado óptimo en el que las estimaciones para cada baliza (gris) coinciden, por lo que la estimación final (azul) corresponde al valor proporcionado por éstas.

En segundo lugar aparecen cuatro marcadores en cámara, sin embargo, el algoritmo sólo es capaz de detectar tres de ellos. Esto se debe a la baja iluminación que recibe la baliza no detectada, que provoca que el algoritmo de detección sea incapaz de diferenciar correctamente los colores de la baliza. En cuanto a la posición estimada, las estimaciones individuales no coinciden de forma tan ajustada como en el caso anterior. Aún así, la estimación final sigue siendo muy precisa.

Finalmente, se muestra un caso en el que existe una mayor diferencia entre la distancia de cada baliza a la cámara. Como se puede observar, la estimaciones para cada baliza están mucho menos ajustadas que en los casos anteriores. Esto es debido a la pérdida de precisión del algoritmo cuando las balizas se encuentran a más de 4 o 5 metros. La estimación final sigue dando resultados fiables gracias a la fusión espacial. En la imagen podemos observar cómo la estimación de la baliza más lejana a la cámara es la que se encuentra más alejada del resto de estimaciones. Para el cálculo de la estimación final apenas se ha tenido en cuenta el impacto de estas estimaciones debido al pequeño valor del peso asignado en comparación al valor que se habrá dado a la estimación de la baliza más cercana.

5.1.3 Pruebas con el sistema completo

Una vez se integraron ambos componentes se realizaron diversas pruebas del sistema completo. Los primeros tests consistieron en el seguimiento de rutas lineales con una única baliza colocada delante del drone. Estas pruebas dieron resutados positivos, el vehículo seguía la ruta de forma estable y con un margen de error casi nulo. Más adelante se añadieron cambios de altura en las rutas pero sin introducir ningún giro. En estas pruebas también se obtuvieron buenos resultados, aunque la estimación de posición tenía un comportamiento algo inestable. En la Figura 5.8 se puede observar cómo el vehículo se ajusta a la ruta en la medida que la maniobra de giro se lo permite. La estela azul corresponde a la posición verdadera del drone mientras que la verde corresponde a la posición estimada.

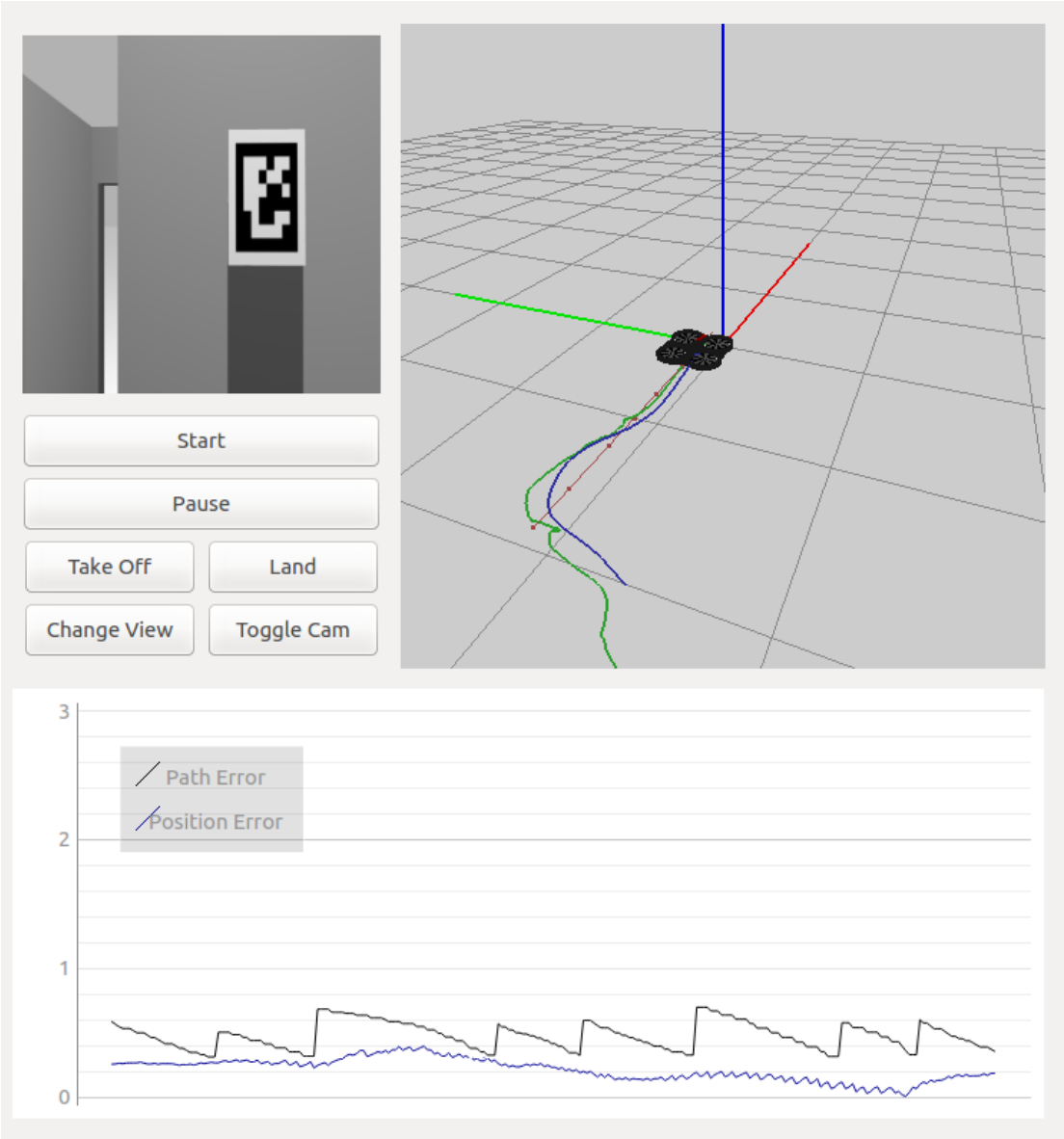


Figura 5.8: Comportamiento del sistema para una ruta lineal

A continuación, en la Figura 5.9 se realiza un experimento similar, aunque añadiendo cambios de altura en la ruta. Los resultados son similares a los anteriores, el drone se ajusta a la ruta con un error mínimo. Se puede observar que no llega a pasar por los puntos de pico, esto se debe al margen de error de 20cm. con lo que se consigue una navegación suave además de suficientemente precisa.

