DISEÑO Y CONTROL DE UN CUADRICÓPTERO CONTROLADO POR BLUETOOTH VÍA ANDROID APP

AUTOR:
Federico, Báguena Camarena

TUTOR:
Leopoldo, Armesto Ángel

Escuela Técnica Superior de Ingeniería del Diseño

Curso 2015-2016

Valencia, Septiembre de 2016

Resumen

Este proyecto consiste en el diseño electrónico, mecánico y el control automático de un cuadricóptero mediante el microcontrolador Arduino Nano y, como mando radiocontrol, una app Android mediante conexión *bluetooth*. Se incluye la metodología aplicada en cada uno de los procesos de diseño, programación y ensambladura; dentro del uso de las disciplinas estudiadas en nuestro campo de la ingeniería electrónica y automática.

Palabras clave: Arduino, Android, microcontrolador, control automático, cuadricóptero, *bluetooth*.

Abstract

This project consists in an electrical and mechanical design of a quadcopter and its automatic control by means of an Arduino Nano microcontroller and, as radio controller, an Android app through bluetooth connection. It includes the applied methodology in each and every one of the design, programming and assembling processes; all of it performed through the studied disciplines in our electronical and automatic engineering field.

Keywords: Arduino, Android, microcontroller, automatic control, quadcopter, bluetooth.

Índice de Tablas

Índice de figuras

A mi familia, que me ha apoyado y animado desde que empecé en esta aventura universitaria. A mi hermano, por sus consejos. A mis amigos y amigas, por alegrarme en los peores momentos. A los técnicos de laboratorio de la ETSID, por su desprendida cooperación. Y, finalmente, a todas aquellas personas que me han ayudado y aconsejado en los momentos de más incertidumbre. A todos vosotros, muchas gracias.

Índice

1. Introducción

1.1 Objetivos

Los principales objetivos de nuestro trabajo son los siguientes:

- Realizar un algoritmo de control que permita al dron mantenerse estable automáticamente.
- Implementar una app Android mediante el software AndroidStudio para el manejo del cuadricóptero a distancia mediante comunicación *bluetooth.*
- Desarrollar una base sólida, ligera, flexible y desmontable.

1.2 Motivación

Desde el comienzo del primer curso, la aventura de emprender el grado en Ingeniería electrónica motiva al alumno a mirar al futuro con las infinitas posibilidades que ofrecen los conocimientos adquiridos en el transcurso del Grado. Durante el primer contacto con disciplinas como el control automático, no llegamos a ver el alcance que tiene hasta que llegado el último curso en esta mención, y con una mínima experiencia práctica en el sector se nos ocurren maneras de implementar sistemas a los que dotar con estos conocimientos que, eventualmente, vemos imprescindibles en la creación de un sistema robotizado. Lo mismo ocurre con la programación en C/C++, cuya lógica y sencillez no vemos hasta alcanzada cierta práctica a través de fallos en la compilación del código y adentrarse en el mundo de la optimización de un código que, al comienzo, parecía inalterable. A su vez, la familiarización con los términos más simples de la electrónica, como lo son el voltaje, la intensidad y la resistencia de un componente o sistema puede llegar a ser un punto clave en la elección de un componente u otro. Así pues, todo esto se decanta en un mismo punto. En el cual cada rama descrita realiza su función para sincronizarse, desde su propia misión, con los otros componentes; ya sean de *hardware* o *software*. Formando entre todos así, un sistema.

El último año del Grado, se le permite al alumno poner en práctica los conocimientos y experiencias de esta travesía. De forma que, motivado por el reto de crear un sistema robótico desde cero, la curiosidad por distintos entornos como son las nuevas tecnologías que mueven el mundo, ideas propias que desarrollar en un futuro, y el alcance que llega a tener esta experiencia en el porvenir de la carrera de un ingeniero, vimos idónea la idea de implementar un robot tan completo como lo es un *quadcopter*(cuadricóptero).

1.3 Alcance y límites del proyecto

- El control mediante comunicación *bluetooth* resulta en una corta distancia de conexión con el dispositivo, lo que significa que no habría problemas en espacios cerrados, pero en espacios abiertos se perdería la comunicación con el mando de control remoto.
- El software de control de estabilidad ha sido diseñado para que se pueda añadir con facilidad cualquier dispositivo nuevo al sistema.
- Es necesario tener en cuenta que el manejo RC de un dron no es sencillo, incluso con el modo estable se necesita práctica y fluidez para no estrellar el dron.

- La app nos permite conectarnos y controlar el vuelo del dron desde cualquier dispositivo Android que la tenga instalada.

2. Estado del Arte

2.1 Control automático

El control automático realiza la regulación de procesos sin la intervención directa del ser humano. En el modo más simple de un control en lazo automático, un controlador compara el valor medido de un proceso con la referencia, resultando en el error de la señal que será procesado para cambiar la señal de entrada del proceso de manera que el sistema se mantenga en su punto deseado (*setpoint*) sin importar las posibles perturbaciones. Este control en lazo cerrado envía una señal de realimentación negativa.

Diseñar una estructura con las características de un sistema de control, generalmente requiere de una alimentación eléctrica o mecánica para mejorar sus características dinámicas. El control se aplica regulando la alimentación en los actuadores. Un concepto a tener en cuenta es el del sistema a controlar y la información recibida de la realimentación en el lazo cerrado para habilitar una correcta alimentación.

Los componentes principales de un sistema de control automático son:

- El sensor, que nos proporciona datos de la medición de algún estado físico en que se encuentre el sistema. En nuestro caso el dispositivo MPU-6050 se encarga de medir la aceleración y la velocidad angular.
- Controlador, encargado de evaluar y procesar la información proporcionada por el sensor, y ordenar qué debe realizar el actuador para corregir el error mediante una variación en la alimentación. En nuestro caso será el microcontrolador Arduino Nano.
- Actuador, efectúa una respuesta a cierto estímulo; como lo es la señal de salida del microcontrolador. En nuestro caso serán los *Electronic Speed Control*(ESC) junto con los motores *brushless* (sin escobillas).

Los tres interactúan como se muestra en el siguiente diagrama de bloques:

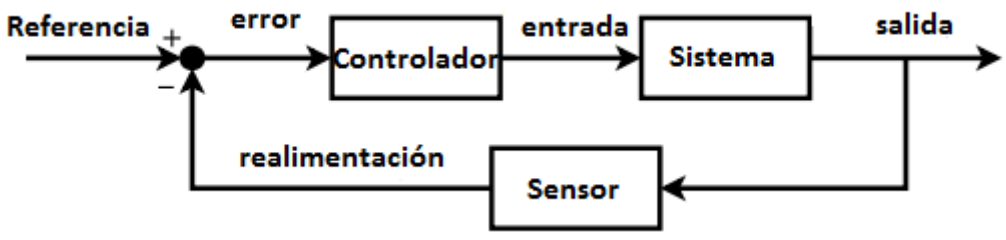


Figura 1. Lazo Cerrado.

"Fuente: propia"

El valor de la realimentación en lazo cerrado para controlar el comportamiento dinámico del sistema es negativo. Debido a la resta aplicada para obtener el error que aleja el sistema de ser estable, y tal error es amplificado por el controlador.

En el control automático de este caso en particular se hace uso del sensor giróscopo, que mide la velocidad angular en grados por segundo, y del sensor acelerómetro, el cual toma las mediciones en metros por segundo al cuadrado; ambos anexados en el mismo componente. Dichas características permiten al microcontrolador actuar según las mediciones realizadas por el dispositivo.

En general se suele utilizar un controlador PID, que son las siglas de Proporcional, Integral y Derivativo; el cual describiremos más tarde en el apartado de diseño.

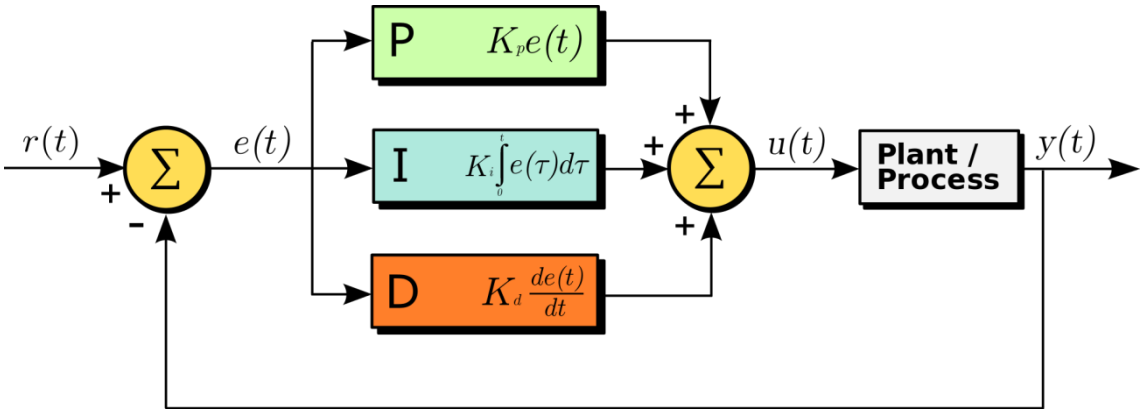


Figura 2. Diagrama de bloques PID.

