



Grado en Ingeniería de Computadores

Curso académico 2013/2014

Trabajo Fin de Grado

Navegación visual en un cuadricóptero para el seguimiento de objetos

Autor:

Alberto Martín Florido

Tutores:

José María Cañas Plaza y Carlos Agüero Durán

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

<http://jderobot.org/Amartinflorido-tfg>



(c) 2014 Alberto Martín Florido

Esta obra está bajo una licencia Reconocimiento-Compartir bajo la misma licencia 3.0 España de Creative Commons.

Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-sa/3.0/es/> o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

Agradecimientos

No podría existir este documento sin esta página de agradecimientos y es que son muchas las personas a las que debo agradecimiento y mi más absoluto respeto. No solo a aquellas que han contribuido a la realización de este proyecto sino también a aquellas que me han convertido en lo que soy.

Quisiera comenzar agradeciendo a mis tutores, José María Cañas y Carlos Agüero Durán, su infinita paciencia, su gran dedicación y, como no podía ser de otra manera, su calidad humana. Gracias.

Gracias a todas las personas que de algún modo han contribuido con su granito de arena a JdeRobot por poner a disposición de cualquiera tantas y tantas líneas de conocimiento.

Quisiera también agradecer a mis padres el tremendo sacrificio que han hecho durante toda su vida por mis hermanos y por mí. Gracias por darme todo lo que vosotros nunca tuvisteis.

A las dos mujeres de mi vida, Arantxa y Ainhoa, Gracias por estar siempre ahí. Con vosotras todo es mucho más fácil.

Resumen

En los últimos años los robots aéreos se han convertido en una disciplina de moda dentro del mundo de la robótica. Hoy en día vemos *drones* que realizan multitud de tareas e incluso se pueden adquirir a bajo precio. Aunque el principal impulso en este campo fueron las posibilidades militares de estos vehículos, son cada día más comunes las aplicaciones civiles de los Vehículos Aéreos no Tripulados (VANT). El VANT usado en este Trabajo Fin de Grado es el cuadricóptero Ar.Drone de Parrot. Los cuadricópteros pueden realizar multitud de maniobras, gracias a su diseño poseen una agilidad nunca antes vista en un vehículo aéreo. Además, con la inclusión de sensores y sistemas embebidos a bordo de estos ingenios aeronáuticos se abrió la puerta a la automatización de tareas para aprovechar el tremendo potencial de estas máquinas.

Este Trabajo Fin de Grado aborda la navegación visual autónoma de un cuadricóptero, para ello se diseñó e implementaron dos algoritmos de navegación visual para el seguimiento de objetos de manera autónoma. Haciendo uso de la cámara frontal de Ar.Drone, el primer algoritmo comanda las órdenes necesarias para que el cuadricóptero sea capaz de seguir un objeto en el espacio 3D. El segundo algoritmo realiza un seguimiento de objetos que se encuentren debajo del *drone*, para ello utiliza las imágenes capturadas con su cámara ventral y ordena al cuadricóptero las maniobras que debe de llevar a cabo para seguir el objeto. Para implementar todo ello fue necesario desarrollar la infraestructura software que nos permitiera el control y acceso a los sensores del cuadricóptero Ar.Drone desarrollando un *driver* y una herramienta para su teleoperación y visualización de los datos sensoriales. Gracias al componente *driver*, integrado dentro de la plataforma JdeRobot, es posible utilizar este *drone* manejando un API abstracto y sencillo liberando así al desarrollador la complejidad intrínseca de trabajar con dispositivos hardware a bajo nivel.

Las soluciones propuestas en los algoritmos de navegación visual han sido validadas experimentalmente demostrando su correcto funcionamiento. Todo el desarrollo de este trabajo se ha realizado sobre la plataforma JdeRobot 5.1, la implementación de los componentes se realizó con el lenguaje de programación C++, para el tratamiento de imágenes por computador se utilizó la librería OpenCV, la comunicación entre los distintos componentes se llevó a cabo con ICE y las interfaces gráficas de usuario se desarrollaron con las librerías gráficas Qt y Qwt.

Índice general

1. Introducción	1
1.1. Robótica	1
1.1.1. Componentes de un robot	4
1.1.2. El software de los robots	5
1.1.3. Visión artificial en Robótica	5
Fundamentos	6
1.2. Vehículos aéreos no tripulados	7
1.2.1. Clasificación de los VANT	9
1.2.2. Cuadricópteros	10
Principios de Sustentación y Movimiento	10
1.3. Robótica aérea	13
1.3.1. Robótica aérea en el mundo	14
Investigación	14
Usos comerciales	15
Competiciones	16
1.3.2. Robótica aérea en España	18
Investigación	18
Usos comerciales	20
1.3.3. Robótica aérea en la Universidad Rey Juan Carlos	22
2. Objetivos	24
2.1. Descripción del problema	24
2.2. Requisitos	24
2.3. Metodología de desarrollo	25
2.4. Plan de trabajo	26

3. Infraestructura	28
3.1. Parrot Ar.Drone 1.0: hardware y software	28
3.1.1. Sensores	29
Sensores inerciales	29
Cámara frontal	29
Cámara vertical	29
Altímetro basado en ultrasonidos	29
3.1.2. Comunicación con Ar.Drone	30
Comandos AT	30
3.1.3. Sistema de control de bajo nivel a bordo	31
3.1.4. Sistema de vídeo a bordo	32
Detección de etiquetas	33
3.2. Ar.Drone SDK	34
3.2.1. Ar.Drone Library	35
3.2.2. Ar.Drone Control Engine	36
3.3. ROS y los paquetes de Ar.Drone	36
3.3.1. <code>ardrone_brown</code>	37
3.3.2. <code>ardrone_autonomy</code>	38
3.4. JdeRobot	39
3.5. Bibliotecas externas	40
3.5.1. <code>libxml++</code>	40
3.5.2. <code>ICE</code>	40
3.5.3. <code>OpenCV</code>	41
3.5.4. <code>Qt</code>	41
3.5.5. <code>Qwt</code>	42
4. Desarrollo de la infraestructura para Ar.Drone	43
4.1. Componente <i>driver</i> : <code>ardrone_server</code>	43
4.1.1. Interfaces ICE, entradas y salidas	44
<code>camera</code>	44
<code>ardrone_control</code>	47
<code>remoteConfig</code>	49
4.1.2. Diseño	49
4.1.3. Gestión de configuración	53
4.2. Herramienta: <code>uav_viewer</code>	54
4.2.1. Diseño	56

5. Navegación visual	60
5.1. Diseño de la navegación visual siguiendo objetos	60
5.2. Percepción visual: Detección de objeto en imagen	62
5.2.1. De RGB a HSV	62
5.2.2. Suavizado de la imagen	64
5.2.3. Filtro de color del objeto	64
5.2.4. Segmentación: extracción de coordenadas y área	65
5.2.5. Filtro paso bajo	66
5.3. Seguimiento frontal basado en el Control PID	67
5.4. Seguimiento ventral basado en una Superficie de Control	69
5.4.1. Control Sólo de Posición	70
5.4.2. Control Sólo de Orientación	72
5.5. Interfaz Gráfica de Usuario	75
6. Validación experimental	79
6.1. Filtro de color	79
6.2. Filtro temporal en estimación de posición	80
6.3. Seguimiento frontal	82
6.4. Seguimiento ventral	83
6.4.1. Controlador único con superficie de control	84
6.4.2. Control de la deriva de Ar.Drone	84
6.5. Alternativas probadas para el control ventral	86
6.5.1. Controlador PID único	86
6.5.2. Dos controladores PID priorizando la orientación	87
6.5.3. Dos controladores PID priorizando la posición	87
7. Conclusiones	89
7.1. Conclusiones	89
7.2. Trabajos futuros	93
Bibliografía	95

Índice de figuras

1.1. El ajedrecista	2
1.2. Historia de la robótica	2
1.3. Robots móviles	3
1.4. Automóvil robótico y robot aspiradora	3
1.5. Robots industriales	4
1.6. Visión artificial en la robótica	6
1.7. Espectro visible	7
1.8. AQM-34 Ryan Firebee	8
1.9. VANT desarrollados en Israel	8
1.10. VANT Ikhana.	9
1.11. Imagen captada por uno de los sensores incluidos en el VANT de investigación Ikhana de la NASA, sobre Harris Fire en San Diego.	9
1.12. VANT Orbiter de tipo Handheld	10
1.13. Estación de tierra para el Orbiter	10
1.14. Boeing V-22 Osprey	10
1.15. Cuadricóptero Pelican	10
1.16. Compensación del par motor	11
1.17. Configuración de los rotores	12
1.18. Roll, Pitch y Yaw	12
1.19. VANT construyendo una torre	14
1.20. Sherpa Box	15
1.21. Aplicaciones comerciales	15
1.22. VANT inspeccionando un edificio en IARC	17
1.23. VANT españoles	18
1.24. VANT españoles	18
1.25. Montaje de estructura en incendio con ARCAS	19

1.26. Seguimiento de un vehículo	20
1.27. Aplicaciones de Aerotools-UAV	20
1.28. Sistema de navegación autónoma	21
1.29. Ministro de Interior junto a una maqueta de un VANT	21
1.30. Estación base desarrollada por Óscar Higuera	22
1.31. Robótica aérea en la URJC	23
2.1. Desarrollo en espiral	25
3.1. Ar.Drone 1.0	28
3.2. Comunicación entre Ar.Drone y PC	30
3.3. Canal horizontal	32
3.4. Canal vertical	32
3.5. Canal vertical sobre frontal	32
3.6. Canal frontal sobre vertical	32
3.7. Imagen dividida en bloques	32
3.8. Macrobloques de 16x16	33
3.9. Espacio de color $YC_B C_R$	33
3.10. Almacenamiento de la imagen en memoria	33
3.11. Carcasa de exterior y etiqueta identificativa	34
3.13. Arquitectura de Ar.Drone SDK	34
3.14. Aplicación oficial de Ar.Drone para iOS	36
4.1. Esquema <code>ardrone_server</code>	43
4.2. Diagrama de <code>ardrone_server</code>	50
4.3. Esquemas <code>ardrone_server</code>	51
4.4. <code>uav_viewer</code>	55
4.5. Diagrama de clases de <code>uav_viewer</code>	56
4.6. Esquema <code>uav_viewer</code>	57
4.7. Hilos de <code>uav_viewer</code>	57
4.8. Teleoperación de Ar.Drone con <code>uav_viewer</code>	58
5.1. Esquema de <code>object_tracking</code>	60
5.3. Diagrama de clases de <code>object_tracking</code>	61
5.2. Hilos de <code>object_tracking</code>	61
5.4. <i>Pipeline</i> del sistema de percepción de <code>object_tracking</code>	62

5.5. Cono de colores del espacio HSV	63
5.6. <i>Gaussian Blur</i>	64
5.7. Función de Gauss en 2 dimensiones	64
5.8. Umbralización de la imagen	65
5.9. Contornos detectados con la función <i>findContours()</i>	65
5.10. Sistemas de referencia	66
5.11. Filtro paso bajo	67
5.12. Control de posición	67
5.13. Correspondencia de los ejes	68
5.14. Cálculo de errores	68
5.15. Controlador PID genérico	68
5.16. Ecuación PID	68
5.17. Colores para el seguimiento ventral	69
5.18. Sistema de actuación	70
5.19. Control de posición	71
5.20. División de la imagen en celdas	71
5.21. Curvas de control	72
5.22. Control de orientación	73
5.23. Media Gaussiana	73
5.24. Superficie de control para la orientación	74
5.25. Histéresis	75
5.26. Pestaña <i>tracking</i> de <i>object_tracking</i>	76
5.27. Pestaña <i>configure filter</i> de <i>object_tracking</i>	77
5.28. Pestaña <i>configure Ar.Drone</i> de <i>object_tracking</i>	78
6.1. Valores registrados en la comparación de ambos filtros	79
6.2. Comparación entre HSV y RGB	80
6.3. Pruebas sobre la detección de objetos	81
6.4. Comparación de la detección de la posición	81
6.5. Seguimiento frontal con controladores PID	83
6.6. Seguimiento ventral con superficie de control	84
6.7. Pruebas sobre la deriva de Ar.Drone	85
6.8. Error en píxeles del modo cernido	86
6.9. Histéresis	88

Capítulo 1. Introducción

En este primer capítulo, el lector podrá conocer de una forma breve qué es la Robótica y los vehículos aéreos no tripulados. A continuación, se revisará el estado actual de la robótica aérea en el mundo, desde sus aplicaciones en la investigación hasta los usos comerciales de esta tecnología. Una vez conocido el contexto internacional, el lector podrá conocer el estado de la robótica aérea en España y más concretamente en la Universidad Rey Juan Carlos.

1.1. Robótica

La robótica es la disciplina encargada del diseño, fabricación, operación y aplicación de los robots. La robótica intersecta con distintos campos como la mecánica, la electrónica, la informática, la inteligencia artificial, la ingeniería de control y la física. La palabra robot proviene del término checo *robota* (que significa trabajo forzado), utilizada por el escritor checoslovaco Karel Capek. En su novela Rossum's Universal Robots de 1923, el autor, hace uso del término *robota* para referirse a ingenios mecánicos capaces de realizar instrucciones dadas por su creador.

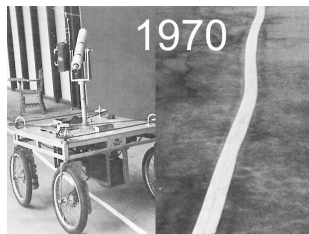
Aunque el término robot fue acuñado en los años 20, estos ingenios mecánicos ya era conocidos en el mundo como autómatas. Los autómatas son los precursores de los robots que conocemos actualmente. Según la Real Academia Española, el término autómata se define como una *máquina que imita la figura y los movimientos de un ser animado*. En la historia podemos encontrar varios ejemplos de autómatas, como el robot de Leonardo, que fue un autómata humanoide diseñado por Leonardo Da Vinci en el año 1495, al igual que muchos otros de sus inventos no llegó a ser construido. El ajedrecista fue un autómata construido por el ingeniero y matemático Leonardo Torres Quevedo en el año 1912. Fue presentado durante la Feria de París de 1914 y en 1915 la revista Scientific American publicó el artículo “*Torres and His Remarkable Automatic Device*”. Utilizando unos electroimanes bajo el tablero de ajedrez, jugaba un final de rey y torre contra el rey de un oponente humano. Con un simple algoritmo que evaluaba las posiciones de las figuras, no siempre llegaba al mate en el número mínimo de movimientos, pero sí que fue capaz de lograr la victoria en todas las ocasiones sin intervención humana.



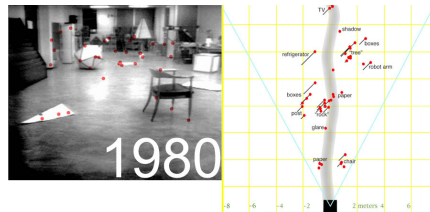
Figura 1.1: El ajedrecista

Con la llegada de los ordenadores en los años 70 la universidad de Stanford desarrolló el *Stanford Cart* (ver Figura 1.2a) que se convirtió en el primer robot móvil controlado por un ordenador. Este robot era capaz de seguir una línea de color blanco usando la cámara que tenía incorporada. Con los datos recolectados por su cámara se comunicaba a través de ondas de radio con una estación de trabajo, que realizaba los cálculos necesarios para su movimiento. Otro ejemplo de esta época es el robot *Shakey* que era capaz de razonar sobre sus propias acciones. A diferencia del resto de robots, que tenían que recibir instrucciones a cada paso para completar una tarea, *Shakey* podía recibir la tarea que tenía que realizar y llevarla a cabo en pasos más pequeños por sí mismo.

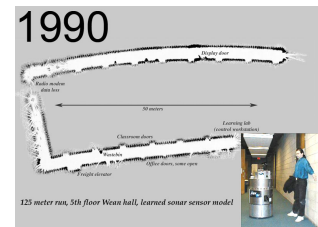
Ya en los 80, con la posibilidad de embarcar ordenadores en los propios robots, la versión mejorada del *Stanford Cart* contaba con dos cámaras de TV que le proporcionaban una visión estéreo del entorno. Con una reconstrucción 3D limitada era capaz de reconocer objetos regulares y navegar por su entorno evitando los obstáculos. En el experimento más famoso que se realizó con este robot, fue capaz de sortear distintos obstáculos durante 30 metros en 5 horas.



(a) Stanford Cart



(b) Stanford Cart mejorado



(c) Robot Xavier

Figura 1.2: Historia de la robótica

En los años siguientes al *Stanford Cart* la investigación se fue centrando en la construcción de mapas y la navegación usándolos con robots como Xavier. Este robot fue desarrollado por la universidad Carnegie Mellon en 1993, su propósito era desarrollar un robot móvil capaz de operar de manera fiable durante largos periodos de tiempo, aprendiendo y adaptándose a su entorno. El resultado fue Xavier y la posibilidad de que un robot navegara de forma autónoma en un entorno generando un mapa a medida que se desplazaba. Desde el año 2000 hasta la actualidad la robótica ha protagonizado grandes avances, debido sobre todo, al avance de la tecnología y el abaratamiento de ésta. La robótica ha sido capaz de conquistar diversos terrenos como la tierra, el mar, el aire, el espacio y la superficies de otros planetas. Se han creado coches autónomos (ver Figura 1.3a), humanoides altamente sofisticados (ver Figura 1.3b), vehículos de exploración espacial (ver Figura 1.3c), robots submarinos (ver Figura 1.3d) y aéreos (ver Figura 1.3e).

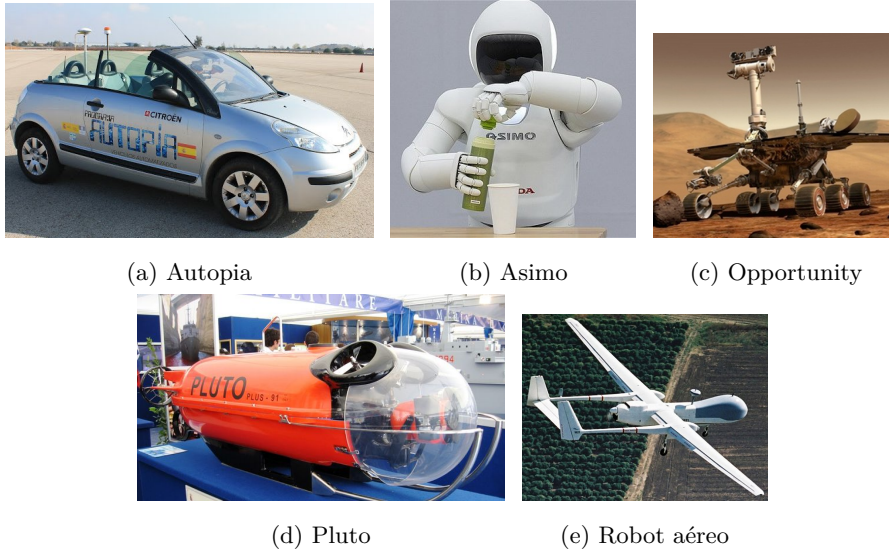


Figura 1.3: Robots móviles

Aunque la mayoría de estos robots son usados principalmente para la investigación, existen otras aplicaciones robóticas mucho más cercanas al día a día de las personas. En este apartado podemos encontrar algunos coches de la marca Volkswagen, como el modelo Touran que ya en el 2007 era capaz de aparcar sin intervención humana. Google entró en el mundo de la robótica con el coche *Google Car*, un coche sin conductor capaz de realizar trayectos urbanos. Fue tal el impacto de este coche que el estado de Nevada, el 29 de junio de 2011, creó una ley que permite que coches autónomos circulen por sus carreteras. El avance en la tecnología robótica en los automóviles fue significativamente impulsado por el DARPA Gran Challenge que es una competición de vehículos autónomos fundada por DARPA (Defense Advanced Research Projects Agency). *Stanley* (ver Figura 1.4a) es el coche robótico de la universidad de Stanford que en 2005 ganó esta competición donde los coches tuvieron que atravesar un entorno desértico lleno de curvas y túneles sin intervención humana.

Pero quizás el mayor impacto de la robótica sobre la vida cotidiana es el robot *roomba* (ver Figura 1.4b) de la compañía *iRobot*. Se trata de un robot aspiradora y fue el primer robot de consumo, se empezó a comercializar en 2002 y actualmente cuenta con una familia de robots capaces de fregar el suelo, limpiar piscinas, abrillantar suelos o limpiar canalones.

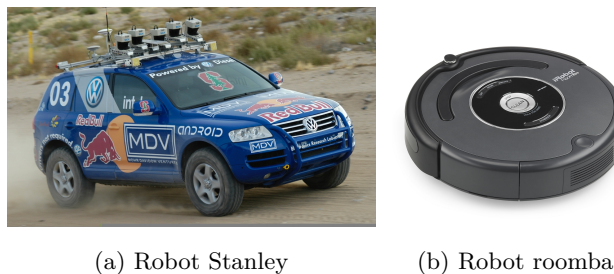


Figura 1.4: Automóvil robótico y robot aspiradora

En el entorno industrial la robótica ha supuesto un nivel de eficiencia y precisión nunca antes conocido en el mundo. Desde las cadenas de montaje automatizadas con brazos robóticos para el montaje de automóviles

(ver Figura 1.5a), robots para tareas de envasado (ver Figura 1.5b), robots clasificadores (ver Figura 1.5c) y un largo etcétera componen un tejido industrial robótico cada vez mayor.



(a) Cadena de montaje



(b) Robot envasador



(c) Robot Baxter

Figura 1.5: Robots industriales

Desde los primeros autómatas hasta los últimos robots enviados a Marte, la robótica ha ayudado al hombre a extender o potenciar las cualidades humanas. El ser humano ha delegado en los robots la realización de las tareas más peligrosas, más repetitivas o aquellas que necesitan de un gran nivel de precisión, para mantener su actual nivel de vida.

1.1.1. Componentes de un robot

Desde un punto de vista informático, un robot es un sistema complejo dotado de sensores, actuadores y procesadores, que combinados, le otorgan capacidades básicas como la percepción, la acción, el procesamiento y la memoria para interactuar con el entorno. De un modo más simple, un robot es un computador con periféricos externos que le permiten obtener información sobre el entorno e interactuar con él.

Los sensores le permiten obtener información del entorno o de sí mismo. Esta capacidad permite que un robot pueda trabajar en escenarios cambiantes modificando su comportamiento en función del entorno en el que se encuentre. Existe una gran variedad de sensores, desde los más básicos como los sensores de luz, de temperatura o de proximidad, a los más complejos y ricos como cámaras, sensores de profundidad, láseres, localizadores GPS, etc.

Los actuadores son dispositivos que permiten al robot interactuar con el entorno y/o realizar movimientos. Dotan al robot de la capacidad de moverse o modificar el entorno en el que trabajan. Permiten que el robot pueda realizar sus tareas en un entorno real, como por ejemplo, la navegación evitando obstáculos o el ensamblado de piezas en una cadena de montaje. Los actuadores puede ser de diversos tipos, como los motores utilizados para el movimiento de las ruedas del robot, los motores para el movimiento de las articulaciones o los motores que hacen girar hélices para proporcionar sustentación en robots aéreos.

Los ordenadores son los encargados de procesar los datos obtenidos por los sensores y de materializar los resultados de los algoritmos de decisión en acciones que llevarán a cabo los actuadores. Los procesadores

otorgan al robot la capacidad de razonamiento que le permite el tratamiento de gran cantidad de datos provenientes de los sensores y la capacidad de ejecutar algoritmos complejos en un periodo corto de tiempo.

1.1.2. El software de los robots

Los componentes vistos en el punto anterior sientan las bases tangibles sobre las que se asienta el software de un robot. El software es un elemento de gran importancia ya que sin él, un robot sería una máquina incapaz de generar un comportamiento inteligente. Como en cualquier otro tipo de desarrollo de software, el software para robots debe cumplir una serie de requisitos:

- **Agilidad:** un robot se encuentra en un escenario dinámico e impredecible en constante interacción con el entorno. Debido a esta característica el software debe ser ágil para proporcionar respuestas rápidas de manera continua.
- **Multitarea:** por la naturaleza de los robots su software debe ser multitarea, tener la capacidad suficiente para recolectar los datos de los sensores, procesarlos, decidir una acción y llevarla a cabo simultáneamente. A todo esto hay que sumarle la capacidad de reacción frente a imprevistos, en un entorno cambiante la realización de una tarea puede verse interrumpida debido a algún cambio en la situación.
- **Distribuido:** con cada vez más sensores, datos más complejos que procesar y algoritmos con una gran carga computacional, la computación distribuida se ha convertido casi en un estándar en el desarrollo de aplicaciones robóticas. Además, a menudo es necesaria la coordinación de varios robots simultáneamente o la utilización de otras fuentes de datos externas al robot, como los sistemas de cámaras para la detección en 3D del robot en un entorno controlado.
- **Lidiar con hardware heterogéneo:** a menudo los dispositivos sensoriales y de actuación son muy diferentes entre sí y cada fabricante desarrolla su propio *driver*. Por eso es necesaria la existencia de una interface software homogénea que permita unificar el acceso a sensores y actuadores con independencia del fabricante o modelo del dispositivo. Esto permite que los algoritmos desarrollados para un determinado robot puedan ser exportados a otro robot distinto para realizar una tarea parecida, posibilitando así la reutilización de código.

Para ayudar al desarrollador encargado de realizar aplicaciones para robots existen varias plataformas software (de código abierto o cerrado) que facilitan la tarea del desarrollo. Estas plataformas normalmente se instalan sobre un sistema operativo, la plataforma es el puente entre el robot y el desarrollador, y aprovechan las características de los sistemas operativos modernos como la abstracción de hardware, la capacidad multitarea y la facilidad para crear aplicaciones distribuidas con librerías externas. Normalmente, si el modelo del robot está muy extendido, el desarrollador contará además con simuladores donde podrá ejecutar su código antes de probarlo en un entorno real.

1.1.3. Visión artificial en Robótica

Desde hace varios años cada vez son más los robots que incorporan cámaras en su conjunto de sensores. Las cámaras nos proporcionan gran cantidad de información sobre el entorno en el que se encuentra el

robot. La visión artificial o visión computacional es una rama de la inteligencia artificial que mediante el uso de computadores es capaz de extraer información de una imagen con el fin de comprender y asimilar lo que en ellas está sucediendo. En 1961 cuando Larry Roberts creó un programa que podía ver una estructura de bloques, analizar su contenido y reproducirla desde otra perspectiva, utilizando una cámara y procesando la imagen desde un computador.

A lo largo de los años se ha demostrado que la visión artificial es muy compleja y que algo que para nosotros pueda resultar automático, puede tardar años en ser resuelto por una máquina. No obstante, cada vez más, las aplicaciones de esta disciplina están saliendo de los laboratorios y se están adentrando más en la vida de las personas. Actualmente no es difícil encontrar aplicaciones de vigilancia basadas en visión artificial, reconocimiento de caras, aplicaciones industriales para la clasificación de objetos o para controles de calidad. Incluso podemos encontrar algunas aplicaciones para *smartphones* o en eventos deportivos.

En la robótica este tipo de sensor ha tenido un gran impacto debido sobre todo a su bajo coste (comparado con otros sensores) y a la gran cantidad de información que se puede extraer de él. Es complicado pensar en un proyecto robótico que no contenga al menos una cámara entre sus sensores.

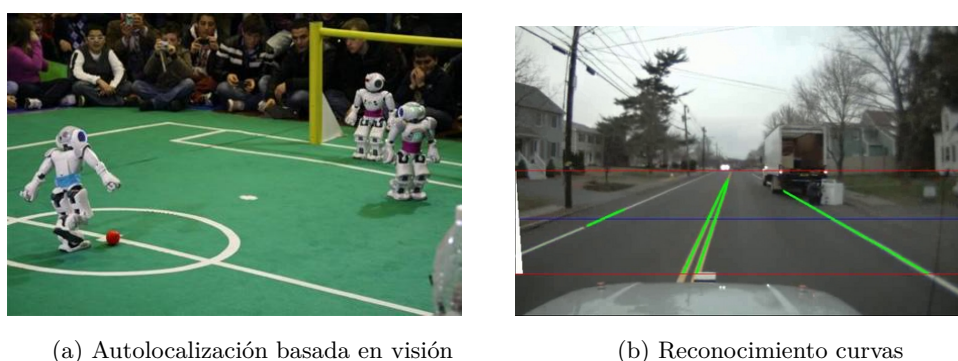


Figura 1.6: Visión artificial en la robótica

En la Figura 1.6a se puede ver a los robots Nao jugando un partido de fútbol. Para poder localizarse a ellos mismos en el terreno de juego, a sus compañeros, a sus contrincantes, la pelota, los límites del campo y la portería contraria, se utiliza una técnica de visión artificial para SLAM (Simultaneous Localization And Mapping). La Figura 1.6b pertenece a uno de los coches robóticos que participó en el DARPA Urban Challenge¹ que gracias a un algoritmo de visión artificial, es capaz de detectar las curvas de la carretera y conseguir que el vehículo se mantenga dentro de su carril sin salirse de la carretera ni cambiar de carril.

Fundamentos

Un sistema de visión artificial es aquel encargado de la formación y procesamiento de las imágenes. Para la formación de las imágenes podríamos dividirlo en varios subsistemas, como el subsistema de iluminación, el de captación y el de adquisición de la señal. Una vez que la imagen se ha captado y se ha introducido en el computador nos encontramos con el subsistema de procesamiento. Éste sistema suele ser un computador que parte de una representación digital de la imagen y la procesa hasta extraer otro

¹<http://archive.darpa.mil/grandchallenge/>

tipo de información de más alto nivel. El procesamiento suele estar organizado en etapas. La primera se conoce como preprocesado. En ella se aplican filtros y transformaciones geométricas con el fin de mejorar la calidad de los datos en la imagen. La segunda etapa, segmentación, consiste en aislar los elementos de la imagen que, en función del tipo de aplicación, puedan resultar más interesantes. Por ejemplo, extraer las zonas boscosas de una imagen capturada por un satélite. Por último, a la tercera etapa se la conoce como la etapa de reconocimiento o clasificación y consiste en extraer características tales como el área o el perímetro, permitiendo así distinguir entre los objetos segmentados.

Espacios de color

Una vez que la señal de vídeo ha pasado por el subsistema de adquisición, éste habrá generado una matriz de valores discretos que almacena en la memoria del computador. El acceso a estos datos se realiza indicando el número de fila y columna, donde la fila corresponde con el eje horizontal y la columna con el vertical. Cada valor contenido en una celda de la matriz se denomina píxel. El origen de coordenadas de la imagen se encuentra típicamente en la esquina superior izquierda. Si la imagen es en blanco y negro, el valor devuelto será la luminancia, si trabajamos en color, devolverá un vector que normalmente suele expresarse en el sistema RGB (Red-Green-Blue).

El cerebro humano percibe el color en función de la longitud de onda dentro del espectro visible. Este espectro comprende las longitudes de onda desde los 380 nm hasta los 780 nm, de este modo el cerebro humano distingue los diferentes colores como muestra la siguiente imagen:

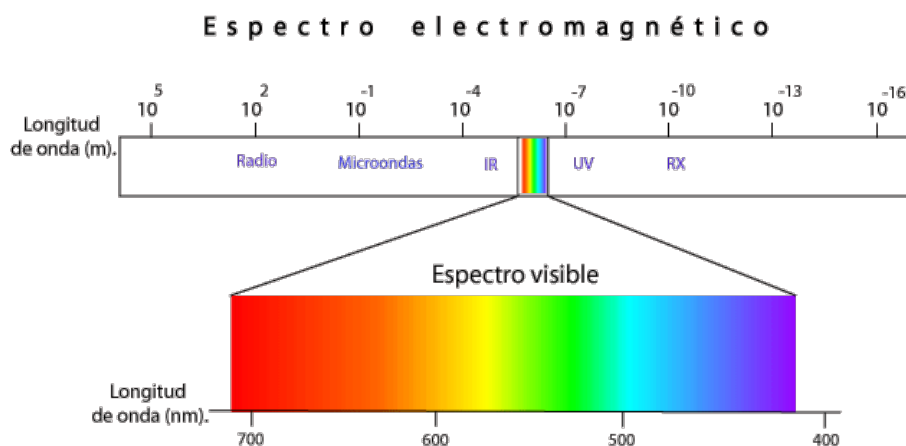


Figura 1.7: Espectro visible

1.2. Vehículos aéreos no tripulados

Un vehículo aéreo no tripulado, de ahora en adelante VANT o UAV por sus siglas en inglés (Unmanned Aerial Vehicle), es una aeronave que vuela sin tripulación humana a bordo. Hay que distinguir entre los VANT y los misiles, un VANT se define como un vehículo sin tripulación y reutilizable, capaz de mantener

un nivel de vuelo controlado y sostenido. Los misiles no son considerados VANT porque el propio vehículo es un arma que no se puede reutilizar.

En 1898 el gobierno de Estados Unidos concedió a Nikola Tesla una patente para un método y un aparato capaz de controlar el mecanismo de movimiento de buques o vehículos, la patente cubre cualquier tipo de embarcación o vehículo capaz de ser impulsado y dirigido. Durante las siguientes décadas, el pentágono planeo invertir billones de dolares en crear una nueva generación de vehículos no tripulados para diversas aplicaciones militares en tierra, mar y aire. El primer intento de crear un VANT fue en el transcurso de la Primera Guerra Mundial. En el año 1916, la Royal Flying Corps propuso utilizar a investigadores civiles para encontrar un modo por el que poder controlar de forma remota un avión, el proyecto era conocido con el nombre “Aerial Target”. En la década de 1930 la British Royal Navy desarrolló un primitivo VANT radio controlado, el Queen Bee, este VANT podría ser reutilizado para otros vuelos y era capaz de alcanzar los 160 km/h. El Queen Bee sirvió, principalmente, para su uso como objetivo de las defensas antiaéreas.

En los años 60 y 70, los Estados Unidos realizaron más de 34.000 vuelos de vigilancia usando un VNAT llamado AQM-34 Ryan Firebee. Este VANT era lanzado desde otro avión donde se encontraban los operadores que se encargaban de teleoperar el aparato. Los Estados Unidos también utilizaron otro VANT llamado Lightning Bugs que era lanzado desde un C-130 en misiones sobre China y Vietnam.



Figura 1.8: AQM-34 Ryan Firebee

A finales de los 70 y en los 80, Israel desarrolló el Scout y el Pioneer, lo que cambió el modelo utilizado para el desarrollo de VANT hacía modelos de tipo planeador ligero. El Scout era capaz de transmitir vídeo en directo con una vista de 360° del terreno. Los Estados Unidos compraron los VANT Pioneer del ejército de Israel para utilizarlos en la Guerra del Golfo.



