“*Computers are incredibly fast, accurate, and stupid; humans are incredibly slow, inaccurate and brilliant; together they are powerful beyond imagination*.”

Autor desconocido

DEDICATORIA Y AGRADECIMIENTOS

A mi madre, qué sería yo sin ella.

A Rosa, que en las buenas y en las malas ha sido un apoyo constante.

A mi antiguo portátil, que me recordó la importancia de guardar copias de seguridad del trabajo que se realiza.

A Hriday, que siempre consiguió sacar tiempo para echarme una mano cuando la necesitaba.

Al resto de personas, compañeros y profesores, que han hecho de mi etapa en la universidad la mejor de mi vida.

RESUMEN

El alto número de accidentes de drones multirrotor y la severidad de dichos accidentes claman por un avance en medidas de seguridad que permitan a estos aparatos mejorar en este aspecto.

Motivado por esa línea de pensamiento, el presente Trabajo de Fin de Grado consiste en el desarrollo de un sistema de protección reactiva para los UAV (*Unmaned Aerial Vehicles*) de tipo multirrotor que permita al dron comprender cuándo una situación supone un peligro de colisión y actuar en consecuencia para evitarlo.

El sistema creado se divide en dos partes atendiendo a la funcionalidad. Dichas partes son la sensoración y el control. La sensoración se encarga de la obtención, lectura y transmisión de datos relacionados con los objetos que rodean al UAV multirrotor. El control se ocupa del filtrado e interpretación de dichos datos, además del cálculo de las acciones necesarias para evitar choques con obstáculos que presenten peligros potenciales.

Además, el sistema deberá estar integrado en la infraestructura preexistente del grupo de investigación en el que se ha desarrollado el trabajo, *Vision4UAV*, quienes realizan su labor en un *framework* conocido como ROS (*Robot Operating System*).

A lo largo del proyecto se han desarrollado ambas partes en su totalidad partiendo desde cero. A modo de resumen, los principales puntos que se tratan en él son:

En la parte de sensoración:

- Estudio y elección de la tecnología de detección de obstáculos más adecuada para la aplicación concreta, en particular teniendo en cuenta que debe poder ser montado en cualquier dron del grupo de investigación en que se ha realizado el proyecto, y por tanto ha de ser flexible a la par que ligero. Se concluye que la tecnología más óptima es la de ultrasonidos.

- Estudio de mercado sobre diferentes marcas y modelos de sensores de ultrasonidos, comprobando su fiabilidad y robustez ante el ruido generado por los motores del dron. Se prueba un pequeño número de ellos hasta que finalmente se decide optar por utilizar cuatro sensores MaxSonar XL EZ0 de la compañía MaxBotix, principalmente por su excelente cancelación del ruido de las hélices.

- Elección y programación del hardware controlador de dichos sensores de acuerdo a la infraestructura de ROS. Se escogió una tarjeta Arduino Nano, que es compatible con ROS, permite controlar hasta ocho sensores diferentes, posee un veloz microcontrolador ATmega328p y pesa tan sólo 5g.

Las pruebas necesarias con los sensores y la placa de Arduino se llevaron a cabo en el dron Oktokopter de la compañía Mikrokopter, propiedad del grupo Vision4UAV y el Departamento de Automática, Ingeniería Electrónica e Informática Industrial de la ETSII. Se construyó un prototipo del sistema, que se puede montar como un accesorio a cualquier dron de forma sencilla.

En la parte de control:

- Desarrollo y programación de un filtro de Kalman para filtrar las medidas obtenidas por los sensores. Ajuste empírico de las covarianzas de ruido necesarias.
- Modelado y programación de un detector de colisiones que comprende, atendiendo a la evolución de las medidas tomadas, cuándo un obstáculo va a colisionar contra el dron.
- Modelado y programación del control que esquive los obstáculos cuando la colisión sea inminente. Se escogió el uso de un control proporcional-derivativo en cascada, dada la disponibilidad de medidas de velocidad en el dron.

Una vez se hubo desarrollado todo el sistema, se procedió a integrar su funcionamiento dentro del sistema de trabajo del grupo *Vision4UAV*. Para ello, se programaron todas las funcionalidades de la parte de control en C++ como código ejecutable de ROS. Se creó un Nodo de ROS que manejase su funcionamiento y las comunicaciones con el resto de módulos del grupo. Todo el código se contuvo en un paquete al que se llamó droneObstacleAvoider, que se pondrá a disposición de Vision4UAV para que reciban los avisos de colisión y acciones de control calculadas por el sistema.

El último paso que se realizó fue la validación del sistema propuesto. Para ello, se programó un simulador del comportamiento del dron en el entorno de programación de código abierto *Processing*, así como las interacciones entre sensores y obstáculos necesarias,

Con el simulador programado, se procedió al ajuste de parámetros necesario para el correcto funcionamiento de los modelos de control, y se comprobó que dichos modelos se comportaban adecuadamente.

Con esto, el desarrollo del sistema queda completo y validado. Para su uso en plataformas reales, el gran esfuerzo realizado por integrar el código desarrollado en la infraestructura existente hace que el único paso restante sea el ajuste de los parámetros de los controles para cada dron en particular en el que se quiera montar. Esto puede hacerse de forma sencilla, ya que dichos parámetros se leen de un archivo de configuración fácilmente editable.

Palabras clave: Esquivo de obstáculos, UAV multirrotor, sensores de ultrasonidos, *Robot Operating System*.

Códigos UNESCO

120325 Diseño de Sistemas Sensores

120326 Simulación

332407 Control de Vehículos

ÍNDICE

1 INTRODUCCIÓN

1.1 Marco de desarrollo del proyecto

El presente Trabajo de Fin de Grado (TFG) se ha desarrollado en el Departamento de Automática, Ingeniería Electrónica e Informática Industrial de la Escuela Técnica Superior de Ingenieros Industriales (ETSII) de la Universidad Politécnica de Madrid (UPM). Sus tutores han sido D. Pascual Campoy Cervera, catedrático de la escuela, junto con el doctorando José Luis Sánchez-López.

El TFG ha sido llevado a cabo con la concesión de una Beca de Colaboración por parte del Ministerio de Educación, Cultura y Deporte y la UPM para su realización, dentro de la División de Ingeniería de Sistemas y Automática del departamento.

El trabajo se ha conducido en colaboración con el equipo de investigación “Vision4UAV” que forma parte del Centro de Automática y Robótica (CAR) de la UPM.

Consiste en el desarrollo de un sistema accesorio para drones multirrotor que dote al dron al que se acople de la capacidad de reaccionar ante obstáculos que se presenten en su camino y los esquive. Dicho sistema será construido para ser utilizado por los drones que tiene el grupo Vision4UAV con el que se ha trabajado, y por tanto se adaptará a sus características y a la necesidad de ser compatible con ellos.

1.1.1 CAR

El CAR es un instituto de investigación científica que surge como colaboración entre la UPM y el Consejo Superior de Investigaciones Científicas, CSIC. Cuenta con más de 90 personas en personal, y complementa su actividad investigadora con la formación en el máster de Automática y Robótica que se imparte en la ETSII.

Su labor de investigación se centra en las áreas de Automática y Robótica de alto nivel con una visión multidisciplinar que trata de resolver problemas concretos a los que se enfrenta la industria. La investigación en el CAR está organizada en cuatro grandes Unidades de Investigación, a las cuales pertenecen distintos Grupos de Investigación. Las Unidades de Investigación son:

- **Robótica inteligente**. Cuenta con tres grupos de investigación dedicados a avanzar en conducción automática de vehículos, e inteligencia y cognición en máquinas.
- **Robótica aplicada**. Sus dos grupos de investigación se centran en el desarrollo de robots y sus aplicaciones, en especial en robots caminantes, escaladores, o de servicios.
- **Supervisión y control inteligentes**. Orientan su investigación a la evolución en inteligencia de la teoría de control y la automatización. Está formada por dos grupos.
- **Percepción.** Formado por cuatro grupos de investigación, esta unidad trabaja sobre técnicas de visión de máquinas, percepción artificial, localización y exploración en exteriores e interiores y el desarrollo de sistemas autónomos.

1.1.2 Computer Vision Group y Vision4UAV

El Grupo de Visión por Computador, -o *Computer Vision Group* (CVG) en inglés -, es uno de los cuatro grupos que conforman la Unidad de Investigación de Percepción del CAR.

Tiene como reto expandir el estado del arte, desarrollando técnicas de procesamiento de imagen que consigan un mayor entendimiento de los datos para sacarles más partido en nuevas aplicaciones. Entre sus valores están el situarse y formarse en los procedimientos más vanguardistas, ser ambiciosos con sus objetivos y fomentar la colaboración entre otras universidades y países.

Cuenta con dos líneas principales de investigación: Visión para vehículos aéreos no tripulados, en la que trabaja el grupo Vision4UAV, dirigidos por el profesor Pascual Campoy, tutor de este proyecto; y Visión en gestión multimedia, dirigidos por el profesor Sergio Domínguez.

El grupo Vision4UAV es un prestigioso grupo que trabaja por mejorar las técnicas de percepción y toma de decisiones basadas en la información obtenida a través de cámaras digitales. Su objetivo es elevar las capacidades de los UAV para que pueda ser viable su uso en diferentes aplicaciones civiles.

Dichas técnicas incluyen, entre otras, estabilización de imagen, reconocimiento y seguimiento de objetos, reconstrucción tridimensional de espacios, mapeado y localización a través de la visión, control de vuelo y guiado visual.

Es un grupo activo con presencia internacional y que participa regularmente en competiciones sobre la técnica, con excelentes resultados. A continuación se listan algunos de los premios que ha recibido en los últimos años:

Competición IMAV2012 (*International Micro Air Vehicles* 2012).
- Premio al Mejor Funcionamiento Automático (*Best Automatic Performance*).
- Segundo premio a Dinámica de Vuelo en Interiores para Vehículos de ala Rotatoria (*Indoor Flight Dynamics -Rotary Wing MAV*).

Competición IMAV2013.
- Primer premio en Autonomía de Interiores (*Indoor Autonomy*).

Competición IARC (*International Aerial Robotics Competition*) Misión 7, 2014
- Premio a la Mejor Detección de Blancos (*Best Target Detection*).
- Premio al mejor sistema de control (*Best System Control Award*)

El presente proyecto se enmarca dentro de este grupo, habiendo utilizando sus recursos y equipamiento.

1.2 Conceptos previos

1.2.1 UAV

Un vehículo aéreo no tripulado se define como una aeronave que vuela sin tripulación, por tanto, un dron o UAV es un concepto muy general que hace referencia a un gran abanico de vehículos y configuraciones distintas.

Los UAV podrían reunirse en dos grandes categorías: aquellos que son controlados de manera remota por operarios, y aquellos dotados de la capacidad de volar de forma autónoma teniendo sus funciones preprogramadas y que no requieren de la ayuda de personas para realizar sus actividades.

Históricamente, los primeros han sido la gran mayoría y sus aplicaciones principales se centraban en el área militar. Sin embargo, con el paso de los años aparecen con cada vez más frecuencia drones automáticos con aplicaciones civiles y de servicios. Este auge es debido en gran medida a la aparición de drones multirrotor.

1.2.2 UAV multirrotor

Los UAV multirrotor son aeronaves sustentadas por más de dos hélices, como la que muestra la figura 1.1. Las configuraciones más típicas incluyen cuatro hélices (cuadricópteros), seis (hexacópteros) u ocho (octocópteros).

Figura 1.1. Cuadricóptero comercial AR.Drone

Diseñar estos drones para conseguir vuelos autónomos presenta diferentes problemas en las áreas de la electrónica y el control automático.

Por un lado, el control tiene que ser capaz de manejar seis grados de libertad, tres de giro y tres de rotación, a través del control de potencia de los rotores de los que dispongan. Además, su modelo de funcionamiento es multivariable y no lineal.

Analizando la aerodinámica del modelo se presentan muchas más dificultades. Por ejemplo, la flexibilidad de las hélices hace que sufran un ligero grado de inclinación al deformarse durante su funcionamiento. O las medidas obtenidas por los sensores pueden verse deterioradas considerablemente por el ruido introducido por los motores.

Por otro lado, la electrónica también presenta varios problemas, siendo el principal de estos la velocidad y capacidad de procesamiento necesarias para manejar el dispositivo y responder a tiempo a las perturbaciones, y procesar de forma eficaz toda la información recogida por los sensores.

Estos dispositivos pueden separarse en dos partes: una de software que incluyen los algoritmos de control, comunicación y percepción de datos; y otra mecánica que comprende la estructura de soporte y todos los componentes físicos que conforman el sistema: motores, batería, sensores, ESCs, microprocesador.

Su control se consigue variando la proporción entre las potencias que consumen sus motores. Para su estabilidad y conseguir un control más avanzado, requieren sensores accesorios como giroscopios, acelerómetros, magnetómetros (unidades inerciales) o sensores de flujo óptico.

A día de hoy, existen multitud de aplicaciones comerciales e industriales de estos dispositivos en los más diversos ámbitos, desde la inspección industrial hasta las áreas de salvamento, seguridad, y agricultura. La gran popularidad que ha obtenido este tipo de UAV ha obligado a los estados a legislar sobre su uso, ya que hasta hace muy poco no había ninguna legislación vigente. En España la Agencia Estatal de Seguridad Aérea recoge las normas sobre el uso de drones y aclara que solamente se podrá hacer uso de ellos en zonas habilitadas para tal fin.

A pesar del gran desarrollo que han sufrido en los últimos años, cuentan también con limitaciones importantes en temas como la percepción del entorno, seguridad o autonomía, sobre los que todavía es posible conseguir un notable desarrollo.

Este trabajo de fin de grado pretenderá mejorar en el área de la seguridad, como se explicará en apartados posteriores.

1.2.3 Modelado dinámico de un UAV multirrotor

Para el presente trabajo, se trabajará con un modelo simplificado de las dinámicas que rigen el movimiento de los drones multirrotor.

En primer lugar, se definen los ángulos de Euler. Los ángulos de Euler son tres ángulos, introducidos por el matemático Leonhard Euler, que describen la orientación tridimensional de un sólido rígido, o de la relación entre dos sistemas de referencia distintos.

Estos tres ángulos son: θ o ángulo de cabeceo (Pitch), φ o ángulo de guiñada (Roll) y ψ o ángulo de alabeo (Yaw). La siguiente figura muestra el significado de cada uno de estos ángulos, utilizados continuamente en la aviación:

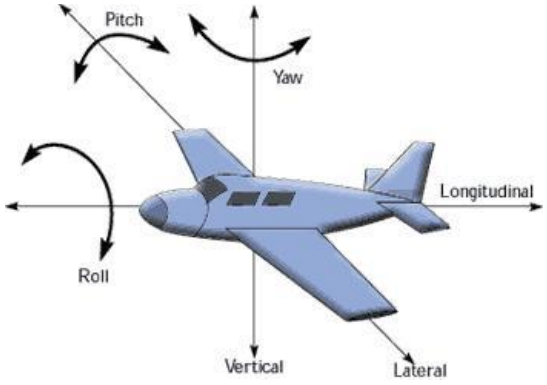

Figura 1.2. Ángulos de Euler en una aeronave

A la hora de hacer un modelo matemático del comportamiento de un UAV multirrotor se pueden incluir más o menos parámetros, que harán que el modelo sea más o menos preciso. Para la aplicación pensada, se usará un modelo bastante simple, que no incluye parámetros como el ángulo que adquieren las palas de las hélices cuando el cuadricóptero se desplaza de manera vertical.

Además, se limitará a describir la capa más alta de las físicas que gobiernan estos aparatos, sus ecuaciones de movimiento, no resultando necesaria la inclusión de capas más bajas como la potencia individual de cada motor o los pares resultantes.

Las ecuaciones que rigen el sistema, son:

$$\ddot{X} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} -(\sin\theta \cdot \cos\varphi \cdot \cos\psi + \sin\psi \cdot \sin\varphi) \cdot \dfrac{E}{m} \\ -(\sin\theta \cdot \cos\varphi \cdot \sin\psi + \cos\psi \cdot \sin\varphi) \cdot \dfrac{E}{m} \\ g - \cos\theta \cdot \cos\varphi \cdot \dfrac{E}{m} \end{bmatrix}$$

1.3 Motivación e impacto

Este proyecto está motivado principalmente por el alto número de accidentes que ocurren a diario relacionados con UAV de tipo multirrotor. Con cada vez más dispositivos de este tipo volando entre nosotros, se vuelve necesario realizar un esfuerzo por aumentar la consistencia y seguridad de los drones.

El mercado de drones tuvo un volumen de ventas de aproximadamente 84 millones de dólares en 2014 con 250.000 unidades vendidas, y presenta una tendencia claramente alcista con pronósticos que sitúan el volumen de ventas de UAV comerciales para 2018 en hasta 300

millones y en hasta 1270 millones para 2020 (PRNewsWire, 2015), con el consiguiente aumento de fallos, choques y accidentes de los mismos de mantenerse el sistema de funcionamiento actual.

Se hace necesario entonces desarrollar sistemas y tecnologías dedicadas a combatir las limitaciones de estos dispositivos y conseguir reducir las situaciones en las que los drones sufran accidentes.

La sensibilidad de los UAV multirrotor los convierte en aparatos extremadamente frágiles. Tanto es así, que el más leve de los accidentes suele implicar el paso de la unidad por mantenimiento. Las reparaciones necesarias abarcan desde la sustitución de cualquiera de sus componentes, la recalibración de los mismos, hasta, incluso, la necesidad de comprar un equipo completamente nuevo con su consecuente gasto.

Además del elevado coste económico que supone lo anterior, se suma una inconveniencia adicional: el tiempo invertido en la reparación suele ser extenso debido a la tardanza que supone el encargo y recepción de piezas.

En consecuencia, el accidente de un UAV no sólo constituye una gran pérdida económica, sino que retrasa significativamente las actividades programadas, retraso que puede alargarse hasta varios meses.

También, aunque constituyan una pequeña minoría, es preocupante la cantidad de accidentes en los que resultan heridas personas. Y es que en todos aquellos casos en que el dron sea utilizado cerca del hombre, el fallo de uno de estos vehículos es potencialmente muy peligroso, ya que estos UAV tienen la capacidad de causar heridas y lesiones considerables.

El desarrollo de un sistema que sea capaz de detectar a tiempo posibles colisiones y evitarlas durante el vuelo es, por tanto, de gran interés. Su ejecución contribuiría a reducir el número de víctimas de accidentes de UAV así como el coste en tiempo y dinero que supone el arreglo de los daños recibidos por el dron, consiguiendo, en definitiva, mejorar la seguridad de uso de estos dispositivos.

En particular, las investigaciones de los laboratorios del departamento de Electrónica y automática suelen requerir las capacidades de los UAV multirrotores, siendo, como se viene diciendo, delicados y costosos. Con el sistema que se presenta se pretender aumentar la seguridad de los usuarios y de las pruebas realizadas por los UAV.

1.4 Estado del Arte

De acuerdo con la gran relevancia del problema y con el elevado potencial de mejora sobre la técnica actual, diferentes equipos y compañías han investigado formas de conseguir el esquivo de obstáculos. Tal ha sido así que recientemente han comenzado a aparecer productos que ya incorporan diferentes tecnologías para evitarlos. En este apartado se recogen una variedad de ellos, cada uno de los cuales soluciona el problema de forma diferente:

Zano

Uno de los primeros drones comerciales que incorporó un sistema de esquivo de obstáculos fue el dron Zano. Zano es un UAV cuadrirrotor de pequeñas dimensiones y relativamente

ligero que se comenzó a financiar en noviembre de 2014 en la plataforma online de crowdfunding Kickstarters.

Este pequeño aparato persigue a su usuario y le realiza fotos desde las alturas, mientras esquiva los obstáculos que va encontrando durante su trayectoria.

Figura 1.3. Dron Zano

Para esquivar los obstáculos con los que se topa hace uso de cuatro parejas de sensores de infrarrojos (IR), que se encuentran distribuidos de forma simétrica alrededor del UAV apuntando hacia delante, izquierda, derecha y detrás. Así el dron consigue una cobertura en todas las direcciones del plano.

Además de los sensores IR, Zano incorpora un sensor de ultrasonidos en la zona inferior del aparato, orientado de forma que pueda calibrar y controlar la altura sobre el suelo a la que se encuentra.

En este caso, el esquivo de obstáculos está integrado de forma natural en el dispositivo, no siendo un accesorio como se pretende conseguir. A pesar de ello, es interesante el uso de los sensores IR y su disposición.

eBumper

Un ejemplo de un sistema que sí es accesorio es el llamado eBumper. Desarrollado por la compañía PanoptesUAV, el eBumper puede añadirse a un pequeño catálogo de drones comerciales que han sido éxito de ventas. Los modelos compatibles con los que trabajan son los siguientes:

- DJI Phantom 2
- Phantom 2 Vision
- Phantom 2 Vision+
- 3DR Iris

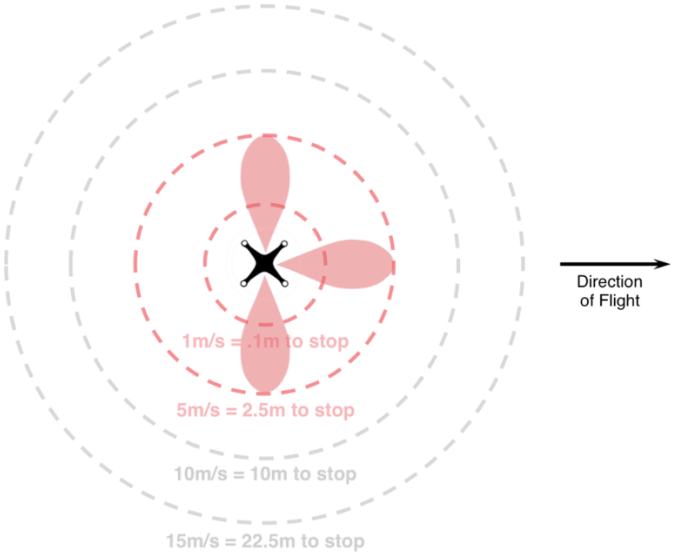

Figura 1.4. Sistema de detección eBumper

Utiliza cuatro sensores de ultrasonidos para detectar los obstáculos. Los sensores están colocados siguiendo la siguiente distribución: Uno apuntando hacia la parte delantera, uno apuntando hacia cada lado, y el último apuntando hacia la parte inferior.

En este caso y como ventaja adicional, el eBumper no se limita a cumplir la función de la detección de obstáculos, sino que incorpora también el control que reacciona a ellos y los esquiva cuando se detecta una colisión inminente integrado de forma natural en el propio accesorio.

Entre las principales cualidades del uso de este sistema se encuentran: su sencilla instalación, que posee un rango bastante aceptable para la detección de obstáculos con pocas zonas muertas y su bajo coste, dado que hablamos de un mecanismo relativamente barato. También cuenta con algunas limitaciones de las cuales la principal es el escaso número de modelos compatibles disponibles. Como se verá más adelante, es el sistema en que más se ha inspirado este proyecto.

eXom

A la vanguardia del estado del arte, se encuentra un sistema más complejo portado por el UAV eXom de la compañía SenseFly. eXom es un dron profesional concebido para ser utilizado en inspecciones industriales, realizar modelado tridimensional o crear mapas de zonas en alta resolución.

