



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE
TELECOMUNICACIÓN

GRADO EN INGENIERÍA TELEMÁTICA

TRABAJO FIN DE GRADO

**Navegación autónoma de rutas 3D en un
entorno real**

Autor: Andres de Jesús Hernández Escobar

Tutor: José María Cañas Plaza

Curso académico 2017/2018

Agradecimientos

En primer lugar, quiero agradecer a mi tutor Jose María por su motivación, paciencia y la confianza que ha depositado en mí.

A mi familia, por apoyarme no sólo durante este proyecto sino a lo largo de mi trayectoria hasta llegar aquí.

Y a Layla, por los buenos momentos, comprensión y apoyo durante la carrera y el desarrollo de este proyecto que parecía no tener fin.

Resumen

La robótica y los robots cada vez más están encaminados hacia resolver tareas autónomamente y a un menor tiempo de supervisión humana. La consolidación de esta autonomía acerca un futuro cada vez más próximo en el que las máquinas ganarán independencia y podrán realizar múltiples tareas por su cuenta, ganando en flexibilidad y aprovechamiento de los recursos. Este Trabajo de Fin de Grado se enfoca en dotar de navegación autónoma a un drone real.

Con el fin de demostrar esta autonomía, el cuadricóptero deberá despegar, navegar en un entorno de 3D siguiendo una ruta y aterrizar, todo esto sin asistencia por parte de un teleoperador humano. Hemos programado el drone dividiendo esta tarea en cuatro fases más sencillas que son: el despegue controlado, búsqueda de balizas visuales, la navegación a partir de la autolocalización relativa y por último el aterrizaje controlado. Se utilizan dos tipos diferentes de balizas visuales, que actúan a modo de referencia o identificador único, unas para aterrizar o despegar y otras para navegar. Se ha integrado todo en una aplicación basada en un autómata de estados finito. Este autómata se ha diseñado empleando la herramienta VISUAL STATES. El componente final desarrollado, llamado *3DPathFollower*, se ha escrito en el lenguaje de programación Python, en la versión de JdeRobot 5.6.4.

Se ha validado experimentalmente la aplicación desarrollada y la infraestructura en un drone real. Para ello, se han realizado tanto experimentos unitarios como globales. Se han comprobado dos configuraciones. Una con todo el procesado todo el procesado de manera externa y otra completamente a bordo del drone en un ordenador embarcado.

Índice general

| | |
|--|------------|
| Índice de figuras | VII |
| 1. Introducción | 1 |
| 1.1. Robótica | 1 |
| 1.1.1. Aplicaciones actuales | 2 |
| 1.2. Software en robots | 6 |
| 1.2.1. Simulación | 6 |
| 1.3. Visión artificial en robots | 8 |
| 1.4. Robótica Aérea | 9 |
| 1.4.1. Aplicaciones actuales | 11 |
| 2. Objetivos | 19 |
| 2.1. Problema a abordar | 19 |
| 2.2. Requisitos | 20 |
| 2.3. Metodología | 21 |
| 2.4. Planificación | 22 |
| 3. Infraestructura | 24 |
| 3.1. Parrot Ar.Drone 2 | 24 |
| 3.2. Intel Compute Stick | 25 |
| 3.3. Biblioteca OpenCV | 26 |
| 3.4. Biblioteca AprilTags | 26 |
| 3.5. Entorno JdeRobot | 28 |

| | |
|---|-----------|
| ÍNDICE GENERAL | IV |
| 3.5.1. Ardrone_Server | 28 |
| 3.5.2. Teleoperadores uav_viewer y uav_viewer.py | 29 |
| 3.5.3. Herramienta Color Tuner | 29 |
| 3.5.4. Componente slam_markers | 30 |
| 3.5.5. VisualStates | 31 |
| 3.6. Bibliotecas ICE | 33 |
| 3.7. Simulador Gazebo | 33 |
| 4. Navegación autónoma de un drone guiado por balizas visuales | 35 |
| 4.1. Diseño | 35 |
| 4.2. Detección visual de las balizas | 37 |
| 4.2.1. Balizas de aterrizaje y despegue | 38 |
| 4.2.2. Balizas de autolocalización | 39 |
| 4.3. Autómata de navegación | 40 |
| 4.3.1. Estado de despegue controlado | 42 |
| 4.3.2. Estados de navegación guiada por balizas | 43 |
| 4.3.3. Estados de aterrizaje | 44 |
| 4.4. Configuración | 45 |
| 4.4.1. Configuración de la percepción | 45 |
| 4.4.2. Configuración de interfaces | 46 |
| 4.4.3. Configuración de la secuencia de balizas | 47 |
| 4.5. Herramienta CalibrationTool | 47 |
| 5. Experimentos | 50 |
| 5.1. Pruebas en Simulador | 51 |
| 5.1.1. Elección de balizas de despegue y aterrizaje | 51 |
| 5.1.2. Ajuste del control de navegación | 52 |
| 5.1.3. Ejecución típica integral | 53 |
| 5.2. Pruebas con el drone real y PC externo | 54 |
| 5.2.1. Ajuste del control de navegación | 54 |

| | |
|---|-----------|
| ÍNDICE GENERAL | V |
| 5.2.2. Desfase entre imágenes y órdenes | 54 |
| 5.2.3. Problemas con la detección de las balizas arlequinadas | 55 |
| 5.2.4. Ejecución típica integral | 56 |
| 5.3. Pruebas con el dron real y un miniordenador | 61 |
| 5.3.1. Preparación y configuración del miniordenador | 61 |
| 5.3.2. Teleoperación con miniordenador a bordo | 63 |
| 5.3.3. Retardo detectado en uav_viewer.py | 63 |
| 5.3.4. Ejecución típica con el miniordenador en tierra | 63 |
| 6. Conclusiones | 65 |
| 6.1. Conclusiones | 65 |
| 6.2. Trabajos futuros | 67 |
| Bibliografía | 69 |

Índice de figuras

| | |
|--|----|
| 1.1. Ejemplos de robots | 3 |
| 1.2. Ventas anuales estimadas de robots industriales por regiones | 4 |
| 1.3. Aplicaciones en la actualidad | 6 |
| 1.4. Personas simuladas en Gazebo | 7 |
| 1.5. Visión artificial en robots | 10 |
| 1.6. UAV Predator. | 10 |
| 1.7. Ejemplos de UAV civiles | 12 |
| 1.8. Relación entre la potencia de los rotores y el movimiento de un cuadricóptero | 13 |
| 1.9. Ejemplo de interfaz de usuario del componentes Uav_viewer | 15 |
| 1.10. Ejemplos de ArDrone simulado en Gazebo | 16 |
| 1.11. Ejemplo de algoritmo de navegación basado en autolocalización en Gazebo. | 16 |
| 1.12. Ejemplos de aterrizaje y despegue en Gazebo y el dron real. | 17 |
| 1.13. Ejemplo de navegación a través del seguimiento de rutas. | 17 |
| 2.1. Representación del desarrollo en espiral. | 21 |
| 3.1. Imagen de Intel compute Stick | 25 |
| 3.2. Ejemplo de detección en AprilTag | 27 |
| 3.3. Ejemplos de familias en AprilTag | 28 |
| 3.4. Estructura de ardrone_server | 29 |
| 3.5. Ejemplo de Color Tuner aplicando filtros de color | 30 |

| | |
|--|----|
| 3.6. Ejemplo de slam_markers identificando una baliza | 31 |
| 3.7. Ejemplo de autómata generado con VisualStates | 32 |
| 3.8. Ejemplo de simulador Gazebo | 34 |
| 4.1. Entradas y salidas de la solución final. | 36 |
| 4.2. Diagrama de bloques del componente 3DPathFollower. | 37 |
| 4.3. Ejemplo de baliza de color arlequinada. | 38 |
| 4.4. Estados de la aplicación 3DPathFollower | 41 |
| 4.5. Fichero de configuración de interfaces ICE. | 47 |
| 4.6. Ejemplo del componente CalibrationTool. | 49 |
| 5.1. Escenario utilizado para las pruebas unitarias en Gazebo. | 51 |
| 5.2. Comparación de balizas visuales en Gazebo. | 52 |
| 5.3. Drone simulado navegando en Gazebo. | 53 |
| 5.4. Secuencia de despegue. | 56 |
| 5.5. Balizas desplegadas en el laboratorio. | 58 |
| 5.6. Secuencia de navegacion. | 60 |
| 5.7. Intel Compute Stick acoplado en Ar.Drone 2. | 62 |

Capítulo 1

Introducción

Este Trabajo de Fin de Grado (TFG) se encuadra en la programación de un robot aéreo para que navegue autónomamente utilizando visión. En este primer capítulo se introducirá brevemente al lector en el mundo de la robótica y más concretamente en los robots aéreos, su estado actual, su evolución y el impacto que está teniendo este sector en la sociedad. Adicionalmente, se contextualizarán las técnicas de visión en robots relacionadas con este TFG.

1.1. Robótica

La robótica es la disciplina involucrada en el diseño, la fabricación y la aplicación de robots. Un robot es una máquina que puede programarse para que interactúe con objetos y lograr un objetivo, como imitar el comportamiento humano o la sustitución de una persona en un entorno peligroso. Típicamente los robots tienen una parte hardware y una parte software. En el hardware están compuestos de sensores, actuadores y procesadores.

Un sensor es un dispositivo eléctrico y/o mecánico que convierte magnitudes físicas (luz, electricidad, presión, etcétera) en valores medibles de dicha magnitud. Dan información del entorno o del propio robot y son equivalentes a los sentidos del cuerpo humano, como la vista o el oído. Por ejemplo, con sensores de temperatura se puede

medir el número de grados Celsius en una habitación.

Un actuador es un dispositivo capaz de transformar energía hidráulica, neumática o eléctrica en energía mecánica que permite al robot hacer algo o desplazarse por su entorno. Un ejemplo es un motor eléctrico que transforma electricidad en un movimiento rotacional para girar una rueda. El actuador se correspondería a los músculos y articulaciones que componen un cuerpo humano.

Por último, un robot está formado por computadores, que obtienen datos de los sensores, los procesan y se encargan de materializar acciones en los actuadores. Volviendo a la analogía con el ser humano, sería nuestro cerebro y nervios.

Los robots pueden ser o no autónomos. Por autonomía se entiende la habilidad para tomar decisiones por uno mismo y llevarlas a cabo. Esto en un robot es la capacidad de percibir la situación y actuar apropiadamente sin intervención humana directa.

En caso de carecer de autonomía, se puede interaccionar con el robot mediante la teleoperación, que es la manipulación y envío de órdenes para ser ejecutadas por un robot que se encuentra en un lugar diferente a la persona. Por ejemplo en medicina se utiliza para realizar operaciones a través de unos brazos que ejecutan los movimientos enviados desde un lugar lejano. En el espacio exterior se aplica a la hora de enviar órdenes a satélites o robots como el Curiosity en Marte.

En robótica típicamente el comportamiento ha de ser en tiempo real incluyendo la toma de decisiones y el análisis de diferentes situaciones y además, debe ser robusto para evitar posibles accidentes o resultados no esperados.

Uno de los objetivos para el futuro de la robótica es la multitarea. Hoy en día un robot está diseñado para un número limitado de posibles trabajos o tareas, a diferencia de los seres humanos, los cuales nos adaptamos sin necesidad de cambiar nuestra naturaleza física.

1.1.1. Aplicaciones actuales

Hoy en día, las aplicaciones de los robots son muy diversas. Fuera y dentro de nuestro planeta, los robots permiten ver sitios en los que el hombre no puede llegar



(a) *Curiosity* en Marte



(b) Teleoperación médica

Figura 1.1: Ejemplos de robots

directamente o en los que el hábitat es hostil como el Global Explorer ROV, que se ha sumergido en diferentes océanos para obtener imágenes nunca antes vistas por el hombre.

Sector industrial

Es uno de los sectores que compra más robots y se encuentra en constante crecimiento. China pasó en 2013 a los E.E.U.U. en densidad de robots por trabajador y el número de ventas de robots industriales en el mundo aumentó un 16 % en 2016 por cuarto año consecutivo (Figura 1.2¹).

Dentro de las aplicaciones industriales de robots encontramos:

- Operaciones de manipulación: Usan pinzas, colaboran con otros robots y/u operarios y desplazan objetos. Este uso es uno de los más extendidos en empresas. Por ejemplo en el montaje de coches, ayudando a operarios a desplazar objetos pesados como las puertas.
- Soldadores. Se encargan de las tareas de soldadura de componentes. La compañía Asus ha creado un método de producción automática para sus tarjetas gráficas. En concreto, este proceso de soldadura mejora la calidad del producto y permite

¹Datos obtenidos de International Federation of Robotics <http://www.ifr.org/>

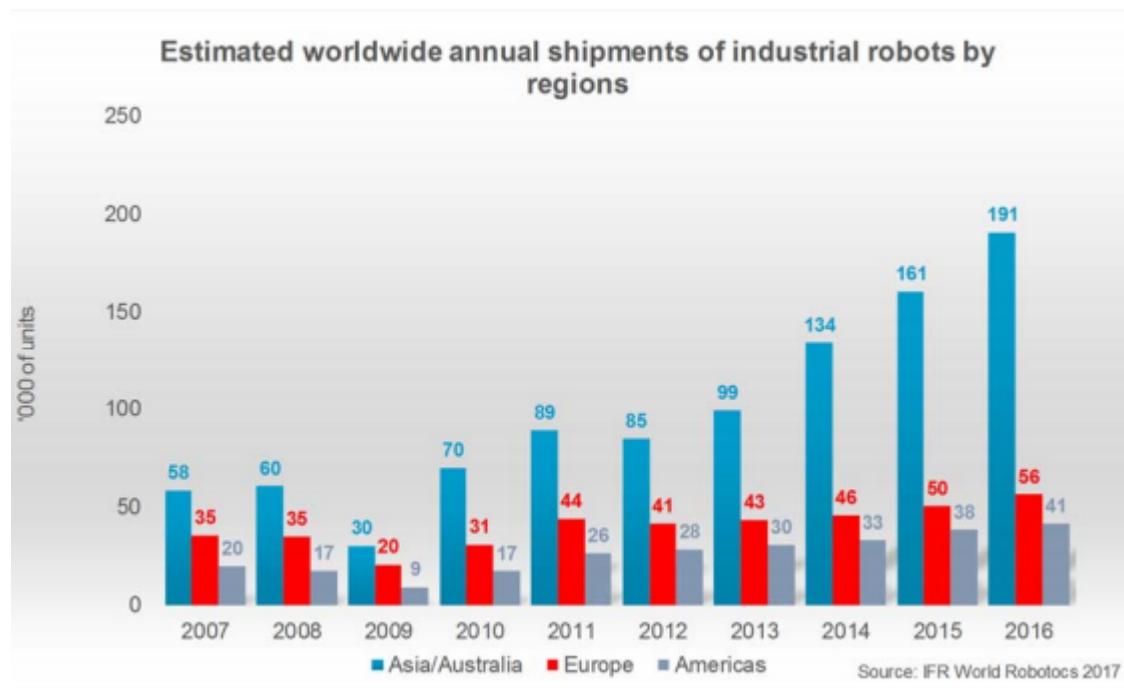


Figura 1.2: Ventas anuales estimadas de robots industriales por regiones

reducir el tamaño de sus tarjetas notablemente.²

- Montaje. Las cadenas de montaje se vuelven más rápidas y eficientes. En las plantas de procesadores Intel, el proceso de montaje es uno de los más avanzados del mundo y se utilizan salas en las que el aire no es respirable para personas y garantizan una densidad de partículas externas muy baja, muy importante para la pureza de los procesadores.