(a) Scout



(b) Pioneer

Figura 1.9: VANT desarrollados en Israel

Aunque la tecnología de los VANT se desarrolló durante el siglo XX, no fue hasta la aparición del

Predator, un VANT desarrollado por los Estados Unidos, cuando se comenzó a intensificar su uso en los ejércitos modernos. Aunque estos vehículos fueron principalmente desarrollados para fines bélicos, desde comienzos del siglo XXI se han utilizados para otros fines como la investigación, la seguridad ciudadana o el control de cosechas, por poner algunos ejemplos.

1.2.1. Clasificación de los VANT

Los VANT se pueden clasificar según la misión para la cual fueron diseñados o por su alcance y capacidades. Según su propósito suelen ser clasificados en 6 tipos:

- De blanco, sirven para simular aviones o ataques enemigos en los sistemas de defensa de tierra o aire.
- Reconocimiento, enviando información militar.
- Combate (UCAV), para el uso de los VANT en operaciones militares de combate.
- Logística, diseñados para llevar carga.
- Investigación y desarrollo, para el desarrollo de nuevas tecnologías.
- VNAT comerciales y civiles, diseñados para el público general.



Figura 1.10: VANT Ikhana.

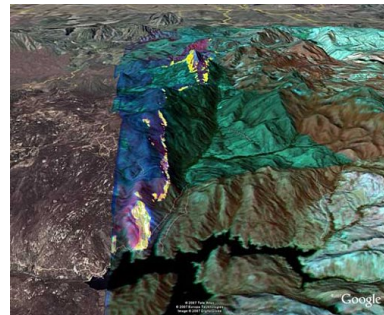


Figura 1.11: Imagen captada por uno de los sensores incluidos en el VANT de investigación Ikhana de la NASA, sobre Harris Fire en San Diego.

Atendiendo al tamaño y alcance, se pueden clasificar en las siguientes categorías:

- Handheld: unos 600 m. de altitud y hasta 2 km de alcance.
- Close: unos 1500 m. de altitud y hasta 10 km de alcance.
- NATO: unos 3000 m. de altitud y hasta 50 km de alcance.
- Tactical: unos 5500 m. de altitud y hasta 160 km de alcance.
- MALE (Medium Altitude, Long Endurance): hasta 9000 m. y hasta 200 km de alcance.
- HALE (High Altitude, Long Endurance): sobre 9000 m. de techo y alcance indeterminado.
- HYPERSONIC: alta velocidad, supersónico (Mach 1-5) o hipersónico (Mach 5+) unos 15000 m. de altitud o altitud suborbital y un alcance de hasta 200 km.

- ORBITAL: en orbitas bajas terrestres (Mach 25+).
- CIS Lunar: viaja entre la Luna y la Tierra.



Figura 1.12: VANT Orbiter de tipo Handheld



Figura 1.13: Estación de tierra para el Orbiter

1.2.2. Cuadricópteros

Hasta ahora todos los VANT introducidos en este capítulo cuentan con capacidades CTOL (*Conventional Take-Off and Landing*) que son aquellos vehículos que necesitan una pista para el despegue y aterrizaje. Los vehículos aéreos no tripulados de tipo VTOL (*Vertical Take-Off and Landing*) son vehículos con capacidad para el vuelo estacionario y despegue y aterrizaje en vertical. Estos vehículos son capaces de mantener el vuelo estacionario en altura y dirección, consiguiendo así mantenerse estáticos en un punto a una altura determinada. Un claro ejemplo de vehículo tipo VTOL son los helicópteros, aunque también existen aviones con capacidades VTOL como el avión Harrier o el menos conocido Boeing V-22 Osprey.



Figura 1.14: Boeing V-22 Osprey

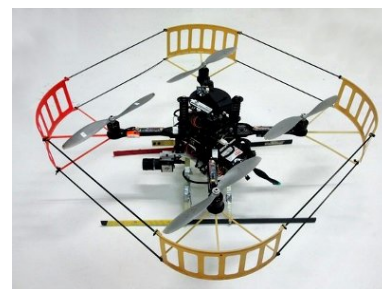


Figura 1.15: Cuadricóptero Pelican

Principios de Sustentación y Movimiento

Dentro de los VANT de tipo VTOL de corto alcance podemos encontrar desde vehículos de única hélice hasta vehículos con 8 hélices pasando por vehículos de 4 hélices, llamados cuadricópteros. Los cuadricópteros son helicópteros con cuatro rotores que, generalmente, están colocados en la extremidad de una cruz. Al igual que los aviones o helicópteros, los cuadricópteros pueden volar debido al movimiento de un plano aerodinámico (las palas del rotor) a través del aire. En este caso las palas describen un movimiento circular a diferencia de los aviones que necesitan el desplazamiento para el movimiento del plano aerodinámico (un ala) a través del aire. Debido a que el aire pasa a través de un plano aerodinámico,

se produce una presión diferencial. La presión que existe en la superficie superior es menor a la presión ejercida en la parte inferior. Esta diferencia de presiones da como resultado la fuerza de sustentación. Cuando la sustentación es mayor que el peso del cuadricóptero, este comienza a volar.

Sin embargo, la capacidad para generar sustentación no es el único problema al que se enfrentan los vehículos VTOL. Si tuviéramos un helicóptero con un único motor, con la capacidad de generar la suficiente sustentación para elevar el vehículo del suelo, este acabaría girando sobre el eje del motor en el sentido contrario al giro del mismo. Esto se debe a que el empuje de la hélice contra el aire es perpendicular al plano de cada pala, por lo que existe una componente horizontal que sigue el sentido del giro de las palas. Estos empujes crean una fuerza inercial en el sentido contrario al giro de cada pala. Cada hélice esta formada por dos palas que forman ángulos contrarios con el plano, por esta razón la componente horizontal inercial en cada pala está formada por el mismo vector pero con sentidos contrarios. Esto hace que las componentes lineales se anulen, pero aparece un momento en sentido contrario al giro de la hélice. Al sumar todas las fuerzas ejercidas, queda una resultante vertical hacia arriba y un momento en el sentido contrario al giro de la hélice.

En los helicópteros, para que al girar su rotor no de vueltas sobre sí mismo, han de tener otro rotor que anule la fuerza generada por el rotor principal. En los helicópteros coaxiales (helicópteros de doble rotor) el rotor secundario se encuentra en el mismo eje que el rotor principal. En los helicópteros no coaxiales, este segundo rotor se encuentra en la cola.

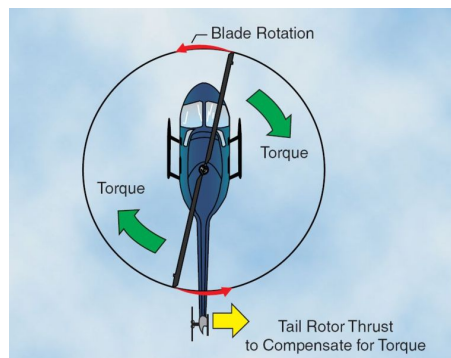


Figura 1.16: Compensación del par motor

Los cuadricópteros, sin embargo, utilizan este efecto físico para cambiar de dirección. Como se explicó anteriormente, la disposición de los rotores en un cuadricóptero sigue una forma de cruz donde existen dos pares de rotores opuestos. Cada par de rotores opuestos gira en el mismo sentido. Un par en el sentido de las agujas del reloj y el otro en el sentido contrario.

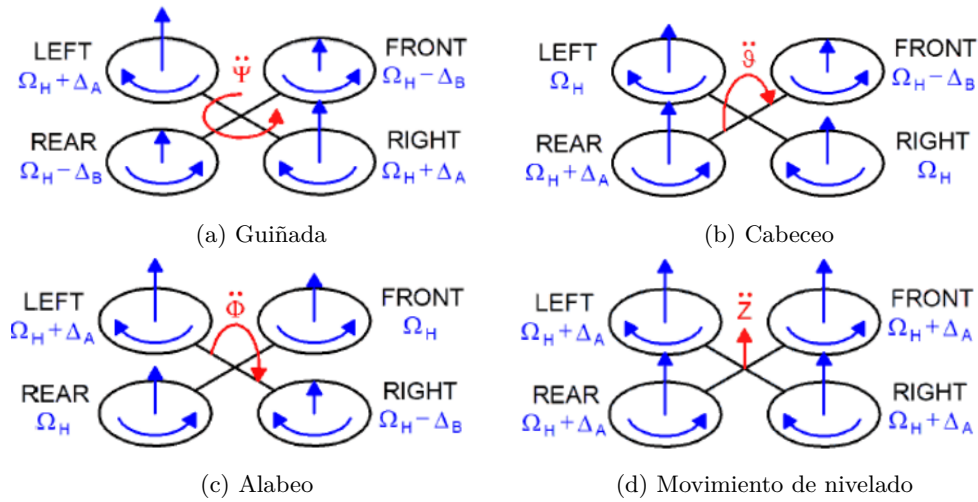


Figura 1.17: Configuración de los rotores

Con esta configuración, si a los rotores que giran en el sentido anti-horario les imprimiéramos más potencia (ver Figura 1.17a), el cuadricóptero giraría hacia la derecha. Si la potencia se la imprimiéramos a los rotores opuestos, el cuadricóptero giraría hacia la izquierda. Este giro en el plano horizontal, que provoca el giro del vehículo sobre sí mismo, se denomina guiñada o *yaw* (ver Figura 1.18).

En el caso de que ampliáramos la potencia en los rotores de un extremo (ver Figura 1.17b), por ejemplo el trasero, el cuadricóptero se desplazaría hacia delante provocando que la cabeza del vehículo se inclinase hacia el suelo. A este movimiento se le conoce como *pitch* o cabeceo (ver Figura 1.18).

Otro movimiento que nos permiten los grados de libertad del vehículo es el *roll* o alabeo (ver Figura 1.18). Este movimiento consiste en aumentar la potencia en los motores de uno de los laterales (ver Figura 1.17c), por ejemplo, si ampliáramos la potencia de los rotores del lateral izquierdo el cuadricóptero se inclinaría hacia la derecha.

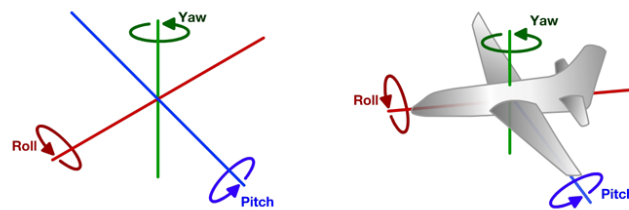


Figura 1.18: Roll, Pitch y Yaw

Por ultimo, para que el cuadricóptero se mantenga nivelado, se debería mantener por igual la potencia en los cuatro rotores (ver Figura 1.17d).

1.3. Robótica aérea

La robótica aérea es la rama de la robótica que se encarga del estudio del comportamiento autónomo de los vehículos aéreos no tripulados, es la intersección entre las áreas descritas en las dos secciones anteriores. En los últimos años, un nuevo reto científico y tecnológico en este área ha sido conseguir que estos vehículos sean totalmente autónomos, es decir, que puedan volar sin la conducción, ni siquiera supervisión, de una persona.

A medida que avanza la tecnología y se abaratan los costes, se están integrando en estos vehículos cada vez tecnologías más avanzadas como sistemas de cómputo más poderosos, sensores más ricos en información (cámaras, láseres,...), baterías con mayor autonomía,... El abaratamiento de los VANT ha conseguido que cada vez más centros de investigación a lo largo del mundo puedan disponer de estos vehículos consiguiendo así logros que eran impensables hace unos años. Esto está aumentando el campo de aplicación de los VANT, han pasado de utilizarse únicamente en entornos militares a poder utilizarse para servicios forestales, agricultura, medio ambiente, hidrología, cartografía, desastres naturales o geología. Sin embargo y a pesar de los grandes avances conseguidos en los últimos años, todavía quedan grandes obstáculos que superar. No sólo avances tecnológicos, sino éticos y de regulación por motivos de seguridad. La robótica aérea acaba de empezar a recorrer lo que seguramente sea un gran camino.

En diciembre de 2013 Amazon sorprendió al mundo demostrando su interés en el envío de paquetes con *drones*. Según la propia empresa actualmente se encuentra en la fase de investigación y pruebas con unos VANTs de ocho rotores (optocóptero) que son capaces de levantar cargas de 2.3kg y entregar el paquete en un periodo de 30 minutos desde que el cliente realizase su pedido a través de la web. En ese mismo mes, la empresa de envíos DHL mostró su interés en el uso de *drones* para el reparto de paquetes. Cuentan con unos VANTs con una capacidad de carga de 2.99kg y un radio de acción de 1km. No sólo hay interés en el envío de mercancías con estos aparatos, el Ayuntamiento de Madrid, en diciembre de 2013 realizó un simulacro de rescate en el teleférico de Madrid en el que participaron los Bomberos, el Samur y la Policía municipal, en el que se utilizó un VANT de ocho rotores dotado con sensores para detectar gases, temperatura y agentes radioactivos, biológicos y químicos. Equipado con una cámara fue capaz de retransmitir a los puestos de control las imágenes de las víctimas involucradas en el rescate.

En este capítulo viajaremos a lo largo del mundo recogiendo las aplicaciones de la robótica aérea. En él podremos encontrar desde aplicaciones comerciales, hasta los desarrollos punteros en algunos de los centros de investigación más prestigiosos del mundo. Daremos una vuelta sobre las distintas competiciones de robótica aérea para acabar con los desarrollos que se están haciendo en nuestro país y en nuestra universidad. No pretende ser exhaustivo, sino visitar unos cuantos ejemplos representativos.

1.3.1. Robótica aérea en el mundo

Investigación

Dentro de la universidad ETH Zürich² se encuentra el Instituto de Sistemas Dinámicos y de Control. Entre sus instalaciones destaca el Flying Machine Arena³ (FMA) donde un grupo de investigadores lleva a cabo sus experimentos sobre el control de vehículos aéreos no tripulados. Este laboratorio cuenta con un sistema de captura de movimientos de alta precisión, una red inalámbrica y un software desarrollado por ellos mismos que, utilizando algoritmos muy sofisticados, permiten la estimación y el control de los VANT con los que trabajan. El sistema de captura de movimientos es capaz de localizar múltiples objetos a un ritmo que excede los 200 fotogramas por segundo. En un sistema muy veloz, pero los VANT en el espacio pueden desplazarse a una velocidad de 10 m/s, lo que resulta que se mueven unos 5 centímetros entre dos capturas sucesivas. La información de este sistema se cruza con otros datos y modelos del sistema dinámico para predecir el estado de un objeto en el futuro. El sistema utiliza estos datos para determinar el siguiente comando que se tiene que ejecutar en el vehículo para conseguir un determinado movimiento, y a continuación, mediante la red inalámbrica se envía este comando al vehículo que realiza la acción.



Figura 1.19: VANT construyendo una torre

La Figura 1.19 anterior muestra uno de los experimentos realizados por este grupo. Se trata de la construcción de una torre de 6 metros de altura que necesita 1500 ladrillos de espuma. El experimento se realizó con la colaboración de los arquitectos Gramazio & Kohler, con la idea de explorar el potencial de los cuadricópteros en la construcción de edificios. La imagen fue captada en un museo de arquitectura en París.

El proyecto SHERPA⁴ es un proyecto financiado por la Unión Europea que propone desarrollar un grupo mixto de robots terrestres y aéreos que den soporte en tareas de búsqueda y rescate en entornos hostiles como en los entornos de montaña. En este escenario el equipo de rescate estará formado tanto por personas como por robots, que trabajarán conjuntamente en las tareas de búsqueda y rescate. Los equipos de rescate contarán con un vehículo de tierra con una gran capacidad de carga conocido como “*intelligent donkey*” y con dos tipos de VANT, “*trained wasps*” y “*patrolling hawks*”. El proyecto se centra en la investigación de la interacción y colaboración entre los robots y los equipos de rescate. El sistema

²<https://www.ethz.ch/en.html>

³<http://www.flyingmachinearena.org/>

⁴<http://www.sherpa-project.eu/sherpa/>

SHERPA hace hincapié en la robusta autonomía de la plataforma, la adquisición de capacidades cognitivas, las estrategias de colaboración y la interacción entre el equipo humano y los robots.

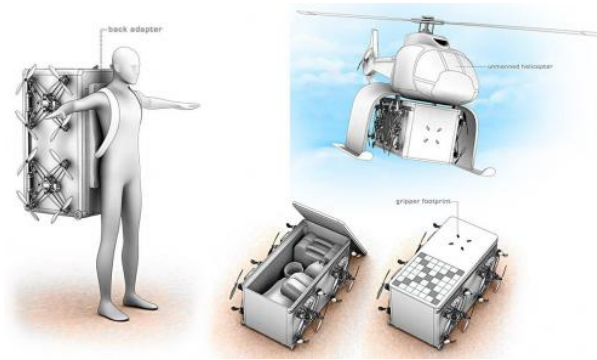


Figura 1.20: Sherpa Box

El equipo aéreo está compuesto por dos tipos de VANT, los primeros son pequeños VANT con una alta maniobrabilidad pensados para la captura de datos, el seguimiento del equipo humano o mantenerse junto a la persona auxiliada. El segundo tipo de VANT son vehículos con una mayor rango de acción que le permite cubrir distancias muy superiores, están diseñados para volar con condiciones adversas y tienen la capacidad de transportar cargas, como la Sherpa Box, que en su interior puede contener una pequeña estación de carga, pequeños VANT o almacenar elementos y herramientas para los equipos de rescate (ver Figura 1.20).

Usos comerciales

Matternet⁵ es una *startup* americana que ha desarrollado un sistema basado en VANT para el envío de provisiones a las regiones más pobres en los países sub-desarrollados. A menudo la población en estas regiones tiene que desplazarse kilómetros atravesando terrenos desérticos, grandes ríos, lagunas, montañas o depresiones geográficas. Las infraestructuras de comunicación existentes en muchas de las regiones son ineficientes o simplemente inexistentes.



(a) Proyecto Matternet

(b) Fly-n-Sense Research

Figura 1.21: Aplicaciones comerciales

La idea de Matternet es crear una red de transporte utilizando VANT, un sistema de bajo coste, energéticamente eficiente, con un pequeño impacto ecológico, fácilmente reconfigurable y requiriendo únicamente una pequeña infraestructura. El sistema consiste en que, mediante el uso de la telefonía móvil,

⁵ <http://matternet.us/>

una persona pueda solicitar medicamentos u otro tipo de provisiones indicando su posición, en ese momento un VANT con una capacidad de carga de 2kg y con un rango de acción de 10 km, emprenderá el vuelo hasta el lugar donde hayan sido solicitadas las medicinas para realizar la entrega. Los VANT estarían esparcidos por el territorio en unas columnas que servirán como centro de aterrizaje y despegue.

Fly-n-sense⁶ es una empresa francesa que se dedica a ofrecer diversos servicios con VANT. Cuentan con cuadricópteros y aviones para realizar sus funciones y trabajan en sectores como la investigación, la industria, la seguridad y el estudio del entorno natural para instituciones del gobierno y agencias de energía. Forma parte del Aerospace Valley que es un conjunto de empresas y centros de investigación aeroespaciales ubicado en Francia.

Entre los servicios que ofrece se encuentra la motorización e inspecciones de edificios, la seguridad de instalaciones interiores o exteriores, la investigación atmosférica, arqueología aérea o estudios sobre el impacto humano en la naturaleza, detección de la polución y catástrofes medioambientales.

Competiciones

Desde hace unos años atrás se han creado competiciones para VANT. En estas competiciones, donde los equipos tienen que competir en crear una aplicación para superar un determinado reto, se pretende fomentar la investigación sobre estos aparatos. Normalmente están enfocadas a estudiantes universitarios aunque existen algunas competiciones para estudiantes de instituto.

IARC⁷ (International Aerial Robotic Competition) es una competición de robótica aérea, creada en 1991 en el Georgia Institute of Technology, en la que compiten universidades de todo el mundo. El principal propósito de la competición es avanzar en la investigación de los comportamientos en la robótica aérea. Desde 1991 hasta el 2013 se han propuesto un total de 7 misiones. Cada una de ellas consiste en desarrollar un comportamiento totalmente autónomo que sea capaz de superar la prueba. Se crean nuevas misiones cada vez que se supera la anterior, por lo que, una misión puede estar años sin resolverse hasta que algún equipo consigue el objetivo.

⁶<http://www.fly-n-sense.com/>

⁷ <http://www.aerialroboticscompetition.org/>



Figura 1.22: VANT inspeccionando un edificio en IARC

La séptima misión tiene tres objetivos, el primero es la interacción de los robots aéreos con robots terrestres, consiste en poder utilizar los robots aéreos desde plataformas móviles como barcos, camiones o incluso otros vehículos aéreos. El segundo consiste en la navegación de interiores con robots aéreos sin ayudas como el uso de GPS o puntos de referencias en las paredes. Por último, el tercer objetivo, consiste la interacción entre múltiples robots aéreos, los robots deberán ser capaces de detectar obstáculos fijos y móviles para evitar una colisión. Los vehículos utilizados para estas misiones suelen ser cuadricópteros o helicópteros de distintos tamaños en función de la misión a realizar. Para los vehículos terrestres, la organización propone una plataforma que ofrece a los equipos, junto con todos los recursos necesarios para que puedan trabajar con ella. Cuando un equipo gana la competición obtiene en premio económico de 10000 dolares por cada año que la misión haya estado sin resolver.

El UAV Outback Challenge⁸ es una competición de robótica aérea creada en el 2007 para institutos. Su propósito es promover el empleo de VANTs para uso civil desarrollando vehículos de bajo coste usado para misiones de búsqueda y rescate. La competición tiene lugar en el aeropuerto de Queensland, Australia.

La misión consiste en que los equipos participantes reciben un paquete de emergencia que se usará el día de la competición. Los equipos tendrán que desarrollar un sistema que sea capaz de entregar el paquete a través de un vehículo aéreo. Por un lado tienen que desarrollar el sistema de navegación del VANT y por el otro el sistema de descarga del paquete. Para este último sistema se puede elegir que el vehículo lo realice de forma autónoma o con intervención humana. Los organizadores establecerán un punto donde los vehículos tendrán que realizar la entrega del paquete. Este punto puede ser proporcionado por los organizadores o localizado por los VANTs de los equipos. Gana el equipo que consiga entregar el paquete en el punto más próximo al objetivo en el menor tiempo posible. Como en otras competiciones,

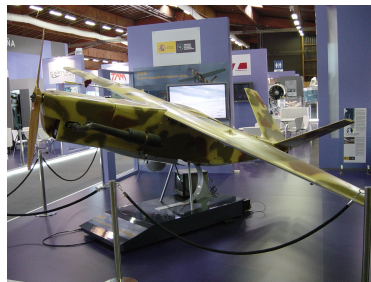
⁸<http://www.uavoutbackchallenge.com.au/index.cfm>

los ganadores reciben un premio en metálico de 5000 dolares.

1.3.2. Robótica aérea en España

Investigación

El Instituto Nacional de Técnica Aeroespacial (INTA) lleva más de dos décadas impulsando un amplio programa de investigación y desarrollo de VANT. La principal tarea de los vehículos desarrollados en este centro es vigilancia y observación, aunque ya se está trabajando en modelos que irán armados.



(a) SIVA



(b) ATLANTE

Figura 1.23: VANT españoles

El primer vuelo de un VANT de fabricación española fue en el año 2000, este VANT (ver Figura 1.23a) conocido como SIVA (Sistema Integrado de Vigilancia Aérea) está fabricado en fibra de carbono de alta resistencia, tiene una longitud de 4 metros, una autonomía de 7 horas y un alcance máximo de 150 kilómetros. Su techo de vuelo es de 4.000 metros y puede alcanzar los 200 kilómetros por hora. Incorpora potentes cámaras de visión e infrarrojos y se le puede dotar de diferentes sensores según la misión encomendada. Puede despegar en pista o mediante una lanzadera neumática, y también puede aterrizar con paracaídas. El ATLANTE (Avión Táctico de Largo Alcance No Tripulado Español) es un VANT (ver Figura 1.23b) diseñado para responder a los requerimientos del Ejército de Tierra Español, este *drone* ofrece al ejército la capacidad ISTAR (*Intelligence, Surveillance, Target Acquisition and Reconnaissance*) las 24 horas del día y en condiciones climatológicas adversas.



(a) Tauro



(b) Milano

Figura 1.24: VANT españoles

Otro VANT desarrollado por el INTA es Tauro (ver Figura 1.24a). Se trata de un prototipo que puede aterrizar y despegar como un helicóptero, en pleno vuelo tiene la capacidad de cambiar su configuración extendiendo unas alas y pasando la potencia del rotor a una hélice colocada en la cola del vehículo

consiguiendo así pasar de la configuración de un helicóptero a un avión en pleno vuelo.

El último proyecto del INTA es Milano (ver Figura 1.24b), se trata de un VANT similar a los Reaper y Predators norteamericanos. Es un VANT de 8 metros de longitud, 12,5 de envergadura y un techo de vuelo de 8.000 metros. Tiene la capacidad de ser controlado vía satélite y podrá transportar hasta 200 kg lo que permitirá que este vehículo pueda ir armado con misiles. Su autonomía llegaría hasta las 24 horas de vuelo. Actualmente se encuentra en fase de desarrollo y se prevé que comenzará a volar en el año 2014.

El proyecto ARCAS⁹ (Aerial Robotics Cooperative Assembly System) está financiado por la Unión Europea y en el participan dos universidades españolas, la Universidad de Sevilla y la Universidad Politécnica de Cataluña. El proyecto está liderado por el profesor Anibal Ollero de la universidad de Sevilla. El propósito del proyecto se basa en desarrollar un entorno para el diseño y desarrollo de operaciones de montaje con robots aéreos.



Figura 1.25: Montaje de estructura en incendio con ARCAS

Según los investigadores del proyecto ARCAS será posible la construcción de estructuras para evacuación de personas en operaciones de rescate, la instalación de plataformas en terrenos irregulares o la construcción, inspección y mantenimiento de estructuras. Los VANT de la plataforma cuentan con brazos robóticos que ayudarán al transporte o manipulación de objetos. Para reconocer e identificar el escenario de trabajo los vehículos contarán con un módulo de percepción que podrá generar modelos 3D. Como sistema de navegación utilizarán técnicas de visión artificial como el SLAM 3D aéreo o seguimiento en 3D. El sistema será responsable de coordinar los múltiples robots que se encuentren trabajando y por si fuera necesario, podrá disponer de la intervención de un operador humano. El proyecto comenzó el 28 de noviembre de 2011 y cuenta con una financiación de más de 6 millones de euros.

Vision4Uavs¹⁰ es un grupo de investigación del Centro de Automática y Robotica (C.A.R) de la Universidad Politécnica de Madrid, su trabajo se desarrolla en el campo de la visión artificial y el control automático y cuentan con una larga experiencia en aplicaciones industriales. El grupo está liderado por el profesor Pascual Campoy que empezó con la investigación en VANT en la década de los 90.

⁹ <http://www.arcas-project.eu/>

¹⁰ <http://www.vision4uav.eu/>

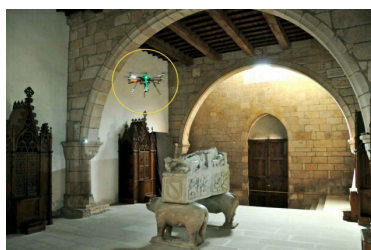


Figura 1.26: Seguimiento de un vehículo

Entre las aplicaciones desarrolladas por este grupo de investigación podemos encontrar la inspección industrial de torres de luz, que consiste en el uso de cuadricópteros que cuentan con algoritmos de visión artificial capaces de detectar objetos por su forma, dimensiones, patrones, etc... Una vez definido el objeto de la torre de luz que se quiere inspeccionar, el cuadricóptero volará a lo alto de la torreta y se posiciona frente al objeto que debe ser inspeccionado. Otro proyecto desarrollado por este grupo es el seguimiento de otros vehículos combinando distintas técnicas de visión artificial, como estimaciones de movimiento y alineamiento de imágenes. En el año 2013 ganaron la competición internacional IMAV (International Micro Air Vehicle Conference and Flight Competition) en la categoría de cuadricópteros en vuelos interiores.

Usos comerciales

Aerotools-UAV¹¹ es una empresa española que se dedica al diseño, fabricación, comercialización y operación de sistemas con VANT. Además colaboran con universidades y centros de investigación en procesos de I+D para el desarrollo de cuadricópteros. Ofrecen servicios a distintos sectores como el uso de estos vehículos en aplicaciones civiles, industriales o militares.



(a) Catalogación de patrimonio



(b) Retransmisión de eventos deportivos

Figura 1.27: Aplicaciones de Aerotools-UAV

El sistema que comercializan está compuesto por un VANT y una estación de control con comunicación bidireccional permanente por distintos canales de radiofrecuencia. El sistema está diseñado para ser

¹¹<http://www.aerotools-uav.com/>

manejado por uno o más operadores. Las plataformas aéreas que construyen pueden ser de ala fija o de despegue vertical, con motores eléctricos o de explosión montados sobre chasis de materiales compuestos y aleaciones aeronáuticas. Su plataforma ha sido utilizada para muchos fines, desde seguridad ciudadana, investigación de accidentes de tráfico o incendios forestales, catalogación, cuidado y vigilancia del patrimonio histórico (ver Figura 1.27a) hasta la retransmisión de eventos deportivos (ver Figura 1.27b).

Ixion¹² es una empresa muy reciente que nace con el objetivo de automatización de la sociedad. Se crea como un *spin-off* tecnológico de TCP Sistemas e Ingeniería que cuenta con más de 20 años de experiencia. Ofrece a sus clientes el desarrollo de monitorización y control de infraestructuras, desarrollo de sistemas para vehículos no tripulados, sistemas de visión artificial para vigilancia y desarrollos I+D con instituciones públicas y privadas. Ixion desarrolla sistemas de navegación que permiten a sus vehículos llevar a cabo tareas con un alto grado de autonomía. Se basan en sistemas avanzados de sensores que perciben su entorno. Los datos obtenidos con estos sensores se cruzan con los obtenidos por la unidad inercial y con información GPS, con el fin de obtener una navegación segura y fiable.



Figura 1.28: Sistema de navegación autónoma

Bruselas ha impulsado el diseño de un programa de refuerzo tecnológico que incluye el uso de VANTs para la vigilancia de las fronteras. Estos vehículos no estarán dotados de armas ya que en Europa no está autorizado de uso de estas en VANT. El proyecto, denominado Perseus, está siendo desarrollado por la empresa Indra con la colaboración de la Guardia Civil.



Figura 1.29: Ministro de Interior junto a una maqueta de un VANT

¹²<http://www.ixion.es/>

El primer vuelo de prueba tuvo lugar el 24 de septiembre de 2013 que despegó desde un aeropuerto de Almería para llevar a cabo un vuelo de dos horas para el que se cerró una zona del espacio aéreo. El vuelo consistía en el simulacro de rescate de una embarcación supuestamente cargada de inmigrantes en el mar de Alborán controlado desde Madrid en el que participaron también dos embarcaciones de la Guardia Civil. El proyecto consiste en ampliar el espacio de control de los sistemas de detección de embarcaciones que intentan alcanzar las costas europeas de forma irregular y en compartir la información recabada por los diferentes países, cada uno de ellos con un centro de coordinación nacional.

1.3.3. Robótica aérea en la Universidad Rey Juan Carlos

En el año 2013, dentro del grupo de robótica de la Universidad Rey Juan Carlos, se puso en marcha una nueva línea de investigación sobre VANT. En ella se pretende aplicar técnicas de visión artificial para el control de los VANT. Aunque todavía se encuentra en una fase inicial ya se han desarrollado varios proyectos dentro de la línea. El objetivo es que las aplicaciones desarrolladas en estos proyectos sean fácilmente exportables a otro tipo de vehículos. De este modo, si un miembro del grupo desarrolla un algoritmo de navegación para un cuadricóptero, este mismo algoritmo podrá ser utilizado para un vehículo de ala fija.

Hasta este momento, los proyectos que se han creado han sido para desarrollar la infraestructura necesaria para el control de los vehículos. Entre estos proyectos no encontramos con los trabajos de: Óscar Higuera, Óscar García y Livio Calvo. Óscar Higuera basó su Proyecto Fin de Carrera [11] en el desarrollo de un cuadricóptero llamado X-Pider. El proyecto abarca tanto el diseño e implementación hardware como el diseño e implementación software del vehículo. El cuadricóptero desarrollado por Óscar tiene la capacidad de vuelo estacionario estable monitorizado y controlado tan peculiar de los cuadricópteros. Para conseguir este comportamiento, el vehículo interpreta adecuadamente todos los datos procedentes de los sensores, los procesa y ejecuta las distintas rutinas de control que permiten que el vehículo realice un vuelo estable, el control automático y estabilización de altura, estabilización del rumbo y la capacidad de mantener comunicaciones inalámbricas con la estación base de motorización, mando y control (ver Figura 1.30).



Figura 1.30: Estación base desarrollada por Óscar Higuera

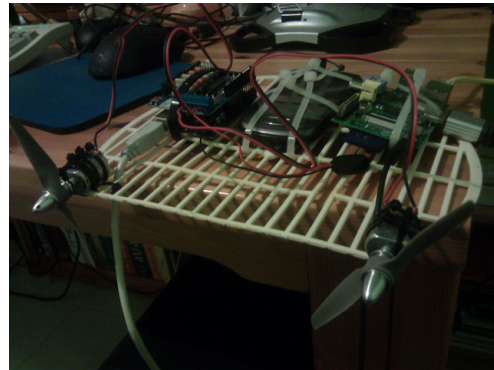
Óscar García, estudiante del master de Visión Artificial de la Universidad Rey Juan Carlos, está desarrollando, para su Trabajo Fin de Master ¹³ una plataforma aérea, que en conjunto con la desarrollada

¹³http://jderobot.org/Oscar_Garcia_TFM

por Livio Calvo, permitirá la ejecución de algoritmos de visión artificial en el ordenador de a bordo que actualmente está en fase de desarrollo, se instalará en la plataforma Phantom-FX61 (ver Figura 1.31a).



(a) Plataforma Phantom-FX61 utilizada por Óscar García



(b) Ordenador de a bordo desarrollado por Livio Calvo

Figura 1.31: Robótica aérea en la URJC

El trabajo de Livio Calvo¹⁴ consiste en el desarrollo de un prototipo inicial de un VANT. El sistema de a bordo está compuesto por un *smartphone* con Android, una placa Arduino y otra Raspberry Pi. El sistema de sustentación comenzó con un dirigible, pero finalmente Livio está trabajando con la plataforma Phantom (ver Figura 1.31a).