El esquivo de obstáculos lo consigue mediante la fusión de cinco cámaras de alta resolución y cinco sensores de ultrasonidos que lo rodean y observan en todo momento su entorno, como se muestra en la figura 1.5:

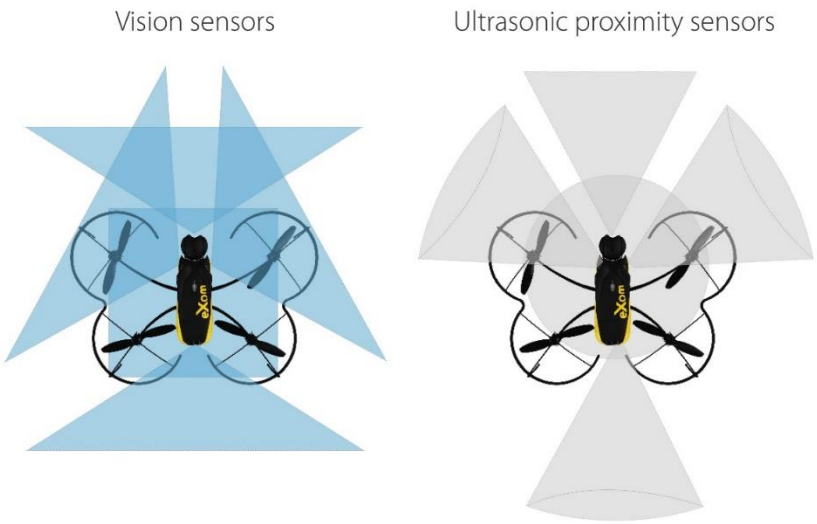

Figura 1.5. Sistema de esquivo de obstáculos del UAV eXom de SenseFly.

Es un sistema avanzado, que fusiona los datos de ambos tipos de sensores, y consigue una mejor detección de obstáculos, más precisa, con menos zonas ciegas que los sistemas anteriores. Por contrapartida, requiere mucho más poder de procesamiento, y tiene un coste y complejidad mayores.

Como estos, existen otros sistemas existentes de *Sense and Avoid* con diferentes aproximaciones. Del estudio del estado del arte, se concluye que las seis principales tecnologías de esquivo de obstáculos son: Sensores de Infrarrojos, Sensores de Ultrasonidos, Cámaras, Cámaras estéreo, LIDAR y LIDAR rotatorio. En capítulos posteriores se analizarán detalladamente.

1.5 Aplicación práctica

El proyecto se ha desarrollado con una mentalidad práctica, teniendo presente en todo momento su inclusión en drones del grupo Vision4UAV y en su entorno de programación. El modelo concreto para el que se ha diseñado es una versión modificada del UAV Oktokopter XL de la compañía Mikrokopter con el que cuenta el grupo.

1.5.1 Oktokopter

Como indica su nombre, el Oktokopter es un UAV de estilo multirrotor que incorpora ocho rotores, en lugar de los más convencionales cuatro. Esto le confiere una mayor capacidad de carga que aumenta hasta los 5kg.

Cuenta con diferentes sensores como acelerómetros y giroscopios, el más interesante de los cuales en relación a este proyecto es un sensor de flujo de campo óptico Px4Flow. Dicho sensor permite medir la velocidad con respecto al suelo, lo que se aprovechará en el diseño del sistema.

1.5.2 Stack

El sistema que Vision4UAV ha ido desarrollando a lo largo de sus años de actividad está recogido en lo que se conoce “CVG Quadrotor Swarm Stack”, al que se referirá de ahora en adelante como *Stack*.

El *Stack* es un proyecto que reúne la labor de programación que hay detrás del control de los UAV que se manejan en Vision4UAV. Está organizado en módulos independientes, cada uno de los cuales comprende una funcionalidad determinada. A lo largo de todos los módulos se recogen todas las funcionalidades, interfaces de comunicación y control, archivos de configuración, librerías, y resto de utilidades disponibles.

Los módulos del *Stack* están programados en el lenguaje de programación C++, utilizando la infraestructura de ROS.

1.6 ROS

El Sistema Operativo de Robots o ROS (del inglés *Robot Operating System*) es una infraestructura digital de software libre desarrollada sobre Ubuntu que se creó con el objetivo de estandarizar y facilitar la comunicación entre las distintas partes y sensores de los equipos robóticos.

Comenzó a desarrollarse bajo el nombre de *Switchyard* en los laboratorios de inteligencia artificial de la universidad de Stanford como parte del proyecto de Stanford STAIR (*Stanford AI Robot*).

En 2007, la incubadora robótica Willow Garage decidió impulsar el proyecto con fuertes recursos y personal y extender sus conceptos de estandarización de comunicaciones y compartimentación del código, renombrándolo ROS. Desde entonces ha crecido hasta tener decenas de miles de usuarios alrededor del mundo, empujado por las colaboraciones de multitud de expertos y su decisión de ser de código abierto.

ROS es un sistema multiplataforma, compatible con multitud de robots existentes. Además, este *framework* permite el uso de varios lenguajes de programación, siendo los principales C++ y Python.

A lo largo de su historia, han existido 9 distribuciones de ROS ilustradas por tipos de tortugas, que se muestran en la siguiente tabla:

Tabla 1.1. Versiones de ROS

Distribución	*Fecha de lanzamiento*	*Símbolo*
9 - ROS Jade Turtle	23 de mayo de 2015	
8 - ROS Indigo Igloo	22 de julio de 2014	
7 - ROS Hydro Medusa	4 de septiembre de 2013	
6 - ROS Groovy Galapagos	31 de diciembre de 2012	
5 - ROS Fuerte Turtle	23 de abril de 2012	
4 - ROS Electric Emys	30 de agosto de 2011	
3 - ROS Diamondback	2 de marzo de 2011	
2 - ROS C Turtle	2 de agosto de 2010	
1 - ROS Box Turtle	2 de marzo de 2010	

El *Stack* se encuentra actualizado en la octava distribución, *ROS Indigo Igloo.* Esta será la versión en que se centre el desarrollo del sistema de esquivo de obstáculos.

1.6.1 Nodos

El entorno de ROS está estructurado en programas individuales llamados Nodos. Cada Nodo ejecuta un proceso aislado del resto, de forma que la programación dentro de ROS está pensada para ser modular. De este modo, si un Nodo falla, el resto de Nodos no se ven afectados.

Los Nodos pueden comunicarse entre sí escribiendo Mensajes en lo que se conoce como Topics, o a través de la invocación de Servicios.

1.6.2 Topics

En ROS existen dos figuras básicas conocidas como Suscriptor y Publicador.

- Un **Publicador** es un Nodo que publica mensajes en un foro, conocido como *Topic*.
- Un **Suscriptor** es un Nodo que se suscribe a un *Topic* y recibe todos los mensajes que encuentra en él.

Los Nodos no están limitados a ser sólo Publicadores o Suscriptores, sino que pueden, y de hecho es lo más común, ser ambas cosas a la vez, e incluso publicar en varios *Topics* diferentes a la vez y estar suscritos también a varios.

Los *Topic*, por lo tanto, son buses de comunicaciones nombrados sobre los que los Nodos intercambian mensajes entre sí. La figura 1.6 muestra un ejemplo de arquitectura de comunicación que puede encontrarse en ROS.

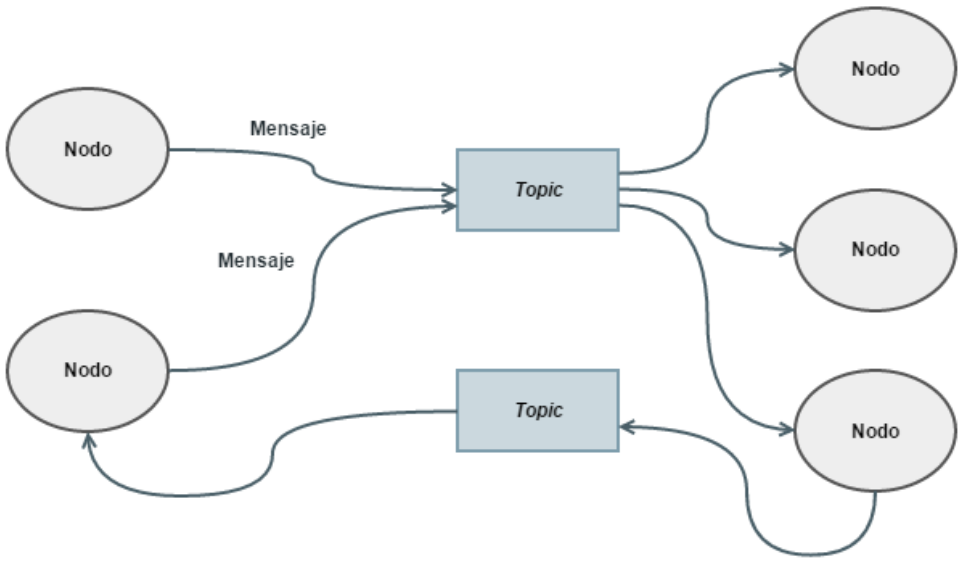

Figura 1.6. Arquitectura ejemplo de ROS.

Así, por ejemplo, un Nodo que se encarga de medir la altura del agua en un tanque puede publicar en el *Topic* "/altura_tanque", y un Nodo que se encarga de cerrar un grifo cuando el nivel de agua del tanque ha llegado a cierta altura tendrá que suscribirse a ese mismo *Topic* "/altura_tanque" para recoger los datos enviados por el primer Nodo y realizar sus operaciones.

1.6.3 Servicios

Además de la comunicación vía la publicación de mensajes en *topics* y suscripción a esos *topics* por parte de nodos, existe otro protocolo de comunicación en ROS, conocido como Servicio.

El Servicio en ROS se utiliza en los casos en que se requieren interacciones de petición-respuesta entre Nodos. Para estos casos no es apropiado el paradigma que define el protocolo Publicador/Subscriptor en el que la comunicación es unidireccional y compartida entre varios nodos.

La comunicación en un Servicio está formado por un par de mensajes, el mensaje Petición (*Request*) y el mensaje Respuesta (*Reply*); y un par de Nodos, uno que actúe como servidor y otro como cliente. El servidor alojará el código que se ejecutará cuando se llame el Servicio, y el cliente será el Nodo que inicie la comunicación.

El cliente invocará el Servicio enviando el mensaje *Request*, y el servidor lo resolverá, realizará las actividades que estén programadas en él, para después devolver el mensaje *Reply* al Nodo cliente. A continuación, la figura 1.7 representa esquemáticamente el funcionamiento de una llamada a un servicio.

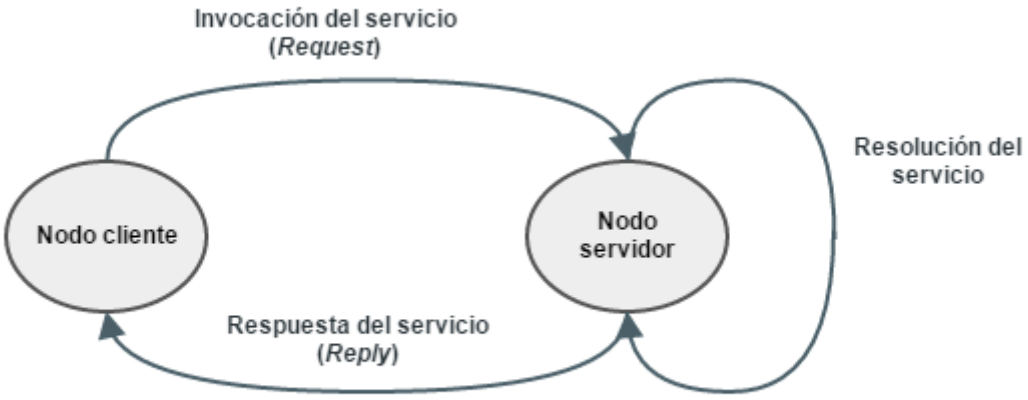

Figura 1.7. Esquema de comunicaciones en un servicio en ROS.

1.6.4 Mensajes

Toda forma de comunicación a través de ROS se produce en forma de mensajes. Los mensajes son estructuras de datos determinadas que incluyen los campos que se consideren necesarios para enviar.

Algunos mensajes básicos vienen incluidos por defecto en las distribuciones de ROS, pero también es posible crear de forma sencilla tipos de mensaje nuevos. Para ello, los campos que vaya a contener el mensaje a crear han de ser descritos en un lenguaje neutral llamado

"*Interface Definition Language*" (IDL), que siguen una estructura sencilla que consta de dos partes. La primera parte definirá el tipo al que pertenece el campo, y la segunda el nombre que recibirá.

Un ejemplo de declaración en IDL de un mensaje típico de ROS, es el tipo Header, tiene el siguiente aspecto:

```
uint32 seq
time stamp
string frame_id
```

Esa breve definición será suficiente para que al compilar el entorno de ROS el tipo de mensaje se genere y pueda utilizarse en el futuro.

1.6.5 Workspace y Paquetes

Para trabajar en ROS, es necesario que el código esté contenido en lo que se conoce como *Workspace* o entorno de trabajo, la mayor unidad organizativa posible y en la que se pueden incluir uno o varios proyectos.

El código dentro de los Workspace se organiza en Paquetes (Packages). Cada Paquete puede contener diferentes archivos, como Tipos de mensaje, Servicios, Nodos, ficheros de configuración, e incluso programas de terceros.

Formalmente, un Paquete no es más que un directorio dentro del Workspace que incluye un archivo con el nombre "*package.xml*". Dicho archivo contiene la descripción del paquete y en él se encuentran sus dependencias, sus propiedades (como su nombre o el de sus autores), y se definen sus contenidos.

1.7 Filtro de Kalman

Para parte del proyecto se requerirá la implementación de un filtro de Kalman. Este apartado pretende introducir las bases teóricas sobre las que se asienta y describir brevemente su funcionamiento.

Un filtro de Kalman es un algoritmo recursivo de fusión de sensores e información que filtra una mediciones tomadas por sensores, las cuales pueden contener ruido u otras fuentes de error, y genera estimaciones de las variables de interés, consiguiendo en la mayoría de los casos una mayor precisión que la que se obtendría al realizar una única medición.

Para realizar sus cálculos, el filtro de Kalman necesita un modelo lineal de la dinámica del sistema, así como las señales de control que intervengan en el sistema y las mediciones obtenidas a lo largo del tiempo.

El algoritmo en que se basa para estimar el estado posterior funciona en dos pasos. En primer lugar, se realiza una predicción del estado posterior con los datos estimados del estado presente. En segundo lugar, cuando se recibe la medición del estado posterior, esta corrige a

la predicción realizada con una ponderación que se conoce como ganancia Kalman, que otorga mayor peso a la parte con menor incertidumbre.

1.7.1 Modelo

Para implementar un filtro de Kalman, es necesario realizar un modelo lineal del sistema dinámico atendiendo a la siguiente estructura:

Se definen el vector de estado, x_k, que recoge las de variables de estado en el instante k, y el vector de control u_k el cual contiene las señales de control en ese instante. Se modela el estado actual como una combinación lineal del estado anterior y las señales de control y ruido actuales:

$$\mathbf{x}_k = \mathbf{F}_k\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k + \mathbf{w}_k$$

Donde:

- $\mathbf{F}_k$ es una matriz que modela la transición del estado anterior al posterior.
- $\mathbf{B}_k$ es una matriz que modela el efecto de las señales de control sobre el estado.
- w_k es el ruido en el proceso, que se asume gaussiano centrado en cero con varianza Q_k.

Al mismo tiempo, se define z_k como el vector que contiene las medidas tomadas por los sensores en el instante k, el cual depende del estado actual y de la siguiente forma:

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k$$

Donde:

- $\mathbf{H}_k$ es el modelo de observación que enlaza las medidas tomadas con las variables reales
- V_k es el ruido presente en la medición, que también se asume gaussiano, centrado en el cero, y con varianza R_k.

1.7.2 Ecuaciones del filtro de Kalman

Una vez definido el modelo, se procede a explicar el mecanismo de estimación del estado que sigue el filtro de Kalman. Se realiza en dos etapas, la etapa de predicción y la etapa de corrección. En cada etapa se irán calculando una serie de parámetros intermedios. Se introduce la notación $\hat{\mathbf{x}}_{n|m}$, que representa la estimación del valor del vector x en el instante n con la información obtenida hasta el instante m.

Predicción

Predicción (a priori) del estado estimado: $$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_k\mathbf{u}_k$$

Predicción (a priori) de la covarianza estimada: $$\mathbf{P}_{k|k-1} = \mathbf{F}_k\mathbf{P}_{k-1|k-1}\mathbf{F}_k^{\mathrm{T}} + \mathbf{Q}_k$$

Corrección

Residuo de la medida $$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k\hat{\mathbf{x}}_{k|k-1}$$

Residuo de la covarianza $$\mathbf{S}_k = \mathbf{H}_k\mathbf{P}_{k|k-1}\mathbf{H}_k^T + \mathbf{R}_k$$

Ganancia de Kalman óptima $$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}_k^T\mathbf{S}_k^{-1}$$

Corrección (a posteriori) del estado estimado: $$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k\tilde{\mathbf{y}}_k$$

Corrección (a posteriori) de la covarianza estimada: $$\mathbf{P}_{k|k} = (I - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_{k|k-1}$$

La naturaleza recursiva de estas ecuaciones permite estimar el estado y covarianza del sistema partiendo de datos únicamente de la etapa anterior. Esto posibilita que el filtro pueda trabajar en tiempo real según se van recibiendo nuevas mediciones.

2 OBJETIVOS

El principal objetivo del presente trabajo de fin de grado consiste en el desarrollo de un sistema sensorial de protección reactiva para UAV de tipo multirrotor utilizable por el grupo Vision4UAV.

El sistema debe ser capaz de detectar obstáculos que aparezcan en el camino del dron y analizar si estos suponen un peligro de posible colisión. En caso de serlo, debe ser capaz de tomar el control y esquivar dicho obstáculo con total seguridad. Es lo que se conoce como *Sense and Avoid*

Para ello, se plantean los siguientes objetivos:

1. Búsqueda del hardware y las técnicas más adecuadas para realizar la tarea propuesta de *Sense and Avoid.*

2. Concepción y desarrollo de un sistema completo y flexible a partir de los resultados de la búsqueda anterior, que se encargará tanto de la parte de *Sense* (medida y análisis de los obstáculos circundantes), como de la parte de *Avoid* (decisión sobre cuándo y cómo esquivarlos).

3. Integración del sistema desarrollado en la infraestructura ROS y el trabajo previo del grupo Vision4UAV.

4. Validación y testeado del sistema desarrollado en un entorno de simulación, que también habrá de ser programado como parte del proyecto.

Es un proyecto multidisciplinar, que incluye tareas de búsqueda de información y estudios de mercado, concepción y montaje físico de un sistema, programación y desarrollo de software de control y simulación, y modelado y validación del sistema pensado.

3 METODOLOGÍA

3.1 Sistema propuesto

Para cumplir los objetivos expuestos en el capítulo anterior, se plantean los siguientes requisitos específicos que debe cumplir el sistema:

Como requisitos principales, se deberá conseguir:

1. Fiabilidad en la detección y esquivo de obstáculos
2. Robustez ante fallos y situaciones desconocidas
3. Integración total en la arquitectura del *Stack* de CVG

Además, se procurará:

4. Conseguir un sistema configurable, que sea flexible para distintas aplicaciones y necesidades de los drones que lo incluyan.
5. Conseguir un sistema versátil, que pueda añadirse, quitarse, o trasladarse a diferentes drones sin necesidad de realizar grandes cambios en su diseño.
6. Conseguir un sistema lo más ligero posible, que suponga poca carga para los drones que lo porten.

En acuerdo con dichos objetivos, se propone el siguiente sistema detector de obstáculos, formado por cinco componentes agrupadas en dos partes funcionales:

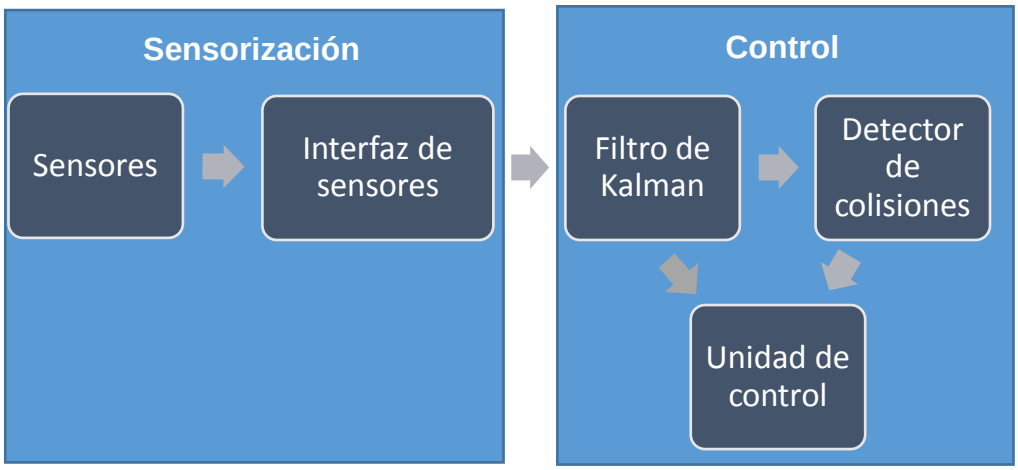

Figura 3.1. Componentes del sistema

Sensorización (*Sense*)

Se encarga de la realización de medidas, su lectura, y su envío al ordenador de a bordo, e incluye el hardware de medición (sensores) así como el de lectura (interfaz) y la programación de los mismos. Incluye las partes de:

- Sensores
- Interfaz de lectura de sensores

Control de esquivo (*Avoid*)

Será el que reciba la información, la filtre y la analice, y decida si presentan un peligro de colisión y en caso afirmativo la esquive. Las componentes que incluye son:

- Filtro de Kalman
- Detector de colisiones
- Unidad de control

3.2 Desarrollo del proyecto

El proyecto se desarrollará atendiendo a la siguiente metodología:

Se realizará un estudio sobre las principales posibilidades tecnológicas de detección de obstáculos para su uso como sensores y se seleccionará una o varias para su uso en el proyecto.

Se realizará un breve estudio de mercado, atendiendo a su viabilidad, sobre diferentes marcas y modelos de sensores del tipo que se escoja en el paso anterior.

Se estudiará qué hardware es el más adecuado para actuar como interfaz entre los sensores y el ordenador de a bordo del UAV.

Se programará dicha interfaz de forma que las medidas de los sensores sean accesibles de forma fácil y ordenada por el UAV.

Se modelará el sistema y se programarán de acorde al modelo el filtro de las medidas de los sensores, el detector de colisiones, y la unidad de control, de forma que se integren de forma natural en el *Stack*.

Se programará un simulador cinemático de un dron y los modelos descritos.

Se validará el diseño del sistema en el simulador y se analizarán los resultados obtenidos a lo largo de todo el proyecto.

4 SENSORACIÓN

4.1 Análisis de tecnologías de medición de distancias

Para el desarrollo de un sistema eficaz de detección de obstáculos, no sólo es necesaria la capacidad de detectar si hay obstáculos cerca o no, sino que también es de suma importancia conocer la distancia a la que se encuentran los mismos.

Se procederá a realizar un análisis sobre las características y su adecuación al sistema que se pretende desarrollar de las seis principales tecnologías que permiten conocer distancias a obstáculos, siendo estas:

1. Sensores de infrarrojo (IR).
2. Sensores LIDAR.
3. Sensores LIDAR rotatorios.
4. Sensores de ultrasonidos.
5. Cámaras.
6. Cámaras estéreo.

Posteriormente se escogerán una o varias de ellas para incluir sus sensores en el sistema a desarrollar. Los criterios de selección que se utilizarán para realizar dicha selección serán los siguientes:

- Peso del sistema.
- Sencillez de implementación y versatilidad.
- Rango y ángulo de detección. Presencia de zonas muertas.
- Coste.
- Capacidad de procesamiento necesaria.
- La existencia cualquier otra característica que imposibilite su aplicación.

4.1.1 Sensor IR

Los sensores de IR se basan en la emisión de luz en el espectro infrarrojo y la detección de la posible luz reflejada para calcular la distancia al objeto que ha hecho rebotar el haz. Se componen de dos partes, un LED emisor de pulsos, y un receptor diseñado para detectar luz de la misma longitud de onda que la que es emitida por el LED.

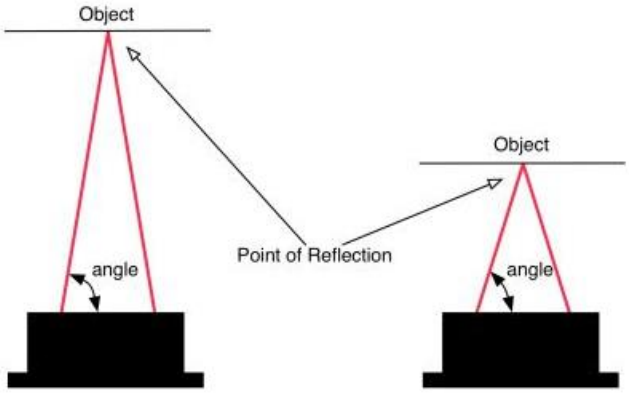

Figura 4.1: Funcionamiento de un sensor de infrarrojos.

