Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Proyecto Fin de Carrera
Ingeniería Industrial
Automatización Industrial y Robótica

Desarrollo de un cuadricóptero operado por ROS

Iván Monzón Catalán

Director: Georgios Nikolakopoulos
Ponente: Ana Cristina Murillo Arnal

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Control Engineering Group
Department of Computer Science, Electrical and Space Engineering
Luleå University of Technology

Septiembre 2013

RESUMEN

Este proyecto se centra en el desarrollo de un cuadricóptero y su control integrado en el entorno de ROS. ROS (Robotic Operating System) es un pseudo sistema operativo orientado a plataformas robóticas. El trabajo desarrollado cubre desde el manejo del sistema operativo en distintas plataformas robóticas o el estudio de las diversas formas de programación en ROS hasta la evaluación de alternativas de construcción, desarrollo de la interfaz con ROS o ensayos prácticos con la plataforma construida.

En primer lugar, se ha realizado un estudio de las posibilidades de ROS aplicadas a robots voladores, las alternativas de desarrollo y su viabilidad de integración. Entre estas aplicaciones cabe destacar las de SLAM (Localización y Mapeo Simultáneos) y navegación autonoma.

Tras la evaluación de las distintas alternativas considerando funcionalidad, autonomía y precio, la plataforma de desarrollo se ha basado en ArduCopter. Aunque existen algunos ejemplos de vehículos aéreos no tripulados en ROS, no hay soporte para este sistema, por lo cual se ha desarrollado el trabajo necesario para hacer estas dos plataformas compatibles.

El hardware ha sido montado sobre una plataforma de fabricación propia, realizada mediante impresión 3D, y se ha evaluado su funcionamiento en entornos reales. También se ha valorado y ensayado una plataforma de aluminio, con resultados menos satisfactorios.

Para el correcto funcionamiento del conjunto se ha tenido que conseguir una conexión entre el cuadricóptero y la estación de tierra. En este caso, se han diseñado alternativas de conexión entre ordenadores (para el caso de que se monte un ordenador en la aeronave) o conexión entre ordenador y ArduCopter (para el caso de que no haya ordenador de a bordo).

También se ha implementado una serie de algoritmos para llevar a cabo el control del cuadricóptero de manera autónoma: navegación de puntos vía, control de la rotación y control de altitud. Estos módulos funcionan bajo el sistema ROS y operan en remoto desde la estación de tierra.

Finalmente, se ha desarrollado un módulo de lectura para una unidad de medida inercial actualmente en desarrollo por la universidad de Luleå (KFly). Este dispositivo sólo se ha probado en entornos controlados y aún no ha pasado a formar parte del cuadricóptero, aunque en un futuro próximo se espera que sirva de reemplazo al ordenador de a bordo.

PRÓLOGO

Este trabajo se ha realizado como un Proyecto de Fin de Carrera para completar mis estudios en Ingeniería Industrial, con especialidad en Automatización Industrial y Robótica, en la Universidad de Zaragoza (España). El proyecto ha sido desarrollado y presentado en la Universidad Técnica de Luleå (Suecia) con el equipo del Departamento de Informática, Ingeniería Eléctrica y del Espacio; a quienes agradezco enormemente su apoyo y colaboración.

Me gustaría agradecer al Programa Erasmus por darme la oportunidad de ir a esta maravillosa ciudad sueca, Luleå. A Georgios Nikolakopoulos, mi supervisor en esta empresa, por su apoyo y optimismo. Y a Ana Cristina Murillo, quien me ha ayudado en todas las fases españolas.

Fue un duro y largo viaje de más de seis meses, lleno de retrasos y problemas. Pero ahora puedo decir que el viaje valió la pena.

Índice general

Capítulo 1 – Introducción **1**

1.1. Motivación y trabajo relacionado 1
1.2. Objetivos del proyecto 3
1.3. Entorno de trabajo 4
1.4. UAVs (Vehículos aéreos no tripulados) 5
1.4.1. Cuadricópteros 7
1.5. Contenido de la memoria 8

Capítulo 2 – Diseño y construcción **9**

2.1. Estructura y fijaciones 9
2.1.1. Impresión 3D 10
2.2. Hardware 11
2.2.1. Arducopter 11
2.2.2. Sensores 15
2.3. Plataformas de seguridad 16

Capítulo 3 – Programación y Desarrollo **19**

3.1. ROS: Sistema Operativo para Robots 19
3.1.1. Definición 19
3.1.2. Uso de la plataforma y primeros ensayos 20
3.2. Arquitectura del sistema 25
3.2.1. IMU: Unidad de medición inercial 25
3.2.2. Módulo de interfaz con RosCopter 28
3.2.3. Módulos de simulación 29
3.2.4. Módulo para control del vuelo 31
3.3. Evaluación del Algoritmo de Control 32

Capítulo 4 – Conclusiones y Trabajo Futuro **37**

4.1. Conclusiones 37
4.2. Trabajo Futuro 38

Apéndice A – Código fuente de los módulos desarrollados **39**

A.1. Analizador sintáctico 41
A.2. RosCopter 45

A.3. Simulación 48
1.3.1. Simulación 48
1.3.2. Simulación con entrada real 51
A.4. Programa de Control 55
1.4.1. Go 55
1.4.2. Landing 59

APÉNDICE B – IMÁGENES Y PLANOS **61**
B.1. Impresora 3D 63
B.2. Planos de la impresión 3D 64
B.3. Montaje 67

APÉNDICE C – DIAGRAMA DE GANTT **69**

MEMORIA EN INGLÉS **73**
CONTENTS 81
CHAPTER 1 – INTRODUCTION 83
CHAPTER 2 – ROS 93
CHAPTER 3 – BUILDING THE QUADROTOR 109
CHAPTER 4 – FUTURE WORK 123
CHAPTER 5 – CONCLUSION 125

GLOSARIO **127**

BIBLIOGRAFÍA **129**

CAPÍTULO 1

Introducción

1.1. Motivación y trabajo relacionado

Este proyecto de investigación se enmarca en la creciente necesidad de construir un vehículo volador, ágil y rápido, con la suficiente flexibilidad en software para poder considerarlo una aeronave de desarrollo. Para ello, se consideró un cuadricóptero (helicóptero de cuatro rotores, como el ejemplo de la Figura 1.1) como plataforma y ROS[1] como sistema operativo. El primero, se eligió por su habilidad para mantener una posición estable en el aire y su capacidad para hacer maniobras complejas, y el segundo, por su fundamento modular, que permite programar tareas de forma independiente del robot que las vaya a ejecutar; esto permite la creación de una gran comunidad de desarrollo que ayuda a acelerar enormemente las investigaciones. En este contexto se inicia un proyecto desde cero en la Universidad de Luleå para construir y poner en marcha este UAV[2].

La fase de documentación y aprendizaje de las plataformas se ha realizado, en una gran mayoría, haciendo uso de comunidades y manuales online; por lo que el trabajo de las universidades y laboratorios que están actualmente inmersos en ello ha sido de gran ayuda. En particular se han estudiado trabajos de las siguientes cinco universidades. Por un lado, el CCNY Robotics Lab de New York está desarrollando, en cooperación con AscTec, el Pelican y el Hummingbird, cuyos paquetes de software están completamente recogidos en los repositorios de ROS. La Universidad de Pennsylvania tiene una larga lista de experimentos con Penn Quadrotors, incluyendo vuelos acrobáticos o juegos de luces con cuadricópteros cooperativos. El Cheetah BRAVO (Figura 1.1) de PixHawk está siendo desarrollado por la ETH de Zurich, y usa un sistema de cámaras estéreo para llevar a cabo la navegación en interiores. El Autonomy Lab, en la Universidad Simon Fraser de Canadá, está desarrollando un controlador basado en ROS para el Parrot AR.Drone (Figura 1.2), uno de los pocos cuadricópteros comerciales. Y por úl-

[1] *Robotic Operating Sistem* (Sistema Operativo para Robots), es un pseudo sistema operativo que usado en plataformas robóticas acelera y facilita el desarrollo, permitiendo el uso de módulos ya desarrollados y con varios lenguajes de programación.

[2] *Unmanned Aerial Vehicle* (Vehículo Aéreo no Tripulado)

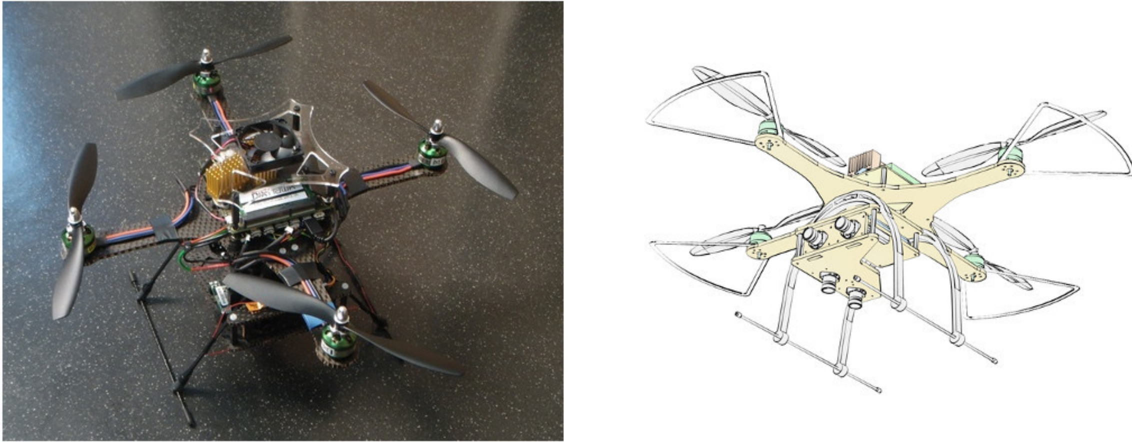


Figura 1.1: PixHawk Cheetah BRAVO

timo, el proyecto STARMAC (Figura 1.3) de la Universidad de Stanford se centra en explorar la cooperación de un sistema multivehículo con un controlador multiagente.

Figura 1.2: Parrot AR.Drone

En cada uno de los trabajos se hace uso de un hardware específico, y en algunos casos se trata de componentes construidos para el correspondiente experimento, como las placas de bajo nivel o las IMUs[3]. Esto restringe la utilización de ROS en un hardware específico. Es

[3] La Unidad de Medida Inercial (*Inertial Measurement Unit*) es una placa de sensores que provee 9 grados de libertad (velocidades, aceleraciones y fuerzas gravitacionales).

por ello, que en este proyecto se ha investigado un enfoque de desarrollo más independiente que permita a cualquiera la creación de su propio cuadricóptero. Más detalles sobre trabajo relacionado en la Sección 1.4.1.

Figura 1.3: Uno de los vehículos de STARMAC en vuelo

1.2. Objetivos del proyecto

Las metas establecidas para este proyecto van desde tareas de documentación y aprendizaje, hasta montaje del prototipo y ensayos en un entorno real. En más detalle, los objetivos del proyecto son:

- Estudiar y poner en marcha la plataforma ROS, incluyendo tanto el entendimiento de su funcionamiento y sus protocolos como su manejo, así como el conocimiento para programar en ella en C++ y en Python y la capacidad de modificar y adaptar programas ya existentes.

- Realizar un estudio de las posibilidades de ROS aplicadas a robots voladores, las alternativas de desarrollo y su viabilidad de integración. Entre estas aplicaciones se consideran necesarias aplicaciones de localización, mapeo y control de movimiento.

- Construir un prototipo. Una vez evaluadas las alternativas se procederá al montaje de un cuadricóptero y al desarrollo de la mejor opción considerando funcionalidad, autonomía y precio. Una estructura creada mediante una impresora 3D y un módulo basado en Arduino Mega (ArduCopter) son los puntos de donde se quiere partir.

- Módulo de conexión remota. Es necesario desarrollar una conexión del cuadricóptero con la estación de tierra, para lo cual se considerarán las diferentes alternativas de comunicación. Para el caso de que se instale un ordenador en la aeronave, deberá haberse desarrollado un módulo de comunicación entre dos ordenadores.
- Desarrollar un módulo de lectura para una unidad de medida inercial (IMU) actualmente en desarrollo por la universidad de Luleå (KFly). Este módulo será también ensayado en tareas de control para en un futuro poder reemplazar los dispositivos de a bordo por este dispositivo y, en el caso de que se viera necesario, un ordenador.
- Desarrollo de módulo de control. Una vez el conjunto esté montado y sea funcional, se desarrollará un control sencillo para dirigir al cuadricóptero a la posición y orientación deseadas.
- Desarrollo de una serie de pruebas sobre entorno real.
- Documentación de la construcción, las opciones evaluadas y los módulos desarrollados para facilitar su uso posterior.

El tiempo aproximado dedicado a cada tarea queda reflejado en el diagrama de Gantt del Apéndice C.

1.3. Entorno de trabajo

El proyecto se ha realizado bajo el amparo del programa de becas ERASMUS, que acoge a más de 250.000 estudiantes cada año, en la Universidad Técnica de Luleå. El trabajo se inició en el grupo de control, en el laboratorio de robótica, ante el reto de poder hacer funcionar una aeronave con un sistema operativo propio y controlada de forma autónoma. El reto se vio incrementado por el hecho de que en la Universidad no había nadie con conocimiento profundo de dicha plataforma, pero la ilusión y el apoyo entre los integrantes del laboratorio nos hizo decidirnos por seguir adelante y hasta el final con este proyecto.

Para su desarrollo se ha contado con el material disponible en el laboratorio del grupo de Control de la LTU (*Luleå Technical University*) y con diverso material adquirido específicamente a tal fin, como material informático, piezas mecánicas o elementos de seguridad.

El desarrollo del proyecto se ha realizado sobre el sistema operativo Ubuntu 12.04 LTS junto con ROS Fuerte, en ambos casos, debido a su estabilidad y su amplia comunidad de ayuda. En cuanto al hardware utilizado, las tareas iniciales de aprendizaje se han realizado en un robot Turtlebot[4]. Y para la construcción del cuadricóptero se ha usado una unidad inercial KFly [1], un ArduCopter (más detalles de los componentes del cuadricóptero en la Sección 2.2) y una impresora 3D para crear todas las piezas plásticas. También se ha contado con una serie de plataformas de seguridad para conseguir en todo momento niveles de seguridad apropiados tanto para el prototipo como para las personas en su entorno.

[4]Robot basado en la aspiradora Roomba. `http://www.willowgarage.com/turtlebot`

1.4. UAVs (Vehículos aéreos no tripulados)

La robótica lleva siglos con nosotros [2], pero hasta mediados de los años cincuenta no empezó a cobrar forma [3]. Fue entonces cuando comenzó una revolución con los primeros robots industriales, y más aun con los primeros robots programables (años setenta). Hoy en día, robots de corte, soldadura o manipulación, entre otros, forman parte de la industria. Durante estas cuatro o cinco décadas, también se han hecho investigaciones más avanzadas; DARPA[5], en 1987, ya había desarrollado un vehículo autónomo [4], o en el año 2000, Honda empezaba a hacer historia con Asimo [5], el primer robot humanoide. Se puede encontrar más información sobre la historia de la robótica en el Apéndice D (Section 1.1). Entre todo este asombroso abanico de posibilidades robóticas, la investigación que aquí nos ocupa se centra especialmente en robots voladores autónomos. Por ello, dicho campo se tratará con mayor extensión.

Los UAVs, del inglés *Unmanned Aerial Vehicle*, o vehículos aéreos no tripulados, llevan en nuestro mundo desde la Primera Guerra Mundial, aunque por aquellos tiempos estaban muy centrados en entornos militares. Entonces, se utilizaban como una forma de volar más segura (al menos para el bando que los controlara), ya que no requerían de piloto. Además, y por esta misma razón, se podía hacer que las aeronaves fueran más pequeñas, más aerodinámicas, que consumieran menos combustible y que se movieran de una forma más ágil.

Hoy, todas esas características siguen pareciéndonos extremadamente útiles, pero además se ha expandido su campo de acción. Como ya se ha comentado en la sección anterior, si bien es cierto que sigue habiendo importantes investigaciones en drones militares (como son llamados de forma coloquial), también en drones civiles.

Un UAV, con mayor o menor grado de inteligencia, puede comunicarse con su controlador para devolverle datos de la imagen (térmica, óptica, etc.), así como información referente a su estado, posición, velocidad del aire, orientación, altitud o cualquier parámetro de su telemetría. Generalmente, estos vehículos están armados con sistemas que en caso de fallo de cualquiera de sus componentes o programas toman medidas correctivas o alertan al operador. En los casos más inteligentes, son capaces de tomar medidas “imaginativas” para afrontar un problema no esperado.

Como en todos los campos, se pueden clasificar los UAVs utilizando multitud de características. Tamaño, autonomía, forma de sustentación o utilidad serían las más acertadas. La Figura 1.4 muestra algunos de los UAVs actuales. Desde baratos y pequeños juguetes que puedes comprar en cualquier parte como aviones radio control, hasta grandes proyectos militares como el Predator. Pueden ser pilotados remotamente, como el Bell Eagle Eye [7], o autónomos, como el AAI Shadow [8].

Centrándonos en los UAVs de bajo coste, en la Figura 1.5 se puede ver una clasificación de dichas máquinas por la física de vuelo empleada. Los UAVs de ala fija tienen la ventaja de un consumo contenido, una mecánica simple y un mayor sigilo en operación, ya que pueden volar sin partes móviles y planear; por contra, su movilidad no es demasiado buena. Son buenos para volar desde un punto hasta otro, pero no gestionan bien maniobras rápidas o en espacios

[5] Agencia de Proyectos de Investigación Avanzados de Defensa

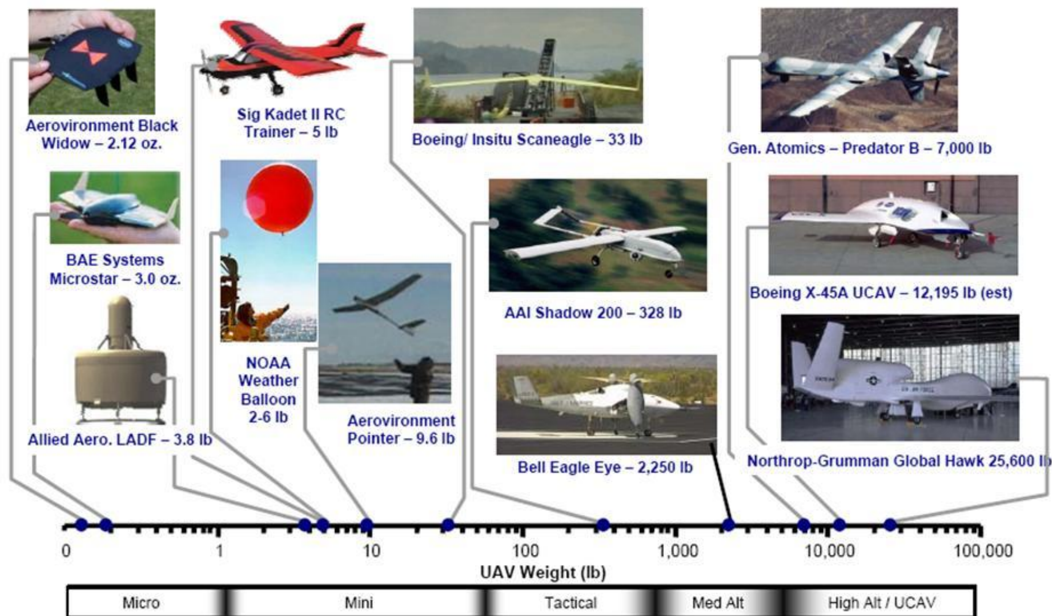


Figura 1.4: Diferentes UAVs comerciales [6]

contenidos.

Los helicópteros autónomos normalmente tienen un mayor consumo y una mecánica más compleja. Pero tienen un mejor manejo, pueden operar en lugares inhóspitos, en pequeños espacios y rutas complejas. En este grupo, un miembro especial es el helicóptero multi rotor. Tiene algunos problemas de consumo energético, con lo que sus baterías no duran demasiado, pero supera a casi todos los otros UAVs en maniobrabilidad, simplicidad mecánica e incluso en facilidad de manejo.

A partir de este punto, el informe se centrará en los vehículos autónomos y de bajo coste por su alto potencial de crecimiento y, más específicamente, en los helicópteros multi rotor autónomos. Pueden tomar decisiones más rápido que los humanos y, más importante, si pierden la conexión con la estación de tierra, pueden seguir funcionando.

La evolución continúa con el tamaño. Con pequeños robots voladores se pueden hacer tareas más precisas (como en [10]), atravesar rutas más estrechas o aumentar su sigilo. También se hace más fácil trabajar con matrices de robots cooperativos al reducir el tamaño de los mismos. Por ejemplo, si se considera un UAV para SLAM, uno más pequeño podría llegar a hacer vuelos más bajos y lograr una reconstrucción del entorno mucho más precisa, reconstrucción que antes requería un vehículo terrestre con cámaras. Además, con ellos, se puede tener una dimensión Z más grande que ayuda a reconstruir altos edificios, montañas, árboles, etc. Hoy en día, es posible mezclar reconstrucciones aéreas con reconstrucciones de tierra en el mismo vehículo, reduciendo tiempo y dinero y haciéndolo más preciso.

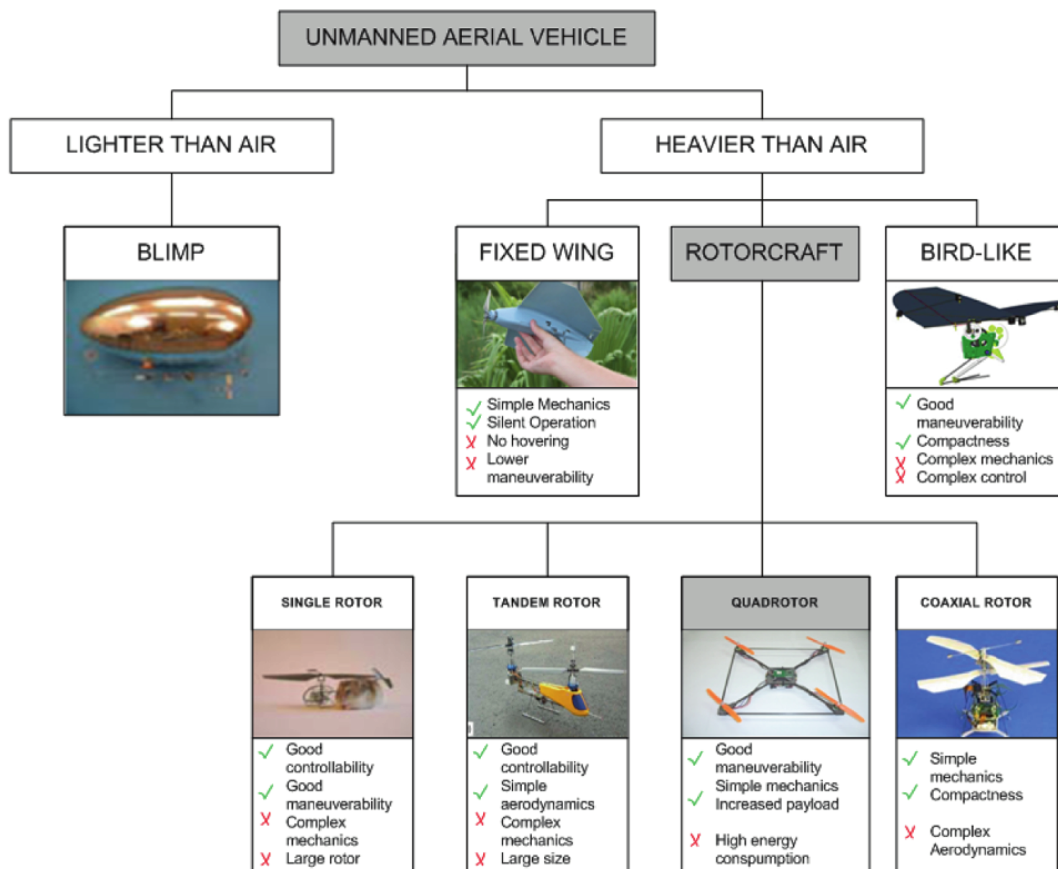


Figura 1.5: Tipos de UAVs [9]

1.4.1. Cuadricópteros

Desde hace algunos años, un nuevo tipo de UAV ha empezado a atraer mucho la atención, logrando un bien merecido reconocimiento, se trata del cuadricóptero. Hasta ese momento, los UAVs eran pesados y no lo suficientemente ágiles como deberían ser (helicópteros [11] y [12]).

Un cuadricóptero tiene una estabilidad realmente buena en el aire (asumiendo unos buenos sensores a bordo). Puede mantenerse en el aire como un helicóptero, pero también puede hacer afilados giros o volar boca abajo sin problema alguno. Además suele ser más ligero que helicópteros y aviones.

Un cuadricóptero, también llamado quadrirotor o quadrotor, es un helicóptero que es elevado y propulsado por cuatro rotores. Como se puede apreciar en la Figura 1.6, es pequeño, entre 10 cm y 150 cm. En tamaños más grandes, la autonomía crece, ya que cuanto mayores son las hélices se consigue sustentación con una rotación más lenta y por lo tanto con una mayor eficiencia. Por otro lado, cuanto menor es el tamaño, mayor la agilidad.

Figura 1.6: Cuadricóptero

1.5. Contenido de la memoria

En este Capítulo 1 de la memoria se han resumido la motivación y los objetivos y se detalla el trabajo relacionado con vehículos aéreos no tripulados (UAVs) (Sección 1.4) y más concretamente con cuadricópteros (Sección 1.4.1).

En el Capítulo 2 se habla de la estructura del cuadricóptero (Sección 2.1) comentando los distintos procesos de fabricación y del hardware que monta la aeronave (Sección 2.2). También se hace un repaso por las plataformas de seguridad empleadas (Sección 2.3).

En el Capítulo 3 se repasa el software desarrollado. Enmarcado en este capítulo es donde se desarrollan la mayor parte de la documentación y las tareas de aprendizaje de uso de las plataformas (Sección 3.1). Posteriormente, el informe entra de lleno en la arquitectura de código (Sección 3.2), donde se repasan los módulos desarrollados de interfaz, simulación y control de vuelo. La Sección 3.3 presenta el control utilizado en las secciones anteriores.

En el Capítulo 4 se exponen las conclusiones y las posibles vías de trabajo futuro.

También se incluye el Apéndice A con los módulos de software desarrollados; el Apéndice B con los planos de la impresión 3D, fotografías de la impresora y del cuadricóptero montado; el Apéndice C con el diagrama de Gantt del proyecto; y el Apéndice D con la memoria original en inglés.

CAPÍTULO 2

Diseño y construcción

2.1. Estructura y fijaciones

El diseño y la estructura de un cuadricóptero es una parte fundamental para el buen funcionamiento del mismo. A pesar de ello, la realimentación de los motores (se utilizan los sensores para reajustar la velocidad de cada rotor) hace que la aeronave pueda volar en cualquier configuración e incluso después de perder alguno de los propulsores [13]. Pero esto no quita para que una buena estructura beneficie, y mucho, al correcto funcionamiento del mismo.

(a) Brazo de aluminio

(b) Brazo de plástico ABS

Figura 2.1: Elementos estructurales usados en el cuadricóptero

Es necesaria una estructura rígida para prevenir la rotura ante un posible accidente, pero también para conseguir las mínimas vibraciones, tanto en funcionamiento, como en el momento del despegue. No sólo la entrada en resonancia del cuadricóptero podría llegar a ser crítica,

sino también una ligera vibración, ya que en el cuadricóptero va montada una serie de sensores y procesadores, y estas vibraciones pueden inducirlos a mediciones incorrectas o incluso a la larga, a roturas.

El peso también es un atributo a tener en cuenta, ya que la autonomía y la agilidad se consideran dos de las características más importantes del helicóptero. Un peso bajo en el marco va a suponer un menor consumo y unas menores inercias. De este modo, en un primer momento se consideró el uso de un marco de aluminio (Figura 2.1a). Era ligero y, al no ser muy flexible, casi no transmitía las vibraciones desde los motores al núcleo central. Pero era ligero por ser una viga tubular con un espesor mínimo. Esto significa que la rigidez del mismo era también mínima.

Al apreciar problemas ante las primeras caídas del cuadricóptero, que aunque desde poca altura, conseguían doblar las barras, se tomó la decisión de reemplazar esta estructura por cuatro brazos en celosía impresos en ABS[1] con una impresora 3D (Figura 2.1b), cuyo proceso de construcción se detalla a continuación.

2.1.1. Impresión 3D

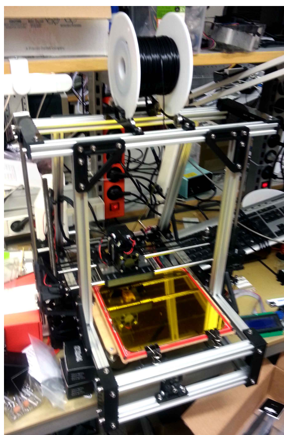


Figura 2.2: Impresora 3D usada en el proyecto

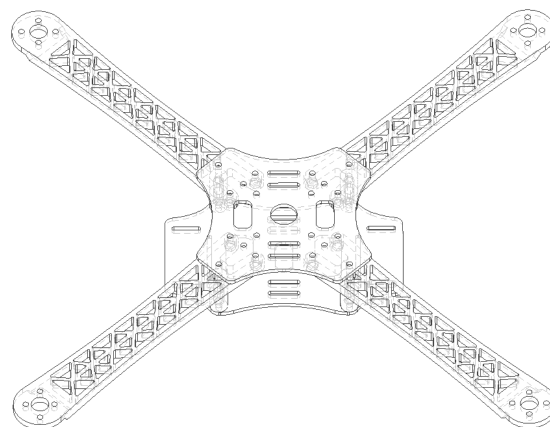


Figura 2.3: Montaje del cuadricóptero para el prototipo impreso

Una impresora 3D (en la Figura 2.2 se muestra la utilizada, a tamaño completo en el Apéndice B.1) es una máquina capaz de realizar "impresiones" de diseños en 3D, creando piezas o maquetas volumétricas a partir de un diseño hecho por ordenador. Surgen con la idea de convertir archivos CAD en prototipos reales.

La impresión 3D está abriendo la puerta a un gigantesco mundo de posibilidades. Se pueden

[1] Acrilonitrilo Butadieno Estireno, es un plástico muy resistente al impacto (golpes) muy utilizado en automoción y otros usos tanto industriales como domésticos. Es un termoplástico amorfo.

conseguir estructuras complejas, resistentes, ligeras y flexibles, con un presupuesto contenido.

Para nuestro diseño, se ha optado por una celosía plana (como se puede ver en los brazos de la Figura 2.3) impresa en ABS. La flexibilidad de esta construcción junto con su resistencia la hacen perfecta para esta tarea.