"Fuente: https://en.wikipedia.org/wiki/PID_controller"

2.2 Microcontroladores y Arduino

Un microcontrolador, está definido como un circuito integrado programable en un lenguaje ensamblador, determinado por el uso que se le vaya a dar, y capaz de ejecutar ciertas órdenes computadas por el código escrito en su memoria. En este trabajo será el microcontrolador del fabricante Arduino, modelo Nano; cuya interfaz es auto-intuitiva y programado en lenguaje C/C++. Con el factor añadido de su pequeño tamaño y peso, perfecto para la instalación en un vehículo aéreo no tripulado (VANT).

Hoy en día podemos encontrar fácilmente y a un precio muy asequible, infinidad de microcontroladores para su uso o experimentación en casi cualquier materia.

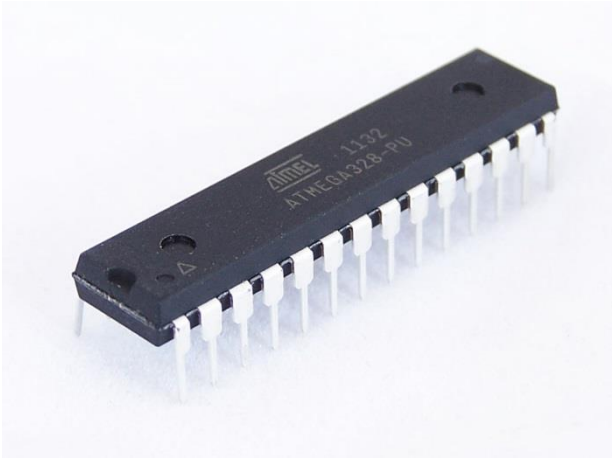


Figura 3. Imagen del microcontrolador ATMega328.

"Fuente: electronilab.co"

Como sistema electrónico en el que confluyen *hardware* y *software,* Arduino se creó con el objetivo fijado en el movimiento del código abierto. Desde entonces, atrás en el año 2006 cuando tan solo era parte de un Proyecto para estudiantes en el instituto de Ivrea en Italia, Arduino ha crecido con el soporte de investigadores, empresas y aficionados de todo el mundo.

Estos son los tres elementos principales del ecosistema Arduino:

- El *hardware*: formado por una placa electrónica con la finalidad de desarrollar proyectos económicos y rápidos. Esta, se puede complementar mediante el uso de otras placas llamadas *shield,* las cuales se conectan al microprocesador.
- El entorno de programación: el cual permite de forma muy sencilla la programación del *hardware* escogido. Además de ser un entorno multiplataforma, y únicamente necesita de un cable USB para la interacción entre la placa y la plataforma (Windows, Linux o Mac).
- La comunidad Arduino: gente de todos lados que participan y ayudan al desarrollo de nuevas aplicaciones, ideas y desarrollos. Existen innumerables foros, blogs, y otro tipo de fuentes de información relacionados con esta sencilla y útil placa electrónica.

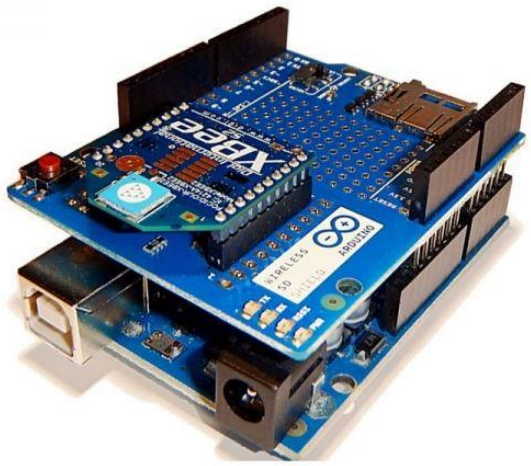


Figura 4. Shield con placa Arduino.

"Fuente: tienda.bricogeek.com"

El entorno integrado de desarrollo o *Integrated Development Environment* (IDE), es un *software* disponible gratuitamente en la página web oficial de Arduino. Posibilita la realización y compilación de *sketch*. La compilación, es una traducción a un formato que el procesador de la placa pueda interpretar y entender.

El éxito y repercusión de este ítem tan asequible se debe a diversas circunstancias: el económico precio del material a utilizar, la sencillez de su plataforma de programación, y por último su cualidad de *software* y *hardware* abierto, con licencia que autoriza su libre estudio, reproducción y modificación.

2.3 Smartphones y Android

Conocidos por todos y usados por la mayoría, los *Smartphone* forman parte de la vida diaria de cualquier persona joven y de mediana edad, más aún, sus aplicaciones hacen de estos dispositivos portátiles una herramienta más que útil. Por eso, apostamos por crear una *app* Android para el manejo a distancia del cuadricóptero, pudiendo conectarse al mismo desde cualquier dispositivo Android con pantalla táctil.

Android es el sistema operativo que pertenece a la compañía estadounidense, Google. Integrando servicios del mismo en el teléfono, como son YouTube, Maps, Gmail y otros.

Figura 5. Logo oficial de Android.

"Fuente: https://www.android.com"

Los dispositivos Android destacan por su sistema configurable y sencillo que proporciona al usuario todo tipo de necesidades que estos puedan cumplir. Priorizando la comunicación en redes sociales y realizar fotografías; este es el uso básico de un usuario medio.

2.4 Impresoras 3D

La impresión 3D es un grupo de tecnologías de fabricación por adición donde un objeto tridimensional es creado mediante la superposición de capas sucesivas de material. Esta es una descripción clara del concepto o arte de la impresión 3D, extraído desde Wikipedia. Su rapidez, precisión, facilidad y precio de producción destaca sobre otras tecnologías basadas en la fabricación por adición. Esta, ofrece al desarrollador del producto, la capacidad de imprimir piezas y montajes hachos de distintos materiales con diferentes propiedades físicas y mecánicas; por ejemplo, en nuestro caso hay un proceso de montaje con las piezas diseñadas e impresas.

El año 2003 empezó el crecimiento en la venta de estas impresoras, y al largo de los años se ha ido abaratando el coste de estas máquinas para su distribución tanto pública como en el ámbito privado. Esta tecnología, también se encuentra en campos como la joyería, arquitectura, ortopedia, educación, en distintos campos de la ingeniería, y en una gran cantidad de especialidades.

Figura 6. Impresora 3D estilo Prusa.

"Fuente: http://www.replikeo.com"

El diseño del objeto a desarrollar se lleva a planos virtuales con un software de *computer-aided design* (CAD), en nuestro caso SolidWorks. El formato estándar de datos para impresión 3D, es el de los archivos *Stereo Lithography* (STL). Este, define la geometría de los objetos en tres dimensiones, sin añadir información como el color, texturas u otras propiedades físicas sí incluidas en otros formatos CAD.

Existen distintos modos de impresión, cuyas diferencias residen en la manera en que las distintas capas son aglomeradas para crear la pieza. Por supuesto, cada método tiene sus ventajas e inconvenientes. En general, las consideraciones primarias son la velocidad de producción, coste del objeto impreso, valor de la impresora 3D, y elección, coste y calidad de materiales.

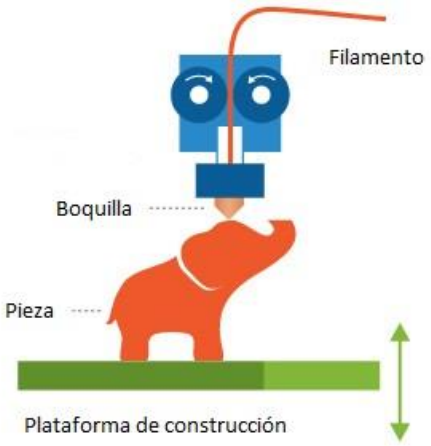


Figura 7. Modelado por deposición fundida.

"Fuente: https://www.printspace3d.com"

En este caso, el método de impresión utilizado es el modelado por deposición fundida. Utiliza una técnica aditiva, depositando el material en capas para formar la figura. Como se puede ver en la figura, hay un filamento plástico almacenado en rollos que inicialmente se introduce en una boquilla, con el objetivo de comportarse como la tinta en un bolígrafo. La boquilla se encuentra en por encima de la temperatura de fusión del material plástico, desplazándose por los tres ejes. Este movimiento es controlado electrónicamente mediante servo-motores o motores de pasos. El modelo es construido con finos hilos del material, solidificándose en el acto al salir de la boquilla.