Aplicaciones de servicios

El acercamiento de los robots a la población ha supuesto que su uso se encuentre en constante crecimiento y el número de aplicaciones es muy variado, desde recreación pasando por la grabación profesional para cine. A continuación se recogen algunas de las aplicaciones ilustrativas:

²ASUS Auto-Extreme Technology: <https://www.youtube.com/watch?v=zVDEcu6-G3s>

- Limpieza doméstica: Incluye robots que limpian piscinas, hasta aspiradoras inteligentes. Este último caso es el de Roomba, de la compañía iRobot, que incorpora algoritmos de construcción de mapas, evasión de objetos o incluso detección de escaleras (para evitar posibles accidentes).
- Transporte de personas: Los mayores representantes de este tipo de productos son Waymo, Tesla y Uber (Figura 1.3). En esta categoría se recogen sistemas de seguridad que controlan la distancia de seguridad respecto de otros vehículos, los sistemas de aparcamiento asistido o habilitar a personas con movilidad reducida o discapacitados para que puedan usar los automóviles. Para ejecutar estas tareas los vehículos están equipados con todo tipo de sensores que permiten la autolocalización, evitar accidentes y llegar al destino deseado. Algunas compañías como Uber y Tesla han sufrido recientemente accidentes mortales que pueden retrasar esta aplicación en el futuro.
- Ocio y entretenimiento: La utilización de drones con la capacidad de volar ha reducido considerablemente el coste de planos aéreos y simplificado el equipo necesario. Airdog (Figura 1.3) es un proyecto nacido de una campaña Kickstarter que permite el seguimiento de actividades deportivas o recreativas a gran velocidad, de forma totalmente autónoma, y la grabación de las mismas.
- Educación: La aplicación de robots para el uso didáctico se contempla como un recurso innovador, que aumenta el interés de los niños y sirve de apoyo para los maestros (Figura 1.3).
- Militar y seguridad: Para evitar la pérdida de bajas humanas, aumentar la capacidad de motorización y mejorar su potencia de combate. Los ejércitos están invirtiendo cada vez más en robots capaces de sustituir a soldados en el frente y como armas de defensa.



Figura 1.3: Aplicaciones en la actualidad

1.2. Software en robots

El software que se encuentra en los robots es el encargado de dotar de un comportamiento inteligente a los mismos.

Puede estar formado por sistemas complejos, aplicaciones e infraestructuras que son las responsables de las acciones del robot. Las fases del desarrollo del software robótico son muy similares a las del desarrollo de software en otros ámbitos, donde a partir de ciertos requisitos, se modela un diseño que será implementado. Durante años este desarrollo se ha centrado en resolver los problemas con soluciones ‘ad-hoc’, es decir, creando un diseño para un robot con sensores y actuadores específicos. Esto requería que se realizase una implementación nueva por cada robot diferente, a pesar de que las características fueran similares. En la actualidad, gracias a la existencia de diferentes plataformas de desarrollo para robots es posible diseñar e implementar soluciones que puedan aplicarse de forma eficiente y genérica. Esto permite reutilizar herramientas, aplicaciones y algoritmos creados con antelación y reducir los costes durante la fase de desarrollo del software.

1.2.1. Simulación

Una parte muy importante a la hora de diseñar el comportamiento de un robot es la simulación ya que permite probar algoritmos sin necesidad de utilizar uno real. Esto nos

aporta información muy valiosa y facilita la familiarización con posibles situaciones que no hayamos pensado con suficiente antelación, además de prevenir accidentes como por ejemplo cualquier daño físico al robot o herir a las personas cercanas. Una vez se alcance un comportamiento que cumpla nuestras expectativas, se procederá a ponerlo a prueba en el robot real teniendo en cuenta que muy probablemente aparecerán anomalías en el comportamiento no detectadas durante la fase de simulación. Esta diferencia en el comportamiento depende de la precisión con la que se ha caracterizado el modelo del robot y el escenario virtual con el que interacciona. Esto se debe a siempre existir un grado de aproximación entre la virtualización y el mundo real. Muchos simuladores incluyen la adición de funciones de ruido en sensores y actuadores para alcanzar un comportamiento mucho más próximo al real.

Un ejemplo es Gazebo³, un proyecto de software libre que incluye multitud de modelos y motores de física virtualizada. Ofrece una interfaz gráfica y control sobre los objetos y el mundo generado, además de la creación y modificación de actuadores y sensores personalizados. Por ejemplo, se pueden crear vehículos con diferentes sensores o casas con las que interactuar.

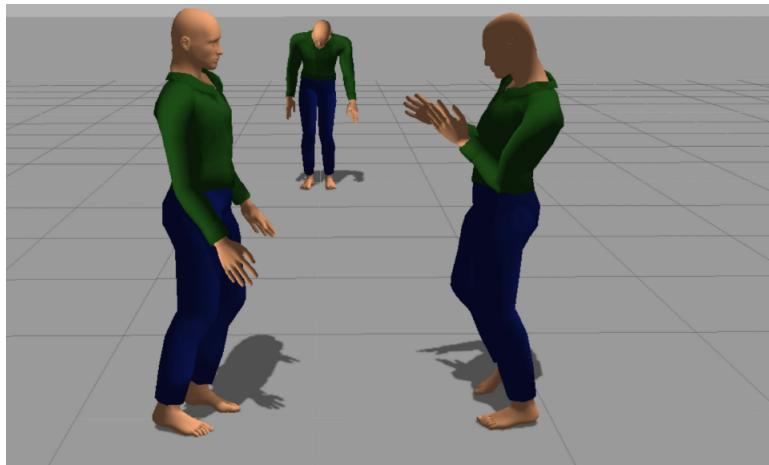


Figura 1.4: Personas simuladas en Gazebo

³Página web oficial de Gazebo : <http://gazebosim.org/>

1.3. Visión artificial en robots

La visión artificial, también conocida como visión por computador, es un subcampo de la inteligencia artificial en la que mediante el procesado y análisis de imágenes se trata de extraer información en un computador aplicando transformaciones y algoritmos basados en diferentes disciplinas como por ejemplo la estadística, geometría o *machine learning*.

La visión para los robots, al igual que para las personas, es una gran fuente de información del entorno. Las personas utilizamos el espectro visible de la luz, que se corresponde con lo que llamamos colores. Además, percibimos el mundo que nos rodea como un mundo en tres dimensiones debido a que tenemos dos ojos. Esto nos permite obtener propiedades del entorno y poder desplazarnos en él sin preocupaciones. Los robots utilizan sensores de visión de todo tipo, capaces de ver en la oscuridad o distinguir diferentes temperaturas. Además, en los últimos años, los sensores de visión han disminuido en precio y su utilización se ha visto incrementada notablemente en el mundo de la robótica, pasando a ser un equipamiento muy común en los robots. En este TFG, utilizaremos dos cámaras para obtener las imágenes que una vez procesadas aportarán la información necesaria para dotar de un comportamiento autónomo al dron.

Existen diferentes bibliotecas que recogen algoritmos y herramientas para simplificar la visión artificial. Cabe destacar *OpenCV*, como la biblioteca de código libre más extendida a la hora de realizar visión artificial en robots. Esta biblioteca, de la que hablaremos más adelante en detalle 3.3, ha facilitado la introducción y la aplicación de técnicas avanzadas de visión artificial para realizar el análisis y procesado de imágenes. Además, podemos encontrar en Internet numerosos ejemplos y tutoriales en los que se muestran las capacidades y aplicaciones de esta biblioteca.

DeepLearning (aprendizaje jerárquico), es una familia de técnicas de procesamiento de imágenes basada en *machine learning*. En los últimos años la aplicación de estas técnicas ha dado muy buenos resultados, especialmente dotando de una mayor robustez en comparación a los algoritmos más tradicionales creados para una tarea en específico.

Actualmente, las aplicaciones de visión artificial en robots son muy variadas, desde seguridad, como sistemas de detección de movimiento, pasando por el entretenimiento, como el sensor Kinect, hasta la accesibilidad para dotar de autonomía a personas que lo necesitan. Para esto se utilizan las siguientes técnicas:

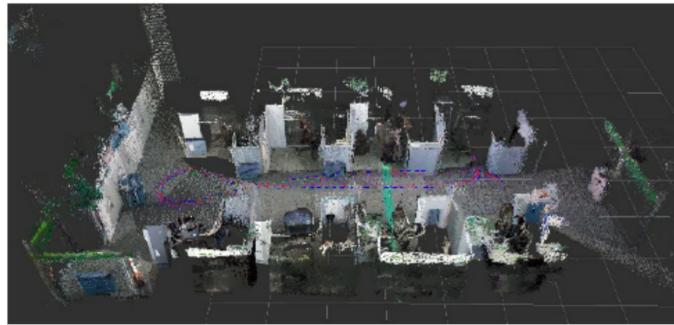
- Construcción de mapas: Es una de las primeras aplicaciones a través de visión y permite a los robots la creación de mapas a través de la detección de bordes, formas o profundidad. Esta información sirve también para poder navegar sobre sitios desconocidos o previamente han sido convertidos a un mapa 1.5.
- Autolocalización: Permite extraer información a un robot sobre la posición relativa respecto al resto del mundo que lo rodea, mediante el reconocimiento de patrones o balizas. Una técnica es el SLAM 1.5 (*Simultaneous Localization and Mapping*), que permite la autolocalización al mismo tiempo que se realiza un mapa del entorno.
- Detección e identificación de objetos: El reconocimiento o identificación de objetos1.5 se consigue a partir de la extracción de características únicas de cualquier objeto o persona que podemos encontrar en una imagen. La aplicación de DeepLearning está llevando cada vez más lejos los límites de este reconocimiento, dando lugar a un reconocimiento de objetos mucho más robusto, superando al ser humano en determinadas condiciones.
- Navegación y control visual: Permite la navegación autónoma o asistida de robots. Actualmente, se emplea en entornos industriales o el sector de la automoción para el desplazamiento de objetos o personas.

1.4. Robótica Aérea

Es una de las ramas de la robótica de mayor auge actualmente y sus aplicaciones son cada vez más extendidas. Pertenecen a este área los *Unmanned Aircraft Vehicle*(UAV),



(a) Reconocimiento de objetos en imagen



(b) Reconstrucción de mapas con técnicas SLAM

Figura 1.5: Visión artificial en robots

en español *Vehículo Aéreo No Tripulado*(VANT), o también conocidos como *drones*. Se trata de un vehículo capaz de volar, que puede o no recibir órdenes del exterior. Incluyen multitud de diferentes sensores para mantenerse en vuelo, aterrizar o despegar.

Históricamente, el origen de los UAV ha sido en aplicaciones militares, como en otras áreas de investigación. Una vez ha sido suficientemente desarrollado comienzan las aplicaciones civiles y su aplicación comercial e industrial. Durante la primera y la segunda guerra mundial se utilizaron drones para la obtención de mapas sin poner en peligro al piloto. Más tarde, en 1995 se utilizaron en Bosnia para tareas de vigilancia o análisis de daños, siendo especialmente importante para el reconocimiento nocturno. El modelo se llamaba *Predator* 1.6.



Figura 1.6: UAV Predator.

1.4.1. Aplicaciones actuales

Actualmente, gracias al avance de la estabilización electrónica, los UAV han alcanzado tamaños mucho más reducidos, como el *Hummingbird*⁴ o colibrí en español, de DARPA. Además son mucho más ágiles y mecánicamente simples. Han aparecido numerosos usos comerciales y civiles, aunque no desaparece el interés militar. Compañías como *Amazon* 1.7, están trabajando en proyectos para conseguir crear un sistema de envío de compra a domicilio utilizando *drones*, en concreto *cuadricópteros*.

Intel está diseñando comportamientos basados en grupos masivos, creando formaciones en el aire y dotando capacidad de pensamiento en grupo a los drones.

Otro de los usos es la exploración aérea, que incluye la inspección de embalses, líneas de alta tensión, campos agrícolas y la vigilancia. Este último caso es el de Alemania, que utiliza drones aéreos para evitar el ataque de grafiteros a vagones de tren⁵.

Uno de los campeonatos más recientes de programación para UAV es el *Mohamed Bin Zayed International Robotics Challenge*(MBZIRC)⁶. Con una recompensa de 5 millones de dólares, una de las pruebas consiste en localizar, seguir y aterrizar, coincidiendo con los objetivos principales de este Trabajo Fin de Grado.

Cuadricópteros

Existen diferentes tipos de drones en función del diseño y los componentes que los forman. Algunos son similares a los aviones, con alas y el mismo método de despegue y aterrizaje. Están pensados para largos períodos de tiempo y altas velocidades. Otros buscan una excepcional maniobrabilidad y estabilidad aérea. En este caso utilizan rotores, al igual que los helicópteros. A este grupo pertenecen los cuadricópteros. Se caracterizan por ser un helicóptero multi-rotor de cuatro brazos en forma de cruz. Los rotores se encuentran en el extremo de cada brazo.

Cuando los motores giran las hélices situadas en ellos generan un fuerza de empuje

⁴Más información en: <http://www.avinc.com/nano>

⁵Alemania pone a prueba drones contra los grafitis: http://www.bbc.com/mundo/noticias/2013/05/130528_tecnologia_drones_graffiti_alemania_aa

⁶Página Web oficial del campeonato: <http://www.mbzirc.com/>



(a) Modelo utilizado por Amazon Air Prime.



(b) Actuación de drones masivos durante la apertura de los J.J.O de Invierno 2018

Figura 1.7: Ejemplos de UAV civiles

vertical llamada *sustentación*. Ésta es perpendicular al movimiento de la hélice y depende de la velocidad a la que gira. La suma de cada fuerza en cada rotor produce una resultante. Los diferentes movimientos que puede describir el cuadricóptero se encuentran recogidos en la Figura 1.8. El color rojo indica que una potencia mayor ha sido aplicada, mientras que el verde, representa una potencia menor. Para evitar un fenómeno que en los helicópteros produce vueltas sobre sí mismo, la disposición de los motores sigue una forma de cruz, en la que cada par opuesto gira en el mismo sentido. Uno en el sentido de las agujas del reloj y el otro anti-horario.

Para que sea posible el despegue (Figura 1.8,e), esta resultante ha de ser superior al peso del UAV. Si es igual, el drone queda cernido en una altitud fija (*hovering*). Para aterrizar sería necesario una resultante menor que el peso del objeto (Figura 1.8,f).

El giro conocido como *yaw* (Figura 1.8,g y h) o *guiñada*, es el giro del plano horizontal al drone. Para girar a la derecha se transmite más potencia al par de motores que giran en sentido anti-horario. Si la potencia fuera superior en el otro par opuesto, giraría hacia la izquierda sobre sí mismo.

En el supuesto de que sólo uno de los motores aplicase más potencia que los demás, por ejemplo el delantero, el cuadricóptero se desplazaría hacia atrás, inclinando la parte trasera del vehículo hacia arriba. Esto se correspondería con el movimiento llamado

pitch (Figura 1.8,a y b) o *cabeceo*.

Por último, si aumentamos la potencia en uno de los motores laterales, por ejemplo la derecha, el vehículo se inclinará y trasladará hacia la izquierda, provocando un movimiento conocido como *roll* (Figura 1.8,c y d) o *alabeo*.

Comportamiento de los rotores

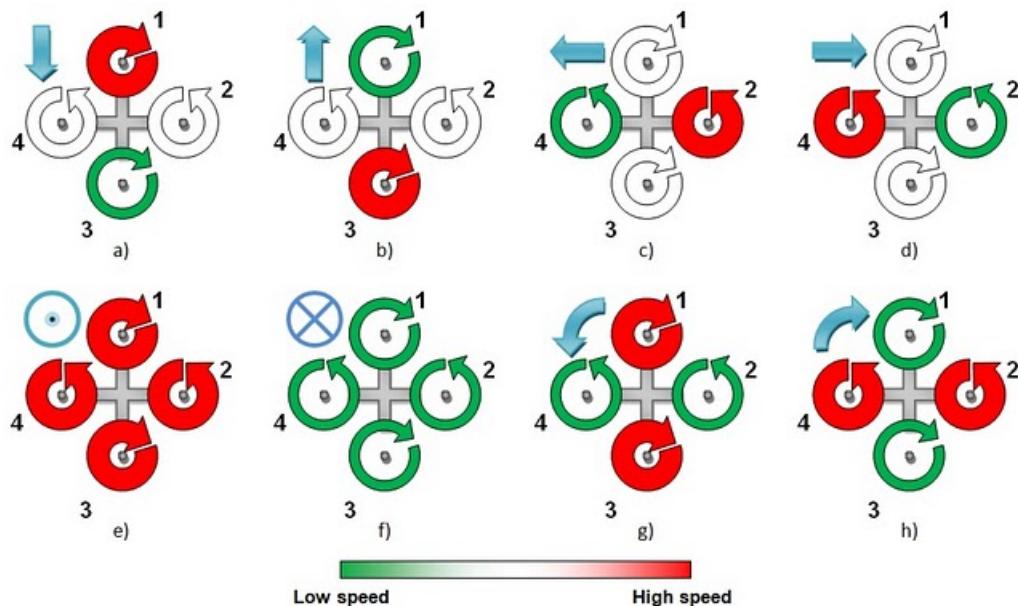


Figura 1.8: Relación entre la potencia de los rotores y el movimiento de un cuadricóptero

Los cuadricópteros pueden tener numerosos sensores a bordo desde acelerómetros, giroscopios, magnetómetros, ultrasonidos, incluso cámaras con resoluciones de hasta 4K. Todo ello se utiliza en combinación para conseguir una mayor estabilización durante el vuelo. Dentro de los actuadores encontramos los cuatro rotores pero también se pueden añadir pinzas para cargar objetos o, en caso de uso militar, armas y sus respectivos gatillos.