Dentro de este contexto se encuadra este Trabajo Fin de Grado, que persigue incorporar el cuadricóptero Ar.Drone 1.0 al conjunto de los robots con soporte en JdeRobot y la realización de pruebas de concepto de navegación aérea basadas en visión artificial.

Para acabar el capítulo de Introducción, a continuación se explican los contenidos que el lector podrá encontrar en las distintas partes de este documento. En el segundo capítulo se describirán los objetivos propuestos para este Trabajo Fin de Grado. El tercer capítulo detalla las plataformas hardware y software sobre las que se ha realizado el trabajo. El cuarto capítulo del documento detalla toda la infraestructura software que ha sido necesaria desarrollar para proporcionar acceso software a los sensores y actuadores del cuadricóptero desde otros componentes o aplicaciones. En el capítulo quinto, Navegación Visual, se explicará cómo se ha aprovechado la información generada por las cámaras de Ar.Drone para la generación de comportamientos autónomos en aplicaciones de seguimiento. Estos comportamientos han sido validados experimentalmente sobre el cuadricóptero real, tal y como muestra el capítulo 6. Para concluir el trabajo en el capítulo final se recapitulan los aportes y lecciones aprendidas.

¹⁴http://jderobot.org/Livio_Calvo-pfc-itis

Capítulo 2. Objetivos

En este capítulo vamos a describir de manera concreta los objetivos planteados para el Trabajo de Fin de Grado, los requisitos exigidos y la metodología utilizada para el desarrollo.

2.1. Descripción del problema

El objetivo general del proyecto es programar un cuadricóptero para el seguimiento de objetos con las cámaras que éste lleva incorporadas, tanto la frontal como la ventral. Para alcanzar este objetivo, se ha subdividido en tres subobjetivos:

- Desarrollo dentro de la plataforma JdeRobot de una infraestructura software que permita el acceso a los sensores y actuadores del cuadricóptero Ar.Drone 1.0 desde una aplicación externa.
- Diseño y desarrollo de una aplicación de control visual que gobierna el movimiento del cuadricóptero para que siga objetos. Se compone de una parte perceptiva, para el análisis de imágenes, y una parte de actuación que tomará las decisiones de control que comandarán el cuadricóptero
- Validación experimental en el cuadricóptero real. Se realizarán varias pruebas que demuestren el correcto funcionamiento de la infraestructura y el funcionamiento del control visual.

2.2. Requisitos

Después de describir el problema, procedemos a definir los requisitos que ha de cumplir la solución que desarrollemos:

- La infraestructura a desarrollar debe estar integrada en la plataforma JdeRobot.
- La infraestructura a desarrollar debe ser eficiente computacionalmente aprovechando los recursos del sistema informático donde se ejecute.
- Se encapsularán en las clases necesarias las operaciones a realizar con Ar.Drone, ofreciendo al programador una interface simple para el manejo del *drone* de bajo nivel que permita ocultar la complejidad del control de vuelo.
- Se definirán aquellas interfaces de comunicación con otros nodos que no estén ya definidas en el entorno JdeRobot. Se reaprovecharán las interfaces ya existentes en JdeRobot para sensores y

actuadores del cuadricóptero, por ejemplo la interface **camera** para la retransmisión de las imágenes capturadas por las cámaras.

- Todos los valores de la aplicación de control que puedan ser configurados, se deberán introducir mediante ficheros XML o con la definición de constantes en las cabeceras de la aplicación de control. Para moldear su comportamiento simplemente cambiado la configuración.
- El control del cuadricóptero deberá ser fluido, de modo que pueda seguir objetos en movimiento a 5 cm/seg.

2.3. Metodología de desarrollo

Dada la naturaleza del proyecto, y al igual que en cualquier otro proyecto de software, es necesario el uso de un modelo que defina el ciclo de vida de la aplicación. Para el desarrollo de este proyecto hemos decidido adoptar el modelo de desarrollo en espiral. Este modelo, definido por Barry Boehm en 1986, se basa en una espiral en la que cada iteración representa un conjunto de actividades. Estas actividades no tienen prioridad, se elegirá en la fase de análisis de riesgos. Así, cada iteración está dividida en las siguientes actividades:

- Determinar objetivos: en esta actividad se definirá el objetivo de la iteración actual. Siguiendo este modelo el objetivo final del proyecto se divide en subobjetivos.
- Análisis de riesgo: en esta actividad, se lleva a cabo varios estudios con el propósito de conocer las posibles amenazas o eventos no deseados que se puedan producir en el objetivo actual.
- Desarrollar y probar: llegados a este punto será necesario la verificación del correcto funcionamiento realizado en la iteración para subsanar los errores y que estos no prosigan en las siguientes iteraciones.
- Planificación: en esta actividad se revisarán las fases anteriores para determinar si se debería continuar con las siguientes actividades.

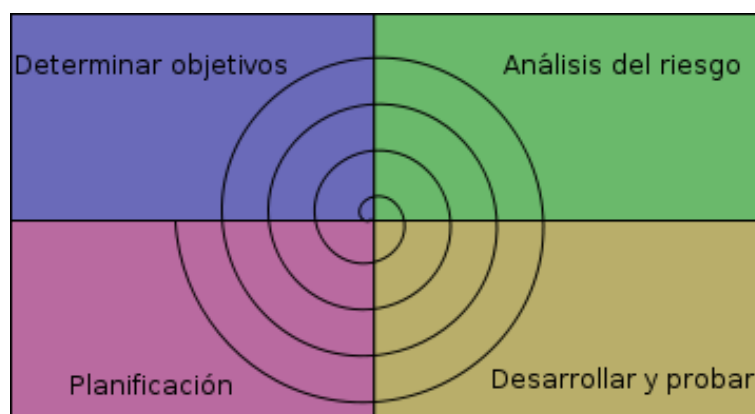


Figura 2.1: Desarrollo en espiral

Para materializar estas actividades e iteraciones mantuvimos reuniones semanales durante todo el desarrollo del trabajo. De este modo, cada semana marcábamos un nuevo subobjetivo. Si el anterior se

había completado, planificábamos el siguiente. Si por el contrario, no sé había completado, ahondábamos en el objetivo actual para corregir los errores o volver a planificarlo.

Durante el transcurso del proyecto se mantuvo una bitacora web¹ donde se publicaban las etapas del proyecto con los éxitos y fracasos obtenidos durante la realización de las pruebas. Además, todo el código fuente generado se mantuvo en un repositorio con un sistema de control de versiones (SVN) accesible desde internet². De este modo el tutor y cualquier otra persona interesada tiene acceso en cualquier momento al código.

2.4. Plan de trabajo

El desarrollo del proyecto ha seguido la siguiente planificación:

1. Desarrollo de la infraestructura

- a) Estudio de la infraestructura del fabricante de Ar.Drone 1.0: para llevar a cabo esta tarea, fue necesario el estudio de la plataforma. Estudiamos las características del *drone*, los usos que le podríamos dar, la plataforma de desarrollo ofrecida por el fabricante así como otras plataformas disponibles. Después del estudio, evaluamos todas las opciones y marcamos los objetivos del proyecto.
- b) Familiarización con el entorno software: una vez definida la plataforma de trabajo, tuvimos que elegir el entorno software donde desarrollaríamos el proyecto. Para ello estudiamos las características de los entornos disponibles, evaluamos las posibilidades y decidimos cuál se aproximaba más a nuestras necesidades. Después de elegir el entorno, fue necesario el estudio de la plataforma JdeRobot y de las librerías que se utilizan en el proyecto como librerías de interfaces gráficas y librerías para análisis de ficheros XML.
- c) Diseño y desarrollo de `ardrone_server`, la aplicación que nos proporciona una capa de acceso a los sensores y actuadores de Ar.Drone 1.0.

2. Navegación visual

- a) Estudio de las herramientas de visión artificial y algoritmos de visión: debido a la necesidad de obtener, analizar y comprender los datos obtenidos por las cámaras del *drone*, tuvimos que estudiar las técnicas de visión artificial necesarias. Haciendo uso de los conocimientos adquiridos sobre las librerías de visión artificial, realizamos ejemplos que nos ayudaron a afianzar los conocimientos. Aprendimos a desarrollar filtros de colores, detección de formas y filtros de suavizado para los objetos reconocidos a seguir durante la navegación.
- b) Estudio de los algoritmos de control: en esta fase estudiamos los distintos algoritmos de control que necesitábamos para la realización del control visual. Para los algoritmos de control fue necesario estudiar los controladores PID, comprender las superficies de control, estudiar el sistema de control a bordo de Ar.Drone 1.0 y realizar numerosas pruebas para comprender el vuelo del *drone*.

¹<http://jderobot.org/Amartinflorido-tfg>

²<https://svn.jderobot.org/users/amartinflorido/tfg/>

- c) Desarrollo del control visual: con todos los conocimientos adquiridos en las etapas anteriores, tuvimos que concentrarlos en crear una aplicación que nos permitiese realizar los experimentos. Esta aplicación, `object_tracking`, implementa el análisis de imágenes y los mecanismos de control reactivo basado en visión.

3. Experimentos y ajustes

- a) Control frontal: en estos experimentos se realizó el seguimiento de un objeto de color detectado en las imágenes proporcionadas por la cámara frontal del *drone* gracias a los algoritmos de visión estudiados en etapas anteriores.
- b) Control ventral: en esta fase tuvimos que profundizar aún más en el estudio del sistema de control de bajo nivel de Ar.Drone, sobre el que trabaja nuestro propio sistema de navegación. Para ello tuvimos que recolectar la información disponible sobre las peculiaridades de dicho sistema de bajo nivel y poder así, comprenderlo en mayor profundidad. Una vez adquiridos los conocimientos necesarios se realizó el seguimiento de un objeto. En este caso el *drone* no sólo tendrá que seguir al objeto, sino que además, tendrá que orientarse respecto a él.

Capítulo 3. Infraestructura

En este capítulo vamos a describir la infraestructura, hardware y software, en la que se apoya el proyecto. En la infraestructura hardware se describirán las características técnicas del cuadricóptero utilizado. A continuación, en la infraestructura software, se detallará cómo se lleva a cabo la comunicación con Ar.Drone y los sistemas de a bordo que éste implementa. Además describiremos otras plataformas que han servido de inspiración para el proyecto y otros componentes software utilizados en los programas desarrollados.

3.1. Parrot Ar.Drone 1.0: hardware y software

Ar.Drone es un VANT desarrollado y comercializado por la empresa Parrot (Figura 3.1). Se trata de un pequeño quadricóptero o helicóptero con cuatro rotores. La primera versión de Ar.Drone fue presentada en la feria Las Vegas International Consumer Electronic Show (CES) en el año 2010. En el año 2012 Parrot lanzó al mercado la segunda versión de Ar.Drone, que incorpora nuevos sensores y es una revisión de hardware de la versión 1.0. El principal uso de Ar.Drone es su teleoperación a través de *smartphones* y *tablets*. Gracias a éstos y a las cámaras incorporadas en Ar.Drone, los programadores pueden crear escenarios de realidad aumentada donde los jugadores pueden librar batallas entre ellos, seguir objetivos o simplemente teleoperar el aparato ofreciendo la posibilidad de grabar el entorno con sus cámaras. Debido a su bajo coste, los sensores que incorpora y los materiales resistentes de los que se compone, Ar.Drone se ha convertido en una plataforma de robótica aérea de bajo coste muy extendida por el mundo.



Figura 3.1: Ar.Drone 1.0

- **Dimensiones:**

- Con casco: 52,5 x 51,5 cm

- Sin casco: 45 x 29 cm

- **Peso:**

- 380 g con la cubierta de exteriores.

- 420 g con la cubierta de interiores.

- **Alcance y velocidad:**

Velocidad máxima: 18 km/h.

Altura máxima: Limitada por el alcance de la conexión Wi-Fi (entre 50 y 120 metros).

Alcance máximo: Limitada por el alcance de la conexión Wi-Fi.

- **Autonomía:**

Autonomía de vuelo: 12 minutos.

Batería de ion de litio de 3 celdas de 1000mAh a 11.1V.

- **Sistema embebido:**

Microprocesador ARM9 RISC de 32 bits a 468 MHz.

Memoria DDR SDRAM de 128 MB a 200 MHz.

Sistema operativo con núcleo Linux.

Wi-Fi b/g.

Conector USB de alta velocidad.

- **Actuadores:**

4 motores sin escobillas, funcionando a 3.500 rpm con una potencia de 15 W.

3.1.1. Sensores

Sensores inerciales

- Acelerómetro de 3 ejes construido con tecnología MEMS.
- Giróscopo de 2 ejes.
- Giróscopo de precisión de 1 eje para el control de guiñada.

Cámara frontal

- Cámara con sensor CMOS de tipo gran angular de lente diagonal. 93 grados de amplitud.
- Resolución 640x480 píxeles (VGA).

Cámara vertical

- Cámara con sensor CMOS de alta velocidad le lente diagonal. 64 grados de amplitud.
- Resolución 176x144 píxeles.
- Es utilizado como sensor adicional de desplazamiento.

Altímetro basado en ultrasonidos

- Frecuencia de emisión: 40 kHz.
- Alcance de 6 metros utilizado para la estabilización vertical.

3.1.2. Comunicación con Ar.Drone

Ar.Drone crea una red inalámbrica en modo ad-hoc a la que cualquier dispositivo que soporte este modo podrá conectarse. Una vez realizada la conexión, el servidor DHCP de Ar.Drone ofrecerá una IP a su cliente. A partir de entonces el cliente estará en condiciones de interactuar con Ar.Drone a través de comandos enviados por la red.



Figura 3.2: Comunicación entre Ar.Drone y PC

La comunicación con Ar.Drone se puede llevar a cabo mediante dos mecanismos; el uso directo de comandos AT (Figura 3.2a) o a través de Ar.Drone SDK (Figura 3.2b), que será explicado más adelante. Con el uso directo de comandos AT es el programador quién se encarga de gestionar toda la comunicación y la codificación de los comandos que se envían al *drone* (Figura 3.2a). Sin embargo, con el uso del SDK de Ar.Drone (ver Figura 3.2b) se utilizarán funciones de alto nivel para realizar estas tareas. Lo gestione el programador o lo realice el SDK de Ar.Drone, existen cuatro vías de comunicación con el *drone* que se utilizan para distintos propósitos:

1. Para el control y la configuración del *drone* se envían comandos AT al puerto 5556 UDP.
2. La información sobre el *drone* (estado, posición, velocidad, etc...) es enviada por Ar.Drone al cliente por el puerto 5554 UDP.
3. El flujo de vídeo se envía al cliente por el puerto 5555 UDP.
4. Para la transmisión de información crítica, como la recuperación de datos de configuración, las comunicaciones se realizan por el puerto 5559 TCP. Esta vía de comunicación es llamada *control port*.

Comandos AT

Un comando AT es una cadena de texto que se envía al Ar.Drone para controlar el movimiento o gestionar la configuración, es un protocolo de bajo nivel, diseñado por el fabricante. La cadena de texto se codifica en caracteres ASCII de 8 bits, con un carácter de salto de línea (*Line Feed*) señalado como <LF>.

Un comando AT consiste en los tres caracteres AT* seguido de un nombre de comando, un símbolo de igualdad, una secuencia de números y opcionalmente, una lista de argumentos separados por comas. Ejemplo:

AT*PCMD=21625,1,0,0,0,0<LF>

PCMD o Progressive Commands es un comando para el movimiento del Ar.Drone.

- 21625 es un número de secuencia.
- 1 activa el uso de comandos progresivos o el modo Combined Yaw.
- 0 valor en coma flotante entre -1 y 1 que indica la inclinación en el eje X (Roll).
- 0 valor en coma flotante entre -1 y 1 que indica la inclinación en el eje Y (Pitch).
- 0 valor en coma flotante entre -1 y 1 que indica la velocidad vertical.
- 0 valor en coma flotante entre -1 y 1 que indica la inclinación en el eje Z (Yaw).

3.1.3. Sistema de control de bajo nivel a bordo

Ar.Drone contiene un sistema de control a bordo que valiéndose de los sensores, es capaz de realizar de forma automática maniobras básicas como el despegue, la estabilización, el aterrizaje y la posibilidad de cernirse en un punto a una altura determinada. Este sistema permite la configuración de ciertos parámetros y la recepción de valores para el movimiento del *drone*. Como vimos en el punto 3.1.2, la configuración se envía al puerto 5559 TCP y los valores para el control del *drone*, al puerto 5556 UDP. La comunicación con este sistema se puede llevar a cabo mediante comandos AT o a través de funciones de alto nivel con el SDK de Ar.Drone.

Existen una gran cantidad de parámetros configurables, a continuación se listan algunos de los más utilizados:

- La altitud máxima
- El límite de la velocidad en *yaw*.
- El límite de la velocidad vertical.
- El ángulo límite de inclinación del Ar.Drone.
- El ángulo límite de inclinación del dispositivo que controla el Ar.Drone (ej: iPhone).

Para el movimiento del *drone* (Figura 1.18) se tendrán que enviar valores para cada uno de los grados de libertad que nos permite Ar.Drone:

- El eje X, movimiento hacia adelante o hacia atrás.
- El eje Y, movimiento hacia la izquierda o hacia la derecha.
- El eje Z, movimiento hacia arriba o hacia abajo.
- La rotación en el eje Z, también conocido como *yaw*.
- La rotación en el eje X (*roll*) y en el eje Y (*pitch*) están combinadas con los movimientos en el eje X e Y por el sistema de control de a bordo, de modo que no es posible su control explícito.

3.1.4. Sistema de vídeo a bordo

Este sistema se encarga de recoger las imágenes de las cámaras, codificarlas y enviarlas al cliente conectado con Ar.Drone, al puerto 5555 UDP. Si el programador no utiliza el SDK de Ar.Drone tendrá que encargarse de la recepción y decodificación de las imágenes. El sistema solamente es capaz de transmitir una imagen cada vez, pero como Ar.Drone dispone de dos cámaras, las imágenes se superponen (Picture in Picture) dividiendo así el flujo en cuatro canales:



Figura 3.3: Canal horizontal



Figura 3.4: Canal vertical



Figura 3.5: Canal vertical sobre frontal



Figura 3.6: Canal frontal sobre vertical

El sistema de vídeo permite utilizar dos *codecs* para la codificación del flujo. Estos son UVLC y P264 que es una implementación para Ar.Drone de H264. Se puede elegir cualquiera de los dos a través de los parámetros de configuración del sistema de control a bordo. Independientemente del *codec* utilizado, el sistema de vídeo envía la imagen al cliente por grupos de bloques (GOB). Cada bloque tiene 16 pixeles de altura. Por tanto la imagen se divide como muestra la siguiente Figura.

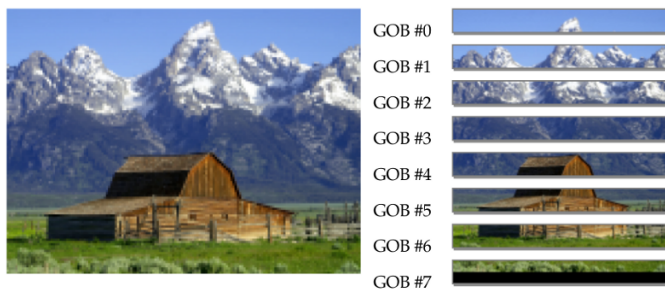


Figura 3.7: Imagen dividida en bloques

A su vez, cada GOB se divide en macrobloques de 16 pixeles de ancho.

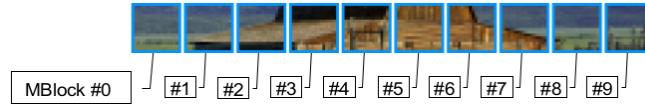


Figura 3.8: Macrobloques de 16x16

Cada macrobloque contiene información de una imagen de 16x16, en el espacio de color $YC_B C_R$.

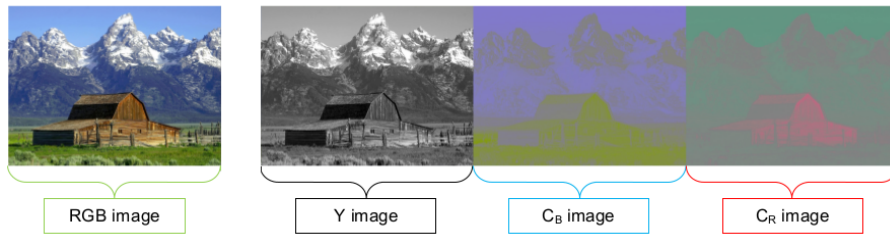


Figura 3.9: Espacio de color $YC_B C_R$

Finalmente cada imagen de 16x16 es almacenada en memoria como 6 bloques de 8x8:

- 4 bloques (Y_0 , Y_1 , Y_2 y Y_3) para formar la imagen Y del componente luma (la imagen en blanco y negro).
- 2 bloques para los componentes chroma (calculados de la imagen original en RGB):
 - C_b : la diferencia de azul en el componente.
 - C_r : la diferencia de rojo en el componente.

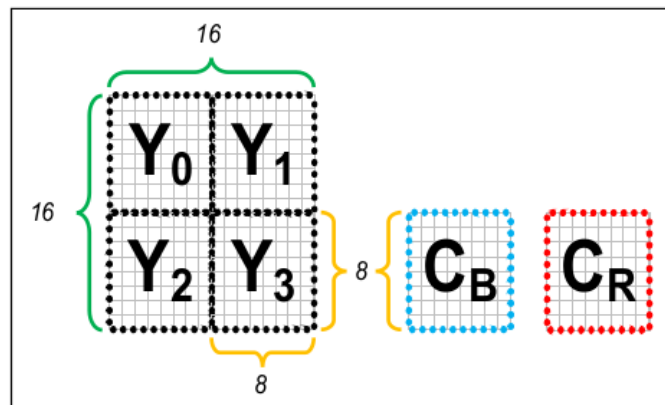


Figura 3.10: Almacenamiento de la imagen en memoria

Detección de etiquetas

Otra característica de este sistema es la capacidad para detectar etiquetas de colores. Todos los kits de Ar.Drone incluyen estas etiquetas que se utilizan para identificar enemigos o compañeros de juego en los

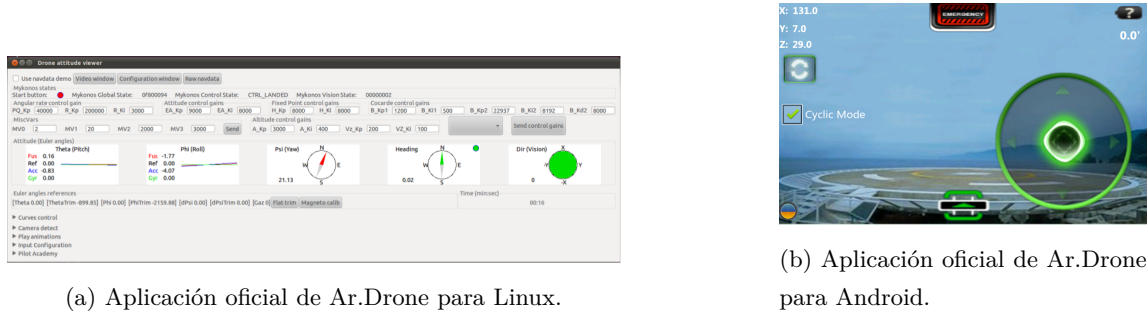
videojuegos de realidad aumentada. Además de las etiquetas, la carcasa de exterior sirve para el mismo propósito. Junto con los datos de navegación, la información referente a las etiquetas detectadas se envía al cliente conectado con Ar.Drone a través del puerto 5554 UDP.



Figura 3.11: Carcasa de exterior y etiqueta identificativa

3.2. Ar.Drone SDK

Es el SDK proporcionado por Parrot para el control de Ar.Drone. Actualmente está disponible para plataformas Linux, iOS y Android, en dispositivos con conexión Wifi. Para otros dispositivos se puede utilizar el protocolo definido en el apartado 3.1.2 y el envío de comandos AT.



(a) Aplicación oficial de Ar.Drone para Linux.

(b) Aplicación oficial de Ar.Drone para Android.

El SDK nos permite una comunicación con Ar.Drone mucho más cómoda que el envío directo de comandos AT. Además, se encarga de decodificar el flujo de vídeo y la información de navegación (navdata). El SDK no permite reescribir el software empotrado en el *drone*. La arquitectura del SDK se compone de varias capas, se encuentra en el dispositivo donde se ha desarrollado la aplicación:

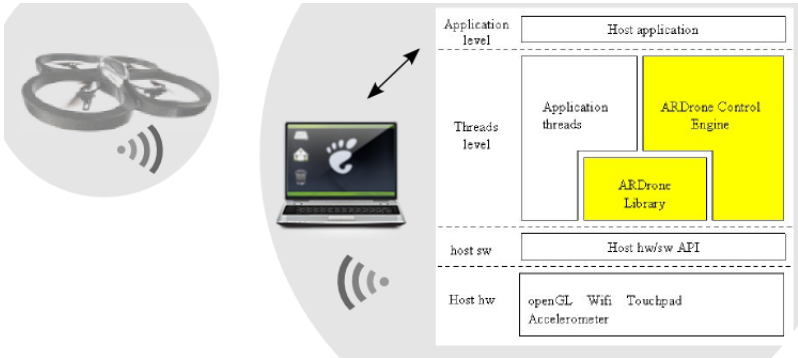


Figura 3.13: Arquitectura de Ar.Drone SDK

3.2.1. Ar.Drone Library

Es una librería de software libre que contiene APIs de alto nivel para acceder al *drone*. La librería se compone de:

SOFT: el código específico del *drone*, incluye:

COMMON: los ficheros de cabeceras (.h) que describen las estructuras usadas en la comunicación con Ar.Drone.

lib/ardrone_tool: es un conjunto de herramientas para el manejo del *drone*, como el envío de comandos AT, el hilo de recepción de la información de navegación y funciones para el uso del *pipeline* de vídeo.

VLIB: es la librería de procesamiento de vídeo. Contiene funciones para la recepción y decodificación del flujo de vídeo.

VPSDK: es un conjunto de librerías de propósito general, incluye:

VPSTAGES: el procesamiento de piezas de vídeo, que podremos ensamblar para construir un *pipeline* de procesamiento de vídeo.

VPOS: envoltorios multiplataforma para funciones del sistema de bajo nivel (manejo de hilos, gestión de memoria, etc.).

VPCOM: envoltorios multiplataforma para funciones de comunicación (Wifi).

VPAPI: funciones auxiliares para el manejo del *pipeline* de vídeo y los hilos.

A continuación podemos ver la función equivalente del comando AT PCMD visto en la sección 3.1.2 del SDK de Ar.Drone:

```
void ardrone_at_set_progress_cmd(enable,phi,theta,gaz,yaw);
```

ardrone_at_set_progress_cmd es una función para el envío de comandos de movimiento a Ar.Drone.

- **enable**, con su valor a 0 desactiva los comandos progresivos (Ar.Drone entra en el modo en el que se cierne en un punto). Si su valor es 1, activa los comandos progresivos.
- **phi**, 0 valor en coma flotante entre -1 y 1 que indica la inclinación en el eje X (Roll).
- **theta**, 0 valor en coma flotante entre -1 y 1 que indica la inclinación en el eje Y (Pitch).
- **gaz**, valor en coma flotante entre -1 y 1 que indica la velocidad vertical.
- **yaw**, valor en coma flotante entre -1 y 1 que indica la inclinación en el eje Z (Yaw).

Ar.Drone Tool forma parte de Ar.Drone Library, e implementa los cuatro servicios que ofrece Ar.Drone. Ar.Drone Tool ofrece:

- Un hilo para la administración de los comandos AT, se encarga de recolectar todos los comandos enviados por otros hilos y los envía de manera ordenada con los números de secuencia correctos.

- Un hilo para la administración del **navdata** (información de vuelo) que automáticamente recibe el flujo de **navdata**, lo descodifica y lo pone a disposición de la aplicación cliente mediante *callbacks*.
- Un hilo para la administración de vídeo, que recibe el flujo de vídeo y lo pone a disposición de la aplicación cliente mediante *callbacks*.
- Un hilo de control que maneja las peticiones de otros hilos para el envío de comandos seguros.

Todos estos hilos se crean y se inician por la función **main**, que se encuentra en **ardrone_tool.c** dentro de ARDroneTool.

3.2.2. Ar.Drone Control Engine

Esta librería sólo está disponible para los sistemas iOS y contiene todo lo necesario para el manejo de Ar.Drone, la recepción de vídeo y el manejo de los sensores de inclinación y eventos especiales de iOS.



Figura 3.14: Aplicación oficial de Ar.Drone para iOS

3.3. ROS y los paquetes de Ar.Drone

ROS¹ (Robot Operating System) es un entorno para el desarrollo de software para robots. Fue desarrollado por el laboratorio de inteligencia artificial de Stanford en 2007. Desde 2008 el desarrollo lo continuó Willow Garage².

ROS ofrece los servicios estándar de un sistema operativo como la abstracción del hardware, el control de dispositivos a bajo nivel, el paso de mensajes entre procesos y el mantenimiento de paquetes de funcionalidad común. Su arquitectura está basada en grafos, donde el procesamiento se realiza en nodos. Éstos pueden mandar y recibir mensajes entre ellos, desde información sobre los sensores hasta planificadores de rutas. Para cualquier aplicación desarrollada con ROS se necesitan como mínimo un nodo central, que es el encargado de la coordinación entre los demás nodos, y al menos un nodo más que contendrá la lógica de la aplicación. En un entorno típico habrá tres nodos; el nodo central, un nodo que publica algún tipo de información (posición del robot, datos de los sensores, etc.) u ofrece algún servicio, como por ejemplo el manejo del robot, y un último nodo que recogerá estos datos y podrá comandar órdenes que provocarán algún efecto en el robot.

¹<http://www.ros.org>

²<http://www.willowgarage.com/>

3.3.1. ardrone_brown

`ardrone_brown`³ es un paquete para ROS desarrollado por el departamento de robótica de Brown University⁴ para el manejo de Ar.Drone. Es un envoltorio del Ar.Drone SDK que permite integrarlo en la plataforma ROS. Este paquete está disponible únicamente para la versión 1.0 de Ar.Drone y actualmente el proyecto se encuentra abandonado.

Este paquete ofrece servicios para el manejo de Ar.Drone así como para maniobras básicas como el despegue, el aterrizaje y en envío de señal de emergencia, donde el *drone* interrumpirá por completo su funcionamiento. Además, publica las imágenes capturadas por sus cámaras y los datos de navegación (navdata). En concreto `ardrone_brown` ofrece los siguientes servicios:

- **Servicios publicados**, un cliente puede enviar mensajes a estos servicios para que tengan algún efecto en el *drone*.
 - `/cmd/vel`: es el encargado de controlar el movimiento de Ar.Drone. El cliente tendrá que enviar un mensaje *Twist*. Este mensaje contiene una estructura con las velocidades lineales en los ejes x,y,z y la velocidad angular en el eje z.
 - `/ardrone/takeoff`: cuando este servicio reciba un mensaje del tipo *std_msgs/Empty* indicará a Ar.Drone que tiene que despegar. Un mensaje *std_msgs/Empty* es un mensaje vacío.
 - `/ardrone/land`: cuando este servicio reciba un mensaje del tipo *std_msgs/Empty* indicará a Ar.Drone que tiene que aterrizar.
 - `/ardrone/reset`: cuando este servicio reciba un mensaje del tipo *std_msgs/Empty* indicará a Ar.Drone que tiene que abortar el vuelo, esto provoca la caída de Ar.Drone desde el aire.
- **Servicios suscritos**, un cliente se puede subscribir a estos servicios y el nodo de `ardrone_brown` enviará una copia de los datos a los clientes suscritos cuando éstos se actualicen.
 - `/ardrone/image_raw`: cada vez que Ar.Drone capture una nueva imagen, `ardrone_brown` enviará a sus clientes suscritos un mensaje del tipo *sensor_msgs/Image*. Este mensaje contiene una marca de tiempo, la imagen capturada por el *drone*, ancho y alto de la imagen, la codificación de los pixeles, la longitud total de una fila en bytes (*step*) y si la organización de los bytes es *bigendian* o no.
 - `/ardrone/navdata`: cuando se actualicen los datos de navegación, `ardrone_brown` enviará a sus clientes un mensaje del tipo *std_msgs/Header*. Este mensaje consiste en una serie de campos con un tipo y un nombre, separados por comas. En este mensaje `ardrone_brown` incluye la siguiente información: el porcentaje de la batería, la rotación en grados de los ejes X,Y,Z la altitud estimada en centímetros, las velocidades lineales en los ejes X,Y,Z expresados en milímetros por segundo y una marca de tiempo.

³https://code.google.com/p/brown-ros-pkg/wiki/ardrone_brown

⁴<http://brown-robotics.org/wp/>

3.3.2. ardrone_autonomy

`ardrone_autonomy`⁵ es el fruto del trabajo de Mani Monajjemi que desarrolló en el laboratorio de robótica de Simon Fraser University⁶, cuando `ardrone_brown` fue abandonado. Este nuevo envoltorio para ROS ofrece los mismos servicios y funcionalidades de `ardrone_brown`, pero además soporta Ar.Drone 2.0 e incorpora nuevas características de ROS como los marcos de coordenadas, para poder realizar transformaciones geométricas o la calibración del sistema de guía inercial, la posibilidad de realizar animaciones con los leds del *drone* o el nuevo conjunto de maniobras de vuelo acrobáticas (flights animations) introducidas en Ar.Drone 2.0. Además de los servicios ofrecidos por su antecesor, `ardrone_brown`, `ardrone_autonomy` ofrece nuevos servicios:

■ Servicios publicados

/ardrone/togglecaml: cuando este servicio reciba un mensaje del tipo *std_msgs/Empty* cambiará el flujo de vídeo por defecto.

/ardrone/setcamchannel: este servicio recibe un entero sin signo de 8 bits como parámetro que indica el canal que debe utilizar la cámara. Para Ar.Drone 1.0 este parámetro puede tener cuatro valores y para Ar.Drone 2.0 solamente dos.

/ardrone/setledanimation: este servicio recibe 3 parámetros. Un entero sin signo de 8 bits para elegir el tipo de animación de leds (14 tipos distintos), un valor en coma flotante para la frecuencia de la animación y un entero sin signo de 8 bits para la duración de la animación.

/ardrone/setflightanimation: este servicio recibe 2 parámetros. Un entero sin signo para el tipo de animación de vuelo predefinida (20 tipos distintos) y un entero sin signo de 16 bits para la duración.

/ardrone/imu_recalib: este servicio recibe un parámetro. Un booleano que indica si tiene que recalibrar la unidad de medida inercial (IMU).