En la figura 4.1 se muestra un esquema de su funcionamiento. El LED emite pulsos de luz que al volver inciden con un determinado ángulo sobre el receptor.

Conociendo la distancia entre emisor y receptor, la lejanía del obstáculo puede calcularse por medio de triangulación sencilla de la siguiente manera:

Sea s la separación entre los dos componentes, d la distancia al objeto, y α el ángulo de recepción del haz reflejado. La distancia podría calcularse de la siguiente manera:

$$d = \frac{s}{2} \cdot tg(\alpha) \tag{4.1}$$

Sin embargo, estos sensores no calculan la distancia. Utilizan una óptica interna y sensores fotovoltaicos para generar un voltaje analógico de salida que no es lineal con la distancia. Se presentan en varios modelos con diferentes rangos de actuación, cada uno de los cuales tiene una curva de respuesta distinta, aunque todos ellos presentan un aspecto como el mostrado en la figura 4.2:

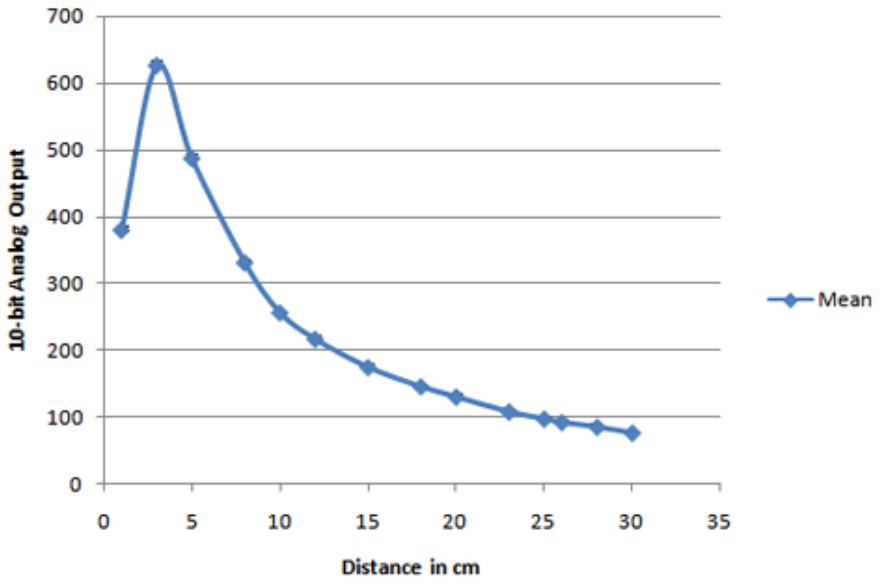

Figura 4.2. Respuesta del sensor de infrarrojos SHARP 4-30cm.

Una característica desfavorable de estos sensores, como puede comprobarse en la gráfica anterior, es que los objetos muy cercanos (menor que 4cm para este sensor el concreto) generan el mismo valor de salida que objetos a distancias ligeramente superiores, pudiendo confundirse ambas situaciones. Este fenómeno ocurre y resta fiabilidad a las medidas tomadas por estos sensores a corto rango.

Otra desventaja que poseen que ver con su haz de detección considerado en el contexto de la aplicación que se pretende realizar. Dicho haz es relativamente estrecho, lo que se traduce en que cada sensor es capaz de percibir distancias de sólo una pequeña parte del entorno que le rodea, dejando muchas zonas muertas.

Además, los valores que arroja en entornos abiertos son poco fiables debido a las interferencias producidas por la luz solar ambiental.

Por contrapartida, son sensores muy sencillos de implementar, ligeros y baratos, con lo que su uso en grupos es asequible. También suponen una muy baja carga de procesamiento. Estas características lo sitúan como una buena elección para sistemas baratos y que no necesitan una fiabilidad elevada.

4.1.2 Sensor LIDAR

LIDAR, acrónimo de *Laser Imaging Detection And Ranging* (Detección y Medición de Distancias por Iluminación láser, en inglés) es una tecnología que, al igual que los IR, también emite un pulso de luz y analiza su reflejo para realizar sus mediciones. Las principales diferencias entre estas tecnologías son la forma de luz utilizada y la forma de calcular la distancia, que ya no serán LED infrarrojos ni se basarán en el ángulo de la luz reflejada.

Los sensores LIDAR cuentan también con dos componentes. La primera es un emisor que iluminará el objeto a medir, que en este caso emitirá pulsos de láser puntuales. La luz de dichos láseres puede emitirse en un gran espectro de longitudes de onda y generalmente se reflejará en el objetivo mediante retrodispersión.

La retrodispersión es un fenómeno de reflexión dispersa en la que el rayo incidente deja de estar enfocado al ser reflejado y sale en todas direcciones. Esto permite al segundo elemento, el receptor, recibir el haz reflejado mucho más fácilmente que si el láser siguiera enfocado en todo momento.

Una vez el haz llega al receptor, existen dos procedimientos de cálculo de la distancia en función del tipo de láser:

- LIDAR emisor de pulsos. El emisor emite pulsos de luz, y se mide el tiempo que tarda un pulso desde que es emitido, hasta que rebota y vuelve al receptor para obtener la distancia.

- LIDAR medidor de fase. El emisor emite un haz láser de forma continua, que se refleja en el obstáculo y se recupera por el receptor. Cuando el reflejo llega la distancia se puede calcular resolviendo cuántas longitudes de ondas se ha desplazado observando la discrepancia de fases.

Originalmente, estos sensores eran muy costosos y pesados, no indicados para UAV multirrotores. Sin embargo, han aparecido recientemente versiones miniaturizadas de bajo peso y medio coste que sí cualifican para equipar drones. La figura 4.3 muestra uno de los sensores de pequeño tamaño:

Figura 4.3: LIDAR miniaturizado LIDAR-lite de PulsedLight.

Son sensores muy fiables y poco susceptibles de ser afectados por interferencias. Esta fiabilidad, unida a su bajo peso, convierte a los sensores LIDAR en una opción muy interesante para diversas aplicaciones, aunque también son más caros que los sensores IR y tienen un haz de detección todavía más estrecho, dejando muchas zonas ciegas.

4.1.3 Sensor LIDAR rotatorio

Los sensores LIDAR rotatorios son la evolución natural de los LIDAR simples. Estos sensores son LIDAR que consiguen realizar mediciones en 360º por medio de su rotación o de la rotación de un espejo a su alrededor que redirecciona el rayo.

Como ocurría con los LIDAR simples, este tipo de sensores eran demasiado pesados para ser llevados encima de un dron multirrotor hasta hace muy poco. En el año 2014, se creó la primera versión comercial reducida de LIDAR rotatorio, mostrada en la figura 4.4:

Figura 4.4: LIDAR rotatorio RPLidar de RoboPeak.

Estos sensores ya no comparten el problema de zonas ciegas del que sufría el LIDAR unidimensional, de hecho la alta resolución que son capaces de alcanzar significa que no tienen zonas muertas en absoluto. Entre los sensores estudiados, es el que consigue la información a sus alrededores más completa.

Aunque tienen una importante desventaja en su elevado coste, y en la mayor dificultad de procesamiento de sus datos.

4.1.4 Sensor de ultrasonidos

Los sensores de ultrasonidos funcionan por medio de la ecolocalización. Dotados de un generador y un receptor de ultrasonidos, miden distancias emitiendo un pulso de ultrasonidos y contando el tiempo que tarda el pulso en volver tras rebotar en la superficie de algún obstáculo con el que se encuentre.

Estos sensores tienen las virtudes de ser muy sencillos en su implementación y uso. Son muy ligeros y baratos. Existen en grandes variedades distintas, con haces de diferentes amplitudes, algunas de ellas convenientemente amplias.

Figura 4.5. Sensor de ultrasonidos HC-SR04

Sus principales desventajas se encuentran en las interferencias que se generan en sus medidas cuando existe ruido de fondo, cuando se reciben ecos secundarios de los pulsos, o cuando se encuentran pulsos de sensores diferentes.

Esta tecnología se desarrollará de forma más detallada en el apartado 4.2

4.1.5 Cámara

La visión que proporcionan las cámaras puede ser analizada mediante técnicas de procesamiento de imagen para detectar la cercanía de los objetos en su campo de visión y la velocidad de acercamiento de estos.

Existen modelos de cámaras de pequeño tamaño y peso, ideales para su uso en UAV multirrotor. Además, las cámaras abarcan un campo visual amplio en el que pueden detectar posibles obstáculos.

Sin embargo, extraer de la visión que proporcionan la información que se busca requiere de algoritmos complejos y alto poder de computación, mientras que los resultados obtenidos no siempre son muy fiables. El alto requerimiento de poder de computación se ve multiplicado además por el número de cámaras que se requerirían instalar para obtener percepción

envolvente.

4.1.6 Cámara estéreo

Las cámaras estéreo se componen de dos cámaras individuales que apuntan hacia la misma zona separadas por una distancia fija.

Gracias a la doble cámara y a algoritmos de combinación de imágenes es posible reconstruir un modelo tridimensional de los objetos captados por las cámaras.

Es un método muy potente que permitiría de forma eficaz reconocer las distancias a todos los obstáculos y objetos en su campo de visión.

Lamentablemente, la mayoría de cámaras estéreo existentes son muy voluminosas y pesadas para ser transportadas en un dron. Y si las cámaras singulares son computacionalmente exigentes, estas lo son todavía más.

4.1.7 Selección de la tecnología a emplear

En este apartado se evaluarán las fortalezas y desventajas de las tecnologías exploradas para seleccionar qué tecnologías se usarán para desarrollar el sistema.

La tabla 4.1 resume de forma breve las principales características positivas y negativas de cada una de ellas:

Tabla 4.1. Comparativa de las distintas tecnologías de detección de distancia.

Tecnología	Características positivas	Características negativas	Evaluación
INFRARROJOS	Sencillez, Bajo coste, ligereza	Baja fiabilidad en exteriores, reducido haz de detección	**No apta**. Sus características negativas la hacen inconveniente.
LIDAR	Sencillez, Bajo coste, ligereza, precisión	Muy reducido haz de detección	**No apta**. Se busca un sensor que sea capaz de detectar obstáculos en un gran abanico.
LIDAR ROTATORIO	Precisión, ángulo de detección de 360º	Coste, complejidad del tratamiento de su información	**No apta**. Descartada debido a su alto coste y complejidad
ULTRASONIDOS	Sencillez, bajo coste, ligereza, buen ángulo de haz	Vulnerables a eco y formas de interferencia por ruido	**Apta**. La existencia de formas de combatir las interferencias hace de este sistema adecuado.
CÁMARA	Bajo coste, ligereza, buen ángulo de visión	Carga computacional elevada, complejidad	**No apta**. El sistema buscado no ha de suponer una necesidad computacional excesiva, y se pretende que sea sencillo
CÁMARA ESTÉREO	Bajo coste, ligereza, buen ángulo de visión, gran precisión	Carga computacional elevada, complejidad	**No apta**. Por las mismas razones que el anterior

Con lo que se decide que la tecnología sobre la que se va a basar el sistema de detección de obstáculos es la tecnología de **ultrasonidos**.

4.2 Tecnología de ultrasonidos

El término ‘sensor de ultrasonidos’ describe un sistema que consta de un emisor y un receptor ultrasónico, comparables a un altavoz y un micrófono, pero diseñados para trabajar a frecuencias ultrasónicas.

El oído humano puede detectar frecuencias desde los 20 Hz hasta los 20.000 Hz. Frecuencias superiores a 20.000Hz son consideradas ultrasónicas. Los ultrasonidos son ondas de presión que se propagan a la velocidad del sonido, lo que significa que son considerablemente más lentas que las ondas electromagnéticas.

El uso típico de los ultrasonidos para medir distancias se basa en el fenómeno de reflexión de ondas. El emisor del sensor emite un pulso de corta dirección en la dirección en la que apunta, y se mide el tiempo que tarda el receptor en recibir la onda reflejada.

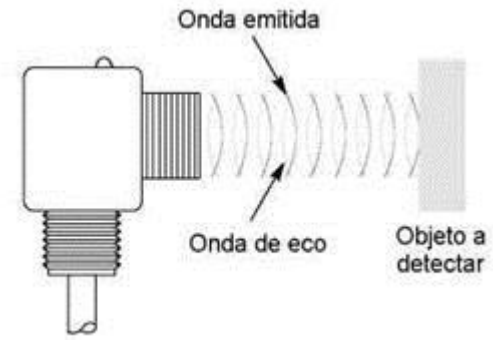

Figura 4.6. Funcionamiento básico de un sensor de ultrasonidos.

Conociendo el tiempo de viaje de la onda, lo único que resta es conocer la velocidad a la que se propaga la onda por el aire, pues:

$$d = v * t \tag{4.2}$$

Donde:

- d es la distancia que recorre la onda sumando la ida y la vuelta, es decir, el doble de la distancia entre el sensor y el obstáculo.
- v es la velocidad de propagación de la onda.
- t es el tiempo de viaje de la onda.

Por tanto, la velocidad a la que se propaga el sonido en el aire es importante. Al tratarse el sonido de una onda de presión, su velocidad sobre un gas puede calcularse como:

$$v = \sqrt{\frac{\gamma R T}{M}} \tag{4.3}$$

Siendo:

- **γ** el coeficiente de dilatación adiabática del gas.
- **R** la constante universal de los gases.
- **T** la temperatura del gas en kelvin.
- **M** su masa molar.

En el caso del aire, el coeficiente de dilatación adiabática, la constante de los gases, y la masa molar, pueden sustituirse por sus valores:

$$\gamma = 1.4$$
$$R = 8{,}314\, J/mol \cdot K$$
$$M = 0{,}029\, kg/mol$$

Que introduciéndolos en (), devuelven la expresión:

$$v = \sqrt{401.4 * T}$$

Linealizando la ecuación en torno a T = 273.15 K, llegamos a la aproximación, usada en la práctica, de:

$$v = 331{,}5 + 0.6\, T' \quad m/s \tag{4.4}$$

En la que *T'* es la temperatura del aire en grados Celsius.

Como todas las ondas, el sonido sufre también fenómenos de refracción y atenuación. La refracción de las ondas sonoras es poco significativa y tan sólo induce errores en las mediciones en casos de muy altas turbulencias en el medio. Por otro lado, la atenuación de las ondas con la distancia sí es importante puesto que es la que limita el rango máximo de detención de esta tecnología.

La expresión que rige la atenuación de las ondas con la distancia, es:

$$I(R) = I(R_0) - 20 \cdot \log_{10}\left(\frac{R}{R_0}\right) \tag{4.5}$$

Siendo $I(R)$ la intensidad del sonido en decibelios y $I(R_0)$ la intensidad de referencia a distancia R_0.

La generación y detección de pulsos se realizan mediante lo que se conoce como transductores piezoeléctricos. Los transductores convierten energía eléctrica en mecánica contenida en la vibración de un pulso ultrasónico y viceversa.

Para generar el pulso, es necesario crear momentos de presiones altas y bajas alternativamente a una frecuencia significativamente alta (>20Khz). Esto se consigue por medio del fenómeno piezoeléctrico, según el cual un material responde con una contracción o dilatación en una dirección determinada a un cambio de voltaje.

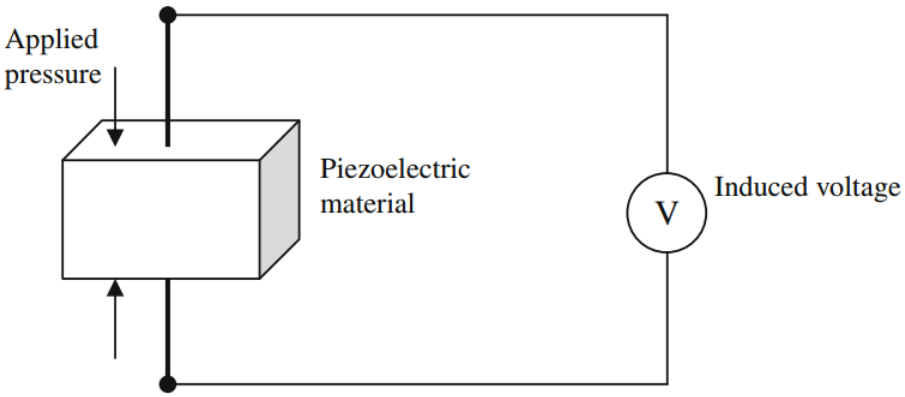

Figura 4.7. Generación de ondas de sonido en un transductor.

Por medio de la aplicación al material piezoeléctrico, generalmente cuarzo, de una tensión alterna lo suficientemente rápida, las contracciones crearán el pulso de ultrasonidos.

La recepción del pulso se realiza del mismo modo, dado que el fenómeno piezoeléctrico también funciona a la inversa. Al recibir el pulso reflejado se manifestará un voltaje entre los extremos del cuarzo medible por el sensor.

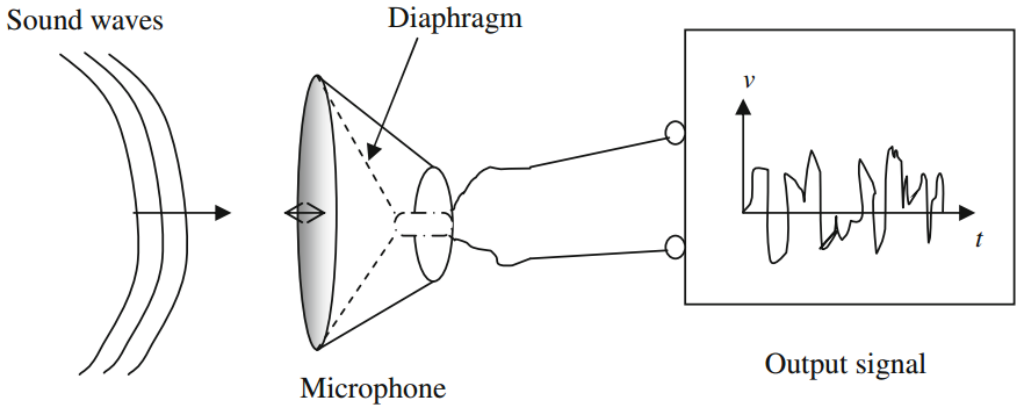

Figura 4.8. Medición de ondas de sonido en un transductor.

El pulso que se genera lo hará con un patrón de radiación definido que determinará el rango eficaz del sensor y la amplitud de su pulso. Un patrón de radiación recoge la potencia sonora radiada o recogida en función del ángulo y distancia desde el transductor.

El patrón de radiación viene determinado por parámetros estructurales y por tanto depende del transductor empleado. La forma de dicho patrón es una de las principales diferencias que puede encontrarse entre distintos sensores de ultrasonidos. Encontrar uno que tenga un patrón de radiación adecuado será vital para conseguir un eficaz sistema de esquivo de obstáculos.

La figura 4.9 muestra un ejemplo de patrón de radiación del sonido muy unidireccional:

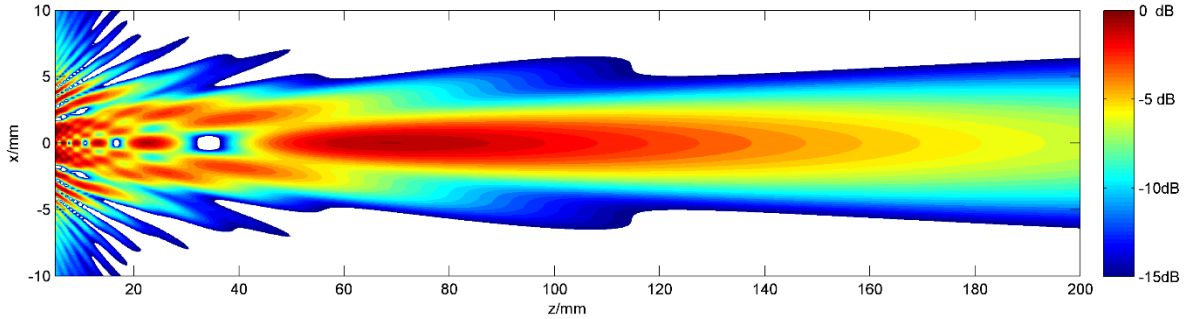

Figura 4.9. Ejemplo de patrón de radiación de un transductor.

Para finalizar este apartado, a continuación se resumirán los posibles problemas que puede acarrear el uso de sensores ultrasónicos en el sistema a desarrollar:

Dependencia de la medida con la temperatura

Como se ha visto, la velocidad de propagación del sonido es dependiente de la temperatura del aire por el que viaja. Por lo tanto, la calibración de sensores para una temperatura específica podrá ser causa de pérdida de precisión si después los sensores se utilizan en otras condiciones distintas.

Interferencias por ruido

Los motores del UAV generan parte de ruido en el rango de frecuencias típico de sensores de ultrasonidos, entorno a los 40.000 Hz, lo que provocará interferencias en las medidas si no se tiene cuidado.

***Crosstalk* entre sensores**

Se conoce como *crosstalk* el fenómeno en el cual pulsos emitidos por un sensor son recogidos por otro distinto que funciona a la misma frecuencia. Estas interferencias conllevan la medición de distancias erróneas.

El *crosstalk* puede ocurrir cuando dos sensores con haces de gran ángulo se colocan cerca entre sí, o cuando sensores de distintos dispositivos se cruzan. Puesto que en Vision4UAV se realizan investigaciones de *swarming* (aplicaciones que coordinan varios UAV), la posibilidad de que ocurran *crosstalk*s no se puede ignorar.

4.2.1 Elección del sensor de ultrasonidos

Existe una amplia variedad de marcas de sensores de ultrasonidos cada una de las cuales trabaja con diferentes modelos. Se procederá ahora a mostrar los resultados de un breve estudio de mercado sobre las diferentes opciones disponibles y la decisión final de un modelo para utilizar.

Se consideraron los siguientes sensores:

Sensor HC-SR04

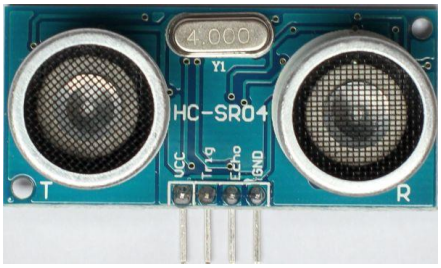

Figura 4.10. Sensor de ultrasonidos HC-SR04.

El sensor *HC-SR04 Ultrasonic Range Finder* está formado por un transmisor y receptor diferenciados junto con la circuitería de control. Trabaja con 5V de corriente continua y requiere una corriente de operación de 15 mA. Su rango de funcionamiento teórico es de 2-400 cm y posee una resolución de 0.3 cm.

Posee un semiángulo de detección de 15º como se aprecia en su patrón de radiación recogido por la figura 4.11:

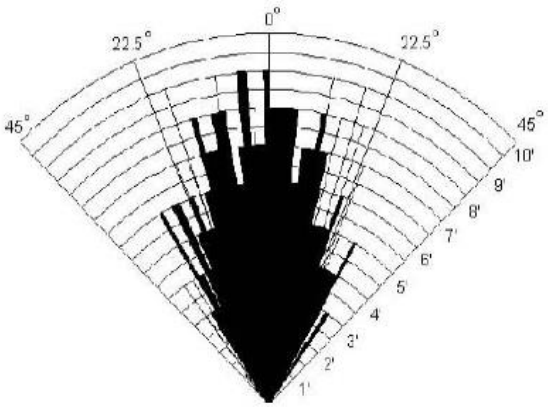

Figura 4.11. Patrón de radiación del sensor HC-SR04.