Algunos de los fabricantes de cuadricópteros más relevantes actualmente son:

- Parrot: Con modelos como el Ar.Drone que acercaron a un gran público el uso de los drones. Actualmente el modelo Bebop 2⁷ ofrece una cámara frontal con

⁷<https://www.parrot.com/us/drones/parrot-bebop-2>

resolución *FullHD*, soporte para controles en dispositivos móviles e integración con gafas de realidad virtual para teleoperar el dron.

- Erle: El Erle-Copter⁸ soportado oficialmente por Ubuntu y por el *middleware* robótico ROS (Robot Operating System), facilitando el desarrollo de software en este dron. Está pensado para la instalación de módulos nuevos y así adaptarse a diferentes situaciones.
- DJI: Una de las compañías que más drones vende anualmente, cargados de todo tipo de sensores, cámaras con las últimas tecnologías de estabilización y algoritmos inteligentes de navegación. Su modelo Inspire 2⁹ es muy utilizado en el mundo audiovisual y el cine.

Robótica Aérea en el proyecto JdeRobot

El proyecto JdeRobot de software libre para robótica lleva años desarrollando proyectos relacionados con la navegación, visión, autolocalización y virtualización de entornos con robots. Gracias a la popularización y a la reducción en coste de los drones se comenzó en el año 2013 una nueva línea de investigación en JdeRobot sobre los UAV. Los primeros proyectos han creado las bases sobre las que seguir investigando y han proporcionado la infraestructura necesaria. Sirven como base antecedentes directos y contexto cercano de este TFG.

Entre estos proyectos se encuentra el Trabajo de Fin de Grado(TFG) *Navegación visual en robots aéreos* de Alberto Martín [1]. Sus aportaciones fueron la de un driver llamado *ardrone_server*, que crea una interfaz capaz de comunicarse con el *AR.Drone* de la compañía Parrot. El mismo trabajo incluye una herramienta llamada *uav_viewer* 1.9, cuya función es obtener la información de los sensores y teleoperar los actuadores de dicho UAV. Por último, aportó un componente de visión y navegación llamado *object_tracking* que utiliza filtros de colores para el seguimiento de objetos a través de

⁸<http://erlerobotics.com/blog/Erle-Copter/>

⁹<https://www.DJI.com/es/inspire-2?site=brandsite&from=nav>

las imágenes recibidas por la cámara frontal y ventral del drone. El drone es capaz de un seguimiento autónomo de objetos, tanto en el suelo como en 3D.

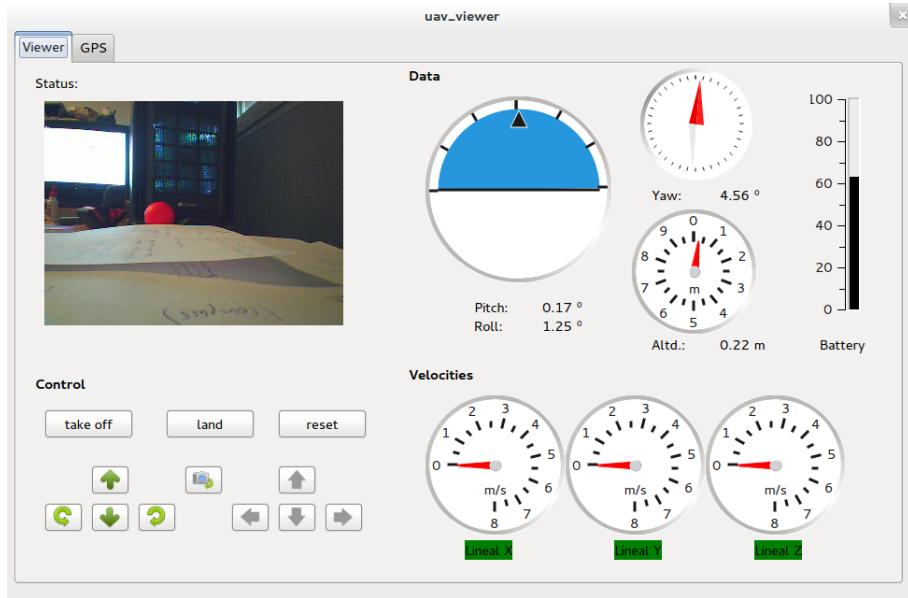


Figura 1.9: Ejemplo de interfaz de usuario del componentes Uav_viewer

Daniel Yagüe en su Proyecto Fin de Carrera *Cuadricóptero AR.Drone en Gazebo y JdeRobot* [2], desarrolló un modelo y un driver en el simulador Gazebo del mismo AR.drone 1.10 que utilizó Alberto Martín. Esto permite tanto la simulación de los datos sensoriales como de la virtualización realista de un comportamiento cercano a dicho drone. Adicionalmente, programó diferentes aplicaciones de navegación autónomas como el seguimiento de balizas por posición, de carretera o de otro cuadricóptero.

Por otro lado Alberto López-Cerón, con su TFM *Autolocalización visual robusta basada en marcadores* [3], creó un algoritmo capaz de estimar la posición de la cámara a partir de la detección de marcadores o balizas. Manuel Zafra siguió desarrollando la idea de Alberto López-Cerón en *Seguimiento de rutas 3D por un drone con autolocalización visual con balizas* [4]. Diseñó un algoritmo de navegación en interiores basado en autolocalización mediante la visión artificial en simulador 1.11.

Jorge Vela se centró en el *Despegue, navegación y aterrizaje visuales de un drone usando JdeRobot* [5]. Sentó las bases para el despegue y aterrizaje controlado utilizando

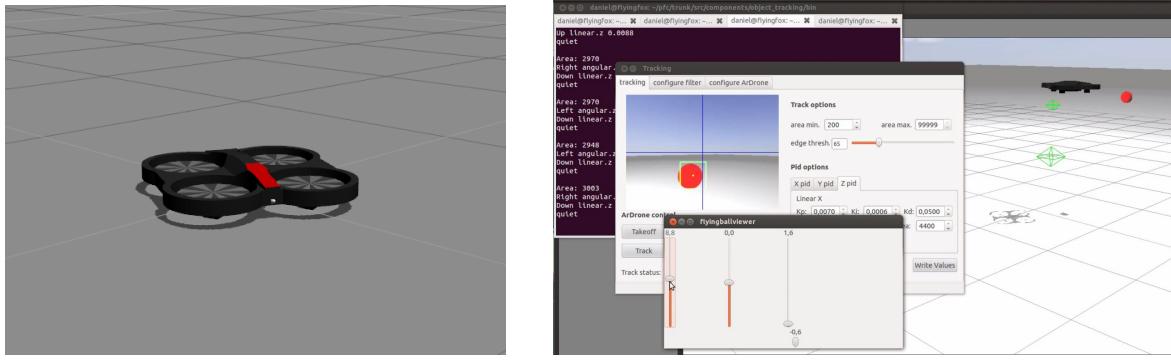


Figura 1.10: Ejemplos de ArDrone simulado en Gazebo

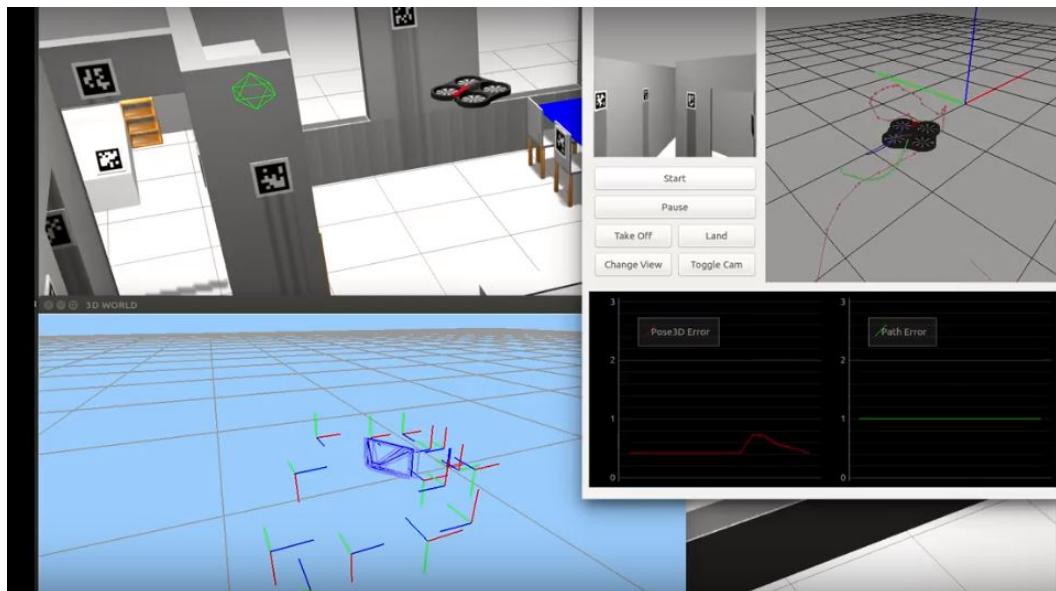
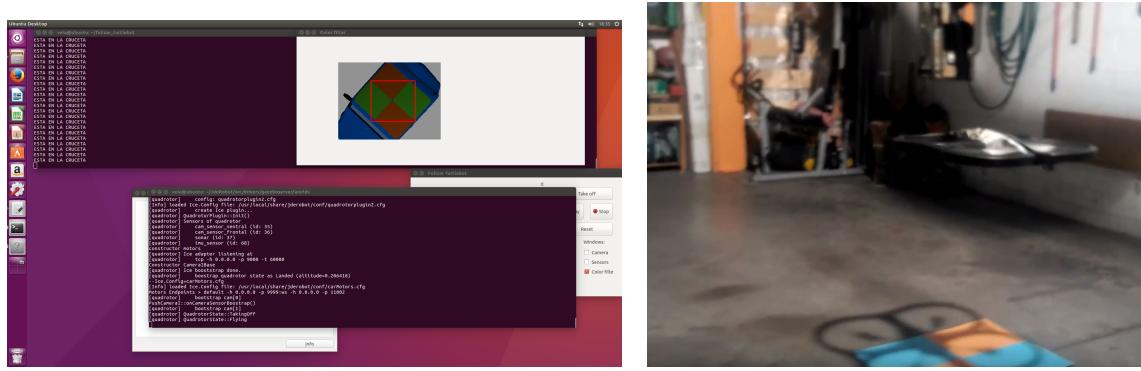


Figura 1.11: Ejemplo de algoritmo de navegación basado en autolocalización en Gazebo.

visión artificial en un drone real 1.12



(a) Algoritmo de despegue y aterrizaje encontrando la baliza visual en Gazabo. (b) Ar.Drone en vuelo sobre baliza visual.

Figura 1.12: Ejemplos de aterrizaje y despegue en Gazebo y el dron real.

Jesús Saiz, en su TFG *Programación de un dron para seguimiento autónomo de trayectorias en 3D* [6], integró el algoritmo de despegue y aterrizaje controlados y la autolocalización visual basada en balizas para crear un autómata de estados finito que navega de forma autónoma a través de rutas predefinidas en el simulador Gazebo.

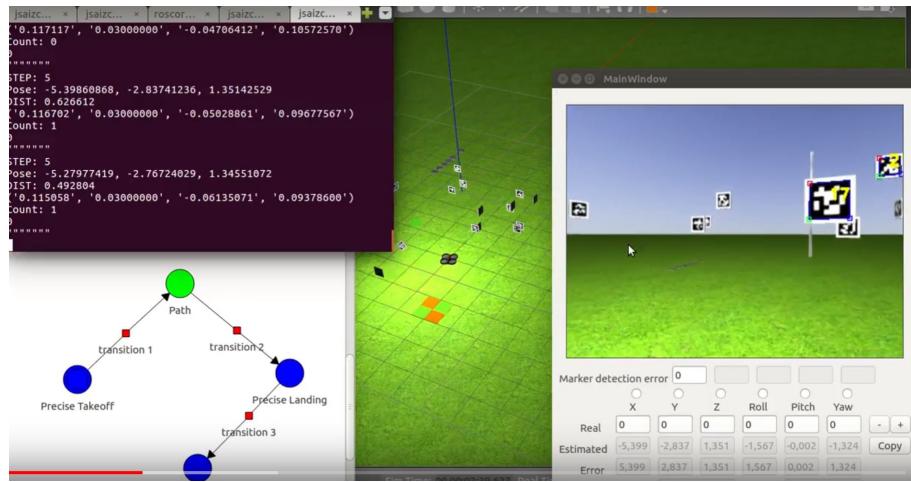


Figura 1.13: Ejemplo de navegación a través del seguimiento de rutas.

Siguiendo con las bases aportadas por todos estos proyectos, en este TFG se programará el comportamiento autónomo de despegue controlado y aterrizaje de un

drone sobre una baliza visual y la navegación a partir de la autolocalización. Se utilizará un drone real, empleando la infraestructura existente en JdeRobot para estos cuadricópteros. Con ello se pretende unificar los avances anteriores en un drone real y dejó la puerta abierta a nuevos comportamientos más sofisticados.

En el próximo capítulo se explicarán los objetivos y la metodología propuesta para resolverlos. En el tercer capítulo se expondrán con profundidad la infraestructura y herramientas utilizadas. En el cuarto, se describirá el desarrollo de todos los componentes que forman este proyecto. En quinto lugar, se detallarán los distintos experimentos para validar experimentalmente la solución programada. Para terminar, unas conclusiones aportarán un visión global del conjunto y los conocimientos extraídos.

Capítulo 2

Objetivos

En este capítulo describiremos los objetivos planteados para el Trabajo de Fin de Grado, los requisitos exigidos y la metodología utilizada para el desarrollo.

2.1. Problema a abordar

El objetivo principal de este trabajo es desarrollar una aplicación que a través de técnicas de visión artificial dote de navegación autónoma a un dron real. El resultado consistirá en un despegue, una navegación en tres dimensiones guiada por balizas y por último, una fase de aterrizaje.

Este objetivo global se ha dividido en varios subobjetivos concretos:

- **Desarrollo e integración del módulo de autolocalización 3D a partir de balizas visuales:** Este módulo proporciona la posición relativa del dron basándose en la detección de balizas visuales y cálculos geométricos. Dentro del proyecto JdeRobot existe un módulo basado en los trabajos previos mencionados en el capítulo anterior [3] [4] [6] que será revisado, refinado e integrado en la aplicación final.
- **Desarrollo e integración de un módulo de navegación por balizas visuales:** Este módulo será el encargado de buscar y navegar en un entorno real

utilizando información del módulo de autolocalización basado en balizas visuales. La navegación deberá ser totalmente autónoma (sin teleoperación).

- **Programación del comportamiento del dron en un autómata de estados finito:** La aplicación final será un único algoritmo basado en un autómata de estados finito que tendrá las siguientes fases: Despegue, aterrizaje, búsqueda y navegación basada en balizas visuales. Esta programación se apoyará en el despegue y aterrizaje existentes en el proyecto de JdeRobot [5] [6], que se refinarán y adaptarán utilizando la herramienta de autómatas **VisualStates**.
- **Validación experimental en el cuadricóptero real** Se realizarán pruebas unitarias para validar los diferentes módulos constituyentes y pruebas del sistema completo para demostrar el correcto funcionamiento de la solución desarrollada. Se explorarán varias configuraciones ejecutando el software tanto en un ordenador externo como en un ordenador a bordo del cuadricóptero.

2.2. Requisitos

La solución desarrollada deberá satisfacer, adicionalmente, los siguientes requisitos:

- Los componentes y aplicaciones desarrollados han de estar integrados en la plataforma JdeRobot-5.6.4.
- El control de navegación del cuadricóptero ha de ser fluido y ágil, de modo que pueda mantenerse estable al menos 5 segundos en cada baliza, evitando en todo momento perder del campo de visión el objetivo.
- Los componentes y aplicaciones desarrollados han de ser computacionalmente eficientes y deben ser modulares.
- El sistema operativo debe estar basado en la distribución GNU/Linux Ubuntu 16.04.
- Programado en el lenguaje Python, versión 2.7.

2.3. Metodología

Para materializar los objetivos y requisitos previamente mencionados es necesario aplicar algún método que defina las distintas etapas y estados. En este Trabajo de Fin de Grado se aplica el método de desarrollo en espiral. Este modelo, creado por Barry Boehm en 1986, se utiliza frecuentemente en la ingeniería de software. Se basa en una serie de iteraciones en bucle. En cada ciclo, se realiza un conjunto de cuatro actividades:

1. **Determinar los objetivos:** Poner limitaciones definidas en forma de objetivos o requisitos. Dividir el proyecto en partes más pequeñas.
2. **Análisis del riesgo:** Estudiar los riesgos de cada uno de los objetivos que se abordan. Evaluar las alternativas posibles en caso de amenazas.
3. **Desarrollar y probar:** Verificación de la tarea actual. Al mismo tiempo, se realiza un análisis para encontrar nuevos factores de riesgo, como errores que se podrían arrastran a la próxima iteración.
4. **Planificación:** Establecer y definir las fases anteriores.