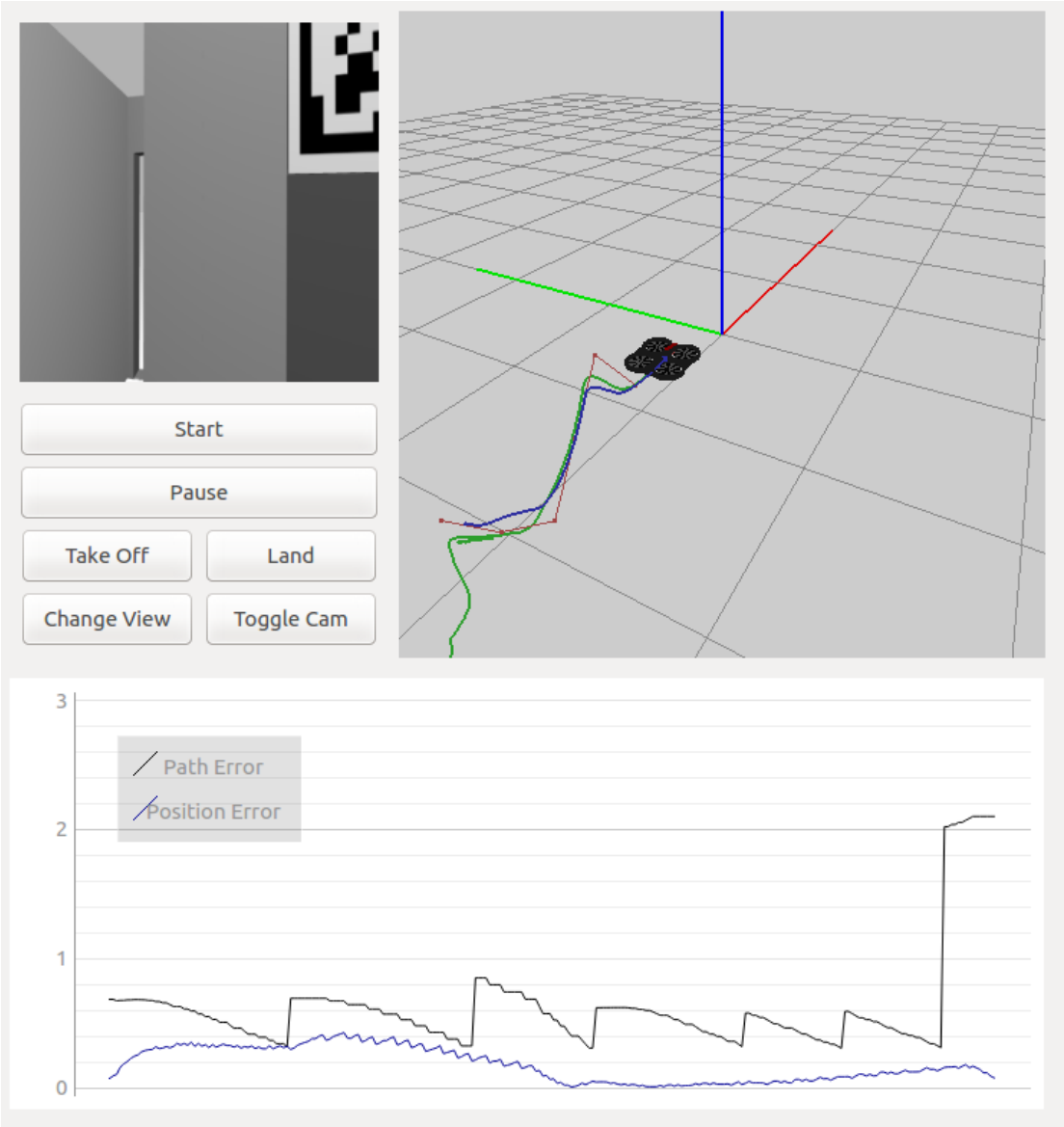


Figura 5.9: Comportamiento del sistema para una ruta recta con cambios de altura

Después de validar el comportamiento del sistema en los experimentos con rutas sencillas se realizaron varios experimentos con rutas más complejas que incluían variaciones en la altura y giros amplios y agudos. Se observaron varios comportamientos inestables en la navegación y la autolocalización. En primer lugar, la autolocalización era muy inestable principalmente en los giros y en los momentos en que las balizas entraban y salían

del campo de visión del cuadricóptero. Esto provocaba cambios bruscos en la posición estimada, y, por consiguiente, en la navegación. El sistema de control, al intentar corregir los cambios en el ángulo de giro introducidos por los fallos en la estimación de la posición, provocaba que el vehículo oscilara. A su vez, esta oscilación, provocaba más errores en la percepción de las balizas de forma que el sistema acaba desestabilizándose.

El origen de la inestabilidad radicaba en la aplicación de las fusiones temporal y espacial del componente *Cam_ autoloc*. Por ello, hubo que reajustar los valores de los pesos asignados y de las matrices de covarianza de error. Hasta ahora los experimentos realizados sobre este componente estaban basados en una cámara estática. Sin embargo, el tener una cámara en continuo movimiento añadía un grado de complejidad que no se había podido probar hasta el momento.

El problema encontrado con el filtro por pesos era que si una baliza se salía del rango de distancia asignado a un peso, variaba el valor del peso asignado y se producía un cambio brusco en la estimación. La solución propuesta fue la de añadir más rangos de distancia y reducir la diferencia de valor entre pesos, de forma que cuando se pasara de un rango a otro, el cambio en el peso fuera más suave. Además, hasta el momento, no se tenía en cuenta las balizas que se encontraban a más de 5m., de forma que se le asignaba un peso de valor 0. El problema que esto originaba era que cuando dichas balizas entraban en el campo de visión, también se producía un cambio brusco en la estimación, por lo que fue necesario asignarles también un peso mínimo.

Otro problema provenía de la aplicación del *Filtro de Kalman.*, las matrices de covarianza de error definidas hasta el momento habían servido para realizar una estimación de posición estable, teniendo en cuenta el modelo estático de las pruebas. Al introducir la variable del movimiento del vehículo, estos valores habían quedado obsoletos, pues el sistema era sensible al ruido y no se realizaba un suavizado de las posiciones obtenidas. Los valores finales se obtuvieron de forma experimental, realizando varias iteraciones en las que se iban ajustando los valores de ruido del proceso y de la medida hasta dar con los valores más apropiados.

Una vez realizadas las modificaciones sobre *Cam_ autoloc*, el sistema presentaba una notable mejora en cuanto a la estabilidad. No obstante, el sistema de control seguía siendo sensible al ruido de la estimación de posición. Tras probar con diferentes tipos de rutas, se observó que la causa de la inestabilidad eran las rutas con giros cerrados. El problema residía en la velocidad a la que las balizas entraban y salían del campo de visión del drone, que provocaban pequeñas alteraciones en la posición estimada. En un primer momento se ajustaron los valores de la fusión temporal del *Filtro de Kalman*, pero si se implementaba un modelo para medidas muy ruidosas, la estimación de la posición se alejaba mucho de los valores reales. Esto provocaba una navegación muy estable aunque el error entre la posición estimada, la posición real y la ruta eran mayores de lo aceptable.