/ardrone/flattrim: cuando este servicio reciba un mensaje del tipo *std_msgs/Empty*, `ardrone_autonomy` recalibrará la estimación de la rotación asumiendo que se encuentra en una superficie plana.

■ Servicios suscritos

/ardrone/front/image: este servicio envía a sus clientes la imagen de la cámara frontal, con los mismos datos que vimos en el paquete `ardrone_brown`.

/ardrone/bottom/image: este servicio envía a sus clientes la imagen de la cámara vertical, con los mismos datos que vimos en el paquete `ardrone_brown`.

/ardrone/navdata: envía los mismos campos que el paquete `ardrone_brown` pero además incluye los datos de los nuevos sensores. Estos datos son: las lecturas del magnetómetro, la temperatura, la presión, la velocidad y ángulo del viento y la velocidad lineal para los ejes X,Y,Z. Además se puede ampliar estos datos con los datos del sistema de detección de etiquetas que nos proporciona: el número de etiquetas detectadas, los tipos de las etiquetas, un vector de posición y el tamaño de cada etiqueta y su orientación.

⁵https://github.com/AutonomyLab/ardrone_autonomy

⁶<http://robotics.sfu.ca/>

3.4. JdeRobot

JdeRobot⁷ es una plataforma de software libre para el desarrollo de aplicaciones con robots, domótica y visión artificial. Este proyecto ha sido desarrollado por el grupo de robótica de la Universidad Rey Juan Carlos. La idea original surge como resultado de una tesis doctoral [4] y el comienzo de su desarrollo data del año 2003.

JdeRobot abstrae al programador del acceso a los dispositivos hardware, como la lectura de los sensores o el envío de comandos a los motores. Principalmente está desarrollado en C y C++, aunque existen componentes desarrollados en otros lenguajes como Java o Python. Ofrece un entorno de programación basado en componentes donde la aplicación está formada por varios componentes y cada componente se ejecuta como un proceso. JdeRobot ofrece una interfaz sencilla para la programación de sistemas de tiempo real y resuelve problemas relacionados con la sincronización de los procesos y la adquisición de datos. JdeRobot simplifica el acceso a los dispositivos hardware desde el componente, permitiendo obtener la lectura de un sensor a través de una interfaz ICE. Normalmente en las aplicaciones robóticas desarrolladas con JdeRobot, los componentes necesitan datos sensoriales y envían ordenes a los actuadores. Los datos sensoriales llegan al componente a través de interfaces ICE. Los *drivers* encargados de controlar los robots también reciben las órdenes a través de interfaces ICE. Gracias a su arquitectura de componentes y a las interfaces ICE, JdeRobot nos ofrece la oportunidad de crear aplicaciones muy complejas a partir de componentes más simples que han sido desarrollados para realizar una tarea específica. En un momento dado podemos encontrar componentes ejecutándose en distintas máquinas interoperando a través de interfaces ICE, incluso estando escritas en lenguajes distintos.

Actualmente JdeRobot soporta un gran variedad de dispositivos como el robot Pioneer de MobileRobotics Inc., el humanoide NAO de Aldebaran Robotics, el robot turtlebot de Willow Garage, cámaras firewire, USB e IP, los escáneres laser LMS de SICK y URG de Hokuyo, los simuladores Stage y Gazebo, sensores de profundidad como kinect y otros dispositivos X10 de domótica. Además, tiene soporte para software externo como OpenGL, GTK, XForms, Player, GSL y OpenCV. Es un proyecto de software libre licenciado como GPLv3⁸, actualmente se encuentra en la versión 5.1.

Una de las partes principales de este Trabajo Fin de Grado ha sido el desarrollo del componente `ardrone_server`, que permite el uso del cuadricóptero desde otros componentes de JdeRobot. Este componente es un envoltorio del Ar.Drone SDK, que nos permite el manejo del Ar.Drone 1.0 y el procesamiento de imágenes obtenidas a partir éste. En `ardrone_server` se ha utilizado ICE para desarrollar la interface Ar.Drone, encargada de ofrecer el control del *drone* y los datos generados por éste, a los distintos clientes de la red. También se han reutilizado interfaces como `camera` para el envío de imágenes y `remoteConfig`, para la configuración en caliente del Ar.Drone.

La interface `remoteConfig` de JdeRobot nos permite leer un fichero XML proveniente desde la red para su posterior escritura en disco o para la lectura de los datos que contenga. En `ardrone_server` a través

⁷<http://jderobot.org/>

⁸<http://www.gnu.org/licenses/gpl-3.0-standalone.html>

de este fichero se reciben los parámetros de configuración que tenían que ser modificados en Ar.Drone. Con la interface `camera` de JdeRobot es posible transmitir a través de la red imágenes capturadas por una cámara así como los datos de ésta, como por ejemplo, su nombre, su distancia focal o la posición x,y,z en la que se encuentra.

3.5. Bibliotecas externas

3.5.1. libxml++

libxml++⁹ es un envoltorio escrito en C++ de la librería libxml para el tratamiento de ficheros XML. Es famosa por su alto rendimiento y por el estricto cumplimiento de las especificaciones del estándar. Tiene versiones para muchas plataformas, entre las que se encuentran Windows y Linux. Soporta tres modos de funcionamiento. El análisis de ficheros mediante DOM, SAX y TextReader, que es un método derivado de SAX donde se pueden implementar controladores para partes específicas del documento, evitando así tener que procesar todos los nodos del fichero.

Como veremos más adelante, en el componente JdeRobot que hemos desarrollado se ha utilizado libxml++ en dos puntos de la aplicación. Para el análisis del fichero de configuración de `ardrone_server`, donde se especifican los datos necesarios para la creación de las interfaces que ofrece. Y donde se especifican algunos parámetros como la altura máxima de vuelo o la cámara que utilizará por defecto. Esta última tarea se realiza en el arranque de la aplicación de navegación visual, para establecer los valores por defecto, y cuando se recibe el fichero de configuración a través de la interfaz `remoteConfig`.

3.5.2. ICE

ICE (Internet Communications Engine), es un *middleware* orientado a objetos que provee llamadas a procedimientos remotos, computación grid y funcionalidad cliente/servidor desarrollada por ZeroC¹⁰ bajo una doble licencia GNU GPL y una licencia propietaria. Está disponible para C++, Java, lenguajes .Net, Objective-C, Python, PHP y Ruby, en la mayoría de los sistemas operativos. También existe una variante para teléfonos móviles denominada Ice-e.

ICE nos permite desarrollar aplicaciones distribuidas con el mínimo esfuerzo, abstrayendo al programador de interactuar con las interfaces de red a bajo nivel. El proceso de desarrollo de la aplicación solamente debe centrarse en la lógica de ésta y no en las peculiaridades de la red. Es un *middleware* multiplataforma y multilenguaje, de este modo, podemos implementar clientes y servidores en distintos lenguajes de programación y en distintas plataformas. ICE trabaja con objetos distribuidos, de tal manera que dos objetos dentro de nuestra aplicación no tienen porque estar ejecutándose en la misma máquina. Los objetos pueden estar en distintas máquinas y comunicarse a través de red gracias al envío de mensajes entre ambos.

⁹<http://libxmlplusplus.sourceforge.net/>

¹⁰<http://www.zeroc.com/>

JdeRobot utiliza ICE para las comunicaciones entre sus nodos, por tanto, la tarea de leer valores de un sensor o comandar órdenes a un robot es tan sencilla como ejecutar un método de un objeto de la aplicación. Una ventaja significativa es la posibilidad de poder desarrollar aplicaciones independientes del contexto. Un programador puede desarrollar un *driver* en C++ para un determinado robot que se encuentre embebido en el propio robot, por otro lado, otro programador puede desarrollar una aplicación para el procesamiento de imágenes en Python que se ejecute en un PC. Gracias a ICE podemos utilizar estas dos aplicaciones, que en un principio eran independientes, como una única aplicación sin tener que preocuparnos por las comunicaciones a bajo nivel. Con esto conseguimos desarrollar aplicaciones modulares de una mayor complejidad sin esfuerzo adicional.

3.5.3. OpenCV

OpenCV¹¹ (*Open Source Computer Vision Library*) es una librería de visión artificial desarrollada por Intel, y ahora mantenida por Willow Garage. Se encuentra bajo una licencia BSD, lo que permite utilizarla libremente para propósitos comerciales. Esto ha supuesto que sea comúnmente utilizada para todo tipo de proyectos, desde sistemas de vigilancia con detección de movimientos, reconocimiento de objetos e incluso en el Gran Desafío DARPA¹² y otros proyectos de investigación.

Está escrita en C++, es multiplataforma y contiene interfaces para C, C++, Java, Python y MATLAB. En el 2010 se desarrolló una interface para las GPUs basadas en CUDA y en 2012 para OpenCL. Al haber sido desarrollada por Intel ofrece el sistema de primitivas de rendimiento integradas de Intel, que son un conjunto de rutinas de bajo nivel específicas para procesadores Intel (IPP).

En nuestro proyecto hemos utilizado OpenCV para dos propósitos. En `ardrone_server` para la recepción y el tratado de las imágenes obtenidas del flujo de vídeo de Ar.Drone. Y, en la aplicación de navegación visual, para el desarrollo de un filtro de color que nos permite localizar objetos de un determinado color y así conocer detalles físicos de éste, como el área que ocupa en pantalla o las coordenadas x,y del centro del objeto.

3.5.4. Qt

Qt¹³ es un entorno para el desarrollo de aplicaciones con interfaz gráfica de usuario (GUI) multiplataforma. Las aplicaciones Qt se desarrollan en C++. Una peculiaridad de este entorno es que hace uso de un generador de código (Meta Object Compiler) junto con varios macros para, entre otras cosas, ofrecer la posibilidad de generar código C++ a partir de un fichero XML que define la interfaz gráfica. Aunque normalmente se utiliza Qt para el desarrollo de aplicaciones con interfaz gráfica, también se puede utilizar para desarrollar otro tipo de aplicaciones. Qt proporciona acceso a bases de datos SQL, análisis de ficheros XML, gestión de hilos, soporte para redes, soporte para la internacionalización y un API unificada para el manejo de ficheros multiplataforma. Qt está disponible bajo una licencia comercial,

¹¹<http://opencv.willowgarage.com/wiki/>

¹²<http://archive.darpa.mil/grandchallenge05/>

¹³<http://qt-project.org/>

GPL v3 y LGPL v2.

Para el proyecto hemos utilizado Qt en el desarrollo de las interfaces gráficas para las aplicaciones de navegación visual y la aplicación para el manejo y visionado de los datos sensoriales de VANTs.

3.5.5. Qwt

Qwt¹⁴ (*Qt Widgets for Technical Applications*) es una librería para Qt que contiene componentes gráficos principalmente para su uso en aplicaciones técnicas. Entre los componentes que se pueden encontrar se encuentran los componentes para la generación de gráficas, histogramas, espectrogramas, termos o diales. Está distribuida bajo una licencia Qwt License en la versión 1.0. Actualmente el proyecto se encuentra en su versión 6.1. Se puede encontrar en los repositorios de las distribuciones Linux más famosas o descargar su código fuente para la compilación en plataformas Linux o Windows.

Para el proyecto hemos utilizado Qwt para la visualización de los datos obtenidos de los sensores de Ar.Drone como el *pitch* y *roll*, para el horizonte artificial, el visor de la velocidad o la batería.

¹⁴<http://qwt.sourceforge.net/>

Capítulo 4. Desarrollo de la infraestructura para Ar.Drone

En este capítulo describiremos los componentes software desarrollados para el proyecto. Primero el componente encargado del acceso a los sensores y actuadores del cuadricóptero desde componentes software de JdeRobot y segundo la aplicación para la monitorización y teleoperación de VANT.

4.1. Componente *driver*: `ardrone_server`

`ardrone_server` es un envoltorio que encapsula el SDK de Ar.Drone dentro de un componente de JdeRobot. Está basado en la versión 1.7 del SDK y actualmente soporta Ar.Drone 1.0. Al igual que los proyectos que sirvieron de inspiración para la creación de este componente, `ardrone_brown` y `ardrone_autonomy`, `ardrone_server` está desarrollado en C++.

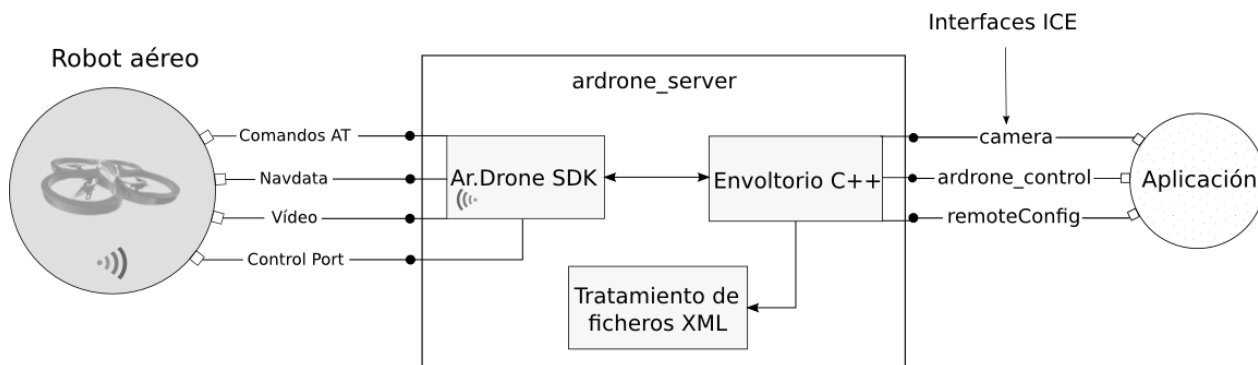


Figura 4.1: Esquema `ardrone_server`

Como muestra la Figura 4.1, `ardrone_server` está compuesto por tres módulos principales: (1) Ar.Drone SDK, que es el SDK oficial para la comunicación con el *drone* a través de una red inalámbrica. (2) El envoltorio C++ que son un conjunto de clases para la interacción con el SDK de Ar.Drone y para la comunicación de `ardrone_server` con otras aplicaciones a través de las interfaces ICE. (3) El tratamiento de ficheros XML, que es un módulo para la gestión de la configuración de `ardrone_server`. El conjunto de estos módulos forman `ardrone_server` que es un *driver* desarrollado para JdeRobot encargado de proveer el acceso a los sensores y actuadores de Ar.Drone desde otros componentes de JdeRobot. Gracias a las interfaces ICE, un componente podrá conectarse a las interfaces que proporciona `ardrone_server` para

interactuar con Ar.Drone, incluso si dicho componente se está ejecutando en un ordenador distinto de donde se esté ejecutando `ardrone_server` y si esta escrito en otro lenguaje como Java o Python.

4.1.1. Interfaces ICE, entradas y salidas

`ardrone_server` ofrece el acceso a sensores y actuadores del cuadricóptero a través de 3 interfaces ICE que se detallan a continuación.

camera

Es el interfaz estandarizado en JdeRobot para el envío y recepción de imágenes, fotograma a fotograma. En `ardrone_server` se ha implementado esta interface modificando ligeramente el comportamiento del SDK nativo. El SDK de Ar.Drone nos permite obtener uno de los cuatro canales de vídeo que ofrece tal y como se contó en el punto 3.1.4. Las imágenes que podemos obtener del canal seleccionado tienen siempre el mismo tamaño, independientemente del canal. Esto supone que cuando queramos obtener la imagen de la cámara vertical, que es de un tamaño inferior al de la cámara frontal, en realidad obtendremos una imagen que en su esquina superior izquierda contiene la imagen de la cámara vertical. Para poder ofrecer la imagen vertical con el tamaño real de ésta, utilizamos lo que en visión artificial se conoce como ROI (*Region Of Interest*): cuando obtenemos el flujo de vídeo generado por Ar.Drone y si la cámara seleccionada es la cámara vertical, recortamos de la imagen obtenida la imagen de la cámara vertical para después poder transmitirla por la red.

La interface `camera` tiene los siguientes campos:

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

```

    float roll;
};

/**
 * Camera interface
 */
interface Camera extends ImageProvider
{
    idempotent CameraDescription getCameraDescription();
    int setCameraDescription(CameraDescription description);

    string startCameraStreaming();

    void stopCameraStreaming();

};

}; /*module*/

```

El código anterior es la definición de la interface **camera** en el lenguaje **slice** (Specification Language for ICE) que se materializa luego en C++, Python, Java,... según el lenguaje del componente que usa esa interfaz. Esta interface contiene una clase para los datos de la cámara como su nombre, la URI donde se está retransmitiendo el vídeo o las distancias focales en los ejes X e Y. La interface **camera** implementa la interface **ImageProvider** que se encuentra definida dentro de la definición de **image**.

```

module jderobot{
    /**
     * Static description of the image source.
     */
    class ImageDescription
    {
        int width; /** Image width [pixels] */
        int height; /** Image height [pixels] */
        int size; /** Image size [bytes] */
        string format; /** Image format string */
    };

    /**
     * A single image served as a sequence of bytes
     */
    class ImageData
    {
        Time timeStamp; /** TimeStamp of Data */
        ImageDescription description; /** ImageDescription of Data */
    };
};

```

```

    ByteSeq pixelData; /** The image data itself.*/
};

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

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

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

}; //module

```

En este fichero de descripción encontramos la definición para la descripción de la imagen (**ImageDescription**) que contiene la meta-información de la imagen. La clase **ImageData** se compone de una descripción de la imagen, una marca de tiempo y la secuencia de bytes que componen la imagen. Las interfaces **ImageConsumer** e **ImageProvider** se encargan de la comunicación con los clientes.

Al ofertarse esta interface como un servicio, varios clientes pueden estar solicitando una imagen en un mismo periodo de tiempo, por ello la interface **camera** dispone de una cola donde se encolan las peticiones de imágenes para después ir sirviéndolas según el orden de llegada.

En **ardrone_server** el hilo que gestiona esta interface tiene un bucle de control, que funciona a 20Hz, donde captura la imagen en función de la cámara activa, rellena los datos de la cámara y retransmite la imagen a los clientes.

ardrone_control

Esta interface tuvimos que desarrollarla para ofrecer el control de vuelo, maniobras básicas y datos de navegación a los componentes cliente de **ardrone_server**. La interface ofrece la posibilidad de comandar maniobras básicas al *drone*, como el despegue, el aterrizaje o de activar el modo de emergencia, que hará que el Ar.Drone se desplome desde el aire. También ofrece el envío de comandos progresivos para el movimiento del *drone*, concretamente ofrece el envío de valores para los ejes X,Y,Z y para la rotación del eje Z (yaw) que se interpretan como órdenes de movimiento.

Aunque la interface formalmente también ofrece la posibilidad de enviar valores para la rotación de los ejes X e Y, el SDK del fabricante no nos permite modificar estos ejes, de modo que esos valores realmente los utilizamos para poder activar o desactivar el modo “cernirse en un punto”. Este modo hace que el *drone* permanezca inmóvil, en la medida de lo posible, a un punto concreto a una altura determinada del suelo.

A continuación se puede ver la parte de la definición de **ardrone_control** para las velocidades de Ar.Drone (lineal y angular) y las maniobras básicas.

```
...
//Values between -1.0 and 1.0
struct LinearVelocity{
    double x;
    double y;
    double z;
};
//Values between -1.0 and 1.0
struct AngularVelocity{
    double x;
    double y;
    double z;
};
struct Velocities{
    LinearVelocity linear;
    AngularVelocity angular;
};
interface ArDrone{
    Navdata getNavdata();
    void cmdVel(Velocities cmd);
    void toggleCam();
    void land();
    void takeoff();
    void reset();
};
```

Además de ofrecer el control de movimiento del Ar.Drone, **ardrone_control** ofrece a sus clientes los datos

obtenidos de los sensores del *drone*. Estos datos se conocen como **navdata** y contienen, entre otros datos, el porcentaje de carga de la batería, la altitud estimada, la rotación y velocidad medidas en los ejes X,Y,Z o las etiquetas detectadas por el sistema de vídeo de a bordo.

Aunque **ardrone_server** ha sido diseñado para Ar.Drone 1.0, la interface **ardrone_control** se ha definido para soportar la versión 1.0 y 2.0 de Ar.Drone, añadiendo al **navdata** datos como la temperatura o la velocidad del viento. Estos datos se obtienen de los nuevos sensores incorporados en Ar.Drone 2.0. En el siguiente listado se puede observar la definición del **navdata** en **ardrone_control**.

```
...
struct Navdata {
    int state;
    //The remaing charge of battery %
    float batteryPercent;
    //Magnetometer ArDrone 2.0
    int magX;
    int magY;
    int magZ;
    //Barometer ArDrone 2.0
    int pressure;
    //Temperature sensor ArDrone 2.0
    int temp;
    //Estimated wind speed ArDrone 2.0
    float windSpeed;
    float windAngle;
    float windCompAngle;
    //rotation about the X,Y,Z axis
    float rotX;
    float rotY;
    float rotZ;
    //Estimated altitude (mm)
    int altd;
    //Linear velocity X,Y,Z (mm/sec)
    float vx;
    float vy;
    float vz;
    //linear accelerations
    float ax;
    float ay;
    float az;
    //Tags in Vision Detectoion
    int tagsCount;
    arrayInt tagsType;
    arrayInt tagsXc;
```

```

    arrayInt tagsYc;
    arrayInt tagsWidth;
    arrayInt tagsHeight;
    arrayFloat tagsOrientation;
    arrayFloat tagsDistance;
    //time stamp
    float tm;
};
...

```

Debido a la naturaleza concurrente de la aplicación, todos los accesos a las variables booleanas para las maniobras básicas así como las funciones encargadas de enviar nuevos comandos al *drone*, están protegidas mediante semáforos. Al igual que la interface `camera`, la interfaz `ardrone_control` está gestionada por un hilo propio que contiene un bucle de control. Este bucle de control funciona a 20 Hz.

remoteConfig

Esta interface desarrollada por el equipo de JdeRobot antes de este proyecto, permite el envío y recepción de ficheros XML de configuración a través de la red. Con esta interface `ardrone_server` ofrece la posibilidad de cambiar valores de la configuración del *drone* en caliente. Dicho de otro modo, podremos variar el comportamiento del *drone* en pleno vuelo. Así, si un componente cliente de `ardrone_server` quiere modificar la altitud máxima permitida por el *drone* en pleno vuelo, simplemente tendrá que enviarle el fichero de configuración indicando el nuevo valor. Cuando `ardrone_server` reciba este fichero, solicitará al *drone* el cambio de altitud máxima.

```

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

```

A medida que la interface reciba los fragmentos del ficheros a través de la red, los irá almacenando en un fichero en disco. En un momento dado, `remoteConfig` puede recibir a través de la red varios ficheros de configuración. Para poder discernir entre todos ellos, cuando se inicializa la transferencia del fichero, se crea un número aleatorio por cada fichero. De este modo se pueden recibir varios ficheros simultáneamente y así aplicar la configuración correctamente.

4.1.2. Diseño

Aunque `ardrone_server` ha sido desarrollado en C++, utiliza algunas funciones contenidas en ficheros de C. Estos ficheros a su vez se comunican con el SDK de Ar.Drone que se encuentra escrito en C. La Figura 4.2 muestra el diagrama empleado en `ardrone_server`, en él conviven clases de C++, interfaces ICE y ficheros con funciones en C, por lo que no se trata de un diagrama de clases estricto.

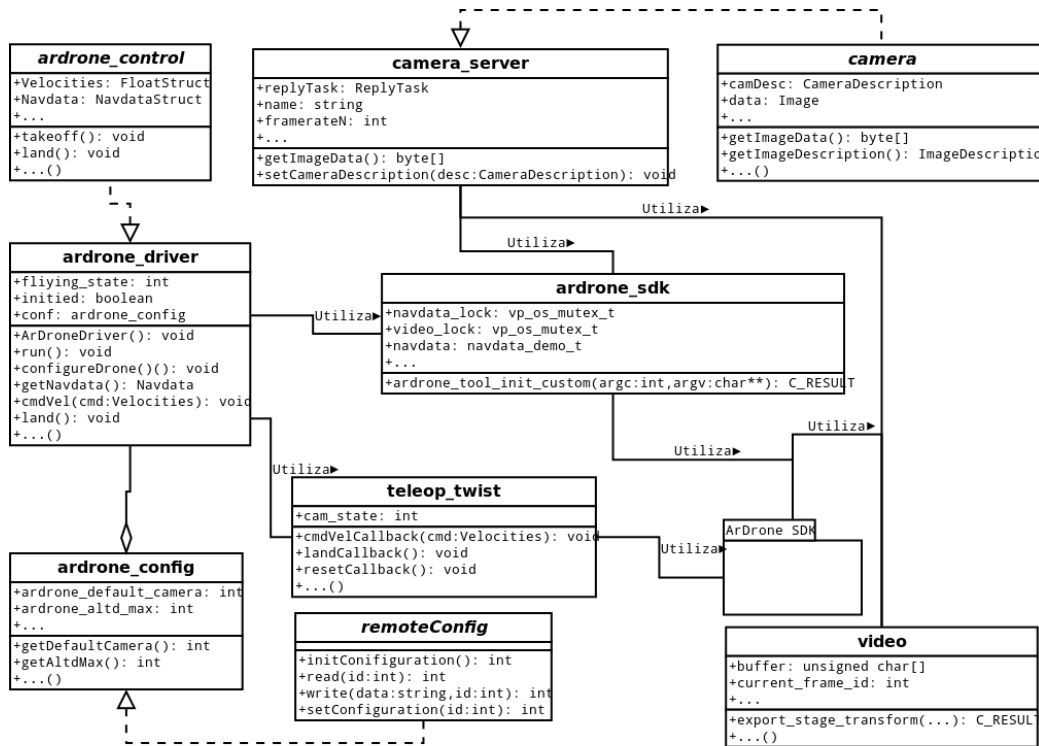


Figura 4.2: Diagrama de `ardrone_server`

El punto de inicio del diagrama de la Figura 4.2 es `ardrone_driver`. Esta clase de C++ implementa la interface ICE `ardrone_control` que ofrece a los clientes de `ardrone_server` la posibilidad de controlar el *drone* y de obtener los datos de sus sensores. También contiene un objeto de la clase `ardrone_config`, este objeto implementa la interface ICE `remoteConfig` que recibirá a través de la red los ficheros XML de configuración. `ardrone_driver` utiliza las *callbacks* definidas en `teleop_twist` para comandar las órdenes y obtener los datos de los sensores de Ar.Drone a través del SDK de Ar.Drone. El hilo que gestiona esta clase se crea en `ardrone_sdk`.

`camera_server` es la clase que implementa la interface ICE `camera` y es la encargada de proporcionar a los clientes las imágenes obtenidas de Ar.Drone. Esta clase utiliza las funciones definidas en `video` para la obtención de las imágenes desde los hilos del SDK de Ar.Drone y es en `ardrone_sdk` donde se crea el hilo que gestiona esta interface.

Tal y como muestra la Figura 4.3 `ardrone_server` es un componente multihilo. En primer lugar tiene los hilos generados por el SDK de Ar.Drone para la recepción y decodificación de vídeo, la recepción y decodificación de los datos de navegación, el hilo de control para los comandos AT y el hilo árbitro que se encarga de coordinar todas las peticiones de los demás hilos. Además hemos añadido otros dos hilos adicionales. Primero un hilo para la administración de la interface `camera` y segundo otro para la administración de las interfaces `ardrone_control`, `remoteConfig` y la inicialización de todo el componente. Para mantener la coherencia y evitar condiciones de carrera, todos los recursos compartidos están protegidos mediante semáforos.

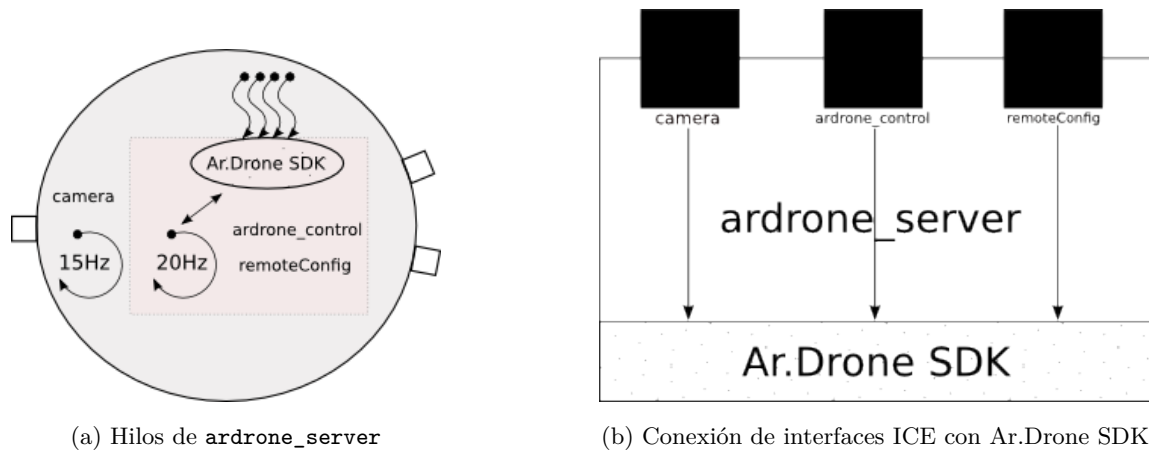


Figura 4.3: Esquemas `ardrone_server`

Del párrafo anterior se puede apreciar que cada hilo tiene una tarea. Cada tarea no tiene la misma carga de trabajo. Así, mientras el hilo encargado de la interface `camera` tiene que recuperar la imagen, procesarla y retransmitirla a todos los clientes que tenga en su cola, el hilo encargado de gestionar las interfaces `remoteConfig` y `ardrone_control`, sólomente tendrá enviar los valores de los comandos de movimiento y de vez en cuando, leer un fichero XML y enviar la configuración al *drone*. Por este motivo los tiempos de los bucles de control de cada hilo son diferentes. El hilo de la interfaz `camera` tiene un ritmo de trabajo menor debido a su carga de trabajo y el otro hilo encargado de gestionar las demás interfaces tiene un ritmo mayor, debido a su pequeña carga. Teniendo al hilo encargado del envío de comandos al *drone* con más frecuencia, aseguramos además una mejor respuesta frente a imprevistos.

El hilo que gestiona las interfaces `ardrone_control` y `remoteConfig` es el encargado de la inicialización de `ardrone_server`. Comienza inicializando los hilos del SDK de Ar.Drone para que comience la comunicación inalámbrica con Ar.Drone y la recepción de los flujos de vídeo y datos (*navdata*) entre `ardrone_server` y el *drone*. También inicializa el hilo que gestiona la interface `camera` encargada de la retransmisión de las imágenes. Una vez inicializado `ardrone_server`, este hilo comienza a ofertar las interfaces `ardrone_control` y `remoteConfig`. Con todo el proceso de inicialización completado el hilo entra en un bucle de control donde quedará a la espera de recibir nuevas órdenes. Debido a que la carga de trabajo de estas dos interfaces no es muy grande, sólomente un hilo se encarga de gestionarlas, dentro de su bucle de control funciona a 20Hz.

A través de la interface `ardrone_control` el hilo puede recibir peticiones de los clientes para comandar órdenes al *drone* o para recuperar la información sensorial de Ar.Drone. El SDK de Ar.Drone nos permite dos modos de funcionamiento para el movimiento del *drone*. Primero el modo de comandos progresivos que consiste en el envío de velocidades a los ejes del *drone*, es el modo de funcionamiento normal para realizar movimientos con Ar.Drone. Segundo, el modo “cernirse en un punto”, que consiste en provocar que Ar.Drone quede cernido en un punto a una determinada altura manteniendo la posición.

Heredado del paquete `ardrone_autonomy`, `ardrone_server` dispone de un mecanismo encargado de detectar pequeñas variaciones en los valores para los comandos de movimiento recibidos por los clientes

ICE. El mecanismo comprueba si la diferencia de valores entre los recién recibidos y los valores del comando anterior es mayor que un determinado umbral, que se considera insignificante. Si no ha habido un cambio considerable, `ardrone_server` no vuelve a comandar al *drone* nuevos movimientos y sigue con los últimos comandados. Con esto se persigue no enviar valores para los ejes que se sabe no van a surgir efecto alguno, consiguiendo así una menor saturación de las comunicaciones.

Además `ardrone_server` dispone de un segundo mecanismo para activar el modo “cernirse en un punto”. El SDK nos permite activar el modo de cernirse en un punto con una variable booleana al llamar a la función que comanda los movimientos, en `ardrone_server` este modo se puede activar estableciendo los 6 valores que la interface ICE `ardrone_control` contiene para los comandos de movimiento a 0. Este mecanismo se encarga de comprobar que se cumplan estas condiciones y si procede, activa el modo de cernirse en un punto. En cualquier caso, `ardrone_server` no permite que los valores de los ejes salgan del rango $[-1,1]$, ya que es una restricción impuesta por el SDK de Ar.Drone.

Para las maniobras básicas como el aterrizaje, el despegue o el modo de emergencia, simplemente se activa o desactiva una variable booleana declarada para tal efecto. Mientras tanto los hilos del SDK de Ar.Drone comprueban los valores de dichas variables para enviar el correspondiente comando AT a Ar.Drone.

Como vimos en el punto 4.1.1 la interface `remoteConfig` permite a `ardrone_server` recibir ficheros de configuración XML a través de la red. Una vez que el hilo encargado de esta interface reciba una petición de transferencia de fichero, comenzará una comunicación con el cliente para retransmitir el fichero. Este fichero se almacena en disco a medida que se reciben los fragmentos del fichero. Una vez completada la transferencia, el hilo se comunicará con el SDK de Ar.Drone indicando las opciones que deben ser configuradas e indicándole el valor de éstas. Esta acción provoca que el hilo del SDK encargado de los comandos AT envíe al Ar.Drone la información de las opciones a configurar codificándola en cadenas binarias que se envían como comandos AT.

De manera paralela, el hilo que maneja la interface `camera` se encarga de recoger de una estructura de datos que contiene el SDK de Ar.Drone, la secuencia de bytes que componen la imagen. La imagen ha sido almacenada en dicha estructura de datos por el hilo del SDK encargado de la recepción y decodificación de vídeo, por lo que el acceso a ésta estructura de datos está protegido con un semáforo. Con los datos de la imagen, este hilo procesa la imagen para extraer del flujo de vídeo el canal de la imagen para la cámara activa. A continuación recorre la lista de los clientes que están esperando obtener la imagen para enviársela. El bucle de control del hilo trabaja a una determinada frecuencia, por lo que, una vez terminada la tarea calcula cuánto tiempo ha tardado. Si ha tardado más de lo esperado, continua enseguida con la siguiente iteración del bucle. Si por el contrario ha tardado menos, el hilo se duerme durante un tiempo especificado por la diferencia entre el tiempo que ha tardado y el tiempo que debe esperar hasta la siguiente iteración para mantener la frecuencia nominal.