Cuenta con cuatro pines:

- **VCC**: Alimentación (5V)
- **Trig**: Pin de control para iniciar el pulso.
- **Echo**: Pin de salida para detectar el eco.
- **GND**: Tierra

Su funcionamiento viene recogido en la siguiente figura y se explica a continuación:

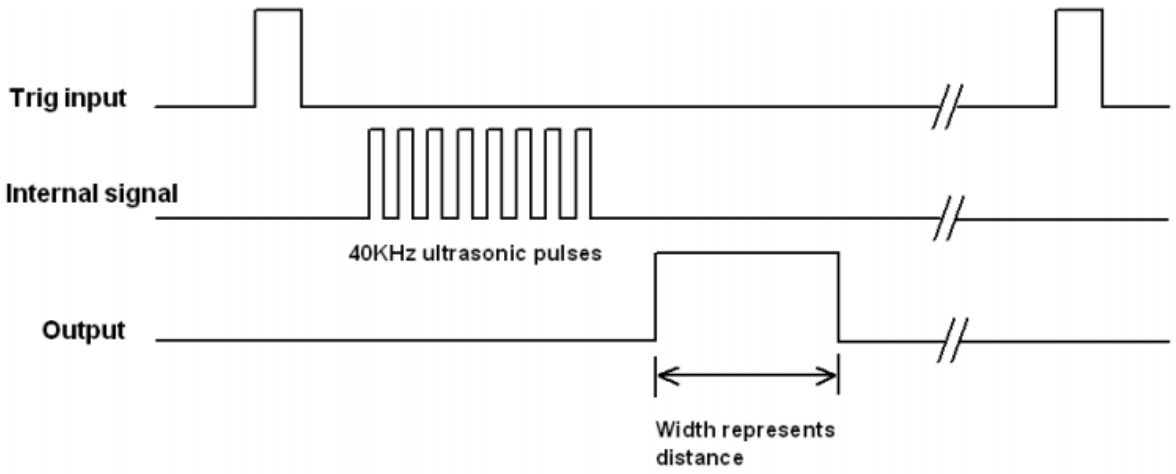

Figura 4.12. Esquema de funcionamiento del sensor HC-SR04.

1. Requiere de un pulso de corta duración (10 µs) en el pin Trig para generar a través del transmisor el pulso de ultrasonidos a 40 KHz.
2. Tras el envío del pulso, el pin Echo se pone a nivel alto.
3. Cuando llega el eco del pulso, el pin Echo se pone a nivel bajo.

El tiempo que el pin Echo ha permanecido a nivel alto es el tiempo de viaje de la onda de ultrasonidos, que deberá medirse externamente y permitirá calcular la distancia al objeto según la expresión (4.1).

Es un sensor muy barato, conseguible fácilmente por 3€ en proveedores de Internet. Se consideró un sensor interesante para el proyecto y se consiguió un modelo para realizar pruebas y comprobar su viabilidad.

Tras probarlo, se obtuvieron las siguientes conclusiones:

- Aunque su rango mínimo es de 2cm, a distancias menores de 20cm las medidas que ofrece son poco fiables.
- Es muy susceptible al ruido, y no fue extraña la aparición de medidas erróneas periódicamente sin motivo aparente.
- En pruebas en las que fue montado sobre un dron con los motores encendidos, el ruido introducido estos invalidaba casi totalmente las medidas del sensor.

Por estos motivos se procedió a descartar el Sensor HC-SR04.

Sensor PING)))

El sensor PING))) de la compañía Parallax es muy parecido al sensor HC-SR04, tanto en diseño como en funcionamiento.

Figura 4.13. Sensor de ultrasonidos PING))) de Parallax.

También se compone de transmisor, receptor, y circuitería, y funciona a 5V. Consume una intensidad media de 0.35mA, algo mayor que el HC-SR04, y su rango es de 2cm a 3m. La principal diferencia constructiva entre ambos se presenta en los pines, ya que en lugar de cuatro utiliza tres:

- **GND**: Tierra
- **5V**: Alimentación
- **SIG**: En este caso, el pin SIG comparte las funciones que tenían los pines Trig y Echo en el sensor HC-SR04

Aunque el principio de funcionamiento es el mismo, aunar las tareas realizadas por dos pines en uno conlleva unas ligeras modificaciones en su algoritmo de uso:

1. Se configura el pin como salida.
2. Se manda un pin de corta duración (5 µs) por el pin SIG.
3. Se configura el pin como entrada.
4. Se espera a que el pin se ponga a nivel alto.
5. Se calcula el tiempo que tarda en ponerse a nivel bajo.

Su patrón de radiación indica tiene un semiángulo de detección de unos 20º, como puede observarse en la figura 4.14:

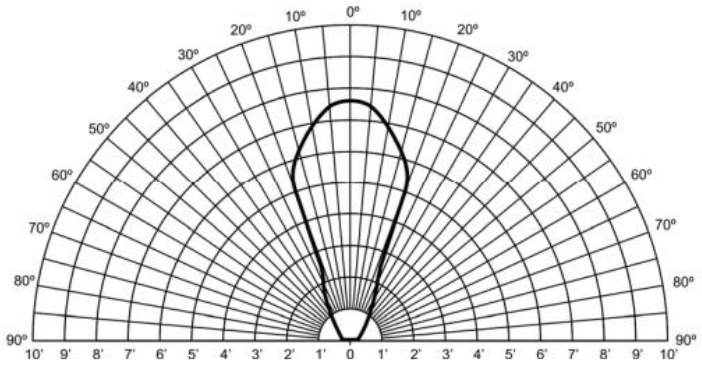

Figura 4.14. Patrón de radiación del sensor PING))) de Parallax.

Puede encontrarse por unos 20€, con lo que tiene un precio bastante superior al sensor anterior, pero que todavía es muy asequible.

Al igual que se hizo con el sensor HC-SR04, se compró una unidad para realizar pruebas de sus prestaciones. Los resultados obtenidos fueron los siguientes:

- De nuevo, las medidas de obstáculos más cercanos a 20cm obtenidas con este sensor son poco fiables.
- Sin embargo, la calidad general de sus medidas es muy superior a la de su homólogo de 4 pines, detectándose en muy pocos casos medidas atípicas erróneas, incluso ante la presencia de ruido ambiente.
- Finalmente, su uso montado en el UAV con los motores encendidos interfería de forma muy grave con las medidas.

Dada la necesidad de fiabilidad en la toma de medidas para desarrollar un correcto esquivador de obstáculos, se descartó el uso del sensor PING))).

Sensores MaxSonar

La compañía MaxBotix se especializa en el diseño y construcción de sensores de ultrasonidos de alto rendimiento. En su catálogo de sensores de ultrasonidos, que reciben el nombre de MaxSonar, cuentan con modelos para distintas aplicaciones. Los más adecuados para el sistema que se pretende desarrollar son los pertenecientes a las categorías MaxSonar LV y MaxSonar XL.

Figura 4.15. Sensor MaxSonar XL EZ0 de MaxBotix.

Estos sensores cuentan con un único transductor que hace los papales de transmisor y receptor. Pueden operar con alimentación de 3.3V a 5.5V y consumen 3.4mA para generar pulsos a 42 kHz. Se presentan en diferentes modelos que varían según su rango de detección y la amplitud de su ángulo de detección.

Cuentan con siete pines que permiten diferentes formas de comunicación de las medidas obtenidas. Los que interesan para este proyecto son:

- **AN**: Entrega un valor analógico proporcional a la distancia obtenida en la última medición.
- **RX**: Pin de control que permite apagar el sensor cuando recibe un nivel bajo de tensión.
- **+5V**: Alimentación (3.3V – 5V)
- **GND**: Tierra

Su modo de operación varía respecto de los sensores estudiados anteriormente. Los sensores MaxSonar funcionan de modo continuo sin necesidad de activar el envío de los ecos, y entregan de forma directa las medidas que toman a través del pin AN, lo que ahorra la necesidad de controlar tiempos de pulso a muy altas velocidades, con lo que su uso es considerablemente más sencillo.

Los sensores MaxSonar cuentan con una tecnología de calibración en tiempo real en función de las condiciones del aire que se traducen en un aumento de la precisión en usos en diferentes situaciones.

Además, sus medidas a corto rango son fiables, puesto que las distancias menores de 25cm son representadas como 25 cm y no como números que pueden ser aleatorios. También son los más ligeros y compactos, y sus unidades son probadas y calibradas en sus fábricas.

Todo ello hace que su precio se eleve con respecto a los demás sensores estudiados, y ascienda a en torno a 30€ para los LV y a 40€ para los XL.

Los sensores MaxSonar LV se diferencian de los MaxSonar XL en tres características principales:

- Los LV son capaces de realizar mediciones a 20 Hz. Los XL tan solo a 10 Hz.
- Los XL cuentan con un sistema de filtrado de ruido de frecuencias cercanas a 40kHz.
- Los XL tienen una mayor resolución.

Los sensores XL, son, en esencia, una versión de mayor gama que los LV que incluyen técnicas de cancelación de ruido.

A la vista de la importancia que tienen las interferencias generadas por los motores observadas en las pruebas con otros sensores, se tomó la decisión de escoger sensores MaxSonar XL, aún a costa de la menor frecuencia de medición y el gasto añadido, para probar su viabilidad en el sistema.

La gama de sensores MaxSonar XL presenta cinco modelos: EZ0, EZ1, EZ2, EZ3 y EZ4. Estos modelos se diferencian entre sí en sus patrones de radiación y sus rangos efectivos, teniendo cada uno menor ángulo de detección y mayor alcance máximo que el anterior.

Se escoge el sensor EZ0 para realizar las pruebas por ser el que mayor ángulo de detección tiene, que presenta el siguiente patrón:

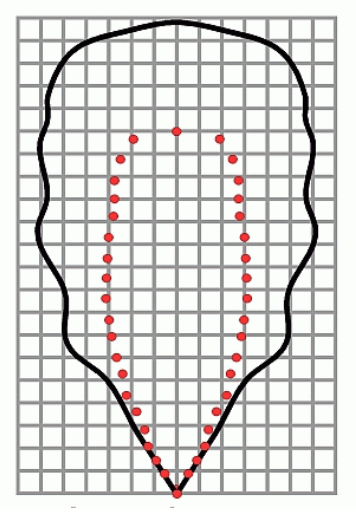

Figura 4.16. Patrón de radiación del sensor MaxSonar XL EZ0 de MaxBotix.

Tras probar el sensor, las conclusiones obtenidas fueron:

- Presentan una gran consistencia en sus medidas comparables a las obtenidas con el sensor PING))).
- El ángulo de detección es mayor que el de los sensores probados, pero no tanto como parecía indicar el patrón de radiación recogido en su hoja de características. Se estima que su semiángulo de detección es de 30º
- El sistema de filtrado de ruido funciona a la perfección. No se apreció ninguna diferencia significativa entre las mediciones tomadas con los motores apagado y con los motores encendidos.

Dada su alta consistencia contrastada en las pruebas, se decidió el uso de los sensores MaxSonar XL EZ0 de MaxBotix para desarrollar el sistema.

4.2.2 Distribución de los sensores

El sistema desarrollado será compatible con diferentes números de sensores y distribuciones de estos, e, incluso, con diferentes modelos de sensores siempre que sean MaxSonar, como parte de un esfuerzo por dotar de versatilidad a la solución desarrollada.

Para el prototipo que se construirá y que está basado en el octocóptero Oktokopter de la compañía Mikrokopter, se plantea el uso de cuatro sensores de ultrasonidos, atendiendo a la siguiente configuración:

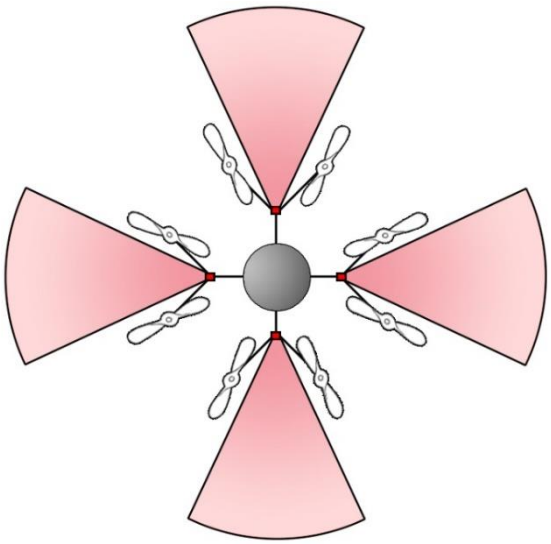

Figura 4.17. Distribución de los sensores de ultrasonidos en el Oktokopter.

Los sensores están colocados en las cuatro direcciones principales del dron, apuntando a los ángulos α = 0º, 90º, 180º y 270º. Es importante conocer la distancia entre cada sensor y el final de la pala de las hélices cercanas, pues será la distancia a partir de la cual el dron colisionará. Dicha distancia resulta ser de $35cm$.

4.3 Interfaz de sensores

En este capítulo se abordan las cuestiones relacionadas con la interfaz que conecta los sensores de ultrasonidos con el ordenador de a bordo del Oktokopter.

La interfaz debe encargarse de cuatro tareas:

Figura 4.18. Tareas de la interfaz de sensores

4.3.1 Elección del hardware

Esta interfaz ha de cumplir los siguientes requisitos:

1. Debe poseer un número de salidas digitales y entradas analógicas suficiente para controlar el número de sensores MaxSonar XL que se vayan a utilizar. Aunque el prototipo cuente con cuatro, será favorable que la interfaz pueda acomodar más sensores para ser compatible con distintos diseños.

2. Debe poder computar y enviar datos por serie a una velocidad suficiente para mantener una tasa de refresco de 10Hz.

3. Debe ser compatible con ROS y poder enviar mensajes y publicar en *Topics* dentro de su framework.

4. En la medida de lo posible, debe ser ligera y consumir poca potencia puesto que irá a bordo del UAV y se alimentará por su batería.

La placa Arduino Nano V3.0 es una buena opción. Cuenta con 8 pines analógicos y 14 digitales y un microprocesador Atmega 328P que corre a 16MHz, alimentado por 5V, el cual la dota de la velocidad necesaria para manejar la lectura de los sensores y el envío de datos de forma simultánea. Tiene 32 KB de memoria flash, 2 KB de SRAM, conexión Mini-B USB y pesa tan sólo 5g.

Figura 4.19. Arduino Nano V3.0

La mayor limitación de la elección de la placa Arduino Nano es que su memoria puede resultar limitada para grandes programas que contengan estructuras pesadas de datos como algunos programas de ROS. Por este motivo, su programación se desarrolló tratando de limitar todo lo posible los requerimientos de memoria necesarios.

4.3.2 Lectura de los datos de los sensores

El lenguaje de programación que usa Arduino dispone de sentencias que permiten recoger el valor de sus puertos. Al tratarse de un valor y pin analógicos, la captura de datos se realiza mediante la sentencia *analogRead()* especificando entre los paréntesis el nombre del pin del cual se quiere conocer su valor. Los pines analógicos van desde A0 hasta A7 en la placa Arduino Nano.

La lectura se ejecuta a una frecuencia constante de 10 Hz, que se corresponde con la frecuencia a la que se actualizan las mediciones de los sensores de ultrasonidos MaxSonar XL.

En ocasiones, se introduce ruido en la lectura de los valores de los pines, por lo que es conveniente la inclusión de un filtro de lectura.

4.3.3 Filtrado de la lectura de los puertos

El filtro desarrollado pretende eliminar los valores atípicos que pudieran aparecer y conseguir una medida más precisa del verdadero valor analógico que llega al puerto. Su funcionamiento es el siguiente:

1. En lugar de realizar una única lectura del puerto, se toman cuatro medidas sucesivas y se almacenan.
2. Una por una, se computa la suma de las distancias que presenta cada lectura con las otras tres.
3. Se elimina la lectura que presenta una mayor diferencia con respecto a las demás.
4. Se promedian las tres lecturas restantes, y se toma el resultado como valor final filtrado de la lectura del puerto.

4.3.4 Cálculo de la distancia

La distancia se calcula a partir de la medida obtenida de la forma explicada en el apartado anterior. Dicha medida será un valor de voltaje de entre 0 y 5V, que se traduce en un número de entre 0 y 1023 en el convertidor analógico-digital de la placa Arduino.

Para calcular la distancia en cm a partir de dicho valor hay que hacer una operación sencilla:

$$D = Lectura \cdot \frac{1}{1024} \cdot 700 = Lectura * 0.684 \quad (4.6)$$

4.3.5 Comunicación con el ordenador de a bordo

Para conseguir la comunicación entre el ordenador de a bordo, que funciona con ROS, y el la placa Arduino que se conecta por USB, se utiliza el protocolo de ROS conocido como Rosserial.

Rosserial cuenta con dos tipos de librerías que facilitan en gran medida la comunicación de un sistema con sus periféricos: servidores de rosserial y clientes de rosserial.

Las librerías de servidor permiten la inicialización de un Nodo en el sistema central que se convierte en el nexo entre el periférico y el resto del sistema. Dicho Nodo realiza todas las comunicaciones con el entorno de ROS requeridas por el periférico, publicando y suscribiéndose a *Topics* en su nombre.

Las librerías de cliente, que se instalan en los periféricos, ajustan los datos que estos quieren enviar o recibir a la estructura presente en ROS de mensajes, *Topics*, publicadores y suscriptores, y se comunica con el Nodo servidor que ha de estar abierto en el sistema principal.

Existe una librería de cliente de rosserial para Arduino, llamada rosserial_arduino, que se utilizará para gestionar las comunicaciones.

Se define un tipo de mensaje específico para comunicar los datos recibidos por los sensores de ultrasonidos al ordenador principal, con el nombre de *UltrasonicRangeArduino.* Contiene la siguiente estructura en *IDL* (ver 1.6.4):

```
Header header
Char id
Uint8 range
Uint8 freq
Bool ok
```

- **header**: El *Header* es un tipo estándar en ROS, útil porque posee un campo que guarda el momento en que se genera el mensaje, y otro que guarda cuántos mensajes se han enviado antes que este para conocer su orden
- **id**: Contiene la ID del sensor al que pertenece la distancia que se está enviando en el mensaje, es un número del 0 al número de sensores
- **range**: Contiene la distancia medida en cm. Admite valores entre 0 y 255 cm
- **freq**: Contiene la frecuencia a la que ese sensor está enviando los datos.
- **ok**: Contiene un booleano que puede utilizarse para decir si la medida es correcta o no.

Los mensajes se publicarán en el *Topic* de nombre "/drone2/ultrasonic", donde "drone2" hace referencia al nombre interno que recibe el Oktokopter sobre el que se ha desarrollado el prototipo.

El envío de los mensajes se realiza a frecuencias individuales, atendiendo al modo de funcionamiento que se escoja para cada sensor.

4.3.6 Modos de funcionamiento

Se han desarrollado tres modos de funcionamiento en la interfaz, que cada sensor puede adoptar. Dichos modos pueden cambiarse en cualquier momento desde el ordenador de a bordo. Los modos son los siguientes:

Modo de funcionamiento apagado. Cuando un sensor entra en este modo, deja de emitir pulsos de ultrasonidos y la interfaz deja de leer el pin asociado a dicho sensor y de enviar sus mensajes.

Modo de funcionamiento constante: Este es el modo por defecto, en el que el envío de sus mensajes se realiza de forma periódica atendiendo a la frecuencia que tiene asociada.

Modo de funcionamiento a corta distancia: En este modo, que ha sido diseñado para aligerar ancho de banda en el canal de comunicación y la necesidad de procesamiento desde el otro lado, las medidas sólo se enviarán en caso de que sus valores sea inferiores a un umbral superior establecido en 1,5m. Con ello, las medidas que no representan peligro de colisión son ignoradas y no consumen recursos.

4.3.7 Parámetros configurables

Además de enviar mensajes al ordenador de a bordo, la interfaz de Arduino cuenta con un servidor de servicios de ROS con el que se puede comunicar el ordenador, que le permite configurar parámetros y solicitar una actualización de medida urgente de un sensor en particular. Los parámetros configurables son:

- El modo de funcionamiento de cada sensor.
- La frecuencia de envío de datos de cada sensor.

Para ello, se define el Servicio *SetAndGet*, que en estructura *IDL* tiene el siguiente aspecto:

Request

```
Int8 req_frequency
Int8 req_mode
Int8 req_update
Char req_id
```

Reply

```
UltrasonicRangeArduino range
```

Los enteros de 8 bits req_frequency y req_mode portan los valores de la frecuencia y el modo

que se quieren imponer, y el carácter contenido en req_id la id del sensor al que se quiere aplicar la imposición. Así, cada vez que se invoque el servicio se podrá ajustar el modo y frecuencia de un sensor individual.

En caso de querer cambiar todos los sensores de una vez, esto puede hacerse dando un valor de -1 al campo que contiene la id. Si el servicio encuentra un -1 en este campo los valores de los campos de frecuencia y modo se volcarán en todos los sensores del sistema.

En todo caso, los valores mandados sólo se actualizarán en los sensores si se encuentran dentro sus rangos válidos, que son de cero a dos para el modo, y de uno a diez para la frecuencia.

Por otro lado el servicio también permite la petición del valor más reciente de un sensor individual, lo que se realiza mandando un valor de req_update distinto de cero. Cuando esto ocurra, se devolverá la última medida actualizada correspondiente al sensor cuya id sea la que se mande en el campo req_id siempre que el valor de dicho campo esté comprendido en el rango cero – número de sensores instalados.

El código completo que controla la interfaz viene recogido en el Anexo I.

5 ESQUIVO DE OBSTÁCULOS

La parte de control de esquivo incluye el filtro de Kalman de las medidas de ultrasonidos, el detector de colisiones y la unidad de control. En este capítulo se explicarán sus principales características.

5.1.1 Recepción y filtrado de las medidas en el UAV

Se abordará ahora la parte siguiente del desarrollo del sistema según el esquema de la figura 3.1. Dicha parte consiste en la recogida de la información que envía la interfaz, y su filtrado mediante un filtro de Kalman.

Para recibir las medidas enviadas por la interfaz, se deberán utilizar las librerías de servidor que incorpora rosserial. Para abrir el servidor se debe especificar el puerto en el que se conecta la placa de Arduino, y una vez abierto manejará de forma automática las conexiones con el periférico.

Como se ha comentado anteriormente, los mensajes enviados por rosserial se publican en el *Topic* "/drone2/ultrasonic". El receptor se suscribe a dicho *Topic* y cada vez que recibe un nuevo mensaje realiza las siguientes operaciones:

1. Detecta a qué sensor pertenece la medida fijándose en su campo 'id'.

2. Convierte la medida al Sistema Internacional, en el que trabaja el *Stack*, dividiendo por 100 el valor obtenido para pasarlo a metros.

3. Ordena la medida en el lugar correspondiente de un vector que contiene las distancias recibidas por cada sensor.

4. Invoca la ejecución del filtro de Kalman sobre la medida.

En ocasiones, en particular en situaciones en las que se presentan superficies planas delante de los sensores de ultrasonidos con elevados ángulos de ataque, o cuando se encuentran objetos justo en el límite del ángulo de detección, las medidas recogidas por los sensores son de baja calidad incluyendo picos y valle. Se incluye una figura que muestra esta situación en el caso más extremo:

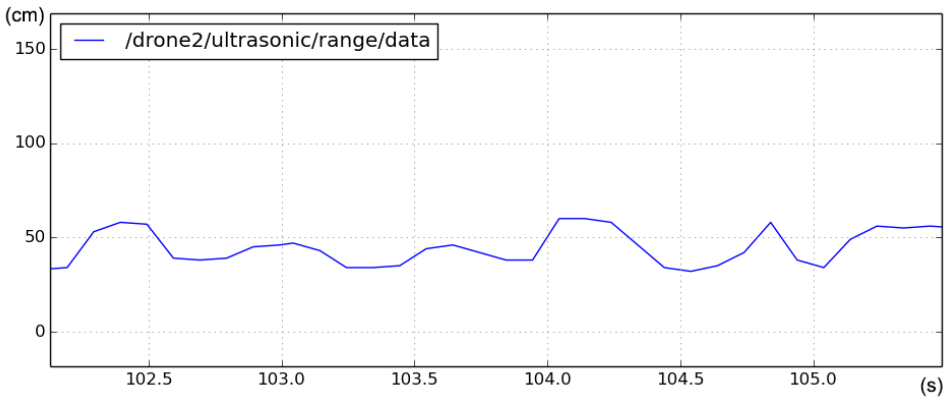

Figura 5.1. Medidas de distancia sin filtrar en condiciones adversas.