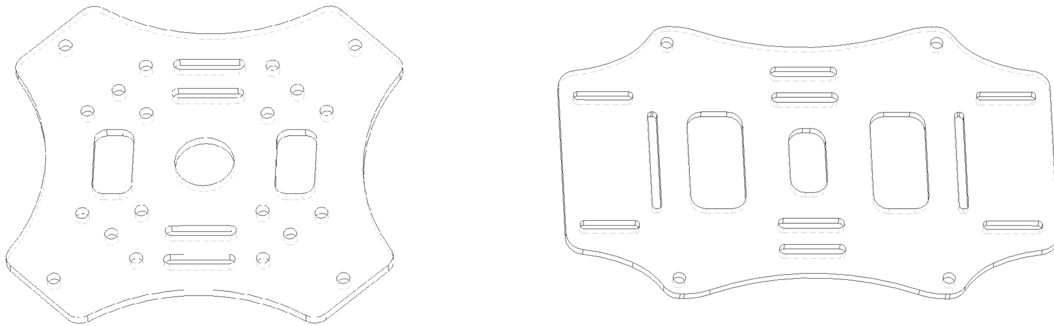


Figura 2.4: Plano de los platos de soporte usados en prototipo impreso

De la misma manera se han impreso los platos de soporte (Figura 2.4) que unirán la estructura y servirán de apoyo para la electrónica de la aeronave. Aprovechando las posibilidades brindadas por la impresión 3D se han dejado las plataformas preparadas para distintos soportes de hardware, realizando agujeros que puedan servir para otras plantillas. De esta forma podremos adaptar la estructura a diferentes configuraciones, a la vez que reduciremos el peso. Más detalles de las piezas impresas y la impresora 3D utilizada en el Apéndice B.

2.2. Hardware

Una vez descrita la estructura de base del cuadricoptero, esta sección describe el hardware y componentes electrónicos del prototipo.

2.2.1. Arducopter

Nuestro diseño (presentado en la Figura 2.5 y en el Apéndice B.3) parte de la plataforma ArduCopter. ArduCopter es una plataforma comercial que sirve de base para construir helicópteros multirotor y otros muchos robots.

ArduCopter es basicamente una placa de Arduino[2] [14] modificada (ArduPilot Mega, en adelante APM) unida a una placa de sensores. Gracias a Arduino y a la gran comunidad que ambos tienen es posible encontrar una gran cantidad de modificaciones para la plataforma Ar-

[2]Arduino es una plataforma de hardware libre, basada en una placa con un microcontrolador y un entorno de desarrollo, diseñada para facilitar el uso de la electrónica en proyectos multidisciplinares.

Figura 2.5: Cuadricóptero de fabricación propia

duCopter con las cuales puede ser facilmente adaptado a aplicaciones específicas. Este sistema contiene una IMU (unidad de sensores inerciales), que proporciona magnitudes de medida para tres direcciones tanto en el caso del acelerómetro como en el del giróscopo, también incluye un magnetómetro de tres grados de libertad y una unidad de XBee [15] para comunicaciones inalámbricas. Además, esta placa proporciona un sistema de cuaternios [16] con el cual se puede calcular el sistema de ángulos de Euler [17] y medir cómo el cuadricóptero está posicionado. Un GPS conectado a la placa del APM se usa para estimar la posición tridimensional del aparato. Por último, hay una placa de distribución de potencia controlada por el APM, esta placa controla los ESC (*Electronic Speed Controller*, o en español, Controlador Electrónico de Velocidad) de los motores. En la Figura 2.6 se muestra un esquema de las partes.

El kit ArduCopter también puede incluir otro componente muy importante del cuadricoptero, las hélices. Es un componente clave, ya que de sus características y de cómo están montadas va a depender que pueda lograr la sustentación requerida para volar. La estabilización en vuelo en este tipo de máquinas voladoras se logra gracias a dos fuerzas opuestas [18]. La fuerza que empuja el UAV hacia abajo es la gravedad, por ello, se necesita crear una fuerza opuesta de la misma magnitud para conseguir mantener el vehículo en el aire. Esta fuerza es generada por las hélices, que crean un área de baja presión sobre las superficies superiores de las mismas.

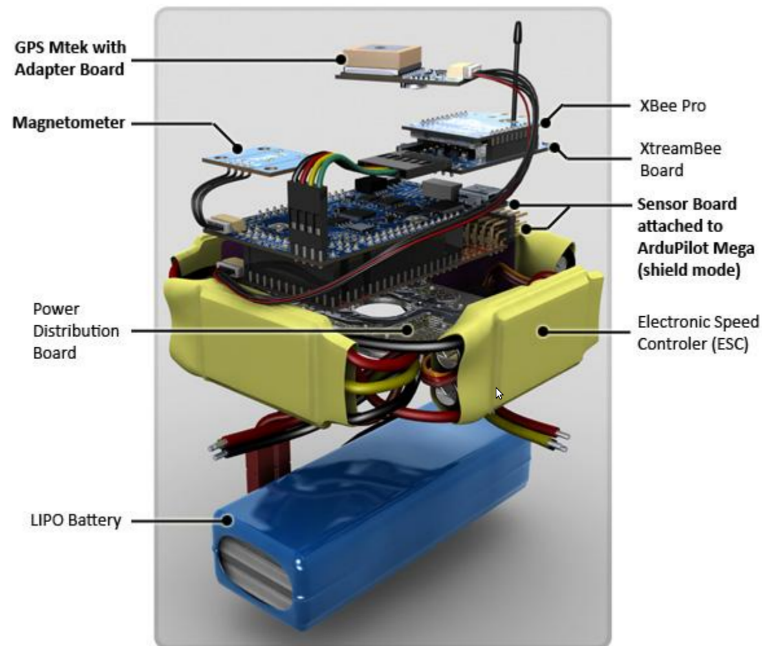


Figura 2.6: Componentes del kit ArduCopter

Estas hélices son capaces de cambiar los ángulos del cuadricóptero. En la Figura 2.7 (a), en negro, se presenta la estructura del cuadricóptero. Los ejes del cuerpo rígido se muestran en verde, mientras que en azul se representa la velocidad angular de las hélices.

La posición relativa del cuadricóptero es representada mediante los giros Roll-Pitch-Yaw:

- El giro Roll es proporcionado por el incremento (o decremento) en la velocidad de la hélice izquierda y por el decremento (o incremento) en la derecha. Esto genera un momento con respecto al eje X_B que hace girar al cuadricóptero. La Figura 2.7 (b) muestra el giro Roll en el esquema del cuadricóptero.

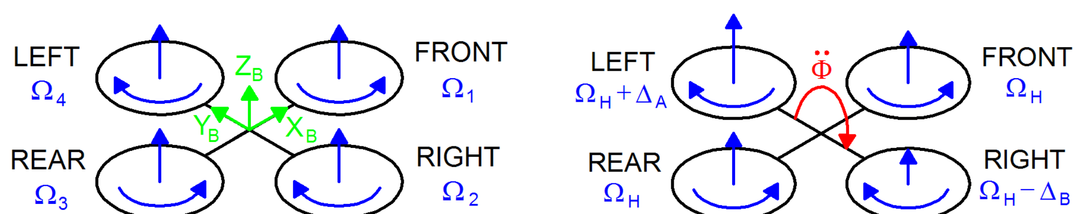


Figura 2.7: a) Esquema de motores simplificado de un cuadricóptero en posición de estabilidad y b) Movimiento Roll, giro en torno al eje X

- El Pitch es muy similar al Roll y es proporcionado por el incremento (o decremento) en la velocidad de la hélice trasera y por el decremento (o incremento) en la delantera. En este caso, se genera un momento con respecto al eje Y_B que de nuevo hace girar el cuadricóptero. La Figura 2.8 (a) muestra el giro de Pitch en el esquema.

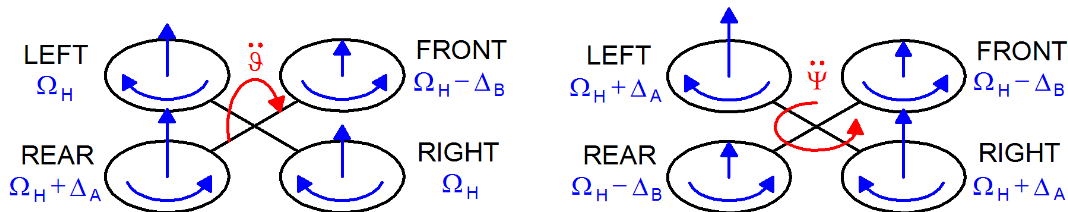


Figura 2.8: a) Movimiento Pitch, o giro en torno al eje Y y b) Movimiento Yaw, o rotación en torno al eje Z

- El Yaw es creado por el incremento (o decremento) de las velocidades de las hélices delantera y trasera y por el decremento (o incremento) de las velocidades en las hélices laterales. Esto crea un momento con respecto al eje Z_B que hace al cuadricóptero rotar. El movimiento de Yaw es generado gracias al hecho de que las hélices laterales giran en sentido horario, mientras que la delantera y la trasera lo hacen en antihorario. De este modo, cuando el momento del conjunto está desequilibrado, el helicóptero gira en torno a Z_B. La Figura 2.8 (b) muestra un esquema con el movimiento de Yaw. [19]

Gracias a todo esto, un cuadricóptero se convierte en una aeronave estable que, por lo tanto, puede ser usada en interiores, pero puede también ser una máquina peligrosa, ya que es rápida y capaz de girar y cambiar de dirección en poco tiempo. En consecuencia, perder el control del cuadricóptero puede suponer su destrucción o la de su entorno, e incluso ocasionar daños personales, por lo cual es necesario utilizar plataformas de seguridad como las descritas en la Sección 2.3.

Conexión remota con ArduCopter

En este proyecto se usará un cuadricóptero estándar ArduCopter que será conectado con una estación de tierra mediante un XBee [15].

La conexión de serie MAVlink[3] [21] proveerá datos sobre *attitude* (pose - orientación y velocidad en los ángulos de Euler), *vfr_hub* (velocidad del suelo y del aire, altitud y tasa de subida), GPS y RC (el valor de los 8 canales de frecuencia de radio). Esta conexión de serie también proporcionará *heartbeat* (latido del corazón - los sistemas pueden usar este mensaje

[3] MAVlink es un ligero protocolo de comunicación mediante mensajes con cabecera ampliamente usado en ROS.

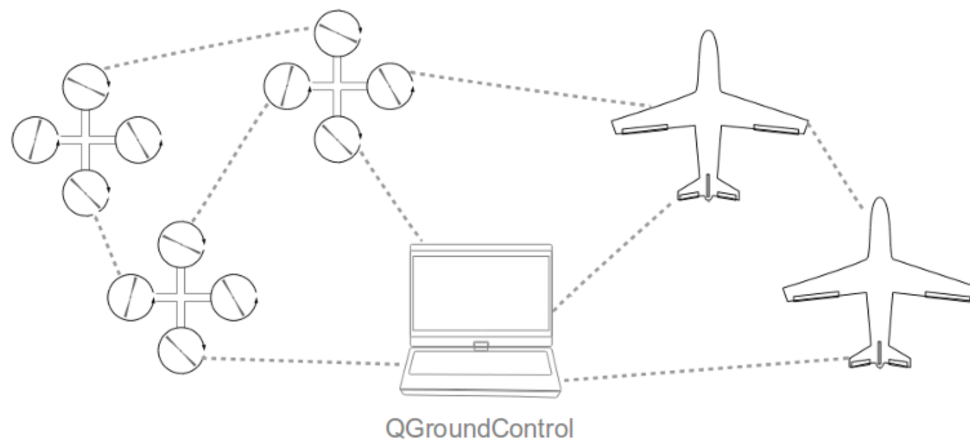


Figura 2.9: Conexión MAVlink entre máquinas voladoras y una estación de tierra. Las lineas de puntos representan conexiones p2p MAVlink [20]

para monitorizar si el sistema está operativo); *navigation controller output* (registro del control de navegación - con información sobre objetivos y errores); *hardware status* (estatus del hardware - voltaje de la placa); *raw IMU* (valores sin procesar de la IMU - valores de aceleración, giróscopo y brújula); *pressure* (presión); y algunos parámetros más de los que no se dará uso en este sistema, mientras que esta misma conexión será usada para enviar la velocidad y los ángulos de Euler requeridos.

En la Figura 2.9 se muestra una configuración de red típica entre varios robots aéreos y una estación de tierra usando dispositivos XBee.

2.2.2. Sensores

Los datos enviados por el protocolo MAVlink son generados por una serie de sensores de a bordo, mencionados anteriormente. Esta sección detalla los sensores utilizados. Una IMU (cuyo funcionamiento, por ser parte básica del kit ArduCopter, se ha detallado en parte en la Sección 2.2.1 y se acabará de ahondar en ello en la Sección 3.2.1); el módulo de GPS de MTek ((a) en la Figura 2.10) que nos permine controlar el cuadricóptero en exteriores con una precisión de unos 3 metros y con una buena estabilidad; un barómetro, que usa la presión atmosférica para conocer la altitud; y un módulo de ultrasonidos LV-EZ0 que permite afinar hasta pocos centímetros la posición relativa con el suelo. Tanto el módulo de GPS y de ultrasonidos como el barómetro son elementros recomendados para ArduCopter, sin ser piezas estándar.

Para envíos y recepción de datos se utiliza un adaptador de conexión de serie *XBee Explorer Regulated* ((e) en la Figura 2.10) conectado a través del puerto UART0 del APM. A éste se le conectará un módulo *XBee Pro 60mW Wire Antenna - Series 1 (802.15.4)* de Digi International ((c) en la Figura 2.10), el cual hace uso de la tecnología ZigBee para mantener una buena

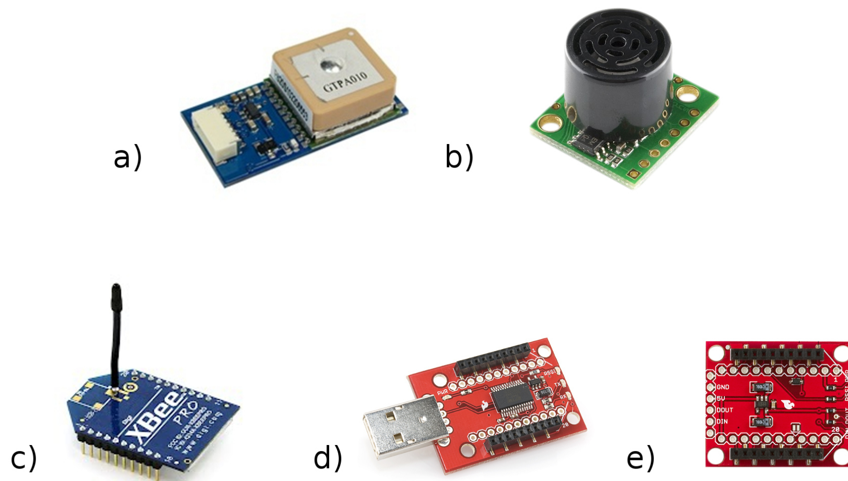


Figura 2.10: Sensores utilizados en la construcción del cuadricóptero

comunicación con la estación de tierra manteniendo bajo el consumo. Este dispositivo permite también rangos de transmisión de incluso más de 1 kilómetro en campo abierto. En el caso de usarse un ordenador, el adaptador a usar será el *XBee Explorer Dongle*, (d) en la Figura 2.10.

También está disponible un control remoto Futaba para ser usado en casos de emergencia. Si algo sucede y el cuadricóptero pierde el control, el operador es capaz de cambiar a modo de control manual únicamente pulsando un interruptor en el mando. Usando otro modo, con este mismo interruptor, también es posible llevar a cabo un aterrizaje de emergencia (el módulo de código está disponible en el Apéndice 1.4.2). Registrando la altura específica desde la que comienza la maniobra, el cuadricóptero puede alcanzar la altura cero, reduciendo progresivamente la potencia en los motores y usando alturas intermedias para estabilizarse.

Se puede encontrar más información sobre los sensores utilizados en el Apéndice D en Section 3.1.

2.3. Plataformas de seguridad

La posibilidad de hacer uso de esta aeronave en interiores junto con su capacidad para cambiar de dirección rápidamente, su agilidad y su velocidad, la convierten (ante un posible fallo) en una máquina peligrosa. En consecuencia, perder el control del cuadricóptero puede llevarlo a destruirse a sí mismo o a su entorno, así como a dañar a alguien.

En nuestro laboratorio, y para ensayar el módulo de código en un entorno seguro, se ha construido una plataforma de seguridad consistente en un cable vertical con amortiguadores (en Figura 2.11). El cuadricóptero está anclado a él y puede rotar y elevarse. Hay también una red de seguridad entre el cuadricóptero y los operadores.

Para las pruebas en exteriores (Figura 2.12) se ha considerado suficiente el uso de un cable

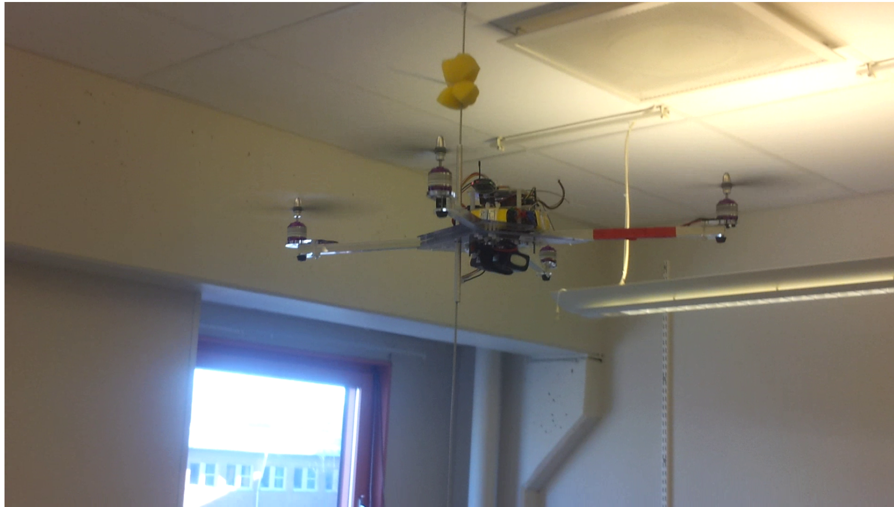


Figura 2.11: Laboratorio de la LTU con un sistema de seguridad para cuadricópteros

de 10 metros al que se ha fijado el cuadricóptero en los momentos de ensayo. Este cable está anclado a una fijación en un patio interior al que se veta la entrada durante los ensayos. El cable cumple el cometido de impedir que el helicóptero salga de la zona de seguridad o impacte contra paredes o personas.

Figura 2.12: Área de pruebas exteriores para cuadricópteros de la LTU

CAPÍTULO 3

Programación y Desarrollo

3.1. ROS: Sistema Operativo para Robots

Desde la aparición de Internet, las comunidades de desarrolladores se han convertido en una parte fundamental para la investigación. Ahora, no es necesario ser un experto para poder entrar en campos hasta hace poco restringidos a áreas profesionales. Hoy en día es posible aprender e investigar en cualquier campo imaginable. Instructables[1] o incluso YouTube[2] hacen más fácil esta labor. Pero si nos alejamos un poco de los servicios generalistas, hay un mundo lleno de posibilidades.

En particular en el ámbito de la robótica, el entorno de código libre ROS facilita el adentrarse en este mundo. ROS no es sólo un sistema operativo, es un ecosistema completo, nutrido de código y herramientas que nos ayudan a dar vida a nuestras creaciones. Pero sobre todo, es una comunidad de desarrollo. Un sistema como ROS ayuda a la difusión de ideas y al uso libre de conocimientos para conseguir un avance más rapido del mundo tecnológico.

3.1.1. Definición

Robot Operating System (Sistema Operativo para Robots), o ROS, como es llamado de forma más coloquial, es un pseudo sistema operativo de código libre (distribuido bajo los términos de la licencia BSD[3]) para el desarrollo de software para robots. Proporciona librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones para robots. También proporciona abstracción de hardware, controladores de dispositivos, visualizadores, interfaz de intercambio de mensajes, gestor de paquetes y más.

[1] Instructables es una página web especializada en la creación y publicación de proyectos “hazlo-tu-mismo”. http://www.instructables.com/

[2] YouTube es un sitio web en el cual los usuarios pueden subir y compartir vídeos. http://www.youtube.com/

[3] La licencia BSD es la licencia de software otorgada principalmente para los sistemas BSD (Berkeley Software Distribution). Es una licencia de software libre permisiva. Esta licencia tiene pocas restricciones, estando muy cercana al dominio público, permite el uso del código fuente en software no libre.

"Un sistema construido usando ROS consiste en un número de procesos, potencialmente en diferentes máquinas, conectado en tiempo real en una red punto a punto." [22]

Además, este sistema se ha desarrollado específicamente para poder reutilizar controladores y código de otros robots. Esto se consigue encapsulando el código en librerías independientes que no tienen dependencias en ROS.

En algunos casos, ROS puede ser usado únicamente para mostrar opciones de configuración o para enrutar datos hacia y desde el software respectivo, con tan poco parcheado como sea posible.

En última instancia, la razón más importante para usar ROS, y no otro sistema, es que es parte de un entorno colaborativo. Debido a la gran cantidad de robots y software de inteligencia artificial que se está desarrollando, la colaboración entre investigadores y universidades es esencial. Para dar soporte a esos desarrollos, ROS proporciona un sistema de paquetes. Un paquete de ROS es un directorio que contiene un archivo XML que describe el paquete y sus dependencias.

En el momento de escribir esto, varios miles de paquetes ROS existen como código libre en Internet, y más de mil quinientos están directamente publicados en `http://www.ros.org/browse`.

3.1.2. Uso de la plataforma y primeros ensayos

Inicialización del Sistema

Para configurar ROS es necesario comenzar realizando la conexión entre máquinas. Para ello, *master* y *hostname* deben ser establecidos a fin de que cada una de las máquinas conozca su papel. Para que sea posible hacer uso de ROS, debe ser lanzado un *roscore*. Este comando iniciará un ROS *master*, que dirigirá los links entre los distintos hilos de ejecución. Estos son editores (*publishers*) y subscriptores (*subscribers*) a temas (*topics*) y servicios (*services*). Una vez que estos nodos se localizan unos a otros, empiezan a comunicarse mutuamente mediante una conexión punto a punto. Por otra parte, también se usará un servidor de parámetros (*parameter server*), el cual será un espacio compartido para almacenar y recuperar parámetros en tiempo de ejecución, y un nodo de registro (*rosout*). En casos donde el programa ROS se ejecute bajo varias máquinas, se deberá iniciar un *roscore* únicamente en una de ellas, en la máquina principal (*master*). Una vez que el sistema está en funcionamiento, es posible lanzar los distintos tipos de hilos de ejecución que sean necesarios.

Comunicaciones

Para establecer la comunicación entre las dos máquinas (la estación de trabajo y el robot) e integrar todas las máquinas en una misma red ROS se usará el programa ***slattach*** (en linea de comandos), que permite crear un punto de red a partir de un puerto serie, haciendo uso (en este caso) de un XBee.

Una vez creados los puntos de red en las dos máquinas se puede usar el comando *sl0* del

programa *ifconfig* para crear la red. Las IPs usadas en este comando únicamente deberán ser IPs no existentes, no hay ninguna otra restriccion, ya que se están creando justo ahora.

Para el correcto funcionamiento del sistema, ambos XBee tienen que funcionar a la misma velocidad de transmisión (en este caso *57600 Bd*), y deberán estar configurados en modo p2p correctamente.

Todos estos procesos pueden ser ajustados automáticamente si alguno de los ordenadores no dispone de pantalla escribiendo un determinado código en */etc/network/interfaces*. Entonces, la interfaz de red será iniciada cada vez que se encienda la máquina. Cuando la red está correctamente configurada es posible usar el comando ***ssh*** para usar el terminal de la otra máquina.

Una vez esté toda la red en Linux configurada, se procederá a la configuración de la red en ROS, para lo que tan solo habrá que registrar el *master* y el *hostname* en cada máquina en sus respectivos ficheros *.bashrc*. Entonces ya se puede abrir una instancia de *roscore* en el *master*, iniciar los servicios que se requieran e iniciar los programas.

Para más información se puede consultar el Apéndice D en Section 2.2, allí se incluyen los códigos arriba mencionados, así como alguna aclaración a los procesos.

Figura 3.1: TurtleBot v1.5

Ensayos con Turtlebot

El primer ensayo de aprendizaje con ROS se ha realizado con un robot **TurtleBot** (Figura 3.1), éste es un robot terrestre, basado en la aspiradora Roomba, que gracias a una cámara Kinect[4] puede identificar el entorno e interactuar con él. Este robot monta un portátil con ROS en él. También se ha usado otro portátil Ubuntu como estación de trabajo. El robot es usado para comprender cómo funciona el sistema de nodos ROS y empezar a trabajar con un robot de tierra, siempre más simple en tareas de aprendizaje.

Con esta máquina se han experimentado paquetes de navegación, SLAM y comunicaciones. Se ha aprendido a enviar mensajes al programa para modificar variables en tiempo real (con el comando *rostopic pub*), a iniciar servicios (con el comando *rosservice call*) y a leer todo tipo de parámetros del sistema durante la ejecución (con *rosrun rqt_graph rqt_graph*, que muestra un diagrama de la distribución de los nodos, o con las diversas variaciones de *rostopic*) y, en general, a tener una mayor conciencia del entorno.

El primer experimento ha sido llevado a cabo evaluando un código de SLAM, una versión modificada de *OpenSlam's Gmapping* (que usa el algoritmo de filtro de partículas Rao-Blackwellised) [23] [24]. Con este algoritmo y este robot se pueden conseguir buenos resultados como puede apreciarse en la Figura 3.2, aunque esto requiere de pocos cambios de trayectoria. Cuando la ruta descrita por el robot se complica, el resultado empeora debido a problemas relacionados con la odometría.

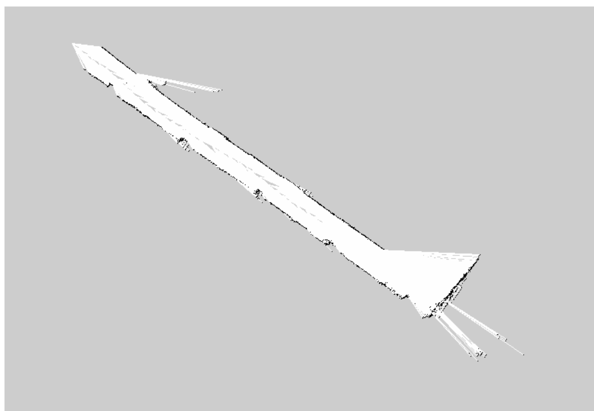


Figura 3.2: Mapa de un pasillo generado por TurtleBot

Para más información se puede consultar el Apéndice D en Section 2.2.

[4] Kinect es un dispositivo de Microsoft (originalmente para XBox 360) que integra dos sistemas de visión: una cámara RGB y un sistema basado en patrón de luz infrarroja para identificar la profundidad de campo.

Ensayos con Hector

Hector es una colección de ROS stacks (un stack - o pila - es una colección de paquetes que provee de una funcionalidad al sistema) originalmente desarrollada por la Technische Universität Darmstadt (Alemania). Estas pilas proporcionan varias herramientas para simular o interactuar con robots. SLAM, localización o modelado son algunas de ellas.

De esta colección se hará uso de la pila hector_quadrotor, que será capaz de modelar (usando parámetros físicos) un modelo de cuadricóptero en 3D (mostrado en la Figura 3.3) en Gazebo, donde habrá una reconstrucción tridimensional del mundo sobre la cual se aplicará una técnica de SLAM.

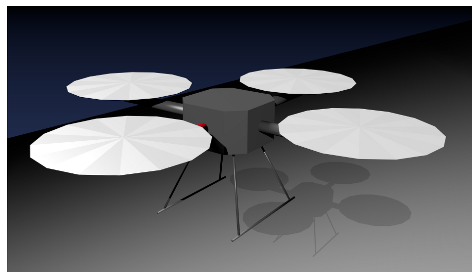


Figura 3.3: Representación de un cuadricóptero Hector

Hector aprovecha las posibilidades de visualización de ROS tanto con Gazebo (simulación y visualización) como con Rviz (visualización). Gazebo es un simulador multi robot. Al igual que Stage [25], es capaz de simular una población de robots, sensores y objetos, pero en un espacio tridimensional. Generan ambos una respuesta realista de los sensores y una interacción física plausible entre objetos (incluida una simulación precisa de la física de cuerpos rígidos) [26]. Mientras que Rviz es un entorno de visualización 3D que es parte de la pila de visualización de ROS [27].

En una primera etapa, y a través de la plataforma, se ha desarrollado un simulador para ensayar los controladores diseñados y para lograr un mejor entendimiento del comportamiento de los cuadricópteros.

Con este fin, se ha diseñado un código de control para llevar a cabo una navegación mediante puntos vía. Con este código es posible seleccionar una serie de objetivos (tantos como sean necesarios), la precisión y la velocidad. También se pueden editar las constantes proporcionales, integrativas y derivativas. Por el momento, todos estos parámetros necesitan ser ajustados en el fichero de código (*cpp*) y, posteriormente, es necesario compilar con un *rosmake*. Aprovechando las características de ROS, y a la hora de ejecutar la aplicación, también se podría añadir al final de la orden *rosrun* el remapeo de alguna de las variables, siempre y cuando se guardaran dentro del código las otras posibilidades a utilizar. Por ejemplo *rosrun hector_quadrotor_controller simulation tolerance:=tolerance2*, con lo que pasaríamos de usar

la variable *tolerance* a usar la variable *tolerance2*.

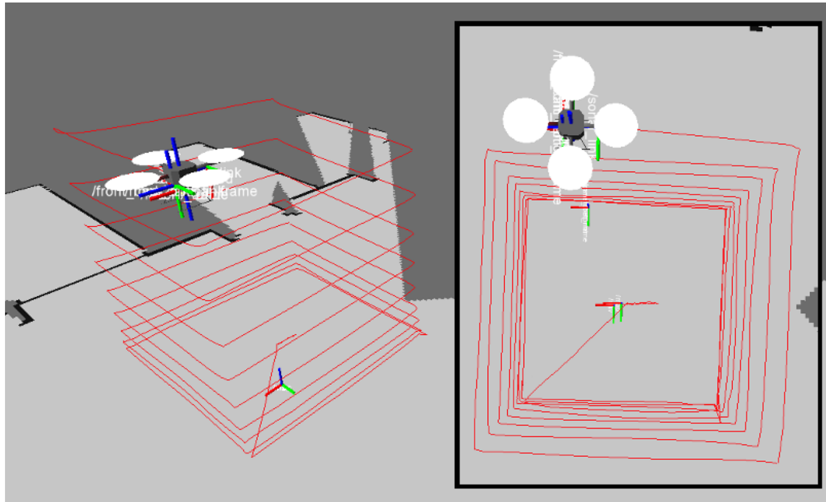


Figura 3.4: Simulación de un cuadricóptero Hector

Este controlador puede mover el cuadricóptero en las direcciónes X, Y y Z. También puede ser controlada la orientación para mantenerse siempre estable y dirección Norte. El código completo de esta simulación se ha incluido en el Apéndice 1.3.1. En la Figura 3.4, se muestra el cuadricóptero simulado siguiendo un casi perfecto cubo (la línea roja representa la ruta seguida).