Por último, para facilitar el mantenimiento de la aplicación hemos creado un fichero `CMakeList.txt` para automatizar la compilación y enlazado del componente. Este fichero incluye todos los directorios y

ficheros necesarios para las fases de compilación y enlazado. Lo hemos dado de alta en el repositorio oficial de JdeRobot.

4.1.3. Gestión de configuración

Con el propósito de agilizar la administración de la aplicación, hemos creado un fichero de configuración XML. Este fichero contiene todos los datos necesarios para la inicialización de las interfaces y algunas opciones de configuración de Ar.Drone, como por ejemplo, la cámara utilizada por defecto, la altitud máxima permitida o si el vuelo se realizará en interior o exterior. En la fase de arranque, **ardrone_server** lee el fichero y carga las opciones configuradas en el archivo de configuración.

```
<?xml version="1.0" encoding="UTF-8"?>
<ArDroneServer>
  <Configuration>
    <interface_camera>
      <end_point>default -h 127.0.0.1 -p 9999</end_point>
      <adapter>ArDroneServerCamera</adapter>
      <name>cameraA</name>
      <framerateN>15</framerateN>
      <framerateD>1</framerateD>
      <format>RGB8</format>
      <image_width>320</image_width>
      <image_height>240</image_height>
    </interface_camera>
    <interface_control>
      <end_point>default -h 127.0.0.1 -p 9998</end_point>
      <adapter>ArDroneServerControl</adapter>
      <name>ArDroneControl</name>
    </interface_control>
    <interface_remote_config>
      <end_point>default -h 127.0.0.1 -p 9997</end_point>
      <adapter>ArDroneServerRemoteConfig</adapter>
      <name>ArDroneRemoteConfig</name>
    </interface_remote_config>
  </configuration>
  <ardrone>
    <default_camera>0</default_camera>
    <outdoor>0</outdoor>
    <max_bitrate>4000</max_bitrate>
    <bitrate>4000</bitrate>
    <navdata_demo>0</navdata_demo>
    <flight_without_shell>0</flight_without_shell>
    <altitude_max>3000</altitude_max>
    <altitude_min>50</altitude_min>
  </ardrone>
</ArDroneServer>
```

```

        <euler_angle_max>0.21</euler_angle_max>
        <control_vz_max>700</control_vz_max>
        <control_yaw>1.75</control_yaw>
    </ardrone>
</Configuration>
</ArDroneServer>

```

Tal y como muestra el cuadro 4.1.3 a modo de ejemplo, el fichero XML se divide en dos secciones; la configuración de las interfaces que ofrece **ardrone_server** y la configuración de algunos valores del Ar.Drone. Dentro de la primera sección se pueden encontrar datos como la dirección IP y puerto en el que estará escuchando una interface, el nombre que tendrá o algunas características como el formato de la imagen y el alto y ancho por defecto. En la segunda sección se podrán encontrar valores referidos a configuraciones internas del *drone* como la cámara por defecto, si el vuelo se realizará o no con casco, la altitud máxima... Estas opciones de configuración del *drone* han sido elegidas en función de su utilidad, si bien Ar.Drone dispone de muchas opciones de configuración, sólomente se han incluido aquellas que hasta el momento han resultado útiles para alguna aplicación o propósito específico.

4.2. Herramienta: **uav_viewer**

Una estación en tierra o GCS (Ground Control Station) es una estación base donde se recibe la información generada por un VANT y con la que se puede monitorizar y controlar toda la actividad de estos vehículos. Estas estaciones pueden ser fijas o móviles y permiten al operario que maneje el VANT conocer su estado de una manera simple. **uav_viewer** es una estación de tierra para VANTs. Para este trabajo se ha focalizado en la visión de los datos generados por Ar.Drone y además permite la teleoperación del cuadricóptero.

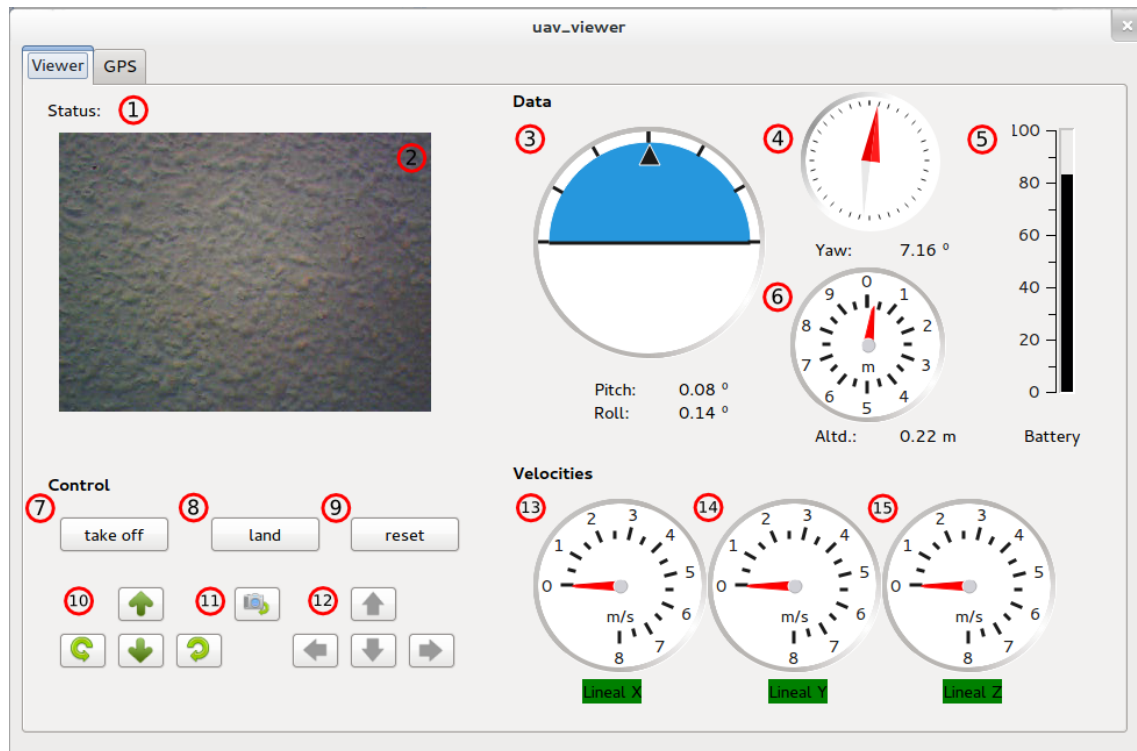


Figura 4.4: uav_viewer

La Figura anterior muestra el interfaz gráfico de la estación de tierra `uav_viewer`. A continuación se listan todas las partes que componen la interfaz gráfica de usuario:

1. Status, muestra el estado actual de Ar.Drone: Initied, TakeOff, Landing, Flying, Moving o Unknown.
2. En este recuadro se muestran las imágenes capturadas por las cámaras de Ar.Drone.
3. Horizonte artificial o indicador de actitud, muestra la orientación de Ar.Drone respecto del horizonte. En él podemos ver el alabeo (ángulo *roll*) y el cabeceo (ángulo *pitch*). De este modo, conoceremos si el cuadricóptero está inclinado lateralmente (alabeo) y la orientación del morro (cabeceo).
4. Con este indicador podemos conocer el ángulo yaw (guiñada) que es la rotación del eje respecto de la vertical.
5. Este objeto gráfico muestra el nivel de la batería.
6. Indicador de altitud, nos muestra la altitud estimada por el software a bordo del robot en metros a la que se encuentra Ar.Drone.
7. El botón de despegue, este botón captura también el evento generado por la pulsación de la tecla T.
8. El botón de aterrizaje, este botón captura también el evento generado por la pulsación de la tecla L.
9. El botón de reset, este botón captura el evento generado por la pulsación de la tecla R. Cuando enviamos la señal reset al cuadricóptero éste interrumpirá su vuelo y caerá al suelo a plomo.
10. Botonera para el control progresivo de orientación y altura, estos botones se encargan de:

- a) Elevar el vehículo, este botón captura el evento generado por la pulsación de la tecla W.
 - b) Bajar el vehículo, este botón captura el evento generado por la pulsación de la tecla S.
 - c) Rotar el vehículo hacia la izquierda, este botón captura el evento generado por la pulsación de la tecla A.
 - d) Rotar el vehículo hacia la derecha, este botón captura el evento generado por la pulsación de la tecla D.
11. Cambio de cámara, este botón envía una señal a Ar.Drone para cambiar de cámara frontal-vertical o vertical-frontal. Captura el evento generado por la pulsación de la tecla C.
 12. Botonera para el control progresivo del desplazamiento, estos botones se encargan de:
 - a) Mover el vehículo hacia adelante, este botón captura el evento generado por la pulsación de la tecla 8 del teclado numérico.
 - b) Mover el vehículo hacia atrás, este botón captura el evento generado por la pulsación de la tecla 2 del teclado numérico.
 - c) Desplazar el vehículo hacia la izquierda, este botón captura el evento generado por la pulsación de la tecla 4 del teclado numérico.
 - d) Desplazar el vehículo hacia la derecha, este botón captura el evento generado por la pulsación de la tecla 6 del teclado numérico.
 13. Velocímetro Lineal X, refleja la velocidad medida de desplazamiento en el eje X, se expresa en m/s.
 14. Velocímetro Lineal Y, refleja la velocidad medida de desplazamiento en el eje Y, se expresa en m/s.
 15. Velocímetro Lineal Z, refleja la velocidad medida de desplazamiento en el eje Z, se expresa en m/s.

4.2.1. Diseño

La aplicación `uav_viewer` ha sido desarrollada en C++ utilizando para el apartado gráfico la librería para interfaces de usuario Qt y la librería Qwt para algunos objetos gráficos.

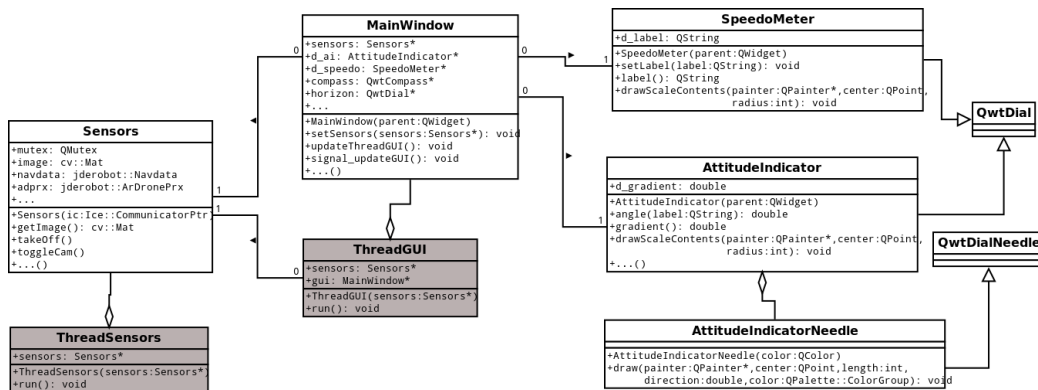


Figura 4.5: Diagrama de clases de `uav_viewer`

La Figura 4.5 muestra el diagrama de clases de `uav_viewer` en el cual se puede apreciar cómo se ha dividido la aplicación en dos hilos. El primer hilo, `ThreadSensor` gestiona toda la comunicación con

Ar.Drone mediante la clase **Sensors**. El hilo **ThreadGUI** se encarga de la interface de usuario definida en la clase **MainWindow**. La interface de usuario contiene un objeto de la clase **Sensors** para recoger los datos sensoriales de Ar.Drone y comandar órdenes de movimiento sobre el *drone*. Para mostrar los datos sensoriales se utilizan varios componentes gráficos, entre estos componentes se encuentran **SpeedoMeter** y **AttitudeIndicator** que por ser más complejos que el resto de componentes se han definido en clases separadas.

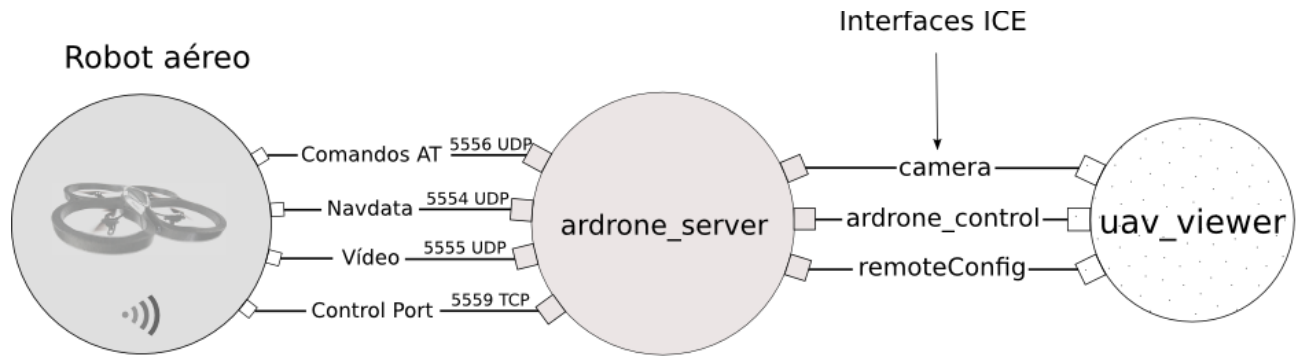


Figura 4.6: Esquema uav_viewer

La Figura 4.7 muestra cómo están organizados los hilos de **uav_viewer**. El primer hilo se encarga de la comunicación mediante interfaces ICE con **ardrone_server**. Este hilo recupera las imágenes capturadas por Ar.Drone a través de la interface **camera** (ver apartado 4.1.1) a una frecuencia máxima de 65Hz. Para obtener los datos sensoriales y comandar órdenes al *drone* se utiliza la interface **ardrone_control** (ver apartado 4.1.1). También se encarga de transferir el fichero XML de configuración a la interface **remoteConfig** (ver apartado 4.1.1) de **ardrone_server**. El hilo que gobierna la interface de usuario recoge los datos obtenidos por el hilo de comunicación y los muestra en los distintos objetos gráficos que componen la aplicación con una frecuencia de 25Hz. El hilo encargado de la comunicación con **ardrone_server** refresca la imagen y los datos sensoriales obtenidos del *drone*, mientras que el hilo de la interface de usuario responde a los eventos generados por el usuario y se encarga de refrescar la ventana de la aplicación con los nuevos datos obtenidos desde **ardrone_server**.

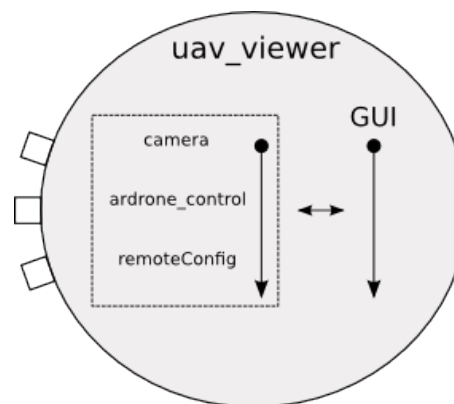


Figura 4.7: Hilos de uav_viewer

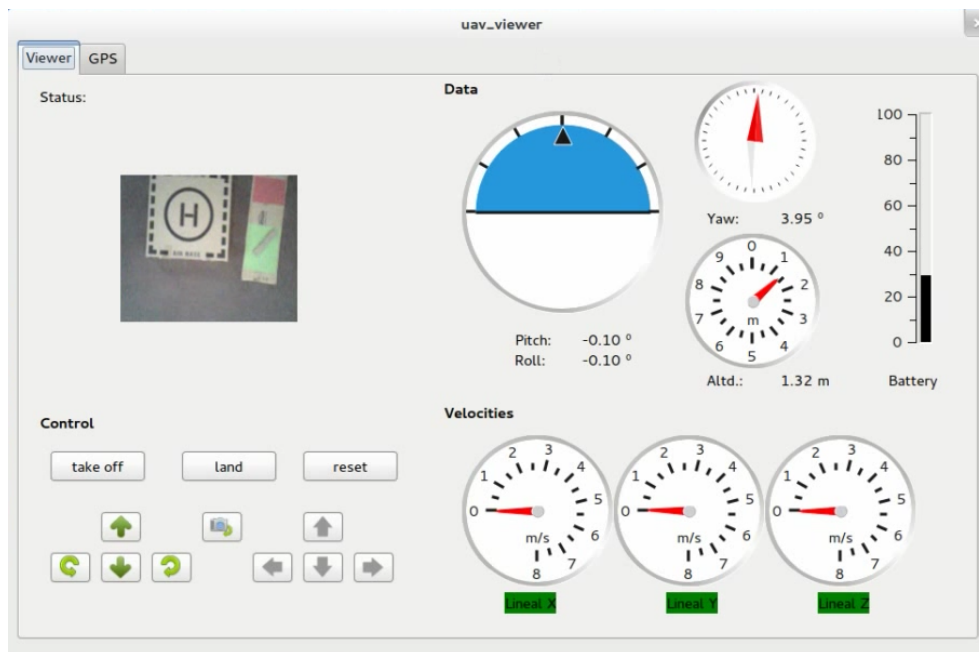


Figura 4.8: Teleoperación de Ar.Drone con `uav_viewer`

`uav_viewer` obtiene las imágenes a través de la interface `camera`, para ello se conecta a la interface que ofrece `ardrone_server`. Con la imagen y los metadatos de ésta recuperados desde `ardrone_server`, `uav_viewer` la transforma en una imagen compatible con Qt. En función de la cámara activa, la etiqueta donde se muestran las imágenes cambiará su tamaño para ajustar al ancho y alto de la imagen obtenida. La adquisición de imágenes la realiza la hebra encargada de la comunicación de Ar.Drone dentro de un bucle de control independiente a la hebra encargada de la interface gráfica. De este modo la hebra encargada de la interfaz gráfica consultará a la hebra que controla el *drone* periódicamente en busca de nuevas imágenes. Cuando las obtenga, refrescará la imagen que muestra en un determinado periodo de tiempo que puede ser configurado.

Para teleoperar el *drone* la interface gráfica captura los eventos generados por el teclado al pulsar las teclas que se han descrito al principio del capítulo. Captura los eventos al presionar y al soltar la tecla, de modo que cada vez que se presiona una tecla se envía al *drone* la orden de incrementar o decrementar su velocidad en un determinado eje. Cuando se deja de presionar, se envía al *drone* un valor 0 en el eje correspondiente para que no ejerza ninguna fuerza sobre él. Con este modo de funcionamiento se consigue el envío de valores simultáneos lo que conlleva una mayor variedad de movimientos, como por ejemplo: avanzar en el eje X mientras se rota en el eje Z.

La interface `ardrone_control` proporcionada por `ardrone_server` además de ofrecer el envío de comandos al Ar.Drone para su control, es capaz de proporcionar los datos de los sensores. A estos datos se los conoce como *navdata* o datos de navegación. El hilo que comunica `uav_viewer` con `ardrone_server` hace uso de ésta interfaz para recibir dichos datos. En la Figura 4.8 se puede apreciar cómo el hilo que gestiona la interface de usuario rellena varios objetos gráficos con los datos sensoriales obtenidos del *drone*. Podemos ver el porcentaje de batería restante, la altitud de Ar.Drone, con el indicador de actitud podemos visualizar fácilmente el alabeo y el cabeceo, además cuenta con tres velocímetros para indicar la velocidad

medida en cada eje. Si esta velocidad es positiva la etiqueta de la velocidad será verde, mientras que si la velocidad es negativa, la etiqueta será roja.

Además de para teleoperar el *drone* y poder visualizar los datos sensoriales de éste, **uav_viewer** nos ha permitido comprobar que el componente **ardrone_server** funciona como es debido y que fácilmente nos permite desarrollar aplicaciones con Ar.Drone. En ¹ se puede visualizar un vídeo de este componente en funcionamiento.

¹http://jderobot.org/Amartinflorido-tfg#uav_viewer_.5BDONE.5D

Capítulo 5. Navegación visual

Una vez presentada en el capítulo anterior la infraestructura software para acceder a los sensores y actuadores del cuadricóptero Ar.Drone de Parrot, en este capítulo se describirá la aplicación desarrollada que implementa las técnicas de navegación visual propuestas. El control visual es un sistema de control en el que la realimentación se obtiene desde una cámara de vídeo y, si lo aplicamos a la robótica, suelen ir colocada en una estructura móvil de forma que es capaz, por ejemplo de mantener en el campo de visión un determinado objeto o escena de la que se requiere la obtención de información.

5.1. Diseño de la navegación visual siguiendo objetos

`Object_tracking` es la aplicación desarrollada en este trabajo para el seguimiento de objetos. Haciendo uso de la cámara del robot y algoritmos de navegación visual logra que el cuadricóptero siga por el espacio 3D a un objeto, bien con su cámara frontal o bien con su cámara ventral. Esta aplicación se ha programado como un componente JdeRobot que se conecta con el servidor `ardrone_server` tal y como muestra la Figura 5.1.

`Object_tracking` se compone de tres partes; la parte perceptiva, que través de algoritmos de visión artificial extrae la posición del objeto de interés en la imagen, la parte actuadora, que en función de la detección del objeto comanda al *drone* los movimientos necesarios para su seguimiento y la parte de la interface de usuario, donde se pueden configurar distintos valores para la detección y seguimiento de objetos.

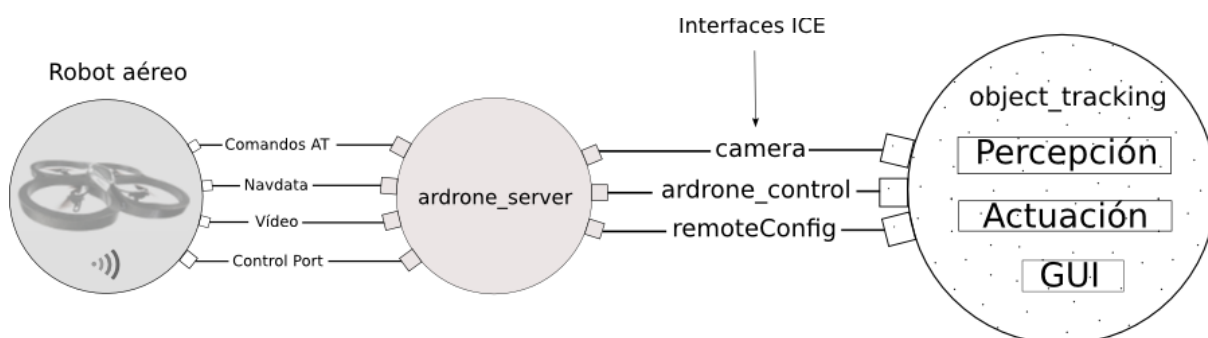


Figura 5.1: Esquema de `object_tracking`

La Figura 5.1 muestra también el esquema de la conexión entre `ardrone_server` y la aplicación `object_tracking`, a través de las distintas interfaces ICE. Internamente, mientras que un hilo gestiona

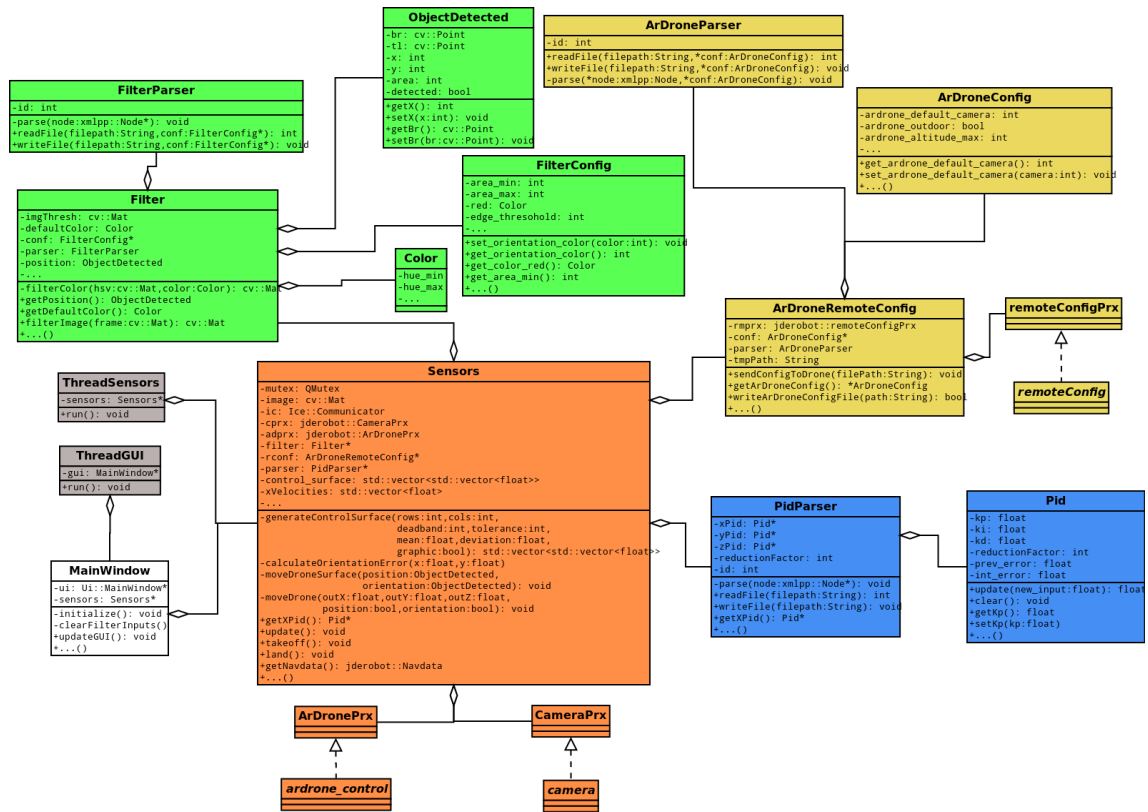


Figura 5.3: Diagrama de clases de object_tracking

las interfaces ICE camera, ardrone_control y remoteConfig, otro hilo gestiona la interface de usuario. Estos hilos, que se ejecutan de manera concurrente funcionan, a una frecuencia determinada (ver Figura 5.2).

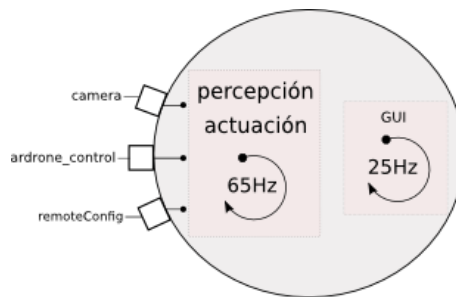


Figura 5.2: Hilos de object_tracking

La información obtenida por el bloque de percepción es compartida con la parte de actuación que alimenta los algoritmos de control para generar una salida que comanda los movimientos del *drone*. Mientras tanto, el hilo encargado de la interface de usuario (GUI) responde a los eventos generados por éste y visualiza los datos obtenidos por el otro hilo, como la imagen obtenida por la parte de percepción o los parámetros usados por el bloque de actuación. La parte perceptiva se explicará en mayor profundidad en el subapartado 5.2, mientras que la parte actuadora se explicará en los puntos 5.3 y 5.4.

La aplicación ha sido desarrollada íntegramente en C++ y la interface de usuario ha sido desarrollada con Qt. La Figura 5.3 muestra el diagrama de clases de la aplicación. El diagrama de clases muestra las relaciones de las clases implementadas en `object_tracking`. La clase `Sensors` (naranja) contiene tres objetos que permiten a la clase comunicarse con Ar.Drone, obtener los datos sensoriales, recibir las imágenes de sus cámaras y configurar el cuadricóptero en pleno vuelo. Estos objetos son: (1) `ArDronePrx` que implementa la interface `ardrone_control` para la obtención de los datos sensoriales y el envío de órdenes al Ar.Drone. (2) `CameraPrx` que implementa la interface `camera` para la recepción de imágenes desde el *drone*. (3) `ArDroneRemoteConfig` que a su vez contiene el objeto `remoteConfigPrx` que implementa la interface `remoteConfig` para modificar la configuración del cuadricóptero en caliente.

La clase `Filter` (verde), que también forma parte de la clase `Sensors`, contiene todos los objetos auxiliares y métodos que implementan el bloque de percepción de `object_tracking`. En el diagrama podemos encontrar algunas clases auxiliares que complementan a la clase `Sensors`, como la clase `PID` (azul) que implementa un controlador PID. Las clases `ArDroneRemoteConfig` (amarillo), `Filter` y `PID`, tienen su clase correspondiente para obtener parámetros de configuración desde ficheros XML. Por último, la clase `MainWindow` es la encargada de la interface de usuario, se genera en tiempo de compilación desde un fichero XML que define la interface gráfica de usuario.

5.2. Percepción visual: Detección de objeto en imagen

La detección de un objeto dentro de una imagen se puede realizar de múltiples maneras. Para este proyecto hemos elegido identificar objetos a partir de su color. Depender del color de un objeto para su detección presenta algunos inconvenientes en comparación con otros métodos de detección. El mayor inconveniente es la luz del entorno donde se encuentre el robot y las sombras que puedan proyectar otros objetos sobre el objeto a identificar. Las imágenes obtenidas de Ar.Drone están codificadas en RGB, este espacio de color es poco robusto frente a los cambios de luz del ambiente. Por este motivo se decidió cambiar del espacio de color RGB a HSV, que si bien no termina de resolver el problema de la luz, es lo suficientemente robusto como para realizar la detección de un objeto en entornos de luz cambiante, siempre y cuando los cambios de luz no sean muy bruscos.

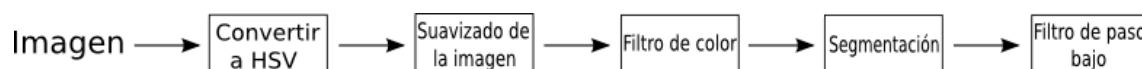


Figura 5.4: *Pipeline* del sistema de percepción de `object_tracking`

La Figura 5.4 muestra el *pipeline* del sistema de percepción de `object_tracking`, que es muy simple y liviano computacionalmente. En los siguientes apartados se explicarán con más detalle cada una de las etapas.

5.2.1. De RGB a HSV

Si se trata de una imagen a color, lo más común es que esos valores estén codificados en el espacio de colores RGB (Red, Green, Blue) que determina la composición del color en términos de la intensidad de

los colores primarios de la luz. El modelo RGB es comúnmente utilizado para la visualización de colores en monitores, televisores, etc... pero presenta una gran desventaja a la hora de intentar identificar un color dentro de una imagen. El espacio de color RGB es poco robusto frente a los cambios de luz. En el campo de la robótica, donde los robots se encuentran en ambientes muy cambiantes es conveniente la utilización de otro espacio de color más robusto ante los cambios de luz, por ejemplo el espacio HSV. HSV (Hue, Saturation, Value) es una transformación no lineal del modelo RGB en coordenadas cilíndricas, de este modo, el color viene definido por:

- Hue (Tinte): es el ángulo entre 0 y 360° que representa color.
- Saturation (Saturación): es la distancia al eje de brillo negro-blanco con valores que van del 0 hasta 100 %.
- Value (Valor): es la altura en el eje blanco-negro tiene valores que van del 0 hasta el 100 %.

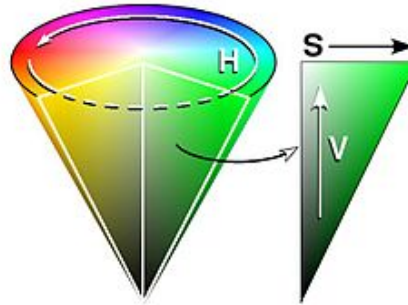


Figura 5.5: Cono de colores del espacio HSV

Ante cambios de luminosidad los cambios suelen reflejarse en V, quedando la S y la H más o menos invariables. Debido a la posibilidad de poder definir un color en base a un tinte y a una saturación, el modelo HSV es un excelente candidato para la identificación robusta de objetos a través de su color en una imagen, dentro de un entorno cambiante.

Una vez obtenida la imagen en RGB, para cambiar el espacio de color a HSV, seguimos el siguiente algoritmo: Sea MAX el valor máximo de los componentes (R,G,B) y MIN el valor mínimo.

$$H = \begin{cases} \text{no definido,} & \text{si } MAX=MIN \\ 60^\circ \times \frac{G-B}{MAX-MIN} + 0^\circ, & \text{si } MAX=R \text{ y } G \geq B \\ 60^\circ \times \frac{G-B}{MAX-MIN} + 360^\circ, & \text{si } MAX=R \text{ y } G < B \\ 60^\circ \times \frac{B-R}{MAX-MIN} + 120^\circ, & \text{si } MAX=G \\ 60^\circ \times \frac{R-G}{MAX-MIN} + 240^\circ, & \text{si } MAX=B \end{cases}$$

$$S = \begin{cases} 0, & \text{si } MAX=0 \\ 1 - \frac{MIN}{MAX}, & \text{en otro caso} \end{cases}$$

$$V = MAX$$

Esta transformación la llevamos a cabo con la función de OpenCV `cvtColor()`.

5.2.2. Suavizado de la imagen

Dentro de la etapa de preprocesado de la imagen existen algunas técnicas que mejoran algún aspecto de la imagen. En el proyecto hemos utilizado una técnica de suavizado. El origen del ruido es múltiple, desde la degradación de la señal, errores en el proceso de captación o de transmisión de la imagen, hasta la falta de iluminación uniforme sobre la escena capturada.



Figura 5.6: *Gaussian Blur*

El sistema de percepción del componente `object_tracking` implementa la técnica de suavizado *Gaussian Blur* (desenfoque Gaussiano) que es el resultado de convolucionar una imagen con la función gaussiana:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Donde x es la distancia desde el origen del eje X, y es la distancia desde el origen del eje Y y σ es la desviación típica de una distribución de Gauss (ver Figura 5.7). Esta función se aplica a todos los píxeles. Los valores de esta distribución se utilizan para construir una matriz de convolución que se aplica a la imagen original. El valor del píxel central recibe un mayor peso (un valor más alto de Gauss) y los píxeles adyacentes reciben un peso más pequeño en función de la distancia del incremento del píxel original. El resultado de este proceso es un desenfoque que conserva las fronteras y bordes del objeto (ver Figura 5.6). Esta tarea la llevamos a cabo con la función de OpenCV `GaussianBlur()`.