Figura 2.1: Representación del desarrollo en espiral.

Durante el desarrollo de este proyecto se han establecido reuniones periódicas con el tutor. En ellas revisábamos los objetivos anteriormente fijados y los resultados

obtenidos. Si alguno de los objetivos generaba algún problema o no se llegaba al resultado deseado, se aplazaban o se profundizaba en la raíz del problema. A continuación se determinaban los objetivos de nuestro próximo encuentro.

Como parte de la evaluación de los objetivos propuestos, ha sido fundamental la utilización del mediawiki¹ de la plataforma JdeRobot. En él están publicados, a modo de cuaderno de bitácora, los hitos y progresos, incluyendo también imágenes o vídeos.

Para el seguimiento y almacenamiento del software desarrollado se ha empleado la herramienta de control de versiones GIT. Todo el código relacionado con este proyecto es software libre y es accesible en el repositorio².

2.4. Planificación

Para conseguir los objetivos fijados anteriormente se ha seguido el siguiente plan de trabajo:

- **Formación y familiarización con el entorno JdeRobot:** Incluye el preparación de las dependencias necesarias para la instalación del entorno. Estudio de las diferentes bibliotecas, interfaces y componentes. Aprendizaje y profundización de lenguajes de programación como Python y C++, así como la herramienta para las comunicaciones ICE y la biblioteca de visión OpenCV.
- **Aprendizaje de la herramienta VisualStates:** Necesaria para la generación de autómatas de estado finito. Se han generado varios ejemplos para conocer las diferentes secciones, para la inserción de variables, funciones y nuevos estados.
- **Aprendizaje e implementación del módulo de despegue, búsqueda en espiral y aterrizaje:** Necesario para conocer el funcionamiento de los antecedentes ya existentes de control de un drone para seguir una ruta de control de un drone para aterrizar usando visión. Se ha adaptado y mejorado el código

¹<http://JdeRobot.org/Andresjhe-tfg>

²<https://github.com/RoboticsURJC-students/2014-tfg-Andres-Hernandez>

para ganar rendimiento y robustez. También se han buscado soluciones propias de control y se han integrado todos en una máquina de estados finito programada con la herramienta **VisualStates**.

- **Aprendizaje y familiarización con técnicas de autolocalización desde balizas visuales:** Necesario para conocer el funcionamiento de las balizas visuales y cómo obtener las diferentes coordenadas a partir de su detección. Se ha realizado el estudio de antecedentes existentes en el proyecto JdeRobot [3] [4] junto a la información publicada en la página oficial de AprilTags ³.
- **Validación experimental:** Se ha diseñado una secuencia de pruebas unitarias para poder validar incrementalmente la solución, siendo la última prueba un ejercicio que reunirá y ejecutará de principio a fin las pruebas unitarias anteriormente realizadas y validadas. Ha sido necesaria la instalación de sistema operativo, herramientas y aplicaciones en el co-procesador para poder ejecutar la aplicación final.

³<https://april.eecs.umich.edu/>

Capítulo 3

Infraestructura

En este capítulo se describen los programas y dispositivos en los que nos hemos apoyado para la elaboración de este proyecto. Debido a la naturaleza de la plataforma JdeRobot, el sistema operativo que se ha elegido para el desarrollo y ejecución de los componentes ha sido Linux (la distribución Ubuntu 16.04) y el lenguaje de programación Python (versión 2.7 para compatibilidad de JdeRobot con ROS-Kinetic).

3.1. Parrot Ar.Drone 2

Este es el drone que ha utilizado durante las pruebas reales. Parrot¹ es un fabricante de dispositivos de diferente naturaleza, que van desde manos libres para teléfonos pasando por todo tipo de robots. Fue uno de los fabricantes que más popularizaron los drones a partir de 2010 con su primer Ar.Drone.

Este drone está dotado de una API de comunicaciones que forma parte de un SDK² que proporciona el fabricante de manera gratuita para obtener los datos de los sensores y/o control de los motores del cuadricóptero. Se ha utilizado la versión 2.4.

Posee dos cámaras, una ventral y otra frontal, incluye un acelerómetro y todo el procesado se realiza en un ARM de dos núcleos a bordo. Por último, está dotado de

¹<https://www.parrot.com/global/>

²<http://developer.parrot.com/>

Wi-Fi como canal de comunicaciones por defecto y genera un punto de acceso sin contraseña para que los dispositivos se conecten y comuniquen con él.

3.2. Intel Compute Stick

Para dotar de autonomía y potencia de procesado al Ar.Drone 2, se ha decidido utilizar un co-procesador a bordo. Se ha utilizado este dispositivo Intel Compute Stick que entra dentro de la categoría de *miniordenadores* u ordenadores de tamaño reducido. La principal ventaja es su relación peso y rendimiento, ya que está equipado con un procesador de dos núcleos, capaz de ejecutar 4 hilos simultáneamente. Cuenta con Wi-Fi integrado, además de un puerto USB, lector de tarjetas SD y HDMI como salida de vídeo.



Figura 3.1: Imagen de Intel compute Stick

3.3. Biblioteca OpenCV

OpenCV³ es una biblioteca de visión artificial de código abierto, que tiene un conjunto de transformaciones y operaciones con imágenes o matrices que facilitan el procesamiento de las mismas. Es el estándar internacional de facto en procesamiento de imágenes. Está programada en C++ y Python y en este TFG se ha utilizado la versión 3.4.

En este proyecto se ha utilizado para realizar filtros de color en imágenes para identificar la balizas de despegue y aterrizaje. Otro uso es para realizar transformaciones morfológicas en imágenes como la erosión y la dilatación, que permiten eliminar la sombra que proyecta el drone. Por último, de esta biblioteca también se han utilizado transformaciones geométricas de matrices para estimar la posición relativas del drone a partir de la información de balizas en una imagen de dos dimensiones.

3.4. Biblioteca AprilTags

En este proyecto se utilizará una librería de balizas visuales denominada *AprilTags*⁴. Es un sistema de visualización fiduciaria. Estas balizas se basan en símbolos diseñados para ser fácilmente reconocidos del resto del entorno 3.2. Puede detectar uno o varios símbolos en la misma imagen, además de proporcionar información como la identificación y posición de cada símbolo dentro de una imagen de cada uno. Es un sistema robusto cuyo funcionamiento es independiente del ángulo y diferentes situaciones de luminosidad en la imagen.

³<https://opencv.org/>

⁴<https://april.eecs.umich.edu/software/apriltag/>

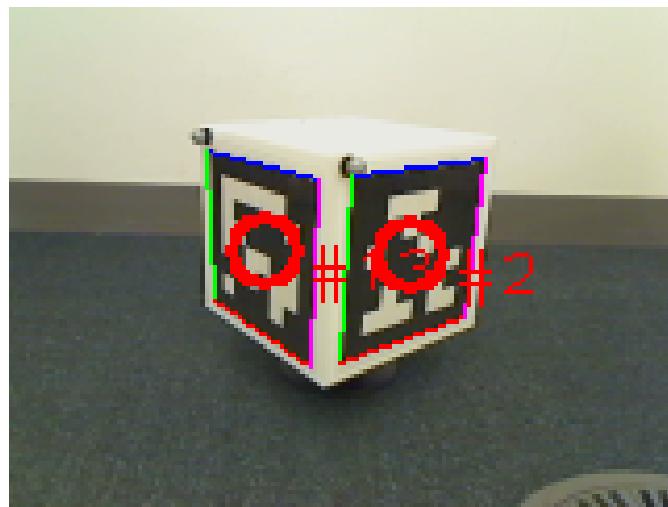


Figura 3.2: Ejemplo de detección en AprilTag

Los símbolos se dividen en diferentes familias. Estas familias utilizan un número de bits y de distancia de Hamming predefinidos. Dentro de cada familia, se generan los diferentes símbolos, asignando un único ID o identificador. Esto permite el reconocimiento en caso de tener varias balizas al mismo tiempo en el campo de visión.

Sus aplicaciones son muy variadas, desde la captura de movimiento en objetos hasta sistemas de navegación basada en balizas. Otra de sus aplicaciones es la de realidad aumentada, sustituyendo el símbolo por una imagen virtualizada.

Además del sistema de balizas se proporciona una biblioteca que ofrece funciones de identificación de estas balizas en imágenes. Esta biblioteca está escrita en C y Java pero Ed Olson de Massachusetts Institute of Technology (MIT) ha creado una adaptación en C++. El código⁵ es abierto y está protegido bajo la LGPL v2.1. Entre sus dependencias se encuentran OpenCV y Eigen3, ambas son bibliotecas de tratamiento de imagen en Linux. A través de la utilización de un recubrimiento⁶ para Python ha sido posible su integración con la aplicación final de este TFG.

⁵<https://svn.csail.mit.edu/apriltags>

⁶<https://github.com/swatbotics/AprilTag/toree/máster/Python>

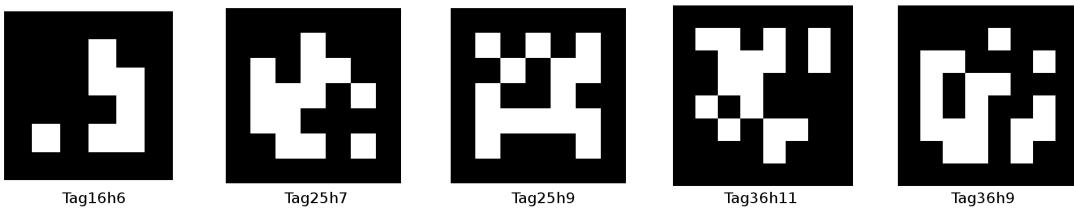


Figura 3.3: Ejemplos de familias en AprilTag

3.5. Entorno JdeRobot

En el mundo de la robótica existen diferentes plataformas que simplifican y aportan las herramientas necesario para el desarrollo de aplicaciones en robots. JdeRobot es una de ellas y consiste en una colección de drivers, herramientas y bibliotecas robóticas, domóticas y de visión artificial. Estas piezas de software están escritas en diferentes lenguajes como C++ o Python y su interoperación se realiza a través de interfaces ICE o interfaces ROS. En ella participan desarrolladores de diferentes niveles desde profesionales del sector, profesores, alumnos de la Universidad Rey Juan Carlos y de otras universidades internacionales. El código fuente es libre y está bajo la licencia GPL v3 y la documentación se encuentra protegido bajo la licencia de Creative Commons by-SA. La versión utilizada en este proyecto es la 5.6.4⁷.

Durante el desarrollo de este TFG, componentes de esta plataforma como `uav_viewer`, `uav_viewer.py`, `slam_markers` y `ardrone_server` han servido de referencia y han tenido una gran relevancia para el aprendizaje de la plataforma.

Se repasan a continuación los más relacionados con este trabajo:

3.5.1. Ardrone_Server

Este componente fue desarrollado en el TFG de Alberto Martín [1] y permite a partir de un protocolo basado en interfaces ICE el envío y lectura de comandos específicos del SDK del Ar.Drone 2. Como resultado, podemos modificar la velocidad

⁷<https://github.com/JdeRobot>

de los motores para cambiar la posición del drone, recibir las imágenes de sus cámaras, envío de órdenes predeterminadas como el aterrizaje o despegue, recibir la información de sensores, etcétera.

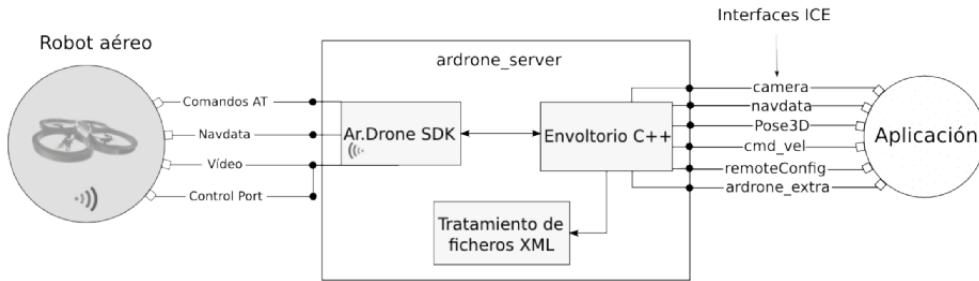


Figura 3.4: Estructura de ardronne_server

Está programado en C++ y es absolutamente necesario para cualquier comunicación con el drone real.

3.5.2. Teleoperadores uav_viewer y uav_viewer_py

Ambos son componentes de JdeRobot y ofrecen una GUI para enviar comandos tanto al cuadricóptero real, como al Ardrone simulado o mostrar la información de los sensores y cámaras. Los comandos se envían a través de interfaces ICE al componente `ardrone_server`, que los transforma a su vez en comandos para el SDK del Ar.Drone 2. La principal diferencia entre ambos es el lenguaje en el que han sido desarrollados: `uav_viewer_py` está escrito en Python y ofrece un GUI más parecido a los mandos físicos de teleoperación de drones mientras que `uav_viewer` 1.9 está en C++.

3.5.3. Herramienta Color Tuner

Este componente facilita la configuración de filtros de color utilizando OpenCV. En su interfaz gráfica se selecciona el espacio de color que queremos utilizar, RGB, YUV y HSV, siendo esta última la utilizada por nuestras aplicaciones. Dentro de la aplicación, las ventanas *Source image* y *Filtered image* mostrarán la imagen recibida y

su versión filtrada respectivamente. Para especificar los valores que se aplicarán para filtrar, se modifica la posición deslizadores o *sliders* que representan los valores máximos y mínimos de H (*hue* o tinte), S (saturación) y V (value o luminosidad).

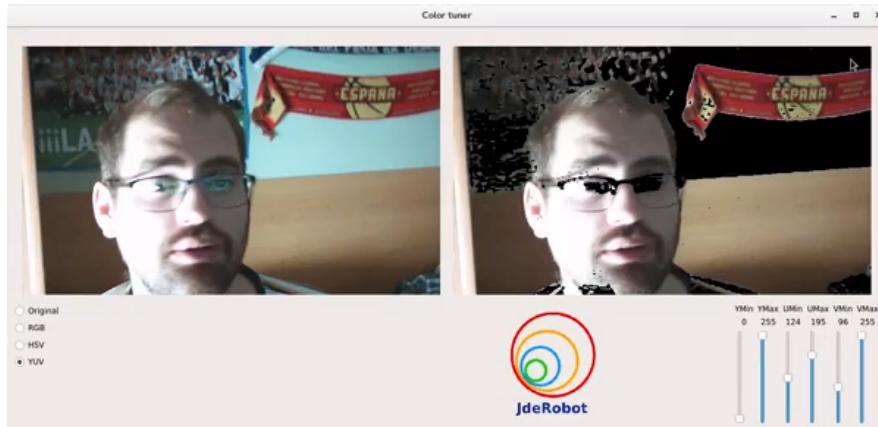


Figura 3.5: Ejemplo de Color Tuner aplicando filtros de color

3.5.4. Componente `slam_markers`

Esta herramienta permite mediante la aplicación de una serie de algoritmos de autolocalización basado en balizas visuales. Su algoritmo analiza las imágenes en 2D recibidas por la cámara y utiliza tanto la biblioteca OpenCV como AprilTags. Primero identifica si existen balizas. Una vez localizada, aplica funciones para calcular la posición y orientación en tres dimensiones de la cámara con respecto a la baliza. En el siguiente paso se ejecuta un proceso de fusión temporal y fusión espacial de la estimación obtenida. Por último, devuelve a través de una interfaz ICE las coordenadas 3D obtenidas como resultado de todo el proceso.

Ha sido desarrollada por Felipe Pérez en su TFM⁸. Ha sido programada en C++ y actualmente se encuentra aún en desarrollo. Para nuestro TFG se ha reimplementado su algoritmo en Python para facilitar la integración con la aplicación final y mejorar el rendimiento.