Se hace necesaria la inclusión de un filtro que lidie con estos casos para facilitar las tareas de los dos componentes siguientes, el detector de colisiones y el control reactivo.

Se eligió utilizar filtros de Kalman, dada la sencillez de obtener un modelo lineal para las distancias de los sensores teniendo en cuenta que el dron dispone de la medida de la velocidad a la que se mueve, medida por un sensor de flujo óptico.

Dicho sensor devuelve medidas de la velocidad de avance del dron en dos componentes, x e y, referidas al sistema de referencia del dron. Antes de aplicar el filtro, será conveniente la obtención de la proyección de dichas velocidades sobre la dirección a la que apunta cada sensor.

Bastará con realizar un cambio de sistema de referencia y quedarnos con una de sus componentes. La expresión (5.1) recoge la forma de calcular dicha proyección:

$$Vs = Vx * cos(\alpha) + Vy * sin(\alpha) \quad (5.1)$$

Siendo:

- Vs la velocidad en la dirección a la que apunta el sensor, buscada.
- Vx y Vy las velocidades medidas por el sensor óptico en el sistema de referencia del dron.
- α el ángulo del sensor de ultrasonidos según el sistema de referencia del UAV.

Habrá un filtro distinto por cada sensor para que puedan calcularse sus estimaciones según vayan llegando, de forma asíncrona.

Conociendo Vs, y siendo T el tiempo que transcurre entre dos medidas consecutivas realizadas por el mismo sensor, la distancia D entre el sensor y el obstáculo, en el instante k, puede expresarse como:

$$D_k = D_{k-1} - Vs \cdot T \quad (5.2)$$

Se procederá a ajustar ahora dicho modelo a la estructura requerida por el filtro de Kalman (ver sección 1.7). En el ajuste:

- El vector de estado, x, será unidimensional y se compondrá por la distancia D.
- El vector de control, u, también será unidimensional y su valor será $Vs \cdot T$.

Añadiéndo los ruidos gaussianos, el modelo queda así:

$$D_k = D_{k-1} - Vs \cdot T + w_k$$

$$z_k = D_k + v_k \quad (5.3)$$

Al tratarse de un problema unidimensional, las matrices $\boldsymbol{F}$, $\boldsymbol{B}$ y $\boldsymbol{H}$ pasan a ser escalares, y su valor en este modelo será de 1.

Con esto, las ecuaciones de la implementación del filtro de Kalman, resultan:

(5.4)

Predicción

$$\widehat{D}_{k|k-1} = \widehat{D}_{k-1|k-1} + Vs \cdot T$$

$$P_{k|k-1} = P_{k-1|k-1} + Q$$

Corrección

$$\tilde{y}_k = z_k - \widehat{D}_{k|k-1}$$

$$S_k = P_{k|k-1} + R$$

$$K_k = P_{k|k-1} \cdot {S_k}^{-1}$$

$$\widehat{D}_{k|k} = \widehat{D}_{k1|k-1} + K_k \cdot \tilde{y}_k$$

$$P_{k|k} = (1 - K_k)\, P_{k|k-1}$$

Para completar las ecuaciones del modelo anterior resta asignar valores a Q y a R. Además, será necesario proporcionar una estimación inicial de las medidas de los sensores y su covarianza para el primer paso del filtro.

Siguiendo la práctica habitual, se seleccionará una covarianza inicial muy alta (500) que representará una alta incertidumbre en el estado inicial. En cuanto a la distancia inicial, se toma el valor máximo que devuelven los sensores, de 2m.

En cuanto a los valores de Q y R, Q es la covarianza que presenta el ruido de proceso y R la desviación del de medida. Un aumento de Q sobre R produce que el filtro se ajuste más al modelo descrito, con lo que se reduce el ruido que aparece en las mediciones. Por

contrapartida, esto lo hace más lento a reaccionar ante obstáculos repentinos que no se predicen por el modelo.

Para la elección de sus valores se realizaron pruebas de la salida del filtro con diferentes valores. Se escogió una solución de compromiso entre fiabilidad y rapidez, de los valores de 1 para Q y 5 para R.

Los resultados del filtro se observan en la siguiente figura:

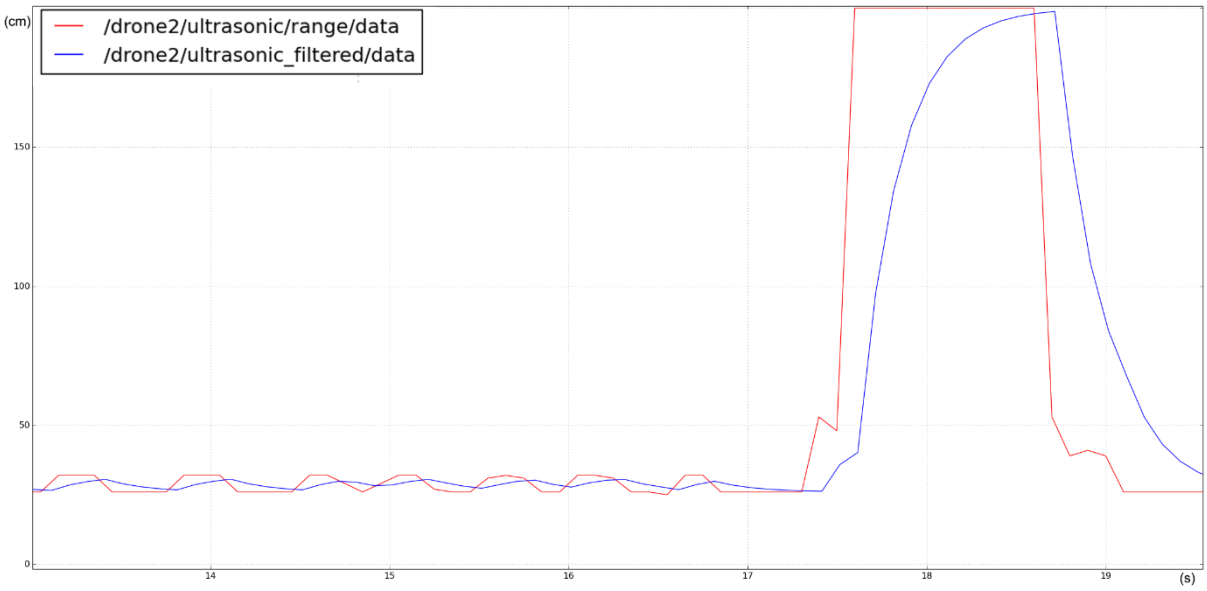

Figura 5.2: Comparación entre medidas sin filtrar y medidas filtradas.

Las medidas filtradas (azul) son más resistentes a la baja calidad de medidas que aparece en condiciones adversas, como puede verse en la parte izquierda de la figura.

Por otro lado, aunque sufren ciertos retardos para cambios bruscos, son comedidos, encontrándose estos entre las dos o tres décimas de segundo.

5.2 Detector de colisiones

El propósito del detector de colisiones será detectar todos aquellos casos en los que el UAV multirrotor se encuentre en peligro de choque con un obstáculo. Para ello, analizará la evolución de las medidas recogidas por los sensores de ultrasonidos y filtradas por el filtro de Kalman.

Se han desarrollado tres configuraciones diferentes de agresividad para el detector de colisiones, entre las que se podrá cambiar en función de la seguridad necesaria en cada caso, que reciben los nombres de configuración normal, configuración agresiva y configuración segura.

En líneas generales, el detector de colisiones realiza dos tareas:

La primera es el cálculo, para cada sensor, de la distancia mínima a partir de la cual es necesario intervenir para evitar la colisión. A esta distancia, que dependerá de la velocidad del dron y las medidas anteriores, recibe el nombre de distancia de seguridad.

La segunda es la comprobación periódica de si alguna de las medidas filtradas de distancias es menor que la distancia de seguridad calculada. En caso de serlo, se activará una bandera de peligro y se invocará la ejecución del control reactivo que tomará el control hasta que el obstáculo se encuentre a una distancia segura.

5.2.1 Cálculo de la distancia de seguridad

En primer lugar, se presenta una simplificación del modelo dinámico para drones multirrotores explicado en el apartado 1.2.3.

En primer lugar, se escoge un sistema de referencia que gira sobre su eje z de forma solidaria con el *yaw* del UAV, con lo que el ángulo ψ pasa a valer 0 en todo momento, con lo que se tiene:

$$\ddot{X} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} -\sin\theta \cdot \cos\varphi \cdot \dfrac{E}{m} \\ -\sin\varphi \cdot \dfrac{E}{m} \\ g - \cos\theta \cdot \dfrac{E}{m} \end{bmatrix}$$

En segundo lugar, se elimina la dependencia de la primera componente con el *roll*, simplificación sólo válida para pequeños ángulos de este. En tercer lugar, nos interesan sólo las componentes x e y de posición del dron. Queda:

$$\ddot{X} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} -\sin\theta \\ -\sin\varphi \end{bmatrix} \cdot \frac{E}{m}$$

Se linealiza el modelo para ángulos bajos de *pitch* y *roll:*

$$\ddot{X} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} -\theta \\ -\varphi \end{bmatrix} \cdot \frac{E}{m}$$

Se resuelve el sistema de ecuaciones diferenciales de segundo orden, y se obtiene el modelo

de la cinemática del dron, donde t es el tiempo:

$$X = \begin{bmatrix} x \\ y \end{bmatrix} = -\begin{bmatrix} \frac{E}{2 \cdot m} \cdot \theta \cdot t^2 + K_{x1} \cdot t + K_{x2} \\ \frac{E}{2 \cdot m} \cdot \varphi \cdot t^2 + K_{x2} \cdot t + K_{y2} \end{bmatrix}$$

Si se toman las condiciones iniciales de velocidad y posición nulas y se aplican a la expresión anterior, se tiene, finalmente:

$$x = -\frac{E}{2 \cdot m} \cdot \theta \cdot t^2$$

$$y = -\frac{E}{2 \cdot m} \cdot \varphi \cdot t^2$$

(5.5)

Del modelo anterior se concluye que la distancia necesaria para que un dron detenga su avance es proporcional con el cuadrado de la velocidad a la que este se está moviendo en el momento de iniciar la parada.

Atendiendo a esta conclusión, se plantea la siguiente expresión para el cálculo de la distancia de seguridad, DS:

$$DS = (Dm + Kv * max(0, Vs)^2 + SAV) * GA \quad (5.6)$$

La expresión (5.6) depende de la velocidad de avance en la dirección del sensor, Vs, y se ajusta mediante cuatro parámetros, los cuales son:

- Dm: Distancia mínima base.
- Kv: Constante de proporcionalidad de la velocidad cuadrática
- SAV: Este parámetro, que se ha denominado Seguro de Alta Velocidad, pretende aumentar la seguridad del sistema en casos en los que el UAV se mueva a velocidades considerables (mayores que $0.3\ m/s$) y permitir que en el resto de casos el control esté más afinado. SAV se activa en los casos de alta velocidad tomando el valor de 0.15 m, y se desactiva para movimientos lentos.
- GA: Ganancia de agresividad, que varía según la configuración que se haya seleccionado. Toma los valores de 1 en control normal, 0.9 en control agresivo y 1.15 en control seguro.

Además, un observador atento notará que la expresión $max(0, Vs)^2$ consigue evitar el aumento de la distancia de seguridad cuando la velocidad de avance del dron es en dirección opuesta a la que apunta el sensor.

El parámetro SAV permite suplir la falta de exactitud del modelo cuando se trabaja a velocidades altas.

La validación del diseño, así como un primer ajuste de los parámetros se llevará a cabo en un Simulador programado para tal fin.

El código se encuentra en el Anexo III.

5.3 Unidad de control

La unidad de control es el componente que cierra el diseño del sistema de esquivo de obstáculos. Cuenta con un control PD aditivo en cascada, el cual toma el control en caso de que el detector de colisiones alerte de una situación de peligro.

El control utilizará la información que proporciona el filtro, junto con la medida de la velocidad de movimiento, para calcular los valores de pitch y roll que detengan el acercamiento del dron al obstáculo y se aleje a una distancia segura.

5.3.1 Control PD en cascada

Se escogió desarrollar un control proporcional – derivativo sobre la distancia entre los sensores y los obstáculos detectados.

La acción proporcional es necesaria para alejar correctamente al dron del obstáculo en cuestión, y la derivativa permite al sistema responder de forma eficaz para distintas velocidades de aproximación.

Se aprovechó que el *Stack* proporciona medidas de la velocidad de movimiento del dron para diseñar un control en cascada, que consigue la acción proporcional-derivativa de la siguiente forma:

- Sobre la diferencia entre la distancia de referencia y la distancia actual, acción proporcional.
- Sobre la velocidad de avance en dirección al sensor, acción proporcional.

La distancia de referencia (DR) será la distancia a la que se inició el control, y hasta que el dron no la supere de nuevo, el control permanecerá activo.

Quedando la acción de control (*pitch* y *roll* de control) descrita por la siguiente ecuación:

$$Acción\ de\ control\ = -\ Kd\ *\ (D\ -\ DR)\ +\ Kv\ *\ Vs \tag{5.7}$$

Siendo Kd y Kv los parámetros del control a ajustar.

Dicha acción de control se proyectará sobre el *pitch* y el *roll* atendiendo al ángulo del sensor en cuestión, quedando:

$$Pitch\ de\ control\ =\ Acción\ de\ control\ *\ cos\ (\alpha)$$

$$Roll\ de\ control\ =\ -\ Acción\ de\ control\ *\ sin\ (\alpha)$$

Cada sensor que detecte peligro de colisionar generará una acción de control individual, que se sumará con la de los demás para obtener la acción de control final. De este modo, si se va a colisionar con obstáculos en más de una dirección, podrán esquivarse todos ellos de forma simultánea en la misma acción de control.

Al igual que el detector de colisiones, la validación de la unidad de control, así como el ajuste de los parámetros Kd y Kv, se realizarán en el Simulador que se ha desarrollado.

5.3.2 Seguridad

Un mal cálculo del control puede resultar en el choque del UAV y la rotura de parte o la totalidad del mismo. Además de conseguir un control estable, es importante incorporar medidas de seguridad adicionales para estar prevenidos ante eventualidades inesperadas que puedan ocurrir.

Se han incluido dos medidas de seguridad adicionales en el control:

En primer lugar, en inicio, hasta contrastar adecuadamente la precisión del sistema, la acción de control se ha limitado a unos valores máximos y mínimos de $\pm 25º$. Así, en el caso de que ocurra un fallo en el control del aparato, el impacto del mismo será contenido.

En segundo lugar, se ha habilitado un comando que desactiva el control reactivo de forma instantánea cuando es invocado, que se le comunica al dron a través del *Topic* "/drone2/stop_ultrasonic".

5.4 Resumen de componentes

En los apartados anteriores se ha descrito el planteamiento y funcionamiento de las cinco componentes del sistema esquivador de obstáculos:

- Sensores
- Interfaz lectora de sensores
- Filtro de las lecturas
- Detector de colisiones
- Unidad de control

A modo de resumen, se incluye la siguiente figura que recapitula las principales tareas de cada componente:

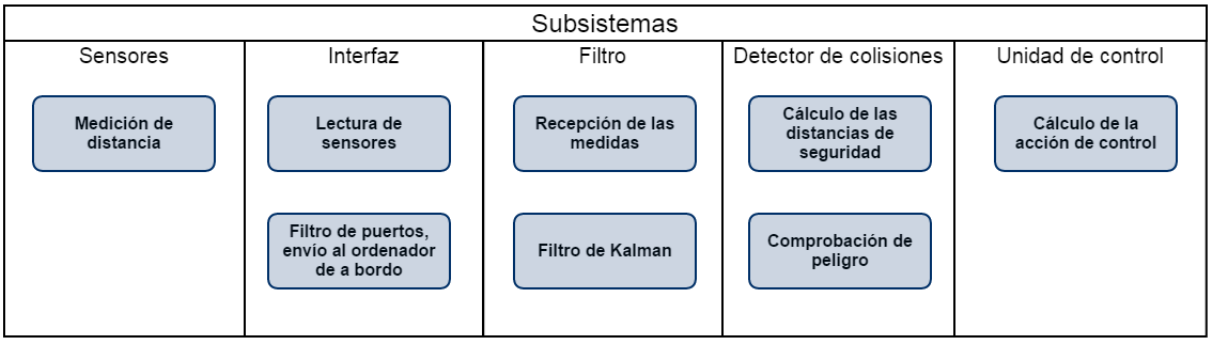

Figura 5.3: Resumen de funcionamiento del sistema esquivador de obstáculos.

6 INTEGRACIÓN

En este capítulo se explica la forma en que el sistema descrito se adapta al *Stack* del grupo CVG y a la arquitectura del *framework* de ROS.

Las componentes del sistema que funcionan en el ordenador de a bordo del UAV, a saber, el filtro de Kalman, la unidad de control y el detector de colisiones, se han incluido en un paquete al que se ha llamado “droneObstacleAvoider”.

Este paquete conforma un módulo del *Stack*, en el que todas sus actividades son manejadas por un Nodo que comparte nombre con el paquete en el que se encuentra: “droneObstacleAvoider”. Dicho Nodo se encarga de gestionar tanto la comunicación con la interfaz así como con el resto de módulos del *Stack*. Su funcionamiento es el siguiente:

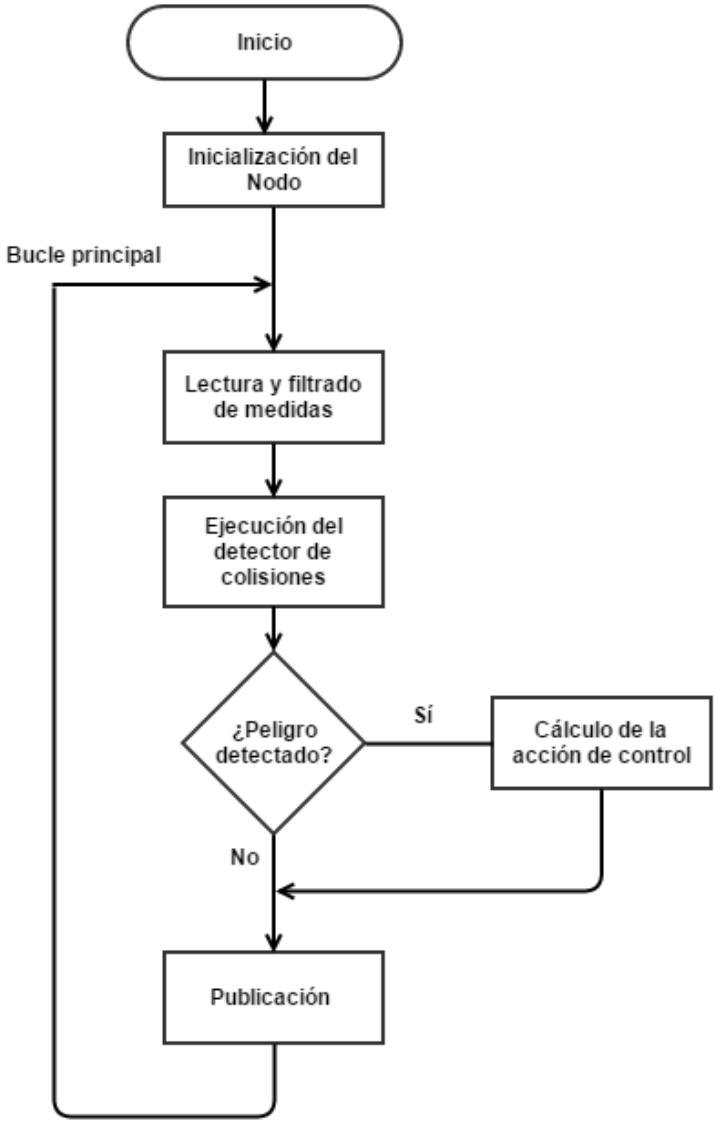

Figura 6.1: Flujograma de funcionamiento del Nodo droneObstacleAvoider.

El nodo se comunica con el resto de elementos del *Stack* a través de tres publicadores y tres suscriptores, que se recogen en las siguientes tablas:

Tabla 6.1: Publicadores del Nodo principal

PUBLICADOR	TOPIC
ACCIONES DE CONTROL	/drone2/command/ultrasonic
BANDERA DE PELIGRO	/drone2/ultrasonic_danger
MEDIDAS DE DISTANCIA FILTRADAS	/drone2/ultrasonic_filtered

Tabla 6.2: Suscriptores del Nodo principal

SUSCRIPTOR	TOPIC
MEDIDAS SIN FILTRAR	/drone2/ultrasonic
DESACTIVADOR DE EMERGENCIA	/drone2/stop_ultrasonic
MEDIDAS DE VELOCIDAD DE AVANCE	/drone2/ground_speed

Las acciones de control y la bandera de peligro son recogidas por el módulo "MidLevelController", que las procesa y las envía al resto del *Stack* para que se lleven a cabo, mientras que las medidas sin filtrar proceden del Nodo cliente de rosserial que se comunica con la interfaz de Arduino. Con todo, las comunicaciones resultan ser:

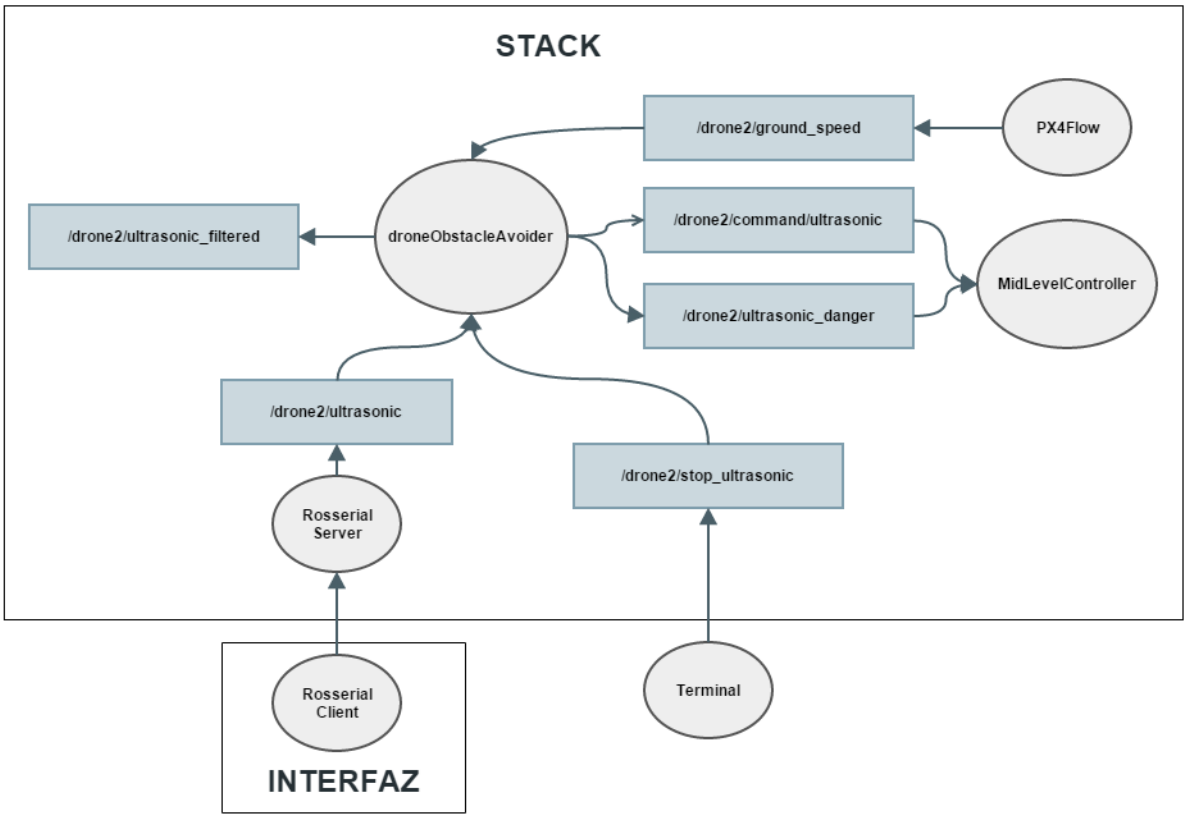

Figura 6.2: Esquema de comunicaciones del sistema

7 SIMULADOR

Una vez construido el sistema y conseguida su integración, se procedió a desarrollar un simulador cinemático que permita probar y validar los diseños realizados en capítulos anteriores. El simulador se ha programado en *Processing*.