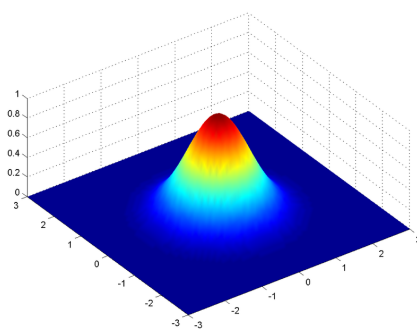


Figura 5.7: Función de Gauss en 2 dimensiones

5.2.3. Filtro de color del objeto

La elección de color es un aspecto importante para la detección del objeto, un color muy vivo será fácilmente detectable, mientras un color con tonos más apagados será más complicado de detectar. La aplicación `object_tracking` nos permite seleccionar los colores que queremos identificar del objeto,

definiendo sus valores HSV.

Mediante la umbralización una imagen de gris o de color se transforma en una imagen binaria, donde los objetos de interés se etiquetan con un valor distinto al de los píxeles de fondo. En el proyecto hemos utilizado la umbralización fija, la imagen resultante B se define a partir de la imagen original I . Si el valor del píxel $I(i,j)$ pasa el umbral el píxel resultante será blanco $B(i,j)$, mientras que si no lo consigue, será negro. El resultado es la imagen binaria de la Figura 5.8. En OpenCV podemos umbralizar una imagen en color definiendo unos rangos con la función `inRange()`.

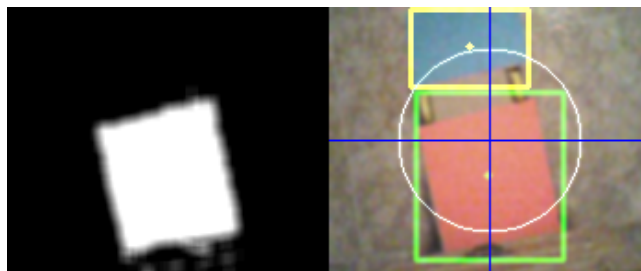


Figura 5.8: Umbralización de la imagen

5.2.4. Segmentación: extracción de coordenadas y área

El objetivo de la detección de objetos es la obtención de las coordenadas del centro del objeto en la imagen. De este modo podemos distinguir si el objeto se ha movido en distintas iteraciones del algoritmo y distinguir hacia dónde. Para llevar a cabo esta tarea el sistema de percepción del componente `object_tracking` realiza una serie de pasos. Primero, una vez obtenida la imagen binaria con los píxeles del color deseado detectados, se utiliza una técnica de la segmentación basada en detección de contornos. La idea de este método es que si se encuentran las fronteras de los objetos, con el fondo contenido dentro de estas fronteras se podrían segmentar los objetos. La Figura 5.9 muestra el resultado de aplicar este algoritmo para la detección de los contornos de un rectángulo. Para la detección de los contornos se utiliza el algoritmo de Suzuki [21] que está implementado en la función de OpenCV `findContours()`.

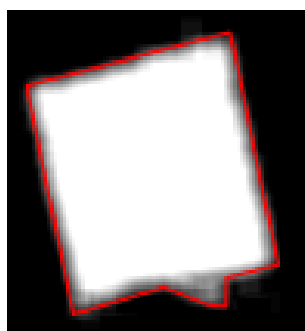


Figura 5.9: Contornos detectados con la función `findContours()`

Con los contornos de los objetos localizados, el siguiente paso es la aproximación de éstos a polígonos. A partir de una lista de puntos que determinan el contorno de un objeto, la función `approxPolyDP()` de OpenCV, que implementa el algoritmo *Ramer-Douglas-Peucker*, es capaz de devolver otra lista de puntos

con un número de puntos reducido. Este algoritmo se basa en la búsqueda de puntos críticos a partir de una tolerancia lineal. Los puntos críticos, que constituirán la línea simplificada, serán los que progresivamente vayan alcanzando una distancia perpendicular mayor respecto a la línea base considerada, siempre superior a la tolerancia impuesta.

Queremos aproximar los contornos a rectángulos, de este modo, conociendo algún punto del rectángulo en la imagen, es posible averiguar el centro del rectángulo siguiendo una simple fórmula. La función `boundingRect()` de OpenCV, devuelve una lista de rectángulos a partir de una lista de puntos. En la imagen derecha de la Figura 5.8 se puede apreciar el resultado de aproximar los contornos detectados en la Figura 5.9 a rectángulos. Esta función intenta crear el rectángulo mínimo que contenga en su interior todos los puntos indicados como parámetro a la función. Siendo P el punto superior izquierdo del rectángulo, podemos hallar el centro del rectángulo del siguiente modo:

$$x = \frac{P_x + b}{2}, y = \frac{P_y + h}{2}$$

Después de calcular las coordenadas del centro del rectángulo, éstas estarán en el sistema de referencia de la propia imagen, por lo que habrá que realizar un cambio del sistema de referencia al utilizado en `object_tracking`, centrado en la imagen.

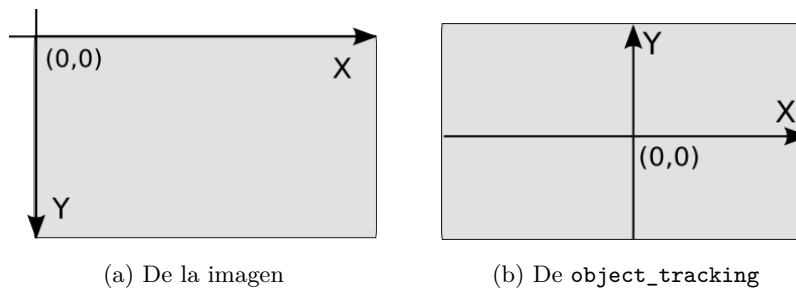


Figura 5.10: Sistemas de referencia

5.2.5. Filtro paso bajo

Debido a las condiciones de la luz ambiente, al propio movimiento del *drone* o a que el objeto que estamos detectando sale de la escena, el rectángulo aproximado por el sistema de percepción fluctúa. Estas fluctuaciones provocan que las coordenadas estimadas del centro del objeto varíen y en muchos casos estas variaciones no significan que el objeto se haya desplazado realmente. Por ejemplo, un cambio de luz en el ambiente provoca que el sistema de percepción detecte un objeto de distinto tamaño diferente al tamaño del mismo objeto detectado con anterioridad.

Para paliar este comportamiento y que la aplicación `object_tracking` conviva con un entorno con ruido, se ha implementado un filtro paso bajo temporal en la salida del cálculo de las coordenadas del objeto detectado. Este filtro tiene en cuenta la imagen capturada con anterioridad ($t-1$) y la imagen actual (t). Consiste en dar un porcentaje de credibilidad a las coordenadas calculadas en ambas imágenes, de este modo conseguimos que el sistema de percepción se vuelva más conservador (si el porcentaje para la imagen $t-1$ es mayor) confiando más en las observaciones anteriores o que confíe más

en las actuales sin tener tanto en cuenta las observaciones pasadas. Siendo $P(t-1)$ el centro estimado del objeto de la observación anterior y O el centro del objeto de la observación actual y aplicando este filtro podemos estimar las coordenadas finales del objeto (P) aplicando la siguiente fórmula:

$$P(t)_x = (O(t)_x \times \alpha) + ((1 - \alpha) \times P(t-1)_x)$$

$$P(t)_y = (O(t)_y \times \alpha) + ((1 - \alpha) \times P(t-1)_y)$$

Figura 5.11: Filtro paso bajo

En `object_tracking` el filtro usado es conservador, ya que confía un 60 % en la observación anterior. Aplicando este filtro de paso bajo se consigue un suavizado temporal en las estimaciones, haciendo que no varíen bruscamente.

5.3. Seguimiento frontal basado en el Control PID

El seguimiento frontal consiste en la detección y seguimiento por el cuadricóptero de un objeto en el espacio 3D haciendo uso de la cámara frontal de Ar.Drone. En este primer seguimiento no tenemos en cuenta la orientación del objeto. De este modo Ar.Drone seguirá al objeto en los ejes X, Y y Z respecto del siguiente sistema de referencias (ver Figura 5.12a) estando su cámara frontal alineada con el eje Z positivo:

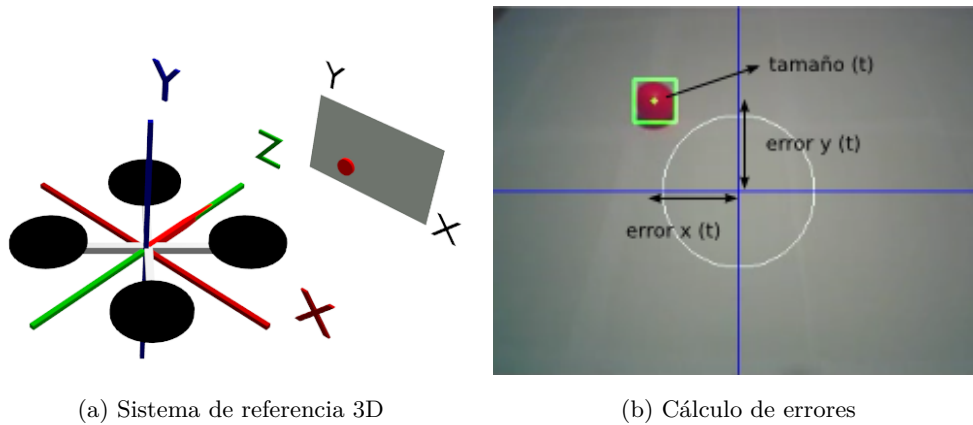


Figura 5.12: Control de posición

Siguiendo con este sistema de referencias 3D, el eje X nos indicará el desplazamiento lateral de Ar.Drone, el eje Y la variación en altura y el eje Z la distancia hacia el objeto, delante-detrás, que mediremos en función del área del rectángulo detectado por el sistema de percepción (ver Figura 5.12). Tomaremos como referencia en la imagen el punto central (0,0) y la diferencia de éste con el punto $P(t)$ estimado por el sistema de percepción nos dará el error de la posición del *drone* respecto al objeto tanto en X como en Y. Para el error en la profundidad utilizaremos como medida el área de un rectángulo de referencia en la imagen, el error será la diferencia entre ese área de referencia y el área del rectángulo detectado. Cuando el área sea más grande, el objeto estará más cerca y el *drone* tendrá que desplazarse hacia atrás. Por el contrario, si el área es más pequeña el objeto estará más alejado y el cuadricóptero tendrá que acercarse a él. Estos errores alimentarán el sistema de actuación que hemos basado en tres controladores PID, uno para cada eje.

$$\begin{aligned}
X &\longrightarrow X_{real3D} \\
Y &\longrightarrow Y_{real3D} \\
Tamaño &\longrightarrow Z_{real3D}
\end{aligned}$$

Figura 5.13: Correspondencia de los ejes

$$\begin{aligned}
error(t)_z &= tamaño(t) - tamaño_{referencia} \\
error(t)_x &= P(t)_x - 0 \\
error(t)_y &= P(t)_y - 0
\end{aligned}$$

Figura 5.14: Cálculo de errores

Un controlador Proporcional-Integral-Derivativo (PID) es un controlador de bucle cerrado con retroalimentación de sus salidas (5.16). Es un sistema al que le entra un error calculado a partir de la salida deseada menos la salida obtenida y ese error es utilizado para corregir la entrada en el sistema que queremos controlar. El controlador intenta minimizar el error ajustando la entrada del sistema. Con esto se consigue que el dispositivo a controlar, en este caso el movimiento del *drone*, reaccione frente a los cambios introducidos, que para este sistema son los errores en los ejes X, Y y el tamaño del objeto detectado. Según muestra la ecuación 5.16 cada controlador PID contiene tres componentes: la proporcional, la integral y la derivativa. Primero, la acción proporcional relaciona mediante la constante K_p la entrada de error y la salida correctiva del controlador. Utilizando únicamente esta constante, nuestro controlador puede tender a oscilar. Segundo, para suavizar el resultado del controlador P y conseguir que el sistema no oscile, tenemos la componente derivativa D. La acción derivativa da una respuesta proporcional a la derivada del error. Este factor mide la tendencia del error, si está subiendo incrementa la acción correctora del controlador. Si está bajando la reduce. La acción derivativa es una constante que se la conoce como K_d . Y tercero, la acción integral reacciona al error acumulado a lo largo del tiempo eliminando *offsets*. Debido a factores externos o al propio mecanismo del sistema a gobernar por nuestro controlador, es posible que a medida que pase el tiempo se vaya acumulando un error, que al final provocaría que la salida del sistema se mantenga alejada del valor prefijado. Para esta acción tenemos la constante K_i .

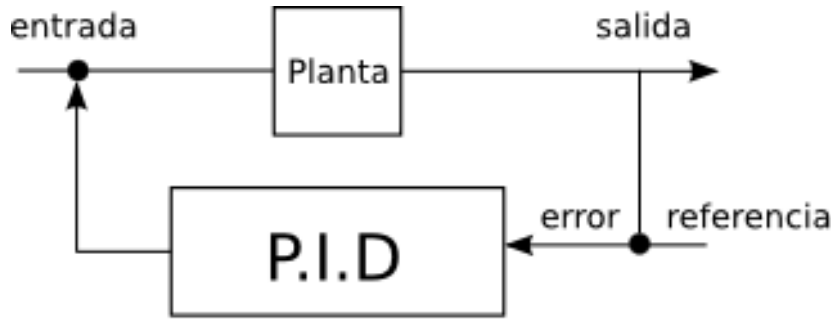


Figura 5.15: Controlador PID genérico

$$u = -K_p e - K_i \int_0^t e dt - K_d \frac{de}{dt}$$

Figura 5.16: Ecuación PID

El sistema de actuación del componente `object_tracking` conoce los errores en la posición X,Y del objeto respecto del *drone* en la imagen y el error del área del objeto con respecto al área de referencia. Cada movimiento que realiza el cuadricóptero se realiza de manera independiente, por lo que para poder seguir al objeto en 3 dimensiones, `object_tracking` implementa 3 controladores PID: uno para

el desplazamiento lateral de Ar.Drone (eje X), otro para la altura del *drone* (eje Y) y el último para la distancia del cuadricóptero con el objeto (eje Z). La entrada de cada controlador PID es el error anteriormente mencionado y la salida es el resultado de aplicar los componentes del controlador PID descritos.

La salida de cada uno de estos controladores es la velocidad que se ordena al *drone* en un determinado eje. Este valor está comprendido entre -1 y 1. Debido a esta restricción los parámetros de los controladores implementados en `object_tracking` contienen un factor de escala que permite adaptar la realimentación a los valores soportados por Ar.Drone para el control de la velocidad.

La diferencia del signo en el valor indica la dirección del movimiento. Para el eje X de la imagen, un valor negativo en la salida del PID indica al *drone* que debe rotar respecto del eje Z hacia la izquierda. Para el eje Y de la imagen, una salida negativa del PID hará que el *drone* se desplace hacia abajo y por último, un valor negativo en el controlador PID del eje Z de la imagen, provocará que el *drone* se desplace hacia atrás. En la sección 6.3 caracterizamos el comportamiento de este controlador en la práctica.

5.4. Seguimiento ventral basado en una Superficie de Control

El seguimiento ventral es el seguimiento de un objeto moviéndose en el suelo usando la cámara vertical de Ar.Drone. Este seguimiento sitúa a Ar.Drone encima del objeto y si el objeto se mueve, el *drone* lo sigue. Si el objeto rota, el *drone* mantiene el alineamiento de su frente con el frente del objeto en todo momento. Además de posicionar el *drone* encima del objeto, también pretende orientar a Ar.Drone respecto del objeto que se está siguiendo de modo que el frente del *drone* se alinee siempre con la cabeza del objeto.

Para conseguir este comportamiento el sistema de actuación se apoya en el sistema de percepción, que le indicará la posición y orientación del objeto en todo momento. Con esta información el sistema de actuación realiza una serie de acciones que serán descritas en los siguientes apartados. Ahora nuestro objeto a seguir contará con dos colores (ver Figura 5.17); uno marca la posición y el otro la orientación, por lo que el sistema de percepción realizará dos filtros a la imagen obtenida de Ar.Drone en busca de los colores que definen al objeto. El sistema de percepción devuelve ahora dos puntos que serán los centros de los rectángulos que conforman cada uno de los dos colores.

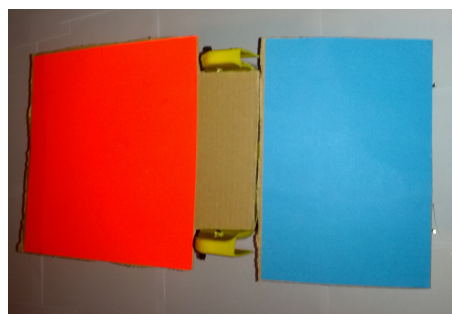


Figura 5.17: Colores para el seguimiento ventral

Para abordar el seguimiento ventral se decidió dividir el sistema de actuación en dos bloques constituyentes (ver Figura 5.18). El bloque Control Sólo de Posición (CSP) (sección 5.4.1) se encarga de posicionar al *drone* encima del objeto a seguir. El segundo bloque, Control Sólo de Orientación (CSO)(sección 5.4.2) se encarga de orientar el *drone* respecto del objeto una vez que el bloque CSP ha posicionado el cuadricóptero encima del objeto, por lo que este sistema parte de un estado en el que Ar.Drone está ya posicionado sobre el objeto. Ambos bloques funcionan continuamente combinando su salida para realizar un seguimiento fluido del objeto.

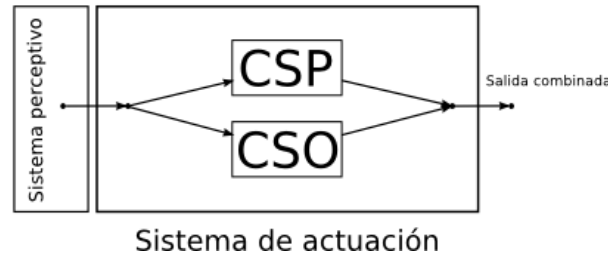


Figura 5.18: Sistema de actuación

Cuando `object_tracking` se está ejecutando para seguir un objeto con la cámara vertical del Ar.Drone el hilo de control recupera del bloque de percepción los centros de ambos rectángulos. Si la luz del entorno cambia bruscamente o debido a algún error del bloque de percepción, `object_tracking` sólo fuese capaz de detectar uno de los rectángulos, el rectángulo detectado sería tratado por la aplicación como el rectángulo de la posición. Este es un modo de que el *drone* no pierda de vista el objeto a seguir. El hilo de control realiza el cálculo de los errores que posteriormente pasará al bloque de actuación. Cuando dicho bloque obtiene los errores comprueba que ambos rectángulos han sido detectados. Si los dos rectángulos se han detectado, se comanda a los motores la velocidad que corresponda al error. Si únicamente se ha detectado un rectángulo, los valores de velocidad que se comandan son aquellos que afecten solamente a los ejes controlados por el CSP, para los demás ejes el valor será 0. En cualquier caso, el hilo de control recoge los valores de velocidad que debe comandar de ambos bloques (CSP y CSO) y mediante la interface ICE `ardrone_control` enviará al componente `ardrone_server` los valores de velocidad que éste tendrá que enviar al *drone* mediante comandos AT (ver sección 3.1.2). Las órdenes de movimiento se comandan de manera simultanea en los tres ejes. Aunque las comanda en el mismo periodo de tiempo, el error de posición y el error de orientación se tratan de maneras distintas. Según se explica en las siguientes secciones.

Aunque la velocidad en el *drone* la determina el sistema de actuación de `object_tracking`, este sistema tiene impuesta una restricción sobre la velocidad máxima que puede comandar al Ar.Drone. El valor de dicha velocidad máxima se ha obtenido de manera experimental resultado de las distintas pruebas realizadas con Ar.Drone.

5.4.1. Control Sólo de Posición

En este control lo único que interesa es posicionar al *drone* encima del color que marca la posición del objeto en tierra. Ahora tenemos únicamente dos ejes que controlar, el eje X y el eje Y (ver Figura 5.19a). El control de altura lo realiza el propio sistema a bordo de Ar.Drone, por lo que no tenemos que

preocuparnos de él. De nuevo el error de la posición del *drone* respecto al objeto será la diferencia de las coordenadas del color que marca la orientación con el centro de la imagen.

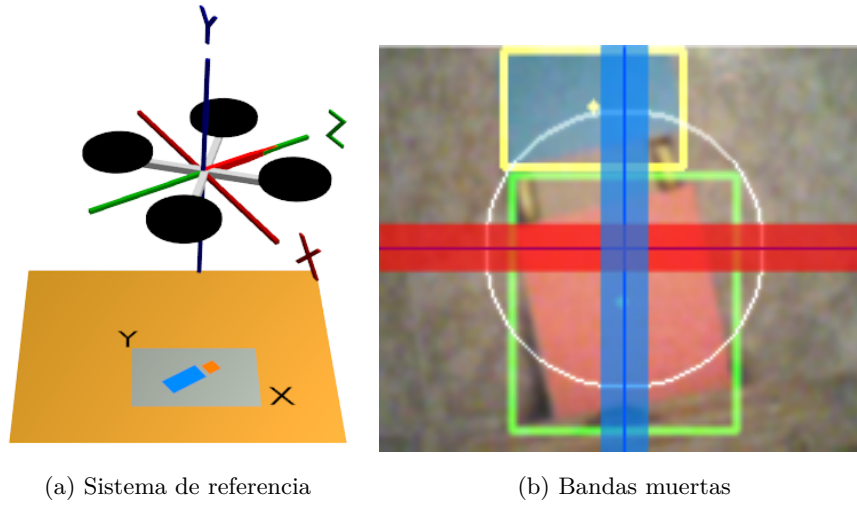


Figura 5.19: Control de posición

Cuando el sistema a bordo interno de Ar.Drone intenta estabilizar el *drone* en el aire, provoca que el cuadricóptero se mueva casi de forma imperceptible a lo largo de los ejes X e Y. Esta leve deriva influye en la detección del objeto, haciendo que el centro del rectángulo que contiene el color fluctúe a lo largo del tiempo, sin que por ello el *drone* se haya desplazado en exceso. Debido a que este comportamiento es intrínseco al sistema a bordo de Ar.Drone, **object_tracking** trata este leve balanceo como ruido, ya que en realidad el *drone* no se está desplazando voluntariamente, sino que intenta estabilizarse. Con el fin de que el CSP encargado de posicionar a Ar.Drone encima del objeto no esté continuamente realizando correcciones en la posición debido a esta deriva generada por la interacción con el control de altura del sistema a bordo de Ar.Drone, se han creado dos bandas muertas (ver Figura 5.19b) a lo largo de los ejes X e Y. Si el error de distancia en el eje X se encuentra dentro de la banda muerta del eje X, el CSP no comanda ninguna acción sobre ese eje. De modo análogo funciona para el eje Y. Cuando el centro del rectángulo que marca la posición se encuentra dentro del cuadrado generado por la intersección de las bandas muertas (ver Figura 5.19b), Ar.Drone entra en modo de cernirse a un punto, queda posicionado con el objeto, y CSP no envía ninguna orden al *drone*.

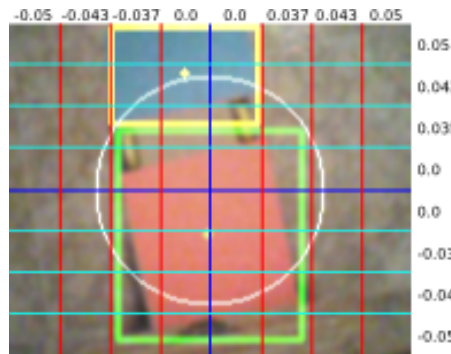


Figura 5.20: División de la imagen en celdas

Mientras que en el seguimiento frontal `object_tracking` comanda la velocidad de los motores en función de la salida de un controlador PID en cada eje, en este seguimiento ventral utilizamos las *curvas de control* de la Figura 5.21. Dado el error de posición de un eje respecto del objeto, la curva contiene el valor a comandar al motor. Para calcular esta curva se ha dividido la imagen en celdas regulares (ver Figura 5.20). Los valores de éstas celdas han sido rellenados tras realizar múltiples pruebas que nos proporcionaron las correcciones adecuadas del control. Estos valores están contenidos entre -1 y 1, donde el signo marca el sentido del movimiento. Para la curva de control del eje X, se divide el ancho de la imagen (ver Figura 5.20) en partes iguales y se rellenan con valores obtenidos de manera experimental a lo largo de las pruebas desarrolladas durante el proyecto.

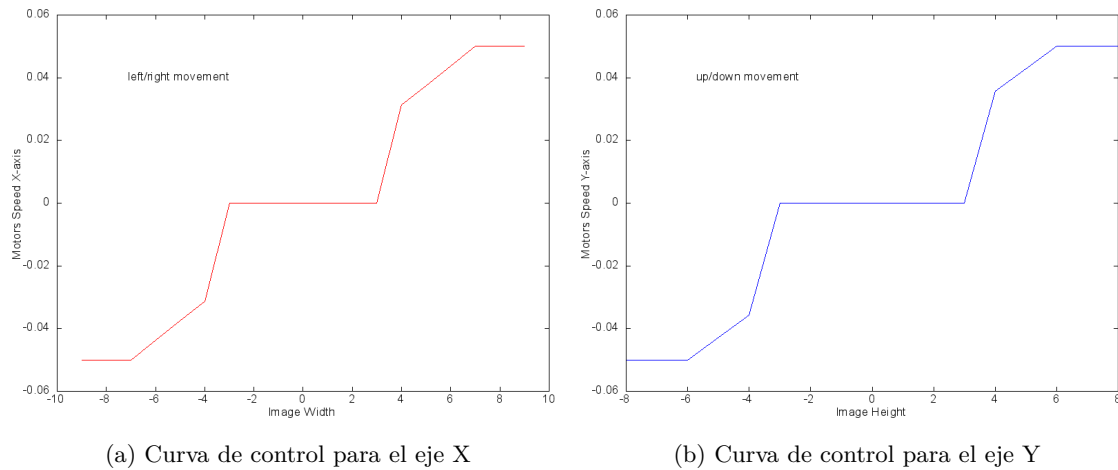


Figura 5.21: Curvas de control

La Figura 5.21a muestra la curva de control para el eje X, en ella se puede apreciar la banda muerta para este eje. En los valores menores al umbral determinado para la banda muerta, en este caso 40 píxeles, el CSP no efectúa ninguna acción sobre la posición comandando a los motores la velocidad 0.0. De modo análogo se puede apreciar en la curva para el eje Y (ver Figura 5.21b). A medida que el centro del rectángulo detectado se aleja del centro de la imagen, la velocidad de los motores aumenta (en ambos sentidos) consiguiendo que el *drone* se sitúe encima del objeto detectado a una velocidad acorde al desplazamiento del objeto. A partir de una determinada desviación la velocidad enviada al cuadricóptero satura en el punto donde coincide la velocidad máxima que se le puede comandar.

5.4.2. Control Sólo de Orientación

Este control persigue alinear el *drone* respecto del frente del objeto que estamos detectando asumiendo que ya está posicionado encima del objeto. Si para el CSP el sistema de percepción detecta el rectángulo más grande, para este control el sistema de percepción detectará también el rectángulo más pequeño (en este caso el azul). Cuando el sistema de percepción localiza el rectángulo pequeño en la imagen verifica que los rectángulos están uno a continuación del otro. Siguiendo con el sistema de referencia de la Figura 5.19a, para orientar al Ar.Drone con el objeto detectado necesitaremos controlar la rotación respecto del eje Y de la Figura 5.12a, ya que, el *drone* está alineado con el eje Z positivo.

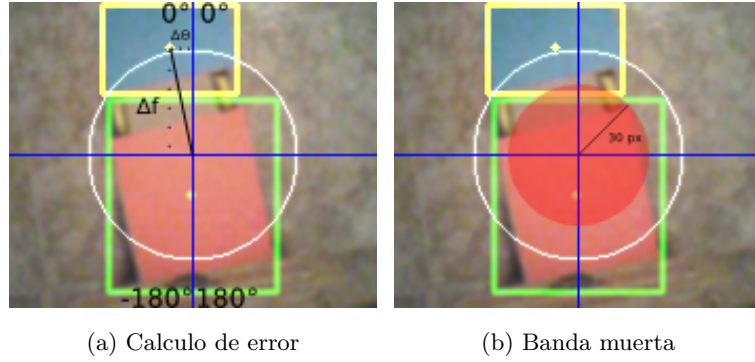


Figura 5.22: Control de orientación

Para calcular el error de orientación del *drone* con respecto al objeto detectado, en primer lugar convertimos las coordenadas cartesianas obtenidas por el sistema de percepción en coordenadas polares (ver Figura 5.22a). Estas nuevas coordenadas nos dan una distancia al centro del segundo rectángulo (Δf) y la diferencia en grados ($\Delta\theta$) entre la recta creada por el centro del rectángulo y el centro de referencia, respecto al eje. Cuando tengamos que comandar la orden de movimiento al *drone*, es interesante distinguir el sentido del movimiento, rotación hacia la izquierda o hacia la derecha. Por ello se han desplazado los ejes 90° hacia la izquierda para dividir la imagen. De este modo, en los cuadrantes de la derecha la diferencia de grados será entre 0° y 180° , mientras que en los cuadrantes de la izquierda la diferencia estará entre 0° y -180° . Cuando el objeto detectado esté en algún cuadrante izquierdo, el valor será negativo y el *drone* tendrá que rotar hacia la izquierda. Si el objeto se encuentra en algún cuadrante derecho el error será positivo y Ar.Drone rotará hacia la derecha. El círculo rojo de la Figura 5.22b conforma la banda muerta para este control.

La región encerrada dentro del círculo rojo de la Figura 5.22b determina la banda muerta radial para la distancia entre el centro del rectángulo pequeño y el centro la imagen. Cuando el centro del rectángulo se encuentra dentro de la banda muerta, los valores $\Delta\theta$ obtenidos varían mucho aunque el objeto no se haya desplazado en exceso. Para evitar que el sistema de actuación esté continuamente corrigiendo la orientación del *drone*, se ha creado ésta banda muerta donde el sistema de actuación no realizará ninguna acción para Δf pequeños. Esta banda muerta tiene un radio de 30 píxeles. Cuando el centro del rectángulo que marca la orientación está fuera de esta banda muerta, el sistema de actuación comienza a rotar el *drone* para orientarlo con el objeto.

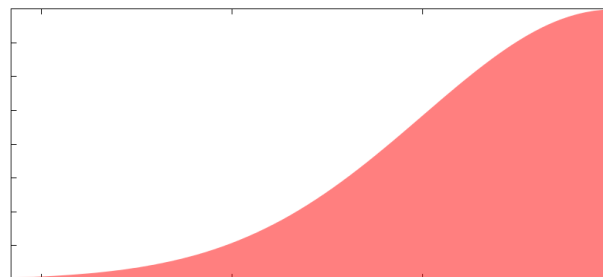


Figura 5.23: Media Gaussiana

Para decidir cuánta velocidad tendremos que imprimir a los motores y que el seguimiento del objeto sea fluido hemos creado una superficie de control (ver Figura 5.24). Esta superficie tiene en cuenta por un lado la distancia del centro de coordenadas al punto detectado (Δf) y por el otro, la diferencia de grados respecto al eje ($\Delta\theta$). La superficie de control se ha rellenado experimentalmente con tramos de una función gaussiana de dos dimensiones que hemos parametrizado empíricamente (ver Figura 5.23). Para implementar la superficie de control se ha creado una matriz donde los índices son los errores anteriormente mencionados discretizando el eje de Δf en 10 niveles y el eje de $\Delta\theta$ en 18. Con este razonamiento si tuviéramos un error de 5° estaríamos en la primera columna, ya que en esa columna se encuadran los errores desde los 0° hasta los 10° . Para los errores negativos se utiliza la misma matriz debido a que cualquier error negativo tiene su correspondiente valor opuesto. Comprobando el signo del error, el valor de la matriz será positivo o negativo. Los 18 valores de la primera fila son tomados de la mitad de una distribución de Gauss de media 0.0 y con una desviación típica de 5. El resto de los valores de las siguientes filas, son los mismos valores de esta distribución, pero desplazados hacia la izquierda. Con estos valores conseguimos que la respuesta del *drone* frente a pequeños errores sea muy suave y a medida que el error va creciendo la respuesta del *drone* es más agresiva.