Finalmente, hubo que reajustar los parámetros de velocidad y ganancia de giro del component *Navigator*, reduciendo la velocidad y la ganancia y además dando un margen de error de unos 30cm. Adicionalmente se ha implementado un sistema de control de ángulo de giro para dotarlo de mayor robustez ante saltos en la predicción. La solución implementada es un filtro de fusión temporal basado en pesos, de forma similar al filtro original de *Cam_ autoloc*. Este filtro guarda los últimos 5 comandos de velocidad angular y asigna pesos a los más recientes. De forma que si hay un salto en la estimación de posición y esto provoca una alteración brusca del ángulo de giro, el cambio en el control de velocidad angular es más suave. Así se evita el caso de que el vehículo quede oscilando intentando corregir los cambios, provocando la consecuente desestabilización del sistema.

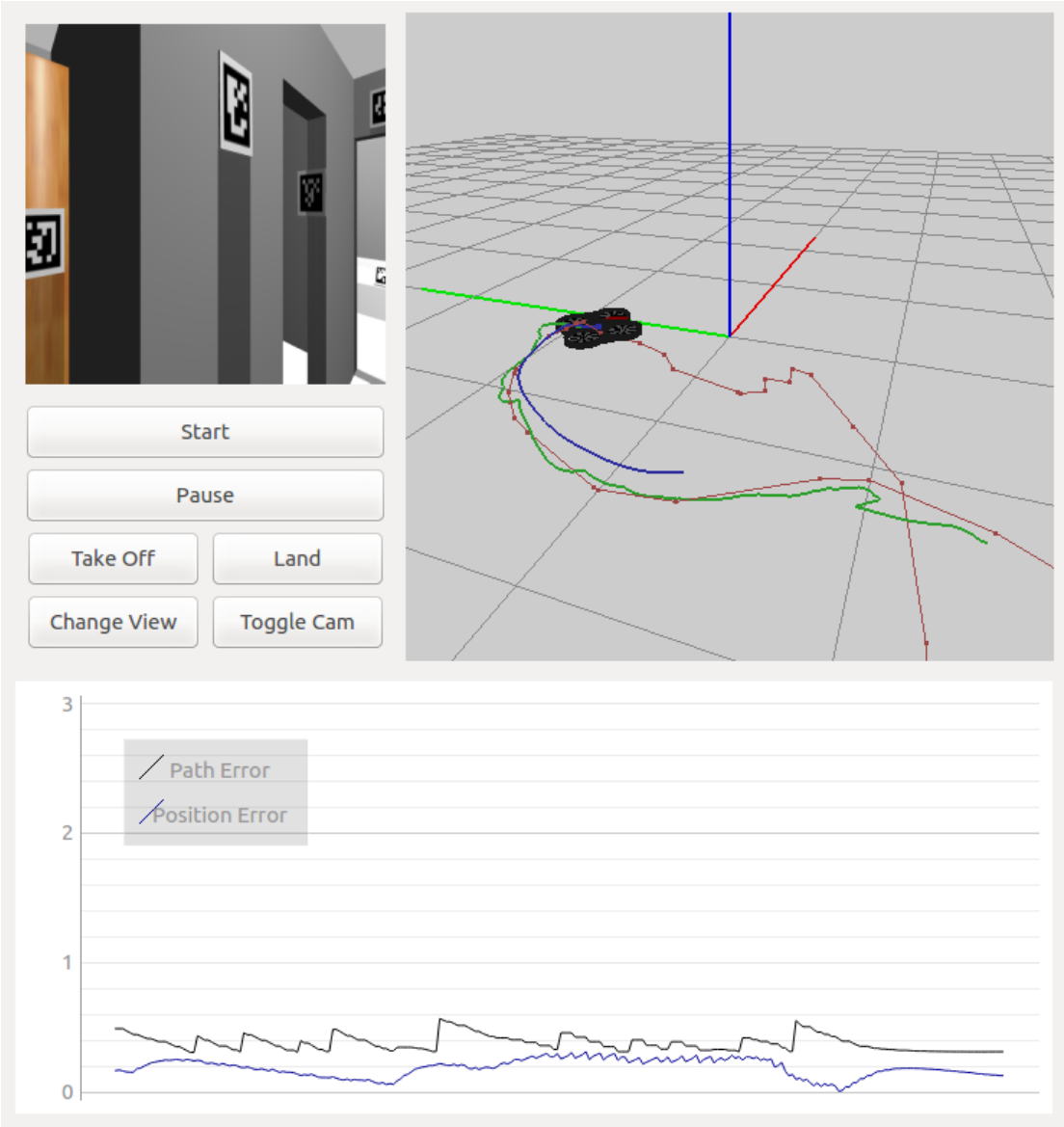


Figura 5.10: Comportamiento del sistema completo en ruta compleja (Parte 1)

En la Figura 5.10 se muestra el correcto comportamiento del sistema ante los giros en la ruta. A pesar de que existen algunos saltos en la estimación de posición (verde), la posición real del drone (azul) demuestra que el sistema de control apenas se ve afectado por el ruido.