En el siguiente ensayo de simulación se ha probado a controlar el movimiento con un cuadricóptero real para analizar una verdadera interoperabilidad entre plataformas y redes. Este simulador toma la lectura de los ángulos de Euler directamente desde la plataforma experimental del cuadricóptero para mostrar ese mismo movimiento en el cuadricóptero virtual. El eje Z puede ser rotado para rotar en la simulación, el eje X para avanzar en dirección Y y el eje Y para avanzar en dirección X, tal y como sería el movimiento si los motores del cuadricóptero lo hicieran inclinarse en Roll, Pitch o Yaw. Para lograr esto se ha necesitado usar un controlador para mover el cuadricóptero virtual desde la posición original (la virtual) hasta la posición objetivo (la real) de forma similar al código anterior.

Para estos ensayos, se ha utilizado el marco experimental (Figura 3.5) y una IMU, el KFly, ambos diseñados por Emil Fresk. Esta estructura ha sido impresa en las instalaciones de la LTU con una impresora 3D hecha a mano, y ha sido ensayada su resistencia a axial y cortante.

El código completo, en este caso, se incluye en el Apéndice 1.3.2.

Se puede encontrar más información sobre Hector en el Apéndice D en Section 2.3.

Figura 3.5: Prototipo de la estructura del cuadricóptero

3.2. Arquitectura del sistema

En el diagrama de bloques de la Figura 3.6 se representa una visión general de la arquitectura del sistema de desarrollo. En el primer módulo, en rojo (y a la izquierda), se muestra esquematizada la organización de ROS. En este módulo, las elipses son programas (o nodos) y los rectángulos son temas (o topics). Los programas utilizan la información proporcionada por los temas para llevar a cabo su tarea y vuelven a volcar ahí sus resultados. El programa de interfaz */RosCopter* sirve de traductor y puente entre esos temas y la información enviada desde Arduino.

En el módulo central, en amarillo, se representa el envío de información a través de XBee. Los mensajes son enviados mediante un protocolo estándar llamado *MAVlink*.

En el tercer módulo, en verde (y a la derecha), se sitúa el sistema ArduCopter. Tiene un módulo APM, que es el dispositivo inteligente de a bordo. Puede procesar tareas simples de control, lee los sensores de medida y controla el movimiento de los motores. Recibe información de los registros del módulo de GPS y de la unidad de medida inercial (IMU). La IMU lee el magnetómetro y el sónar externos. Por último, el sistema ArduCopter incluye una placa de distribución controlada por el APM que manda las señales de control a los ESC.

3.2.1. IMU: Unidad de medición inercial

Una unidad de medición inercial, o IMU, es un sistema para la medida del movimiento de un objeto en espacio libre relativo a un marco inercial. Los sensores usados para la medida inercial son fundamentalmente acelerómetros y giróscopos.

Un acelerómetro triaxial puede medir tres grados de libertad. Si puede asumirse que un objeto

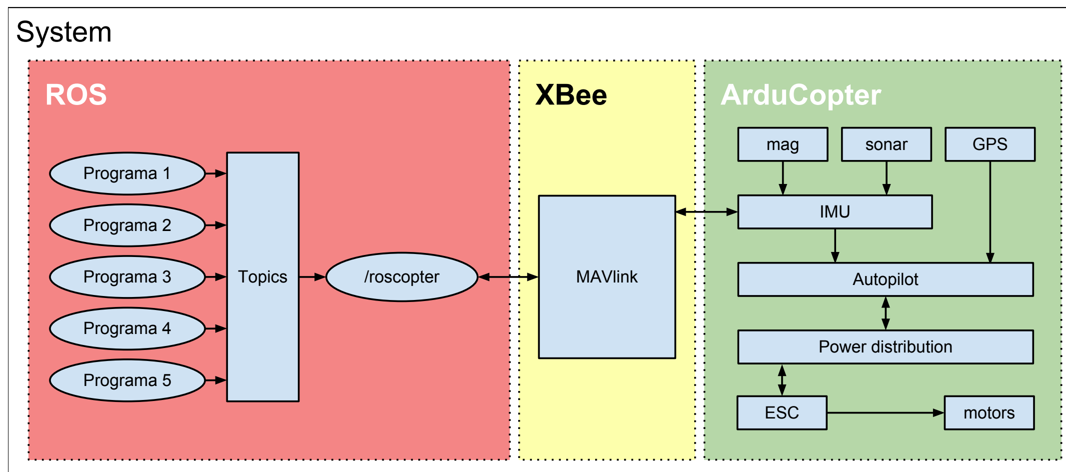


Figura 3.6: Diagrama de Nodos

únicamente tiene o translaciones o rotaciones, entonces un acelerómetro triaxial es suficiente para medirlas. En la tierra, siempre hay presente una fuerza gravitacional de 1G. Asumiendo rotación sin translación, un acelerómetro triaxial puede ser usado para medir la inclinación utilizando la orientación de este vector gravitatorio de 1G. Si los datos del sensor muestran una aceleración constante de 1G en una orientación fija por un tiempo determinado, entonces uno puede adivinar que el sujeto está en una postura estática.

Para distinguir entre rotación y translación, con frecuencia se hace uso de giróscopos. Un giróscopo tradicional consiste en un rotor con giro libre para mantener el momento angular y por lo tanto su orientación original. Hoy en día, giróscopos de estructura vibrante reemplazan los rotores por masas de prueba que vibran. Éstas también tienen el efecto de mantener la orientación original pero pueden fabricarse en tamaños más pequeños y, por consiguiente, pueden ser más económicos. La desviación de la orientación original puede ser medida para obtener la rotación en términos de ángulo y velocidad angular. Después de cancelar la proyección de la rotación en tres ejes, la aceleración lineal remanente puede ser interpretada como translación. Sin embargo, los giróscopos también tienen sus problemas. Cada giróscopo está limitado por la velocidad angular máxima que puede tolerar antes de que la orientación original del rotor o de la estructura vibrante cambien. Otro problema es el consumo relativamente alto comparado con los acelerómetros. [28]

La IMU que se va a utilizar, como se ha comentado anteriormente, ha sido diseñada y ensamblada en los laboratorios de la Universidad Técnica de Luleå por E. Fresk [1] (Figura 3.7). En dicha IMU, a estos dos sensores se les añade un magnetómetro para registrar las direcciones y magnitudes del campo magnético en los tres ejes. Un magnetómetro es un instrumento de medida usado para determinar la fuerza en una dirección concreta del campo magnético. Si este sensor es puesto en una localización donde las propiedades del campo magnético de la tierra son conocidas, puede determinarse la orientación del vehículo. Sin embargo, no se puede ob-

tener una completa estimación de la postura, ya que no se pueden detectar las rotaciones sobre el vector del campo magnético. Para solucionar esto, se puede hacer uso de una estimación indirecta de la medida con los acelerómetros de la IMU.

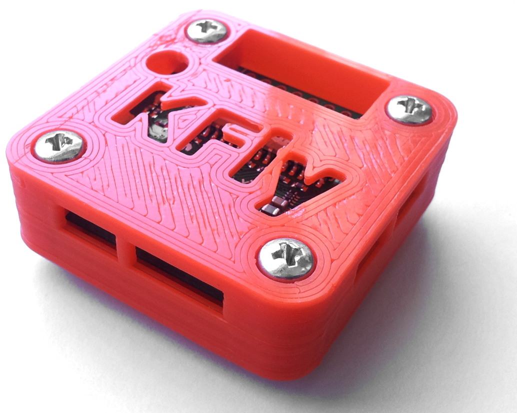


Figura 3.7: Módulo de medición inercial

Parser (Analizador sintáctico)

Para interactuar con este dispositivo, se ha desarrollado un analizador sintáctico que, leyendo el puerto de serie al que está conectado el dispositivo, puede extraer e interpretar las medidas de los sensores.

El código, después de inicializar las variables, puertos y parámetros adecuados, comienza una lectura inicial del puerto de serie para sincronizar la lectura con la escritura. Lee lo que hay almacenado en ese puerto (vaciándolo), proceso que se repite hasta conseguir una cadena completa. Sabremos que esta cadena es completa porque el byte dos del vector leído nos indicará la longitud y, a su vez, sabremos que la cadena empieza en el lugar adecuado por que las cadenas siembre empiezan con el valor hexadecimal *0xa6*. Una vez que se ha hecho la sincronización, se entra en un bucle infinito (que no se corta a no ser que ROS se detenga). De la cadena de datos se pueden sacar los valores detallados en la siguiente Tabla 3.1.

Nombre	SYNC	CMD	SIZE	CRC8	DATA	CRC-16-CCITT	SYNC	CMD	SIZE	CRC8
Tamaño	1 byte	1 byte	1 byte	1 byte	0 - 255 bytes	2 byte	1 byte	1 byte	1 byte	1 byte
Valor	0xa6	0x19	0x22	0x31	-	-	-	-	-	-

Tabla 3.1: Estructura del paquete de datos de la unidad de medida KFly

En esta tabla el bloque *DATA* corresponde a 4 bytes para cada una de las cuatro componentes de cuaternio[5] (q_i) y 2 bytes para cada una de las componentes de la aceleración lineal (a_i), la angular (w_i) y el campo magnético (m_i).

Este programa también transforma los valores del cuaternio en los ángulos de Euler (Roll, Pitch y Yaw), aun cuando estos valores no se pasarán al mensaje de ROS. Con eso ya se puede crear el mensaje para ROS y almacenar en él todos los parámetros considerados necesarios. El código completo se encuentra en el Apéndice A.1.

Conclusión

Pese a haber conseguido traducir estos datos a variables sensibles de ser usadas, se ha considerado desestimar esta vía de trabajo para la construcción del primer prototipo por varios motivos.

En primer lugar, al no ser el protocolo de datos usado por KFly un protocolo estándar, no daba pie al uso de los paquetes AutoPilot que actualmente hay en los repositorios de ROS. En el caso de ser usados, conllevaría una enorme cantidad de tiempo la modificación de estos para su uso con la nueva estructura de datos. En segundo lugar, el uso en una primera instancia de KFly requería un ordenador de a bordo, el cual aumentaba el peso por encima de valores recomendables.

En las fases finales de trabajo se empezó a adaptar esta unidad para su uso junto con una unidad de XBee para enviar los datos a la estación de tierra sin necesidad de ordenador de a bordo. Además, el módulo de código final constituye por sí solo un AutoPilot, por lo que también sufraga el otro problema para futuras versiones del prototipo.

3.2.2. Módulo de interfaz con RosCopter

La interfaz que se ha adaptado entre ArduCopter y ROS es un código en Python que conecta la máquina de la estación de tierra con ArduCopter usando XBee y creando temas y servicios para establecer una buena comunicación. El código, adaptación de RosCopter [30], se puede encontrar en el Apéndice A.2

En este módulo se crea una conexión entre el XBee en ArduCopter y el XBee en la estación de tierra usando el protocolo *MAVlink*. Arranca los editores (*publishers*) para GPS, RC (señal del control de radio), *state* (armado, GPS disponible y actual modo), *vfr_hub* (velocidad del viento, velocidad del suelo, orientación, velocidad, altitud y tasa de subida), *attitude* (ángulos y velocidades de Euler) y *raw_IMU*. También arranca los subscriptores (*subscribers*) para *send_rc* (será el tema usado para enviar comandos) y los servicios para armar y desarmar el cuadricóptero. Ahora, en el cuerpo principal del módulo se crea el nodo ("roscopter") y se almacena el mensaje enviado desde ArduCopter con la función *recv_match*. El mensaje

[5] Extensión de los números reales, similar a la de los números complejos. Mientras que los números complejos son una extensión de los reales por la adición de la unidad imaginaria i, tal que $i^2 = -1$, los cuaternios son una extensión generada de manera análoga añadiendo las unidades imaginarias: i, j y k a los números reales y tal que $i^2 = j^2 = k^2 = ijk = -1$. En este caso se utiliza para representar la matriz de rotación de Euler. [29]

es comprobado y se publica en el tema correspondiente dependiendo del tipo de mensaje. Por otro lado, cada vez que un mensaje es enviado a *send_rc*, se lanza un callback[6] para enviar los datos a ArduCopter.

3.2.3. Módulos de simulación

En la Figura 3.8 se presenta el diagrama de bloques utilizado para pruebas en simulación. Éste representa una vista general de la simulación que combina el cuadricóptero virtual con el real, situación descrita en la Sección 3.1.2. El diagrama de simulación pura sólo incluiría el bloque ROS (a la izquierda); de este bloque habría que eliminar también el tema *"/attitude"* y el nodo *"/roscopter"*. En este caso, el control es gestionado por el paquete *pr2_teleop*. Este paquete usa las interrupciones de teclado para modificar el tema *cmd_vel*, el cual controla la velocidad en la gran mayoría de robots ROS.

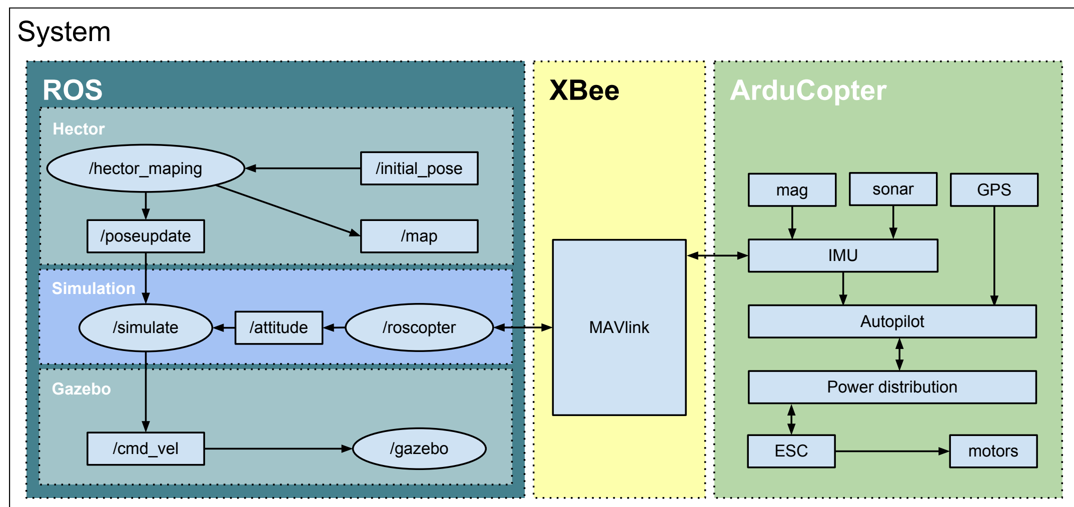


Figura 3.8: Arquitectura del sistema usado para simulación

El módulo ROS, en azul (y a la izquierda), emplea tres programas básicos para llevar a cabo su tarea. En el área central, **Simulation**, el nodo *"/simulate"* es el responsable de realizar el control. Éste lee los datos de ArduCopter (a través del nodo *"/roscopter"*) y del bloque *Hector* y actúa en el comando de velocidad del cuadricóptero (velocidades de Roll, Pitch y Yaw).

En la parte superior, **Hector**, el nodo *"/hector_maping"* crea el mundo y gestiona el mapa de SLAM que el UAV está viendo. Utiliza parámetros físicos para dar forma al robot y situarlo en *Gazebo*. *Hector* usa el tema */poseupdate* para compartir la posición en los tres ejes y el

[6] Un callback es una función "A" que se usa como argumento de otra función "B". Cuando se llama a "B", ésta ejecuta "A". Para conseguirlo, usualmente lo que se pasa a "B" es el puntero a "A".

cuaternio. *Gazebo* y *hector_maping* están también conectados con el tema */scan* (omitido en la imagen para mejorar su comprensión). Con este tema, Hector sabe cómo es el mundo para poder dibujar su mapa de SLAM. Hector también necesita una posición inicial (*/initial_pose*). Esto permite la utilización de cualquier tipo de mapa sin problemas. Esta posición inicial tiene que ser una posición despejada, sin otros objetos.

Por otra lado, en la parte inferior, **Gazebo** gestiona la física (con los parámetros provistos por Hector) en la simulación y crea las visualizaciones del cuadricóptero en un mapa tridimensional. El módulo de código completo utilizado en las simulaciones puede encontrarse en el Apéndice 1.3.2. En este programa, todas las partes activas están en los callbacks. De los cuales hay dos. El primero se ejecuta cuando hay una actualización en el tema *"attitude"* que almacena los ángulos de Euler y sus velocidades. El segundo se lanza cuando la actualización ocurre en el tema *"poseupdate"* que almacena la simulación del cuaternio. El callback *"poseupdate"* transforma el cuaternio a los ángulos de Euler Roll, Pitch y Yaw y los almacena en variables globales. El callback *"attitude"* se encarga del control.

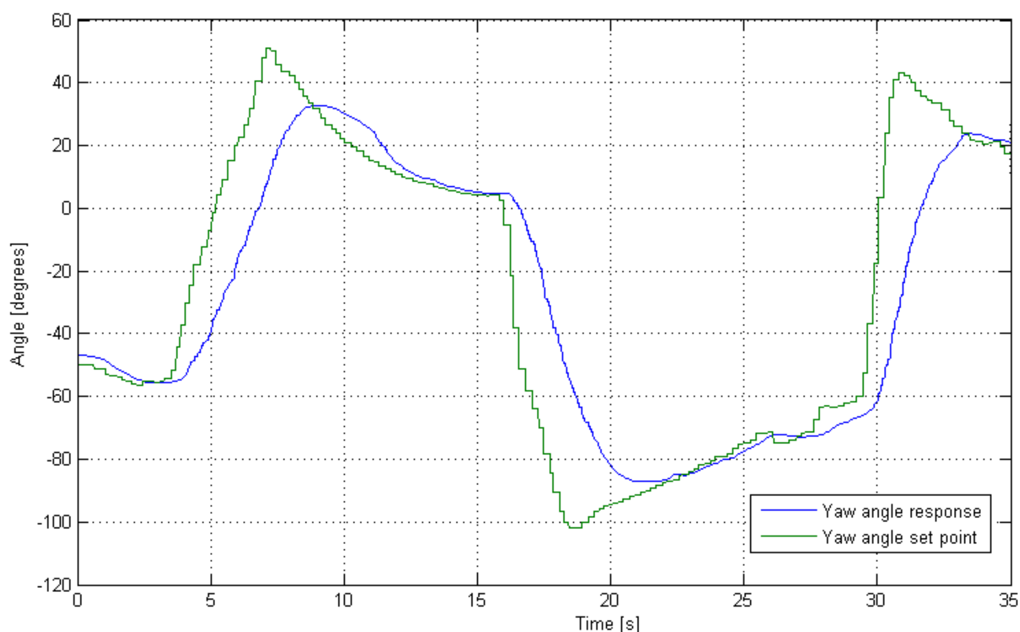


Figura 3.9: Respuesta de la rotación manual en la simulación

En la Figura 3.9 se aprecia cómo, en el cuadricóptero virtual, el ángulo Yaw cambia cuando el ángulo en el cuadricóptero real cambia. Con esta prueba se pretende estudiar el seguimiento que el simulador es capaz de hacer a los sensores reales, el retraso que sufre debido a la lectura y envío de datos, y la sobreoscilación que pudiera tener. Pese a no ser una prueba perfecta que

pueda representar el funcionamiento del control de posición implementado en el simulador, ya que la señal de entrada no es del todo estable (el objetivo se establece a mano), puede verse una rápida respuesta y un perfecto seguimiento. No se aprecia nada de sobreoscilación y el tiempo de respuesta podría establecerse en torno a los dos segundos; retraso debido al envío de datos mediante XBee y a la física propia del cuadricóptero en el simulador. Esta prueba ha sido repetida con iguales resultados en dos ocasiones más, considerando válido el experimento.

3.2.4. Módulo para control del vuelo

Se han desarrollado diferentes programas para mover el cuadricóptero y todos ellos pueden ser llamados desde distintos parámetros en la función *go*, cuyo módulo de código (en Python) está disponible en el Apéndice 1.4.1). De este modo, es posible navegar con puntos vía, rotar a un ángulo concreto o ir a una altitud específica. Hay también un programa de seguridad para despegue o aterrizaje del cuadricóptero.

En la Figura 3.10 se representa el diagrama de trabajo del cuadricóptero. Consiste en tres módulos. El primero, rojo en la figura (izquierda), es el sistema ROS que tiene las mediciones de los sensores (posición global, ángulos de Euler, canales RC o estado del cuadricóptero) y puede actuar en los ángulos Roll y Pitch y en la velocidad de cambio del ángulo Yaw a través de los parámetros del control remoto. Usa dos programas para llevar a cabo su tarea: el principal, */go*; y la interfaz, */roscopter*. El programa principal es usado para cerrar el bucle de control. Los otros dos bloques ya han sido explicados en la introducción de esta sección.

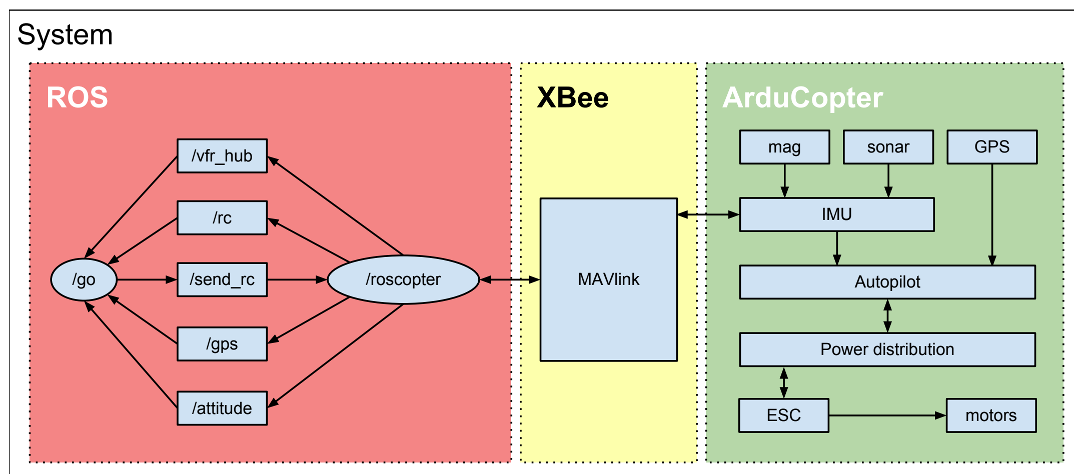


Figura 3.10: Diagrama de nodos del control de vuelo

3.3. Evaluación del Algoritmo de Control

Se usará la plataforma ROS para realizar el control, ya que temas y nodos son buenas herramientas para hacer un control discreto. No es necesario crear bucles periódicos ni una discretización porque los datos son enviados periódicamente desde ArduPilot al sistema ROS a través de MAVlink. Esto significa que el programa de ROS ejecutará la actualización de temas periódicamente, cuando aparezca un nuevo mensaje en el tema.

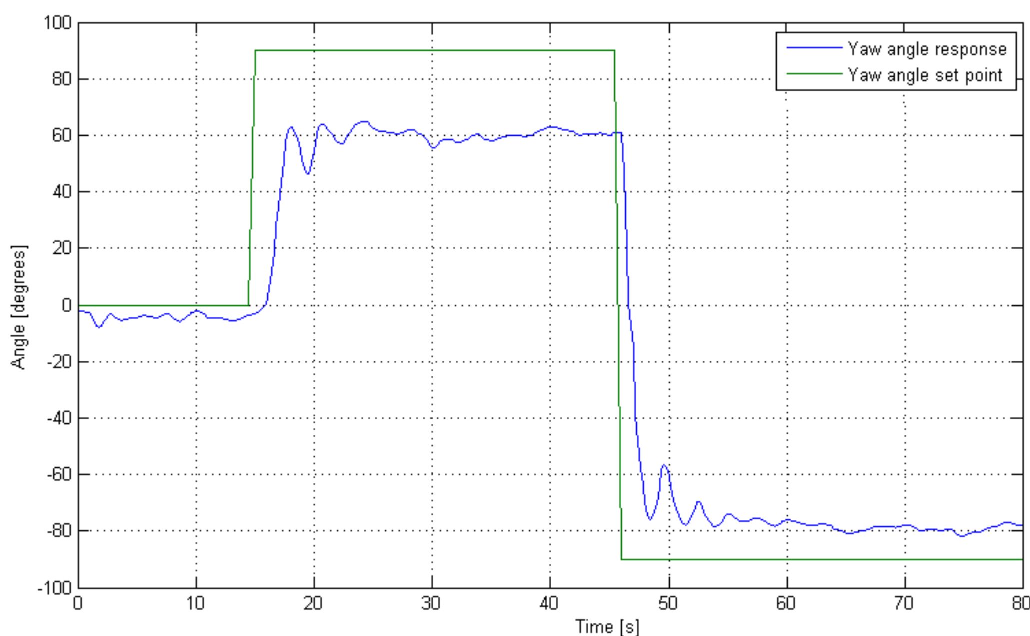


Figura 3.11: Respuesta al escalón en rotación en el rango de ángulos [90°,-90°]

Se han ensayado varios controles para conseguir una rápida respuesta, evitando todo lo que sea posible la sobreoscilación, y logrando una rápida estabilización.

Control Proporcional

En un primer intento se usó un control proporcional para probar las similitudes entre el rendimiento del cuadricóptero real y el de la simulación. Para este experimento, y con el cuadricóptero fijado al cable de seguridad (limitando sus movimientos en el plano horizontal) volando a una altura constante de 1,5m y partiendo desde la orientación 0°, se lanzó un comando de rotación vía terminal de comandos desde el entorno de ROS a +90°, pasados 35 segundos se

modificó la referencia a -90°. Los resultados de la prueba mostraron una ganancia no unitaria y un desplazamiento en la respuesta tal y como puede verse en la Figura 3.11. Esto es debido a la aparición de perturbaciones externas y al umbral de arranque de los motores. Aquí, puede verse tanto que el ángulo nunca alcanza el valor deseado, como que los motores no tienen suficiente velocidad al empezar el movimiento. También es posible ver un error mayor en la zona positiva causado por un pequeño desequilibrio en la velocidad cero.

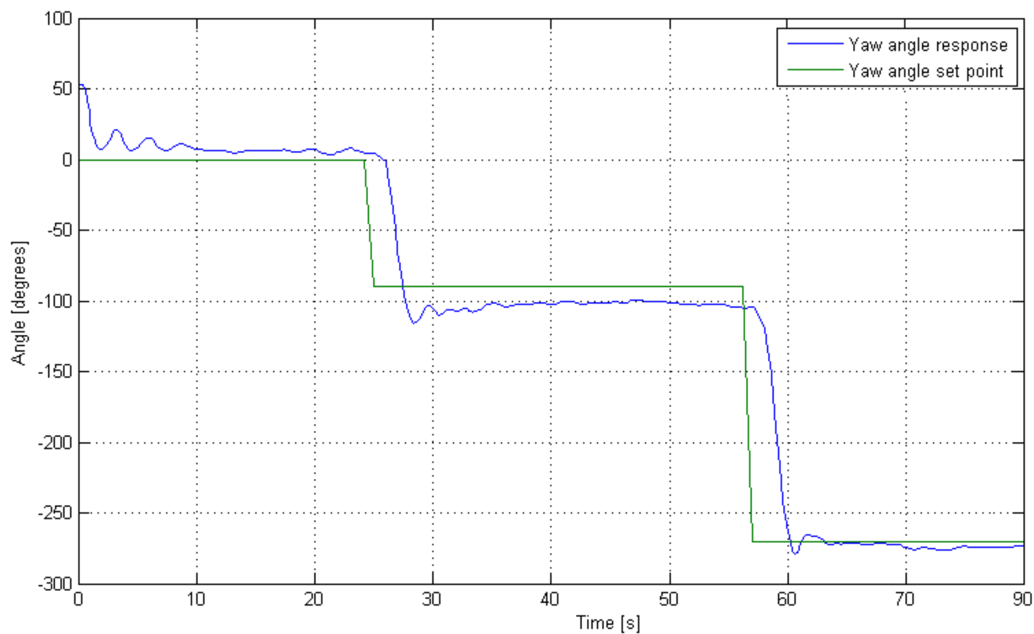


Figura 3.12: Respuesta al escalón en rotación en el rango de ángulos [90°,-90°] con una constante proporcional mayor

Control Proporcional con alta ganancia

Con la misma configuración del experimento anterior, aunque en este caso realizando un giro a -90° seguido de otro a +90° 30 segundos después, y admitiendo una ligera sobreoscilación, con algunos ajustes más en la constante del proporcional es posible solucionar la mayoría del umbral de arranque de los motores. Con este tipo de controlador, y aumentando la constante proporcional, se puede lograr amplificar la lectura del error, consiguiendo así corregir hasta errores pequeños. El problema es que también se crea un sistema más nervioso, propenso a la sobreoscilación, y que ante un cambio de entrada grande puede llegar incluso a inestabilizar el sistema. De esta forma, y aunque se pueden minimizar los errores, es imposible conseguir una ganancia unitaria (Figura 3.12) y, por lo tanto, se aplicará un controlador PID.

Control Proporcional integral y derivativo PID

Manteniendo la misma configuración experimental, la tercera prueba se realizó usando un controlador PID [31]. Con este tipo de controlador se puede añadir el error acumulado a la acción (con la parte integral). Cuanto más tiempo se encuentre el UAV en zona de error, más grande será la acción. También se puede añadir el comportamiento futuro debido al error en la acción usando la velocidad de cambio del error (con la parte derivativa). Finalmente, las siguientes constantes produjeron un comportamiento aceptable:

$$\kappa_p = 2 \qquad \kappa_i = 0{,}1 \qquad \kappa_d = 0{,}07$$

Como es posible ver ahora en la Figura 3.13, si el cambio en la orientación no es demasiado (menos de 120°), la respuesta es perfecta. Emplea 3 segundos en alcanzar el objetivo con un 10 % de error. Después de 16 segundos, el error oscila entre un 2 y un 3 %

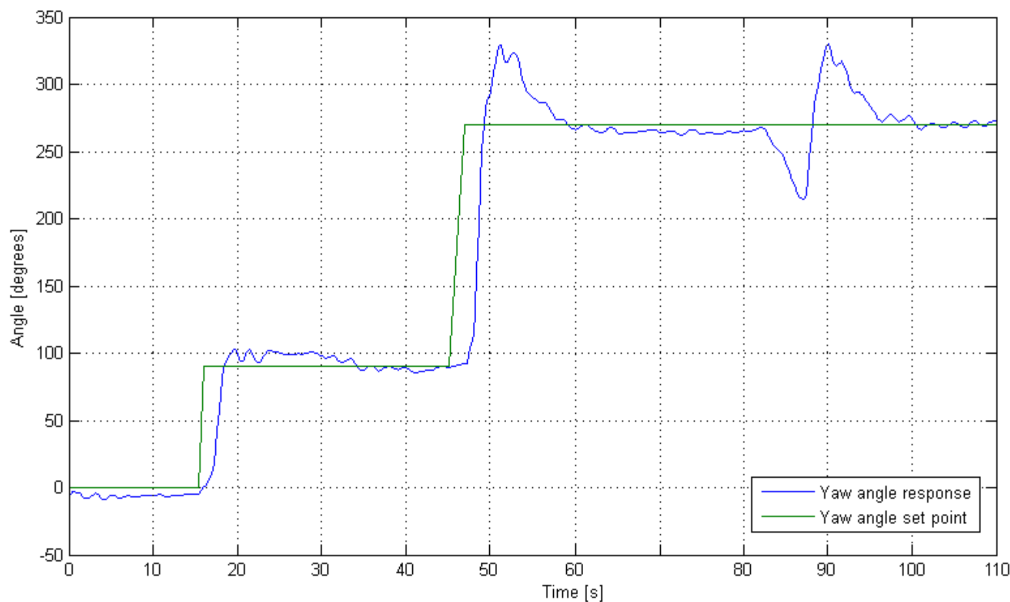


Figura 3.13: Respuesta al escalón y perturbación en la rotación entre [90,-90] con un PID

Si el cambio es mayor (en el segundo 50) se aprecia algo de sobreoscilación (en torno a 50 grados). Ahora emplea 10 segundos en alcanzar el objetivo con un 10 % de error. Despues de 2 segundos más, el error es menor a un 3 %.