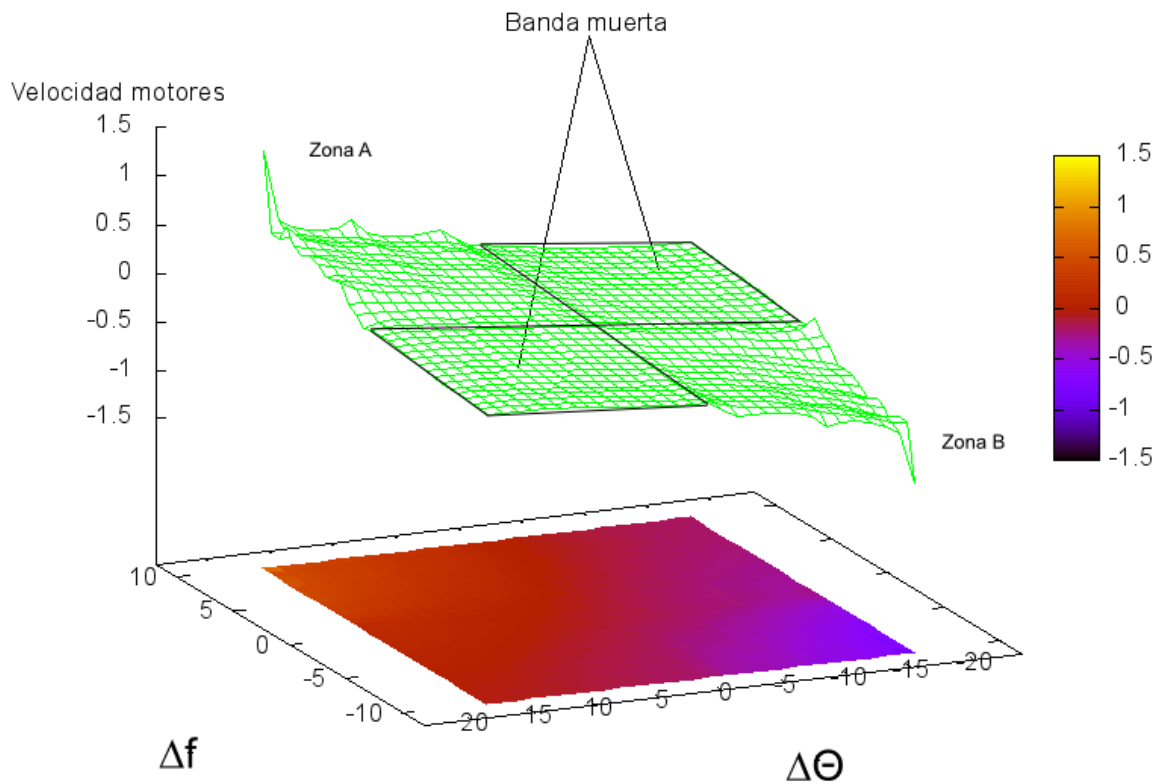


Figura 5.24: Superficie de control para la orientación

La Figura 5.24 muestra la superficie de control creada para el CSO de `object_tracking`. En el eje de ordenadas vemos las distancias del centro de la imagen al objeto (Δf), en el eje de abscisas la diferencia en grados ($\Delta\theta$) y en el eje vertical el valor de salida hacia los motores. La banda muerta que vimos en

la Figura 5.22b se puede apreciar también la superficie. En ésta banda muerta radial el CSO no realiza ninguna acción sobre el cuadricóptero. La Zona A y la Zona B, son las zonas donde el sistema de actuación trabaja para orientar al *drone* con el objeto. Vemos cómo son dos zonas simétricas ya que en realidad el error es el mismo, sólo cambia la dirección de la rotación de Ar.Drone. Si nos fijamos en una de estas dos regiones, vemos cómo a medida que el error crece, la altura de nuestra superficie aumenta (en ambos sentidos) de una manera suave, hasta que finalmente crece bruscamente y satura cuando se alcanza el error máximo

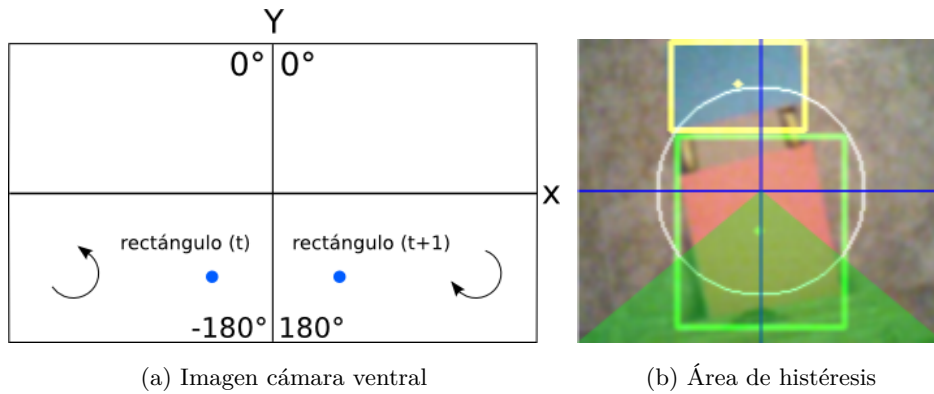


Figura 5.25: Histéresis

Cuando el sistema de actuación comienza a rotar el *drone* para orientarlo con el objeto, podemos encontrarnos con una situación como la descrita en la Figura 5.25a. En esta situación el sistema de percepción detecta el centro del rectángulo que marca la orientación en t , lo que provoca que el sistema de actuación haga rotar hacia la izquierda el *drone*. Mientras Ar.Drone está rotando en el aire, el sistema de percepción detecta el centro del rectángulo en $t + 1$ que está justo en el lado opuesto al detectado en t . De nuevo, el sistema de actuación rotará el *drone*, pero ahora hacia la derecha. Llegados a este punto el *drone* puede entrar en una oscilación, realice el movimiento que realice, en un momento posterior tendrá que realizar el movimiento opuesto. Esto lo constatamos en varios experimentos y para evitar que Ar.Drone entre en este bucle se ha definido una histéresis (ver Figura 5.22b). Si aplicamos este concepto a nuestro problema, el movimiento actual a realizar depende de la posición del objeto detectado en $t - 1$ y t . De este modo, volviendo a la Figura 5.25a, cuando el *drone* comience a rotar hacia la izquierda, debido a la detección del punto en t , éste seguirá rotando hacia la izquierda hasta que el punto $t + 1$ salga fuera del área de histéresis definido en la Figura 5.22b. Cuando `object_tracking` tenga que corregir la orientación del *drone* respecto del objeto, antes de coger los valores de la superficie de control, realiza las comprobaciones pertinentes para evitar que el *drone* oscile en la región que separa los dos cuadrantes inferiores

5.5. Interfaz Gráfica de Usuario

Una vez descritas la parte perceptiva y de actuación que se ejecutan en el hilo de control del componente `object_tracking` en este punto se explica en mayor detalle la interfaz de usuario, que se ejecuta en un hilo propio. Con el fin de facilitar la gestión de las características del seguimiento de un objeto, se decidió agrupar las funciones más utilizadas en una interface gráfica de usuario que se divide en tres pestañas.

La primera pestaña, *tracking*, permite ajustar los parámetros para los algoritmos de percepción y control. Según muestra la Figura 5.26:

1. Es la imagen capturada por el *drone* y procesada por el sistema de percepción.
2. Es el área mínimo en píxeles que la aplicación detecta.
3. Es el área máximo en píxeles que la aplicación detecta.
4. Es el umbral para la detección de bordes en el suavizado de la imagen.
5. Es la botonera del control del *drone* contiene:
 - a) *Takeoff*: este botón envía la señal de despegue al *drone*.
 - b) *Land*: este botón envía la señal de aterrizaje al *drone*.
 - c) *Reset*: este botón envía la señal de emergencia al *drone*.
 - d) *Track*: este botón activa el seguimiento de objetos.
 - e) *Finish Track*: este botón desactiva el seguimiento de objetos.
 - f) *Up/Down*: este botón indica al *drone* que tiene que elevarse/descender.
6. *Track status*: muestra el estado actual del *drone*. Estos estados son: Initied, Take Off, Hover, Moving, Fliying, Landing, o Reset.
7. *PID options*: muestra y permite cambiar los siguientes valores:
 - a) *Kp*: la componente P del controlador PID para el eje X/Y/Z.
 - b) *Ki*: la componente I del controlador PID para el eje X/Y/Z.
 - c) *Kd*: la componente D del controlador PID para el eje X/Y/Z.
8. *Reduction factor*: es el factor de reducción del error. Se explicará más adelante.
9. *Battery*: es el porcentaje de batería del *drone*.
10. *Write Values*: este botón almacena en un fichero XML los valores para los controladores PID.

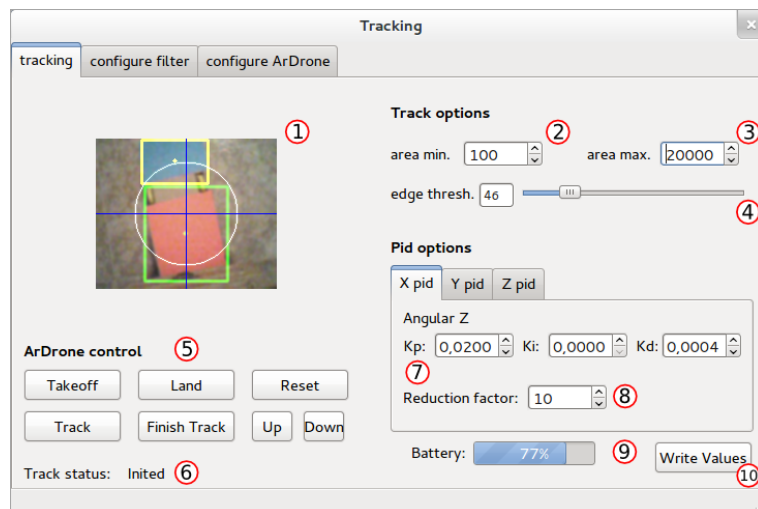


Figura 5.26: Pestaña *tracking* de *object_tracking*

La Figura 5.27 muestra la segunda pestaña, *configure filter*, del interfaz gráfico de este componente que nos permite ajustar el filtro de color viendo en tiempo real el resultado:

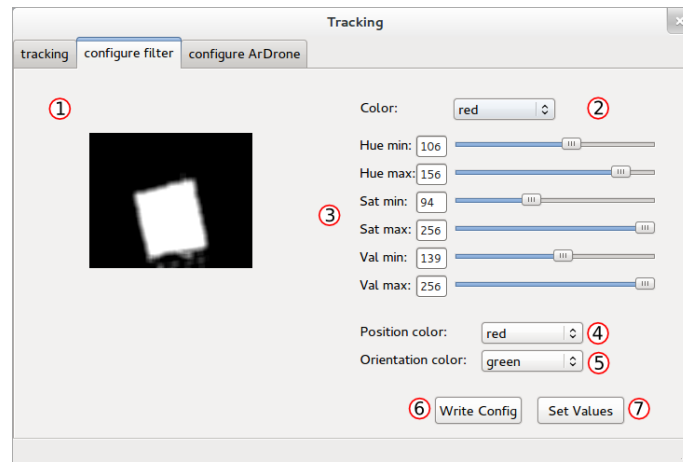


Figura 5.27: Pestaña *configure filter* de *object_tracking*

Donde:

1. Es la imagen binaria (imagen en blanco y negro) obtenida después de binarizar la imagen tras el filtro (ver Figura 5.26) en busca de objetos con los valores definidos para el color rojo. En ella se pueden ver los resultados a medida que se modifiquen los valores HSV del filtro de color, lo que nos dará una idea clara de qué es lo que estamos detectando.
2. Es un selector que selecciona la etiqueta con la que vamos a designar los valores del filtro para cada color seleccionado en (3). Cada filtro tiene una etiqueta que se asigna a este *widget* y un juego de parámetros $\{H_{min} - H_{max}\}$, $\{V_{min} - V_{max}\}$ y $\{S_{min} - S_{max}\}$. De este modo se pueden crear tantos filtros como queramos, asignando una etiqueta a cada uno de ellos.
3. Son los valores HSV del color, que se definen como un rango.
4. *Position color*: es un selector que elige la etiqueta del filtro que usaremos para la posición del objeto.
5. *Orientation color*: es un selector que elige la etiqueta del filtro que usaremos para la orientación del objeto.
6. El botón *Write Config*, escribe en un fichero XML los valores HSV límites del filtro modificados para todos los colores.
7. El botón *Set Values*, aplica los valores HSV del color seleccionado en *Position Color* y *Orientation Color*, como los valores actuales para la detección del objeto.

Con los valores límite del filtro HSV para el color o colores designados para la detección del objeto, el sistema de percepción de *object_tracking* tiene todo lo necesario para la detección del objeto a seguir. Gracias al selector de colores y al fichero XML donde se almacenan los colores predefinidos, *object_tracking* nos permite cambiar los colores a detectar de una manera muy simple.

A veces es necesaria la modificación de algunos valores del sistema de navegación del *drone* mientras éste se encuentra en funcionamiento. Por ello se creó la tercera pestaña, *configure Ar.Drone*: (ver Figura 5.28)

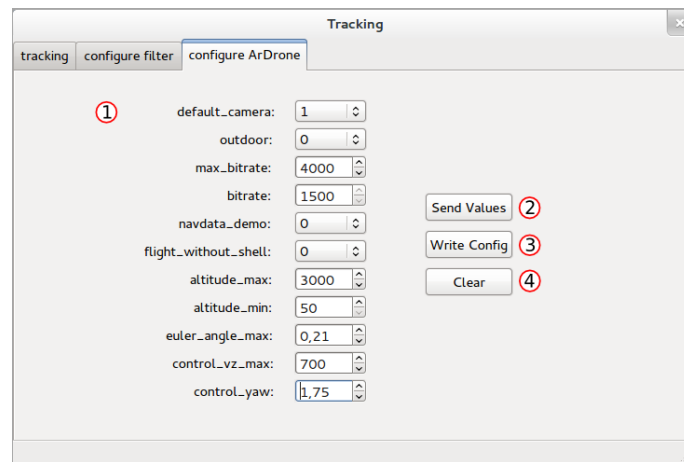


Figura 5.28: Pestaña *configure Ar.Drone* de *object_tracking*

1. Contiene todas las opciones configurables cuando el *drone* está en funcionamiento implementadas por el componente *object_tracking*.
2. *Send Values*: este botón envía los datos de la configuración al *drone*.
3. *Write Config*: almacena los datos de configuración en un fichero XML (*config.xml*).
4. *Clear*: elimina los valores modificados.

Capítulo 6. Validación experimental

En este capítulo se describen las pruebas realizadas con la navegación visual desarrollada, y que se detalló en el capítulo 5. Por un lado, han sido necesarias para ajustar los parámetros del subsistema perceptivo y de control y por el otro lado, sirven de validación experimental de la solución final. Se describen también otras aproximaciones desarrolladas y que finalmente fueron descartadas.

6.1. Filtro de color

El sistema de percepción de `object_tracking` captura las imágenes de Ar.Drone en el formato RGB. Como vimos en el apartado 5.2.1 `object_tracking` cambia el espacio de color de RGB a HSV, y es en éste último espacio de color donde se realiza el filtro de color. Para comprobar la diferencia de resultados entre emplear el filtro en el espacio RGB o HSV se realizó el siguiente experimento. Se grabó un vídeo con las imágenes obtenidas de la cámara ventral de Ar.Drone, en el que se mostraba un rectángulo con dos colores (ver Figura 5.17). Para que los datos fueran realistas se varió la intensidad de la luz de ambiente. Primero, con una luz ambiente normal sin ningún foco de luz artificial. Segundo, a la luz ambiente se le sumo la luz generada por una bombilla colocada encima del *drone*. Por último se eliminó la luz ambiente y se dejó únicamente la luz generada por la bombilla. Durante el transcurso de la grabación se desplazó el rectángulo para simular cambios de movimiento y con éstos cambios en la intensidad de la luz.

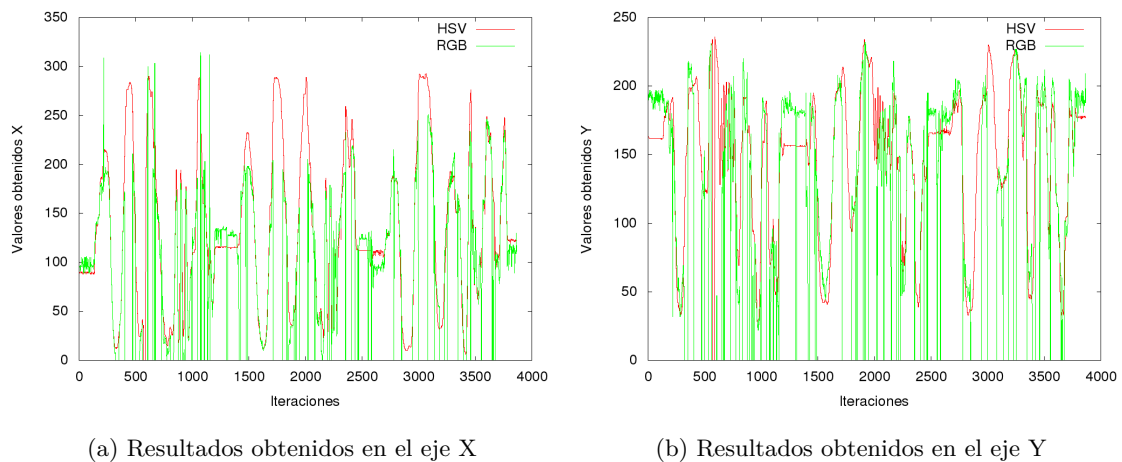


Figura 6.1: Valores registrados en la comparación de ambos filtros

Las Figuras 6.1a y 6.1b muestran respectivamente los resultados obtenidos de la estimación de la posición X e Y del cuadrado que marca la orientación (cuadrado azul). En el eje de ordenadas se muestran

los valores de la coordenada observada y en el eje de abscisas el número de iteraciones empleadas por el algoritmo. En ambas imágenes se puede apreciar cómo las estimaciones obtenidas aplicando el filtro en el espacio de color HSV son más robustas que en el espacio de color RGB. En éste último se observa una mayor variación en la estimación de las coordenadas e incluso la pérdida del color a filtrar en bastantes ocasiones. En las curvas dibujadas con las estimaciones (ver Figura 6.1a) se pueden distinguir tres regiones. La primera que va desde la iteración 0 a la 1200 aproximadamente, con una luz ambiente normal, la segunda comienza en la iteración 1500 y termina en la número 2500, añadiendo un foco de luz artificial a la luz ambiente y la tercera eliminando la luz ambiente normal comienza en la iteración 2800 y termina en la iteración 3800.

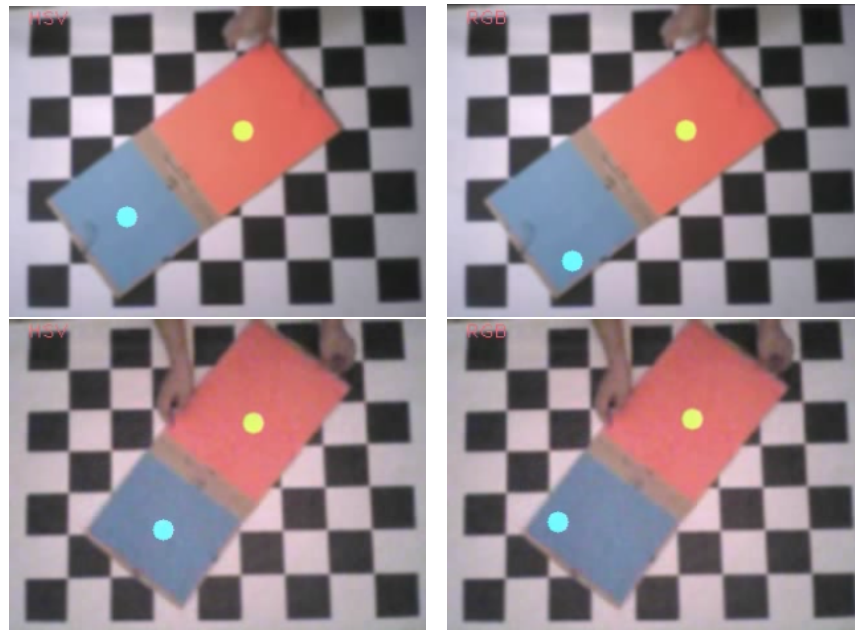


Figura 6.2: Comparación entre HSV y RGB

La Figura 6.2 contiene algunos fotogramas extraídos del vídeo que muestra la comparativa entre un filtro de color HSV y RGB. A la izquierda se encuentran los fotogramas del filtro HSV y a la derecha los del filtro RGB. En estas imágenes se aprecia que los centroides calculados por el filtro RGB no están bien estimados, mientras que los calculados por el filtro HSV sí. Dicho vídeo se encuentra disponible en la bitácora del proyecto¹. Los valores del filtro para los espacios de color fueron:

Filtro HSV	Azul	Naranja
H min/max	40/160	0/20
S min/max	70/160	70/256
V min/max	70/256	190/256

Filtro RGB	Azul	Naranja
R min/max	100/200	190/255
G min/max	100/255	85/140
B min/max	100/110	0/140

6.2. Filtro temporal en estimación de posición

Para comprobar el funcionamiento del sistema de percepción de `object_tracking` se colocó al *drone* suspendido en el aire sujeto con unos finos cables fijados a una estructura. Los valores del filtro de color

¹http://jderobot.org/Amartinflorido-tfg#Color_filter_test:_HSV_vs_RGB

fueron configurados para la detección de dos rectángulos de color rojo. Las pruebas se realizaron con una bombilla situada encima de Ar.Drone. Se utilizó la cámara vertical tal como muestra la Figura 6.3.

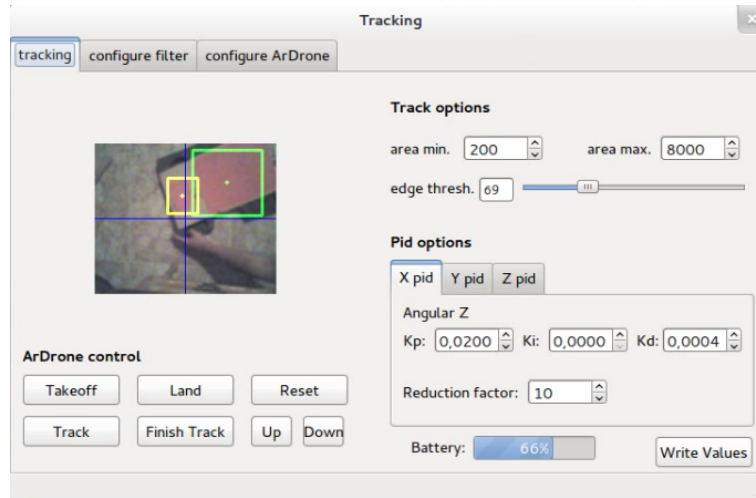


Figura 6.3: Pruebas sobre la detección de objetos

Una parte esencial del sistema de percepción es la extracción de las coordenadas en imagen de los objetos detectados. Para que el sistema de actuación no se vuelva inestable, conviene que estas coordenadas sean muy consistentes en las distintas iteraciones del sistema de percepción. Para ajustar el valor más adecuado de α se ejecutó `object_tracking` para detectar dos rectángulos rojos con la cámara vertical de Ar.Drone. En cada iteración se almacenaron las coordenadas de uno de los rectángulos sin el filtro y con el filtro. Para un α de 0.6 (ver sección 5.2.5) se obtuvieron los resultados que muestran las Figuras 6.4a y 6.4b. Como se puede apreciar en ambas Figuras, entre la iteración 200 y la 300, el objeto salió temporalmente de la escena. Mientras que con los datos sin filtrar se produce una bajada repentina de la curva, con los datos filtrados, esta bajada es menos pronunciada y más larga en el tiempo. De este modo cuando el objeto vuelve a la escena, las coordenadas filtradas no sufren un cambio tan brusco como en las coordenadas sin filtrar.

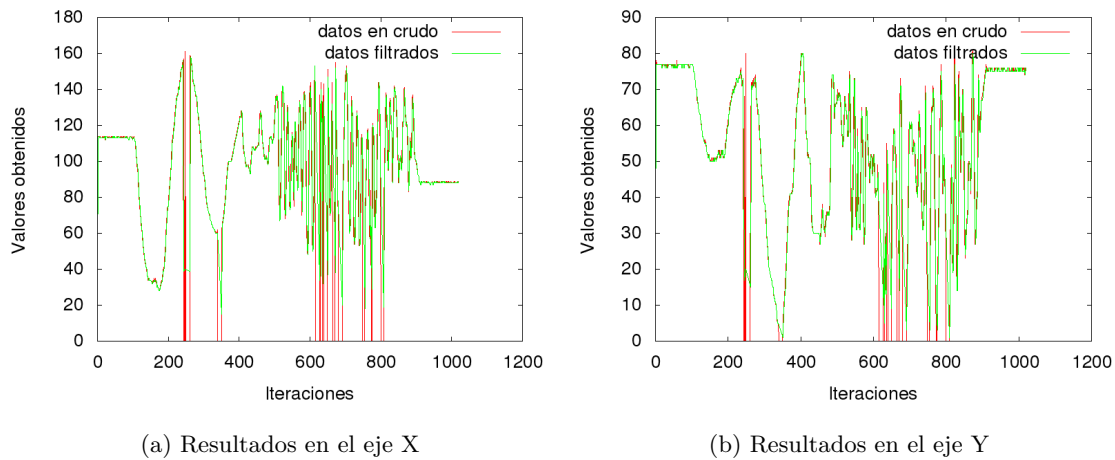


Figura 6.4: Comparación de la detección de la posición

En el resto de las iteraciones se comprueba cómo la curva de los datos sin filtrar alcanza máximos muy rápidamente, mientras que los máximos de la curva filtrada no son tan altos. Cuando el objeto permanece inmóvil (a partir de la iteración 900) los datos sin filtrar contienen fluctuaciones debido a la luz cambiante del entorno. Los datos filtrados también contienen fluctuaciones, pero en este caso son menos numerosas y tienen menor pendiente.

Se realizaron pruebas similares para distintos valores de α . Se comprobó que para valores menores de 0.5, el comportamiento del sistema de percepción era demasiado perezoso. Con valores mayores a 0.8 no se obtuvieron buenos resultados. Finalmente, para obtener el valor final de α se realizaron pruebas con Ar.Drone en movimiento y se observó que con un valor de 0.6 se obtenían los mejores resultados. Se puede encontrar el vídeo de la prueba del filtro temporal para la estimación de la posición en la bitácora del proyecto².

6.3. Seguimiento frontal

El sistema de actuación del seguimiento frontal está gobernado por 3 controladores PID. Cada controlador gobierna el movimiento de cada uno de los ejes. Con el eje X se controla el avance o retroceso de Ar.Drone. Con el eje Y el desplazamiento lateral y con el eje Z la altura del *drone* respecto del suelo. La dificultad de este seguimiento reside en encontrar los valores adecuados para cada una de las tres constantes de cada controlador.

El valor de las constantes de los controladores PID de `object_tracking` se llevó a cabo por el método de prueba y error. Para ello se trató a cada eje de manera independiente, con la premisa de que a mayor error, mayor debería ser la respuesta, siempre y cuando esta respuesta no sobrepasará los límites máximos en la velocidad ni hiciera oscilar al robot aéreo. Se ejecutó `object_tracking` para que Ar.Drone siguiera una pelota roja, comenzando con un controlador P, para que la salida del controlador fuera proporcional al error. Este controlador provoca unos movimientos muy agresivos y a menudo sobreactúa. Para atenuar esa respuesta agresiva se añadió la componente D en el controlador, que suavizó los movimientos del *drone*. Con el controlador PD la respuesta Ar.Drone era la adecuada para el seguimiento frontal. Cuando se añadieron los controladores para los ejes Y y Z al sistema de actuación, se comprobó que era necesario añadir la componente I a los controladores, ya que sin ella, al cabo del tiempo los movimientos del *drone* comenzaban a ser más agresivos. Este proceso se realizó para cada uno de los controladores PID (en altura, derecha-izquierda y delante-detrás), primero sin mover en exceso el objeto a seguir y aumentando progresivamente la velocidad con la que el objeto se desplazaba, realizando numerosas pruebas en busca de valores óptimos para las constantes.

²http://jderobot.org/Amartinflorido-tfg#Raw_data_vs_low-pass_filter_tests

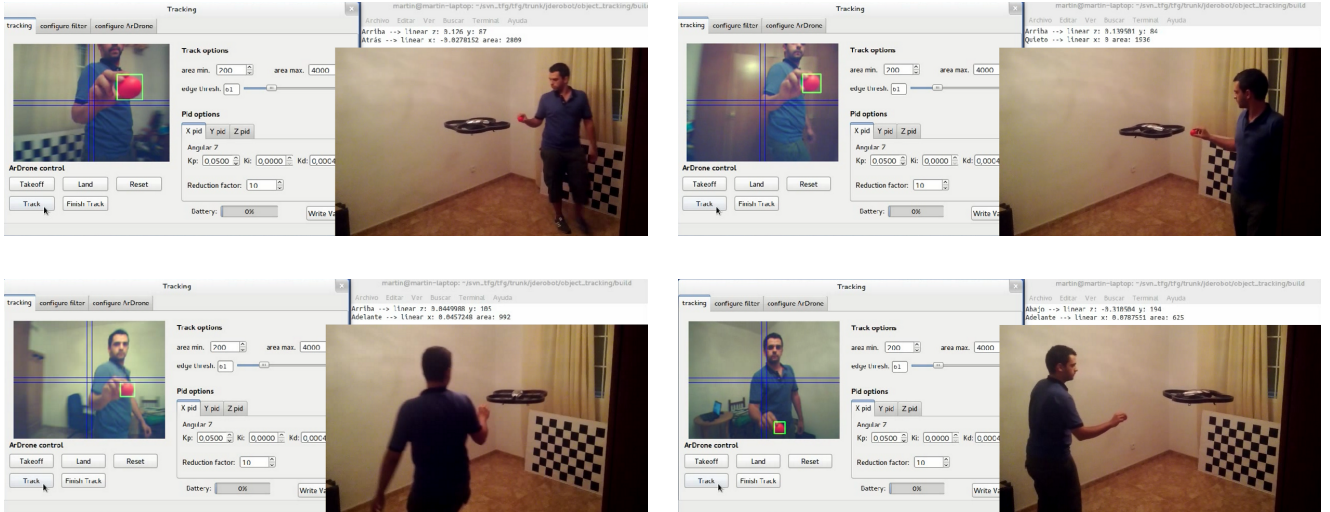


Figura 6.5: Seguimiento frontal con controladores PID

La búsqueda de valores para las constantes de los controladores PID es una tarea ardua pero, sin duda, es un buen ejercicio para comprender de primera mano cómo influyen las componentes del controlador PID en los movimientos del *drone*. La Figura 6.5 contiene algunas capturas del vídeo disponible en la bitácora del proyecto³ donde se observa como el cuadricóptero sigue una pelota roja que está siendo desplazada por una persona. Estas capturas muestran como el *drone* rota sobre el eje Z a medida que la pelota se desplaza a su alrededor. Con este vídeo se demuestra el correcto funcionamiento de los controladores PID para el seguimiento de la pelota con la cámara frontal de Ar.Drone. Finalmente los valores elegidos para cada una de las 9 variables de los controladores PID de *object_tracking* en el seguimiento frontal son:

Controlador PID	K_p	K_i	K_d
Eje X	0.05	0.001	0.0004
Eje Y	0.045	0.001	0.0004
Eje Z	0.009	0.005	0.0005

6.4. Seguimiento ventral

El seguimiento ventral ha sido el más complejo de los desarrollados. Es fácil pensar que una vez desarrollado el seguimiento frontal pueda resultar sencillo exportarlo al ventral, pero éste último tiene características que lo diferencian. En el seguimiento frontal, Ar.Drone sigue un objeto en el espacio de 3 dimensiones controlando la altura y los movimientos en los ejes X e Y. El seguimiento ventral sin embargo, mantiene la posición en los ejes X e Y, pero el control de altura lo realiza el sistema nativo a bordo del *drone*. Este sistema utiliza los sensores de ultrasonido y la cámara vertical para determinar y mantener la altura a la que se encuentra del suelo. Por tanto coexisten en el mismo periodo de tiempo dos controles: el sistema de control de *object_tracking* y el sistema nativo a bordo del *drone* que se encarga de controlar la altura. La ejecución e interacción simultánea de ambos añade un grado de dificultad mayor como ya se explicó en las secciones 5.4.1 y 5.4.2.

³http://jderobot.org/Amartinflorido-tfg#Horizontal_Tracking

6.4.1. Controlador único con superficie de control

Este controlador, que fue explicado en la sección 5.4, es la solución propuesta de este proyecto para el problema del seguimiento ventral en posición y orientación. Es un enfoque distinto al utilizado en el seguimiento frontal (ver sección 5.3) y se llegó a él después de probar distintas alternativas que se verán en el apartado 6.5.

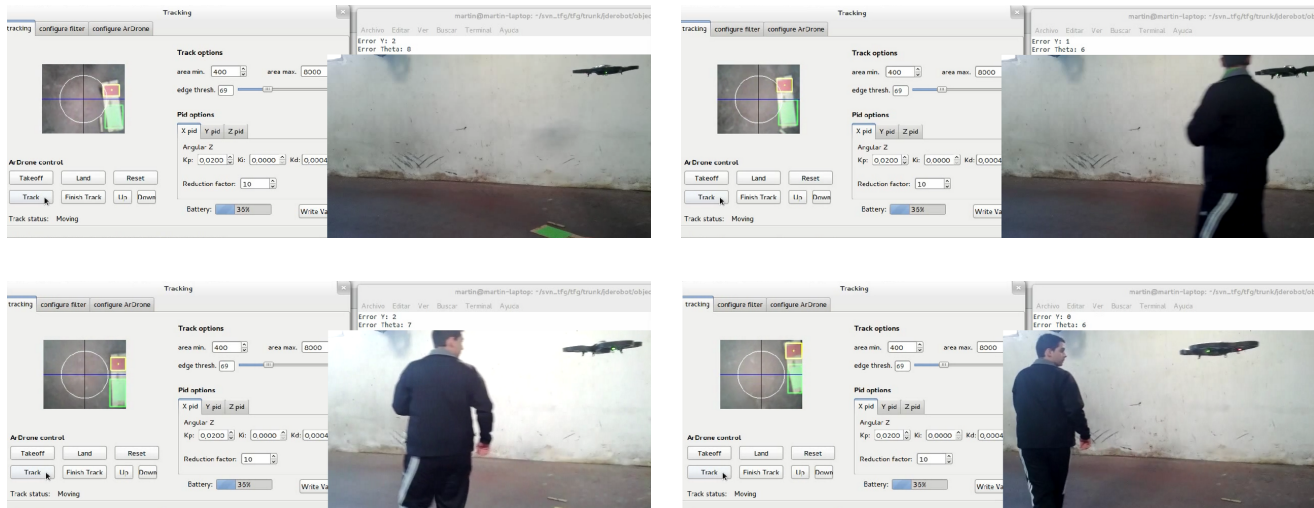


Figura 6.6: Seguimiento ventral con superficie de control

La Figura 6.6 muestra una captura del vídeo que se puede encontrar en la bitácora del proyecto⁴. En esta prueba se deslizó por el suelo un objeto para que el *drone* lo siguiera. Mientras que dicho objeto se deslizaba se introdujeron cambios en la dirección del objeto haciendo que éste girase sobre el suelo. En las capturas de la Figura 6.6 se aprecia como el rectángulo rojo del objeto siempre se encuentra por encima del verde en la ventana de la aplicación `object_tracking` reflejando que el cuadricóptero gira automáticamente a medida que la persona rota y traslada (mediante unos hilos enganchados al objeto) el objeto en el suelo. El vídeo de la prueba muestra como el *drone* realiza un giro en el aire y, a pesar de ello, no pierde el objeto al que está siguiendo. Con la superficie de control se comanda directamente la corrección en todos los ejes, con esto se consigue que los movimientos sean más fluidos.

6.4.2. Control de la deriva de Ar.Drone

Si bien es cierto que el viento podría desviar a Ar.Drone de su rumbo, no es la única situación en la que el *drone* podría desviarse. En el transcurso de este Trabajo Fin de Grado, observamos que el sistema a bordo (cuando intenta estabilizar el cuadricóptero sobre un punto cualquiera a una altura determinada) provoca que el *drone* poco a poco se aleje del punto inicial haciendo que Ar.Drone navegue a la deriva. Para una aplicación de navegación visual que pretende seguir objetos que se encuentran en el suelo esto supone un problema.

⁴http://jderobot.org/Amartinflorido-tfg#Vertical_Tracking_x.2Cy.2Ctheta

Cuando el sistema a bordo de Ar.Drone intenta cernir el *drone* en un punto, éste realiza correcciones constantemente para mantener la altura y la posición. Estas correcciones provocan un leve balanceo que acaba por convertirse en una deriva que arrastra al cuadricóptero en alguna dirección.