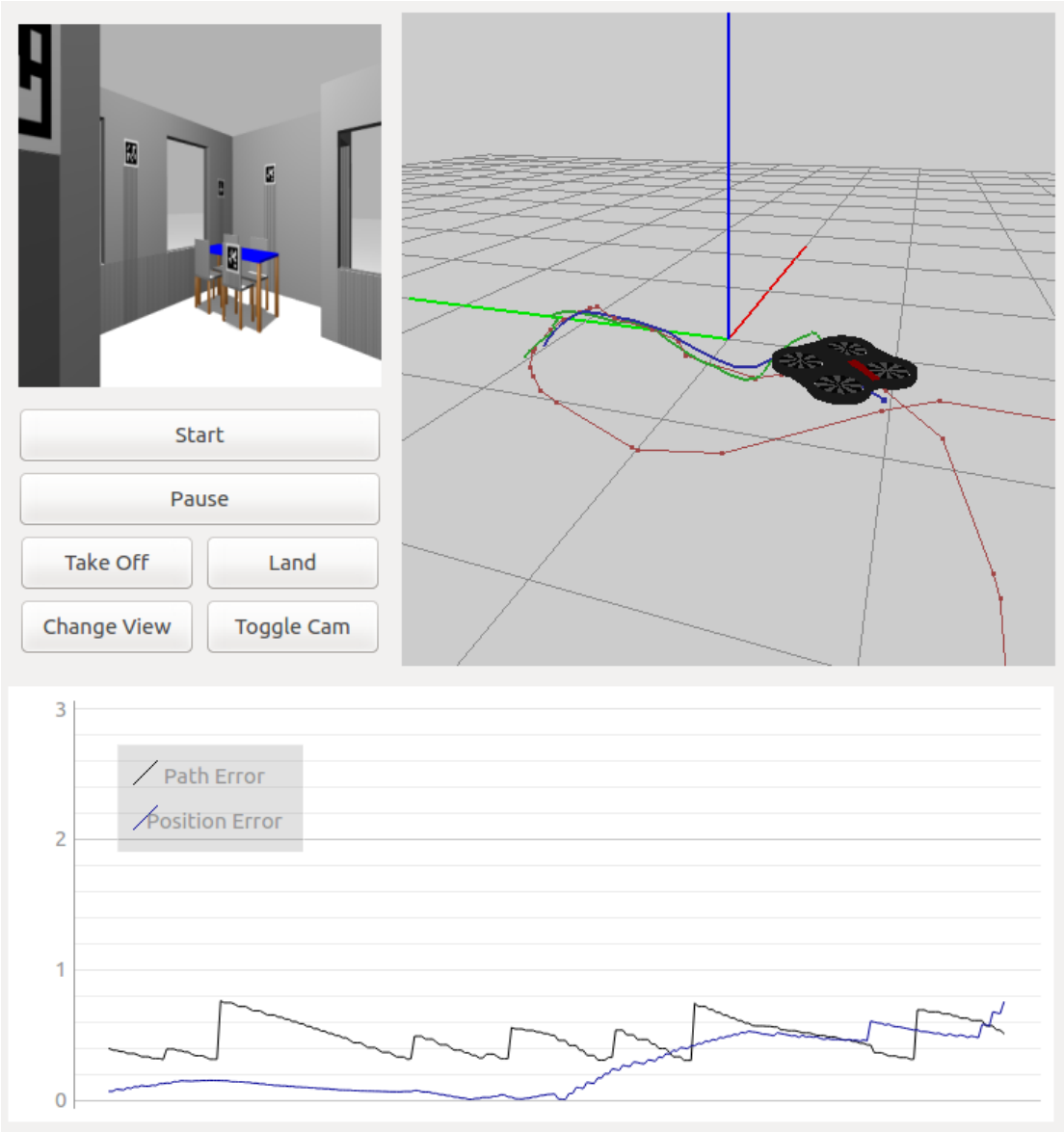


Figura 5.11: Comportamiento del sistema completo en ruta compleja (Parte 2)

En la Figura 5.11 se observa un comportamiento similar a la Figura 5.10. En la gráfica se muestra un repunte en el error de estimación de posición debido a la diferencia en las distancias de las balizas, caso ilustrado en la Figura 5.7. Al salir del campo de visión la baliza más cercana, cuya estimación tenía un peso mucho mayor que las del resto, la estimación de posición se desestabiliza momentáneamente. Sin embargo, al tener un

sistema de control de reacción lenta este cambio abrupto apenas afecta al sistema de control. Esto se puede observar en el punto de posición verdadera, que permanece ajustado a la ruta.

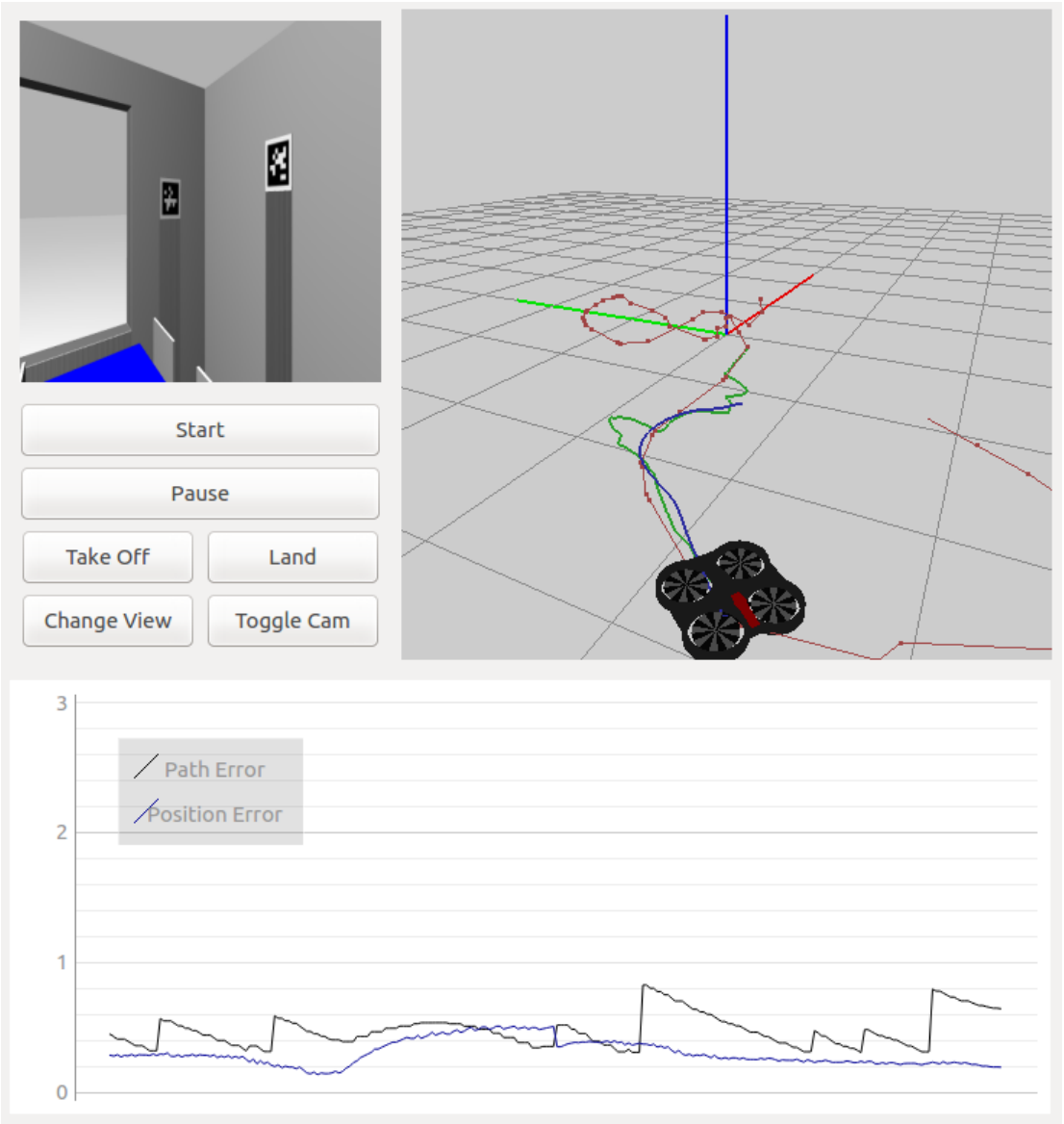


Figura 5.12: Comportamiento del sistema completo en ruta compleja (Parte 3)

De nuevo, en la Figura 5.12 podemos observar un comportamiento similar en el que pequeñas variaciones en la estimación de posición no afectan al seguimiento de la ruta.