⁸<http://jderobot.org/Flperez-tfm>

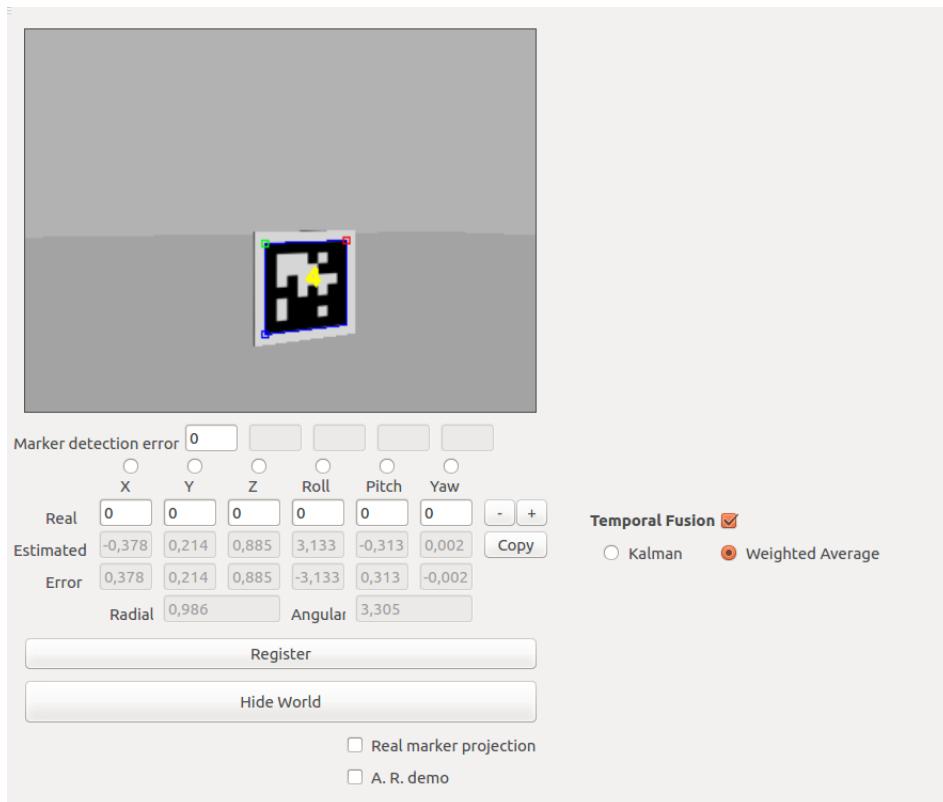


Figura 3.6: Ejemplo de `slam_markers` identificando una baliza

3.5.5. VisualStates

Esta herramienta⁹ forma parte de la infraestructura JdeRobot y su principal función es facilitar la creación de programas para robots basados en máquinas de estados finito con estados y transiciones.

Se caracteriza por tener una interfaz de usuario en la que podemos generar o modificar estados, teniendo siempre uno como principal a partir del cual comenzará la ejecución. Cada estado se compone por un código que será ejecutado en bucle hasta que se realice una transición a otro estado diferente. La herramienta genera como salida un programa que materializa el autómata, y que puede estar escrito tanto en Python como en C++.

⁹<http://jderobot.org/VisualStates>

Las transiciones se dibujan entre los diferentes estados existentes especificando las condiciones temporales o basadas en variables que provocarán el cambio de estado.

Adicionalmente proporciona un menú de configuración de interfaces ICE compatibles con JdeRobot, una sección de variables y funciones para que se compartan entre los diferentes estados. El menú también permite la modificación de la duración de las iteraciones periódicas en las que se van ejecutando los estados.

Por último, tiene la capacidad de generar ejecutables y sus respectivos ficheros de configuración ICE. Esto facilita el empaquetado y el despliegue de los programas en un único fichero ejecutable.

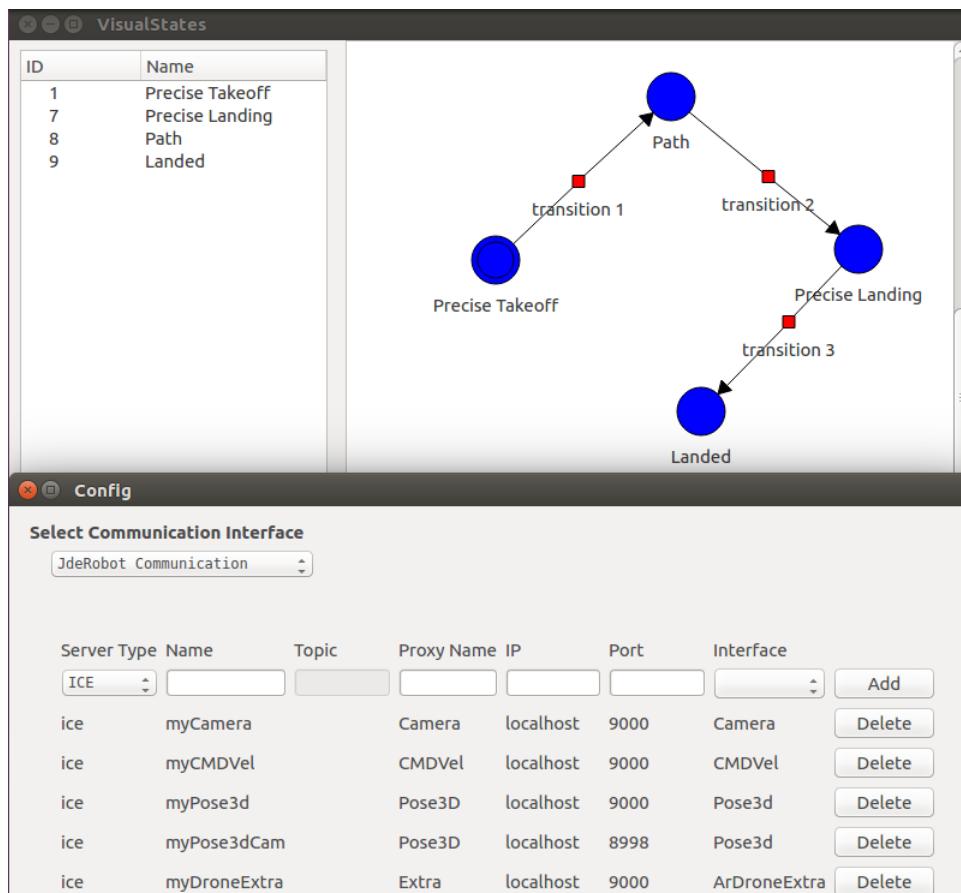


Figura 3.7: Ejemplo de autómata generado con VisualStates

3.6. Bibliotecas ICE

ICE¹⁰ es un *middleware* de Código Abierto orientado a objetos que proporciona las herramientas, APIs y librerías necesarias para simplificar las comunicaciones entre componentes usando modelos basados en cliente y servidor y objetos distribuidos. En JdeRobot es la plataforma elegida como infraestructura para las comunicaciones entre nodos. Proporciona una capa transparente que se encarga de abrir y cerrar conexiones, la serialización de información, retransmisión de paquetes perdidos, etcétera. La versión utilizada en este proyecto es la 3.6

3.7. Simulador Gazebo

Este simulador de código abierto es la herramienta en la que se realizarán algunas las pruebas experimentales. En este TFG se ha utilizado la versión Gazebo 7.12 para simular al drone en diferentes escenarios. De este modo podremos evaluar previamente el código antes de ejecutarlo en el drone real. Los escenarios simulados o mundos se crean con la propia aplicación y se definen con la extensión “.world“. Están escritos mediante SDF (*Simulation Description Format*) que a su vez, está basado en el lenguaje XML y es muy popular en entornos de simuladores para robots. Estos mundos contienen modelos virtuales de robots que pueden tener un papel activo como un drone o pasivo como las balizas. El comportamiento activo se materializa mediante plugins que permiten la ejecución de código, interacciones con el motor de físicas del simulador, la obtención de imágenes desde cámaras virtuales, etcétera.

¹⁰<https://zeroc.com/>

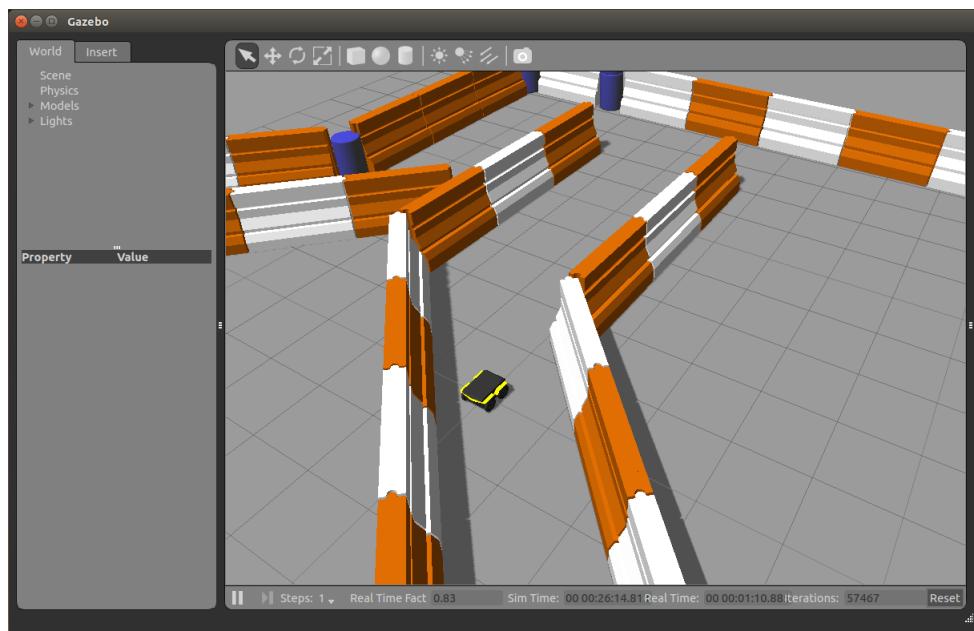


Figura 3.8: Ejemplo de simulador Gazebo

Capítulo 4

Navegación autónoma de un drone guiado por balizas visuales

En este capítulo se describe la solución desarrollada para conseguir que un drone navegue autónomamente guiado por balizas visuales y utilizando la infraestructura mencionada anteriormente. Se explicará el diseño alto nivel de la solución y se describirá con mayor detalle las partes desarrolladas.

4.1. Diseño

El objetivo de este proyecto es desarrollar un algoritmo para dotar a un drone de un comportamiento completamente autónomo desde el despegue, hasta el aterrizaje, ambos controlados, pasando por la localización y aproximación a diferentes posiciones 3D asociadas a las balizas, pero desconocidas de antemano. Todo esto basándose únicamente en balizas de apoyo visual.

El componente desarrollado necesita dos ficheros de configuración. Primero un fichero xml llamado *calibration.xml* que almacena los parámetros necesarios para la detección robusta de balizas arlequinadas. Segundo un fichero con la secuencia deseada de balizas AprilTags a las que se quiere que el drone visite en su navegación autónoma.

En la figura 4.1 se pueden ver las entradas y salidas de flujos de información.

CAPÍTULO 4. NAVEGACIÓN AUTÓNOMA DE UN DRONE GUIADO POR BALIZAS VISUALES

Adicionalmente, se han sido incluido los ficheros imprescindibles en cada módulo para su correcto funcionamiento. Esto nos dará una imagen de alto nivel de la solución. Las comunicaciones entre procesos se llevan a cabo mediante interfaces de la biblioteca ICE.

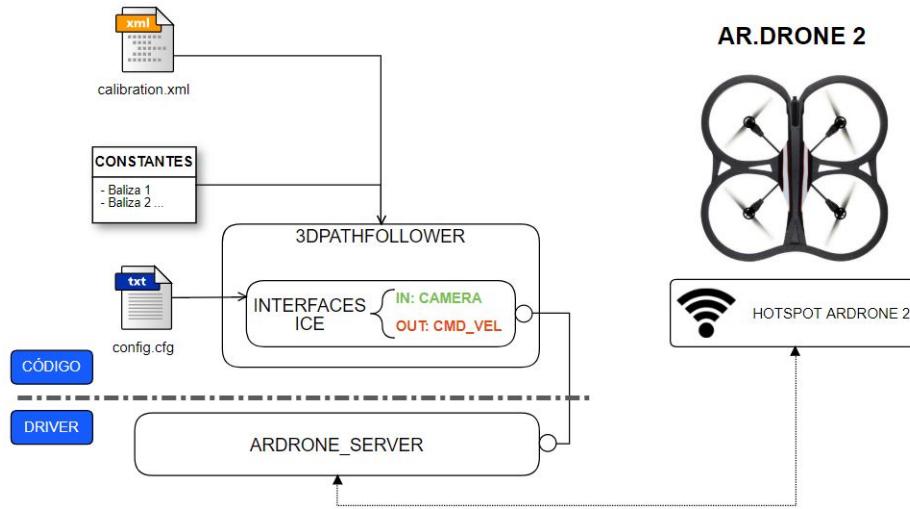


Figura 4.1: Entradas y salidas de la solución final.

El componente 3DPahtFollower es la aplicación mencionada anteriormente. Está compuesta por los diferentes estados del autómata, tal y como se puede apreciar en la figura 4.2. Se han agrupado los diferentes estados para una mejor comprensión del despegue controlado, navegación y búsqueda rotacional y por último, búsqueda en espiral y aterrizaje controlado. Cada uno de los estados se componen de una capa de percepción visual, que llevará a cabo la tarea de identificar las balizas y otra capa de control, que calculará y generará el comportamiento que se enviará al drone a través de las interfaces ICE. La navegación autónoma del drone se entiende aquí como la visita en secuencia a una serie de balizas AprilTags que están situadas en posiciones desconocidas por el drone, pero cercanas cada una a la siguiente en la secuencia.

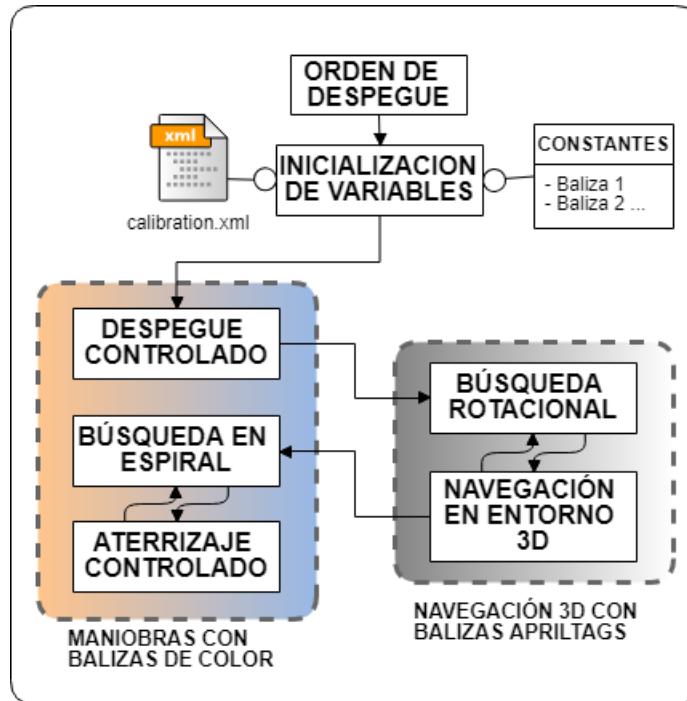


Figura 4.2: Diagrama de bloques del componente 3DPathFollower.

A continuación vamos a explicar cada estado y sus respectivas transiciones de los que se compone *3DPathFollower* con mayor profundidad, siendo la creación de un algoritmo de navegación, el diseño de la infraestructura y la aplicación *Calibration Tool* las mayores aportación de este TFG, por lo que serán los apartados en los que haremos mas hincapié. Sin embargo, al tratarse de un TFG de integración también explicaremos los módulos en los que nos hemos basado y los cambios que hemos aplicado para su correcto funcionamiento en la aplicación final.

4.2. Detección visual de las balizas

La percepción se ha basado en la detección de dos tipos de balizas visuales: Una baliza bicolor para las fases de despegue y aterrizaje y otro tipo de balizas tipo AprilTags.

4.2.1. Balizas de aterrizaje y despegue

Durante la realización de este TFG ha sido necesaria la configuración de filtros los de color de las balizas en los que se basa principalmente el despegue y aterrizaje controlado de la aplicación principal. Se utiliza la biblioteca de OpenCV para realizar las operaciones sobre imágenes.

Se ha escogido utilizar una baliza de color arlequinada 4.3 compuesta por cuatro cuadrantes y 2 colores lo más distantes posibles en el espacio de color HSV (como por ejemplo naranja y verde), colocados de forma no contigua. Esto permitirá aumentar la robustez y reducir los posibles errores que se produzcan debido al ruido de otros colores en el ambiente.