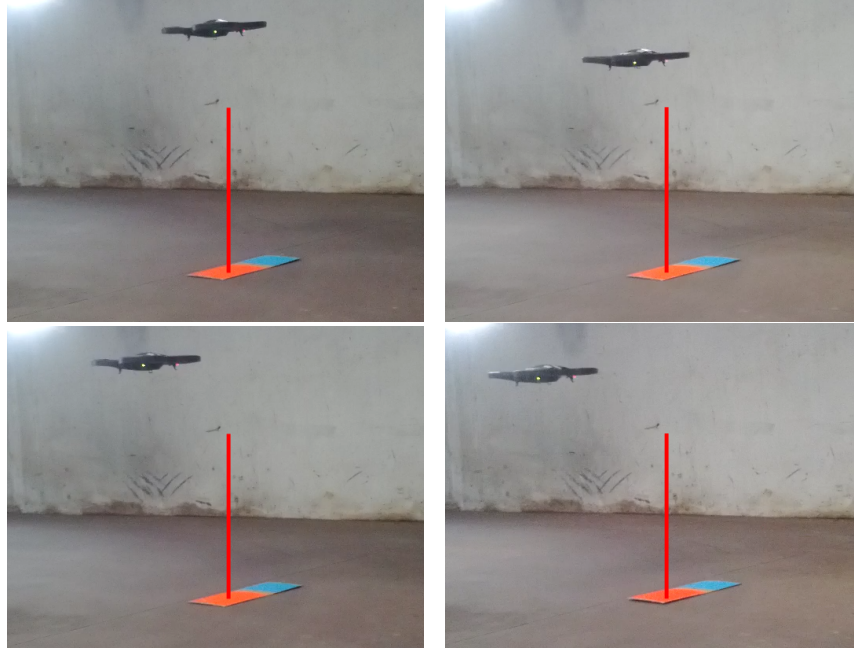


Figura 6.7: Pruebas sobre la deriva de Ar.Drone

Para medir esta deriva y establecer una región de seguridad (banda muerta) se realizaron varias pruebas que determinasen el valor de ese desplazamiento a la deriva de Ar.Drone en el tiempo. El escenario es el descrito por la Figura 6.7, en él el sistema de actuación de `object_tracking` no está funcionando, el *drone* se mantiene “quieto” en un punto por el propio sistema a bordo de Ar.Drone. La prueba consistió en medir el desplazamiento del centro de la imagen con una posición de referencia. Dicha posición es el centro del rectángulo naranja registrado la primera vez que el *drone* se colocó encima del objeto que se mantiene fijo en el suelo. En las siguientes iteraciones se registraron la distancia entre el centro de la imagen y el centro del rectángulo. La primera de las cuatro imágenes corresponde al momento en el que se cogió la referencia (posición en pantalla del rectángulo naranja) y la última de ellas se corresponde con una captura realizada tres minutos después. Al rectángulo se le ha superpuesto una línea roja colocada de manera vertical en la imagen para que sirva de referencia.

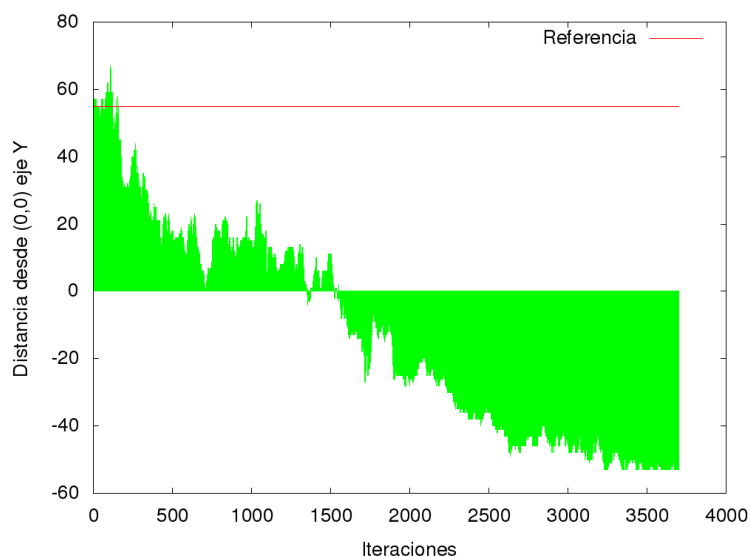


Figura 6.8: Error en píxeles del modo cernido

Como muestra la Figura 6.8 la estimación de la posición del rectángulo rojo en el eje Y decrementa a medida que pasa el tiempo. En la toma de referencia el objeto se encontraba en el cuadrante positivo de la imagen, a medida que pasa el tiempo, se observa cómo poco a poco se desplaza hacia el cuadrante inferior hasta llega la máxima diferencia que coincide con el final de la prueba. La máxima diferencia entre la primera observación y la distancia más larga es de 90 píxeles. Esto nos indica que el *drone* se ha desplazado, sin ejercer ninguna fuerza sobre él a excepción de la ejercida por el propio sistema a bordo de Ar.Drone, 90 píxeles en aproximadamente 3 minutos. El sistema de actuación de `object_tracking` deberá tener en cuenta que, cuando el sistema nativo a bordo Ar.Drone intenta cernir a un punto el *drone*, éste se desplaza lentamente. La intersección de las bandas muertas que muestra la Figura 5.19b sirve para que `object_tracking` controle esta deriva del *drone*. El vídeo de esta prueba se encuentra en la bitácora⁵ del proyecto.

6.5. Alternativas probadas para el control ventral

En este punto describimos las pruebas que se llevaron a cabo para explorar variantes de los algoritmos de control necesarios para el control de la posición y orientación. Se siguió un procedimiento incremental donde se trató de identificar la mejor respuesta de Ar.Drone para el seguimiento de objetos ventral. A lo largo de los siguientes puntos se detallarán los distintos algoritmos de control implementados y los motivos que llevaron a descartarlos.

6.5.1. Controlador PID único

Con los buenos resultados obtenidos en el seguimiento frontal haciendo uso de controladores PID, la primera prueba consistió en aplicar el mismo sistema de actuación para el control ventral. Al igual que en el seguimiento frontal, para el control ventral se necesitan 3 controladores PID; dos para el control del

⁵http://jderobot.org/Amartinflorido-tfg#Drift_tests

desplazamiento en los eje X e Y y un tercer controlador para el movimiento de rotación respecto del eje Z (ver Figura 5.19a). Los errores de posición y orientación del *drone* respecto del objeto son los mismos que se describen en las secciones 5.4.1 y 5.4.2. Con los errores calculados y las constantes de los controladores PID definidas, `object_tracking` calcula la velocidad de cada de eje y ordena a Ar.Drone el movimiento.

Con esta aproximación los resultados obtenidos no fueron los deseados. Mientras que los controladores encargados de posicionar a Ar.Drone encima del objeto funcionaban razonablemente bien, el problema surgió a raíz del controlador de la orientación del *drone*. El continuo desplazamiento del cuadricóptero para posicionarse sobre el objeto provoca que los errores para el control de orientación fluctúen en exceso. Lo que a su vez originó que el controlador encargado de la orientación comandase a menudo órdenes contradictorias que condujeron a la desestabilización del *drone*. En este sistema caótico la tarea de encontrar los valores apropiados para las constantes del controlador PID encargado de la orientación resultó ser demasiado compleja y en la mayoría de los casos no tenían buenos resultados.

6.5.2. Dos controladores PID priorizando la orientación

Se comenzó con un enfoque progresivo, en primer lugar se desarrolló y se ajustó el controlador PID que gobierna la orientación del *drone*. Cuando se comprobó que por si solo el controlador orientaba correctamente al cuadricóptero si el objeto a seguir no está muy descentrado, se desarrolló y se ajustó el segundo controlador PID encargado de posicionar a Ar.Drone sobre el objeto. Con ambos PID funcionando por separado se introdujo el árbitro encargado de coordinar los controles en base a una prioridad.

Los resultados con este sistema de actuación no fueron los esperados, pronto se comenzaron a ver los problemas. El *drone*, en su afán de orientarse respecto al objeto, a menudo salía de la escena perdiendo el objeto de vista. Como la orientación tenía la prioridad, en la mayoría de los casos el control de posición no actuaba, y el Ar.Drone salía de la escena perdiendo de vista al objeto.

6.5.3. Dos controladores PID priorizando la posición

Con los resultados de la prueba anterior se decidió probar dar prioridad a la posición en lugar de a la orientación, dado que es lógico que el *drone* primero se sitúe sobre el objeto para después orientarse respecto a él. Con este nuevo enfoque, el *drone* primero tendría que posicionar sobre el objeto para después comenzar a orientarse. De nuevo este sistema de actuación se basa en 2 controladores PID y un árbitro. Los errores se calculan de la misma manera que en la prueba anterior.

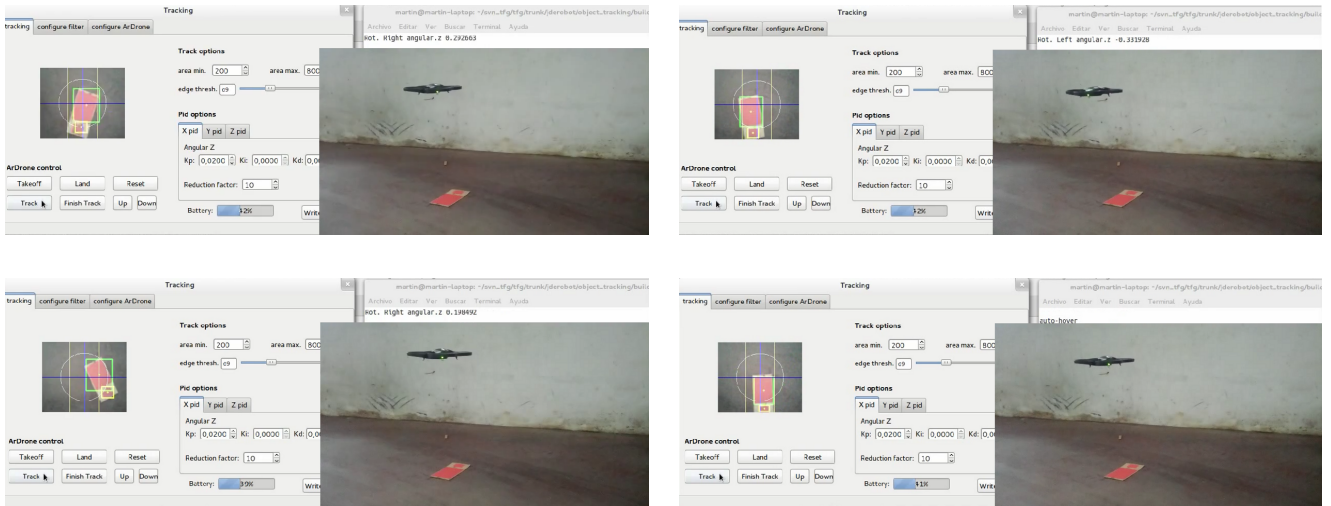


Figura 6.9: Histéresis

Con el árbitro dando prioridad a la posición se consigue que el Ar.Drone primero se posicione sobre el objeto y una vez posicionado, se oriente sobre él. Con este cambio de prioridad se logró que Ar.Drone no saliera de la escena y se evitó que perdiera el objeto de vista. Pero cuando el sistema de actuación trataba de orientar al *drone* con el objeto el cuadricóptero comenzaba a oscilar.

En la bitácora del proyecto ⁶ dentro de la sección *Failed hits* en el apartado *Vertical Tracking x,y,theta* de *object_tracking* se encuentra un vídeo que muestra este comportamiento. En este vídeo se puede ver cómo el *drone* comienza a oscilar cuando está muy cerca de orientarse con el objeto, después de varias oscilaciones Ar.Drone acaba por desorientarse del objeto y comienza de nuevo a intentar posicionarse y orientarse con el objeto. Para solventarlo decidimos crear el área de histéresis (ver sección 5.4.2) para evitar que *object_tracking* comandará órdenes opuestas cuando en realidad no era necesario. Aun así, después de haber superado el problema de la oscilación, los resultados que obtuvimos no eran demasiado buenos, principalmente debido a la dificultad para encontrar en esta configuración unos valores adecuados para las 9 constantes de los controladores PID. Mientras que para el seguimiento frontal se obtuvieron muy buenos resultados con los controladores PID con ajuste manual, para el seguimiento ventral resultó ser una tarea mucho más compleja debido sobre todo al control de orientación.

⁶http://jderobot.org/Amartinflorido-tfg#Vertical_Tracking_x.2Cy.2Ctheta_2

Capítulo 7. Conclusiones

A lo largo del documento se han descrito los objetivos de este Trabajo Fin de Grado, la plataforma hardware utilizada y la infraestructura software realizada para la integración del Ar.Drone en JdeRobot, incluyendo un *driver* y una herramienta para la teleoperación y visualización de sensores. Además, se ha detallado el diseño, la implementación y las pruebas de los dos sistemas de navegación visual desarrollados, uno para que el robot aéreo siga objetos en el suelo (cámara ventral) y otro para que el Ar.Drone siga objetos en 3D percibidos con su cámara frontal. En este capítulo se exponen las conclusiones obtenidas durante todo el proceso y los trabajos futuros que pueden continuar o mejorar al actual.

7.1. Conclusiones

El principal objetivo de este trabajo era proponer una solución para el desarrollo de una aplicación de navegación visual que permitiera al cuadricóptero Ar.Drone el seguimiento fluido de objetos en 3D. Para conseguir dicho objetivo, en primer lugar se llevó a cabo el desarrollo de una infraestructura software que permitiera el control del *drone* y la lectura de sus sensores desde cualquier componente de la plataforma JdeRobot. En segundo lugar, y haciendo uso de la plataforma desarrollada, se diseñó y se construyó una aplicación que, implementando algoritmos de control visual, comanda las órdenes necesarias para que el Ar.Drone realice un seguimiento fluido de un objeto, tanto si el objeto se encuentra enfrente del *drone*, como si se encuentra debajo de él.

Teniendo en cuenta los resultados obtenidos durante las pruebas, detallados en el capítulo de 6, se puede afirmar que se ha logrado el objetivo propuesto. Se han cubierto todas las fases de un proyecto software, desde el análisis de requisitos y su especificación, hasta la realización de pruebas unitarias y de integración, pasando por el diseño e implementación. Además, se ha documentado el progreso mediante vídeos, imágenes y texto en la bitácora del proyecto¹, que es accesible públicamente desde Internet y donde cualquier usuario puede tener acceso a toda la documentación y código generado en este trabajo. El *drone* con los algoritmos desarrollados ha sido expuesto en público en dos ocasiones: en la Semana Europea de la Robótica y en el evento Tech Fest² (a partir de 1h:34min) de la Universidad Rey Juan Carlos.

- El primer subobjetivo del proyecto era el desarrollo dentro de la plataforma JdeRobot de la infraestructura software que permitiera el acceso a los sensores y actuadores del cuadricóptero Ar.Drone 1.0. Se comenzó con un estudio de las plataformas disponibles para el manejo y adquisición

¹<http://jderobot.org/Amartinflorido-tfg>

²<http://chaplin.urjc.es/pcomu/videos.php?id=9579>

de los datos de los sensores de Ar.Drone, desde el SDK oficial de Parrot hasta otros componentes software incluidos en las plataformas ROS o Urbi. Finalmente se decidió seguir la filosofía de ROS y desarrollar un envoltorio que permitiera la interacción de componentes JdeRobot con el SDK oficial de Ar.Drone. El resultado es el componente **ardrone_server** (ver sección 4.1).

Una de las mayores ventajas de **ardrone_server** es el uso de interfaces ICE, tanto las interfaces estándar de JdeRobot (que permiten que cualquier componente obtenga las imágenes del *drone* o enviar ficheros de configuración “en caliente”), como la interfaz específica que permite el control del cuadricóptero desde otros componentes de una manera sencilla y eficiente. Dada la arquitectura distribuida de ICE y la facilidad para implementar componentes en casi cualquier lenguaje de programación, hacen que interactuar con Ar.Drone a través del componente **ardrone_server** sea una tarea sencilla, reduciendo la dificultad que supone trabajar con los dispositivos hardware a bajo nivel. La concurrencia del componente permite que éste pueda responder a las peticiones de otros componentes mientras realiza otras acciones como la obtención de las imágenes del cuadricóptero o el envío de órdenes de movimiento

Utilizando las interfaces ICE que **ardrone_server** ofrece, **uav_viewer** las explota permitiendo la teleoperación del *drone* y el visionado de sus sensores. Este visor ha sido desarrollado para poder operar con otros VANT, siempre y cuando éstos ofrezcan las interfaces necesarias (ver sección 4.2). La interfaz gráfica de éste componente está descrita en un fichero XML, lo que permite una rápida modificación de la estética e incluso la funcionalidad de dicha interfaz. Los distintos hilos que componen la herramienta permiten atender debidamente la interfaz de usuario sin provocar bloqueos en la aplicación. Su diseño concurrente le permite desacoplar la interfaz gráfica de usuario de los hilos de control que gestionan la comunicación con **ardrone_server**.

- El segundo subobjetivo del proyecto era el diseño y desarrollo de una aplicación de navegación visual que gobernase el movimiento del cuadricóptero para el seguimiento de objetos. Para alcanzarlo se desarrolló el componente **object_tracking**, el cual se divide en una parte perceptiva para la detección de objetos en la imagen y en una parte actuadora para el seguimiento de éstos.

En la parte perceptiva (ver sección 5.2) se ha utilizado un filtro de color para detectar el objeto, combinado con un filtro de paso bajo temporal que suaviza las coordenadas estimadas de dicho objeto en la pantalla. El filtro de color se ha realizado en el espacio de color HSV, el cual es más robusto frente a los cambios de luz que el espacio de color RGB, tal y como se ha comprobado empíricamente en la sección 6.1. Aunque el filtro de color funciona mejor en HSV sigue teniendo cierta dependencia de la luz de ambiente. Para trabajar con este filtro se necesitan unas condiciones de luz óptimas, no necesariamente ideales. Es necesario calibrar el filtro de color en función de la luz presente en la escena, para que pueda ajustarse lo máximo posible.

Para la parte actuadora se han implementado dos tipos de seguimiento de objetos. El primero, el seguimiento frontal (ver sección 5.3), está gobernado por 3 controladores PID y se encarga de seguir un objeto en el espacio 3D que se encuentra enfrente del *drone*. El segundo, el seguimiento ventral (ver sección 5.4), es capaz de seguir un objeto que se encuentra debajo del cuadricóptero. Para ello

basar la salida del sistema de actuación en una superficie de control. En este control la altitud del Ar.Drone la gestiona el sistema nativo a bordo. La interacción entre el sistema de a bordo y el sistema de actuación de `object_tracking` provocó algunos inconvenientes que fueron descritos en la sección 6.4.

La dificultad de manejar un controlador PID reside en encontrar unos valores apropiados para las constantes que lo componen. Mientras que para el seguimiento frontal se encontraron unos valores apropiados que ofrecen un seguimiento fluido (ver sección 6.3), en el seguimiento ventral se acabó por desechar la idea de utilizar controladores PID por la dificultad de encontrar dichos valores. La superficie de control resultó ser un método para rebajar la dificultad de encontrar los parámetros necesarios. Además de ser un método sencillo y elegante, ha sido suficiente para lograr que el cuadricóptero siga objetos que se encuentran en el suelo tal y como se vio en la sección 6.4.1.

- El tercer y último subobjetivo fue la validación experimental de la infraestructura desarrollada y de la aplicación de navegación visual que implementa los algoritmos de percepción y de control. En esta fase se realizaron una gran cantidad de pruebas, desde experimentos para justificar el correcto funcionamiento de la infraestructura y los dos seguimientos implementados (ver capítulo 6) hasta pruebas de configuración o pruebas para estudio del comportamiento del sistema a bordo del cuadricóptero que muestran el correcto funcionamiento de la solución alcanzada.

En este proyecto, además de tener una serie de objetivos y sub-objetivos, también se definieron una serie de requisitos que la solución tenía que cumplir.

- El primer requisito era que la infraestructura a desarrollar para Ar.Drone tenía que estar integrada en JdeRobot. Todas las aplicaciones desarrolladas en el proyecto son componentes JdeRobot (ver secciones 4.1, 4.2 y 5.1).
- El segundo requisito indica que la infraestructura debe ser eficiente computacionalmente. Todos los componentes desarrollados en el proyecto se basan en un diseño multihilo (ver secciones 4.1, 4.2 y 5.1) que se aprovechan de las arquitecturas modernas *multicore*. Además, se ha estudiado el comportamiento, la carga y la duración de los ciclos de cada hilo lo cual nos permite ajustar la duración de cada ciclo controlando así el tiempo que un hilo permanece ocioso.
- El SDK oficial de Parrot para el desarrollo de aplicaciones con Ar.Drone, es un SDK complejo y a menudo complicado de entender. Para cumplir con el tercer requisito se implementaron las interfaces ICE (ver sección 4.1.1), que permiten que un desarrollador que quiera hacer aplicaciones con Ar.Drone utilizando `ardrone_server` pueda utilizar el cuadricóptero como si fuera un objeto más de su clase obviando todas las dificultades que supone trabajar con dispositivos de bajo nivel.
- Cumpliendo con el cuarto requisito, los componentes del proyecto utilizan las interfaces `camera` y `remoteConfig` (ver secciones 4.1, 4.2 y 5.1), ambas son interfaces estándar de JdeRobot para la transmisión de imágenes y para la configuración remota de componentes respectivamente. Además, se ha definido la interface `ardrone_control` (ver sección 4.1.1) que es una interface específica para el envío de comandos al *drone* y para la obtención de los valores de los sensores de éste.

- Acordes con el quinto requisito, los componentes `ardrone_server` (ver sección 4.1.3) y `object_tracking` (ver sección 5.5) disponen de ficheros de configuración XML que cargan al inicio de su ejecución los valores almacenados en dichos ficheros. Además, `ardrone_server` permite la modificación en caliente de su configuración aceptando ficheros XML a través de la red implementado la interface `remoteConfig`, estos ficheros pueden ser enviados desde `object_tracking` para modificar valores como la altura máxima del cuadricóptero o el ángulo máximo de inclinación en un determinado eje.
- En los distintos vídeos del proyecto se puede apreciar como el cuadricóptero realiza un seguimiento fluido de los objetos a una velocidad superior a 5 cm/s, que era el sexto requisito.

Este es un proyecto heterogéneo y se han utilizado tecnologías variadas: (1) el desarrollo de software a bajo nivel que, haciendo uso del SDK de Parrot se utilizó para el desarrollo de `ardrone_server`; (2) El desarrollo de interfaces gráficas de usuario con el uso de librerías gráficas que permitieron desarrollar las interfaces de `uav_viewer` y `object_tracking`; (3) Tratamiento y procesamiento de imágenes mediante técnicas de visión artificial que permitieron extraer información sobre el objeto a seguir desde las imágenes capturadas por el cuadricóptero; Y por último, (4) el estudio y desarrollo de algoritmos de control que permiten al *drone* el seguimiento de objetos.

A lo largo de todas estas pruebas se aprendieron bastantes lecciones. Para este trabajo se ha utilizado la primera versión del cuadricóptero Ar.Drone. Sin entrar en las especificaciones técnicas del *drone*, el desarrollo de aplicaciones con este robot y en general con cualquier robot u otro dispositivo hardware conlleva una serie de inconvenientes. Al tratarse de un VANT el primer requisito indispensable para poder realizar aplicaciones sobre él es disponer de un lugar lo suficientemente amplio como para que el *drone* pueda volar sin que colisione con ningún objeto. Sobre todo en el comienzo del desarrollo, es altamente probable cometer errores en el código que provoquen que el cuadricóptero se estrelle, lo que trae consecuencias negativas como la rotura de algún componente del cuadricóptero, objetos que se encuentren en el mismo espacio o incluso daños físicos para las personas que se encuentren cerca. Una ventaja del Ar.Drone frente a otros cuadricópteros es que sus componentes son baratos y fácilmente reemplazables. Durante el transcurso del proyecto se tuvieron que reponer distintos componentes sin que ésto supusiera un gran problema. Además, y pese a las piezas reemplazadas, Ar.Drone es una plataforma muy robusta que soportó numerosos accidentes a lo largo de este trabajo sin sufrir daño alguno.

La versión 1.0 de Ar.Drone tiene una batería de litio de 1000mAh a 11.1V con una autonomía de vuelo de aproximadamente 12 minutos. La carga de la batería con el cargador oficial oscila entre los 35 y 40 minutos. Si todo va bien, esto significa que disponemos de 12 minutos para realizar las pruebas necesarias con nuestro código. Si algo falla y no tenemos más baterías, supone que como mínimo tendremos que esperar 35 minutos para reanudar las pruebas. Por esta razón es muy recomendable tener varios juegos de baterías y registrar todo el proceso que se realiza en las pruebas, desde la grabación en vídeo del vuelo del *drone* hasta el registro de los datos de la aplicación que nos ayuden a identificar el problema con exactitud. A veces puede resultar frustrante la corta vida de las baterías: aparte de disponer de un periodo de tiempo corto para realizar las pruebas, con el uso continuado de la batería la calidad de ésta se deteriora provocando que la autonomía disminuya o que el manejo del cuadricóptero se vea afectado por la falta de

potencia necesaria.

El cuadricóptero dispone de dos cámaras, una frontal y otra ventral. Mientras que la frontal ofrece unas imágenes de una calidad razonable, las imágenes obtenidas por la cámara ventral se alejan de la calidad de la primera. Principalmente por la baja resolución de ésta, que además no se proporciona en una resolución estándar debido a que dicha imagen en realidad está incrustada en otra (*Picture in Picture* como vimos en la sección 3.1.4). Otro aspecto negativo es la imposibilidad de poder obtener de manera simultánea las imágenes de ambas cámaras. El SDK de Ar.Drone proporciona un flujo único para las dos imágenes.

7.2. Trabajos futuros

Quizás la primera tarea sea la mejora de la infraestructura para Ar.Drone. En primer lugar se puede llevar a cabo una actualización de `ardrone_server` para soportar la versión 2.0 de Ar.Drone. En esta nueva versión se han paliado algunos problemas como la calidad de las cámaras que incorpora y la duración de las baterías. Además, se han añadido más sensores e incluso la posibilidad de utilizar un GPS. Para normalizar `ardrone_server` dentro de la plataforma JdeRobot sería necesario reemplazar la interfaz `ardrone_control` por la interfaz `Pose3D`, que es la utilizada por los demás robots dentro de la plataforma para especificar posición y orientación. Con esa nueva versión de `ardrone_server` válida para Ar.Drone 2.0 sería necesario incluir en `uav_viewer` la visualización de los datos proporcionados por los nuevos sensores. Para consolidar la herramienta dentro de JdeRobot sería necesario añadirle la interfaz estándar `Pose3D`. De este modo, `uav_viewer` podría funcionar también con otros robots aéreos como el Phantom FX-61 (ver sección 1.3.3). Igualmente, se podrían ampliar las funcionalidades de la herramienta ofreciendo la posibilidad de teleoperar un VANT mediante *joysticks* o la capacidad de replicar las operaciones realizadas en ejecuciones previas.

Aprovechando el conector USB de Ar.Drone 2.0 y el sistema Linux que lleva incorporado en su interior, se podría interconectar el cuadricóptero, mediante el puerto USB, con un microprocesador ligero a bordo con mayores prestaciones que el sistema interno. Por ejemplo con el mini-PC MK802 que, entre otras cosas, contiene un procesador de cuatro núcleos. De este modo el mini-PC se encargaría de realizar todas las operaciones de los algoritmos de visión y, a través del puerto USB, comandaría localmente las órdenes al *drone*. Con esta solución el cuadricóptero podría ser totalmente autónomo eliminando así dos de las mayores desventajas de Ar.Drone: (1) la comunicación con el cuadricóptero mediante redes inalámbricas, que limitan su radio de acción e introducen latencia de red en el control y (2) la necesidad de disponer de un PC externo donde realizar el procesamiento de la información obtenida por los sensores.

En lo referente a la detección de objetos, se podría ampliar el estudio y la implementación de nuevos métodos más eficaces para esta tarea, basando la detección y el seguimiento en imagen en otros descriptores del objeto como SURF, SIFT u ORB. Para un mejor seguimiento en 3D, se podrían emplear nuevas estrategias de control que usen la predicción de la posición del objeto en función de las lecturas anteriores (filtros de kalman), control borroso o flujo óptico.

Por último, y para subsanar los problemas derivados de trabajar con un robot real, se podría implementar un modelo 3D del Ar.Drone para el simulador Gazebo. Con la simulación se acelerarían los

desarrollos: gran parte de los ajustes se realizarían en simulación. Los algoritmos de percepción y control se podrían madurar y depurar en el simulador. Con el uso de esta herramienta se abre la posibilidad de disponer de otros robots aéreos simulados como el Phantom FX-61 e incluso la posibilidad de trabajar sin la necesidad de disponer físicamente del cuadricóptero o del espacio suficiente para trabajar de una manera segura con estos vehículos.

Bibliografía

- [1] Aplitop.com. Algoritmo de Douglas-Peucker. <http://www.aplitop.com/webaplitop/subidas/ayuda/mdt/referencia/curvado/default.htm?turl=algoritmodedouglaspeucker.htm>. [Online; accedido 24-Diciembre-2013].
- [2] J. M. Cañas, M. González, A. Hernández, and F. Rivas. Recent advances in the JdeRobot framework for robot programming. *Actas de RobotCity2030 12th Workshop "Robótica Cognitiva". UNED*, pages 1–21, 2013.
- [3] José María Cañas Plaza. Apuntes de la asignatura de Robótica. Ingeniería Técnica Informática de Sistemas. Universidad Rey Juan Carlos.
- [4] José María Cañas Plaza. *Jerarquía Dinámica de Esquemas para la generación de comportamiento autónomo*. Tesis doctoral, Universidad Politécnica de Madrid, 2003.
- [5] Les Earnets. Stanford Cart. <http://www.stanford.edu/~learnest/cart.htm>, 2012. [Online; accedido 19-Noviembre-2013].
- [6] EcuRed. Vehículos aéreos no tripulados. http://www.ecured.cu/index.php/Veh%C3%ADculo_a%C3%A9reo_no_tripulado. [Online; accedido 19-Noviembre-2013].
- [7] Christian González. Drones 'made in Spain'. <http://www.publico.es/477283/drones-made-in-spain>. [Online; accedido 22-Noviembre-2013].
- [8] Maikel González Baile. Plataforma de desarrollo de aplicaciones robóticas: Jderobot 5.1. Proyecto Fin de Carrera, Ingeniería Técnica Informática de Sistemas, Universidad Rey Juan Carlos, 2012-2013.
- [9] Heliworks. ¿Cómo vuelan los helicópteros? <http://www.heliworks.co.cl/volar.htm>. [Online; accedido 19-Noviembre-2013].
- [10] O. Higuera, C. Agüero, J.M Cañas, and S. Millán. Qsearh: búsqueda optimizada aplicada a la calibración de controladores PID en un cuadricóptero autónomo. *Robots de exteriores. Arganda del Rey*, pages 269–288, 2010.
- [11] Óscar Higuera Rincón. Xpider: Desarrollo de un Vehículo Aéreo No Tripulado de Cuatro Motores. Proyecto Fin de Carrera, Ingeniería Informática, Universidad Rey Juan Carlos, 2010-2011.
- [12] InfoUas. Pruebas del prototipo TAURO, proyecto HADA. <http://www.infouas.com/pruebas-del-prototipo-tauro-proyecto-hada/>. [Online; accedido 22-Noviembre-2013].

- [13] Flying Machine Arena. Web del grupo de investigación. <http://www.flyingmachinearena.org/>. [Online; accedido 21-Noviembre-2013].
- [14] Federico Marín Bellón. El primer robot de ajedrez era español y cumple cien años. <http://abcblogs.abc.es/poker-ajedrez/public/post/el-primer-robot-de-ajedrez-era-espanol-y-cumple-cien-anos-14155.asp/>, 2012. [Online; accessed 19-Noviembre-2013].
- [15] B. Menéndez, J. M. Cañas, E. Perdices, J. Vega, R. Salamanqués, F. Rivas, and F. Martín. Programming a Humanoid Social Robot Using the JdeRobot Framework. *Actas de RobotCity2030 "Robots Sociales". U. Carlos III*, pages 193–203, 2013.
- [16] Hans Moravec. To See the World. <http://www.frc.ri.cmu.edu/~hpm/talks/cevo.slides/seegrid.html>, 2004. [Online; accedido 19-Noviembre-2013].
- [17] NASA. Ikhana Mission. http://www.nasa.gov/centers/dryden/news/Features/2007/wildfire_social_10_07.html, 2007. [Online; accedido 19-Noviembre-2013].
- [18] C. Platero. *Apuntes de Visión Artificial*. Universidad Politécnica de Madrid, 2007.
- [19] Tom Scheve. How the MQ-9 Reaper Works. <http://science.howstuffworks.com/reaper1.htm>. [Online; accedido 19-Noviembre-2013].
- [20] SpookyCopters. ¿En qué consiste un cuadricoptero? http://www.spookycopters.com/maxrpm_000019.html. [Online; accedido 19-Noviembre-2013].
- [21] S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. In *Computer Vision, Graphics and Image Processing*, chapter 30, pages 32–46. 1985.
- [22] Alba Tobella. Bruselas estudia un sistema de vigilancia de fronteras con drones. http://politica.elpais.com/politica/2013/09/24/actualidad/1380030208_405736.html. [Online; accedido 22-Noviembre-2013].
- [23] J.F. Vélez, A. B. Moreno, A. Sánchez, and J.L. Esteban. *Visión por Computador*. Universidad Rey Juan Carlos, 2003.
- [24] Electrónica webcindario. Histéresis. <http://electronica.webcindario.com/glosario/histeres.htm>. [Online; accedido 31-Diciembre-2013].
- [25] Austin Weber. Nikola Tesla: Father of Unmanned Vehicle Technology. <http://www.assemblymag.com/articles/87689>, 2010. [Online; accedido 19-Noviembre-2013].
- [26] Wikipedia. Leonardo Torres Quevedo. http://es.wikipedia.org/wiki/Leonardo_Torres_Quevedo/, 2013. [Online; accedido 19-Noviembre-2013].
- [27] Wikipedia. Quadcopter. <http://en.wikipedia.org/wiki/Quadcopter>, 2013. [Online; accedido 19-Noviembre-2013].
- [28] Wikipedia. Robot de Leonardo. http://es.wikipedia.org/wiki/Robot_de_Leonardo, 2013. [Online; accedido 19-Noviembre-2013].
- [29] Wikipedia. Rotor de helicóptero. http://es.wikipedia.org/wiki/Rotor_de_helic%C3%B3ptero, 2013. [Online; accedido 19-Noviembre-2013].