También se ensaya una perturbación externa aplicando una rotación manual del cuadricóptero en el segundo 83 para comprobar el comportamiento de la aeronave ante estos imprevistos. Esta perturbación se aplicó con una fuerza considerable, puesto que desde el primer instante el cuadricóptero comenzó a compensar la rotación, haciendo dificil alcanzar una rotación manual excesiva. El sistema puede corregir estas perturbaciones con bastante prontitud, siendo ésta corregida en 10 segundos.

Arquitectura de código y algoritmos

Algoritmo 1: Pasos de una iteración del algoritmo de Control

leer datos
almacenar prev_error
calcular error: goal - data
calcular acción proporcional: κ_p * error
calcular acción integral: prev_integral + κ_i * error
calcular acción derivative: κ_d * (error - prev_error)
calcular acción: Proportional + Integral + Derivative
ajustar la acción al nivel del RC
publicar la acción

En el módulo de código presentado, como se ha indicado, se hace uso de los callbacks para llevar a cabo el control en lugar de bucles. De esta manera el Algoritmo 1 se ejecuta cada vez que el callback *"attitude"* actualiza su tema (lo cual ocurre de manera periódica).

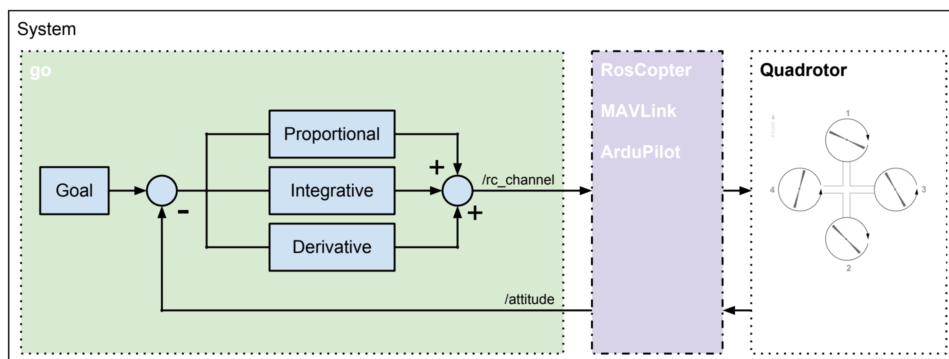


Figura 3.14: Diagrama del control implementado

El algoritmo usa tres términos correctivos, la suma de los tres representa la acción de control. La acción **proporcional** produce un cambio absoluto en la energía del sistema, que modifica el

tiempo de respuesta. La acción **integral** ayuda a eliminar el error residual del periodo estacionario. Mantiene una cuenta del error acumulado y lo añade a la acción para que los errores más pequeños puedan ser eliminados. La acción **derivativa** predice el comportamiento del sistema y, por lo tanto, mejora el tiempo de estabilización del mismo.

En el caso del control por puntos vía, la inclinación del ángulo Roll controla el movimiento en la coordenada Y, la inclinación del ángulo Pitch controla el de la coordenada X y la aceleración del giro de las hélices (*throttle*) controla la ascensión en la coordenada Z. La implementación de este controlador se muestra en la Figura 3.14. En esta representación las interfaces han sido simplificadas para centrar la atención en el código de *go*.

CAPÍTULO 4

Conclusiones y Trabajo Futuro

4.1. Conclusiones

Al término de este proyecto, se ha logrado construir la aeronave con dos configuraciones distintas (en aluminio y en plástico). También se ha logrado hacerla volar usando ArduCopter y una serie de códigos diseñados para el proyecto. Estos códigos van desde la comunicación con la estación de tierra hasta los códigos de control de rotación o de *waypoints*. Pese a que las pruebas en entorno real no han tenido todo el alcance deseable por falta de tiempo, se consideran también un objetivo cumplido. Con todo este trabajo de investigación y construcción también se ha logrado un amplio conocimiento en ROS, así como en diversos lenguajes de programación como Python o C++.

Pero sobre todo, he aprendido que los cuadricópteros son una plataforma con muchas posibilidades en múltiples aplicaciones. El potencial de este tipo de máquinas es enorme y, más aún, cuando se consiga miniaturizar y reducir los costes de las aeronaves. Con ello, se hace acuciante la necesidad de un sistema operativo capaz de trabajar con multitud de robots de forma simultánea y, en caso necesario, con una estación en tierra. Aquí, ROS se despliega como una herramienta capaz de gestionar todo eso, además de ayudar en la tarea de compartir controladores y programas entre la comunidad de desarrolladores.

Con este sistema, es posible recibir datos en una unidad central (por ejemplo un ordenador de a bordo) y procesarlos directamente en ROS. Aquí, se pueden albergar todos los programas de control, telemetría y comunicación. En caso necesario, también se puede gestionar video o realizar SLAM sin salir de ROS.

Con este proyecto, me he dado cuenta de que es posible crear un robot a partir de código abierto, información y librerías, públicos y gratuitos. Aunque no es simple, y lleva un largo tiempo, es posible construir de cero un robot volador. Hay varios diseños libres a lo largo de Internet que pueden ayudarte a construir tu propio cuadricóptero (p.e. [32], [33] o incluso [34]). Así que lo único que te resta por hacer es otorgarle inteligencia y empezar a jugar.

4.2. Trabajo Futuro

El trabajo que se puede hacer en 6 meses es limitado, y por ello la labor que queda por delante es enorme.

Despues de los códigos de navegación por puntos vía y, derivado de esto, la navegación por trayectorias generadas, será posible trazar rutas de acción para el SLAM mediante un software de reconstrucción que aproveche el movimiento del propio quadrotor para tener una imagen estereoscópica. El hardware para dicho propósito sería una cámara USB conectada al ordenador de a bordo. Dicha reconstrucción sería utilizada también para el planeamiento de rutas y la evitación de obstáculos.

En futuras investigaciones se tratará la validez de que el robot sea capaz de realizar su tarea al completo en local (en el ordenador on-board y sin ayuda de la estación). Las alternativas a esto serían el procesamiento de forma remota enviando las imágenes a la estación de tierra para que ésta realizara las tareas de reconstrucción 3D y enviara las nuevas rutas al quadrotor. O también la grabación de dichas imágenes para una reconstrucción en “diferido” del entorno 3D para su uso como mapa en otras plataformas. Estas dos alternativas podrían contribuir a la reducción del gasto energético del cuadricóptero. En el caso del procesamiento remoto se podría evitar el ordenador de a bordo. El problema principal de esto es que la estación de tierra y el cuadricóptero deberían mantener en todo momento una conexión estable.

APÉNDICE A

Código fuente de los módulos desarrollados

A.1. Analizador sintáctico

```
#include <ros/ros.h>
#include <sensor_msgs/Imu.h>
#include <stdio.h>   /* Standard input/output definitions */
#include <string.h>  /* String function definitions */
#include <unistd.h>  /* UNIX standard function definitions */
#include <fcntl.h>   /* File control definitions */
#include <errno.h>   /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */
#include <iostream>

#define PI 3.14159

struct imu
{
  struct orientation
  {
    float x;
    float y;
    float z;
    float w;
  } ;
  struct angular_velocity
  {
    float x;
    float y;
    float z;
  } ;
  struct linear_acceleration
  {
    float x;
    float y;
    float z;
  } ;
} ;

struct euler
{
  float roll;
  float pitch;
  float yaw;
} ;

float toDegrees(float radians)
{
  return radians*180/PI;

}
euler QuaternionToRoll(float x, float y, float z, float w)
{
  float test = x * y + z * w;
  float roll, pitch, yaw;
  euler solution;
  if (test > 0.499) { // singularity at north pole
    pitch = (2 * atan2(x, w));
    yaw = (PI / 2);
    roll = 0;
    solution.roll = toDegrees(roll);
    solution.pitch = toDegrees(pitch);
    solution.yaw = toDegrees(yaw);
    return solution;
```

```
61   }
     if (test < -0.499) { // singularity at south pole
63     pitch = (-2 * atan2(x, w));
       yaw = (-PI / 2);
65     roll = 0;
       solution.roll = toDegrees(roll);
67     solution.pitch = toDegrees(pitch);
       solution.yaw = toDegrees(yaw);
69     return solution;
     }
71   float sqx = x * x;
     float sqy = y * y;
73   float sqz = z * z;
     pitch = atan2(2 * y * w - 2 * x * z, 1 - 2 * sqy - 2 * sqz);
75   yaw = asin(2 * test);
     roll = atan2(2 * x * w - 2 * y * z, 1 - 2 * sqx - 2 * sqz);
77   solution.roll = toDegrees(roll);
     solution.pitch = toDegrees(pitch);
79   solution.yaw = toDegrees(yaw);
     return solution;
81 }

83
   float FourIntsToInt(unsigned char buffer[34], int i)
85 {
     unsigned long HexNumber_aux = buffer[i+0]|(buffer[i+1]<<8)|(buffer[i+2]<<16)|(buffer[i
         +3]<<24);
87
     union u { unsigned long HexNumber_aux2; float solution; };
89   float solution = u{HexNumber_aux}.solution;

91
     return solution;
93 }

95 float TwoIntsToInt(unsigned char buffer[34], int i)
   {
97   unsigned long HexNumber_aux = buffer[i+0]|(buffer[i+1]<<8);

99   union u { unsigned long HexNumber_aux2; float solution; };
     float solution = u{HexNumber_aux}.solution;
101

103  return solution;
   }
105
   int main(int argc, char *argv[])
107 {

109  ros::init(argc, argv, "telemetry");

111  ros::NodeHandle nh;

113  ros::Publisher imuData_pub = nh.advertise<sensor_msgs::Imu>("imuData", 1000);

115  ros::Rate loop_rate(10);

117  float q0, q1, q2, q3;
     float ax, ay, az, wx, wy, wz, mx, my, mz;
119  euler euler_angles;

121  //Open buffer_file
     int buffer_file;
```

```
123     open("/asctec_autopilot-RelWithDebInfo@asctec_autopilot/src/imuData.data", O_CREAT,(mode_t
            )0600);
      buffer_file = open("/asctec_autopilot-RelWithDebInfo@asctec_autopilot/src/imuData.data",
          O_WRONLY);
125   int nw;

127   //Initialize port
      int fd = open("/dev/ttyACM0", O_RDONLY | O_NOCTTY);
129   if (fd == -1) /* Could not open the port. */
        ROS_INFO("open_port: Unable to open /dev/ttyACM0 - ");
131   else
        ROS_INFO("All right");
133
      //Read serial port
135   unsigned char buffer[128] =  { 0};
      int n = read(fd, buffer, sizeof(buffer));
137
      // We only want full chains
139   while ((buffer[0] == 0) || (n != buffer[2] + 6) || (buffer[0] != 0xa6))
      {
141     for (int i = 0; i < n; i++)
          buffer[i] = 0;
143     while ((buffer[0] == 0) || (n < 4))
        {
145       n = read(fd, buffer, sizeof(buffer));
        }
147   }

149   unsigned char recievedData[buffer[2]];

151
      while (ros::ok())
153   {

155     //Look for the first sync byte //This test is always successful
        if (buffer[0] != 0xa6)
157     {
          //sleep(0.02);
159       n = 0;
          for (int i = 0; i < n; i++)
161         buffer[i] = 0;
          while ((buffer[0] = 0) || (n != buffer[2] + 6))
163       {
            for (int i = 0; i < n; i++)
165           buffer[i] = 0;
            n = read(fd, buffer, sizeof(buffer));
167         while ((buffer[0] = 0) || (n < 4))
            {
169           n = read(fd, buffer, sizeof(buffer));
            }
171       }
        }
173
        //Store data values
175     for (int i = 4; i < (buffer[2] + 4); i++)
          if (i < n)
177         recievedData[i - 4] = buffer[i];

179     q0 = FourIntsToInt(recievedData, 0);
        q1 = FourIntsToInt(recievedData, 4);
181     q2 = FourIntsToInt(recievedData, 8);
        q3 = FourIntsToInt(recievedData, 12);
183     ax = TwoIntsToInt(recievedData, 16);
```

```
    ay = TwoIntsToInt(recievedData, 18);
185 az = TwoIntsToInt(recievedData, 20);
    wx = TwoIntsToInt(recievedData, 22);
187 wy = TwoIntsToInt(recievedData, 24);
    wz = TwoIntsToInt(recievedData, 26);
189 mx = TwoIntsToInt(recievedData, 28);
    my = TwoIntsToInt(recievedData, 30);
191 mz = TwoIntsToInt(recievedData, 32);

193 euler_angles = QuaternionToRoll(q0,q1,q2,q3);
    char data[128]="Hi nikhil, How are u?";
195 nw = write(buffer_file,data,128);

197 sensor_msgs::Imu imuMsg;
    imuMsg.header.frame_id = "imu";
199 imuMsg.header.stamp = ros::Time::now();
    imuMsg.header.seq++;
201 imuMsg.orientation.x = q0;
    imuMsg.orientation.y = q1;
203 imuMsg.orientation.z = q2;
    imuMsg.orientation.w = q3;
205 imuMsg.angular_velocity.x = wx;
    imuMsg.angular_velocity.y = wy;
207 imuMsg.angular_velocity.z = wz;
    imuMsg.linear_acceleration.x = ax;
209 imuMsg.linear_acceleration.y = ay;
    imuMsg.linear_acceleration.z = az;
211
    n = 0;
213 for (int i = 0; i < n; i++)
      buffer[i] = 0;
215 while ((buffer[0] = 0) || (n != buffer[2] + 6))
    {
217   for (int i = 0; i < n; i++)
        buffer[i] = 0;
219   n = read(fd, buffer, sizeof(buffer));
      while ((buffer[0] = 0) || (n < 4))
221   {
        n = read(fd, buffer, sizeof(buffer));
223   }
    }
225
    imuData_pub.publish(imuMsg);
227 ros::spinOnce();

229 }

231 return 0;
}
```

A.2. RosCopter

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('roscopter')
import rospy
from std_msgs.msg import String, Header
from std_srvs.srv import *
from sensor_msgs.msg import NavSatFix, NavSatStatus, Imu
import roscopter.msg
import sys,struct,time,os

sys.path.insert(0, os.path.join(os.path.dirname(os.path.realpath(__file__)), '../mavlink/
    pymavlink'))


from optparse import OptionParser
parser = OptionParser("roscopter.py [options]")

parser.add_option("--baudrate", dest="baudrate", type='int',
                  help="master port baud rate", default=57600)
parser.add_option("--device", dest="device", default="/dev/ttyUSB0", help="serial device")
parser.add_option("--rate", dest="rate", default=10, type='int', help="requested stream rate")
parser.add_option("--source-system", dest='SOURCE_SYSTEM', type='int',
                  default=255, help='MAVLink source system for this GCS')
parser.add_option("--enable-control",dest="enable_control", default=False, help="Enable
    listning to control messages")

(opts, args) = parser.parse_args()

import mavutil

# create a mavlink serial instance
master = mavutil.mavlink_connection(opts.device, baud=opts.baudrate)

if opts.device is None:
    print("You must specify a serial device")
    sys.exit(1)

def wait_heartbeat(m):
    '''wait for a heartbeat so we know the target system IDs'''
    print("Waiting for APM heartbeat")
    m.wait_heartbeat()
    print("Heartbeat from APM (system %u component %u)" % (m.target_system, m.target_system))


#This does not work yet because APM does not have it implemented
#def mav_control(data):
#     '''
#     Set roll, pitch and yaw.
#     roll                     : Desired roll angle in radians (float)
#     pitch                    : Desired pitch angle in radians (float)
#     yaw                      : Desired yaw angle in radians (float)
#     thrust                   : Collective thrust, normalized to 0 .. 1 (float)
#     '''
#     master.mav.set_roll_pitch_yaw_thrust_send(master.target_system, master.target_component,
#                                                          data.roll, data.pitch, data.
    yaw, data.thrust)
#
#     print ("sending control: %s"%data)


def send_rc(data):
```

```
58      master.mav.rc_channels_override_send(master.target_system, master.target_component,data.
            channel[0],data.channel[1],data.channel[2],data.channel[3],data.channel[4],data.
            channel[5],data.channel[6],data.channel[7])
        print ("sending rc: %s"%data)
60

62  #service callbacks
    #def set_mode(mav_mode):
64  #     master.set_mode_auto()

66  def set_arm(req):
        master.arducopter_arm()
68      return True

70  def set_disarm(req):
        master.arducopter_disarm()
72      return True

74  pub_gps = rospy.Publisher('gps', NavSatFix)
    #pub_imu = rospy.Publisher('imu', Imu)
76  pub_rc = rospy.Publisher('rc', roscopter.msg.RC)
    pub_state = rospy.Publisher('state', roscopter.msg.State)
78  pub_vfr_hud = rospy.Publisher('vfr_hud', roscopter.msg.VFR_HUD)
    pub_attitude = rospy.Publisher('attitude', roscopter.msg.Attitude)
80  pub_raw_imu =  rospy.Publisher('raw_imu', roscopter.msg.Mavlink_RAW_IMU)
    if opts.enable_control:
82      #rospy.Subscriber("control", roscopter.msg.Control , mav_control)
        rospy.Subscriber("send_rc", roscopter.msg.RC , send_rc)
84
    #define service callbacks
86  arm_service = rospy.Service('arm',Empty,set_arm)
    disarm_service = rospy.Service('disarm',Empty,set_disarm)
88

90  #state
    gps_msg = NavSatFix()
92

94
    def mainloop():
96      rospy.init_node('roscopter')
        while not rospy.is_shutdown():
98          rospy.sleep(0.001)
            msg = master.recv_match(blocking=False)
100         if not msg:
                continue
102         #print msg.get_type()
            if msg.get_type() == "BAD_DATA":
104             if mavutil.all_printable(msg.data):
                    sys.stdout.write(msg.data)
106                 sys.stdout.flush()
            else:
108             msg_type = msg.get_type()
                if msg_type == "RC_CHANNELS_RAW" :
110                 pub_rc.publish([msg.chan1_raw, msg.chan2_raw, msg.chan3_raw, msg.chan4_raw,
                        msg.chan5_raw, msg.chan6_raw, msg.chan7_raw, msg.chan8_raw])
                if msg_type == "HEARTBEAT":
112                 pub_state.publish(msg.base_mode & mavutil.mavlink.MAV_MODE_FLAG_SAFETY_ARMED,
                                      msg.base_mode & mavutil.mavlink.MAV_MODE_FLAG_GUIDED_ENABLED
                                          ,
114                                   mavutil.mode_string_v10(msg))
                if msg_type == "VFR_HUD":
```

```
116                 pub_vfr_hud.publish(msg.airspeed, msg.groundspeed, msg.heading, msg.throttle,
                        msg.alt, msg.climb)

118             if msg_type == "GPS_RAW_INT":
                    fix = NavSatStatus.STATUS_NO_FIX
120                 if msg.fix_type >=3:
                        fix=NavSatStatus.STATUS_FIX
122                 pub_gps.publish(NavSatFix(latitude = msg.lat/1e07,
                                              longitude = msg.lon/1e07,
124                                           altitude = msg.alt/1e03,
                                              status = NavSatStatus(status=fix, service =
                                                  NavSatStatus.SERVICE_GPS)
126                                           ))
                #pub.publish(String("MSG: %s"%msg))
128             if msg_type == "ATTITUDE" :
                    pub_attitude.publish(msg.roll, msg.pitch, msg.yaw, msg.rollspeed, msg.
                        pitchspeed, msg.yawspeed)
130

132             if msg_type == "LOCAL_POSITION_NED" :
                    print "Local Pos: (%f %f %f) , (%f %f %f)" %(msg.x, msg.y, msg.z, msg.vx, msg.
                        vy, msg.vz)
134
                if msg_type == "RAW_IMU" :
136                 pub_raw_imu.publish (Header(), msg.time_usec,
                                         msg.xacc, msg.yacc, msg.zacc,
138                                      msg.xgyro, msg.ygyro, msg.zgyro,
                                         msg.xmag, msg.ymag, msg.zmag)
140

142

144 # wait for the heartbeat msg to find the system ID
    wait_heartbeat(master)
146

148 # waiting for 10 seconds for the system to be ready
    print("Sleeping for 10 seconds to allow system, to be ready")
150 rospy.sleep(10)
    print("Sending all stream request for rate %u" % opts.rate)
152 #for i in range(0, 3):

154 master.mav.request_data_stream_send(master.target_system, master.target_component,
                                        mavutil.mavlink.MAV_DATA_STREAM_ALL, opts.rate, 1)
156
    #master.mav.set_mode_send(master.target_system,
158 if __name__ == '__main__':
        try:
160         mainloop()
        except rospy.ROSInterruptException: pass
```

A.3. Simulación

1.3.1. Simulación

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>

float goal[6][3]={{0,0,0},{1,1,0},{-1,1,0},{-1,-1,0},{1,-1,0},{1,1,0}};
float tolerance = 0.05;

double kp = 0.5;
double ki = 0.0002;
double kd = 0.00005;

float w;

float error_x = 0;
float error_y = 0;
float error_z = 0;
float error_w = 0;
float prev_error_x = 0;
float prev_error_y = 0;
float prev_error_z = 0;
float prev_error_w = 0;
float rise = 1;
float nonstop = true;

float proportional_x = 0;
float proportional_y = 0;
float proportional_z = 0;
float proportional_w = 0;
float integral_x = 0;
float integral_y = 0;
float integral_z = 0;
float integral_w = 0;
float derivative_x = 0;
float derivative_y = 0;
float derivative_z = 0;
float derivative_w = 0;
float action_x = 0;
float action_y = 0;
float action_z = 0;
float action_w = 0;

geometry_msgs::Point real;
geometry_msgs::Twist twist;

bool must_exit = false;
int waypoint_number = 0;

void odoCallback(const geometry_msgs::PoseWithCovarianceStamped::ConstPtr& msg)
{
  real.x=msg->pose.pose.position.x;
  real.y=msg->pose.pose.position.y;
  real.z=msg->pose.pose.position.z;
  w=msg->pose.pose.orientation.z;

  prev_error_x = error_x;
  prev_error_y = error_y;
  prev_error_z = error_z;
```

```
        prev_error_w = error_w;
59      error_x = goal[waypoint_number][0] - real.x;
        error_y = goal[waypoint_number][1] - real.y;
61      error_z = (goal[waypoint_number][2] + 0.001 * rise) - real.z;
        error_w = 0 - w;
63
        proportional_x = kp * error_x;
65      proportional_y = kp * error_y;
        proportional_z = kp * error_z;
67      proportional_w = kp * error_w;
        integral_x += ki * error_x;
69      integral_y += ki * error_y;
        integral_z += ki * error_z;
71      integral_w += ki * error_w;
        derivative_x = kd * (error_x - prev_error_x);
73      derivative_y = kd * (error_y - prev_error_y);
        derivative_z = kd * (error_z - prev_error_z);
75      derivative_w = kd * (error_w - prev_error_w);

77      action_x = proportional_x + integral_x + derivative_x;
        action_y = proportional_y + integral_y + derivative_y;
79      action_z = proportional_z + integral_z + derivative_z;
        action_w = 10 * proportional_w + integral_w + derivative_w;
81
        twist.linear.x = action_x;
83      twist.linear.y = action_y;
        twist.linear.z = action_z;
85      twist.angular.z = action_w;

87      ROS_INFO("Error X: %0.2f \n", error_x);
        ROS_INFO("Error Y: %0.2f \n", error_y);
89      ROS_INFO("Error Z: %0.2f \n", error_y);
        ROS_INFO("W: %0.2f \n", w);
91      ROS_INFO("Action X: %0.2f \n", action_x);
        ROS_INFO("Action Y: %0.2f \n", action_y);
93      ROS_INFO("Action Z: %0.2f \n", action_y);
        ROS_INFO("Action W: %0.2f \n", action_y);
95      ROS_INFO("Voy hacie el objetivo %d \n", waypoint_number+1);

97
        if ((fabs(error_x) < tolerance) && (fabs(error_y) < tolerance)) {
99        if (must_exit == true)
          {
101         twist.linear.x = 0;
            twist.linear.y = 0;
103         twist.linear.z = 0;
            twist.angular.z = 0;
105         exit(0);
           }
107        else
           {
109         waypoint_number += 1;
            rise += 1;
111        }
        }
113
        if (waypoint_number == (sizeof(goal)/sizeof(goal[0])))
115     {
          if (nonstop)
117       {
            waypoint_number = 1;
119       }
          else
```

```
121     {
          waypoint_number = 0;
123       must_exit = true;
        }
125   }
    }
127
    int main(int argc, char **argv)
129 {

131   ros::init(argc, argv, "test_viapoint");

133   ros::NodeHandle np;
      ros::NodeHandle nh;
135   ros::Publisher pub_vel = nh.advertise<geometry_msgs::Twist>("cmd_vel", 1);

137   ros::Subscriber sub = np.subscribe("poseupdate", 1000, odoCallback);

139   ros::Rate loop_rate(10);
      int count = 0;
141
      while (ros::ok())
143   {
        pub_vel.publish(twist);
145     ros::spinOnce();
        loop_rate.sleep();
147     ++count;
      }
149 }
```

1.3.2. Simulación con entrada real

```
#include <ros/ros.h>
#include "roscopter/RC.h"
#include <roscopter/Attitude.h>
#include <sensor_msgs/Imu.h>
#include <geometry_msgs/Quaternion.h>
#include <geometry_msgs/Twist.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <signal.h>
#include <termios.h>
#include <stdio.h>
#include "boost/thread/mutex.hpp"
#include "boost/thread/thread.hpp"

#define PI 3.14159

#define KEYCODE_SPACE 0x20

struct euler
{
  float roll;
  float pitch;
  float yaw;
} ;

geometry_msgs::Twist vel;
geometry_msgs::Quaternion virtual_quaternion;

euler real_angles;
euler virtual_angles;

float kp_p = 0.09;
float kp_y = 0.09;
float kp_r = 0.09;

float ki_p = 0.00004;
float ki_y = 0.00004;
float ki_r = 0.00004;

float kd_p = 0.00001;
float kd_y = 0.00001;
float kd_r = 0.00001;

float yaw_prev_error = 0;
float pitch_prev_error = 0;
float roll_prev_error = 0;

float yaw_error = 0;
float pitch_error = 0;
float roll_error = 0;

float yaw_action = 0;
float pitch_action = 0;
float roll_action = 0;

float yaw_proportional = 0;
float pitch_proportional = 0;
float roll_proportional = 0;

float yaw_integral = 0;
float pitch_integral = 0;
```

```
float roll_integral = 0;

float yaw_derivative = 0;
float pitch_derivative = 0;
float roll_derivative = 0;

float toDegrees(float radians)
{
  return radians*180/PI;
}

float toRads(float degrees)
{
  return degrees*PI/180;
}

int diffAngle(float target, float source)
{
  int a;
  target = (int) target;
  source = (int) source;
  a = target - source;
  a = (a + 180) % 360 - 180;
  return a;
}

euler QuaternionToRoll(float x, float y, float z, float w)
{
  float test = x * y + z * w;
  float roll, pitch, yaw;
  euler solution;
  if (test > 0.499) { // singularity at north pole
    pitch = (2 * atan2(x, w));
    yaw = (PI / 2);
    roll = 0;
    solution.roll = toDegrees(roll);
    solution.pitch = toDegrees(pitch);
    solution.yaw = toDegrees(yaw);
    return solution;
  }
  if (test < -0.499) { // singularity at south pole
    pitch = (-2 * atan2(x, w));
    yaw = (-PI / 2);
    roll = 0;
    solution.roll = toDegrees(roll);
    solution.pitch = toDegrees(pitch);
    solution.yaw = toDegrees(yaw);
    return solution;
  }
  float sqx = x * x;
  float sqy = y * y;
  float sqz = z * z;
  pitch = atan2(2 * y * w - 2 * x * z, 1 - 2 * sqy - 2 * sqz);
  yaw = asin(2 * test);
  roll = atan2(2 * x * w - 2 * y * z, 1 - 2 * sqx - 2 * sqz);
  solution.roll = toDegrees(roll);
  solution.pitch = toDegrees(pitch);
  solution.yaw = toDegrees(yaw);
  return solution;
}

void odoCallback(const roscopter::Attitude& msg)
{
```

```
125   real_angles.roll=toDegrees(msg.roll);
      real_angles.pitch=toDegrees(msg.pitch);
127   real_angles.yaw=toDegrees(msg.yaw);

129   yaw_prev_error = yaw_error;
      yaw_error = diffAngle(real_angles.yaw,virtual_angles.yaw);
131   yaw_proportional = kp_y * yaw_error;
      yaw_integral += ki_y * yaw_error
133   yaw_derivative = kd_y * (yaw_error - yaw_prev_error)

135   yaw_action = yaw_proportional + yaw_integral + yaw_derivative

137   pitch_prev_error = pitch_error;
      pitch_error = real_angles.pitch - virtual_angles.pitch;
139   pitch_proportional = kp_p * pitch_error;
      pitch_integral += ki_p * pitch_error
141   pitch_derivative = kd_p * (pitch_error - pitch_prev_error)

143   pitch_action = pitch_proportional + pitch_integral + pitch_derivative

145   roll_prev_error = roll_error;
      roll_error = real_angles.roll - virtual_angles.roll;
147   roll_proportional = kp_r * roll_error;
      roll_integral += ki_r * roll_error
149   roll_derivative = kd_r * (roll_error - roll_prev_error)

151   roll_action = roll_proportional + roll_integral + roll_derivative

153   vel.angular.z = yaw_action;
      vel.linear.x = pitch_action;
155   vel.linear.y = roll_action;
      printf("Action: %0.2f \n", yaw_action);
157   printf("Error: %0.2f \n", yaw_error);
      printf("Roll: %0.2f \n", real_angles.yaw);
159   printf("Virtual roll: %0.2f \n", virtual_angles.yaw);
    }
161
    void odoCallback2(const geometry_msgs::PoseWithCovarianceStamped::ConstPtr& msg)
163 {
      virtual_quaternion.x=msg->pose.pose.orientation.x;
165   virtual_quaternion.y=msg->pose.pose.orientation.y;
      virtual_quaternion.z=msg->pose.pose.orientation.z;
167   virtual_quaternion.w=msg->pose.pose.orientation.w;

169   virtual_angles = QuaternionToRoll(virtual_quaternion.x,virtual_quaternion.y,
          virtual_quaternion.z,virtual_quaternion.w);
    }
171

173 int main(int argc, char **argv)
    {
175     ros::init(argc, argv, "simulate");
        ros::NodeHandle n;
177     ros::NodeHandle np;
        ros::NodeHandle nh;
179     ros::Publisher pub_vel = n.advertise<geometry_msgs::Twist>("cmd_vel", 1000);

181     ros::Subscriber attitude_sub = np.subscribe("attitude", 1000, odoCallback);
        ros::Subscriber imu_virtual_sub = nh.subscribe("poseupdate", 1000, odoCallback2);
183
        ros::Rate loop_rate(10);
185     int count = 0;
```

```
187     while (ros::ok())
        {
189         pub_vel.publish(vel);
            ros::spinOnce();
191         loop_rate.sleep();
            ++count;
193     }
        return 0;
195 }
```