2.5 Robótica

La robótica es una ciencia o rama de la ingeniería mecatrónica, ingeniería mecánica, ingeniería eléctrica, ingeniería electrónica y ciencias de la computación que se ocupa del diseño, construcción, operación, disposición estructural, manufacturación y aplicación de los robots. En general, la robótica combina diversas disciplinas como son la mecánica, la electrónica, la informática, la inteligencia artificial, la ingeniería de control y la física.

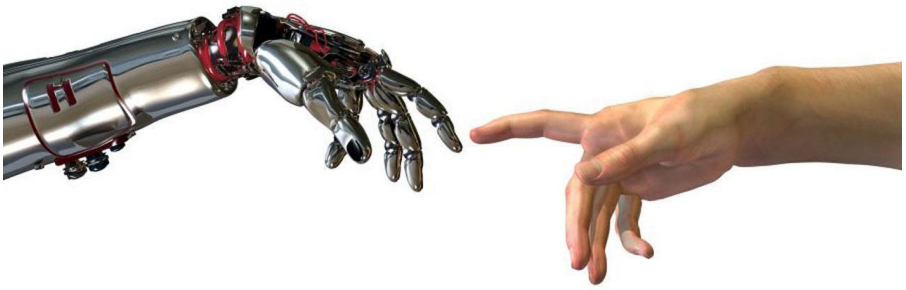


Figura 8. Ilustración simaluando la obra "La creación de Adán" de Miguel Ángel.

"Fuente: http://www.adslzone.net/2014/07/15/dedos-artificiales-de-silicona-otro-gran-avance-de-la-robotica/"

Con la imagen mostrada arriba, se pretende expresar y comparar de forma ilustrada la creación de estos entes tecnológicos mediante la imagen de la conocida pintura residente en el techo de la Capilla Sixtina, del excelente artista histórico Michelangelo Buonarroti, más conocido como Miguel Ángel, llamada "La creación de Adán".

La robótica empezó con la intención de crear artefactos que supusieran de ayuda o como sustitución al trabajo realizado por el ser humano. Evolucionando de manera que, actualmente existen trabajos que no podrían ser realizados más que por autómatas o maquinaria programada e instalada con ese único fin. Por ejemplo, en la factoría automovilística es imprescindible el pulido de las piezas que forman el motor mediante fresadoras, y dependiendo del modelo de motor que esté en producción, se definen unas medidas y otro tipo de características relacionadas con la forma y encajes de las diversas piezas que lo forman. Lo que conlleva a la implementación de distintos programas adaptados a la maquinaria disponible dentro del mismo sistema.

Figura 9. Cadena de montaje en la industria automovilística Ford, Almussafes, Valencia.

"Fuente: http://www.levante-emv.com"

En nuestro proyecto, procedemos a desarrollar un robot cuadricóptero. Este, es controlado a distancia mediante un mando radiocontrol, programado para actuar según las órdenes de dicho mando y las medidas de los sensores instalados. Es decir, no contiene ningún atisbo de inteligencia artificial, pero si de un sistema de computación programado en lenguaje C/C++ condicionado por los datos provenientes del exterior del sistema. El punto de esta aclaración es el de entender que los robots son entidades creadas para cumplir expresamente uno o diversos propósitos, solucionar un problema, investigar nuevas vías de la tecnología, exploración de alternativas a métodos tradicionales de trabajo, y un largo etcétera. En la imagen inferior se aprecia un robot mula de carga con fines militares capaz de atravesar terrenos con dificultades para otros transportes de ruedas de manera que su sistema de control automático de equilibrio le permite cargar con distintas mercancías sin llegar a dañarlas en el transporte por golpes debidos a la inestabilidad del terreno.

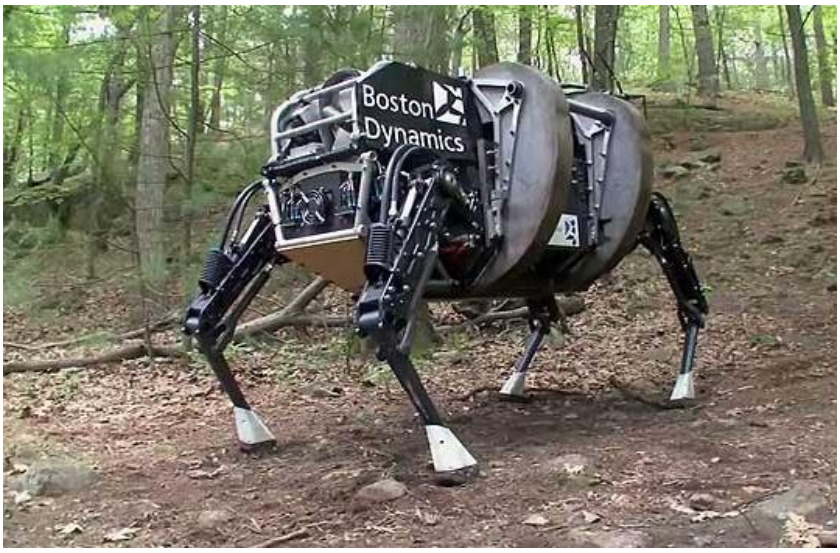


Figura 10. Robot AlphaDog de Boston Dynamics, propiedad de Google.

« Fuente: http://www.parentesis.com »

2.6 Aeromodelismo y drones

La historia del aeromodelismo con drones es bastante reciente si lo comparamos con el control automático, aunque su futuro es tan prometedor como este último. De la mano del primer fabricante de dispositivos manos libres del mundo, Parrot, vino el modelo AR.Drone. Un *quadcopter* que, dirigido al mercado de juguetes como parte de un programa de diversificación de sus productos, fue una revolución en el sector. Desde hace unos años, ya está a la venta la versión 2.0 de este modelo; destacando la posibilidad de usar gafas de realidad virtual haciendo uso de la cámara frontal y su calidad de imagen.

Dejando atrás el mundo comercial, en el desarrollo de nuestro dron nos hemos enfocado en conocer y aplicar las bases para conseguir un control de estabilidad robusto y fiable, sin olvidarnos de los movimientos que debe realizar. Para así, en un futuro no muy lejano ser capaces de implementar un código que rivalice en funciones y novedad con el mercado actual.

Figura 11. Policía londinense manejando un dron de vigilancia mediante realidad virtual

"Fuente: http://www.smartdrone.com/drones-to-help-london-police-track-bike-thieves.html"

El mundo del aeromodelismo, desde el punto de vista de los drones, se ha visto dividido en distintos sectores en los que estas aeronaves son desarrolladas. Hablamos de su uso, todo empezó como mero entretenimiento, aunque están resultando ser de mayor utilidad y decisivos en otros ámbitos. Empezando por su empleo en actividades de investigación y desarrollo, observación y vigilancia aérea en incendios forestales y operaciones de emergencia, como hobby, en el estudio académico de sistemas de control, y terminando con su ocupación en el entorno deportivo en las carreras de drones.

Figura 12. Dron profesional para fines de investigación en la Geografía.

"Fuente: http://www.drones-mx.com"

El fin más común que estos aparatos experimentan es el deportivo, participando en carreras de vehículos aéreos no tripulados a radiocontrol. De manera que no necesitan estar

registrados, al contrario que ocurre en otros casos. Existen leyes que limitan su uso dependiendo del fin que vaya a servir. Por lo que, para su empleo en trabajos aéreos como los citados anteriormente se necesita de registro, placa de identificación con datos del contacto, número de serie, y el nombre de la empresa operadora. De otra forma, su uso sería ilegal.

Figura 13. Imagen del circuito usado en el concurso de Carreras de Drones celebrado en Dubai.

"Fuente: www.youtube.com en IDRA World Drone Prix 2016 in Dubai"

3. Planificación

Con respecto a la planificación de este Trabajo de Fin de Grado, hemos agrupado en ella toda información detallada de los componentes y su función a lo largo de cada etapa de desarrollo. Así evitaremos repetirnos en demasía con los mismos tópicos en otros apartados, además de proporcionar datos relevantes sobre el avance del proyecto en cada fase.

Para empezar, elegimos los componentes con esmero debido a la importancia de la materia, y teniendo en cuenta las características principales que deben cumplir. Por supuesto, se tuvo en cuenta la compatibilidad entre los distintos elementos.