7.1 Processing

Processing es a la vez un entorno de programación y un lenguaje de código abierto y gratuito basado en Java que apareció en 2001 en los laboratorios *MIT Media Lab*. Una de sus principales características es que proporciona herramientas muy sencillas para crear programas visuales que cuenten con interfaces gráficas.

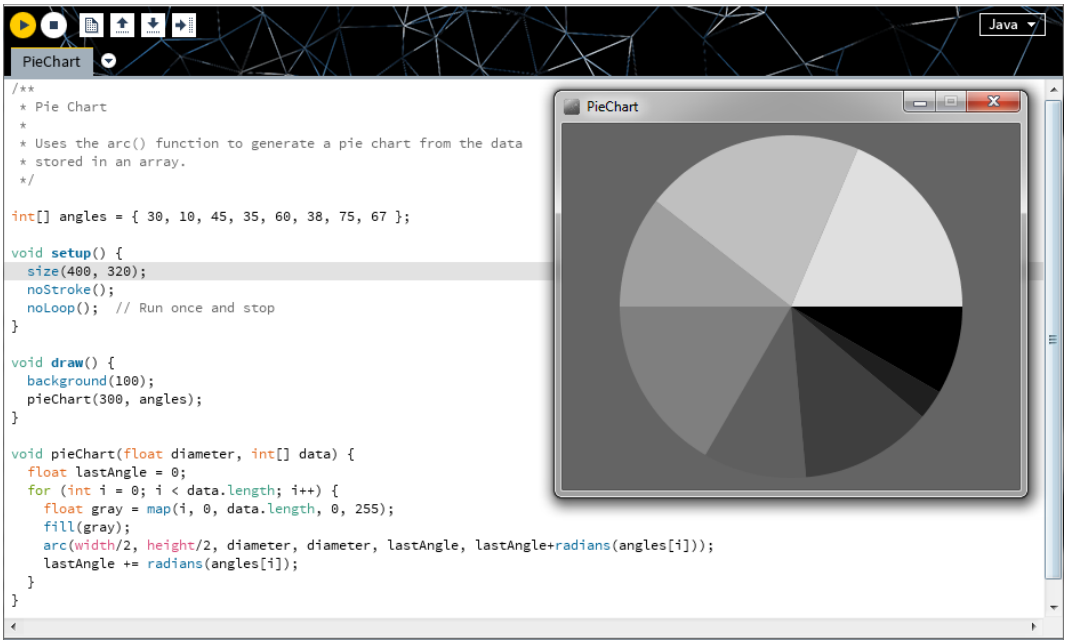

Figura 7.1: Interfaz del *IDE* de *Processing*. *Sketch* ejemplo incluido de serie 'PieChart'.

Los programas en *Processing* reciben el nombre de *sketches*. Cada *sketch* tiene que tener dos funciones principales que serán la base del código. Estas funciones son *setup* y *draw*:

- *setup*: es una función que se ejecuta una vez al comenzar el programa. En ella han de incluirse todos los parámetros de configuración, como el tamaño de la ventana o la tasa de refresco.
- *draw*: es la función del bucle principal. Se ejecuta lo que haya en ella cada ciclo, con lo que todas las actividades que quieran realizarse en el *sketch* tienen que ser invocadas desde aquí.

Dada la sencillez de su estructura y la facilidad con que se pueden integrar elementos visuales en el desarrollo de una aplicación, *Processing* se ha considerado idóneo para el desarrollo del simulador cinemático.

7.2 *Sketch* del simulador

El *sketch* desarrollado cuenta con tres objetos:

Dron

El dron es la estructura principal y tiene los siguientes atributos principales:

- Vector de posición.
- Velocidad en x, velocidad en y.
- Roll, pitch, y ángulo en el plano.

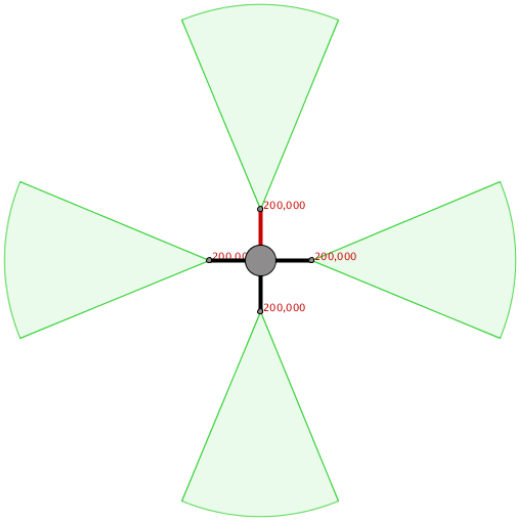

Figura 7.2: Simulación de un UAV y sus sensores.

Sensor

Los sensores son los objetos que detectan las distancias a los obstáculos en el simulador. Sus principales atributos son:

- Vector de posición
- Ángulo de apunte
- Apertura de detección
- Rangos mínimos y máximos

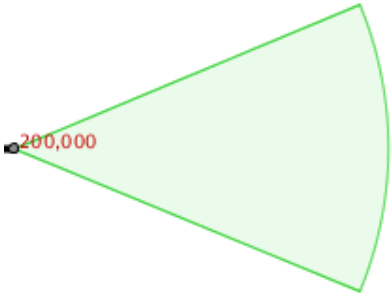

Figura 7.3: Sensor y su campo de detección.

Pared

Los obstáculos en el simulador se construyen a base de líneas rectas llamadas paredes. Toda pared se define por:

- Vector de posición del punto inicial.
- Vector de posición del punto final.

Figura 7.4: Obstáculo formado por tres paredes

7.3 Simulación de la medida de distancia

Uno de los problemas que debía resolver el simulador es la simulación de la forma más fiel posible de las mediciones de distancia obtenidas por los sensores. Para resolverlo, se ha utilizado la técnica siguiente:

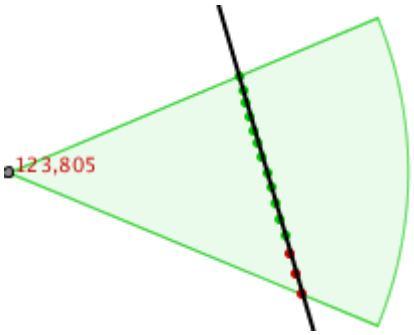

Figura 7.5. Medición de la distancia de una pared.

1. Se divide el ángulo de detección en quince partes, y se crean quince líneas virtuales hacia cada una de estas partes.

2. A continuación, se computa si alguna de estas líneas se cruza con cualquiera de las paredes existentes. El algoritmo que realiza esta comprobación, al igual que el resto del programa, se encuentra en el Anexo V.

3. En caso de que se crucen, se calcula el ángulo entre la línea virtual y la línea que forma el muro. Si dicho ángulo es menor de 60º, la medida se toma como inválida

(puntos rojos). Si es mayor de 60º, se calcula la distancia desde el sensor hasta el punto de intersección (puntos verdes). De este modo se simula de una forma realista una situación que ocurre a menudo en la que la medida no se recibe cuando se trata de medir superficies inclinadas respecto al sensor.

4. Por último, se toma como medida del sensor a la menor distancia entre todas las obtenidas en el paso anterior.

7.4 Simulación de las cinemáticas

Se introduce en el *sketch* la programación del modelo descrito en el apartado 1.2.3, que puede consultarse en la función *update()* perteneciente a la clase *Drone* en el código del programa. El programa actualiza de forma discreta los atributos de la instancia de *Drone* atendiendo a las ecuaciones que se desarrollaron en el modelo.

7.5 Validación de modelos y ajuste de parámetros

Una vez desarrollado el simulador, se procedió a comprobar si era posible conseguir un sistema eficaz de detector de colisiones y del posterior control reactivo atendiendo a las formas propuestas en los capítulos anteriores.

Tras un proceso iterativo de ajuste, se consiguió una configuración de parámetros que lo consiguió. La figura 7.6 muestra un ejemplo de detección de colisión contra una pared por el dron simulado y la corrección de su rumbo para alejarse de ella:

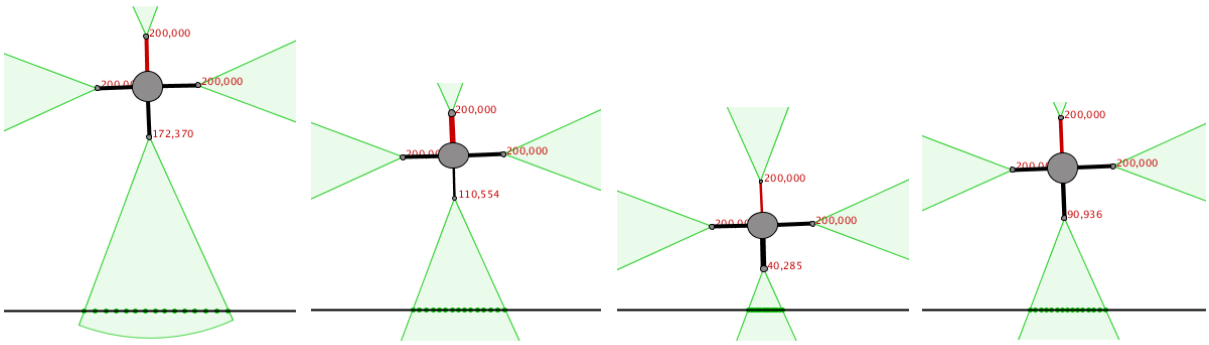

Figura 7.6: Esquivo de un obstáculo en el simulador

Cabe destacar que los parámetros encontrados en el simulador no son aplicables al sistema real en el dron, ya que dependerán de las propiedades estos, como pueden ser su peso o su envergadura. Para ajustar estos parámetros en un equipo real, será necesario realizar el ajuste a bordo del propio equipo.

La importancia de encontrar estos parámetros radica en la validación de los modelos propuestos, que ha demostrado ser positiva en el simulador. A modo indicativo, se recogen los resultados obtenidos:

Parámetros del detector de colisiones en el simulador

- Dm: $0.75m$
- Kv: 20
- SAV: $0.2\mathrm{m}$

Parámetros del control en el simulador

- Kd: 0.666
- Kv: 5.33

8 RESULTADOS Y CONCLUSIONES

En este capítulo se sintetizan los resultados obtenidos a lo largo del desarrollo del proyecto que se han ido consiguiendo en los apartados anteriores, seguidos de una valoración sobre los principales aspectos de los mismos.

En primer lugar, el estudio de las tecnologías de medición de distancia permite concluir que las más interesantes para llevar en un dron son la **tecnología de ultrasonidos** y de LIDAR rotatorio. Se decide utilizar la primera por sencillez, peso y coste.

Tras un análisis de la tecnología de ultrasonidos y sus limitaciones, se pone de manifiesto que las tres características más importantes para escoger un sensor en particular son su patrón de radiación, su fiabilidad en la toma de medidas, y su capacidad de anular interferencias por ruido. Atendiendo a estos parámetros se considera el sensor **MaxSonar XL-EZ0** como el más adecuado para el sistema y se compran cuatro unidades de ellos.

Se escoge un **Arduino Nano V3.0** como hardware de interfaz entre los sensores y el ordenador de a bordo del UAV, completando la parte de obtención de medidas de sensores, y se programa su comunicación con los sensores y el ordenador de a bordo. Con ello se obtienen unos resultados de medida, que aunque fiables en general, en ocasiones presentan irregularidades como los que se presentan en la figura 5.1 que se vuelve a mostrar a continuación:

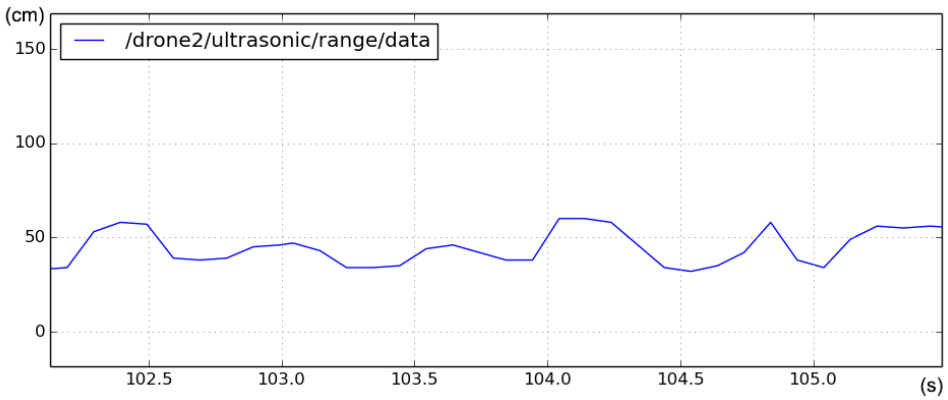

Figura 8.1. Medidas de distancia sin filtrar en condiciones adversas.

La presencia de picos y valles que se aprecia ocurre sólo en determinados casos pero que no se pueden ignorar. Estos casos incluyen la medida de superficies con poco grado de perpendicularidad al sensor, y objetos que se encuentran justo en el límite de detección del sensor, casos que pueden ocurrir de forma natural fácilmente.

El desarrollo del **filtro de Kalman** y el ajuste de las covarianzas de ruido que se realiza para minimizar el efecto de estos picos ofrece el siguiente resultado, que ya se presentó en el capítulo 5:

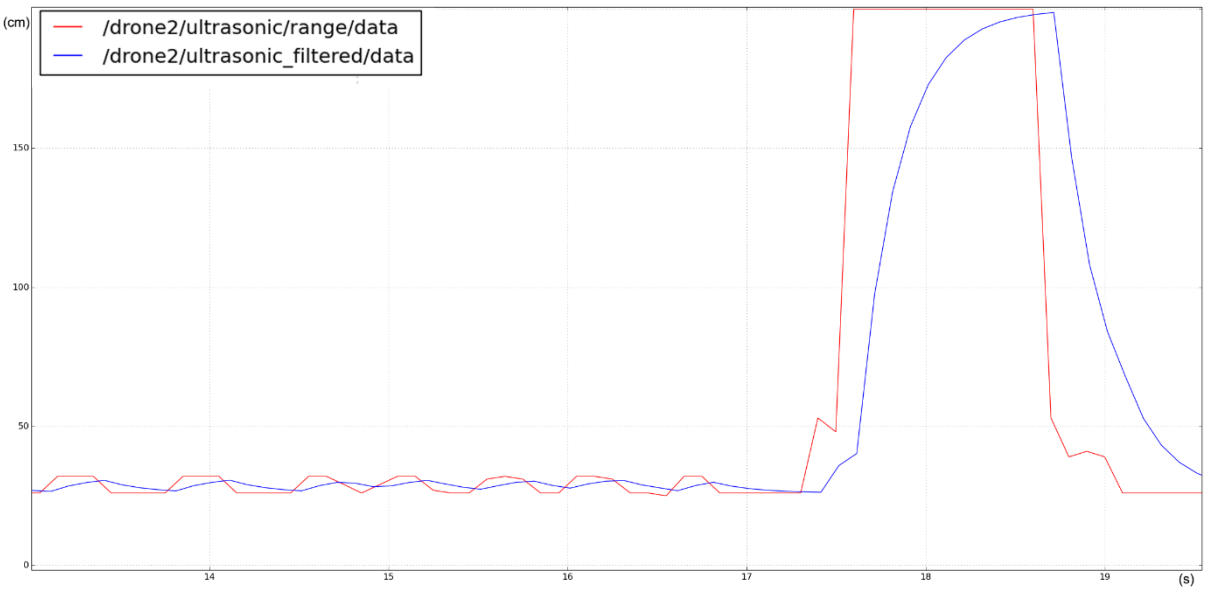

Figura 8.2: Comparación entre medidas sin filtrar y medidas filtradas.

Finalmente, la programación de los algoritmos del detector de colisiones y del control en cascada y su posterior prueba en el simulador nos permiten observar el resultado final del sistema, que consigue muy satisfactoriamente evitar colisiones contra obstáculos sencillos. La siguiente figura, que se presentó en el capítulo 7, muestra el resultado final del sistema desarrollado:

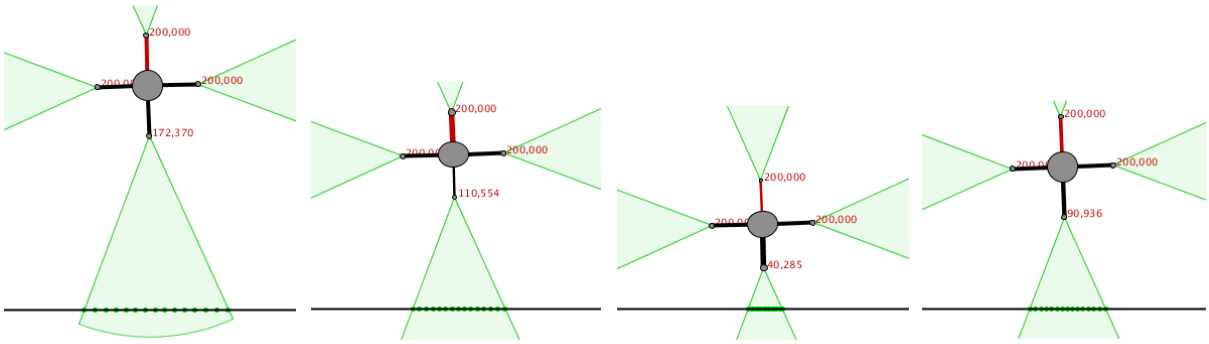

Figura 8.3: Esquivo de un obstáculo en el simulador

En ella se aprecia como el dron simulado consigue evitar una colisión contra una pared plana que se situaba en su trayectoria.

La finalización del proyecto cumple con los cuatro objetivos propuestos inicialmente, con lo que se considera un trabajo exitoso en su realización y con gran potencial de impacto para los drones del grupo Vision4UAV, aumentando la seguridad de las pruebas que realicen los drones del grupo en los que se monte el sistema.

Además, el Trabajo de Fin de Grado demuestra la posibilidad de desarrollar un sistema de protección contra choques en drones versátil, barato y eficiente, que en caso de extenderse podría contribuir al aumento general de la seguridad en estos vehículos y la consiguiente reducción del número de accidentes que sufren.

A nivel personal, el proyecto ha supuesto un reto muy enriquecedor, cuya persecución me ha permitido no sólo ampliar conocimientos, sino también comprender las dificultades prácticas que conlleva la aplicación de la teoría estudiada a lo largo del grado. Me siento muy orgulloso de los resultados obtenidos y agradecido por la experiencia adquirida trabajando con los miembros de Vision4UAV.

9 LÍNEAS FUTURAS

Una vez desarrollado el sistema y comprobado su viabilidad, está listo para su prueba y ajuste en entornos reales fuera del simulador, es decir, en los UAV del grupo Vision4UAV.

Para ello será necesario el ajuste conjunto de los parámetros del detector de colisiones y del control reactivo para cada dron particular en el que se quiera implementar, ya que sus valores dependerán de características específicas del aparato, como pueden ser su tamaño, su peso o la disposición de sus motores.

La flexibilidad del sistema empleado, que admite diferentes números de sensores e incluso diferentes disposiciones, hace que pasar de un dron a otro no requiera un excesivo esfuerzo. En la misma línea, todos los parámetros de ajuste son cargados en el programa desde un archivo de configuración fácilmente editable, con lo que no es necesario cambiar el código fuente ni recompilar el programa para distintas aplicaciones.

Como alternativa futura, se plantea el desarrollo de un sistema con las mismas características pero que se base en un LIDAR rotatorio en lugar de en sensores de ultrasonidos. Aunque más complejo, el sistema resultante dispondrá de una mayor cantidad de información, lo que le otorga el potencial de contribuir a la mejora de la seguridad de los UAV multirrotor de forma más importante.

10 PLANIFICACIÓN Y PRESUPUESTO

La realización del TFG comenzó a finales de enero de 2015 y se extendió hasta finales de junio del mismo año. Ha tenido una duración estimada de 450h de trabajo y se divide en dos periodos, separados por los exámenes finales que tuvieron lugar en la ETSII entre finales de mayo y principios de junio.

10.1 Estructura de Descomposición del Trabajo (EDT)

A continuación se presenta la Estructura de Descomposición del Trabajo junto con el diagrama de Gantt de los diferentes puntos y tareas por los que se ha formado.

1. **Preparación:**
    a. Estudio del estado del arte.
    b. Análisis de tecnologías de medición de distancias.
    c. Estudio de la tecnología de ultrasonidos.
    d. Planteamiento del sistema y sus partes.
    e. Instalación y aprendizaje de ROS y el *Stack.*
    f. Estudio de mercado de sensores de ultrasonidos.
    g. Elección de la interfaz.

2. **Desarrollo del sistema:**
    a. Construcción de la estructura
    b. Pruebas de modelos de sensores.
    c. Programación de la interfaz.
    d. Programación del filtro de Kalman.
    e. Programación del detector de colisiones.
    f. Programación del control.

3. **Pruebas y resultados:**
    a. Programación del simulador
    b. Validación del sistema y ajuste de parámetros en el simulador.
    c. Resultados y análisis de los mismos.

4. **Documentación:**
    a. Elaboración de la memoria.

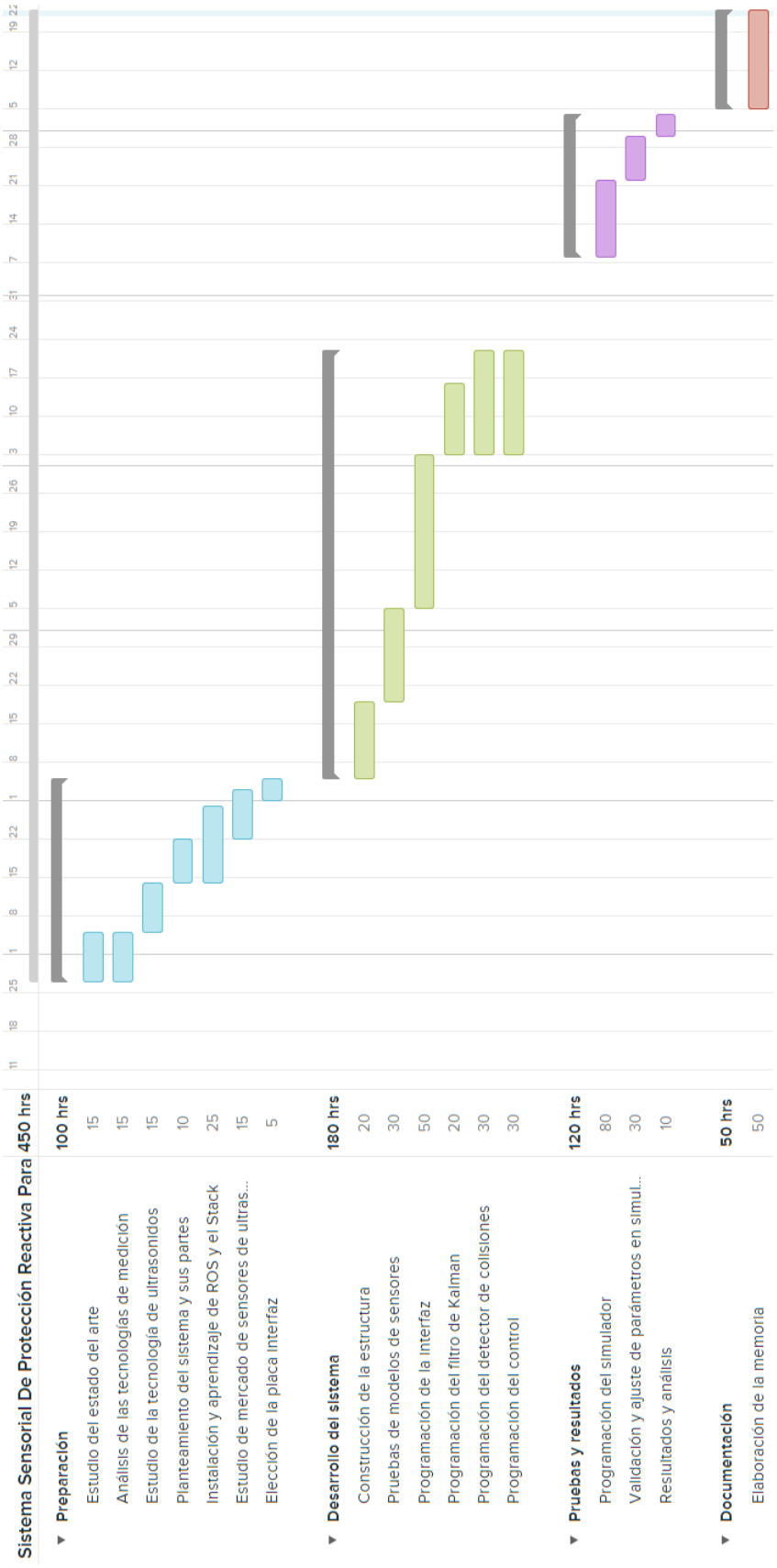

Figura 10.1: Diagrama de Gantt

10.2 Presupuesto

El presupuesto estimado para la realización del proyecto se compone principalmente por la suma de:

- Coste de personal
- Coste de equipo
- Coste de material informático
- Coste de componentes
- Encuadernación

El tiempo estimado para la realización del proyecto ha sido de 450h. Su salario se supone el salario medio de un ingeniero recién graduado, 12.4€ por hora. Además, se ha contado con la colaboración de aproximadamente 20h de tutorías, que se computan a sueldo de ingeniero veterano, de 40€ por hora.