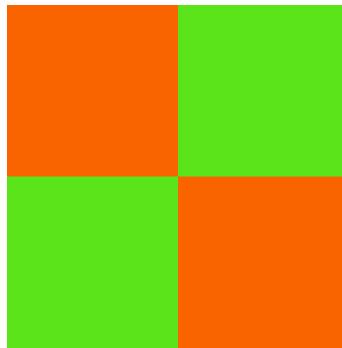


Figura 4.3: Ejemplo de baliza de color arlequinada.

3DPathFollower utiliza la suma de dos filtros de color, llamados *Primary* y *Secondary* los diferentes valores máximos y mínimos de los componentes HSV y de las transformaciones morfológicas sobre las imágenes filtradas que explicaremos a continuación. Estas transformaciones son la erosión y dilatación. La combinación de estas técnicas permite la eliminación de impurezas y el suavizado de la imagen, dotando de mayor robustez al filtro de color.

Se aplicará el procesado para cada uno de los dos colores, llamando a la función `hsvFilter` (importada de Color Tuner) y acto seguido se realizarán las transformaciones morfológicas a la imagen filtrada en HSV:

CAPÍTULO 4. NAVEGACIÓN AUTÓNOMA DE UN DRONE GUIADO POR BALIZAS VISUALES

```
Pmask , PmaskHSV = hsvFilter (PH_max , PS_max , PV_max ,  
PH_min , PS_min , PV_min , image , hsv )  
PmaskHSV = cv2 . erode (PmaskHSV , kernel , iterations = PErode )  
PmaskHSV = cv2 . dilate (PmaskHSV , kernel , iterations = PDilate )
```

Donde *kernel* es el tamaño de píxeles al que se aplicará y *iterations* se corresponde con la intensidad. El resultado consistirá en la suma de ambas imágenes filtradas:

```
filterImage = PmaskHSV+SmaskHSV
```

4.2.2. Balizas de autolocalización

Para la identificación de las balizas, tanto durante la búsqueda como la aproximación, se utilizarán las funciones nativas de AprilTags. Necesitaremos crear un detector para nuestra familia (36H11 3.3), para más adelante aplicarlo sobre una imagen convertida a escala de grises:

```
self . options = apriltag . DetectorOptions ()  
self . detector = apriltag . Detector ( self . options )  
detections = self . interfaces . detector . detect (gray )
```

Para obtener esta estimación hemos tomado como referencia el algoritmo utilizado por la aplicación Slam_Markers 3.5.4 y lo hemos implementado en el lenguaje Python.

El algoritmo para estimar la posición relativa en 3D a partir de una imagen en 2D, serie de transformaciones y cálculo de matrices que a partir de las cuatro esquinas detectadas por AprilTags obtiene el vector basado en las coordenadas X,Y,Z:

```
retVal , rvec , tvec =cv2 . solvePnP ( self . interfaces . m_MarkerPoints  
, detection . corners  
, self . interfaces . cameraMatrix  
, self . interfaces . distCoeffs )  
rodri = cv2 . Rodrigues ( rvec )  
#We get X,Y and Z
```

CAPÍTULO 4. NAVEGACIÓN AUTÓNOMA DE UN DRONE GUIADO POR BALIZAS VISUALES

```
cameraPosition = -np.matrix( rodri [0] ).T * np.matrix( tvec )
self.interfaces.x = cameraPosition.item(0)
self.interfaces.y = cameraPosition.item(1)
self.interfaces.z = cameraPosition.item(2)
```

Para calcular conocer la rotación relativa de nuestro drone con respecto a la posición baliza, se realiza una serie de transformaciones que se basan en los ángulos de euler:

```
#We get roll , pitch and yaw from Euler Angles
eulerAngles =
self.interfaces.rotationMatrixToEulerAngles( rodri [0] )
```

4.3. Autómata de navegación

Este componente está formado por un algoritmo de navegación autónoma basado en un control representado por un autómata de estados finito. Ha sido realizado con la herramienta VISUAL STATES, la cuál nos ha permitido tanto desarrollar el código, como la integración con las diferentes bibliotecas y partes del sistema en un solo programa.

CAPÍTULO 4. NAVEGACIÓN AUTÓNOMA DE UN DRONE GUIADO POR BALIZAS VISUALES

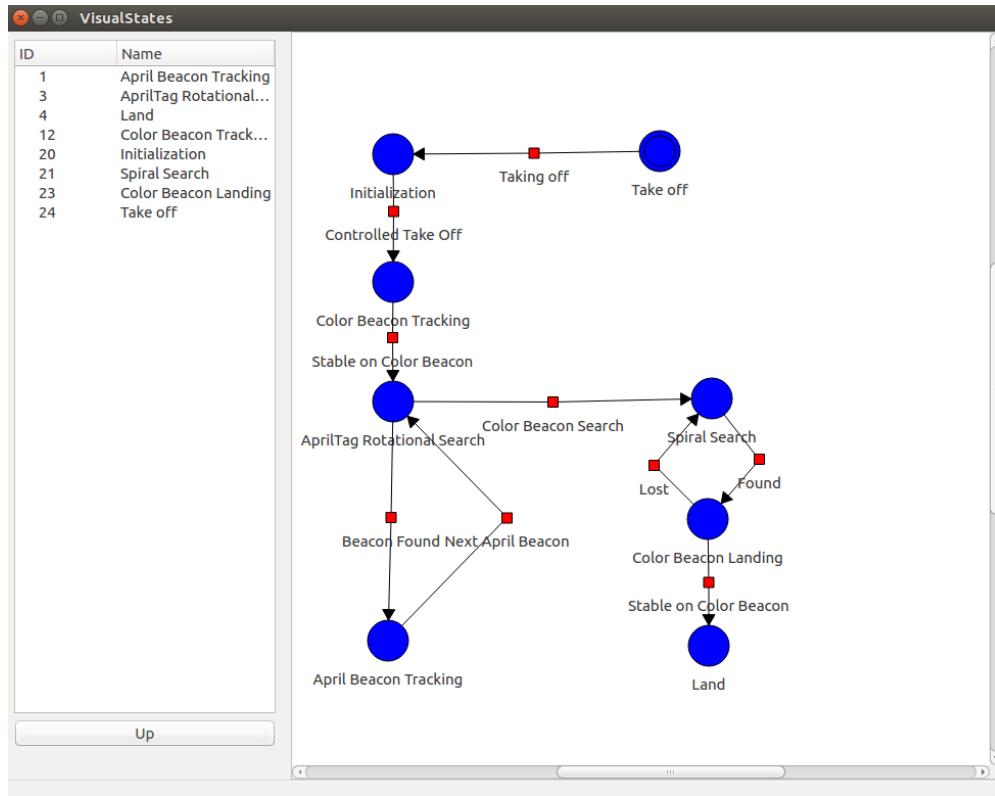


Figura 4.4: Estados de la aplicación 3DPathFollower

En la figura 4.4 se observan los diferentes estados y transiciones que forman la aplicación. El primer estado es el de *Take-off*, en el que se envía un comando al drone para iniciar los motores en modo despegue para ganar altura y despegarse del suelo. Dado que esta fase tiene una duración de unos 2 segundos, el siguiente estado es el de *Initialization* en el que, junto a otras tareas de inicialización, se realiza la lectura del archivo de configuración *calibration.xml* mostrado en la Figura 4.2, del que obtendremos los parámetros para el filtro de color. *Color Beacon Tracking* se encargará de mantener estable y de forma contralada al drone encima de la baliza de color. Transcurridos 4 segundos en el objetivo, damos por estable el despegue y procedemos al estado *AprilTag Rotational Search*, en el que el drone buscará mientras rota sobre sí mismo la baliza AprilTag que actualmente se considera el objetivo. Tras encontrarla pasamos a *April Beacon Tracking*, el cual navegará y se situará a una distancia predeterminada durante

CAPÍTULO 4. NAVEGACIÓN AUTÓNOMA DE UN DRONE GUIADO POR BALIZAS VISUALES

8 segundos, dando por satisfactoria la navegación y marcando como activa la siguiente baliza almacenada en memoria. Una vez finalizada la lista de balizas AprilTags, se activa el estado *Spiral Search*, que realizará un movimiento en espiral con el objetivo de buscar nuestra baliza de aterrizaje. Si ha sido encontrada, entra en acción el estado *Color Beacon Landing*, que se encargará de aterrizar de forma controlada, es decir, alineado encima de la baliza durante segundos. Tanto en *April Beacon Tracking* como en *Color Beacon Landing*, en el momento en el que se pierda el campo de visión con el objetivo, se volverá al estado de búsqueda respectivo. Por último, *Land* envía el comando de aterrizaje al drone y así dar por finalizado el ejercicio.

4.3.1. Estado de despegue controlado

Este componente se basa en el TFG de Jorge Vela [5], el cual se ha refactorizado y acoplado dentro del estado *Color Beacon Tracking* en VISUAL STATES. Este estado se inicia una vez hemos enviado al motor la orden de iniciar el despegue e inicializado los parámetros de configuración para las balizas de color. En este estado se emplea un procesado de la imagen y un PID (Proporcional, Integral y Derivativo) (a diferencia del PD empleado originalmente por Jorge Vela), una detección de bandas muertas y un limitador de velocidad como medidas de control.

Estableceremos como necesaria la condición de que el drone debe despegar encima de una baliza arlequinada 4.3 para comenzar el comportamiento satisfactoriamente. Por lo tanto, debemos asegurarnos de que la cámara ventral del drone esté activa y no la frontal para evitar comportamientos no deseados.

Una vez obtenemos la imagen resultante filtrada, se aplica un procesado para encontrar la cruceta que forman los cuadrantes de la baliza y encontrar así el punto central que será nuestro objetivo.

La incorporación de un PID se debe a que el comportamiento no era lo suficientemente ágil con un PD y se ha refactorizado y reimplementado esta parte. Ésta función de PID se aplicará en todos los algoritmos en los que este tipo de control sea necesario:

#Proporcional

```

vp = kp * error
#Derivada
vd = kd * ((error-pError)/cycle)
#Integral
viModified = ki * (vi+(error*cycle))
#Total
result = vp + vd + viModified
return result , viModified

```

El desarrollo de este PID se ha basado en la siguiente fórmula matemática que lo describe:

$$\begin{aligned}
 PID(t) &= (P(t) + I(t) + D(t)) \\
 P(t) &= K_p e(t); \quad I(t) = K_i \int e(t) dt; \quad D(t) = K_d \frac{de(t)}{dt}
 \end{aligned}$$

4.3.2. Estados de navegación guiada por balizas

Con el drone estable ya en el aire, el siguiente paso es cambiar a la cámara delantera y ejecutar una búsqueda rotacional y aproximación a las diferentes balizas de tipo AprilTags 3.4. La posición de las balizas es desconocida y el orden viene dado por una lista con los identificadores de las balizas predefinida en la sección de constantes globales de VISUAL STATES 4.2.

El estado de búsqueda rotacional iniciará un movimiento de giro en el sentido a las agujas del reloj y continuará girando hasta que encuentre la baliza que tiene como objetivo. Una vez encontrado y mientras que el objetivo no se escape del campo de visión de la cámara, se ejecutará el estado de aproximación. Éste mantendrá la baliza objetivo a una altura y posición relativa centradas y una distancia preconfigurada. Para ello, necesitaremos estimar la posición relativa de la cámara del drone con respecto a la baliza en tres dimensiones.

Una vez tenemos las estimaciones de las coordenadas y nuestra rotación con respecto a la baliza, aplicaremos un control PID, banda muerta y límites de velocidad para

CAPÍTULO 4. NAVEGACIÓN AUTÓNOMA DE UN DRONE GUIADO POR BALIZAS VISUALES

corregir la altura, distancia y rotación con respecto a la baliza. Este sería el código para cada una de estas componentes:

```
error_xy = [ self.interfaces.center[0] - self.interfaces.x,
             self.interfaces.center[1] - self.interfaces.y]

#VX

if (abs(error_xy[0]) < self.interfaces.dead_band_x):
    vx=0
else:
    vx ,vxiMod = self.interfaces.getPIDSpeed(error_xy[0]
                                                ,self.interfaces.error_xy_anterior[0]
                                                ,self.interfaces.vxi
                                                ,self.interfaces.cycle
                                                ,self.interfaces.kp
                                                ,self.interfaces.kd
                                                ,self.interfaces.ki)
    self.interfaces.vxi=vxiMod
vx=self.interfaces.limitSpeed(vx)
```

Para finalizar, enviaremos las velocidades combinadas a los motores para corregir al mismo tiempo las tres componentes y conseguir un comportamiento ágil y fluido. Una vez nos encontramos delante del objetivo durante más de ocho segundos, pasaremos a buscar al siguiente identificador que haya en la lista, hasta que lleguemos al final y pasemos al siguiente algoritmo.

4.3.3. Estados de aterrizaje

Para finalizar, se cambia de nuevo a la cámara ventral para dar paso a los tres últimos estados que componen el aterrizaje: *Spiral Search*, *Color Beacon Landing* y *Land*. El estado *Spiral Search* se corresponde con la búsqueda previa al aterrizaje controlado, para así asegurar que se encontrará de forma autónoma la baliza arlequinada. El drone describirá un movimiento en espiral, ampliando el radio de giro en función del periodo

que establezcamos, hasta que encuentre la baliza:

```
if ( self . interfaces . cycleCounter>self . interfaces . cyclePeriod ):  
    self . interfaces . cycleCounter=0  
    self . interfaces . xSearchSpeed+=self . interfaces . searchIncrement  
xSpeed=-self . interfaces . xSearchSpeed  
wSpeed=self . interfaces . wSearchSpeed  
self . interfaces . cycleCounter+=1
```

En cuanto la baliza sea detectada y mientras no se pierda el contacto visual con la baliza, se cambiará de estado para fijar la baliza en el centro del drone, aplicando las mismas técnicas que en el *Algoritmo de Despegue 4.3.1*. Una vez encontremos la cruceta y permanezcamos durante más de cuatro segundos en el objetivo, se activará el siguiente estado. En último lugar, procederemos a ejecutar el estado de *Land* que enviará la señal al drone de que se debe realizar un aterrizaje. El drone aterrizará y detendrá los motores al llegar al suelo.

4.4. Configuración

A continuación se describirá y se mostrará las configuraciones necesarias para la ejecución correcta de la aplicación.

4.4.1. Configuración de la percepción

El fichero de configuración (*calibration.xml*) que aparece en la Figura 4.1, que la herramienta calibrationTool genera para obtener los valores de las transformaciones y filtros de color que se utilizarán para la percepción de balizas arlequinadas.

```
<data>  
    <colour name="Primary">  
        <Hmax>135</Hmax>  
        <Hmin>97</Hmin>
```

```
<Smax>255</Smax>
<Smin>101</Smin>
<Vmax>255</Vmax>
<Vmin>186</Vmin>
<Erosion>4</Erosion>
<Dilation>4</Dilation>

</colour>
<colour name="Secondary">
    <Hmax>172</Hmax>
    <Hmin>150</Hmin>
    <Smax>199</Smax>
    <Smin>69</Smin>
    <Vmax>255</Vmax>
    <Vmin>0</Vmin>
    <Erosion>8</Erosion>
    <Dilation>8</Dilation>
</colour>
</data>
```

4.4.2. Configuración de interfaces

La Figura 4.5 muestra el fichero de configuración (*config.cfg*) de las interfaces ICE que aparece en la Figura 4.1, que la herramienta genera para realizar las comunicaciones necesarias con el drone.

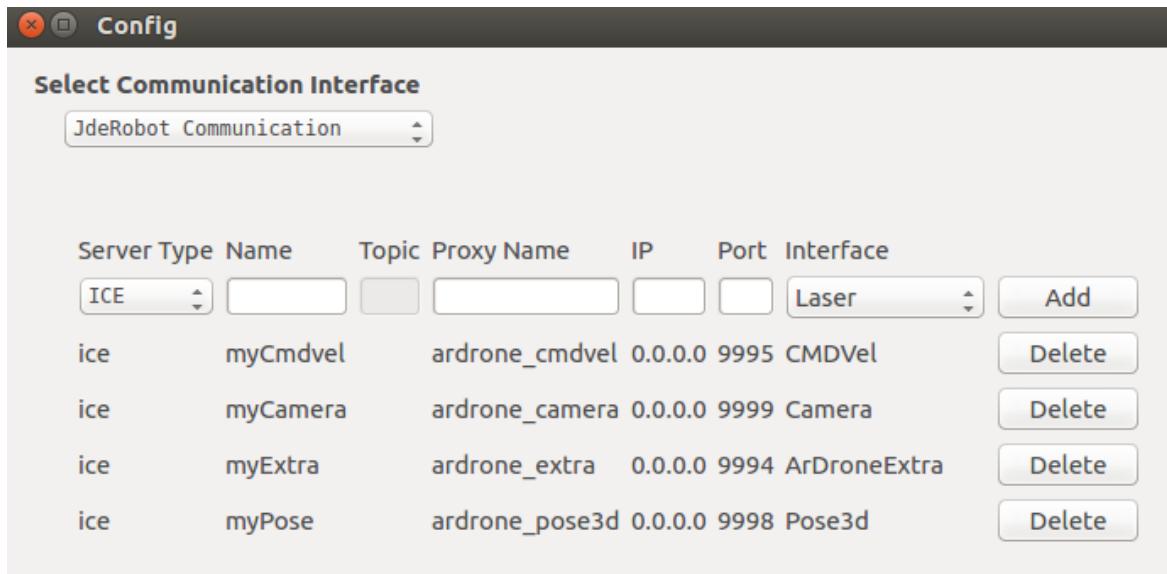


Figura 4.5: Fichero de configuración de interfaces ICE.