3.1 Elección y familiarización con los componentes

3.1.1 El microcontrolador Arduino Nano

El Arduino Nano es una pequeña, completa y sencilla placa programable basada en la tecnología ATMega328. Estas son sus características más destacables:

Tabla 1. Especificaciones Arduino Nano

Microcontrolador	ATmega328
Pines I/O digitales	14 (6 de ellos con salida PWM)
Pines Input Analógicos	8
Corriente DC de salida por Pin	40 mA
Voltaje pines I/O	5V
Memoria Flash	32 KB (de los cuales 2 KB los usa el *bootloader*)
SRAM	2 KB
EEPROM	1 KB
Clock Speed	16 MHz

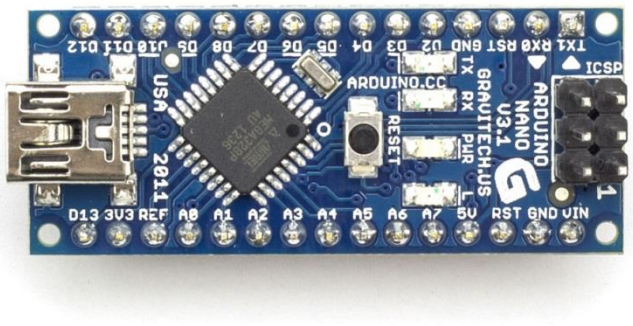


Figura 14. Arduino Nano.

"Fuente: https://www.arduino.cc/en/Main/ArduinoBoardNano"

Puede ser alimentado vía conexión USB, o bien desde una fuente externa que deja dos opciones. Alimentar mediante una fuente fija de 5V u otra fuente variable; recomendable entre 7-12V según la documentación.

Por una parte, cada uno de los catorce pines digitales pueden usarse como puertos de entrada o salida, haciendo uso de distintas funciones. Primero, la función pinMode() que sirve para adjudicar un cometido al pin deseado; entrada o salida. Y por último la función que va a determinar la acción a realizar en el pin; lectura o escritura. Las funciones son, digitalWrite para escritura, y digitalRead para lectura de datos. Operan a 5V. Cada pin puede recibir o proporcionar un máximo de 40 mA, y añaden una resistencia *pull-up* de 20 a 50 KOhms que está desconectada por defecto. Además, algunos pines tienen funciones específicas; de las cuales explicamos las que son de utilidad en nuestro *sketch* Arduino:

- Puertos de comunicación serie: 0 (RX) y 1 (TX). Se usan para recibir y enviar datos serie del tipo TTL.
- *Power Width Modulation*(PWM), también conocido como modulación por ancho de pulso: 3, 5, 6, 9, 10, y 11. Pines que proporcionan una señal de salida PWM de 8-bits.

Por otra parte, también incluye ocho pines de entrada y salida analógicos. Los cuales ofrecen una resolución de 10-bits(1024 valores). Estos pueden operar desde 0 a 5V. Por supuesto, como ocurría con los puertos I/O digitales, también tienen pines con una funcionalidad especializada para ciertas tareas:

- I2C: A4 (SDA) and A5 (SCL). Comunicación I2C (TWI) mediante la librería Wire.h, la cual usamos en la inicialización del sensor acelerómetro y giróscopo.

Para la placa Arduino Nano, la comunicación entre distintos dispositivos es posible gracias a que el modelo ATmega328 dispone de puertos de comunicación en serie; concretamente los pines digitales 0(RX) y 1(TX). Sin olvidarnos de la conexión e intercambio de datos entre la computadora y el dispositivo mediante el puerto USB. Y, por último, el monitor serie que permite el envío textual de datos desde el mismo *sketch* a la consola y viceversa. Los LEDs que corresponde a los pines RX y TX se iluminan cuando se transmiten datos desde el USB conectado al ordenador, pero no lo harán cuando ocurra una comunicación serie en esos pines.

3.1.2 El sensor acelerómetro y giróscopo MPU-6050:

Se trata de una unidad de medida inercial (IMU) que incluye dos sensores en el mismo dispositivo, acelerómetro y giroscopio. Estos miden la fuerza y velocidad del mismo, es decir la IMU no mide ángulos directamente; requiere unos cálculos. El MPU-6050 es una unidad de medida inercial de seis grados de libertad (6DOF). Esto quiere decir que incorpora un

acelerómetro y un giroscopio de 3DOF cada uno. Opera con 3.3V y utiliza el protocolo de comunicación I2C. El bus I2C es una tecnología que permite la comunicación entre microcontroladores, memorias y otros dispositivos. Tan solo necesita de tres puntos de conexión y masa. Entraremos en más detalle con este estándar a explicar su función en el sistema.

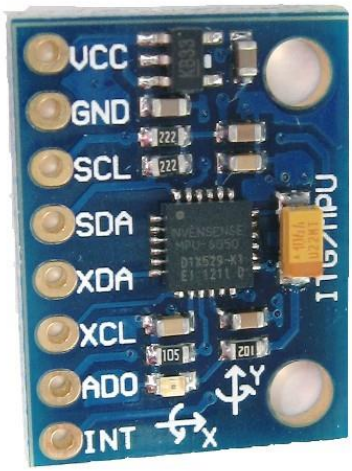


Figura 15. Sensor MPU-6050.

"Fuente: http://playground.arduino.cc/Main/MPU-6050"

Como hemos explicado, el sensor consta de un acelerómetro y un giroscopio. Estas son las características principales de cada uno:

El sensor acelerómetro mide la aceleración en los ejes X, Y y Z; las tres dimensiones del espacio. La IMU también mide la aceleración de la gravedad terrestre. Gracias a estos valores se pueden usar las lecturas para conocer el grado de inclinación respecto los ejes X e Y. Conocemos la aceleración de la gravedad, 9,8m/s^2, y los datos de medida de los ejes del sensor. Por tanto, es posible calcular el ángulo de inclinación aplicando la trigonometría. Es necesario añadir que la inclinación del eje Z no se puede calcular, ya que necesitamos de otro componente en la IMU. Se trata de un magnetómetro, también se puede expresar como una brújula digital. El MPU-6050 no incluye esta característica, por lo que no será viable calcular el con exactitud el ángulo Z. El acelerómetro es sensible al ruido, por lo que necesitaremos de un filtro paso-bajo para afinar los valores de lectura. Con la fórmula tangente calculamos la inclinación en grados.

$$Ángulo Y = \text{atan}(\frac{\text{x}}{\sqrt{y^2 + z^2}})$$

$$Ángulo X = \text{atan}(\frac{\text{y}}{\sqrt{x^2 + z^2}})$$

En cambio, el sensor giroscopio mide la velocidad angular del dispositivo en los ejes X, Y y Z. Es decir, los grados que gira en un segundo; mesurado en grados por segundo(º/seg). Para un cálculo exacto, ya que este es más preciso que el acelerómetro, debemos conocer el

ángulo inicial en que se encuentra el dispositivo, para así poder sumarle el valor que marca el giroscopio y saber el valor de la nueva inclinación a cada momento. Esta es la fórmula que nos permite calcular el nuevo ángulo en cada iteración:

$$ÁnguloY = ánguloYanterior + medidaY * \Delta t$$

Dónde Δt es el tiempo que transcurre cada vez que se calcula esta ecuación, ángulo Y anterior es el ángulo calculado la última vez que se usó, y medidaY es la lectura del ángulo Y del giroscopio.

Aunque el giroscopio es más preciso que el acelerómetro, es necesario corregir un error llamado *drift*. Esto se traduce en una acumulación de las medidas anteriores, alterando el punto de referencia y creando confusión en el equilibrio del objeto en cuestión. Mientras que el acelerómetro sufre de alteraciones en las medidas por su gran sensibilidad al ruido externo. Debido a estos dos inconvenientes, vamos a hacer uso del Filtro complementario, el cual combina un filtro paso-alto(HPF) y un filtro paso-bajo(LPF). Y así disfrutar de unos datos más precisos que usar en nuestro control automático.

3.1.2 Módulo Bluetooth HC-05

Dispositivo de comunicación Bluetooth que permite el intercambio de datos mediante puertos de conexión en serie; RX y TX. Puede configurarse como Esclavo o Maestro. En este caso, cumple el papel de esclavo; ya que este se encarga de cumplir las órdenes enviadas desde el dispositivo que actúe como maestro, en este caso, nuestra app Android. Opera tanto a 3.3V como a 5V y conectado a masa. Su sencillez a la hora de configurarlo lo convierte en una herramienta de lo más simple y útil.