A.4. Programa de Control

1.4.1. Go

```
#!/usr/bin/env python
from __future__ import print_function
import roslib; roslib.load_manifest('roscopter')
import rospy
from std_msgs.msg import String
from sensor_msgs.msg import NavSatFix, NavSatStatus, Imu
import roscopter.msg
import sys,struct,time,os
import time

from optparse import OptionParser
parser = OptionParser("go.py [options]")

parser.add_option("-m","--mode", dest="mode",
                  help="Working mode: waypoints, goup, rotation", default="waypoints")

parser.add_option("-w","--waypoints", dest="goal_waypoints", type='float',
                  help="Array with the waypoints [x1,y1,z1,x2,y2,z2...]", default=[0,0,0])

parser.add_option("-a","--altitude", dest="altitude", type='float',
                  help="Altitude goal", default=1.0)

parser.add_option("-r","--angle", dest="goal_deg", type='int',
                  help="yaw angle goal", default=180)

(opts, args) = parser.parse_args()

waypoint_number = 0

def rc(data):
    global security
    global throttle
    security = data.channel[5]
    throttle = data.channel[2]

def landing(rate):
    if rate < 1450 :
        for x in range(rate, 1300, -1):
            for x2 in range(x-5,x+5):
          channel = [0,0,x2,0,0,0,0,0]
          pub_rc.publish(channel)
          time.sleep(0.05)
    else:
        for x in range(1450, 1300, -1):
            for x2 in range(x-5,x+5):
          channel = [0,0,x2,0,0,0,0,0]
          pub_rc.publish(channel)
          time.sleep(0.05)
    channel = [0,0,1000,0,0,0,0,0]
    pub_rc.publish(channel)

def security_vel(channel):

    channel_sec = channel

    if (security < 1200):
  print ("Changing to secure mode")
```

```
            channel = [0,0,0,0,0,0,0,0]
59          pub_rc.publish(channel)
        elif (1200 < security < 1700):
61      for i in range(channel[2]-5, channel[2]+5):
            channel_sec[2] = i
63              pub_rc.publish(channel_sec)
        elif (security > 1700):
65      landing(throttle)
            os._exit(1)
67      else:
        for i in range(990, 1010):
69              channel = [0,0,i,0,0,0,0,0]
                pub_rc.publish(channel)
71

73 if opts.mode == "waypoints":
        def gps(data):
75      global error_x
        global error_y
77      global error_z
        global Integral_x
79      global Integral_y
        global Integral_z
81
            x = data.x
83          y = data.y
            z = data.z
85
        prev_error_x = error_x
87      prev_error_y = error_y
        prev_error_z = error_z
89          error_x = opts.goal_waypoints[3*waypoint_number+0] - x
            error_y = opts.goal_waypoints[3*waypoint_number+1] - y
91          error_z = opts.goal_waypoints[3*waypoint_number+2] - z

93          action_x = kp * error_x
            action_y = kp * error_y
95          action_z = kp * error_z

97          position = {'x':x, 'y':y, 'z':z}
            goal = {'gx':opts.goal_waypoints[3*i+0], 'gy':opts.goal_waypoints[3*i+1], 'gz':opts.
                goal_waypoints[3*i+2]}
99          error = {'ex':error_x, 'ey':error_y, 'ez':error_z}
            action = {'ax':action_x, 'ay':action_y, 'az':action_z}
101
            print ("Position: %(x).0.4f, %(y).0.4f, %(z).0.4f" % position)
103         print ("Goal: %(gx).0.4f, %(gy).0.4f, %(gz).0.4f" % goal)
            print ("Error X, Y, Z: %(ex).0.4f, %(ey).0.4f, %(ez).0.4f" % error)
105         print ("Actions X, Y, Z: %(ax).0.4f, %(ay).0.4f, %(az).0.4f" % action)

107     Proportional_x = kp * error_x
        Proportional_y = kp * error_y
109     Proportional_z = kp * error_z
        Integral_x += ki * error_x
111     Integral_y += ki * error_y
        Integral_z += ki * error_z
113     Derivative_x = kd * (error_x - prev_error_x)
        Derivative_y = kd * (error_y - prev_error_y)
115     Derivative_z = kd * (error_z - prev_error_z)

117     action_x = Proportional_x + Integral_x + Derivative_x
        action_y = Proportional_y + Integral_y + Derivative_y
119     action_z = Proportional_z + Integral_z + Derivative_z
```

```
121         rc_action_roll = 1500 + action_y
            rc_action_pitch = 1500 + action_x
123         rc_action_throttle = 1000 + action_z
            channel = [rc_action_roll,rc_action_pitch,rc_action_throttle,0,0,0,0,0]
125
            security_vel(channel)
127
            global waypoint_number
129         if error_x < tolerance and error_y < tolerance and error_z < tolerance:
                if must_exit == 1:
131           landing(throttle)
        os._exit(1)
133       else
              waypoint_number += 1
135
      if waypoint_number == (len(opts.goal_waypoints) / 3):
137       waypoint_number = 0
          must_exit = 1
139
        kp = 2
141     ki = 0.1
        kd = 0.07
143     error = 0
        Integral = 0
145     tolerance = 0.1
        must_exit = 0
147     gps_sub = rospy.Subscriber("gps", NavSatFix, gps)

149 elif opts.mode == "goup":
        def vfr_hud(data):
151   global error
      global Integral
153         global security

155         altitude = data.alt

157   prev_error = error
            error = goal - altitude
159         action = kp * error

161   Proportional = kp * error
      Integral += ki * error
163   Derivative = kd * (error - prev_error)

165   action = Proportional + Integral + Derivative

167         rc_action = 1000 + action
            channel = [0,0,rc_action,0,0,0,0,0]
169
      security_vel(channel)
171
        kp = 2
173     ki = 0.1
        kd = 0.07
175     error = 0
        Integral = 0
177     goal = opts.altitude
        vfr_hud_sub = rospy.Subscriber("vfr_hud", roscopter.msg.VFR_HUD, vfr_hud)
179
    elif opts.mode == "rotation":
181     def attitude(data):
            global error
```

```
        global Integral
        global throttle

        yaw = data.yaw * 180 / 3.14

        prev_error = error
        error = goal - yaw
        error = (error + 180) % 360 - 180 #only in yaw
        Proportional = kp * error
        print ("Error: %0.4f"%error)
        print ("Action: %0.4f"%Proportional)
        print ("Yaw: %0.4f"%yaw)
        print ("Security: %0.4f"%security)

        Integral += ki * error
        Derivative = kd * (error - prev_error)

        action = Proportional + Integral + Derivative

  rc_action = 1480 + action
        channel = [0,0,1300,rc_action,0,0,0,0]
        channel = [0,0,security,rc_action,0,0,0,0]

  security_vel(channel)

        f = open('yaw_data','a')
        ticks = time.time()
        to_file = {'time': ticks, 'yaw': yaw, 'goal': goal, 'error': error, 'prev_error':
            prev_error, 'Proportional': Proportional, 'Integral': Integral, 'Derivative':
            Derivative}
        f.write("%(time)d %(yaw)0.4f %(goal)0.4f %(error)0.4f %(prev_error)0.4f %(Proportional
            )0.4f %(Integral)0.4f %(Derivative)0.4f \n"%to_file)

    kp = 2 # 1.75
    ki = 0.1
    kd = 0.07
    error = 0
    Integral = 0
    goal = opts.goal_deg  #in rads
    attitude_sub = rospy.Subscriber("attitude", roscopter.msg.Attitude, attitude)

security = 0
pub_rc = rospy.Publisher('send_rc', roscopter.msg.RC)
rc_sub = rospy.Subscriber("rc", roscopter.msg.RC, rc)

def mainloop():

    pub = rospy.Publisher('rolling_demo', String)
    rospy.init_node('rolling_demo')

    while not rospy.is_shutdown():
        rospy.sleep(0.001)

if __name__ == '__main__':
    try:
        mainloop()
    except rospy.ROSInterruptException: pass
```

1.4.2. Landing

```
#!/usr/bin/env python
from __future__ import print_function
import roslib; roslib.load_manifest('roscopter')
import rospy
from std_msgs.msg import String
from sensor_msgs.msg import NavSatFix, NavSatStatus, Imu
import roscopter.msg
import sys,struct,time,os
import time

def vfr_hud(data):

    w = data.heading
    altitude = data.alt

    error = 0 - altitude
    error_w = 0 - w
    action = kp * error
    action_w = kp * error_w

  Proportional = kp * error
    Proportional_w = kp * error_w

  action = Proportional
    action_w = 10 * Proportional_w

    rc_action = 1000 + action
    rc_action_w = 1500 + action_w
    channel = [0,0,rc_action,rc_action_w,0,0,0,0]

  security_vel(channel)

kp = 0.5
vfr_hud_sub = rospy.Subscriber("vfr_hud", roscopter.msg.VFR_HUD, vfr_hud)

security = 0
pub_rc = rospy.Publisher('send_rc', roscopter.msg.RC)
rc_sub = rospy.Subscriber("rc", roscopter.msg.RC, rc)

def mainloop():

    pub = rospy.Publisher('landing', String)
    rospy.init_node('landing')

    while not rospy.is_shutdown():
        rospy.sleep(0.001)

if __name__ == '__main__':
    try:
        mainloop()
    except rospy.ROSInterruptException: pass
```

APÉNDICE B

Imágenes y Planos

B.1. Impresora 3D

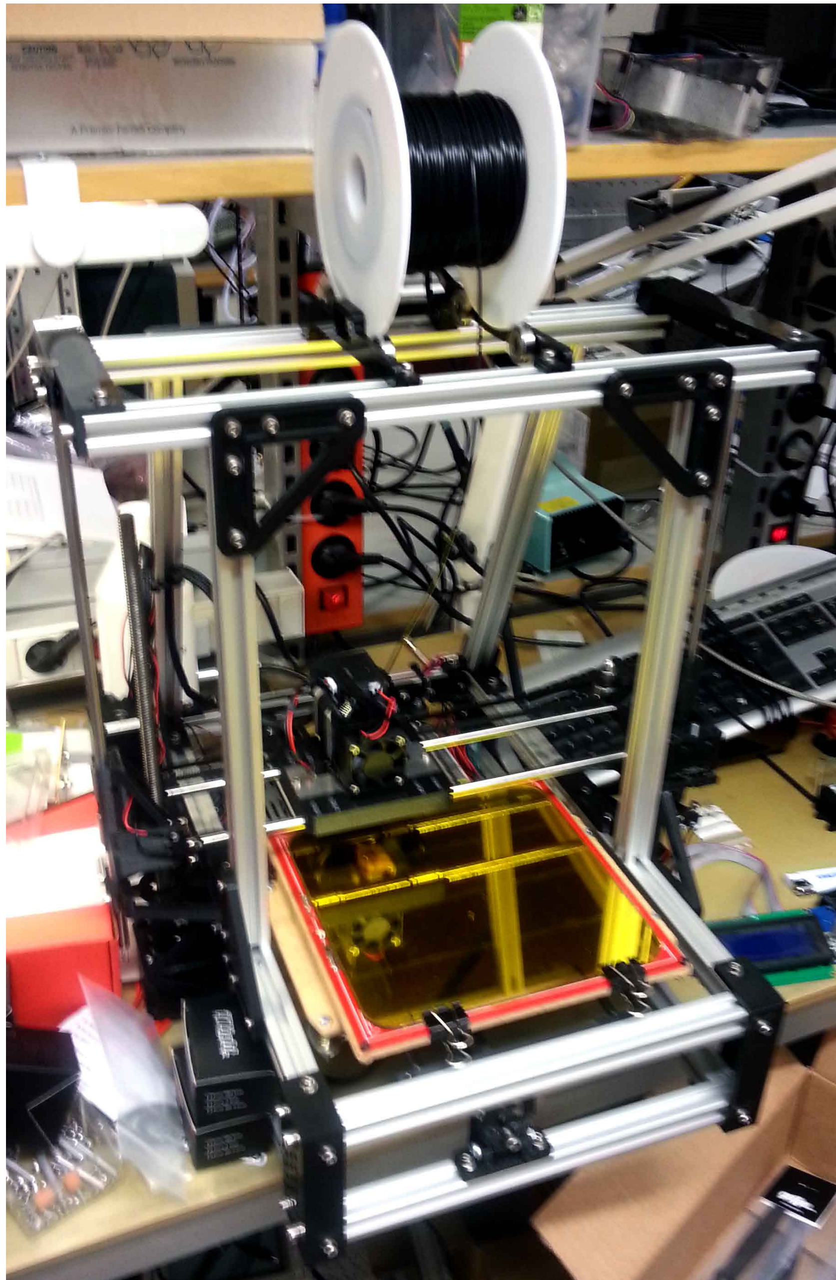


Figura B.1: Impresora 3D usada en el proyecto

B.2. Planos de la impresión 3D

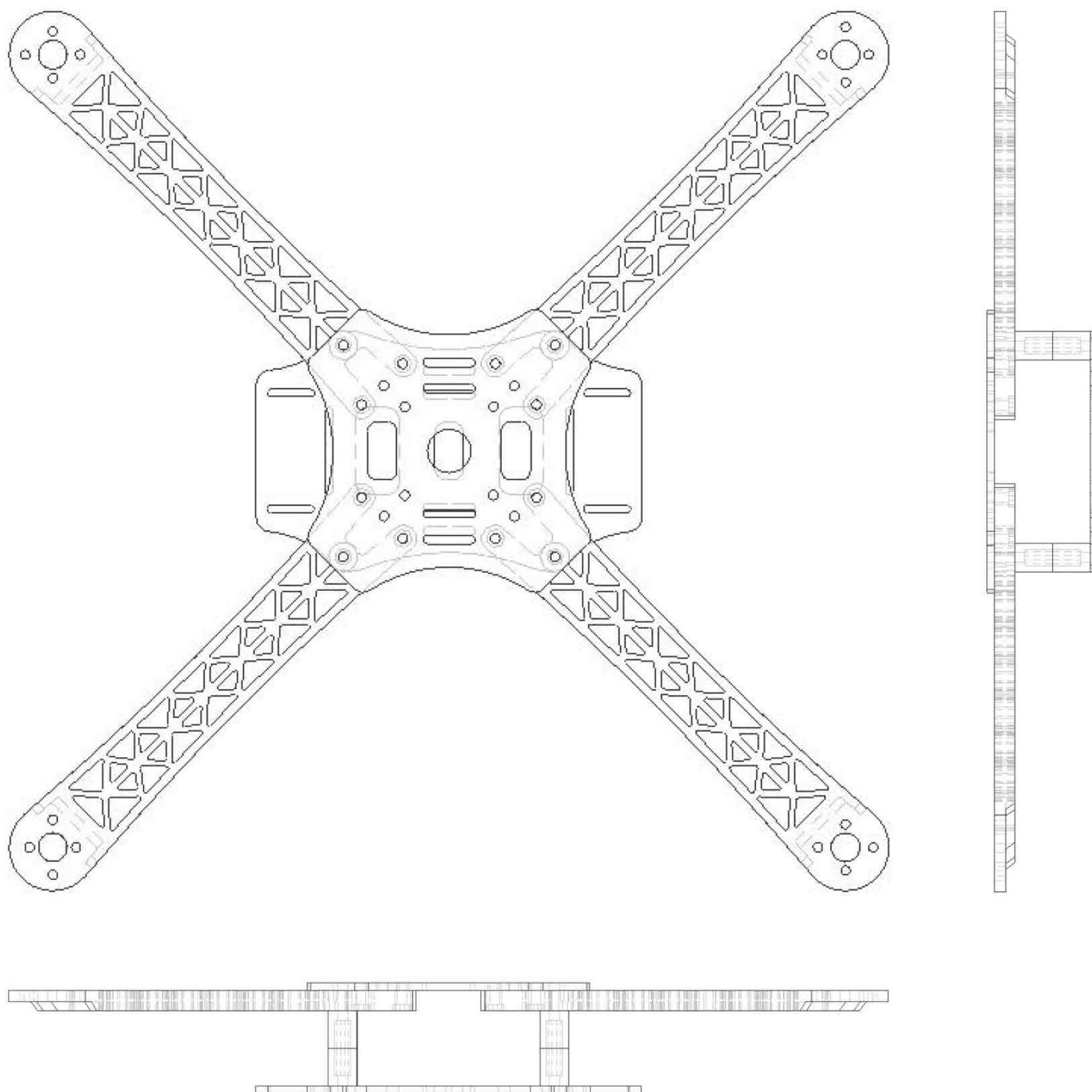


Figura B.2: Plano del ensamblaje del cuadricóptero

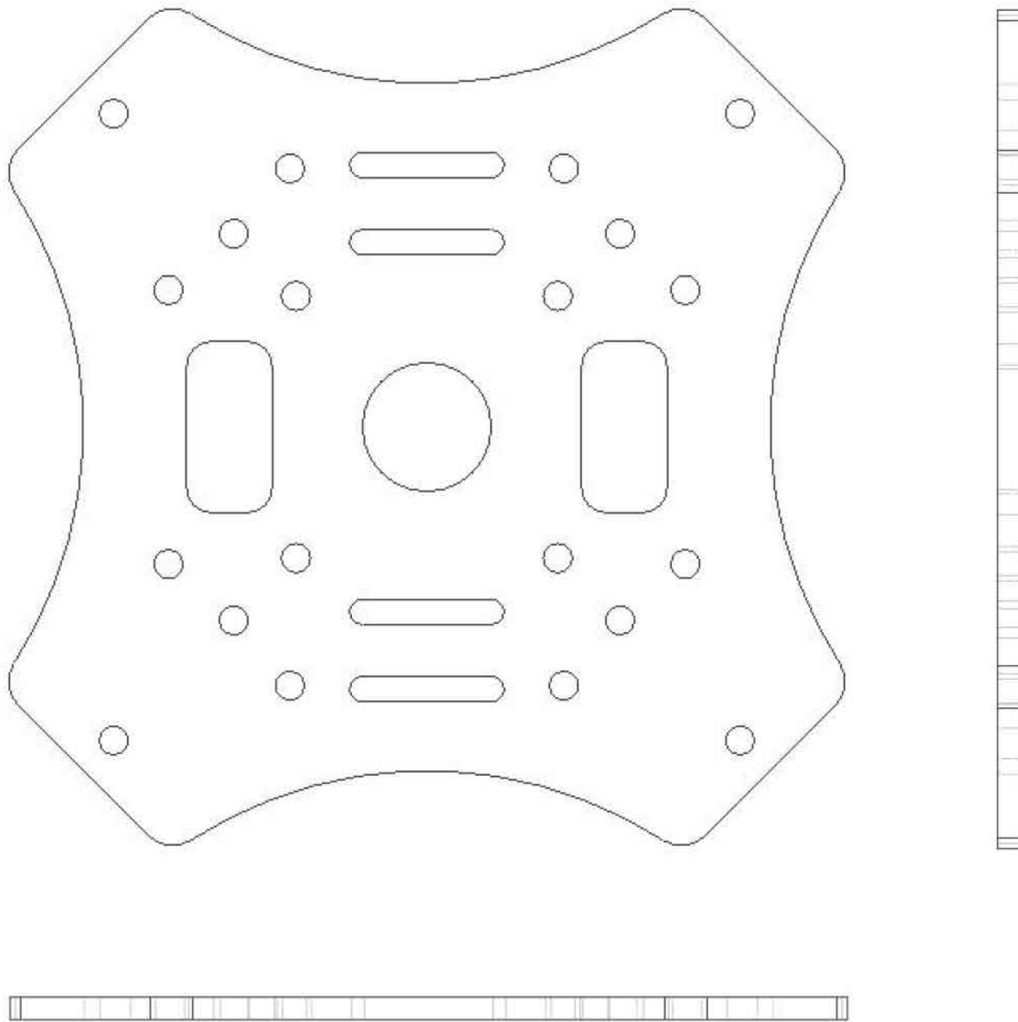


Figura B.3: Plano del plato superior del cuadricóptero

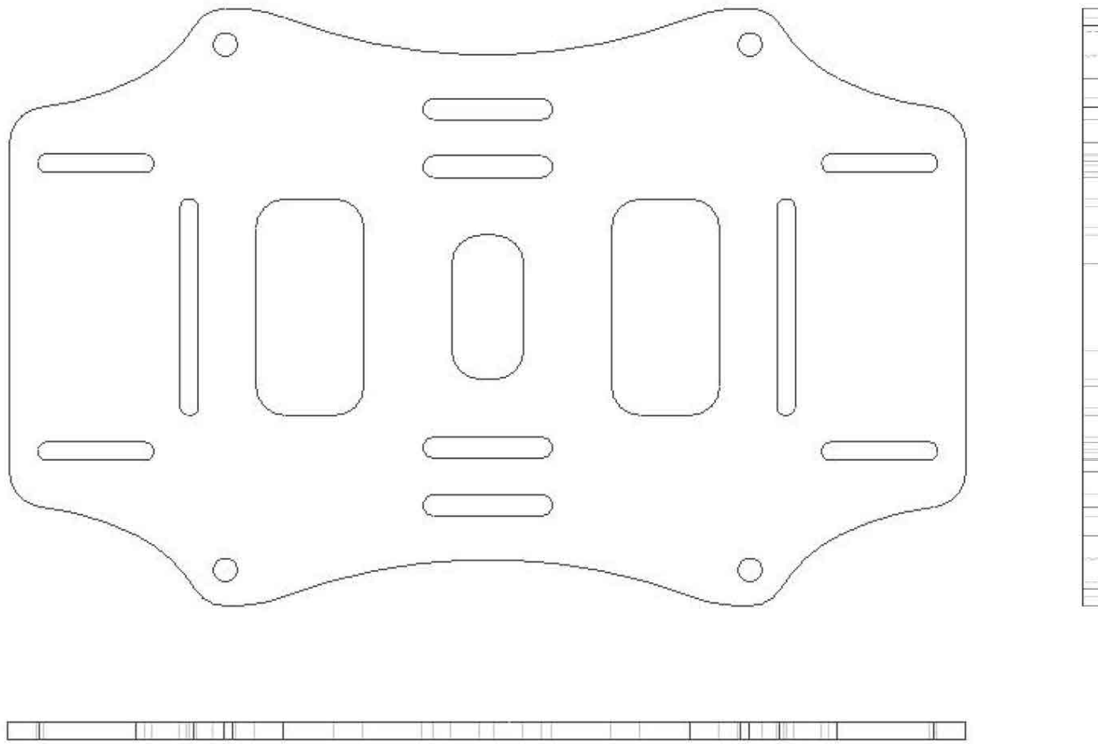


Figura B.4: Plano del plato inferior del cuadricóptero

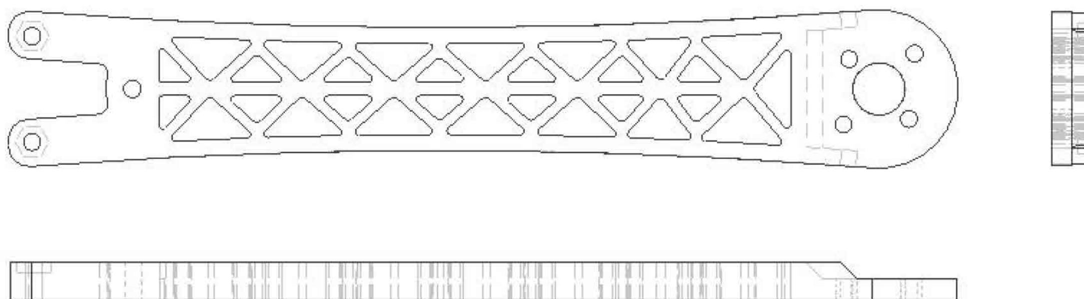


Figura B.5: Plano de los brazos del cuadricóptero

B.3. Montaje

Figura B.6: Cuadricóptero de fabricación propia usado en el proyecto

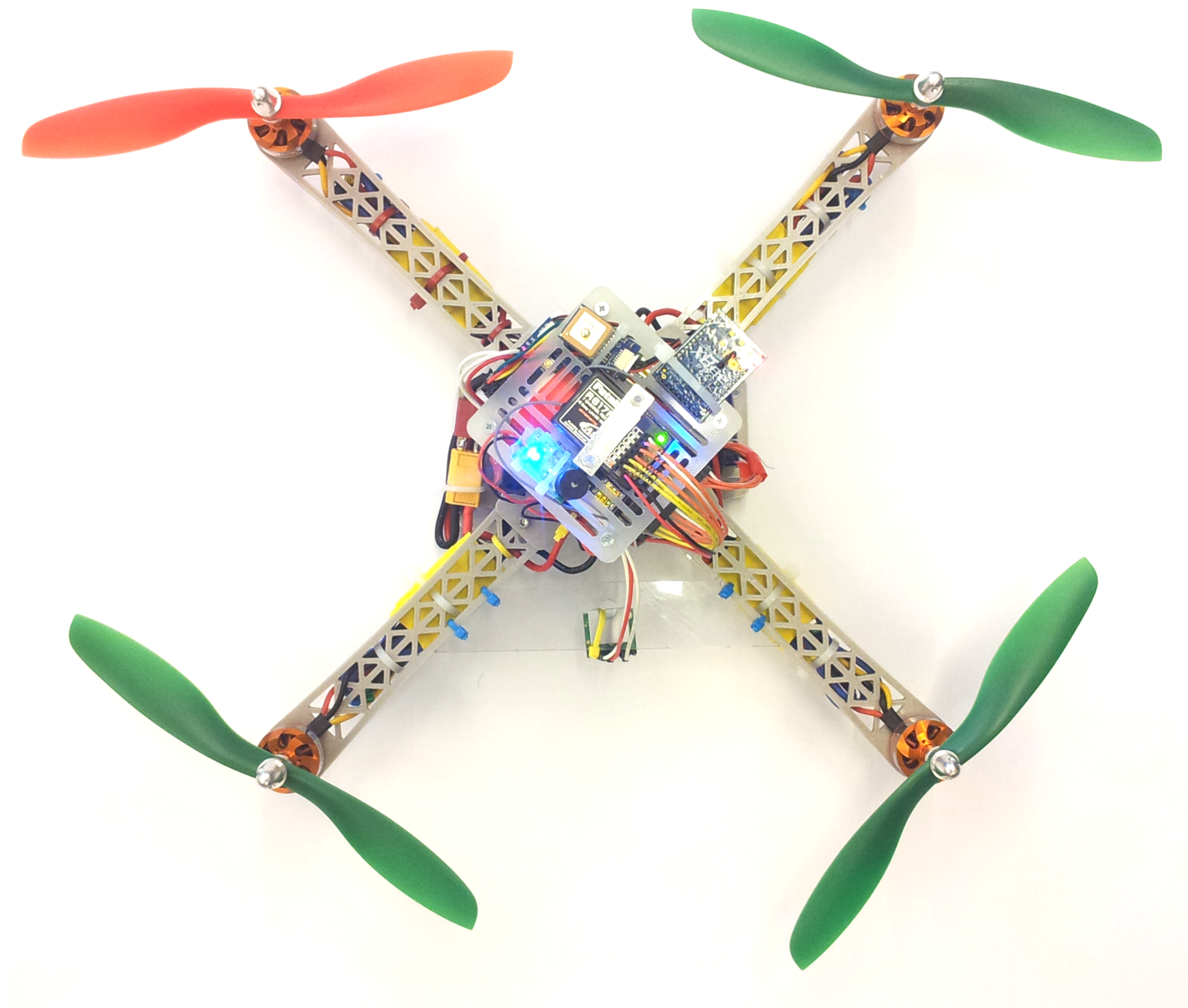
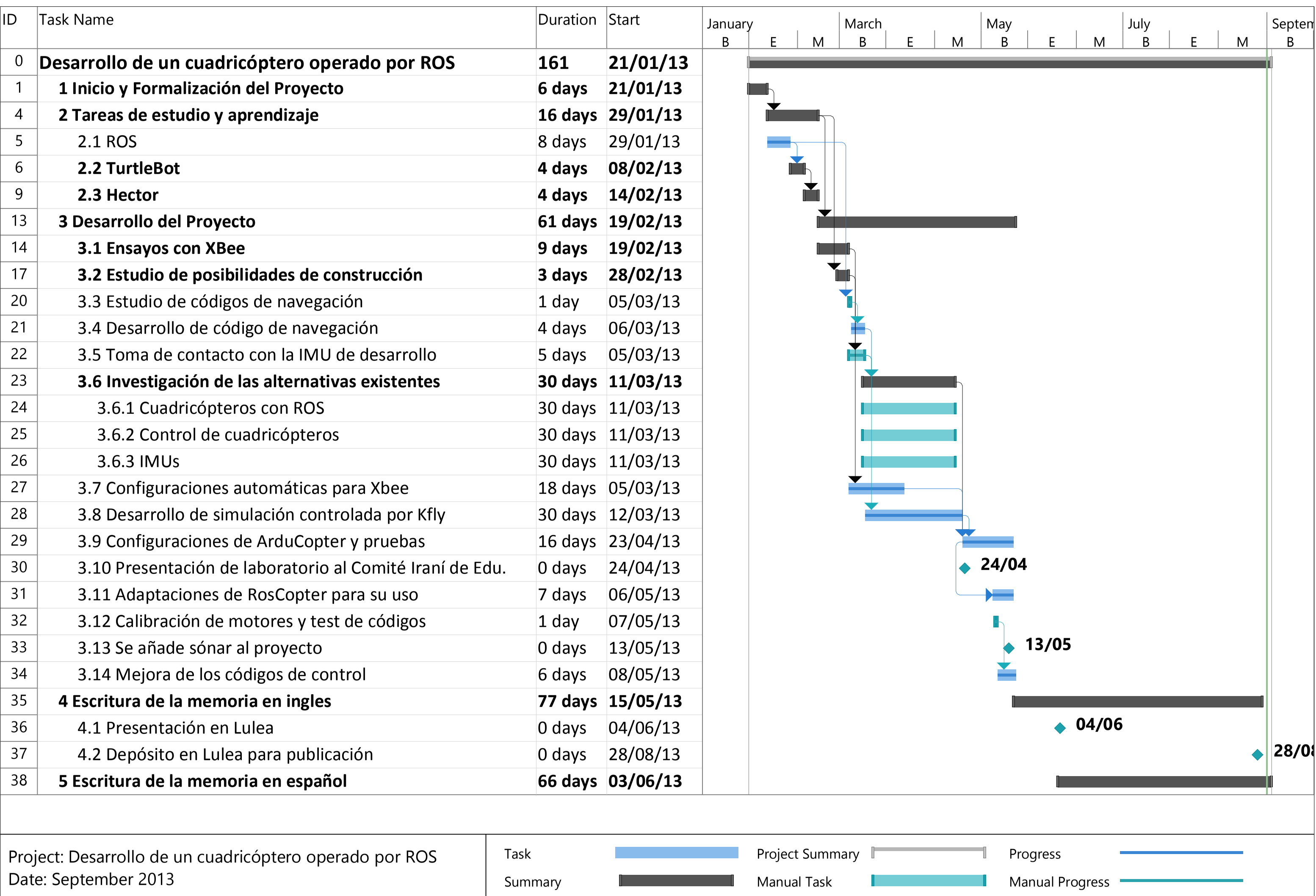


Figura B.7: Cuadricóptero de fabricación propia usado en el proyecto (vista en planta)

APÉNDICE C

Diagrama de Gantt


ID
Task Name
Duration
Start
January
March
May
July
Septem
B
E
M
0 Desarrollo de un cuadricóptero operado por ROS 161 21/01/13
1 1 Inicio y Formalización del Proyecto 6 days 21/01/13
4 2 Tareas de estudio y aprendizaje 16 days 29/01/13
5 2.1 ROS 8 days 29/01/13
6 2.2 TurtleBot 4 days 08/02/13
9 2.3 Hector 4 days 14/02/13
13 3 Desarrollo del Proyecto 61 days 19/02/13
14 3.1 Ensayos con XBee 9 days 19/02/13
17 3.2 Estudio de posibilidades de construcción 3 days 28/02/13
20 3.3 Estudio de códigos de navegación 1 day 05/03/13
21 3.4 Desarrollo de código de navegación 4 days 06/03/13
22 3.5 Toma de contacto con la IMU de desarrollo 5 days 05/03/13
23 3.6 Investigación de las alternativas existentes 30 days 11/03/13
24 3.6.1 Cuadricópteros con ROS 30 days 11/03/13
25 3.6.2 Control de cuadricópteros 30 days 11/03/13
26 3.6.3 IMUs 30 days 11/03/13
27 3.7 Configuraciones automáticas para Xbee 18 days 05/03/13
28 3.8 Desarrollo de simulación controlada por Kfly 30 days 12/03/13
29 3.9 Configuraciones de ArduCopter y pruebas 16 days 23/04/13
30 3.10 Presentación de laboratorio al Comité Iraní de Edu. 0 days 24/04/13
31 3.11 Adaptaciones de RosCopter para su uso 7 days 06/05/13
32 3.12 Calibración de motores y test de códigos 1 day 07/05/13
33 3.13 Se añade sónar al proyecto 0 days 13/05/13
34 3.14 Mejora de los códigos de control 6 days 08/05/13
35 4 Escritura de la memoria en ingles 77 days 15/05/13
36 4.1 Presentación en Lulea 0 days 04/06/13
37 4.2 Depósito en Lulea para publicación 0 days 28/08/13
38 5 Escritura de la memoria en español 66 days 03/06/13
24/04
13/05
04/06
28/0
Project: Desarrollo de un cuadricóptero operado por ROS
Date: September 2013
Task
Summary
Project Summary
Manual Task
Progress
Manual Progress

MEMORIA EN INGLÉS

Developing a ROS Enabled Full Autonomous Quadrotor

Iván Monzón

Luleå University of Technology
Department of Computer science,
Electrical and Space engineering, Control Engineering Group

June 2013

Abstract

The aim of this Master Thesis focuses on: a) the design and development of a quadrotor and b) on the design and development of a full ROS enabled software environment for controlling the quadrotor. ROS (Robotic Operating System) is a novel operating system, which has been fully oriented to the specific needs of the robotic platforms. The work that has been done covers various software developing aspects, such as: operating system management in different robotic platforms, the study of the various forms of programming in the ROS environment, evaluating building alternatives, the development of the interface with ROS or the practical tests with the developed aerial platform.