Tabla 10.1: Costes de personal

Concepto	*Coste por hora (€)*	*Horas*	*Total (€)*
Salario ingeniero recién graduado	12.4	450	5580
Salario ingeniero veterano	20	40	800
Total			**6380**

En cuanto al equipo utilizado, el proyecto se ha desarrollado en un ordenador portátil de 900€ de coste, y se ha trabajado un número estimado de 40h días con el UAV Oktokopter de Mikrokopter. Se incluye también el soldador, que tiene un gasto marginal. Para ellos, se consideran las siguientes amortizaciones:

Ordenador portátil:

- Precio: 900€
- Vida estimada: 10 años con uso de 5h / día.
- Coste por hora: 0.06€/h

Oktokopter:

- Precio: 2600€
- Vida estimada: 3 años con uso de 3h / día
- Coste por hora: 0.96€/h

Estación de soldadura:

- Precio: 23.5€
- Vida estimada: 3000h
- Coste por hora: 0.008€

Tabla 10.2: Costes de equipo

Costes	*Coste por hora (€)*	*Horas*	*Total (€)*
Ordenador portátil	0.06	350	21
Oktokopter	0.96	40	38.4
Estación de soldadura	0.008	10	0.08
Total			**59.48**

Los costes de material informático son reducidos, ya que se han trabajado con herramientas de desarrollo de código abierto como *Processing*, ROS o Qtcreator. La única licencia de pago necesaria ha sido la del programa Word del paquete Microsoft Office.

Tabla 10.3: Costes de material informático

Concepto	*Coste (€)*
Licencia Microsoft Office 2010	**245**

Los costes de componentes se recogen en la siguiente tabla:

Tabla 10.4: Costes de componentes.

Concepto	*Coste por unidad (€)*	*Unidades*	*Total (€)*
Bolsa de 30 tornillos 3x20mm	2.85	1	2.85
1m de cable de 1.5mm de grosor	0.16	15	2.4
Sensor HC-SR04	2.99	1	2.99
Sensor PING)))	27.32	1	27.32
Sensor MaxSonar XL EZ0	40.95	4	163.8
Arduino Nano 3.0	39.20	1	39.20
Total			**238.56**

Finalmente, la encuadernación de la memoria supone un gasto de 60€, con lo que queda:

Tabla 10.5: Resumen de costes

Costes	*Cantidad (€)*
Personal	6380.00
Equipo	59.48
Material informático	245.00
Componentes	238.56
Encuadernación	60.00
Total	**6983.04**

Con lo que el coste total de la realización del proyecto asciende a SEISMIL NOVECIENTOS OCHENTA Y TRES EUROS CON CUATRO CÉNTIMOS.

10.3 Impacto social y ambiental

De ser desarrollado, el impacto social principal del proyecto vendrá en la reducción de víctimas de accidentes en los que hay UAV multirrotor involucrados. Las lesiones causadas por estos aparatos son numerosas y en ocasiones graves. El sistema construido en el proyecto contribuye a aumentar la seguridad de los drones multirrotor y a fomentar un uso más responsable.

El posible impacto ambiental positivo, difícil de cuantificar, pasa por la reducción del número de rotura de partes. Un número más bajo de estas significaría una reducción en los desechos que se generan en el mercado de estos UAV, algunos de ellos de difícil reciclaje como las baterías de Li-Po. Además, contribuiría a una bajada del número de piezas creadas, con la consiguiente bajada de la huella de carbono que conlleva su fabricación.

El impacto ambiental negativo se cuantifica como el consumo eléctrico que ha sido necesario para la elaboración del proyecto. Se procede a calcular la cuantía de la huella de carbono de dicho consumo eléctrico.

El ordenador utilizado es un MSI Apache 2Pe, que según sus especificaciones tiene un consumo medio de 75W.

$$Consumo\ del\ proyecto\ = 75W * 350h = 26.25\ KWh$$

Estimando una huella de carbono de 0.65kg de CO_2 por cada KWh, se tiene:

$$Huella\ de\ carbono\ = 26.25\ KWh \cdot \frac{0.65\ kg\ CO_2}{1\ KWh} = 17.06\ kg\ CO_2$$

Con lo que el impacto ambiental causado por el desarrollo del proyecto asciende a unas emisiones de 17.06kg de CO_2.

11 BIBLIOGRAFÍA

A Gibiansky. (2010). *Quadcopter Dynamics, Simulation, and Control*

Arlington. (2014). *Let Them Fly: CEA Applauds FAA's Ruling on Drones* - Consumer Electronics Journal.

Arturo Rodríguez et al. (2014). *Autonomous Pursuing Vehicle* – Department of Electronic Systems Student Report, Aalborg University.

Carnie R., Walker, R. y Corke, P. (2006). *Image processing algorithms for UAV "sense and avoid*" - IEEE Robotics and Automation 2006 issue.

Donald P. Massa (1999). *An Overview of Some Fundamentals of Electroacoustics* – Massa Products Corporation

Donald P. Massa (1999) *Choosing an Ultrasonic Sensor for Proximity or Distance Measurement* – Sensors online. Obtenido de http://www.sensorsmag.com/sensors/acoustic-ultrasound/choosing-ultrasonic-sensor-proximity-or-distance-measurement-838.

J Pestana, JL Sanchez-Lopez, R Suarez-Fernandez, JF Collumeau, P Campoy, J Martin-Cristobal, M Molina, J De Lope, D Maravall (2014). *A Vision Based Aerial Robot solution for the IARC 2014 by the Technical University of Madrid*

Jeng-Tze Huang et al. (2014). *Indoor Autonomous Navigation Technology Research Based on Lidar for Quad-Copter Aerial Robot.*

Jose Luis Sanchez-Lopez, Jesús Pestana, Paloma de la Puente, Adrian Carrio, Pascual Campoy (2014). *Visual Quadrotor Swarm for the IMAV 2013 Indoor Competition* - ROBOT2013: First Iberian Robotics Conference

Kalman, R. E. (1960). *A New Approach to Linear Filtering and Prediction Problems, Transactions of the ASME* - Journal of Basic Engineering Vol. 82

Kristian Lauzus. (2012). *A practical approach to Kalman filter and how to implement it.* Obtenido de http://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it/.

Luis Rodolfo García Carrillo, Alejandro Enrique Dzul López, Rogelio Lozano, Claude Pégard, (2011), *Combining Stereo Vision and Inertial Navigation System for a Quad-Rotor UAV* – Journal of Intelligent and Robotic Systems, Vol. 65.

M Quigley et al. (2009). *ROS: an open-source Robot Operating System*

MaxBotix MaxSonar XL datasheet. (2015). Obtenida de http://www.maxbotix.com/documents/XL-MaxSonar-EZ_Datasheet.pdf

Omid Shakernia, Won-Zon Chen y Vince Raska. (2005). *Passive Ranging for UAV Sense and Avoid Applications* - The American Institute of Aeronautics and Astronautics.

P. C. P. Pounds y R. Mahony, (2006). *Modelling and control of a quad-rotor robot*.

Paul Allen Tipler, Gene Mosca (2005). *Fundamentos de ondas* – Tipler Mosca 3ª edición, capítulo 7.

PRNewsWire (2015). *Global Commercial Drones Market 2015-2020: Market is estimated to grow at a CAGR of 109.31% to reach $1.27 billion by 2020. Obtenido de* http://www.prnewswire.com/

Samir Bouabdallah, Andre Noth and Roland Siegwart. (2004). *PID vs LQ Control Techniques Applied to an Indoor Micro Quadrotor* - Autonomous Systems Laboratory at Swiss Federal Institute of Technology paper.

T Luukkonen (2011). *Modelling and control of quadcopter* – Independent project at School of Science, Aalto University

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ABREVIATURAS Y ACRÓNIMOS

CAR: Centro de Automática y Robótica

CVG: Computer Vision Group

EDT: Estructura de Descomposición del Trabajo

ESC: *Electronic Speed Controller*

ETSII: Escuela Técnica Superior de Ingenieros Industriales

IDL: *Interface Definition Language*

IR: Infrarrojos

LED: *Light-Emitting Diode* (Diodo Emisor de Luz)

LIDAR: *Laser Imaging Detection And Ranging* (Detección y Medición de distancias por Imagen Láser)

PD: Proporcional Derivativo

ROS: *Robot Operating System*

STAIR: *STanford AI Robot*

TFG: Trabajo de Fin de Grado

UAV: *Unmaned Aerial Vehicle* (Vehículo Aéreo No Tripulado)

UPM: Universidad Politécnica de Madrid

GLOSARIO

Aeronave: Vehículo capaz de navegar por el aire.

Conversor analógico-digital: Dispositivo electrónico capaz de convertir una señal analógica en una digital.

Dron: Aeronave que vuela sin tripulación. En el contexto del proyecto, se refiere a dron multirrotor, que son drones propulsadas por más de dos hélices.

Filtro: Sistema que realiza un procesamiento de la señal de entrada para mejorar sus condiciones.

Ganancia: Expresa la relación entre el valor de la salida y el valor de la entrada de un sistema.

Interfaz de sensor: En el contexto del proyecto, se ha referido como Interfaz de sensor al hardware y software intermedio presente entre el sensor y el ordenador de a bordo del dron.

Modelo: Conjunto de ecuaciones que rigen el comportamiento de un sistema.

Nodo de ROS: Figura del esquema comunicativo de ROS que maneja el envío y la recepción de comunicaciones con el resto de nodos.

Patrón de radiación: Representación gráfica de la potencia radiada en función del ángulo y distancia desde el transductor.

Pitch, Roll* y *Yaw: Ángulos de Euler que indican la orientación tridimensional de un objeto.

Processing: Entorno de desarrollo de lenguaje abierto basado en Java orientado a la programación de aplicaciones con interfaz gráfica.

Publicador de ROS: Figura del esquema comunicativo de ROS que envía mensajes para que otros los reciban.

Sense and Avoid: Conjunto de técnicas que permiten obtener información del entorno para evitar colisiones contra los objetos circundantes de sistemas autónomos.

Sensor: Dispositivo capaz de detectar magnitudes químicas o físicas y transformarlas en variables eléctricas.

Simulación: "Proceso de diseñar un modelo de un sistema real y llevar a término experiencias con él, con la finalidad de comprender el comportamiento del sistema o evaluar nuevas estrategias -dentro de los límites impuestos por un cierto criterio o un conjunto de ellos - para el funcionamiento del sistema." (R. E. Shannon).

Sistema autónomo: Sistema que no necesita intervención humana, más allá de la preprogramación, para realizar las tareas para las que ha sido diseñado.

***Sketch*:** Nombre que reciben los programas en el entorno de programación *Processing*.

Stack: En el contexto del TFG, el código preexistente controlador de los drones desarrollado por Vision4UAV.

Suscriptor de ROS: Figura del esquema comunicativo de ROS que recibe mensajes que otros han enviado.

***Topic* de ROS**: Figura del esquema comunicativo de ROS que conforma el canal por el que los mensajes se envían y reciben.

Transductor: Dispositivo que convierte pequeños valores de una forma de energía a otra. En el proyecto, forman parte de los sensores de ultrasonidos y transforman energía eléctrica en acústica y viceversa.

Ultrasonidos: Sonidos por encima del umbral superior de audición del hombre, de 20.000 Hz.

ANEXOS

ANEXO I. CÓDIGO DE LA INTERFAZ EN ARDUINO

Ultrasonic_interface.ino

```
/*    rosrun rosserial_python serial_node.py _port:=/dev/ttyUSB0
*
*
----------------------------------*/

/*--------------------------------
* Includes
----------------------------------*/

#include <ros.h>
#include <ros/time.h>
#include <ultrasound/RequestUpdate.h>


#include <droneMsgsROS/ultrasonicArduinoService.h>
#include <droneMsgsROS/UltrasonicRangeArduino.h>

ros::NodeHandle nh;
using droneMsgsROS::ultrasonicArduinoService;

droneMsgsROS::UltrasonicRangeArduino range_msg;

ros::Publisher range_publisher( "/drone2/ultrasonic", &range_msg);

void ultrasonicArduinoServiceCallback(const ultrasonicArduinoService::Request &
req, ultrasonicArduinoService::Response & res);
ros::ServiceServer<ultrasonicArduinoService::Request,
ultrasonicArduinoService::Response>
ultrasonicArduinoService_service("ultrasonicArduinoService_srv",&ultrasonicArduino
ServiceCallback);


unsigned int minimum_close_range = 100;
const int number_of_sensors = 1;
const int max_range = 200;
unsigned long range_time;



class uSensor{

  private:

    char id;
    unsigned int pin;
    unsigned int range;
    unsigned int frequency=10;
    unsigned int state=2;  //0 Off, 1 Close,  2 Normal

    long last_publish;
```

```
    ros::Time measurement_time;

  public:

    uSensor(int myID, int myPin){
      id=myID;
      pin=myPin;
      last_publish=millis()+500; // 500ms Antes de realizar la primera medidicón
para que se estabilice.
    }


    void updateRange(){
      /**
      Filtra la lectura del puerto analógico para reducir posibles ruidos y
convierte el valor a centimetros.
      */

      range = 0;
      int range_measurement[4], measurement_difference[4];
      int max_difference=0, max_difference_index=0;

      // Lee 4 veces el pin analogico correspondiente
      for(int i=0;i<4;i++) range_measurement[i]= analogRead(pin);
      // Se elimina la medida más alejada de las demás
      for(int i=0;i<4;i++) measurement_difference[i] =
      abs(range_measurement[i]-range_measurement[0]) +
      abs(range_measurement[i]-range_measurement[1]) +
      abs(range_measurement[i]-range_measurement[2]) +
      abs(range_measurement[i]-range_measurement[3]);

      for(int i=0;i<4;i++){
        if(measurement_difference[i]>max_difference){
          max_difference=measurement_difference[i];
          max_difference_index=i;
        }
      }
      range_measurement[max_difference_index]=0;

      // Se promedian las tres restantes (la eliminada valdrá cero en la suma)
      for(int i=0;i<4;i++) range += range_measurement[i];

      range = floor(range / 2.052); // (0.684 * 3) -> conversion a centimetros
      if ( range > max_range ) range = max_range;
      measurement_time=nh.now();

    }


    void publish(){

      /**
      Vuelca los datos del sensor en el mensaje y lo envía por Serial al ordenador
de a bordo.
      */

      range_msg.id=id;
```

```
      range_msg.header.stamp=measurement_time;
      range_msg.range.data=range;
      range_msg.freq.data=frequency;
      range_publisher.publish(&range_msg);
      last_publish = millis() - 2;   // tiempo que tarda en publicar

    }


    int checkPublish(){


      if( millis() > last_publish + 1000/frequency ) {

        if      ( state == 2 ) { publish(); return 1;}
        else if ( state == 1 && range < minimum_close_range ) { publish(); return
1;}

      }

    return 0;

    }


    /**
    Getters y setters
    */

    unsigned int getRange(){

      return range;

    }

    char getId(){

      return id;

    }

    unsigned int getFrequency(){

      return frequency;

    }

    unsigned int getState(){

      return state;

    }

    void setRange( unsigned int range_){

       range=range_;
```

```
    }

    void setFrequency( unsigned int freq_){

      frequency=freq_;

    }

    void setState( unsigned int state_){

      state=state_;

    }

};

uSensor sensor[4]={uSensor(0,A0),uSensor(1,A1),uSensor(2,A2), uSensor(3,A3)};
void setup(){

  nh.initNode();
  nh.advertise(range_publisher);
  nh.advertiseService(ultrasonicArduinoService_service);
 // nh.advertiseService(RequestUpdate_service);
  range_time=millis()+500;
  range_msg.ok=true;

}



void loop(){

  if ( millis() >= range_time ){

    for(int i=0;i<number_of_sensors;i++){
      if(sensor[i].getState() != 0) sensor[i].updateRange();
    }

    range_time=millis()+100; // Cada 100ms se miden los sensores
  }

  for(int i=0;i<number_of_sensors;i++){
    sensor[i].checkPublish();
  }

  nh.spinOnce();
}


void ultrasonicArduinoServiceCallback(const ultrasonicArduinoService::Request &
req, ultrasonicArduinoService::Response & res){                              //

  //Set
  if( req.req_id == -1 ){
    for(int i=0;i<number_of_sensors;i++){
      if( req.req_mode.data >=0 && req.req_mode.data <=2 )
sensor[i].setState(req.req_mode.data);
```

```
      if( req.req_frequency.data >= 1 && req.req_frequency.data <=20 )
sensor[i].setFrequency(req.req_frequency.data);
    }
  }

  else if (req.req_id < number_of_sensors ){
    if( req.req_mode.data >=0 && req.req_mode.data <=2 )
sensor[req.req_id].setState(req.req_mode.data);
    if( req.req_frequency.data >= 1 && req.req_frequency.data <=20 )
sensor[req.req_id].setFrequency(req.req_frequency.data);
  }

  //Get
  if ( req.req_id >=0 && req.req_id <= number_of_sensors ){
    if(sensor[req.req_id].getState() != 0) sensor[req.req_id].updateRange();

  }


}
```

ANEXO II. CODIGO DEL FILTRO DE KALMAN

droneUltrasonicEstimator.h

```
#ifndef ULTRASONICESTIMATOR_H_
#define ULTRASONICESTIMATOR_H_

#include "UltrasonicRangeArduino.h"
#include "xmlfilereader.h"
#include <ros/console.h>

#include <string.h>

class UltrasonicEstimator {

private:
    static const int number_of_sensors=4;  //CONFIG

    double estimated_range[number_of_sensors];
    double unfiltered_range[number_of_sensors];
    double previous_estimated_range[number_of_sensors];
    float P[number_of_sensors];
    float Q;//????     Mayor R/Q  -> Mayor importancia a la velocidad que al
sensor (el ruido del sensor desaparece pero se vuelve mas lento)
    float R;//??
    float sensorAngle[number_of_sensors];  //CONFIG


public:
    UltrasonicEstimator(int idDrone, const std::string &stackPath_in);

    void filter(float velocity_x, float velocity_y,
droneMsgsROS::UltrasonicRangeArduino measure);
    inline float getSensorAngle(int i) { return sensorAngle[i]; }
    inline double getRange(int i) { return estimated_range[i]; }
    inline double getPreviousRange(int i) { return previous_estimated_range[i]; }
    inline double getUnfilteredRange(int i) { return unfiltered_range[i]; }
    inline void setRQratio (float ratio) { R=ratio; }   //Greater ratio -> greater
weight for velocities rather than new measures when updating distances
};

#endif
```

droneUltrasonicEstimator.cpp

```
#include "droneUltrasonicEstimator.h"

#define FEEDFORWARD 0

UltrasonicEstimator::UltrasonicEstimator(int idDrone, const std::string
&stackPath_in){


    try {
```

```
            XMLFileReader
my_xml_reader(stackPath_in+"configs/drone"+cvg_int_to_string(idDrone)+"/midlevel_a
utopilot.xml");

            Q =                         my_xml_reader.readDoubleValue(
"midlevel_autopilot_config:ultrasound_estimator:Q" );
            R =                         my_xml_reader.readDoubleValue(
"midlevel_autopilot_config:ultrasound_estimator:R" );
            for (int i=0; i< number_of_sensors; i++){
                sensorAngle[i] =     my_xml_reader.readDoubleValue(
"midlevel_autopilot_config:ultrasound_estimator:SENSOR_ANGLE_" +
cvg_int_to_string(i) );
                P[i] =               my_xml_reader.readDoubleValue(
"midlevel_autopilot_config:ultrasound_estimator:INITIAL_UNCERTAINTY");
                estimated_range[i] = my_xml_reader.readDoubleValue(
"midlevel_autopilot_config:ultrasound_estimator:INITIAL_RANGE");

            }

        } catch ( cvg_XMLFileReader_exception &e) {
            throw
cvg_XMLFileReader_exception(std::string("[cvg_XMLFileReader_exception!
caller_function: ") + BOOST_CURRENT_FUNCTION + e.what() + "]\n");
        }
}



void UltrasonicEstimator::filter(float velocity_x, float velocity_y,
droneMsgsROS::UltrasonicRangeArduino measure){

    unfiltered_range[measure.id] = measure.range.data/100.0;

    float theta = sensorAngle[measure.id];
    float v = velocity_x*cos(theta)+velocity_y*sin(theta);  //Solo componente en
direccion del sensor ?  //serÃ-a +sin o -sin? Creo que -
    float t = 1/measure.freq.data + FEEDFORWARD;

    //Filtro de Kalman sobre rango
    double predicted_range = estimated_range[measure.id] - v*t;
    float predicted_P = P[measure.id]+Q;
    float Kg = predicted_P/(predicted_P+R);
    previous_estimated_range[measure.id] = estimated_range[measure.id];
    estimated_range[measure.id] = predicted_range + Kg * (measure.range.data/100.0
- predicted_range);
    P[measure.id] = (1 - Kg)*predicted_P;
    ROS_INFO("Received measure from sensor %d: %lf",measure.id,
estimated_range[measure.id]);

}
```