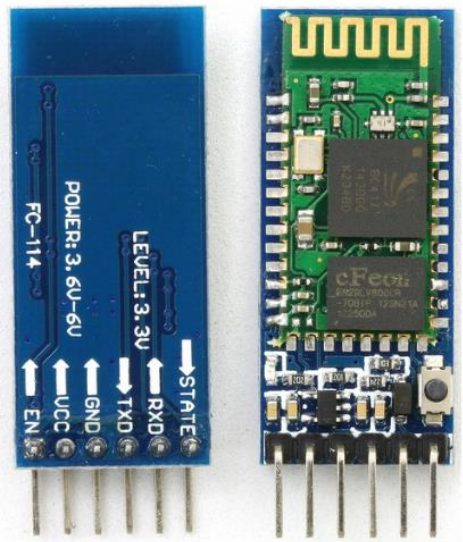


Figura 16. Módulo Bluetooth HC-05.

"Fuente: http://www.martyncurrey.com/hc-05-fc-114-and-hc-06-fc-114-part-2-basic-at-commands/"

3.1.3 Motores sin escobillas(brushless), Electronic Speed Controller(ESC) y batería Li-po:

El motivo principal por el que nuestros motores deben ser sin escobillas es porque estas ejercen un rozamiento, que, aunque mínimo, reducen en gran parte el rendimiento que pueden llegar a ofrecer. Por lo tanto, ya que nuestro proyecto necesita de motores pequeños, vamos a tener que eliminar las escobillas. En este tipo de motor, la corriente eléctrica pasa directamente por los bobinados de la carcasa, así, no son necesarias ni las escobillas ni el colector, usados en los motores con escobillas. Esta corriente eléctrica genera un campo electromagnético que interacciona con el campo magnético creado por los imanes permanentes del rotor, generando una fuerza que hace girar al rotor, y por consiguiente el eje. Suele utilizarse corriente trifásica para este tipo de motores.

Figura 17. Motor sin escobillas.

"Fuente: http://tienda.bricogeek.com"

Tabla 2. Especificaciones motores sin escobillas

Parámetro KV	1000 rpm/V
Corriente mínima de funcionamiento	4 A
Corriente máxima de funcionamiento	10 A
Eficacia máxima	80%
Modelo	A2212/13T

Estas máquinas eléctricas funcionan con corriente alterna (AC), pero nuestra batería proporciona corriente continua(CC). Esto nos lleva a la búsqueda de un ítem que proporcione corriente alterna para el funcionamiento de los motores; aquí es donde entran en juego los *Electronic Speed Controller*. Los ESC convierten la corriente continua de la batería con una tensión constante a una fuente de tensión variable y de sentido reversible por cada polo del

motor. Las características que nos interesa conocer de un variador de voltaje son su amperaje y su tensión de entrada máximos.

Figura 18. Electronic Speed Controller(ESC).

"Fuente: https://rc-innovations.es/"

En la figura de arriba se puede distinguir distintos cables del mismo color. En el extremo derecho se encuentra la entrada por la que se alimenta al variador. Mientras que al extremo derecho vemos tres cables negros correspondientes a la entrada de corriente trifásica del motor y el cable de entrega de pulsos. Junto a los cables de alimentación del ESC se sitúan dos salidas que corresponden a la conexión a masa y al cable de recepción de pulsos desde el microcontrolador. Al que se le añade, externamente, un limitador de voltaje para alimentar la placa Arduino, el UBEC (*Universal Battery Eliminator Circuit*). Este añadido, confiere un voltaje de salida óptimo para dicha alimentación, ya que lo abastece con 5V i 3 A; valores que entran en el rango seguro de alimentación del dispositivo.

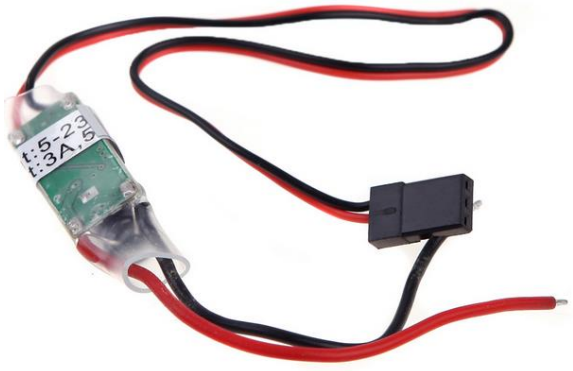


Figura 19. Universal Battery Eliminator Circuit(UBEC).

"Fuente: https://es.aliexpress.com"

Aquí mostramos las especificaciones técnicas conocidas de ambos componentes:

Tabla 3. Especificaciones ESC & BEC

Rango acho de pulsos ESC	1-2 ms
Máximo amperaje de salida ESC	30 A
Voltaje entrada UBEC	5-23 V
Voltaje salida UBEC	5V
Amperaje salida UBEC	3 A
Batería adecuada	11.1 V Li-po 2-6 celdas

Estos motores sin escobillas permiten alcanzar un gran rendimiento y una gran potencia a coste de un alto consumo. Esta es la razón por la que debemos utilizar una batería de polímero litio (Li-Po), cuya densidad de energía es menor en comparación con otras, pero con la ventaja que pueden entregar un gran nivel de potencia para el correcto funcionamiento de los motores *brushless*.

Tabla 4. Especificaciones batería Li-Po

Voltaje salida	11.1V
Capacidad de descarga contínua	25C
Capacidad de carga	1-3C recomendado
Capacidad	2200 mAh

Figura 20. Batería Li-Po.

"Fuente: http://craftmodel.com"

3.1.4 Material para el chasis del Dron:

En la creación del chasis para el cuadricóptero, apostamos por el software de diseño SolidWorks para la impresión y ensambladura de piezas en 3D. Nos basamos en otros diseños

que cumplían con nuestras necesidades de tamaño y estética, adaptando las medidas a los componentes empelados. En la siguiente figura se muestra el diseño en que nos basamos para el modelado del nuestro.

Figura 21. Piezas impresas.

"Fuente: www.thingiverse.com"

3.2 Mecánica del sistema

Hasta aquí el primer paso en la construcción del dron. El siguiente punto fue estudiar el modelo de vehículo aéreo no tripulado que íbamos a implementar desde diferentes fuentes, y entender el cómo de su ciencia. Existe un gran número de ejemplos dentro del mundo del aeromodelismo, como lo son helicópteros, aviones, drones de tres hélices y otros. Nuestro VANT tiene cuatro hélices, por tanto, es un *quadcopter*. Una vez establecido qué prototipo tanteamos, sólo quedó documentarnos acerca de dicho funcionamiento.

Cuando hayamos obtenido los componentes electrónicos, y conocido qué vamos a hacer, nos familiarizamos con cada dispositivo y las funciones que van a desenvolver para una precisa ejecución de sus cometidos; por los cuales han sido escogidos. En la siguiente imagen podemos ver la placa de prototipo con los elementos soldados y posicionados.

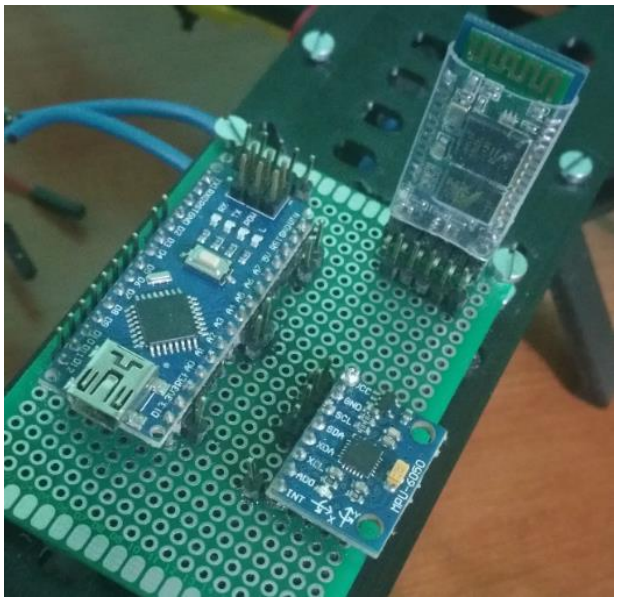


Figura 22. Placa de prototipo y dispositivos.

"Fuente: propia"

3.2.1 Diseño del software

En lo que atañe al *sketch* principal, empezamos con la programación de un control PID fiable y robusto, ya que de él depende la estabilidad del VANT, y además tenemos conocimientos previos de su estructura. Después de redactar y testear el código con valores predeterminados, procedimos a continuar con las siguientes partes del código principal como son la obtención y transformación de datos del sensor MPU-6050, inicialización del módulo *bluetooth* para la recepción de datos desde la app Android y calibrado de los ESC. Además de las ecuaciones que determinan el movimiento que se va a efectuar en función del ancho del pulso determinado para cada acción; cuyos valores dependerán de las variables aceleración(*throttle*), eje Z (Yaw), eje Y (Pitch) y eje X (Roll). En la siguiente ilustración se muestra el comportamiento del dron según qué motor aumente su velocidad.