In more detail, initially in this thesis, a study of the ROS possibilities applied to flying robots, the development alternatives and the feasibility of integration has been done. These applications have included the aerial SLAM implementations (Simultaneous Location and Mapping) and aerial position control.

After the evaluation of the alternatives and considering the related functionality, autonomy and price, it has been considered to base the development platform on the ArduCopter platform. Although there are some examples of unmanned aerial vehicles in ROS, there is no support for this system, thus proper design and development work was necessary to make the two platforms compatible as it will be presented.

The quadrotor's hardware has been mounted on an LTU manufactured platform, made through 3D printing, and its functionality has been evaluated in real environments. Although an aluminium platform has been also evaluated and tested with less satisfactory results.

For proper operation of the whole system, a connection between the quadrotor and the ground station has been established. In this case, an alternative connection between computers (for the case of an on board computer is mounted on the aircraft) or connection between computer and ArduCopter (for the case of no on board computers) have been designed.

A series of algorithms to perform the quadrotor control autonomously has been also implemented: Navigation with way points, rotation control and altitude control. The novelty of the proposed activities is that for the first time all these control modules have been designed and developed under the ROS system and have been operated in a

networked manned from the ground station.

Finally, as it will be presented, a reader module for the adopted inertial measurement unit, currently under development by the University of Luleå (KFly), has also been developed. This device, although tested in controlled laboratory environments, has not yet became part of the quadrotor, but in the near future is expected to serve as a replacement to the on board computer.

PREFACE

This project has been done as a degree thesis to finish my studies in Industrial Engineering with a major in Industrial Automation and Robotics by the Zaragoza University (Spain). The project has been developed and presented in the Luleå University of Technology with the team of the Department of Computer science, Electrical and Space engineering; whom I thank for their support and collaboration.

It was a hard and long journey for almost six month, full of delays and problems. But now I can say it was worth it.

Therefore, I would like to thank the Erasmus project to give me the chance to go to this wonderful Swedish city, Luleå.

Finally, I would like to thank too George Nikolakopoulos, my supervisor in this endeavor, for his support and optimism.

Contents

Chapter 1 – Introduction 1
1.1 Robots 1
1.1.1 History 1
1.1.2 Robots in service of man 3
1.2 UAVs (Unmanned Aerial Vehicle) 4
1.2.1 Types 5
1.3 Quadrotors 7
1.4 The Problem 8
1.5 Goals 9

Chapter 2 – ROS 11
2.1 Robot Operating System 11
2.1.1 Definition 11
2.1.2 Robots running ROS 12
2.1.3 Nomenclature 14
2.1.4 Applications, sensors and more 16
2.1.5 Setup 18
2.2 First tests to understand the platform 18
2.3 Evaluating ROS: Hector Quadrotor package 22

Chapter 3 – Building the Quadrotor 27
3.1 Hardware 27
3.1.1 ArduCopter 27
3.1.2 XBee 30
3.1.3 Positioning Systems 31
3.1.4 Remote Control 32
3.2 Code Architecture and Evaluations in Real Tests 32
3.2.1 Interface 32
3.2.2 Simulation 32
3.2.3 Flying code 34
3.3 Controller 35

Chapter 4 – Future work 41

Chapter 5 – Conclusion **43**

Appendix A – Program codes **45**
A.1 RosCopter . . . 45
A.2 Simulate . . . 49
1.2.1 Simulation . . . 49
1.2.2 Simulation with real input . . . 52
A.3 Control code . . . 56
1.3.1 Go . . . 56
1.3.2 Landing . . . 60

Appendix B – Glossary **61**

CHAPTER 1

Introduction

1.1 Robots

1.1.1 History

Robotics has came to our world, and it has been to stay. Since Heron of Alexandria edited *Pneumatica* and *Automata* twenty two centuries ago, many things changed in our approach to the robotics and more on how robotics changed our world. At that time, Robots have been considered magic, illusions to entertain an ignorant public. Now, robotics is part of our daily life, and continuously serve to entertain, although now they target at a much more cultured public, more specialized and thirsty to be aware of what happens after each of the movements of the robots.

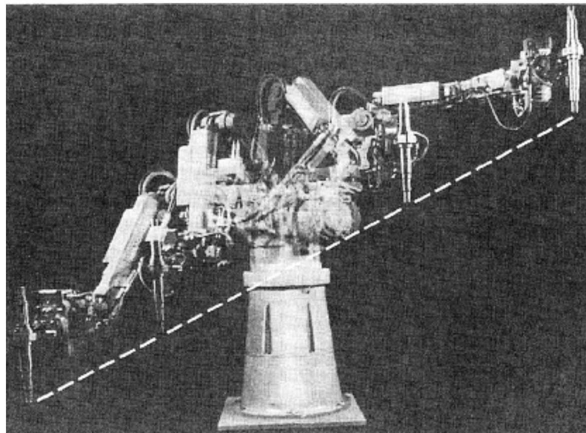


Figure 1.1: Cincinnati Milacron T3 Industrial Robot

Furthermore, it has exploited many different avenues of growth. When we began to

see that robotics was able to help us in our tasks (mid-50s), there was a revolution, in the moment we realized that we did not need to create things, but we were also able to create things to work for us.

As you can see in these few lines, robotics is a science with history, but it was asleep for a long time. Therefore, it can be stated that the evolution in robotics has been started half a century ago and from that point until now this evolution has been unstoppable.

In the early stages of this evolution, progress was mechanical and hydraulic mostly. But after about 18 years, this began to change. Cincinnati Milacron created the first computer-controlled robot (in Figure 1.1), and at this time robotics began to grow without boundaries.

In parallel to the robotic evolution, the significant advances in the field of computer science has launched robotics to a fantastic world to discover. Now it was possible to program much more complex tasks than before in minimum space, while it was not necessary to create unitask robots, as it was possible even to teach them to work and reprogram operations as often as needed.

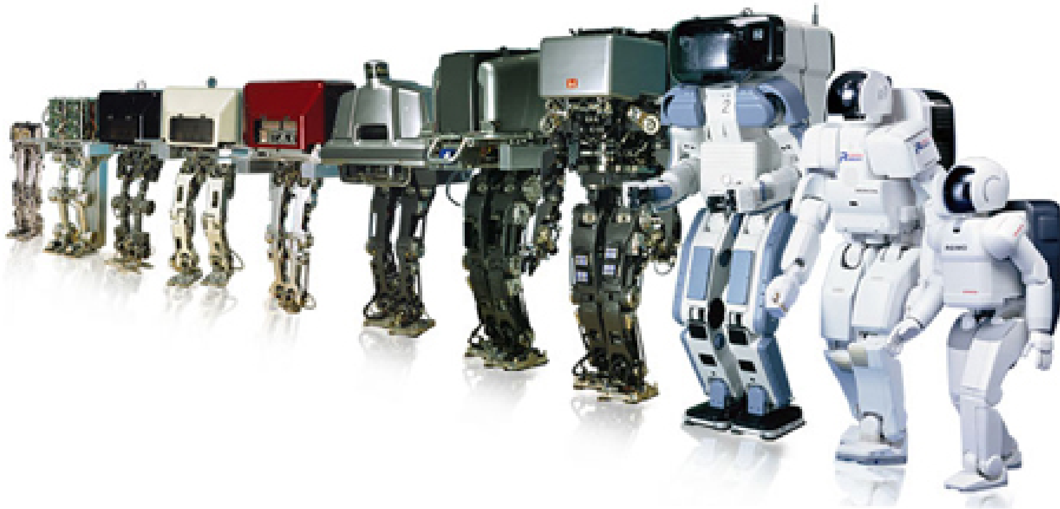


Figure 1.2: Honda ASIMO and its predecessors

From that day until today the history of robotics is already known. Broadly speaking, there are industrial robots in all fairly large factories in the developed world. Painting robots, welding robots, cutting robots, etc. (such as PUMA [1] or Kuka [2]). It should be mentioned that is only the part that was born and evolved with those first industrial robots in the mid 20th century.

At the same time we began to investigate about intelligence and autonomous robots.

In 1987 DARPA showed that they could create an autonomous vehicle [3] that using laser and computer vision creates a path to move between two points. In 1997 the first computer was able to beat a chess master in a game (Kasparov vs. Deep Blue [4]). Asimo (in Figure 1.2) [5], the Honda humanoid robot is created in 2000.

1.1.2 Robots in service of man

From this historical point and until the beginning of the new century, the XXI century, a significant change in the way of thinking towards robots took place. The robotics ceases to be a exclusive domain of research and it is becoming something commercial, something that can be close to the ordinary people.

In the world of toys it begins to arise a variety of robotic pets with basic interaction capabilities. On medical fields, wheelchairs managed with the mind, rehabilitation robots, prostheses controlled by nerve endings or telesurgery (the da Vinci Surgical System).

Figure 1.3: Roomba, the iRobot cleaning robot

Those robots can be found also in industrial tasks, inspection robots, etc.

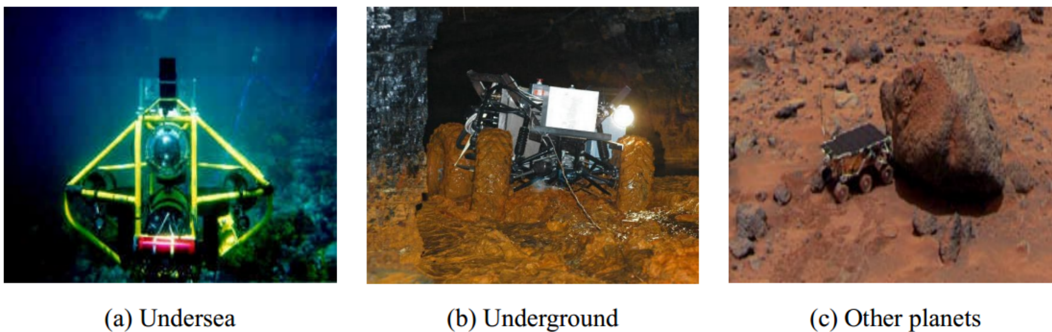
In domestic areas, kitchens that run unattended, automatic cleaners (Roomba [6], in Figure 1.3); systems that control the temperature, humidity, or light in the house; refrigerators that detect a shortage of some product and it automatically ask the supermarket; inspection robots to check and fix the pipes; or home theatre systems that adapt the sound to the room geometry and to the playback features.

In parallel to these activities, the research efforts have focused in two branches: a) the military, and b) the civilian. In military matters, DARPA has been since its inception the reference entity. In 1994 they developed the MQ-1 Predator [7], one of the first unmanned aerial vehicles. Also emerged from their laboratories the most current Boeing X-45 (2004)

[8]. Currently, they are developing the ACTUV (autonomous military vessel) [9], the BigDog and the LS3 (4-legged autonomous robots) [10], the EATR (an autonomous vehicle which is able to run on any biomass) and the future Vulture (a UAV able to be in the air for five years).

Even with a much smaller funding, civil investigations have been no less. The European project AWARE [11] has designed autonomous unmanned aerial robots, distributed and able to cooperate with each other for intervention in disasters and fires. Researchers at IFM-GEOMAR have created the largest fleet of underwater exploration robots in Europe [12]. The Google driverless car is capable of running autonomously both city and road [13], while brands such as Volvo, BMW and Audi also have similar projects underway.

A very interesting topic that significantly added intelligence in the field of autonomous robots has been the SLAM (Simultaneous Localization and Mapping) concept. SLAM addresses the problem of robot to traverse a previously unknown terrain. The robot makes a relative observation on the surrounding environment to reconstruct a map of that environment so it can create a motion path to follow. If that map is available, the robot uses the SLAM information to position itself properly in it. Robots of all kinds (as you can see in Figure 1.4) make use of this technique for orientation.



(a) Undersea (b) Underground (c) Other planets

Figure 1.4: Robots using SLAM

1.2 UAVs (Unmanned Aerial Vehicle)

Among all this astonishments range of robotic possibilities, the research that concerns us here is specifically focused on the problem of autonomous flying robots.

UAVs, or unmanned aerial vehicles, have been in our world since the First World War, but at that time they were very focused on military environments. Then, UAVs have been used as a safer way to fly (at least for the side that controlled it), since no pilot was required. Moreover, for this reason, you could make smaller aircraft, more aerodynamic, more agile, and with a less fuel consumption.

A UAV, with greater or lesser degree of intelligence, can communicate with the controller or the base station and transmit: the image data (thermal, optical, etc..), information regarding its status, the position, the airspeed, the heading, the altitude or any parameter of its telemetry. Usually, these vehicles are armed with systems that in the case of failure of any of the on board components or program, are able to take corrective actions and/or alert the operator. In the most intelligent cases, they are able to take "imaginative" actions if they tackle a unexpected problem.

1.2.1 Types

As in every field, UAVs can be sorted using many features. Size, range, flying principle or utility would be the most successful.

The Figure 1.5 shows some of the current UAVs. From cheap and small toys that you can buy anywhere as the RC planes; to big military projects such as the Predator. They can be remotely piloted, as the Bell Eagle Eye [14], or autonomous, as the AAI shadow [15].

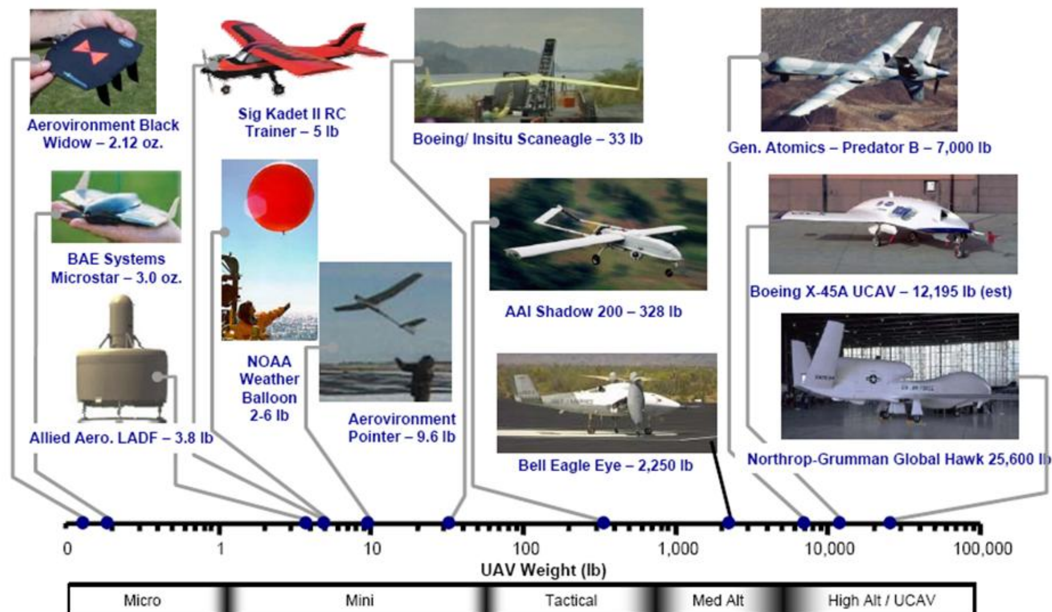


Figure 1.5: Diferents commercial UAVs [16]

Focusing on the low cost UAVs, in Figure 1.6 we can see a classification of these machines by the flying principle.

Fixed wing UAVs have the advantage of a balanced consumption; the simple mechanism; and the stealth operation, because they can fly without mobile parts and gliding,

while the mobility is not too good. They are good for flying from one place to another, but not to much more.

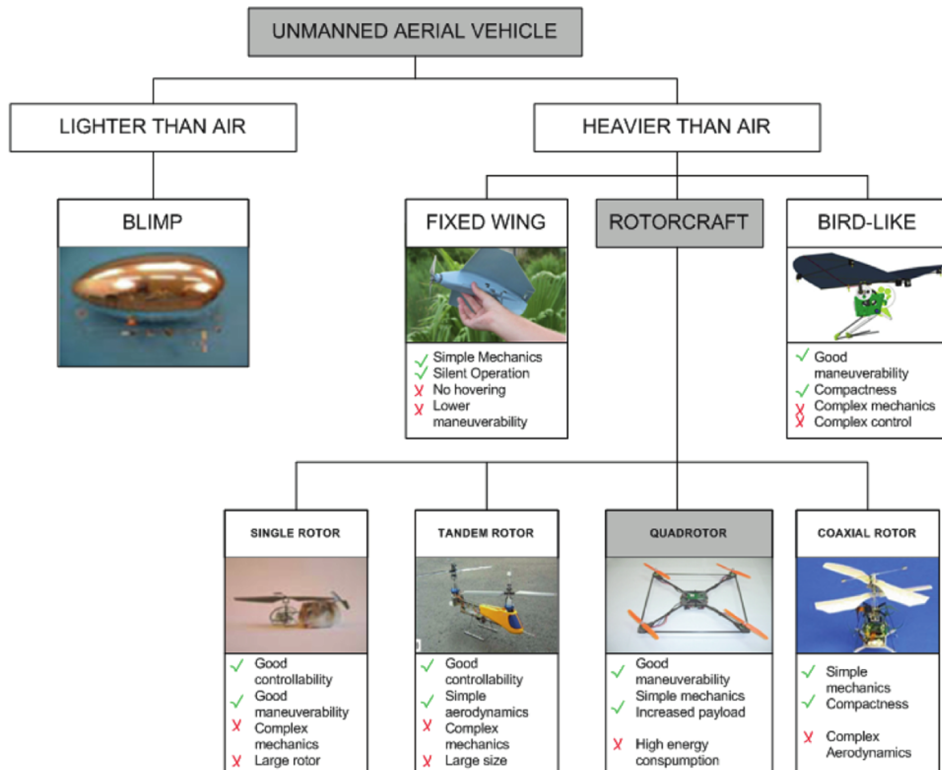


Figure 1.6: Types of UAVs [17]

Rotorcraft UAVs usually have a higher consumption and complex mechanics but they have higher handling capabilities. They can operate in inhospitable places, in small spaces and complex paths.

In this group, the multi rotor helicopter is a special member. It has some problems with the energy consumption as its batteries do not last long, but it outperforms almost every other UAV on manoeuvrability, simplicity of mechanics and even they are easier to drive.

From now, we are going to focus in the autonomous and low cost vehicles for its high growth potential and, more specifically, in autonomous multi rotor helicopters. They can make decisions faster than humans and most important, if they lose the ground connection, they can continue working.

The evolution continues with the size, with small flying robots, more precise jobs (as in [18]) can be done, go through smaller paths or be more stealth. Also a cooperation robot matrix is possible with small size robots.

For example, if we consider a SLAM UAV, a smaller one could perform lower flights and could achieve a really good world reconstruction, reconstructions that before you had to make by hand (or terrestrial vehicles) with cameras. In addition, with them, you can have a bigger z-dimension that helps us to reconstruct high buildings, mountains, trees, etc. Nowadays, you can mix an aerial reconstruction with a ground one with the same vehicle, thereby reducing time and money and making it more accurate.

Figure 1.7: Quadrotor

1.3 Quadrotors

For some years now, one new kind of UAV has begun to have a lot of attention, and well-deserved recognition, the quadrotor. Until that time, UAVs were heavier and not enough agile than it should be (helicopters [19] and [20]).

A quadrotor has a really good stability in the air (by assuming on board good sensors). It can keep still in the air as a helicopter, but it also can do sharp turns or fly upside down without any problem, and it is lighter than helicopters and planes.

A quadrotor, also called a quadrotor helicopter or quadcopter, is a multicopter that is lifted and propelled by four rotors. As you can see in Figure 1.7, it is small, between 10 cm and 150 cm. The bigger the helicopter is, the more autonomy may have, because if the propeller is bigger it can rotate slower to lift and the efficiency is better, while the smaller the size is, the more agile it is.

A quadrotor is a stable aircraft, therefore, it can be used indoors, but it can also be a dangerous machine, because it is fast, and it can turn and change direction quickly. Thus, losing control of the quadrotor can destroy itself or its environment or harm someone.

In our lab, and to test the code in a safe environment, a vertical wire with bumpers (in Figure 1.8) has been built. The quadrotor is hooked up on it and it can rotate and

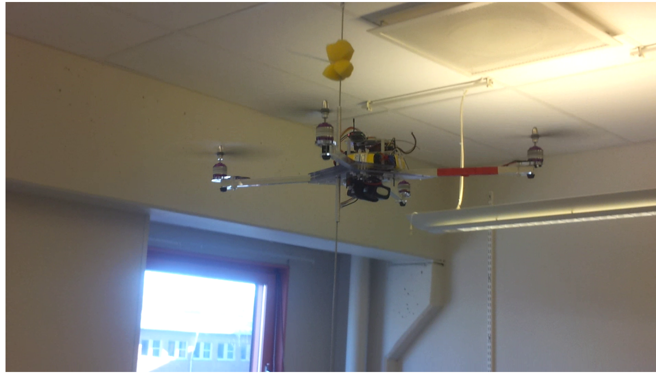


Figure 1.8: LTU lab with a safety system for quadrotors

lift. There has been also a safety net between the quadrotor and the operators.

1.4 The Problem

The aim of this Master Thesis is to develop a fully ROS enabled quadrotor and experimentally demonstrate the capabilities of the platform on a realistic flight scenarios, such as attitude and altitude stabilisation and position hold. ROS (Robotic Operating System) is an open source distributed robotics platform designed to accelerate robotic research and development by supporting reusable and modular components able to implement a variety of low and high level functionalities, while during the last 3 years it has received significant attention from the scientific community, as the most prestigious research groups in the area of robotics are putting efforts in this field.

Currently, there are already some existing ROS enabled software implementations for quadrotors but these software implementations are platform depended and they aim mainly in performing a tele-operation task, which means that a user should always fly the quadrotor from a distance. The results of this Master Thesis will focus in a novel approach, within the ROS field, where the aim is to create a quadrotor being able to perform autonomous tasks, such as: attitude and altitude stabilisation in a fully ROS enabled environment. The performance of the proposed software architecture will be evaluated in both simulations and extended experimental tests. The main contribution of this Thesis will be in the design and implementation of ROS enabled software control modules, while covering various software developing aspects, such as: operating system management, forms of programming in the ROS environment, software building alternatives, ROS interfaces and finally various practical software implementation issues with the developed aerial platform.

1.5 Goals

The general goal for this Mater Thesis is to extend the vast amount of ROS libraries and the novel approach in designing robotic applications, quadrotors in this case, by creating a proper modular software architecture that will allow the autonomous operation of a quadrotor, like attitude and altitude stabilisation, without the need of human interference. The control modules will be based on the classical PID control architecture, but the novelty of the current implementation will be that all these control modules will be developed to be ROS enabled and will be synchronised with the rest of the ROS operating modules for achieving proper flight performance.

To achieve the previous goal the current Thesis has been divided in a number of related activities that include the development of the experimental platform, the design and development of the ROS enabled software architecture and the final extended experimental verifications in real environments.

To be able to design a full ROS enabled software architecture, a broad knowledge of the ROS software is considered necessary, which should include both the understanding of its performance and handling as well as the knowledge to program in both C++ and Python, and the ability to modify and adapt existing modules, create new ones and synchronise the whole software architecture of the system. More analytically the goals of this Master Thesis will be the following ones:

1. Literature study in the area of ROS enabled robots and on ROS documentation and functionalities, especially for the case of Unmanned Aerial Vehicles (UAV).

2. Experimental quadrotor platform customization and adaptation to ROS needs.

3. Programming and synthesizing ROS functionalities and system architecture for allowing the autonomous flight (such as attitude and altitude stabilisation), including sensor integration and motor control.

4. Developing ROS enabled control modules, such as On/Off, P and PID controllers.

5. Comparison of ROS modules and experimental evaluation of the quadrotor's flight performance.

Chapter 2

ROS

2.1 Robot Operating System

Since the advent of the Internet, developer communities have become an essential part of scientific discovery. Now, it is not necessary to be a professional researcher to get into fields, until recently restricted to professional areas.

Now, just as a hobby, you can learn and research in any imaginable field. Instructables or even Youtube make this task easier. But if we break away from the general services, there is a world of possibilities.

And that is where ROS comes in. ROS allows people all over the planet, people with any level of knowledge, to insert into the robotics world. ROS is not just an operating system, is a complete ecosystem, nourished code and tools that help us to give life to our creations. But above all, it is a community. A system such as ROS helps the spread of ideas, the free use of knowledge to achieve faster progress in this technological world.

2.1.1 Definition

Robot Operating System, or ROS, as it is known, is an open source (distributed under the terms of the BSD license) software framework for robot software development. It provides libraries and tools to help software developers create robot applications. Moreover, it provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more functionalities for delivering fast integrated solutions.

In general "A system built using ROS consists of a number of processes, potentially on a number of different hosts, connected at runtime in a peer-to-peer topology." [21]

ROS framework is also specifically developed to reuse drivers and code from another robots. This functionality is being achieved by encapsulating the code in standalone libraries that have no dependencies on ROS.

In some cases, ROS can be used only to show configuration options and to route data into and out of the respective software, with as little wrapping or patching as possible.

At the end of the day, the most important reason for using ROS and not other framework is that it is part of a collaborative environment. Due to the big amount of robots and artificial intelligence software that have been developed, the collaboration between researchers and universities is essential. To support that development ROS provides a package system; the ROS package which is a directory containing an XML file to describe the package and the related dependencies.

At the time of writing, several thousand ROS packages exist as open source on the Internet, and more than fifteen hundred are directly exposed in `http://www.ros.org/browse`.

2.1.2 Robots running ROS

Figure 2.1: Modlab's CKBots

Since 2007, when ROS was initially released, there have been one hundred different ROS enabled robots in the market. This covers a considerable number of types of robots, mobile robots, manipulators, autonomous cars, humanoids, UAVs, AUVs, UWVs and even some uncategorizable robots as CKBot (in Figure 2.1). The complete list of ROS enabled robots can be located here: `http://www.ros.org/wiki/Robots` (Some of them can be seen in Figure 2.2).

This allow us to use this platform for example in a iRoomba or in a Lego NXT without changing nothing in the code, so that you can focus more in new developments, without the need for time-consuming initial settings.

Figure 2.2: ROS enable robots

In the field of quadrotor there has not been much content in ROS servers with which we could help. Right now, there are 5 universities fully involved in this field, as it will be presented in the sequel.

On one hand, the CCNY Robotics Lab in New York is developing along with AscTec the Pelican and the Hummingbird, which includes the complete software packages in the repositories of ROS. The University of Pennsylvania has a long list of experiments with Penn Quadrotors, including aerobatics flights or light performances with quadrotors swarms.