5.2 Experimentos en entorno real

El primer paso para la exprimentación con un cuadricóptero real fue la familiarización con la plataforma. Para ello se hizo uso del componente *Uav_viewer* con el que se teleoperaba al drone a través de la herramienta *ardrone_server*, que añade la capa de comunicaciones entre el software de *ArDrone2* y ICE. En estas primeras pruebas se observó que el cuadricóptero tenía deriva, un fenómeno que no se había tenido en cuenta en el caso de la simulación. También se observó que si se enviaban comandos de velocidad fijos, el cuadricóptero tendía a acelerar alcanzando velocidades mayores a las deseadas. Además, fue necesario calibrar la cámara del cuadricóptero (Figura 5.13).

Figura 5.13: Proceso de calibración del drone real

Los primeros experimentos se realizaron de forma unitaria sobre el componente de autolocalización *Cam_autoloc*. En primer lugar, se experimentó sobre una única baliza y se fueron añadiendo balizas progresivamente, además de aumentar la distancia a éstas. Los resultados fueron similares a los obtenidos en el entorno simulado, sin embargo, la distancia a la que el sistema perdía precisión era menor.

Inicialmente se utilizaron balizas impresas en un papel A4, teniendo éstas un tamaño de 16cm. Este tamaño ofrecía una buena estimación hasta 2m. con una cámara estática, sin embargo, cuando el cuadricóptero estaba en movimiento la estimación contenía muchos errores. A continuación se utilizaron balizas impresas en un papel A3, con un tamaño de baliza de 24cm. Aunque los resultados que se obtuvieron mejoraban notablemente con respecto al caso anterior, la precisión era aún insuficiente. El movimiento del drone causaba que la imagen captada fuera borrosa y dificultaba la tarea de detección de los marcadores.

Finalmente, se optó por un tamaño de papel A2, siendo el tamaño de las balizas de 33cm. Los resultados mejoraban de manera sustancial con respecto a los tamaños anteriores, aunque seguía habiendo errores en la estimación cuando el drone estaba en movimiento. Este hecho se puede observar en la Figura 5.14, en la que el drone se encuentra a 4m. de distancia de las balizas. Aunque la estimación obtenida a partir de la fusión espacial es correcta, la diferencia entre las posiciones calculadas es alta. Esto puede provoca inestabilidad en el sistema cuando las balizas entran y salen del campo de visión del cuadricóptero, como se comprobó en los experimentos en simulación.

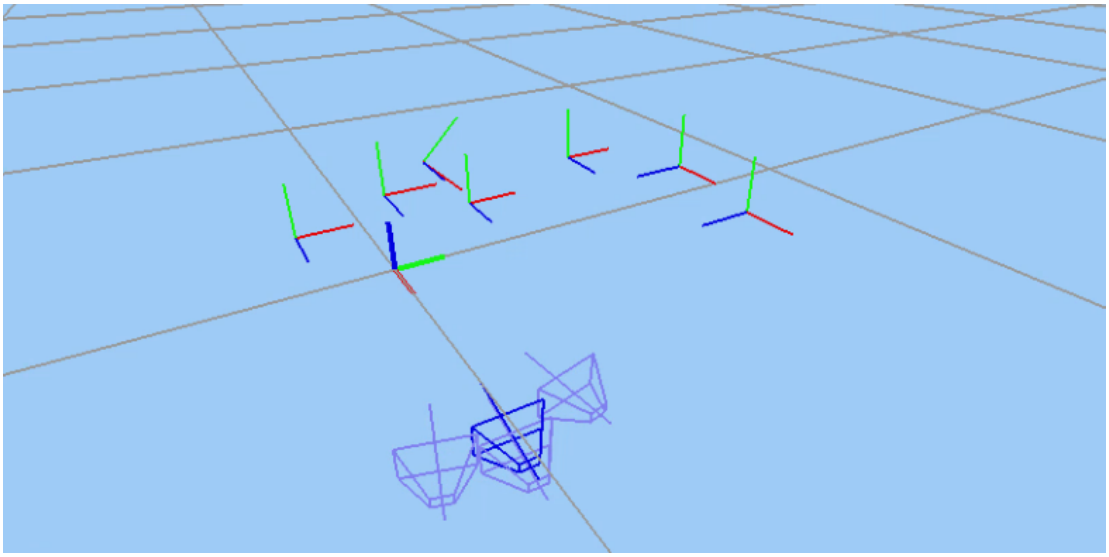


Figura 5.14: Autolocalización en drone real a 4 metros

Los primeros experimentos con el sistema completo se realizaron sobre una ruta lineal. No obstante, siempre era necesario que el cuadricóptero realizara giros para seguir el primer punto de la ruta ya que la maniobra de despegue provocaba que el vehículo se desplazaba de 1 a 3 metros hasta estabilizarse. Estos experimentos mostraron resultados muy inestables, ya que estos giros provocaban que la estimación de la posición sufriera cambios muy bruscos.

Para solucionar estos problemas hubo que redefinir varios parámetros del componente de autolocalización. En primer lugar se modificaron los valores de los pesos de la fusión espacial, disminuyendo los valores asignados a las estimaciones con distancias superiores a 3 metros. También se redefinieron los valores de las matrices de covarianza de error del *Filtro de Kalman* en la parte de fusión temporal. De esta forma se estableció un modelo para medidas muy ruidosas que reduce el impacto de las estimaciones reales en beneficio de una mayor estabilidad. Aún así, los experimentos seguían mostrando un comportamiento inestable del sistema, como se puede observar en la Figura 5.15. La captura fue tomada en un momento en el que el drone realizaba un giro mientras mantenía la posición a 3 metros de las balizas, sin embargo, el componente de autolocalización mostraba variaciones en la posición de un par de metros. Esto se debe a dos factores: la dificultad en detectar balizas en movimiento, y la salida y entrada del campo de vision de estas balizas.

Se realizaron diversos experimentos disminuyendo progresivamente la velocidad del cuadricóptero con el objetivo de encontrar un rango de velocidades a las que la detección de balizas fuera robusta. Durante los experimentos a velocidades más bajas se observó que el vehículo no respondía adecuadamente a los comandos de velocidad enviados. Tras varias pruebas con el componente *Uav_viewer* se observó que el propio sistema de estabilización integrado por el fabricante interfería con los comandos enviados. Este hecho se muestra en la Figura 5.16, donde el drone intenta acercarse al primer punto de la ruta. Al realizar el giro a la derecha, se produce un desplazamiento lateral hacia su izquierda que provoca que el sistema de control introduzca una mayor velocidad de giro, desestabilizando el sistema.