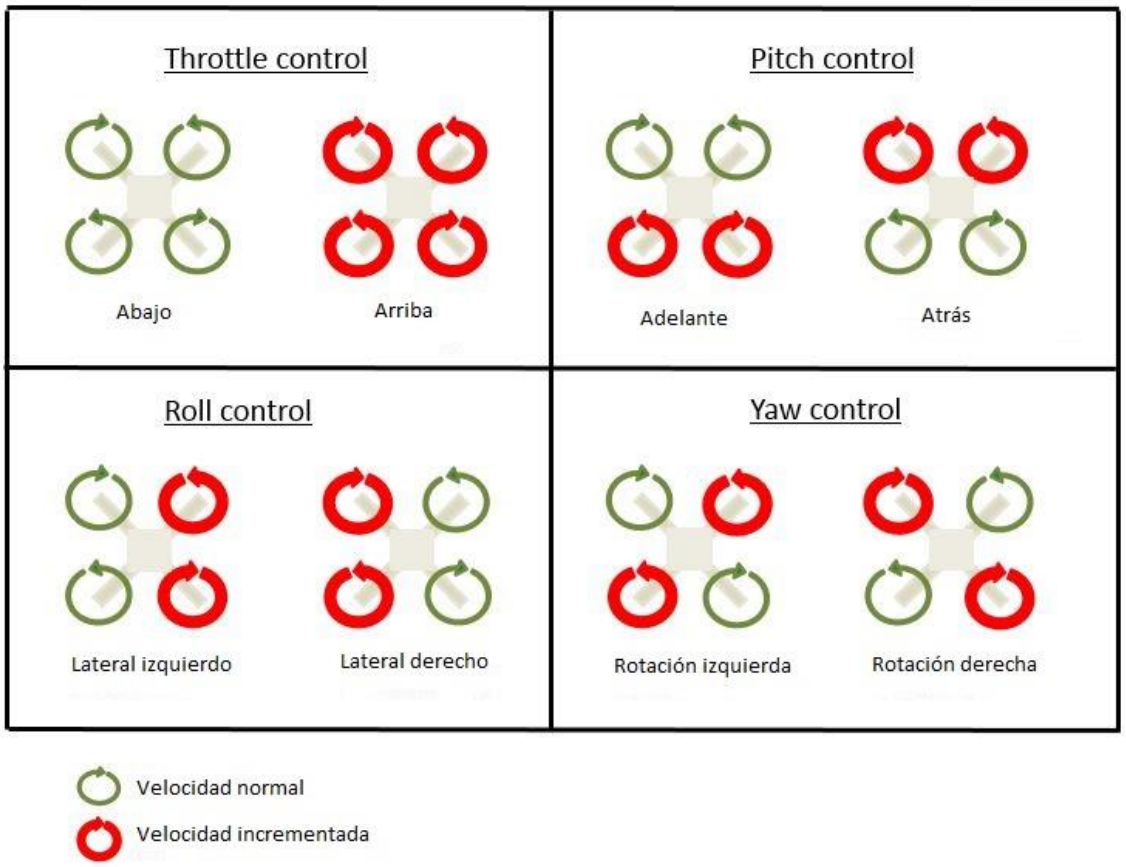


Figura 23. Comportamiento y rotación motores.

"Fuente: http://dronenodes.com/how-to-fly-a-quadcopter-beginner-guide/"

Cabe añadir, que cada parte del código principal ha sido escrita por separado con el fin de comprobar su preciso funcionamiento de manera individual, para eventualmente unir cada una de las partes en el mismo *sketch*.

3.3 Impresión 3D y diseño con SolidWorks

Otro punto, es el diseño e impresión 3D del chasis(*frame)* que dará forma a nuestro vehículo aéreo no tripulado. Su desarrollo se dejó para ser creado justo en el momento en que terminásemos el software con la IDE Arduino. El principal motivo, fue la sencillez con que abarcamos cada paso en el esbozo del chasis y el software de CAD SolidWorks. Finalmente tomamos ventaja de los recursos de nuestra escuela, la Escuela Técnica Superior de Ingeniería del Diseño(ETSID) para hacer uso de las impresoras 3D al abasto de cualquier alumno con causa justificada; como es el caso.

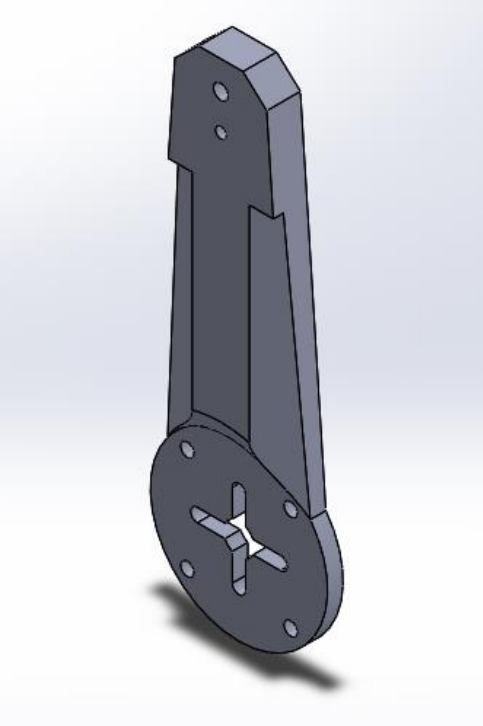


Figura 24. Ejemplo diseño SolidWorks.

"Fuente: propia"

3.4 Diseño electrónico

La siguiente etapa abarca el diseño electrónico, el cual ha sido esquematizado con el software Eagle para mejor entendimiento y orden en el ensambladura de la parte de potencia. Primero imprimimos una placa de circuito impreso llamada *Power Distributor Board*(PDB) que repartirá, como su nombre indica, alimentación desde la batería a los distintos elementos del sistema desde un mismo punto en común.

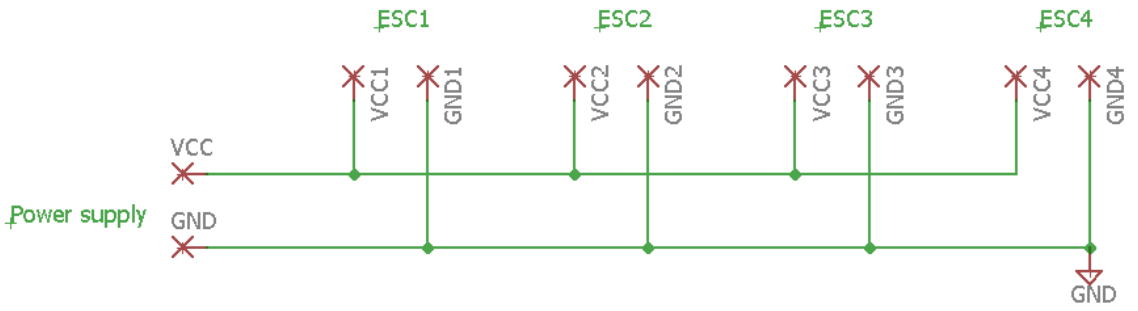


Figura 25. Esquema Power Distributor Board.

"Fuente: propia"

Segundo y último, ordenamos qué pines del microcontrolador iban a ser utilizados en el trabajo. Así mismo, podemos agrupar todos los puertos I/O en un mismo punto y conectarlos a los distintos dispositivos con orden y una sencillez equiparable a la conexión de un puerto USB. En el siguiente punto llegamos al montaje íntegro de todas las piezas y componentes del dron.

3.5 Ensamblamiento

En síntesis, el montaje y unión de los componentes del cuadricóptero es el punto más delicado de todos los anteriores. Hay que tener en cuenta las dimensiones, la sujeción, y ante todo las conexiones entre dispositivos. Es crucial tener claras las instrucciones de ensamblaje y empalmado entre componentes.

Empezamos por la base inferior, que no incluye aparato electrónico alguno; dejándolo en un simple atornillado entre piezas. Este componente incluye las patas en que el robot se apoya y la estructura gemela al objeto intermedio en el conjunto. Seguido, atornillamos los brazos pertenecientes a las sujeciones de los motores y variadores de velocidad. Se diseñó con premeditación el espacio exacto para fijar la placa distribuidora de potencia al centro de la pieza. Se aprecia en la siguiente imagen los casos descritos.