4.4.3. Configuración de la secuencia de balizas

Se realizará a través de la herramienta **Visual States**, en concreto, en la sección de variables globales. Se ha creado una lista con el nombre *beacons* que contiene los números de los identificadores de las balizas AprilTags que queramos detectar y navegar:

```
self.beacons = [4, 7, 16, 30]
```

4.5. Herramienta CalibrationTool

Para facilitar la tarea de calibración, nos hemos basado en la aplicación *Color Tuner* 3.5.3 de JdeRobot para aplicar los filtros a la imagen y la hemos enriquecido con un par de funcionalidades. Dado que *3DPATHFOLLOWER* utiliza la suma de dos filtros de color, llamados *Primary* y *Secondary*, la principal funcionalidad añadida de este herramienta es permitir la modificación simultánea de ambos filtros de color de manera que podremos visualizar al mismo tiempo la suma de las dos imágenes filtradas en

CAPÍTULO 4. NAVEGACIÓN AUTÓNOMA DE UN DRONE GUIADO POR BALIZAS VISUALES

la ventana *Filtered_image*. La modificación de estos filtros se realizará a través de deslizadores o *sliders* tal y como se puede apreciar en la figura 4.6.

La segunda funcionalidad añadida es la capacidad de generar o leer de un fichero de configuración en formato XML, que contendrá los diferentes valores máximos y mínimos de los componentes HSV y de las transformaciones morfológicas sobre las imágenes filtradas que explicaremos a continuación. Estas transformaciones son la erosión y dilatación. La combinación de estas técnicas permite la eliminación de impurezas y el suavizado de la imagen, dotando de mayor robustez al filtro de color.

La adquisición se realiza a través de la interfaz de cámara ICE que contiene el fichero de configuración *calib_config.cfg* 4.1. Para poder procesar la imagen primero es necesario verificar si ya existe un archivo de calibración (*calibration.xml*) o crearlo con valores por defecto en caso negativo. Se leerán los datos del fichero de calibración y una vez finalizados los cambios que queramos efectuar, pulsando la tecla *Escape* del teclado cerrará la aplicación y generará o modificará el fichero de configuración.

En cuanto a la interfaz de usuario, para la creación de los *sliders* se ha utilizado la función *createTrackbar* de OpenCV:

```
cv2.createTrackbar('PH_max', 'filtered_image', 0, 180, nothing)
cv2.createTrackbar('PS_max', 'filtered_image', 0, 255, nothing)
cv2.createTrackbar('PV_max', 'filtered_image', 0, 255, nothing)
cv2.createTrackbar('PH_min', 'filtered_image', 0, 255, nothing)
cv2.createTrackbar('PS_min', 'filtered_image', 0, 255, nothing)
cv2.createTrackbar('PV_min', 'filtered_image', 0, 255, nothing)
cv2.createTrackbar('PErode', 'filtered_image', 0, 100, nothing)
cv2.createTrackbar('PDilate', 'filtered_image', 0, 100, nothing)
```

Para mostrar las imágenes en pantalla se ha utilizado la función *imshow* que mostrará la imagen deseada como muestra el siguiente ejemplo:

```
cv2.imshow("filtered_image", filtered_image)
```

CAPÍTULO 4. NAVEGACIÓN AUTÓNOMA DE UN DRONE GUIADO POR
BALIZAS VISUALES

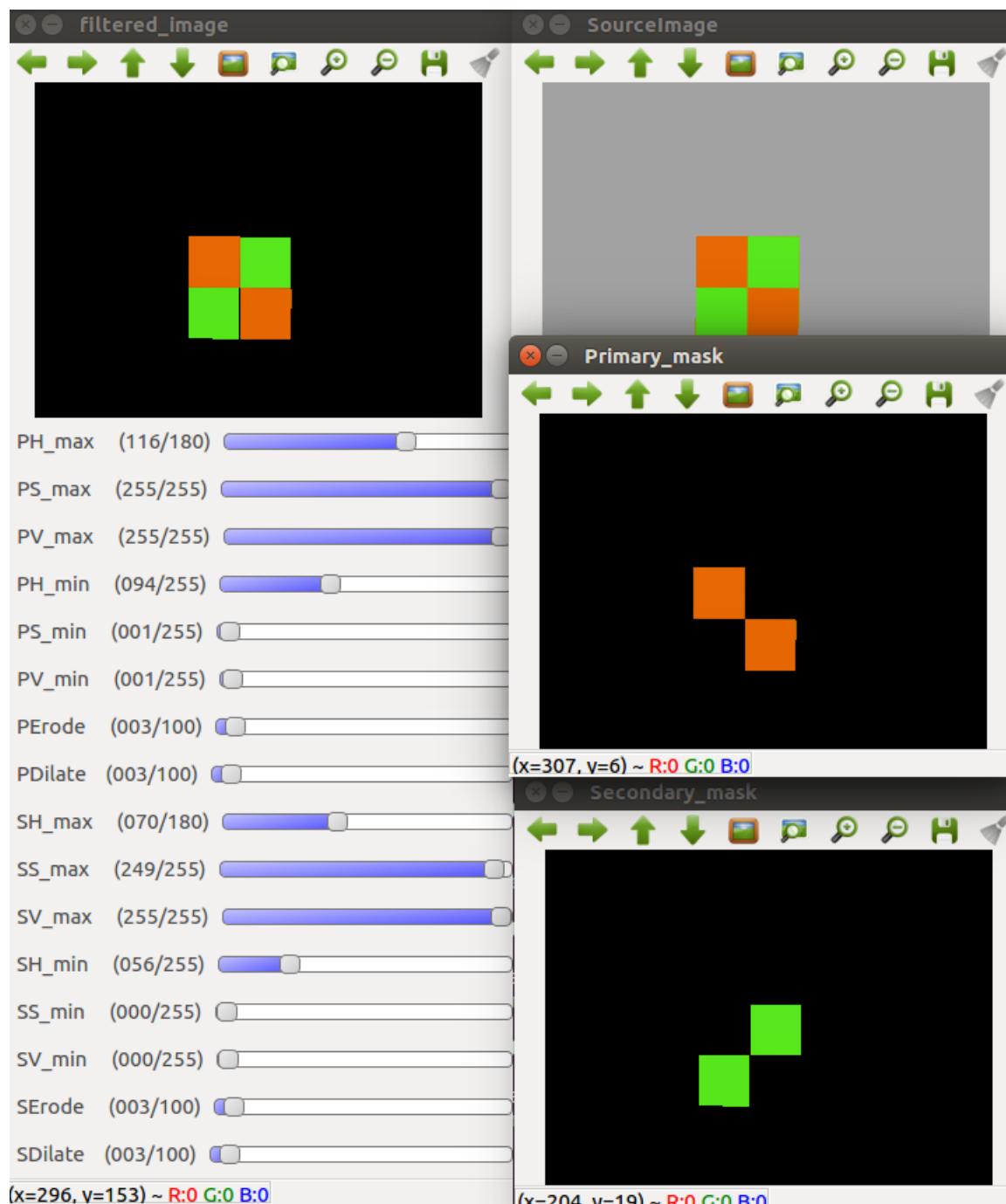


Figura 4.6: Ejemplo del componente CalibrationTool.

Capítulo 5

Experimentos

Este capítulo servirá para validar experimentalmente la solución desarrollada y permitirá comprobar los límites existentes. Los experimentos se han realizado en tres fases: la primera con el drone simulado en Gazebo (estas pruebas además han permitido refinar y ajustar la solución), la segunda con el drone real pero el procesamiento en un ordenador externo en pruebas unitarias que tienen como colofón pruebas globales con el sistema completo y la tercera con el drone real pero el procesamiento en un miniordenador a bordo, añadirá complejidad realizando exactamente las mismas pruebas que en la segunda fase. Las pruebas unitarias se corresponden directamente con los tres grupos de estados descritos en el apartado 4.3: despegue controlado, navegación autónoma guiada por balizas y aterrizaje controlado.

Para la ejecución de los experimentos ha sido necesaria la utilización del aula de robótica situada en el Campus de Fuenlabrada de la Universidad Rey Juan Carlos. Esto implica que todas las pruebas con el dron real tengan el mismo escenario. Los mundos simulados dentro de Gazebo se han diseñado para aproximar el escenario real, excluyendo escenarios abiertos o balizas situadas a distancias lejanas.

Se han desarrollado funciones en el código para obtener rendimiento computacional tanto en tiempo de ejecución como los recursos utilizados durante las pruebas con idea de facilitar la detección y depuración de problemas en el código. También ha sido necesario el ajuste de los valores de la velocidad entregada a los motores y el cálculo

del error para el controlador PID.

5.1. Pruebas en Simulador

El escenario de la Figura 5.1 diseñado en Gazebo fue modificando a lo largo del desarrollo conforme se afianzaban los conocimientos en las diferentes técnicas de visión. Contiene dos tipos de balizas visuales: arlequinadas para poder poner a prueba los algoritmos de despegue y aterrizaje controlado y Apriltags para tener una ruta en la que probar la navegación en 3D.

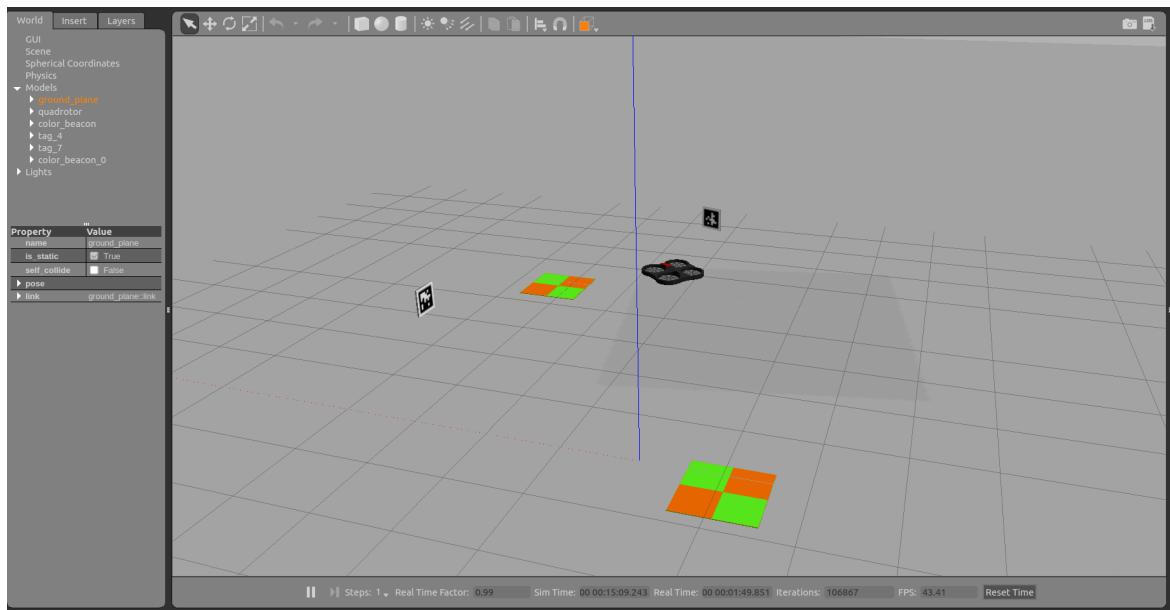


Figura 5.1: Escenario utilizado para las pruebas unitarias en Gazebo.

5.1.1. Elección de balizas de despegue y aterrizaje

Se han seleccionado las balizas de colores arlequinadas, ya que se detectó una posible limitación de las balizas AprilTags a la hora del despegue y aterrizaje. Esta limitación se debe a que si nos acercamos demasiado a las balizas, no entrarán en nuestro campo de visión y por lo tanto AprilTags dejarán de tener utilidad en esas situaciones límite

5.2. Con las balizas arlequinadas se puede mantener la cruceta en pantalla, siempre y cuando se mantenga la baliza en el centro de la imagen. Hay que añadir que cuanto más descendamos, más difícil será mantener la cruceta en el campo de visión.

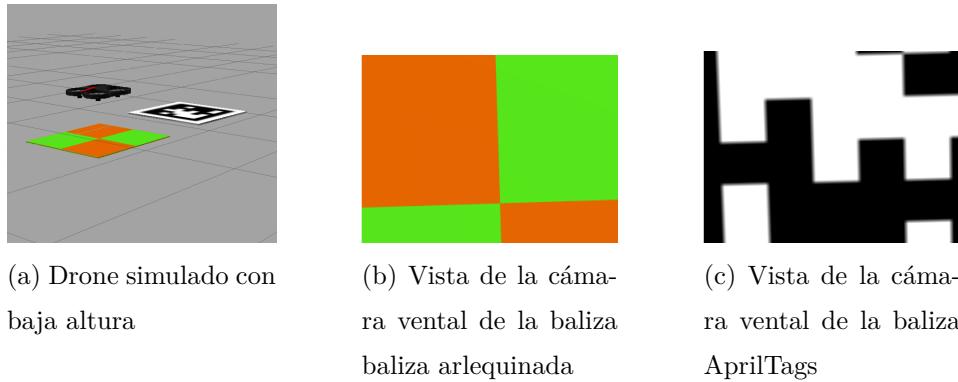


Figura 5.2: Comparación de balizas visuales en Gazebo.

5.1.2. Ajuste del control de navegación

Se han realizado pruebas con las balizas de AprilTags para entender el funcionamiento y los valores devueltos. La tarea más tediosa durante esta fase ha sido la calibración de los coeficientes para el controlador PID de forma manual. Se trata de un proceso de prueba y error en el que se modifican los valores de los coeficientes, cuyo impacto sólo puede verse reflejado durante la ejecución del programa. Las observaciones en estas pruebas reflejan la facilidad con la que podemos prescindir de las componentes ID del controlador PID en un entorno virtual, ya que el ruido es prácticamente nulo y el comportamiento del dron es muy próximo al ideal.

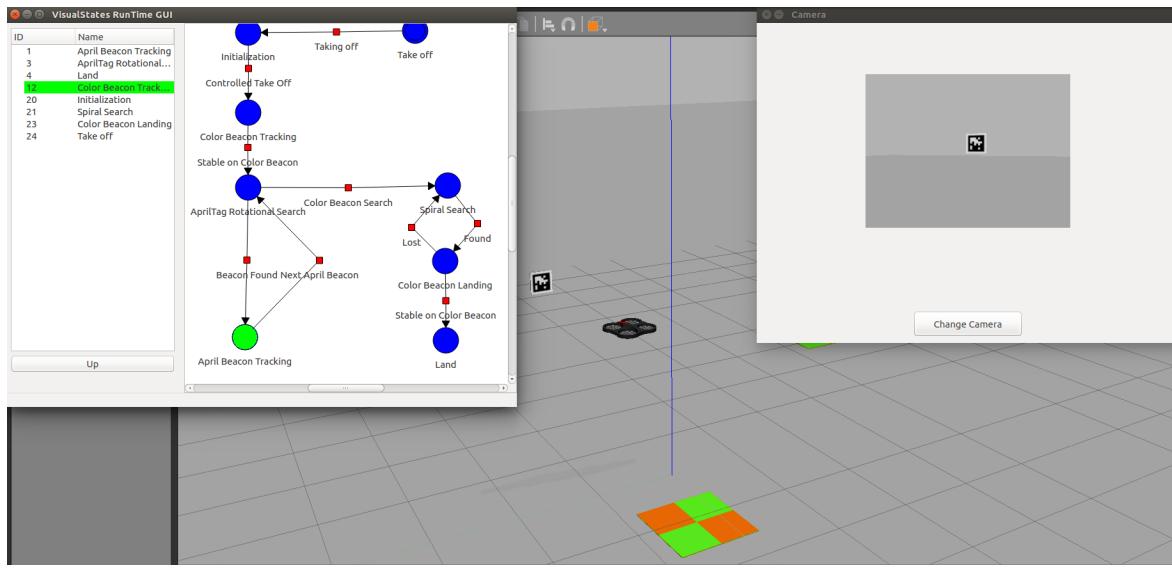


Figura 5.3: Drone simulado navegando en Gazebo.