ANEXO III. CÓDIGO DEL DETECTOR DE COLISIONES Y DE LA UNIDAD DE CONTROL

droneObstacleAvoider.h

```
#ifndef DRONEOBSTACLEAVOIDER_H_
#define DRONEOBSTACLEAVOIDER_H_

#include "droneUltrasonicEstimator.h"
#include "xmlfilereader.h"
#include <ros/console.h>
#include <string.h>

class DroneObstacleAvoider {
public:
    DroneObstacleAvoider(int idDrone, const std::string &stackPath_in);

    static const int number_of_sensors=4;

    void enable(bool e);
    bool dangerDetected();
    inline bool isEnabled() { return enabled; }
    inline double getPitch() { return pitch_cmd; }
    inline double getRoll() { return roll_cmd; }
    bool run(double* roll_, double* pitch_);
    void checkDanger();
    inline void setSafety(bool stop) { safety = stop; }
    float getDetectionDistance();
    void filterUltrasonic(droneMsgsROS::UltrasonicRangeArduino msg);
    void stopUltrasonicControl (bool stop);
    void setDroneMeasurementGroundOpticalFlow(float vx_mps, float vy_mps);
    float getRange();

private:
    bool enabled;
    bool danger[number_of_sensors];
    bool safety;
    int token[number_of_sensors];
    float detection_distance;
    float dist_to_safe[number_of_sensors];
    double pitch_cmd;
    double roll_cmd;
    double time_gap;
    UltrasonicEstimator ultrasonicEstimator;

    // Configuration parameters
    bool   SYSTEM_MOUNTED;
    double GAIN_DIST, GAIN_SPEED;
    double MAX_OUTPUT;
    float last_ground_speed_X_measurement, last_ground_speed_Y_measurement;

public:
    bool close();
    bool reset();
    bool start();
    bool stop();
```

```
    bool run();
};


#endif
```

droneObstacleAvoider.cpp

```
#include "droneObstacleAvoider.h"


DroneObstacleAvoider::DroneObstacleAvoider(int idDrone, const std::string
&stackPath_in) :
    ultrasonicEstimator(idDrone, stackPath_in)
{

    try {
            XMLFileReader
my_xml_reader(stackPath_in+"configs/drone"+cvg_int_to_string(idDrone)+"/midlevel_a
utopilot.xml");

            GAIN_DIST =       my_xml_reader.readDoubleValue(
"midlevel_autopilot_config:obstacle_controller:GAIN_DIST" );
            GAIN_SPEED =      my_xml_reader.readDoubleValue(
"midlevel_autopilot_config:obstacle_controller:GAIN_SPEED" );
            MAX_OUTPUT =      my_xml_reader.readDoubleValue(
"midlevel_autopilot_config:obstacle_controller:MAX_OUTPUT" );
            AGGR_GAIN  =      my_xml_reader.readDoubleValue(
"midlevel_autopilot_config:obstacle_controller:AGGR_GAIN"  );
            int sys_mounted = my_xml_reader.readIntValue(
"midlevel_autopilot_config:obstacle_controller:SYSTEM_MOUNTED" );
            if(sys_mounted==1) SYSTEM_MOUNTED = true;
            else SYSTEM_MOUNTED = false;

        } catch ( cvg_XMLFileReader_exception &e) {
            throw
cvg_XMLFileReader_exception(std::string("[cvg_XMLFileReader_exception!
caller_function: ") + BOOST_CURRENT_FUNCTION + e.what() + "]\n");
        }

    enabled = false;
    for(int i=0;i<number_of_sensors;i++)danger[i]=false;
    safety = false;
    pitch_cmd = 0;

    roll_cmd = 0;
    time_gap = 0.1;
}

void DroneObstacleAvoider::enable(bool e)
{
    if (e && SYSTEM_MOUNTED && !safety) enabled = e;
    else (enabled = false);
    //ROS_INFO("enabled = %d",enabled);
}

bool DroneObstacleAvoider::run(double* roll_, double* pitch_)
```

```
{
    roll_cmd=0;
    pitch_cmd=0;
    double action_command;

    for(int i=0; i<number_of_sensors; i++)
    {
        if(danger[i]){
            float theta = ultrasonicEstimator.getSensorAngle(i);
            action_command = AGGR_GAIN * ( GAIN_DIST * (0.55 -
ultrasonicEstimator.getRange(i)) + GAIN_SPEED / time_gap *
(ultrasonicEstimator.getPreviousRange(i) - ultrasonicEstimator.getRange(i)));

            pitch_cmd += action_command*cos(theta);
            roll_cmd  -= action_command*sin(theta);
        }
    }
    //DUDA TIME GAP

    if (pitch_cmd > MAX_OUTPUT) pitch_cmd = MAX_OUTPUT;
    if (pitch_cmd < -MAX_OUTPUT) pitch_cmd = -MAX_OUTPUT;

    if (roll_cmd > MAX_OUTPUT) roll_cmd = MAX_OUTPUT;
    if (roll_cmd < -MAX_OUTPUT) roll_cmd = -MAX_OUTPUT;

    //ROS_INFO("Ultrasonic distance = %f",ultrasonicEstimator.getRange(0));
    ROS_INFO("Pitch command = %lf!", pitch_cmd);
    ROS_INFO("Roll command  = %lf!",  roll_cmd);

    *roll_  = roll_cmd;
    *pitch_ = pitch_cmd;
}

void DroneObstacleAvoider::checkDanger()
{
    for(int i=0; i<number_of_sensors; i++)
    {
        float vel_sensor = (ultrasonicEstimator.getPreviousRange(i) -
ultrasonicEstimator.getRange(i)) / time_gap;
        if (vel_sensor > 1) vel_sensor = 1;
        if (vel_sensor < -1) vel_sensor =-1;
        float speed_safety = 0;
        if(vel_sensor > 0.15) speed_safety = 0.15;
        if( i==0) detection_distance = 0.4 + speed_safety
+vel_sensor*vel_sensor*20;
        if(!danger[i] && ultrasonicEstimator.getRange(i) < 0.4 + speed_safety
+fmax(0,vel_sensor)*vel_sensor*2) {
            token[i]++;
            if(token[i]>=2){
                danger[i] = true;
                dist_to_safe[i]= ultrasonicEstimator.getRange(i);
                if(dist_to_safe[i] > 1) dist_to_safe[i] = 1;
                if(dist_to_safe[i] < 0.4) dist_to_safe[i] = 0.4;
            }
        }
        else token[i]=0;
```

```
        if(danger[i]  && ultrasonicEstimator.getRange(i) > dist_to_safe[i]) {
danger[i] = false; ROS_INFO("DANGER %d",i);}
    }
}

float DroneObstacleAvoider::getDetectionDistance(){
    return detection_distance;
}

bool DroneObstacleAvoider::dangerDetected(){
    for(int i=0; i<number_of_sensors; i++) if(danger[i]) return true;
    return false;
}

void DroneObstacleAvoider::filterUltrasonic(droneMsgsROS::UltrasonicRangeArduino
msg)
{

ultrasonicEstimator.filter(last_ground_speed_X_measurement,last_ground_speed_Y_mea
surement,msg);
}

void DroneObstacleAvoider::stopUltrasonicControl(bool stop)
{
    safety = stop;
    if (stop==true) enabled = false;
}

void DroneObstacleAvoider::setDroneMeasurementGroundOpticalFlow(float vx_mps,
float vy_mps)
{
    last_ground_speed_X_measurement = vx_mps;
    last_ground_speed_Y_measurement = vy_mps;
}

float DroneObstacleAvoider::getRange()
{
    //Function used for debugging purposes
    return ultrasonicEstimator.getRange(0);
}

bool DroneObstacleAvoider::stop()
{
    return true;
}

bool DroneObstacleAvoider::close()
{
    stop();
    return true;
}

bool DroneObstacleAvoider::reset()
{
    return true;
}
```

```
bool DroneObstacleAvoider::start()
{
    return true;
}
```

ANEXO IV. CÓDIGO DE LA INTEGRACIÓN EN EL STACK

droneObstacleAvoiderROSModule.h

```
#ifndef DRONE_OBSTACLE_AVOIDER_H_
#define DRONE_OBSTACLE_AVOIDER_H_

#include "ros/ros.h"
#include "droneModuleROS.h"
#include "droneObstacleAvoider.h"
#include "droneMsgsROS/dronePitchRollCmd.h"
#include "communication_definition.h"
#include "droneMsgsROS/vector2Stamped.h"
#include "std_msgs/Bool.h"
#include "std_msgs/Float32.h"

const float OBSTACLE_AVOIDER_RATE = 10.0;

class DroneObstacleAvoiderROSModule : public DroneModule
{

protected:
    DroneObstacleAvoider MyObstacleAvoider;

public: // Constructors and destructors
    DroneObstacleAvoiderROSModule();
    ~DroneObstacleAvoiderROSModule();
public: // Init and close, related to Constructors and destructors
    void init();
    void close();

protected: // DroneModule
    bool resetValues(); //Reset
    bool startVal(); //Start
    bool stopVal(); //Stop
public:
    bool run();     //Run
public: // Open, initialize subscribers and publishers
    void open(ros::NodeHandle & nIn, std::string moduleName);

    // Subscribers
private:
    ros::Subscriber drone_rotation_angles_subscriber;
    ros::Subscriber drone_ultrasonic_subscriber;
    void droneUltrasonicCallback(const droneMsgsROS::UltrasonicRangeArduino& msg);
    ros::Subscriber drone_ground_optical_flow_subscriber;
    void droneGroundOpticalFlowCallback(const droneMsgsROS::vector2Stamped& msg);
    ros::Subscriber drone_stop_ultrasonic_subscriber;
    void droneStopUltrasonicControlCallback(const std_msgs::Bool &msg);


    ros::Publisher ultrasonic_danger_publisher;
    void publishUltrasonicDanger();

    ros::Publisher ultrasonic_filtered_publisher;
    void publishFiltered();
```

```
    ros::Publisher ultrasonic_command_pitch_roll_publisher;
    void publishUltrasonicCommand();

    ros::Publisher drone_autopilot_command_publisher;
    bool publishDroneAutopilotCommand();

    //DEBUGR//
    ros::Publisher pitch_rafa_publisher;
    ros::Publisher ultra_rafa_publisher;
    bool publishRafa();

    double pitch_command;
    double roll_command;
};



#endif
```

droneObstacleAvoiderROSModule.cpp

```
#include "droneObstacleAvoiderROSModule.h"


DroneObstacleAvoiderROSModule::DroneObstacleAvoiderROSModule():
    DroneModule(droneModule::active,OBSTACLE_AVOIDER_RATE),
    MyObstacleAvoider(getId(), stackPath)
{
    std::cout << "DroneObstacleAvoiderROSModule(...), enter and exit" <<
std::endl;
    init();
    return;
}

DroneObstacleAvoiderROSModule::~DroneObstacleAvoiderROSModule()
{
    close();
    return;
}

void DroneObstacleAvoiderROSModule::init()
{
    return;
}


void DroneObstacleAvoiderROSModule::close()
{
    if(!MyObstacleAvoider.close())
        return;
    return;
}

bool DroneObstacleAvoiderROSModule::resetValues()
{
    if(!DroneModule::resetValues())
        return false;
```

```
    if(!MyObstacleAvoider.reset())
        return false;

    return true;
}

//Start
bool DroneObstacleAvoiderROSModule::startVal()
{
    if(!DroneModule::startVal())
        return false;

    if(!MyObstacleAvoider.start())
        return false;

    return true;
}

//Stop
bool DroneObstacleAvoiderROSModule::stopVal()
{
    if(!DroneModule::stopVal())
        return false;

    if(!MyObstacleAvoider.stop())
        return false;

    return true;
}

void DroneObstacleAvoiderROSModule::open(ros::NodeHandle & nIn, std::string
moduleName)
{
    DroneModule::open(nIn,moduleName);

    /*
    // Sensor Measurement
    */
    ultrasonic_command_pitch_roll_publisher =
n.advertise<droneMsgsROS::vector2Stamped>
(DRONE_ULTRASONIC_PITCH_ROLL_COMMAND,1,true);
    ultrasonic_danger_publisher             = n.advertise<std_msgs::Bool>
(DRONE_ULTRASONIC_DANGER_DETECTED,   1,true);
    drone_ultrasonic_subscriber             = n.subscribe(
DRONE_DRIVER_SENSOR_ULTRASONIC, 4,
&DroneObstacleAvoiderROSModule::droneUltrasonicCallback       , this);
    drone_stop_ultrasonic_subscriber        = n.subscribe(
DRONE_DRIVER_STOP_ULTRASONIC, 1,
&DroneObstacleAvoiderROSModule::droneStopUltrasonicControlCallback, this);

    //Flag of module opened
    droneModuleOpened=true;

    //Autostart the module
    moduleStarted=true;
    MyObstacleAvoider.start();
```

```
    ultrasonic_filtered_publisher = n.advertise<std_msgs::Float32>
("ultrasonic_filtered",1,true);

    //End
    return;
}

void DroneObstacleAvoiderROSModule::droneStopUltrasonicControlCallback(const
std_msgs::Bool &msg)
{
    MyObstacleAvoider.stopUltrasonicControl(msg.data);
}


void DroneObstacleAvoiderROSModule::droneGroundOpticalFlowCallback(const
droneMsgsROS::vector2Stamped &msg)
{
    MyObstacleAvoider.setDroneMeasurementGroundOpticalFlow(msg.vector.x,
msg.vector.y);

}

void DroneObstacleAvoiderROSModule::droneUltrasonicCallback(const
droneMsgsROS::UltrasonicRangeArduino &msg)  ///Acordarse de meter este mensaje
bien
{
    MyObstacleAvoider.filterUltrasonic(msg);
}


bool DroneObstacleAvoiderROSModule::run(){

    MyObstacleAvoider.checkDanger();
    MyObstacleAvoider.run(&roll_command, &pitch_command);

    publishUltrasonicDanger();
    publishUltrasonicCommand();
    return true;
}

void DroneObstacleAvoiderROSModule::publishUltrasonicCommand()
{
    droneMsgsROS::vector2Stamped current_msg;
    current_msg.vector.x = pitch_command;
    current_msg.vector.y  = roll_command;
    current_msg.header.stamp = ros::Time::now();
    ultrasonic_command_pitch_roll_publisher.publish(current_msg);
}

void DroneObstacleAvoiderROSModule::publishUltrasonicDanger()
{
    std_msgs::Bool current_msg;
    current_msg.data = MyObstacleAvoider.dangerDetected();
    ultrasonic_danger_publisher.publish(current_msg);
}
```

```
void DroneObstacleAvoiderROSModule::publishFiltered(){
    std_msgs::Float32 current_msg;
    current_msg.data = MyObstacleAvoider.getRange();
    ultrasonic_filtered_publisher.publish(current_msg);
}
```

droneObstacleAvoiderROSModuleNode.cpp

```
#include <iostream>

//ROS
#include "ros/ros.h"
#include "communication_definition.h"
#include "droneObstacleAvoiderROSModule.h"

int main(int argc,char **argv)
{
    // Ros Init
    ros::init(argc, argv, "droneObstacleAvoider");
    ros::NodeHandle n;


    DroneObstacleAvoiderROSModule MyRosObstacleAvoider;
    MyRosObstacleAvoider.open(n,MODULE_NAME_DRONE_OBSTACLE_AVOIDER);

    try
    {
        while(ros::ok())
        {
            //Read messages
            ros::spinOnce();

            MyRosObstacleAvoider.run();

            MyRosObstacleAvoider.sleep();

        }
        return 1;
    }
    catch (std::exception &ex)
    {
        std::cout<<"[ROSNODE] Exception :"<<ex.what()<<std::endl;
    }
}
```

ANEXO V. CÓDIGO DEL SIMULADOR

Drone_Simulator.pde

```
class Drone{
  PVector position;
  float angle=5*PI/4;
  float speedx=1;
  float speedy=0;
  Sensor[] sensor;

  Drone(){
  sensor = new Sensor[4];
  position = new PVector(500,200);

  }

  void update(){
  position.x = position.x + speedx*cos(angle) + speedy*sin(angle);
  position.y = position.y + speedy*cos(angle) - speedx*sin(angle);


  sensor[0] = new Sensor (position.x+30*cos(angle),position.y-30*sin(angle),
angle);
  sensor[1] = new Sensor (position.x-30*sin(angle),position.y-30*cos(angle),
angle+PI/2);
  sensor[2] = new Sensor (position.x-30*cos(angle),position.y+30*sin(angle),
angle+PI);
  sensor[3] = new Sensor (position.x+30*sin(angle),position.y+30*cos(angle),
angle-PI/2);

  for(int i=0;i<4;i++){
    textSize(10);
    fill(200,0,0);
    text(sensor[i].getMeasure(),sensor[i].position.x,sensor[i].position.y);
  }

  }

  void draw(){
    strokeWeight(4);
    stroke(200,0,0);
    line(position.x+30*cos(angle),position.y-30*sin(angle),position.x,position.y);
    stroke(0);
    line(position.x-30*sin(angle),position.y-
30*cos(angle),position.x+30*sin(angle),position.y+30*cos(angle));
    line(position.x-30*cos(angle),position.y+30*sin(angle),position.x,position.y);

    fill(#8E8C8C);
    strokeWeight(1);
    ellipse(position.x,position.y,20,20);

    for(int i=0;i<4;i++)sensor[i].draw();

    }

};
```

```
class Sensor{
 PVector position;
 float angle,apperture=PI/4;
 int number_of_beams=15;


 Sensor(float x_, float y_, float angle_){
   position = new PVector(x_, y_);
   angle=angle_;
 }

 void draw(){
   ellipse(position.x,position.y,5,5);
 }

 float getMeasure(){
   PVector beam[];
   beam = new PVector[number_of_beams];
   float range;
   float min_range=200;
   for (int i=0; i< number_of_beams; i++){
      beam[i] = new PVector(position.x + 200*cos(angle - apperture/2 +
i*apperture/(number_of_beams-1)), position.y - 200 * sin(angle - apperture/2 +
i*apperture/(number_of_beams-1)));
      if ( i==0 || i == number_of_beams-1) {

        strokeWeight(1);
        stroke(0,200,0,100);
        fill(0,200,0,10);
        arc(position.x,position.y, 400, 400, 2*PI - (angle + apperture/2), 2*PI -
(angle - apperture/2), PIE);
        line(position.x,position.y, beam[i].x, beam[i].y);
      }
   }
   for(int i=0; i<wallcount; i++){
      for (int j=0; j<number_of_beams; j++){
        if(linesIntersect( position, beam[j], wall[i].start, wall[i].end)) {
          range = linesDistance ( position, beam[j], wall[i].start, wall[i].end );
          if (range < min_range) min_range = range;
        }

      }
   }

   fill(200,0,0);
   return min_range;
 }

};

class Wall{

  PVector start, end;
  boolean booting=true;
  boolean detected=false;
  float angle;
```

```
  Wall (int startX, int startY){
    start = new PVector(startX, startY);
    end = new PVector(startX, startY);

  }


  void draw(){
    strokeWeight(2);
    if(detected) stroke(255,0,0);
    else stroke(0);
    if(booting) line(start.x,start.y, mouseX, mouseY);
    else line(start.x,start.y,end.x,end.y);
    stroke(0);
  }

  void complete(){
    end.x=mouseX;
    end.y=mouseY;
    booting=false;
    angle = atan ( - ( end.y - start.y ) / (end.x - start.x));
    if ( end.x < start.x) angle = angle + PI;
    println(angle);
  }


};

Drone my_drone;
Wall[] wall;
int wallcount=0;
int max_distance=200;
float[] ranges;

void setup(){

  size(900,900);
  background(255);
  my_drone = new Drone();
  wall = new Wall[50];
  ranges = new float[4];
 //frameRate(3);
}


void draw(){
  background(255);
  my_drone.update();
  my_drone.draw();
  for (int i=0; i<wallcount; i++) wall[i].draw();
  controlAction();
}


void mousePressed(){
```

```
  if(mouseButton==LEFT && mouseButton!=RIGHT){
    wall[wallcount] = new Wall(mouseX,mouseY);
    wallcount++;
  }

}

void mouseReleased(){

  if(mouseButton==LEFT && mouseButton!=RIGHT) if(wallcount!=0)wall[wallcount-
1].complete();

}



boolean linesIntersect(PVector p1, PVector q1, PVector p2, PVector q2)
{
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    if (o1 != o2 && o3 != o4)
        return true;

    return false; // Doesn't fall in any of the above cases
}

int orientation(PVector p, PVector q, PVector r)
{
    float val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0;  // colinear

    return (val > 0)? 1: 2; // 1 es horario, 2 antihorario
}

float linesDistance(PVector p1, PVector q1, PVector p2, PVector q2)
{

  float A1 = q1.y - p1.y;
  float B1 = p1.x - q1.x;
  float C1 = A1*p1.x + B1*p1.y;

  float A2 = q2.y - p2.y;
  float B2 = p2.x - q2.x;
  float C2 = A2*p2.x + B2*p2.y;

  float det = A1*B2 - A2*B1;

  if(det==0) return 0;
  else{
    PVector intersection = new PVector((B2*C1-B1*C2)/det, (A1*C2 - A2*C1)/det);
    if ( angleBetweenLines ( p1, q1, p2, q2 ) > PI/3){
      strokeWeight(5);
```

```
      stroke(0,200,0);
      point(intersection.x,intersection.y);
      if (p1.dist(intersection) < 25) return 25;
      else return p1.dist(intersection);
    }
    else{
      strokeWeight(5);
      stroke(200,0,0);
      point(intersection.x,intersection.y);
      return 200;
    }
  }

}

float angleBetweenLines ( PVector p1, PVector q1, PVector p2, PVector q2 ){
    float angle1 = atan ( - ( q1.y - p1.y) / (q1.x - p1.x));
    float angle2 = atan ( - ( q2.y - p2.y) / (q2.x - p2.x));

    float angle = abs( angle2 - angle1 );
    if (angle > PI/2 ) angle = PI - angle;

    return angle;


}


float[] integral = {0,0,0,0};
float[] last_proportional = {0,0,0,0};
boolean[] control_on= {false,false,false,false};
boolean[] safe = {true,true,true,true};
float[] last_derivative = {0,0,0,0};

void controlAction(){

    float control_pitch=0;
    float control_roll=0;
    float derivative;
    float proportional;

    for (int i=0; i<4; i++){
    ranges[i] =
ranges[i]+my_drone.armlength*cos(my_drone.pitch)*abs(cos(my_drone.sensor[i].angle
-
my_drone.angle))+my_drone.armlength*cos(my_drone.roll)*abs(sin(my_drone.sensor[i].
angle - my_drone.angle))-my_drone.armlength;
    if(ranges[i] < 25) ranges[i]=25;
    if (ranges[i] < 75 + 20*cos(my_drone.sensor[i].angle-
my_drone.angle)*my_drone.speedx - 20*sin(my_drone.sensor[i].angle-
my_drone.angle)*my_drone.speedy && control_on[i] ==false) { control_on[i] = true;
integral[i] = 0; last_proportional[i] = 100 - ranges[i]; safe[i]=false;}
    if (ranges[i] > 75 && safe[i]) control_on[i] = false;

    if( control_on[i]) {
      proportional = max(0, 100 - ranges[i]);
      if(ranges[i]!=25) {
```

```
        derivative = proportional - last_proportional[i];

      }
      else derivative = proportional - last_proportional[i];
      integral[i] += proportional;

      // Remember the last position.
      float debug = last_proportional[i];
      last_proportional[i] = proportional;
      last_derivative[i] = derivative;

      if(derivative < 0) { /*derivative = 0;*/ safe[i] = true;}
      if(derivative > 10) derivative = 10;
      float action = - (proportional + integral[i]/3000 + derivative*8) / 150; //

      println(i + "P: " + proportional + " I: " + integral[i]/3000 + " D: " +
derivative*8 + " LP: " + debug);
      control_pitch += cos(my_drone.sensor[i].angle-my_drone.angle)*action;
      control_roll += -sin(my_drone.sensor[i].angle-my_drone.angle)*action;
    }
    }
    //if (control_pitch != 0) println("control pitch:  " + control_pitch + ";  |||
control roll:  " + control_roll +";");
    control2drone(control_pitch,control_roll);
}

void control2drone(float pitch, float roll)
{

    if (abs(pitch) >= 0.01){
      if(pitch < my_drone.pitch) my_drone.pitch += max( -0.1, (pitch -
my_drone.pitch) );
      if(pitch > my_drone.pitch) my_drone.pitch += min(  0.1, (pitch -
my_drone.pitch) );
    }
    if (abs(roll ) >= 0.01) {
      if(roll < my_drone.roll) my_drone.roll += max( -0.1, (roll - my_drone.roll)
);
      if(roll > my_drone.roll) my_drone.roll += min(  0.1, (roll - my_drone.roll)
);
    }
}

void keyPressed(){
 switch(key){
    case 'w':
      my_drone.pitch+=0.03;
      break;

    case 's':
      my_drone.pitch-=0.03;
      break;

    case 'q':
      my_drone.roll-=0.03;
      break;
```

```
    case 'e':
      my_drone.roll+=0.03;
      break;

    case 'a':
      my_drone.angle+=0.1;
      break;

    case 'd':
      my_drone.angle-=0.1;
      break;


    case 'r':
      wallcount=0;
      break;

    case ' ':
      my_drone.roll=0;
      my_drone.pitch=0;
      my_drone.speedx=0;
      my_drone.speedy=0;
      break;
 }

}
```