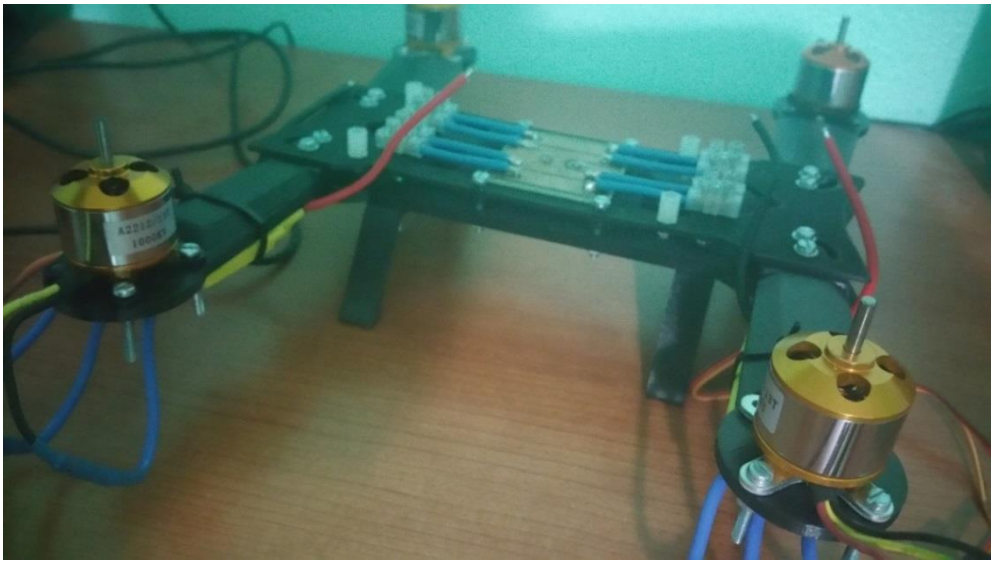


Figura 26. Montaje base principal y fijación motores.

"Fuente: propia"

Como se ha visto en la imagen, los motores eléctricos están fijados con tornillos y tuercas a su respectivo sitio. En cambio, los variadores de velocidad lo están de manera que no estorben y mantengan sus cables a la distancia calculada de su empalme con la alimentación. Antes de colocar la última pieza del conjunto, se debe preparar. Esta incluye, el sistema programable con Arduino Nano, sensor MPU-6050 y módulo externo de conexión bluetooth HC-05, placa de prototipo donde están unidos los componentes citados, y la batería.

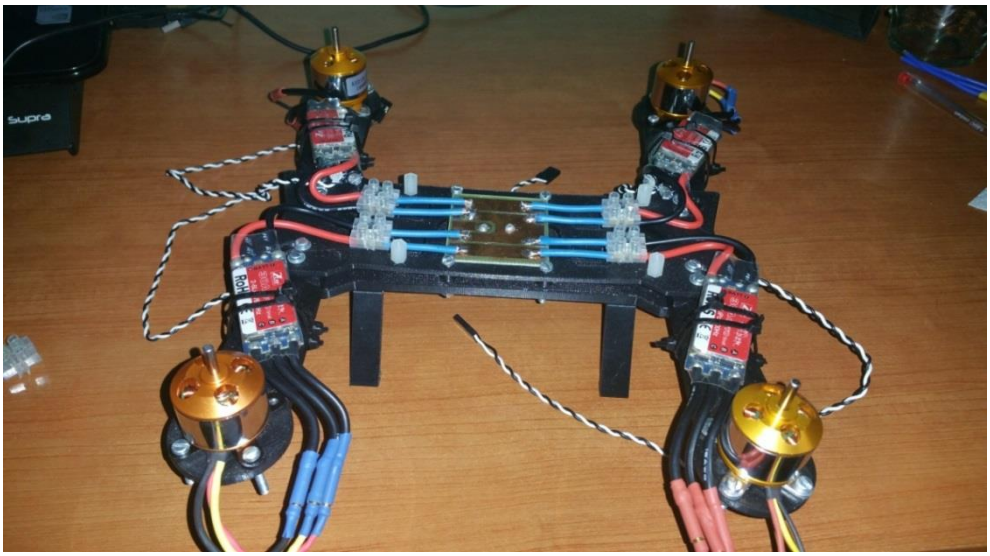


Figura 27. ESCs instalados en la estructura.

"Fuente: propia"

Después de ordenar estos elementos, ensamblamos con mucho cuidado la base a la que están ligados con el conglomerado que forman el chasis de nuestro vehículo aéreo no tripulado. Seguido, adherimos la batería al espacio reservado justo debajo y bien centrada para tener el dron lo más simétrico posible. A diferencia de otros elementos, la batería debe ser extraíble con facilidad, debido a la necesidad de recargarla. Llegamos al último punto del montaje. Aquí es donde se necesita de organización a la hora de conectar cada pin y cada entrada de alimentación de forma correcta. Nosotros hemos escrito una serie de pasos para empalmarlos de uno en uno, con el fin de no olvidarnos de ninguna conexión importante. Empezando por las conexiones a masa y alimentación, seguido de las demás conexiones relacionadas con las entradas y salidas de datos del sistema en los distintos pines que intercomunican todos y cada uno de los dispositivos.

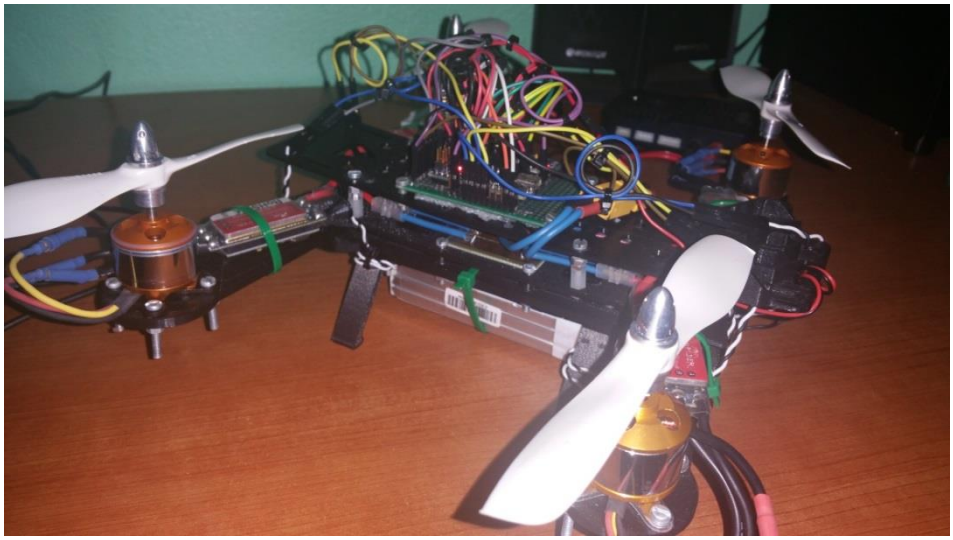


Figura 28. Dron montado.

En la figura proporcionada, se distinguen distintos elementos como las hélices y su posición, los motores fijados a las patas, la batería fijada a la parte de abajo. Aunque no se pueda divisar, la batería LiPo está sujeta primero por una tira de velcro para mantenerla firme, y también ofrecer una sencilla extracción. En la parte de arriba, encontramos las conexiones entre los dispositivos de *hardware* y *software*. Esta manera de dejar desprotegidos los cables, nos permite observar bien los LEDs de los elementos, así sabemos en todo momento, en caso de avería qué puede estar fallando. Por supuesto, la idea era, una vez terminado el proceso de calibrado PID y haber volado el dron sin problemas técnicos, desarrollar una carcasa que sirviese de protección para las diversas conexiones, y asimismo mejorar la estética del robot.

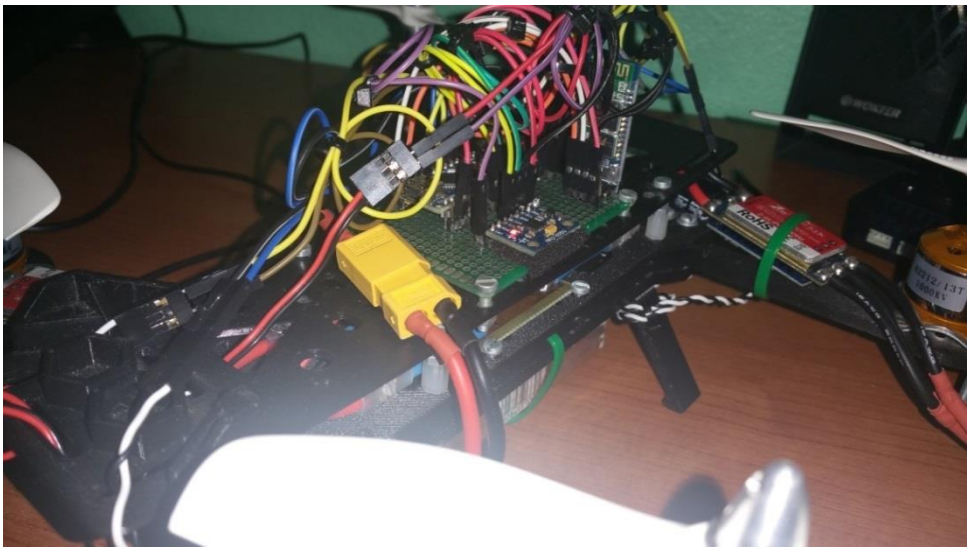


Figura 29. Cableado.