5.1.3. Ejecución típica integral

Durante las pruebas generales se ha observado que el algoritmo es capaz de correr a una frecuencia máxima de 77 Hz por estado. Esto permite una navegación fluida y ágil, lo que se descartará cualquier fallo debido al rendimiento o velocidad de procesado de imágenes.

El despegue y aterrizaje controlado aplica dos PID, cada uno para controlar las dos dimensiones del plano paralelo a la baliza arlequinada que se encuentra en el suelo del simulador. Debido a que las imágenes de la cámara ventral y la frontal tiene el formato 4:3 y 16:9 respectivamente, los coeficientes correspondientes a la altura de la imagen serán proporcionalmente más pequeños para conseguir un control más preciso.

En el caso de la navegación autónoma en 3D 5.3, se añade complejidad aplicando un PID a una distancia objetivo predefinida, cuyos coeficientes han tenido que ser calibrados independientemente.

Una vez termina la navegación, se activará un estado que buscará la baliza arlequinada y acto seguido iniciará el estado de aterrizaje controlado. Ha sido necesaria

la calibración de la velocidad de búsqueda en espiral para que el PID del aterrizaje consiga controlar su posición encima de la baliza y evitar un bucle de búsqueda- aterrizaje continuo.

5.2. Pruebas con el drone real y PC externo

En este escenario el principal objetivo era la familiarización con el drone real y la adaptación del código a las fluctuaciones y diferencias con el simulador.

5.2.1. Ajuste del control de navegación

La primera observación fue que los coeficientes del controlador PID no aplican directamente al drone real, dotando de un comportamiento demasiado violento y nada controlado. La calibración de los coeficientes se presentaba mucho más complicada que en el simulador, por lo que para comprender mejor el funcionamiento del drone real, se modificó el componente `uav_viewer.py` 3.5.2 para poder teleoperar con las teclas del teclado el dron. A raíz de esto se pudo comprobar el efecto de las velocidades en el dron y se modificó el código para limitar la velocidad máxima al mismo tiempo que calibrar los coeficientes en los casos en que se saturase la velocidad.

5.2.2. Desfase entre imágenes y órdenes

Durante las pruebas se pudo observar un desfase entre las órdenes y las imágenes (provocado por `slam_markers` 3.5.4).

Se pudo dar solución a al desfase sustituyendo `slam_markers` por un módulo integrado la parte de percepción del estado de navegación. Se caracterizó la diferencia entre el rendimiento de la nueva aplicación, ejecutando a una media de 33 Hz por ciclo a diferencia de `slam_markers`, cuyo rendimiento oscilaba entre los 3 y 4 Hz (insuficientes para realizar el control).

5.2.3. Problemas con la detección de las balizas arlequinadas

Una tarea que retrasaba la ejecución de los experimentos estaba provocada por el efecto que tenía la variación de luz en la calibración de las balizas arlequinadas. La variación de luz fue mitigada desarrollando la herramienta CalibrationTool 4.5. Estas dos soluciones permitieron realizar con éxito las pruebas unitarias que fueron grabadas como prueba.



(a)



(b)



(c)

Figura 5.4: Secuencia de despegue.

5.2.4. Ejecución típica integral

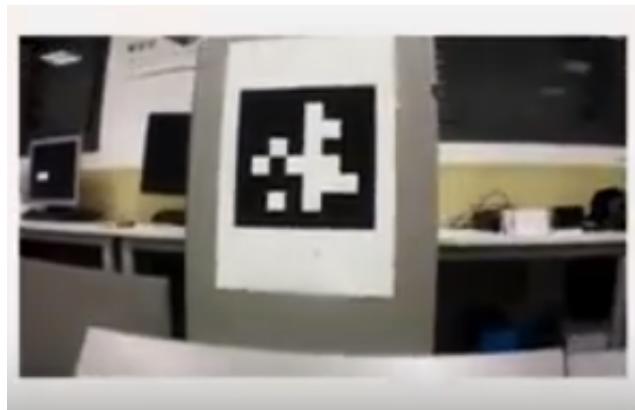
Se ha conseguido realizar de forma satisfactoria diferentes ejecuciones en las que se logrado despegar y aterrizar de forma controlada en una única baliza arlequinada y navegar a las cuatro balizas de AprilTags que se han desplegado a lo largo del laboratorio 5.5.



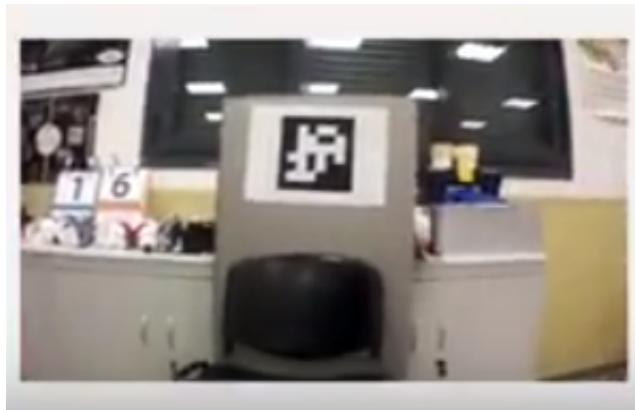
(a) Baliza arlequinada



(b) Baliza AprilTags 4



(c) Baliza AprilTags 7



(d) Baliza AprilTags 16



(e) Baliza AprilTags 30

Figura 5.5: Balizas desplegadas en el laboratorio.

Se encontraron problemas menores relacionados con la deriva del robot, en concreto durante el estado de búsqueda rotacional, en la que el movimiento que describe es realmente un toroide, desplazándose continuamente mientras gira. El aumento de la velocidad rotacional disminuyó el problema, pudiendo grabar un vídeo como validación de la prueba.



(a)



(b)



(c)



(d)

Figura 5.6: Secuencia de navegacion.

En comparación a la ejecución del simulador, la frecuencia máxima a la que se ha podido ejecutar la aplicación ha sido 33 fotogramas por segundo. Este descenso se debe principalmente a que la resolución de las imágenes del simulación son mucho más pequeñas comparadas con las del drone real (640x480 píxeles para el simulador y 1280x720 píxeles para el drone real).

5.3. Pruebas con el dron real y un miniordenador

En este escenario el principal objetivo era comprobar los límites físicos del drone real y probar la posibilidad de crear un sistema totalmente autónomo con el Intel Stick Compute a bordo.

5.3.1. Preparación y configuración del miniordenador

Se ha instalado la versión de Ubuntu 16.04 para ser compatible con los requisitos anteriormente citados. Es necesario añadir que se ha instalado JdeRobot en su versión más reciente y que se ha configurado un servidor SSH para realizar las comunicaciones con el dispositivo cuando esté operando y no esté conectado el HDMI a un monitor externo. Este dispositivo irá acoplado mediante cinta adhesiva.



Figura 5.7: Intel Compute Stick acoplado en Ar.Drone 2.

Además de haber instalado Ubuntu en la versión 16.04, junto a todas las dependencias y bibliotecas necesarias para correr la aplicación, se ha configurado el asistente de la red Wi-Fi para que se conecte de manera automática cuando detecte que la red que genera el drone. Se configuró la IP como estática para poder acceder de forma remota y conocer la dirección exacta de nuestra unidad de computación externa.

Por último, se ha configurado un servidor utilizando la tecnología OpenSSH ?? de manera que se inicie durante el lanzamiento del Sistema Operativo y que escuche en el puerto 22998. Esto aportará una capa de seguridad, actualmente no existente además de habilitar el acceso remoto.

5.3.2. Teleoperación con miniordenador a bordo

El primer paso sería realizar una prueba teleoperando la infraestructura 5.7 desde nuestra herramienta `uav_viewer.py` modificado para observar cualquier cambio en el comportamiento. Lamentablemente el drone se volvía completamente inestable: desde el despegue, pasando por movimientos erráticos y no controlados, independientemente del envío de órdenes o no al dron. La limitación de potencia por parte del dron es la culpable, por lo que la única solución posible es la de realizar las pruebas sin montar a bordo el co-procesador.

5.3.3. Retardo detectado en `uav_viewer.py`

La ejecución de las pruebas unitarias reflejaba un comportamiento muy oscilante debido posiblemente a algún tipo de retardo, por lo que las primeras sospechas apuntaban a la diferencia de rendimiento entre el Intel Compute Stick y el ordenador externo de la sección anterior. Se caracterizó este retardo, mostrando valores de hasta más de un segundo en algunas de las iteraciones de los estados. Para descartar posibles problemas de rendimiento, se analizó el periodo de ejecución del estado, el consumo de recursos como el procesador, la memoria RAM y el ancho de banda utilizado. Sin embargo, los resultados de estas pruebas mostraban una frecuencia mínima de 20 Hz por ejecución, chocando con los valores previamente obtenidos.

Una investigación mas profunda llevó a la conclusión de que la herramienta `uav_viewer.py` que se utilizó para obtener las imágenes del drone en remoto, provocaba una saturación en el canal Wi-Fi, por lo que la solución se basó en la omisión de esta aplicación durante las pruebas con el Intel Compute Stick.

5.3.4. Ejecución típica con el miniordenador en tierra

Para iniciar la aplicación, necesitaremos acceder remotamente mediante el siguiente comando:

```
ssh 192.168.1.3 -p 22988
```

Una vez dentro, lanzaremos la aplicación de manera que la ejecución se realice en el *Intel Compute Stick*.

Para prevenir cualquier accidente, se ha desarrollado un programa muy sencillo, cuya única misión es la de enviar la señal de aterrizaje al drone. Este comando evita la ejecución de futuras órdenes y asegura que el aterrizaje se realizará en el menor tiempo posible.

Se pudieron ejecutar sin problemas las pruebas unitarias, dando paso a la realización de la última prueba global o ensayo general. Este fue gratamente satisfactorio, a pesar de presentar un comportamiento ligeramente inferior en cuanto a precisión y agilidad comparado con las pruebas de la fase anterior, oscilando en mayor medida.

Capítulo 6

Conclusiones

En este capítulo analizaremos si los objetivos 2.1 planteados se han cumplido y se recapitulan las soluciones desarrolladas y los resultados obtenidos. Adicionalmente, sugeriremos las posibles líneas futuras de investigación por las que se puede extender este TFG. En líneas generales el objetivo global se ha alcanzado satisfactoriamente. Se ha conseguido por primera vez en el proyecto JdeRobot la navegación completamente autónoma en un entorno 3D utilizando un drone real, además de abrir la puerta al uso de miniordenadores embarcados a bordo de drones que tengan motores más potentes que los del Ar.Drone 2 de Parrot.

6.1. Conclusiones

Gran parte del trabajo aquí expuesto no ha formado parte de la programación directamente, sino que incluye el tiempo de investigación, aprendizaje de herramientas, tecnologías y habilidades que han tenido que ser adquiridas o adaptadas para programar con éxito la solución. Otra parte que se encuentra oculta detrás de los programas es el mérito de haber realizado los experimentos en un drone real cuyo comportamiento es inestable, en el que el ruido de los sensores condiciona el resultado de las pruebas y en el que la duración de las baterías reduce el número de pruebas diarias que se pueden realizar.

A continuación analizamos todos los subobjetivos para extraer las conclusiones que hemos obtenido en cada uno de ellos, recapitulando a la vez cómo han sido conseguidos:

1. *Integración del módulo de autolocalización 3D a partir de balizas visuales:* El funcionamiento ha sido validado mediante pruebas unitarias. Se ha reimplementado en Python el algoritmo que utilizaba `slam_markers` y se ha integrado dentro del nodo `3DPathFollower`. Este módulo es capaz de estimar las coordenadas relativas con respecto a una baliza visual correctamente, permitiendo a la aplicación final realizar las correcciones necesarias en tiempo real y con la precisión necesaria.
2. *Desarrollo e integración de un módulo de navegación por balizas visuales:* Se ha desarrollado e integrado un PID capaz de ser aplicado para las diferentes balizas visuales utilizadas, aprovechando los recursos y mejorando el rendimiento de nuestra aplicación. Además, se ha ajustado el control del pilotaje tanto para escenarios simulados como para el drone real.
3. *Programación del comportamiento del dron en un autómata de estados finito:* Se han integrado la autolocalización, los controladores de aterrizaje, de despegue y de navegación siguiendo balizas visuales en un único autómata compacto, basado en 8 estados conectados a través de transiciones. Esto ha sido posible con la ayuda de `VISUAL STATES`.
4. *Validación experimental en el cuadricóptero real:* Se han realizado pruebas tanto en el cuadricóptero simulado en Gazebo como principalmente con el cuadricóptero real Ar.Drone 2 de Parrot. Estas pruebas han sido tanto unitarias como globales, mostrando las capacidades de la navegación autónoma en 3D utilizando por primera vez balizas visuales. El comportamiento es ágil y preciso, además de eficiente ya que el algoritmo es capaz de ejecutar a una frecuencia de 33 Hz.

Los resultados obtenidos con miniordenador a bordo se han visto limitados debido a las restricciones de potencia del dron, que imposibilitan una prueba completamente a bordo. Aun así, se ha conseguido validar experimentalmente las

capacidades de procesamiento del Intel Compute Stick 3.2 de forma externa, dando como resultado un comportamiento ágil, suficientemente preciso, y con una frecuencia de ejecución de 20Hz, no muy diferente de la de un computador. De esta manera se deja abierta la puerta a posibles futuras aplicaciones con comportamientos autónomos totalmente a bordo. Se ha validado experimentalmente a través de la prueba global, en la que se ha ejecutado de principio a fin el algoritmo sin intervención externa de un humano, completando todos los estados y finalizando el ejercicio con el aterrizaje y detención del drone en la posición designada.

Se ha aportado una nueva herramienta a la plataforma de JdeRobot, que ahora cuenta con un ejemplo de probado en un dron real navegación de forma autónoma, desde el despegue hasta el aterrizaje, realizando desplazamientos controlados.

Todo el material audiovisual y avances que han ido teniendo lugar se pueden consultar en la Wiki oficial del proyecto: <http://jderobot.org/Andresjhe-tfg>

El código está subido al repositorio oficial del TFG y puede ser accedido sin restricciones para su revisión y mejora: <https://github.com/RoboticsURJC-students/2014-tfg-Andres-Hernandez/>

6.2. Trabajos futuros

Con este proyecto se han abierto algunas fronteras inexploradas lo que facilita el desarrollo de nuevas aplicaciones o la mejora de las ya existentes, en concreto en el ámbito de la navegación autónoma y la utilización de un miniordenador a bordo. A continuación se proponen algunas posibles líneas futuras a partir de los resultados y la experiencia obtenida durante la realización de este TFG:

- Aumentar la complejidad de la navegación o probar en un escenario de exteriores. Ambas vías están directamente relacionadas con la mejora de los algoritmos desarrollados.

- Sustitución de los algoritmos de visión por otros más sofisticados como técnicas de SLAM para autolocalización o el uso de *DeepLearning*. Aumentaría la robustez y la cantidad de escenarios en los que puede ser utilizados, a pesar de que se sacrificaría eficiencia computacional y en latencia.
- Sustitución del dron existente. La mayor dificultad a la hora de cumplir todos los objetivos ha estado directamente relacionada con las limitaciones del dron actual (derivas, falta de empuje vertical, ruido o defectos en sensores, etcétera). Un nuevo dron más potente puede ser la solución a los problemas encontrados.

Bibliografía

- [1] ALBERTO MARTÍN FLORIDO. Navegación visual en un cuadricóptero para el seguimiento de objetos. *Trabajo Fin de Grado*, URJC, 2014.
- [2] DANIEL YAGÜE SÁNCHEZ. Cuadricóptero AR.Drone en Gazebo y JdeRobot. *Proyecto Fin de Carrera*, URJC, 2014.
- [3] ALBERTO LÓPEZ-CERÓN PINILLA. Autolocalización visual robusta basada en marcadores. *Trabajo Fin de Master*, URJC, 2015.
- [4] MANUEL ZAFRA VILLAR. Seguimiento de rutas 3D por un drone con autolocalización visual con balizas. *Trabajo Fin de Grado* URJC, 2016.
- [5] JORGE VELA PEÑA. Despegue, navegación y aterrizaje visuales de un drone usando JdeRobot. *Trabajo Fin de Grado* URJC, 2017.
- [6] JESÚS SAIZ COLOMINA. Programación de un dron para seguimiento autónomo de trayectorias en 3D. *Trabajo Fin de Grado* URJC, 2018.