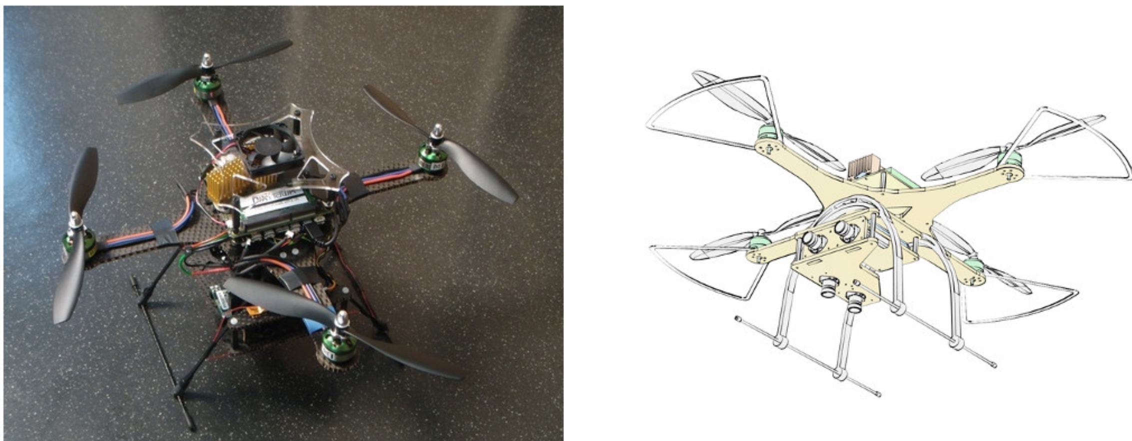


Figure 2.3: PixHawk Cheetah BRAVO

PixHawk Cheetah BRAVO (in Figure 2.3) is being developed from the ETH Zurich, it uses a stereo camera system to be able to perform indoor navigation. The Autonomy Lab, at the Simon Fraser University in Canada, is developing a ROS-based driver for the Parrot AR.Drone (in Figure 2.4), one of the few commercial quadrotors. Finally the project STARMAC (in Figure 2.5) by Stanford University aims to explore cooperation in a multi-vehicle system with a multi-agent controller.

Figure 2.4: Parrot AR.Drone

In every case, except with PixHawk, specific hardware is used, and in some cases hardware built for the experiment, such as low level boards or IMUs. This restricts the ROS utilization in specific hardware, and thus in this thesis a more broader and hardware independent ROS development approach has been investigated.

2.1.3 Nomenclature

Everything on the ROS system is based on nodes, messages, topics and services. This allows us to have a simple code structure and to follow a schematic way of work.

The nodes

The *nodes* are processes that execute a specific task. It is like a software module, while nodes can use the *topics* to communicate each other and use the services to do some simple and external operations. The nodes usually manage low complexity tasks. One moves the motors, another does the SLAM, another handles the laser, etc.

Figure 2.5: One of the STARMAC vehicles in flight

It is possible to see the nodes in a graphical representation easily with the *rxplot* command, as for example it has been presented in Figure 2.6 for the case of *sr_hand* program, which is used to access and experiment with Shadow Robot's hardware, where the *shadowhand* shares data with the logout ROS node (*rosout*).

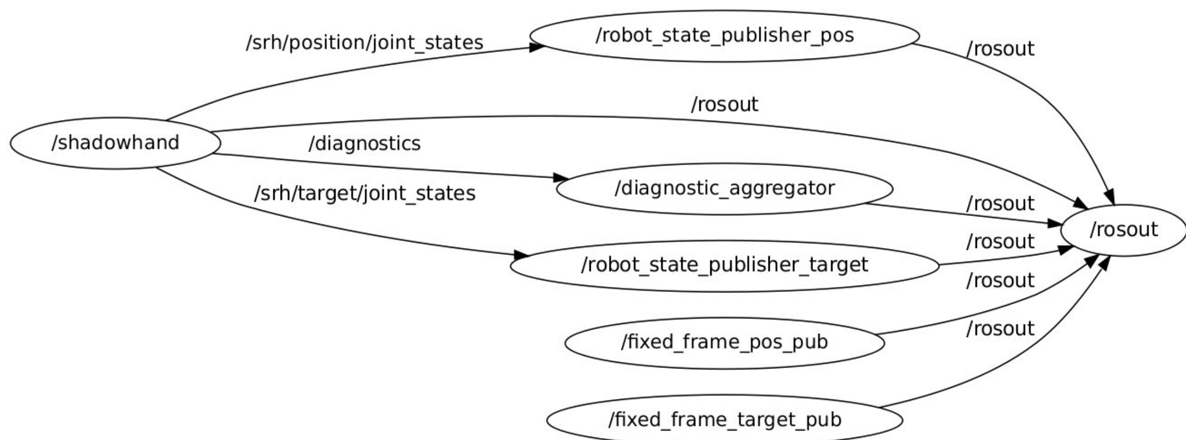


Figure 2.6: Graphical nodes representation

Topics

Topics are the places which nodes use to exchange *messages*. In Figure 2.6 the name on the arrows are the exchanged messages.

Every device that wants to be public (or that wants to publish something) has to create a topic. This topic behaves like a mailbox; when a device has a new sensor input, it sends it to the mailbox. On the other hand, each program that wants to know what happens with some device has to be subscribed to this topic (the mailbox). In this way, if the topic receives new data, it sends a signal to their subscribers; and they actualize their values.

This system of publisher-subscriber is not enough for some tasks than need a feedback, and for theses specific tasks the *service* functionality should be utilized. For that tasks is the **service**.

2.1.4 Applications, sensors and more

Very much researches use the open source system to share their publications and their programs. This behavior can be exploited only if there is a system to reuse and share the code. And ROS is this system. The way in which ROS is based helps to create a file sharing system. With ROS is possible to create an independent code to move an arm or to read a sensor. This code could be used for any robot that has to read a sensor or that has a similar arm.

ROS dependencies are a large ecosystem with several software packages.

In this place it is possible to find both, whole systems to move a given robot and drivers for specific applications, sensors or actuators. Some of these useful "whole system" packages are:

- **Shared Control for a Quadrotor Platform**, which is used to facilitate inspection of vertical structures using Quadrotors. It uses simple controls related to the structure in order to simplify the operators commands.
- **AscTec drivers** provide autopilot tools to manage a UAV.
- **Kinton apps**, which uses the Pelican Quadrotor from AscTec to the project ARCAS (Aerial Robotics Cooperative Assembly System). In this project, they use SLAM and recognition techniques for path planning and cooperative aerial manipulation.
- **Starmac ROS package** is the software used in the STARMAC project.
- **Hector packages**, which will be explained later in the Section 2.3.

Moreover, there are other packages that provide just one, or some, functionalities as:

- **PR2 Teleop**. Although PR2 is a complete software package for one robot, individual packets that make up its stack are quite useful and easy to use separately. PR2 Teleop is a package that provides some tools to control the *cmd_vel* topic, which is generic topic used by ROS to control the robot speed with 6 parameters (3 linear speeds and 3 angular speeds), from a joystick, a game pad or just a keyboard. It is also easily reprogrammable.

- **Humanoid localization** can be used on UAVs for localization of objects in a known map in order to avoid obstacles in defined routes.

- **Quadrotor Altitude Control and Obstacle Avoidance** is a software that makes use of a Kinect camera on board to perform its task.

- **Laser Drivers** is a stack that contains drivers for laser range finders, including Hokuyo SCIP 2.0-compliant and SICK models. It publishes the *scan* topic, which it will be utilized in Section 3.2.2.

- **USB Cam** is a stack that provides drivers for most standard USB camera. There are many of these generic drivers to fit perfectly with a camera system.

Figure 2.7: ROS terminal running several programs at the same time

In general there are many drivers and packages for a variety of applications, and thus a detailed search needs initially to be performed http://www.ros.org/

Since one of the advantages of ROS is its ability to launch many nodes in parallel (as you can see in Figure 2.7), you can launch these packages to control a specific device of your robot. After that you have to launch your own code collecting all the topics you need to operate the robot.

In the example of Figure 2.7, in the first row, there is a node with RosCopter [22] (connecting the ROS system with ArduCopter), there is a terminal window to read the */gps* parameter (rostopic echo /gps), the other two terminal windows have services to start or finish the service */arm*. In the second row, there are a *roscore* (in order to start the ROS system), two topic readers (*/rc* and */vfr_hub*), one python program (sending rotation commands) and one topic publisher (overwriting the rc channel).

2.1.5 Setup

For setting up ROS, the first thing to do is to stabilize a connection between machines. Then, master and host name have to be set in order to each machine know their role.

In order to be possible to utilize ROS, a *roscore* has to be started. This is a collection of nodes and programs that are needed for a ROS-based system. It also will start up a ROS master, that will manage the links between nodes monitoring publishers and subscribers to topics and services. Once these nodes are located each other, they communicate mutually peer-to-peer. Moreover, a parameter server will be utilized, that will be a share space to store and recover parameters at runtime, and a *rosout* logging node.

In cases where the ROS program runs over several machines, you should start a *roscore* just in one of them, which will be in the master machine.

Once the system is running, it is possible to launch a program, node, script or whatever in needed.

2.2 First tests to understand the platform

The first ROS learning test has been carried out with a **TurtleBot** (in Figure 2.8), this is a ground robot, Roomba-based, that thanks to a Kinect camera can sense and interact with the environment. This robot mounts ROS on a laptop located on it. Other Ubuntu laptop is used as a workstation machine. The robot is used to understand how the ROS nodes system works and to start working with a ground robot, always simpler in learning tasks.

The first step is to establish communication between both machines (the workstation and the robot) in order to create a network in the whole ROS system. For this, it is used the ***slattach*** command to create a network point with the serial port that the XBee is using.

Figure 2.8: TurtleBot v1.5

```
1 sudo slattach -s 57600 /dev/ttyUSB0
```

Then, it is created a connection peer-to-peer between the IPs as:

```
1 sudo ifconfig sl0 10.0.0.1 pointopoint 10.0.0.2 netmask 255.255.255.0
```

Of course, the port in the first command have to be the same port where the XBee has been connected and the IPs in the second command have to be, first, the machine where the command has been written and, second, the other machine. These commands have to be executed in both machines.

The baudrate selected in the first command can be chosen by the operator (between 1200 and 115200, even more if a patched *slattach* command is used). The only limitation is that the rate has to be the same in both ZigBee radios. In this case, and because there aren't any problems with the batteries, the higher rate without problems (57600) will be set.

All these processes can be set automatically, as it follows, if some of the PCs haven't screen writing this code in the */etc/network/interfaces*, then the network interface will be set up each time that you start the quadrotor.

```
auto sl0
iface sl0 inet static address 10.0.0.2
pointopoint 10.0.0.1
netmask 255.255.255.0 mtu 512 pre-up /usr/local/bin/slattach -s 57600 -p slip /dev/
    ttyUSB0 & pre-up sleep 1 post-down kill ‘cat /var/lock/LCK..ttyUSB0‘
```

When the network is properly configured it is possible to use the ***ssh*** command to use the terminal on the other machine.

```
ssh machine-name@machine-IP
```

The second step is to configure the master and the host name in every machine. Then you can open an instance of *roscore* in the master, start the services that you need and start the programs.

The first experimentation has been performed by evaluating the SLAM code, while the commands have been:

```
Terminal 1 (master):
roscore

Terminal 2 (bot):
sudo service turtlebot start

Terminal 3 (master):
rosrun turtlebot_dashboard turtlebot_dashboard&

Terminal 2 (bot):
roslaunch turtlebot_navigation gmapping_demo.launch

Terminal 3 (master):
rosrun rviz rviz -d ‘rospack find turtlebot_navigation‘/nav_rviz.vcg

Terminal 4 (bot):
roslaunch turtlebot_teleop keyboard_teleop.launch

Move in the SLAM area

Terminal 4 (bot): (first kill the process with Ctrl+C)
rosrun map_server map_saver -f /tmp/my_map
```

If you try to do a lot of turns, quick acceleration and intense breaking the odometry losses its location and the results are presented in the next SLAM map (lab room map) in Figure 2.9.

Figure 2.9: Bad map created by TurtleBot

In case that the previous experimentation is being repeated by driving the robot in a straight line, better results can be achieved and with a greater accuracy, as it can be presented in Figure 2.10 which is a corridor map at LTU.

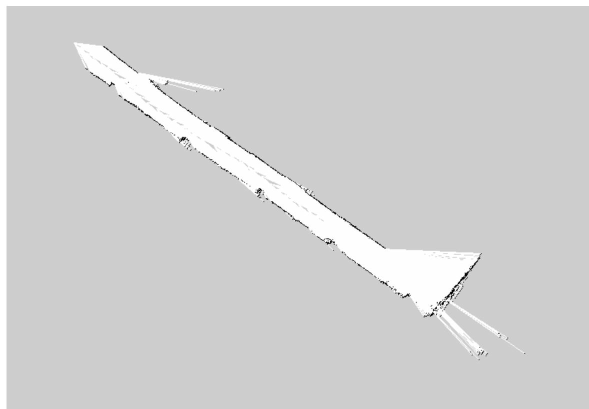


Figure 2.10: Good map created by TurtleBot

2.3 Evaluating ROS: Hector Quadrotor package

Hector [23] is a collection of ROS stacks (a **stack** is a collection of packages that provides a functionality) originally developed by the Technische Universität Darmstadt (Germany). These stacks supply several tools to simulate or to interact with robots. SLAM, location or modelling are some of them.

In this collection, the hector_quadrotor stack is going to be utilized with that tool, it is going to be able to moderate (with the physical parameters) a 3D quadrotor model (presented in Figure 2.11) in Gazebo and to have a 3D reconstruction of the world with a SLAM tool.

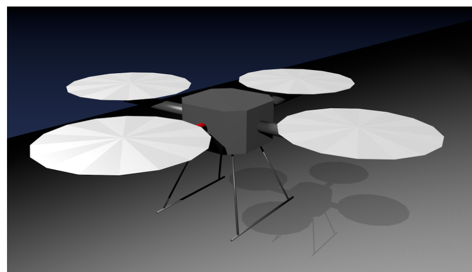


Figure 2.11: Hector quadrotor render

In Figure 2.12 you can see the Gazebo representation (on the left in the figure), as well as the world representation provides by Rviz (on the right in the figure). Gazebo is a multi-robot simulator which generates both realistic sensor feedback and physically plausible interactions between objects (it includes an accurate simulation of rigid-body physics). While Rviz is a 3D visualization environment that is part of the ROS visualization stack. On this figure it has been also presented a vision range representation (the cone on the left).

For utilizing the Hector platform, the first thing to do is to start the world.

```
1 roslaunch hector_quadrotor_demo indoor_slam_gazebo.launch
```

Then, and thanks to the easy access and interoperability between nodes, it is possible to use private code to take the control using the topic "*cmd_vel*" to control the quadrotor speed.

```
1 rosrun hector_quadrotor_controller simulation
```

In a first stage, and through that platform, a simulator has been developed to try the produced controllers and for a better understanding of the quadrotor behaviour.

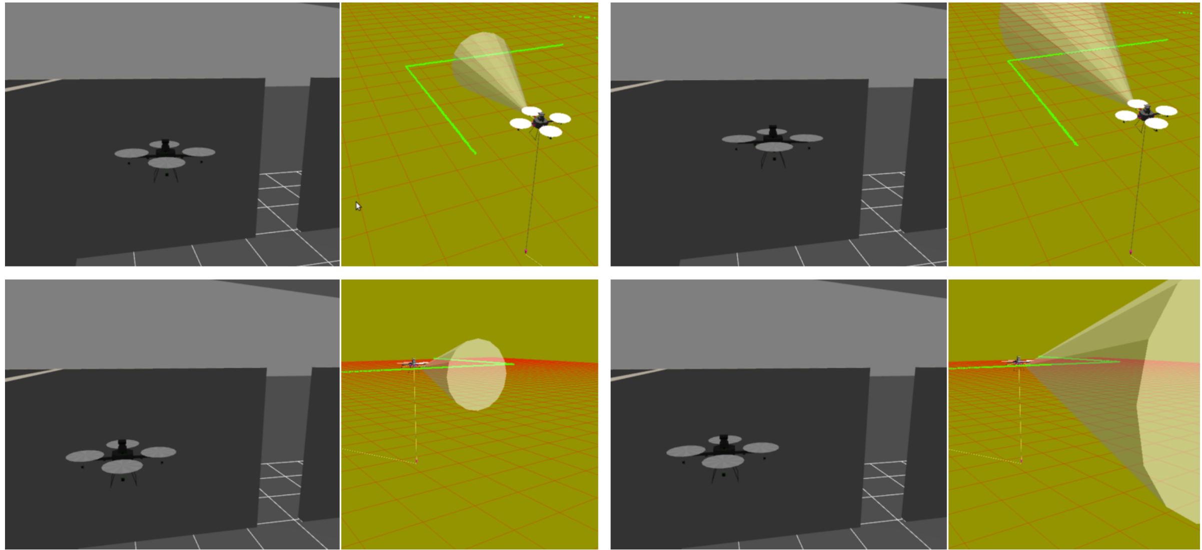


Figure 2.12: Hector quadrotor simulation

For this purpose, a control code has been developed to carry out the way point navigation and all the positioning codes. With this code you have to set some goals (as many as are needed), the precision and the speed. You can edit the proportional, integrative and derivative constants too. For the moment, all of these parameters needs to be set in the code file (cpp) and a *rosmake* needs to be performed. It is also possible to modify every parameter at running time (but then it is not possible to modify the number of goals) when the program is being executed with this structure.

```
1   rosrun package program parameter:=new_parameter
```

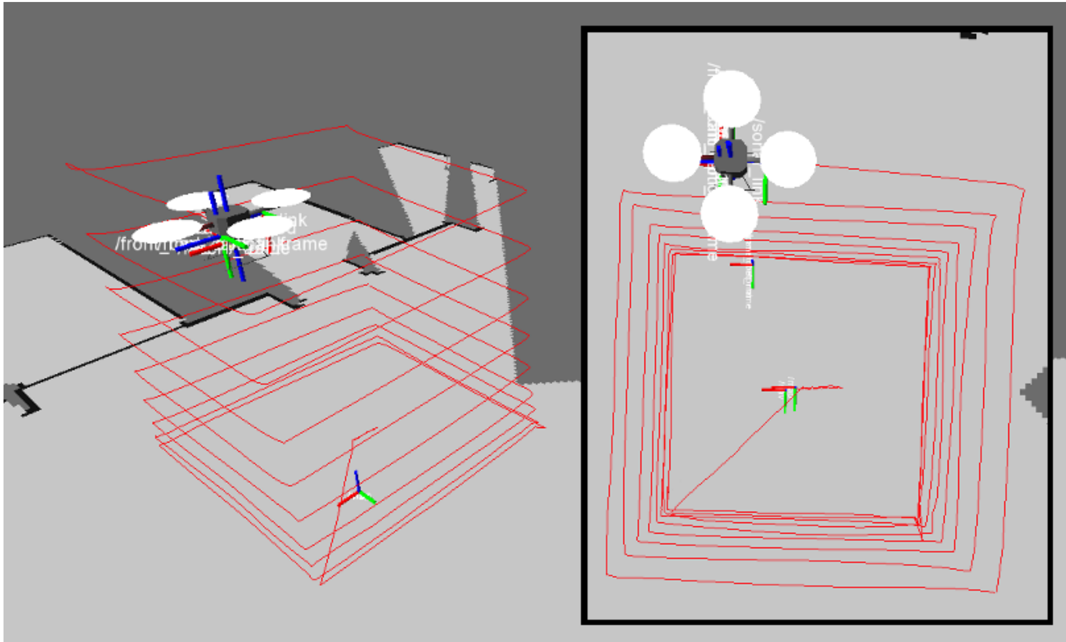


Figure 2.13: Hector quadrotor simulation

This controller can move the quadrotor in X, Y or Z speed direction, it also controls that orientation to be always stable and North. For performing this simulation, the complete code (in C++) has been included in the Appendix 1.2.1.

In Figure 2.13 the simulated quadrotor following an almost perfect cube (the red line is the path followed) is being presented. For this goal, the PID constants to have a stable system have been set as the equation 2.1 shows.

$$\kappa_p = 0.5 \qquad \kappa_i = 0.0002 \qquad \kappa_d = 0.00005 \tag{2.1}$$

The next simulation test was to try to control the simulator movement with the real quadrotor while performing true interoperability between platforms and networks.

This simulator took the Euler angle outputs directly from the experimental quadrotor platform to show the same movement in the virtual one. The Z axis can be rotated to rotate in the simulation, the X axis to advance in Y direction or Y axis to advance in X direction. Just as it would move if the quadrotor motors roll, pitch or yaw in these directions.

In order to achieve this we need to use a controller to move the virtual quadrotor from the original position (the virtual position) to the goal position (the real position) similar as before.

For these tests,the experimental frame has been utilized (in Figure 2.14) and an IMU, the k-Fly, which has been designed and printed by Emil Fresk. This frame has been printed in our lab with a handmade 3D printer, and it has been tested its strength to axial tension and bending.

The code (in C++) for this second simulation can be located in Appendix 1.2.2.

Figure 2.14: Prototype quadrotor frame

CHAPTER 3

Building the Quadrotor

3.1 Hardware

3.1.1 ArduCopter

ArduCopter (presented in Figure 3.1) is an easy to setup and easy to fly platform for multirotors and helicopters, it is a basic RC multicopter which can be found in the today market. Thanks to Arduino and the great community that both have it is possible to find lots of modifications to the platform which can be easily adapted to specific applications.

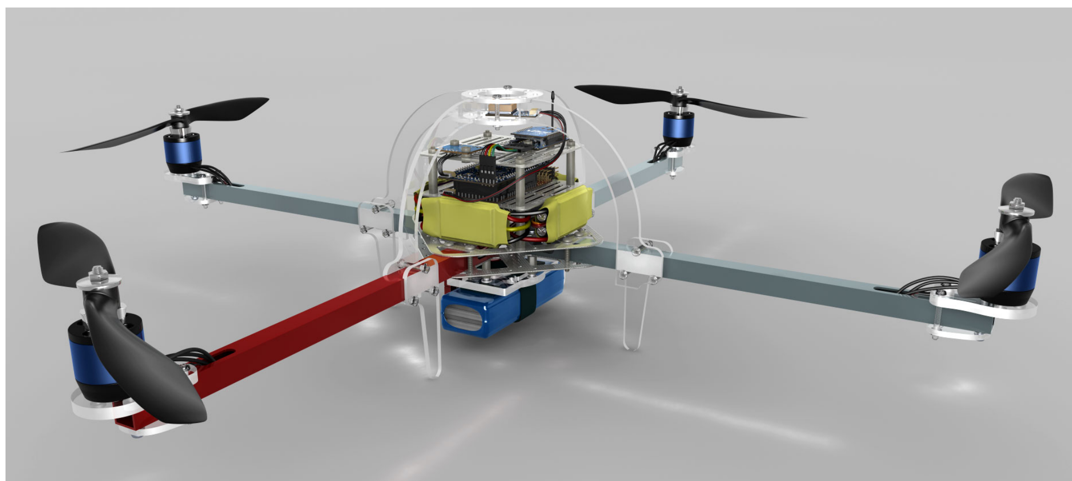


Figure 3.1: Arducopter quadrotor

ArduCopter is mainly a modified Arduino board (ArduPilot Mega) with an IMU board

attached. The IMU (inertial measurement unit), that provides 3 accelerometer and 3 gyroscope directions, also has an attached magnetometer (with 3 degrees of freedom) and an XBee unit to support wireless communication. Moreover, this board can provide a quaternion [24] system with which you can calculate an Euler angle [25] system and measure how the quadrotor is positioned. A GPS attached also to the ArduPilot Mega board is used to estimate the 3D position of the quadrotor. Finally, there is a power distribution board controlled by the ArduPilot Mega board, this low level board controls the ESC (Electronic Speed Controller) of the motors. In Figure 3.2 there is a scheme of this parts.

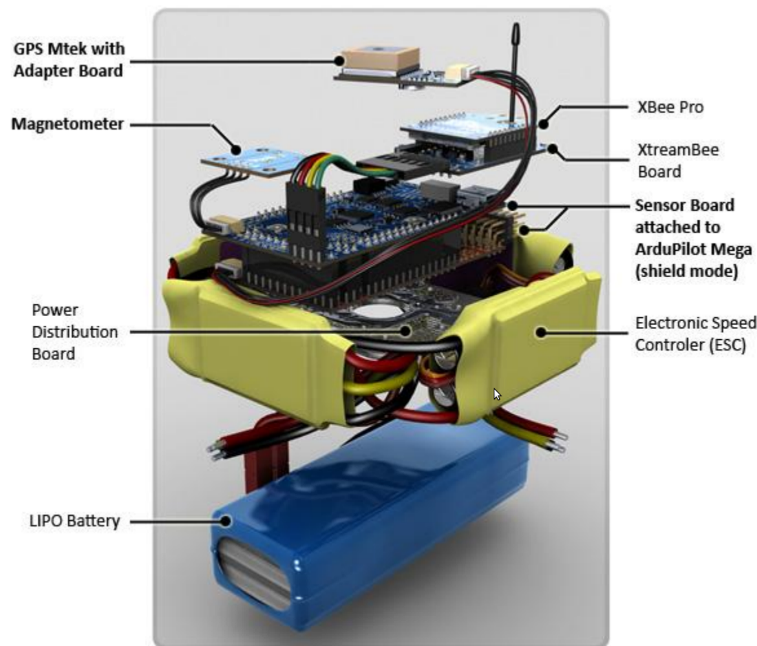


Figure 3.2: ArduCopter components

One of the most important parts in the quadrotor are the propellers, how they are mounted and how they can give lift to the quadrotor.

The flying stabilization in that kind of flying machine is achieved by two opposite forces [26]. The force that pushes the UAVs down is the gravity, an opposite force needs to be created of same magnitude to get hold the vehicle flying. That force is generated by the propellers that create a low pressure area over the top surface of them.

These propellers are able to change the quadrotor angles. In Figure 3.3 (a) a sketch of the quadrotor structure is presented in black. The Ω symbol represents the angular speed, and the Δ symbol represents the increase in speed. $\ddot{\Phi}$, $\ddot{\Theta}$ and $\ddot{\Psi}$ represent the angular acceleration created in that moment in X, Y and Z directions. The fixed-

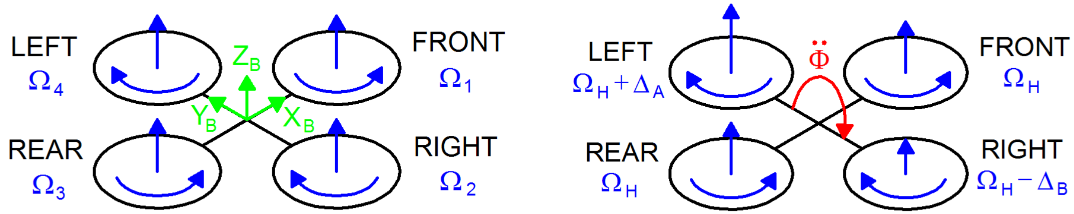


Figure 3.3: a) Simplified quadrotor motor in hovering (left) and b) Roll movement (right)

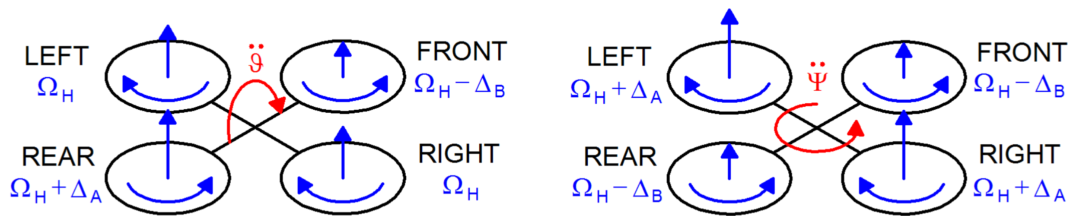


Figure 3.4: a) Pitch movement (left) and b) Yaw movement (right)

body B-frame is shown in green and in blue is represented the angular speed of the propellers. Roll movement is provided by increasing (or decreasing) the left propeller speed and by decreasing (or increasing) the right one. It leads to a torque with respect to the X_B axis which makes the quadrotor turn. Figure 3.3 (b) shows the roll movement on a quadrotor sketch. Pitch movement is very similar to the roll and is provided by increasing (or decreasing) the rear propeller speed and by decreasing (or increasing) the front one. It leads to a torque with respect to the Y_B axis which makes the quadrotor turn. Figure 3.4 (a) shows the pitch movement on a quadrotor sketch. The Yaw moment is provided by increasing (or decreasing) the front-rear propellers' speed and by decreasing (or increasing) that of the left-right couple. It leads to a torque with respect to the Z_B axis which makes the quadrotor turn. The yaw movement is generated thanks to the fact that the left-right propellers rotate clockwise, while the front-rear ones rotate counter-clockwise. Hence, when the overall torque is unbalanced, the helicopter turns on itself around Z_B. Figure 3.4 (b) shows the yaw movement on a quadrotor sketch. [27]

Working with ArduCopter

In this thesis we will use a default ArduCopter quadcopter that will be connected with a ground workstation through XBee [28]. The MAVlink [29] serial connection will provide data about attitude (Euler angles and Euler angle speeds), vfr_hub (ground speed, air speed, altitude and climb rate), GPS, RC (the values of the 8 radio frequency channels).

This serial connection also provides heartbeat (systems can use this message to track if the system is alive); navigation controller output (with information about goals and errors); hardware status (board voltage); raw IMU (acceleration, gyroscope and compass values); pressure; and some more parameters which have not been used in this system, while the same connection will be used to send the required throttle, and the euler angles.

In Figure 3.5 it is shown that it is possible to create a network between several aerial robots and ground stations using XBee devices.

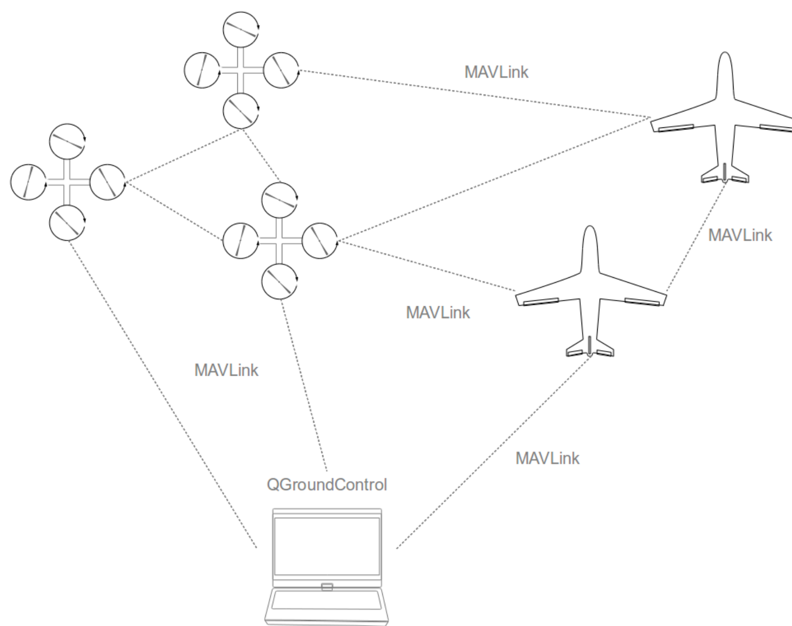


Figure 3.5: MAVlink connection between flying machines and a ground station [30]

3.1.2 XBee

XBee (in Figure 3.6) Digi International radio module is being displayed that uses the ZigBee protocol (it is based on an IEEE 802 standard).