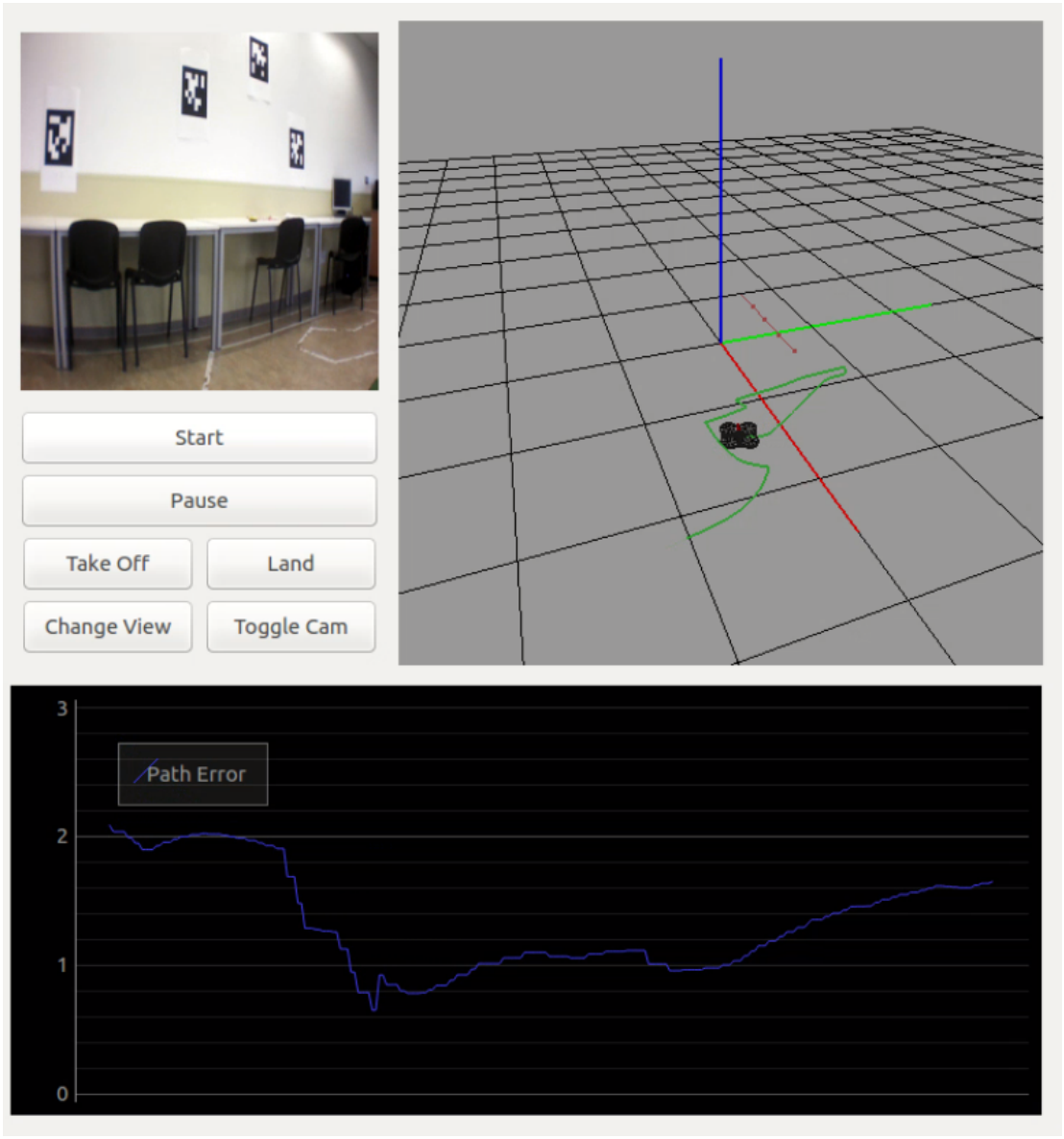


Figura 5.15: Navegación en entorno real 1

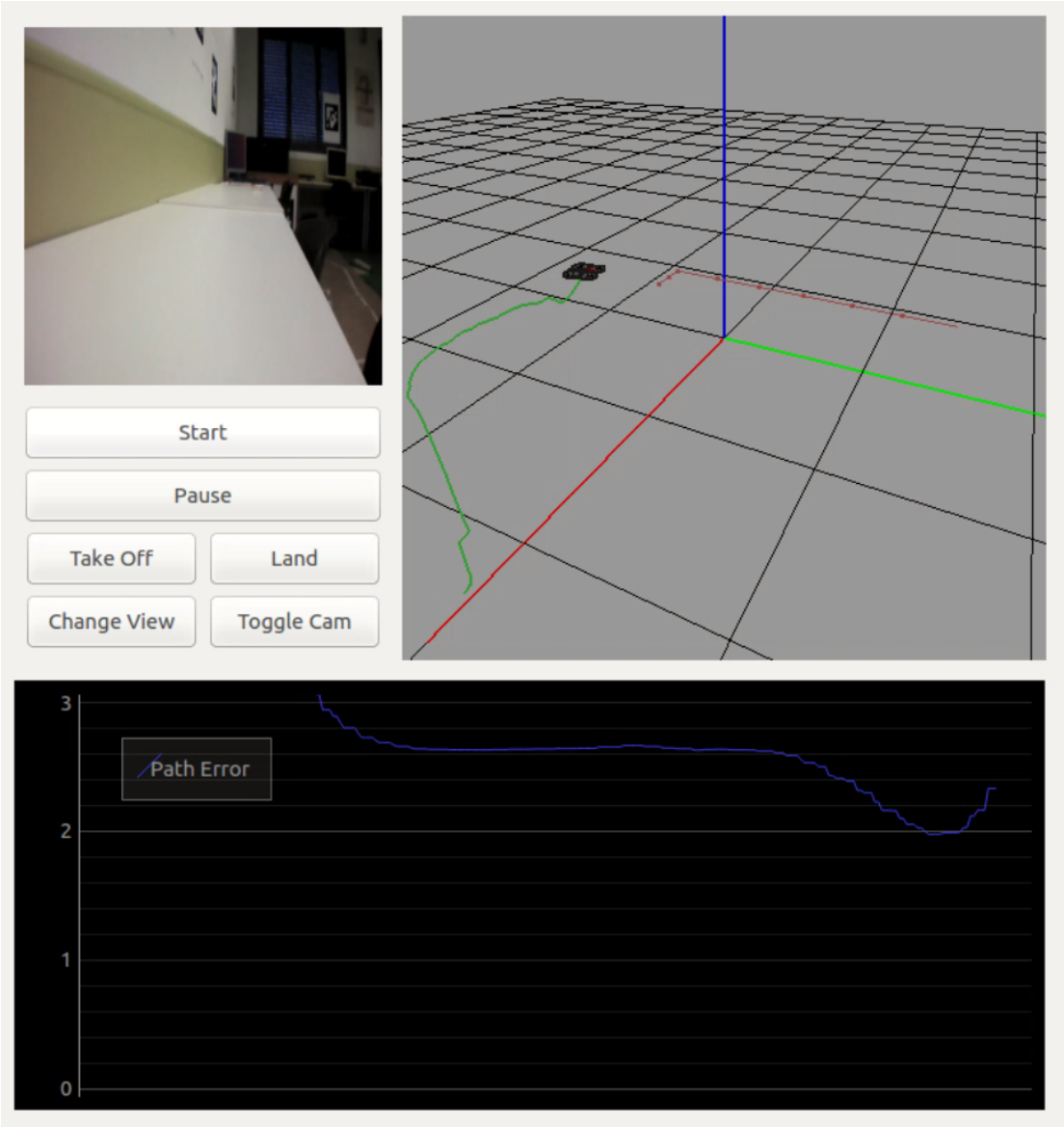


Figura 5.16: Navegación en entorno real 2

En la Figura 5.17 se muestra un experimento en el que la velocidad establecida era de 0.05 m/s con el propósito de obtener una autolocalización lo más precisa posible. Como se puede observar al comienzo de la navegación hay unos pequeños saltos en la estimación que apenas afectan al control de posición. El drone alcanza de forma exitosa los primeros puntos de la ruta, aunque cuando realiza la maniobra de giro el sistema se destabiliza

debido a que las balizas cercanas salen del campo de visión y pasa a ver otras balizas más lejanas.

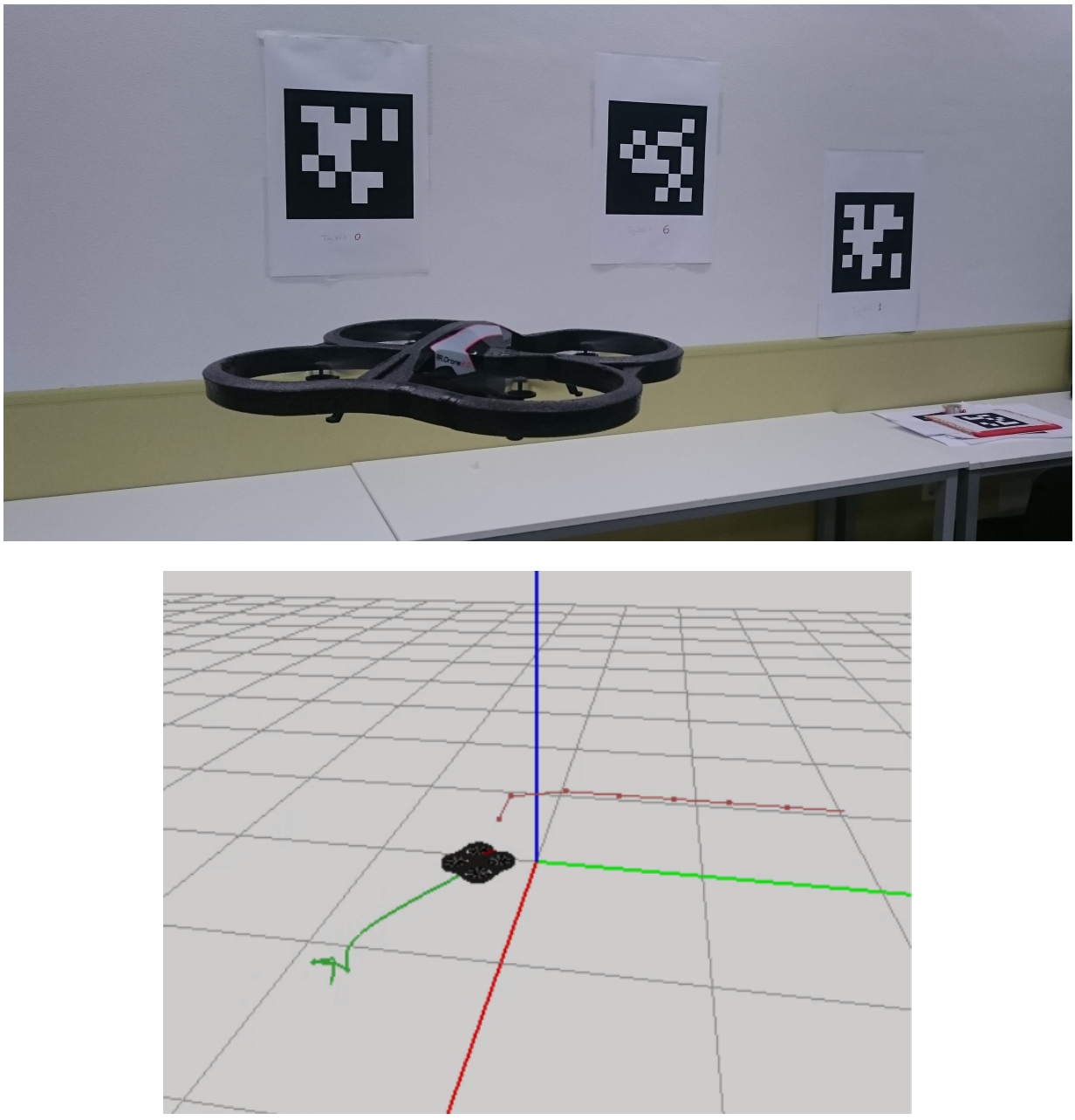


Figura 5.17: Navegación en entorno real 3

Capítulo 6

Conclusiones

A lo largo de esta memoria se ha explicado el desarrollo y validación de un sistema para vuelo autónomo de drones en espacios interiores. Para conseguir estos objetivos, hemos hecho uso de un gran número de herramientas y tecnologías que han tenido que ser comprendidas, domadas y, en muchos casos, adaptadas a nuestros propósitos. Se ha conseguido validar el funcionamiento del sistema desarrollado en simulación, obteniendo un sistema estable. Aunque el ruido de las medidas y el comportamiento de los soportes físicos pueden afectar el comportamiento del sistema, este ha probado ser lo suficientemente robusto para satisfacer las metas propuestas. Adicionalmente, se han realizado numerosos experimentos en plataformas y entornos reales, añadiendo nuevas perspectivas sobre el sistema programado.

A continuación se realiza una comparación con las metas inciales planteadas y los hitos conseguidos, y finalmente se propondrán futuras líneas de trabajo.

6.1 Conclusiones

La solución desarollada en este trabajo satisface los objetivos propuestos. Además, se ha llegado más allá de los hitos marcados llevando el sistema a un entorno real. Si

volvemos al *Capítulo 2* podemos hacer una análisis acerca de los objetivos conseguidos y los requerimientos satisfechos:

1. **Desarrollo de componente de navegación.** Se ha conseguido desarrollar un sistema de navegación autónomo controlado por posición para un cuadricóptero con un funcionamiento fiable. En el proceso se han sopesado distintos métodos para implementar el sistema y finalmente se ha implementado una solución original. Para encontrar esta solución se han tenido en cuenta las carácteristicas del medio en el que se desarrolla el vuelo del cuadricóptero, así como las carácteristicas del movimiento del vehículo. La solución se basa en el control de giro del drone para acercarlo a la trayectoria deseada mediante la predicción de su posición.

   Además, en el desarrollo del componente se refactorizaron las herramientas *Recorder* y *Replayer* para dotarlas de las funcionalidades necesarias para compatibilizarlas con el tipo de datos *Pose3D* y que formaran parte del repositorio oficial de *JdeRobot*. Durante este proceso se aprendió con profundidad el funcionamiento de la arquitectura de *JdeRobot*, además de otras herramientas como *GitHub* y *QtCreator*.