For this thesis, it is needed a device with a low consumption. The normal flying time for this kind of UAVs usually is lower than 15 minutes, so any extension in the autonomy is a big improvement as is also needed a long working range. The quadrotor can fly autonomously, but it is usually good to keep the telemetry running to see what happened in the air and to overcome some minor problems.

Therefore, an XBee Pro 60mW Wire Antenna - Series 1 (802.15.4) has been selected. It is a cheap (less that $40) and reliable device with more than 1 km connectivity range

Figure 3.6: XBee module

and a low consumption.

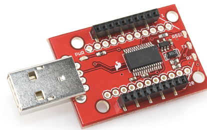


Figure 3.7: XBee Explorer

For the interface of this XBee with the workstation an *XBee Explorer Dongle* (in Figure 3.7) has been selected, while for the interface with the ArduCopter the selection has been an *XBee Explorer Regulated* through the UART0 port.

3.1.3 Positioning Systems

Figure 3.8: GPS module

A MTek GPS plus an adapter board (in Figure 3.8) has been used to control the quadcopter outdoor with a sufficient accuracy (about 3 meters) and a good stability.

This device along with a barometer, that uses the pressure to know the altitude and a ultrasonic sensor (LV-EZ0) allows to have a good position control in every moment.

The GPS and the barometer are used for the absolute position, while the sonar is used for a relative position. This helps the platform to take off and landing autonomously without any external help [31].

3.1.4 Remote Control

A Futaba Remote Control is used in case of emergency. If something happens and the quadrotor looses the control, the operator can switch to manual control just changing by the mode in the remote control, or using another mode, is possible to perform automatic emergency landing (the code in the Appendix 1.3.2). By setting a specific height, the quadrotor can go to height zero, progressively reducing the motors power and using intermediate height way points to stabilize.

3.2 Code Architecture and Evaluations in Real Tests

3.2.1 Interface

The interface between ArduCopter and ROS is a python code to connect the workstation machine to ArduCopter using XBee and then to create the topics and services to establish good communication. The code can be found in Appendix A.1.

In that code it is created a connexion between the XBee in the ArduCopter and the XBee in the workstation using MAVlink protocol. It starts the publishers for the GPS, RC (radio control signal), state (armed, GPS enable and current mode), vfr_hub (air speed, ground speed, orientation, throttle, altitude, and climb rate), attitude (Euler angles and speeds) and raw_IMU and the subscribers for the send_rc (it will be the topic used to send the commands). It also starts the services for arm and disarm the quadrotor.

Now, in the main it is created the node ("roscopter") and it is stored the MAVlink message with the function *recv_match*. The message is checked, and it is published in the right topic.

On the other hand, each time a message is sent to *send_rc*, a callback is launched to send the data to the ArduCopter.

3.2.2 Simulation

Figure 3.9 block diagram is being presented, which represents the overall simulation that combines the virtual quadrotor with the real one, situation described in Section 2.3. The pure simulation diagram would be just removing from the "*/attitude*" topic to the right.

In that case, the control is managed by the *pr2_teleop* package. That package uses the keyboard interruptions to modify the *cmd_vel* topic, which controls the speed in almost all ROS robots.

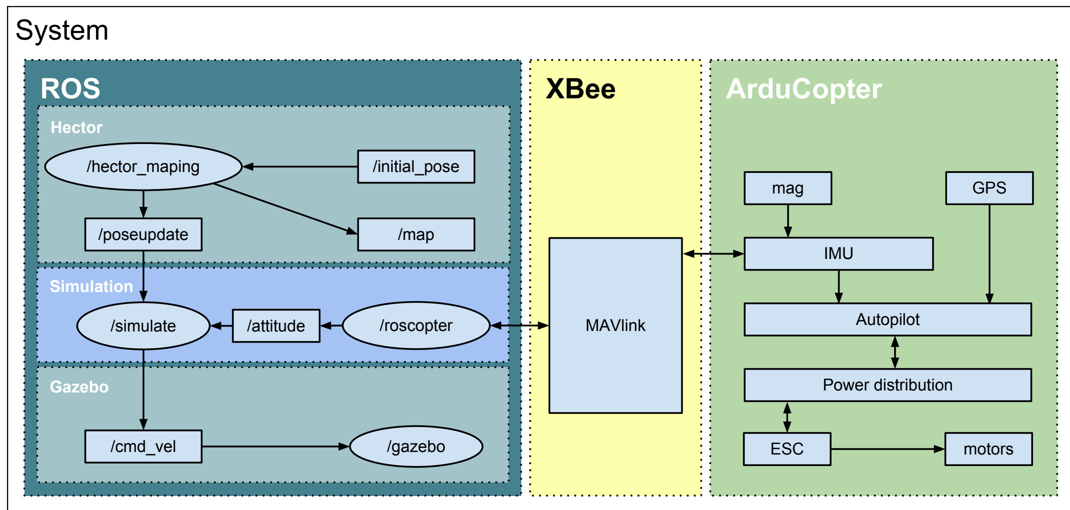


Figure 3.9: Simulation nodes diagram

In the ROS module, the blue one, circles are programs (or nodes) and squares are topics. That module uses three basic programs to carry out the task. In the middle area, the simulate node is the responsible of managing the controller. It reads data from the ArduCopter (through the RosCopter node) and from Hector quadrotor and actuates in the quadrotor speed command (roll, pitch and yaw speed).

In the upper part, the hector_maping node creates the world and manages the SLAM map that the UAV is seeing. It has the physical parameters to shape the robot and it places the robot in Gazebo. Hector uses the */poseupdate* topic to share the position in the 3 axes and the quaternion. Gazebo and hector_maping are also connected with a */scan* topic (omitted in the picture to improve the clarity). With this topic, hector knows how the world is in order to draw its SLAM map. Hector also needs a */initial_pose*. This allows the utilization of any kind of map without problems. This initial pose has to be a position without any other object.

On the other hand, in the bottom, **Gazebo** manages the physics (with the provided parameters) in the simulation and visualizes the quadrotor in a 3D map.

The simulation code can be located in the Appendix 1.2.2.

In this program, every active part is in the callbacks. There are two of them. The first one is launched when there is an update on the topic "*attitude*" that stores the euler angles and their velocities. The second one is launched when the update happens on the

topic "*poseupdate*" that stores the simulator quaternion.

The "poseupdate" callback transforms the quaternion to the roll, pitch, yaw euler angles and stores them in global variables. The "attitude" callback is in charge of controlling.

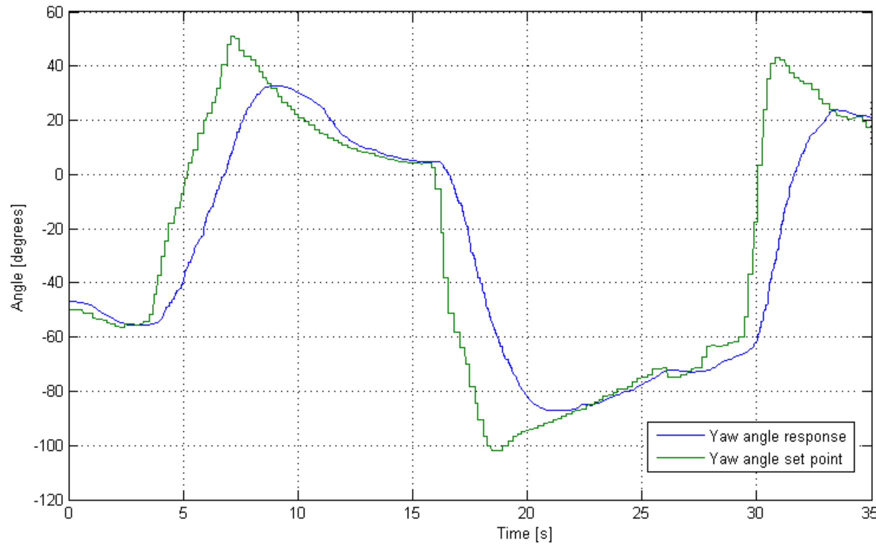


Figure 3.10: Response of manual rotation in simulation

In Figure 3.10 you can see how the yaw angle in the virtual quadrotor changes when the real quadrotor angle is changed. It cannot be a good representation of how the control is because is hard to have a stable input signal (the goal is by hand), but you can see a fast response and a perfect tracking.

3.2.3 Flying code

Different programs have been developed to move the quadrotor and all of them can be called with different parameters in the function *go* (the code (in python) in Appendix 1.3.1). In this way, it is possible to navigate with way points, to rotate to a specific angle or to go to a specific altitude. There is also a safety code for the take off or landing of the quadrotor.

Figure 3.11 is a representation of how the quadrotor performs its task.

It consists of three modules. The first one, red in the picture (left), is the ROS system that has sensor measurement (global position, euler angles, RC channels and the state) and can actuate on the roll and pitch angles and the yaw speed through the RC parameters. It uses two programs to perform its task. The main one is */go*. It is the control

program and it is used to close the loop. Instead, */roscopter* is just a interface (as it has been presented in Section 3.2.1), it bridges both platforms taking messages from one side and leaving to the other side.

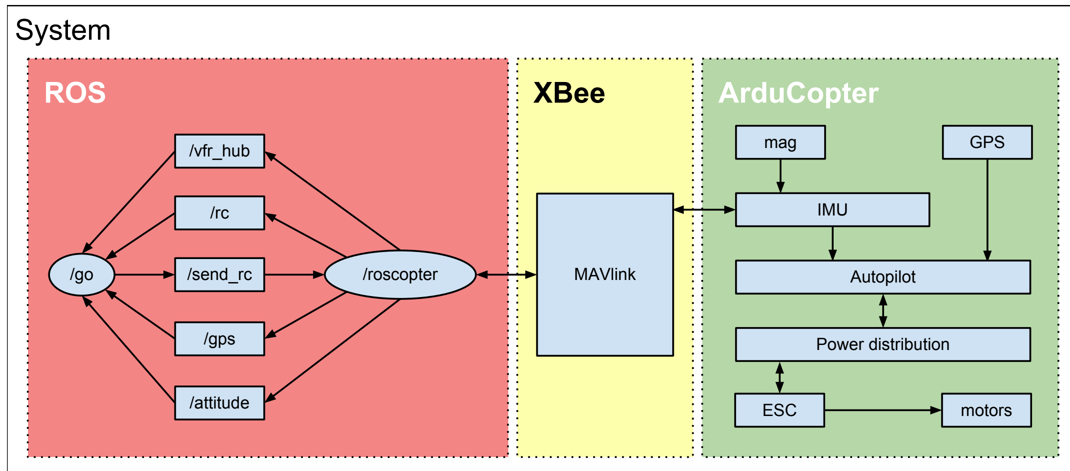


Figure 3.11: Control nodes diagram

The second one, in yellow (middle), is a *MAVlink* interface that is responsible for taking data in both sides and send them to the other. Between */roscopter* and *MAVlink* are carrying out the task of converting ArduCopter data into ROS data.

The third part, in green (right), is the ArduCopter system. It has an autopilot module, it is the smart device on board, which can manage simple control tasks and it has the sensor measurements and can control the movement of the motors. It receives data from the IMU and the GPS inputs. It has also an inertial measurement unit and receiving the data from a external magnetometer. Finally, the ArduCopter system mounts a power distribution board controlled by the autopilot module. It can give the command signals to the motors' controllers.

3.3 Controller

We will use the ROS platform to perform the control, because topics and nodes are good tools to have a discrete control. We don't need to create a periodical loop nor a discretization because the data is sent periodically from ArduPilot to the ROS system, through MAVlink. This means that the ROS program will launch the topic update callbacks periodically too (when a new message appear in the topic).

Several controllers have been tested to get a quick response, avoiding as much as possible overshooting, and achieve a fast settlement.

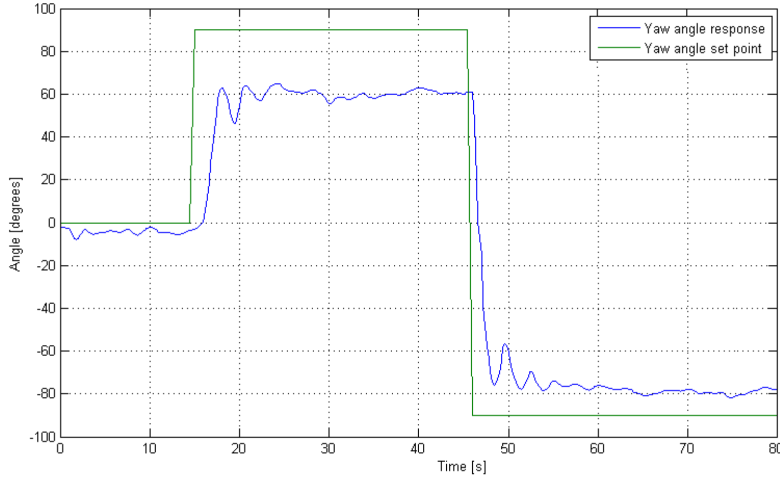


Figure 3.12: Step response of rotation in [90,-90] range

In the first attempt a proportional control was used to test the similarity between the real performance of the quadrotor and the performance in the simulation. However, the test results show a non-unity gain and an offset in the response as can be seen in Figure 3.12. This is due to the appearance of external perturbations and the speed threshold of the motors. Here, we can see the angle never reach the desired value as the motors have no enough velocity to start the movement. It is also possible to see a bigger error in the positive zone caused by a little imbalance in zero speed.

With some adjustments more in the proportional constant is possible to save most of the speed thresholds but still do not have a unity gain (in Figure 3.13), and thus a PID controller will be applied.

So the second attempt was using a PID controller [32]. With this kind of controller you can add the cumulative error in the action (with the integrative part). The longer the UAV is within an error, the biggest is the action. And you can add the future behaviour for the error in the action using the rate of change of the error (with the derivative part). Finally, the following constants (equation 3.1) have produced an acceptable behaviour.

$$\kappa_p = 2 \qquad \kappa_i = 0.1 \qquad \kappa_d = 0.07 \tag{3.1}$$

As it is possible to see now in Figure 3.14, if the change is not too much (less than 120°), the response is perfect. It uses 3 seconds to reach the goal with 10% of error. After 16 seconds the error oscillate between 2 and 3 %.

If the change is higher (at second 50) it has some overshooting (about 50 degrees). Now it uses 10 seconds to reach the goal with 10% of error. After 2 seconds more, the

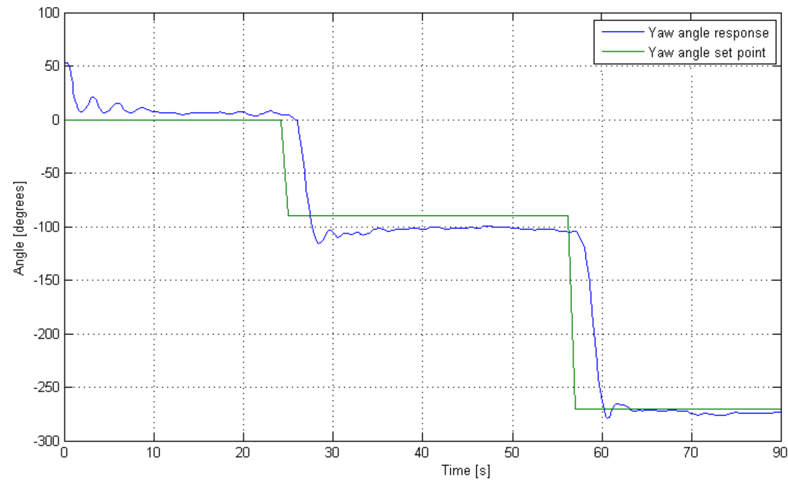


Figure 3.13: Step response of rotation in [90,-90] range with a bigger constant

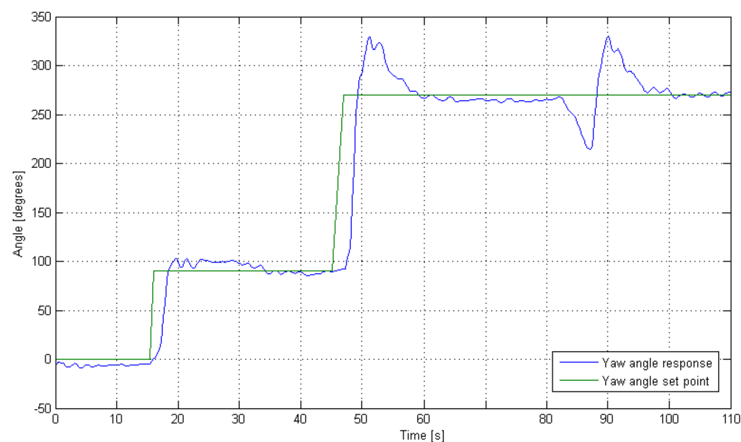


Figure 3.14: Step response and perturbation of rotation in [90,-90] with a PID

error is lower that 3%.

Furthermore, this system can correct the perturbations quite soon (there was a manual quadcopter rotation at second 83).

Algorithm 1: Control algorithm

read data
store prev_error
calculate error: goal - data
calculate proportional action: κ_p ** error*
calculate integral: prev_integral + κ_i ** error*
calculate derivative: κ_d ** (error - prev_error)*
calculate action: Proportional + Integral + Derivative
ajust the action to the rc level
publish the action

In the presented codes, as it has been stated, topic callbacks are used in order to carry out the control instead of loops. So the Algorithm 1 is launched each time that "attitude" callback has an update in its topic.

That algorithm uses three correcting terms, whose sum is the control action. The **proportional** action produces an absolute change in the energy of the system, which modifies the response time.

The **integral** action helps to remove the residual steady-state error. It maintains a count of the cumulative error, and it adds to the action in order that the slightest errors can be eliminated. The **derivative** action predicts system behavior and thus improves settling time and stability of the system.

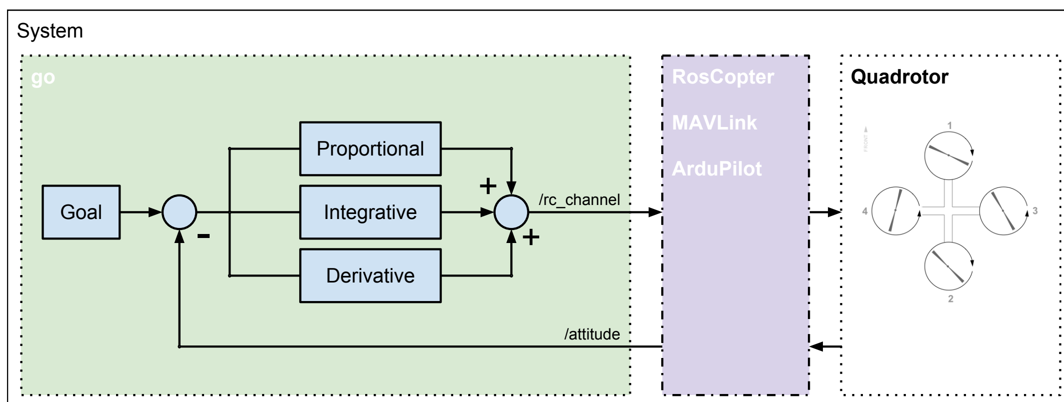


Figure 3.15: Control diagram

In the case of the waypoint control, the roll controls the Y coordinate, the pitch controls the X coordinate and the throttle controls the Z coordinate. The implementation of this

controller is presented in Figure 3.15. In this representation, the interfaces have been simplified, since the focus has been provided in the control code *go*.

CHAPTER 4

Future work

The work you can do in six months is limited, and therefore the work that lies ahead is enormous.

After the waypoint navigation codes and, derived from it, paths generated navigation, will allow us to trace routes of action for SLAM.

Having ensured the proper operation of navigation codes would have to start working with SLAM tasks. The hardware for this purpose would be a USB camera connected to the onboard computer. By reconstruction software that take advantage of the quadrotor own motion for a stereoscopic image.

This reconstruction would also used for route planning and obstacle avoidance.

Future research will address how the robot performs its task. While perform its task completely in local (on-board computer without help from the ground station) is the option that seems more complete. Would be equally satisfactory options. One alternative would be the remote processing of data sending the images to the ground station. It would perform the 3D reconstruction and would send the new routes to the quadrotor. The second one would be to record such images for off-line 3D reconstruction for use as a map on other platforms.

This two alternatives could help in the quadrotor consumption. In this case, you could avoid the onboard computer. The main problem of this course is that the ground station and the quadrotor would have to have always a stable connection.

Chapter 5

Conclusion

The aim of this Master Thesis was to design and develop a quadrotor that will have the merit to operate under a fully ROS enabled software environment. As it has been presented in the Thesis, based on an existing flying frame, a quadrotor has been designed and integrated, where all the necessary hardware components, including the motor controllers, the IMU, the wireless communication link based on the xBees and the GPS have been properly adjusted and tuned for allowing the proper flight capabilities of the platform. Except from the first necessary stage of developing the flying platform, the main focus has been provided in the design and implementation of the necessary ROS enabled software platform, that would be able to connect all the sub-components and support novel ROS enabled control algorithms for allowing the autonomous or semi-autonomous flying.

As it has been indicated in the introduction, there are already some existing ROS enabled software implementations for quadrotors but these software implementations are platform depended and they aim mainly in performing a tele-operation task, which means that a user should always fly the quadrotor from a distance. The results of this thesis have been presented a novel approach where the quadrotor is able to perform autonomous tasks, such as: attitude and altitude stabilisation in a fully ROS enabled environment. The results have been performed in both simulation and extended experimental studies.

The software developments that have been performed have covered various software developing aspects, such as: operating system management in different robotic platforms, the study of the various forms of programming in the ROS environment, the evaluation of software building alternatives, the development of the ROS interface and finally various practical software implementation issues with the developed aerial platform.

Upon completion of this project, two different configurations (aluminium and plastic) of the quadrotor have been delivered. The ArduCopter has been utilised and integrated in the ROS platform while as it has been presented in the thesis, a series of codes have been designed for the project, ranging from establishing the communication with the ground station to the way points following, with the most important to be the

software implementation of the ROS enabled PID controllers for the attitude and altitude stabilisation of the quadrotor.

Although the tests in a real environment have not had all the desirable extent due to the luck of having a full controllable laboratory environment, the developed software architecture for the attitude and altitude stabilisation of the quadrotor have been extended evaluated and the obtained experimental results have proven the overall feasibility of the novel ROS enabled control structure implementation and the achievement of a acceptable flying performance for the quadrotor.

The potential of such autonomous quadrotors is enormous, and it can be even bigger when the size of these devices gets miniaturized, with a relative decrease in the overall hardware cost. For these quadrotors, a proper real time and modular operating system is needed in order to extend the capabilities of these platforms in higher standards. With this thesis it has been experimentally demonstrated that ROS is a proper operating system to manage all the requirements that might arise, while it has the unique merit of being an open source software, allowing the sharing of drivers and programs among the developer community, for speeding the developing phase.

Glosario

APM AutoPilot Mega

AUV Autonomous Unmanned Vehicle (Vehículo Autónomo no Tripulado)

BSD Berkeley Software Distribution (Distribución de Software Berkeley)

Δ Cantidad de Incremento

ESC Electronic Speed Controller (Controlador Electrónico de Velocidad)

GPS Global Positioning System (Sistema de Posicionamiento Global)

IMU Inertial Measurement Unit (Unidad de Medida Inercial)

IP Internet Protocol (Protocolo de Internet)

κ_p Constante proporcional usada en el control

κ_i Constante integrativa usada en el control

κ_d Constante derivativa usada en el control

Ω Velocidad angular

$\ddot{\Phi}$ Velocidad angular en dirección X

$\ddot{\Psi}$ Velocidad angular en dirección Z

PID Proportional-Integral-Derivative Controller (Control Proporcional-Integral-Derivativo)

p2p Peer-to-peer (Red de pares o red punto a punto)

UAV Unmanned Aerial Vehicle (Vehículo Aéreo no Tripulado)

UWV Unmanned Water Vehicle (Vehículo Acuático no Tripulado)

RC Remote Control (Control Remoto)

ROS Robotic Operating System (Sistema Operativo para Robots)

SLAM Simultaneous localization and mapping (Localización Y Mapeado Simultáneos)

$\ddot{\Theta}$ Velocidad angular en dirección Y

BIBLIOGRAFÍA

[1] E. Fresk, “KFly.” Retrieved from http://kflyproject.blogspot.com.es/.

[2] W. Schmidt, *Pneumatica et Automata: Heron Alexandrinus ; accedunt Heronis fragmentum de horoscopiis aquariis, Philonis De ingeniis spiritualibus, Vitruvii capita quaedam ad pneumatica pertinentia ; recensuit Guilelmus Schmidt*. Bibliotheca scriptorum Graecorum et Romanorum Teubneriana, B.G. Teubner, 1899.

[3] S. Nof, *Handbook of Industrial Robotics*. No. v. 1 in Electrical and electronic engineering, Wiley, 1999.

[4] R. D. Leighty and ARMY ENGINEER TOPOGRAPHIC LABS FORT BELVOIR, *DARPA ALV (Autonomous Land Vehicle) Summary*. Defense Technical Information Center, 1986.

[5] B. Llc, *Robotics at Hond: Asimo, Honda E Series, Humanoid Robotics Project, Honda P Series*. General Books LLC, 2010.

[6] NewEagleWiki, “Unmanned Systems,” http://www.neweagle.net/support/wiki/index.php?title=Unmanned_Systems.

[7] Scaled composites, “Bell Eagle Eye TiltRotor UAV.” Retrieved from http://web.archive.org/web/20070113222345/http://www.scaled.com/projects/eagleye.html.

[8] OLIVE-DRAB, “RQ-7 Shadow UAV.” Retrieved from http://olive-drab.com/idphoto/id_photos_uav_rq7.php.

[9] K. Alexis, G. Nikolakopoulos, A. Tzes, and L. Dritsas, “Coordination of Helicopter UAVs for Aerial Forest-Fire Surveillance,” in *Applications of Intelligent Control to Engineering Systems* (K. Valavanis, ed.), vol. 39 of *Intelligent Systems, Control, and Automation: Science and Engineering*, pp. 169–193, Springer Netherlands, 2009.

[10] V. Kumar, "Vijay Kumar: Robots that fly ... and cooperate [Video file]." Retrieved from http://www.ted.com/talks/lang/es/vijay_kumar_robots_that_fly_and_cooperate.html, Feb. 2012.

[11] F. Caballero, L. Merino, J. Ferruz, and A. Ollero, "Vision-Based Odometry and SLAM for Medium and High Altitude Flying UAVs," *Journal of Intelligent and Robotic Systems*, vol. 54, no. 1-3, pp. 137–161, 2009.

[12] A. Nemra and N. Aouf, "Robust cooperative UAV Visual SLAM," in *Cybernetic Intelligent Systems (CIS), 2010 IEEE 9th International Conference on*, pp. 1–6, 2010.

[13] R. D'Andrea, "Raffaello D'Andrea: The astounding athletic power of quadcopters [Video file]." Retrieved from http://www.ted.com/talks/raffaello_d_andrea_the_astounding_athletic_power_of_quadcopters.html, June 2013.

[14] M. Banzi, D. Cuartielles, T. Igoe, G. Martino, and D. Mellis, "Arduino." Retrieved from http://arduino.cc/.

[15] R. Faludi, *Building Wireless Sensor Networks: with ZigBee, XBee, Arduino, and Processing*. O'Reilly Media, 2010.

[16] W. R. Hamilton, "On Quaternions, or on a New System of Imaginaries in Algebra," *Philosophical Magazine*, vol. 25, no. 3, pp. 489–495, 1844.

[17] L. Euler, "Formulae generales pro translatione quacunque corporum rigidorum," pp. 189–207, 1776. http://math.dartmouth.edu/~euler/docs/originals/E478.pdf.

[18] G. Hoffmann, D. Rajnarayan, S. Waslander, D. Dostal, J. S. Jang, and C. Tomlin, "The Stanford testbed of autonomous rotorcraft for multi agent control (STARMAC)," in *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, vol. 2, pp. 12.E.4–121–10 Vol.2, 2004.

[19] J. Macdonald, R. Leishman, R. Beard, and T. McLain, "Analysis of an Improved IMU-Based Observer for Multirotor Helicopters," *Journal of Intelligent & Robotic Systems*, pp. 1–13, 2013.

[20] L. Meier, "Ground Control Station for small air-land-water autonomous unmanned systems." http://qgroundcontrol.org/about, 2009.

[21] L. Meier, "MAVLink Micro Air Vehicle Communication Protocol." http://qgroundcontrol.org/mavlink/start, 2009.

[22] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[23] G. Grisetti, C. Stachniss, and W. Burgard, "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters," *Robotics, IEEE Transactions on*, vol. 23, no. 1, pp. 34–46, 2007.

[24] N. D. Freitas, "Rao-Blackwellised Particle Filtering for Fault Diagnosis," in *IEEE Aerospace*, pp. 1767–1772, 2001.

[25] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," in *In Proceedings of the 11th International Conference on Advanced Robotics*, pp. 317–323, 2003.

[26] N. Koenig, J. Hsu, M. Dolha, and A. Howard, "Gazebo." Retrieved from `http://gazebosim.org/`.

[27] D. Hershberger, D. Gossow, and J. Faust, "Rviz." Retrieved from `http://www.ros.org/wiki/rviz/`.

[28] Y.-L. Tsai, T.-T. Tu, H. Bae, and P. Chou, "EcoIMU: A Dual Triaxial-Accelerometer Inertial Measurement Unit for Wearable Applications," in *Body Sensor Networks (BSN), 2010 International Conference on*, pp. 207–212, 2010.

[29] H. Goldstein and J. Ferrer, *Mecánica clásica*. Editorial Reverté, S.A., 1987.

[30] pechkas@gmail.com and juancamilog@gmail.com, "ROS interface for Arducopter using Mavlink 1.0 interface." Retrieved from `https://code.google.com/p/roscopter/`.

[31] K. Ogata, *Modern Control Engineering*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 4th ed., 2001.

[32] F. Zhao, "RC Quadrotor Helicopter." `http://www.instructables.com/id/RC-Quadrotor-Helicopter/`.

[33] F. Zhao, "Picopter," `http://www.instructables.com/id/Picopter/`.

[34] H. Lim, J. Park, D. Lee, and H. Kim, "Build Your Own Quadrotor: Open-Source Projects on Unmanned Aerial Vehicles," *Robotics & Automation Magazine, IEEE, vol.19, no.3, pp.33-45*, Sept. 2012. `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6299172&isnumber=6299141`.