2. **Refactorización e integración del componente de autolocalización.** Esta herramienta tuvo que ser estudiada en profundidad hasta comprenderla completamente. También se adquirieron conocimientos acerca los métodos y algoritmos de visión artificial. De esta forma se ha podido modificar y adaptar a las necesidades del problema presentado. Además, en el proceso de integración, se aprendió a utilizar herramientas de compilación como *qmake* y *cmake*.

   Durante la fase de integración del componente fue necesaria una refactorización de todo el sistema, debido a la publicación de la nueva versión de *JdeRobot* y todas las actualizaciones en las librerías que conlleva. Esta refactorización implicó la aparición de nuevos errores que se solventaron con el examen minucioso del código de ambos componentes y las modificaciones pertinentes.

3. **Validación experimental en simulación.** Se diseñaron diversos experimentos mediante los cuales se pudo validar la estabilidad del sistema y sus limitaciones.

Durante este proceso también se realizaron las modificaciones pertinentes al sistema para mejorar su funcionamiento y aumentar la robustez que ofrece. Finalmente el sistema implementado ha mostrado satisfacer los requisitos iniciales, teniendo la suficiente robustez como para presentar un comportamiento fiable.

Además de cumplir todos los objetivos propuestos, se ha llevado el sistema a una plataforma real. Esto ha permitido ir más allá de los objetivos iniciales y ha arrojado nuevas perspectivas acerca del sistema desarrollado. Para ello se ha hecho uso de los recursos disponibles en el *Grupo de Robótica de la URJC*.

Durante el proceso de experimentación en entorno real se han aprendido numerosos conceptos acerca del funcionamiento real de los cuadricópteros. Esto ha permitido tener en cuenta factores que en simulación no afectaban al comportamiento del sistema. Los principales factores han sido la deriva propia del vehículo, que añade un mayor grado de complejidad al sistema de control; los sistemas de estabilización establecidos por el fabricante, que en ocasiones entran en conflicto con el control proporcionado por el componente; y el ruido de los sensores.

El estudio del comportamiento real del sistema ha facilitado establecer unas pautas mediante las cuales se pueden diseñar mejoras en el sistema. De esta forma las bases quedan sentadas para las futuras versiones del componente de navegación.

6.2 Trabajos futuros

El prototipo desarrollado satisface las necesidades propuestas, y aunque su funcionamiento en plataformas reales no muestra la robustez esperada, sirve de base para seguir mejorando. El sistema ha establecido los cimientos sobre los que seguir añadiendo mejoras para poder así desarrollar un nuevo prototipo que funcione en casos reales de forma fiable. A continuación se detallan distintas líneas de mejora que se podrían aplicar para obtener un sistema totalmente robusto, con nuevas funcionalidades y aplicaciones.

En primer lugar, el componente de navegación se ha desarrollado conforme a las caracterísiticas del medio, con una gama de movimientos limitados a favor de la estabilidad del sistema. Sin embargo, futuras líneas de desarrollo podrían implicar la mejora del sistema de control introduciendo nuevos tipos de movimiento como un movimiento transversal puro o el aumento de la velocidad de vuelo.

Otra de las líneas de mejora del sistema podría ser inclusión de nuevas funcionalidades en el software de control como, por ejemplo, la captura de fotografías de puntos concretos del entorno o la inserción de bifurcaciones en la ruta de navegación. Estas funcionalidades serían de gran utilidad en tareas de vigilancia y monitorización de espacios interiores.

Adicionalmente, se puede añadir un sistema propio de estabilización del vehículo. Como se ha observado en los experimentos con el cuadricóptero real, el sistema de estabilización integrado por el fabricante inteferiere con los controles en el componente de navegación. Un sistema propio, integrado junto al sistema de control existente, podría mejorar significativamente el rendimiento del control de navegación.

El sistema de autolocalización ha demostrado ser bastante robusto en simulación, sin embargo, los experimentos en entorno real han probado que el sistema de visión artificial es insuficiente. Debido al movimiento dinámico y en ocasiones brusco del cuadricóptero, se dificulta la tarea de detección de balizas. Esto podría solventarse añadiendo otra técnica de apoyo como la odometría visual o *Visual SLAM*. Mediante técnicas de odometría visual se solventarían los problemas originados al no detectar balizas, ya que esos intervalos de tiempo serían los suficientemente pequeños para que esta técnica estimara la posición sin acumular muchos errores. Añadir *VisualSLAM* sería una gran mejora en el sistema, en conjunto con el sistema fiducial implementado se obtendrían resultados de gran estabilidad, a la vez que se daría espacio a un mayor número de aplicaciones gracias al abanico de posibilidades que esta técnica ofrece.

Bibliografía

[1] Cañas Plaza, J.M. Jerarquía dinámica de esquemas para la egneración de comportamiento autónomo. *Tesis Doctoral, Universidad Politécnica de Madrid*, 2003.

[2] G. Roberts, L. Machine Perception of three-dimensional solids. *Thesis (Ph. D.),Massachusetts Institute of Technology. Dept. of Electrical Engineering*, 1963.

[3] Apvrille, L., Dugelay, J.L., Ranft, B. Indoor autonomous navigation of low-cost mavs using landmarks and 3d perception. *Proc. Ocean and Coastal Observation, Sensors and Observing Systems*, 2013.

[4] Nikolic, J., Leutenegger, S., Burri, M.,Huerzeler, C., Rehder, J., Siegwart, R. A UAV System for Inspection of Industrial Facilities. *IEEE Aerospace Conference*, 2013, 10.1190/AERO.2013.6494959

[5] Martín Martínez, S. Mapas 3D de Parches Planos en Entornos Reales utilizando Sensores RBG-D. *Trabajo Fin de Grado, Universidad Rey Juan Carlos*, 2016.

[6] López-Cerón, A. Autolocalización visual robusta basada en marcadores. *Trabajo Fin de Máster, Universidad Rey Juan Carlos*, 2015.

[7] Yagüe Sánchez, D. Cuadricóptero AR.Drone en Gazebo y JdeRobot. *Proyecto Fin de Carrera, Universidad Rey Juan Carlos*, 2015.

[8] Martín Florido, A. Visual navigation on a quadricopter for object tracking. *Proyecto Fin de Carrera, Universidad Rey Juan Carlos*, 2014.

[9] Wu, A., Johnson, E.N., Kaess, M., Dellaert, F., Chowdhary, G. Autonomous Flight in GPS-Denied Environments Using Monocular Vision and Inertial Sensors. *J. Aerospace Inf. Sys.*, 2013 Apr 1,10(4):182-86.

[10] Olson, E. A robust and flexible visual fiducial system. *IEEE International Conference on Robotics and Automation (ICRA)*, 3400-2407, 2011.

[11] Kalman, R.E. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 1960.