3.6 Diseño Android App

Con respecto al diseño de la app Android, lo apartamos hasta casi llegado el final. Hay que destacar la importancia de los puntos anteriores en comparación a este en concreto. Esto es debido a la incertidumbre de no saber si el proyecto se terminaría a tiempo, en cuyo caso hubiésemos prescindido de crear por nosotros mismos la app y buscar una alternativa prefabricada en la red.

Figura 30. Logo Android Studio.

"Fuente: https://developer.android.com"

3.7 Modos de vuelo

Este tramo está dedicado a presentar y entender los dos modos de vuelo que se suelen incluir en el software de control de un cuadricóptero, y que nosotros hemos programado en el nuestro. Se trata simplemente de dos estilos de manejo, Manual y Estable. La diferencia entre ambos, es la forma en que el sistema usa los distintos valores del sensor acelerómetro y giróscopo.

Modo Manual o Acrobático: de los dos, es el más trabajoso en términos de práctica de vuelo. Esto se debe a la sensibilidad de reacción de los actuadores ante la entrada de órdenes desde el mando RC y el tipo de sistema de control. Dicho de otra forma, esta manera de manejar el dron toma como referencia los datos del giroscopio, es decir, de la velocidad angular a la que se desplaza. Y no incluye un control en cascada como el modo Estable. Este estilo de vuelo no es recomendable para aficionados novatos en el aeromodelismo, ya que requiere de mucha práctica con la maniobrabilidad del mando radiocontrol. Aquí, el sistema debe mantener la velocidad angular de cada eje a cero, lo que significa que su finalidad es la de conservar una posición sin que fuerzas externas como el viento creen inestabilidad en el vuelo. Cabe añadir que, en el tiempo desde que se enciende el aparato, el sensor no tiene un punto de referencia fijo, por lo que necesita de un continuo control vía RC para mantenerse perpendicular al suelo.

Modo Horizon o Estable: como su nombre indica, difiere del anterior por la capacidad a la hora de mantenerse en equilibrio sin ayuda del mando radiocontrol. Aquí, sí interviene un control en cascada, compuesto por los valores de ambos sensores presentes en el dispositivo MPU-6050 como entrada. Debido a su estructura en cascada, la obtención de coeficientes es ligeramente distinta que la del modo manual, ya que se tiene en prioridad la velocidad y

estabilidad del comportamiento ante el error de posición. Esto se explica mejor en el apartado del control PID. Volviendo al tema en cuestión, esta técnica de vuelo se aplica con frecuencia en utilidades de captura de imágenes desde las alturas. Permitiendo obtener resultados con mínimos balanceos del objetivo, y consiguiendo así una buena calidad en la imagen captada.

4. Funcionamiento

Antes de empezar con la descripción entera de los códigos implementados en la creación de este proyecto, explicamos de forma clara qué va a realizar el sistema y de qué herramientas va a hacer uso con el fin de dejar clara cuál es la función a cumplir por cada componente y por el mismo sistema. En resumen, procedemos a redactar una explicación con los detalles técnicos necesarios con la meta de dejar clara la solución implementada dirigida a cumplir con los objetivos del proyecto.

Para empezar, el código define una serie de constantes en la librería interna creada por nosotros. Aquí se encuentran datos relacionados con las constantes PID, rango de pulsos, pines a usar, ratios de conversión, límites, entre otros. Dichas constantes no van a ser alteradas en el transcurso de la secuencia, por eso se han declarado como constantes del *sketch*. Continuamos con la declaración de librerías, ya en el código principal. Estas, son un conjunto de definiciones y funciones escritas en C++ de Arduino que conceden al programador más recursos en el momento de desarrollar su código. En nuestro caso, autorizan, mediante funciones exclusivas de estas librerías, entre las que se encuentran el acceso a la comunicación I2C para el intercambio de datos entre el microcontrolador y el sensor, librería de comunicación *serial* para la comunicación serie entre dispositivos, y una para realizar operaciones matemáticas sin error de compilación.

Con las librerías ya especificadas, empieza la declaración en inicialización de las variables del sistema. Se incluyen, las variables de error de cada PID, cada *setpoint* existente, acciones de control para regulación de giro de las hélices, variables de tiempo para *debug* de variables, datos referentes a los pulsos a aplicar en cada motor después de la corrección PID, las cifras de adquisición y transformación del sensor acelerómetro-giroscopio, y la adjudicación de los pines de comunicación serie. Una vez se han concretado los datos que variarán según el sistema lo requiera, continuamos con la etapa de *setup*, que solo ocurre una vez al iniciar el microcontrolador. En esencia, se podría decir que es el tramo de configuración previo al programa principal. En este caso, inicia los puertos para la comunicación serie con el módulo *bluetooth* externo, especificando la velocidad de comunicación entre dispositivos en bytes por segundo. Seguido, inicia la comunicación I2C y la configuración elegida para el sensor referente a la velocidad de medida, sensibilidad, el filtro digital paso-bajo y añade los *offset* que ayudarán al MPU-6050 a darnos valores exactos sin apenas desviación. Después, convoca las funciones que adjudican un pin a cada motor, y preparan los motores al mínimo pulso. Esto es debido a que los ESC necesitan de un pulso mínimo de referencia para empezar a trabajar. Y, por último, se llama a la función que determina, mediante la orden que le sea dada desde el mando RC, qué tipo de vuelo queremos realizar con el *quadcopter*. Lo que significa que el sistema no iniciará ningún tipo de acción hasta que se elija un estilo o tipo de desplazamiento para el dron.

Llegamos al bucle principal, en el cual ocurren desde las medidas del sensor hasta el control de velocidad de los motores. En otras palabras, la adquisición de datos desde el sensor es la primera función llamada en el bucle continuo. Calculando así la posición y velocidad angular simultáneo para hacer uso de estos en el algoritmo de control; además de transformar las cifras obtenidas en valores válidos a las unidades de medida. Como es lógico, al terminar la secuencia de la función de lectura del sensor, se definen unos *setpoint* que pueden, o no, ser los que haga uso el control PID. Esto se debe a la siguiente interacción del *sketch* donde, en caso de haber un caracter recibido desde el módulo *bluetooth* por la app radiocontrol, compara entre las distintas condiciones para averiguar qué movimiento debe cumplir el robot. Por ejemplo, si recibiese la orden de moverse hacia la derecha de forma lateral, cambiaría el punto objetivo del eje Roll; aunque mantenga el de los otros dos a cero. O lo que es lo mismo, el sistema entiende que solo queremos variar la inclinación del eje correspondiente al movimiento seleccionado. En el caso de no recibir dato alguno, el sistema de control interpreta que precisamos un equilibrio con cero grados de pendiente en todos los ejes. Acto seguido, la función de control es convocada. Donde los datos nombrados con anterioridad son adjudicados al conjunto de variables usadas en el cálculo de la acción de control. Estas variables se envían a la función PID de su respectivo eje, y se computa una salida que se aplica en las ecuaciones de pulsos aplicados para cada motor. Antes de enviar los pulsos de control de velocidad, en secuencia intervienen unas condiciones que limitan en mínimos y máximos a los pulsos que reciben los ESC y evitar saturaciones. Para terminar con el bucle, desde la misma función, se llama a la siguiente función que otorgará a cada uno de los pines conectados a la entrada de los variadores de velocidad el pulso calculado para su respectiva corrección.

5. Diseño

En este apartado se tratan todos los procedimientos estudiados e implementados para alcanzar los resultados fijados. Las etapas de diseño están compuestas por distintas disciplinas, como son electrónica, programación en C/C++ y Java, y diseño mecánico. Han sido divididas en tres apartados correspondientes a su campo.

5.1 Diseño mecánico

5.1.1 Montaje chasis

En este apartado tratamos el diseño en su totalidad de la estructura en la que se ensamblan todos los dispositivos que conforman el *quadcopter*. Para empezar, nos planteamos el tamaño que deseábamos en función con los componentes que integran el sistema. Al consultar una página de diseños 3D llamada Thingiverse, en la cual todo el mundo comparte sus proyectos e ideas, nos maravilló la cantidad de diseños que podíamos tomar como fuente de inspiración. Hecho esto, hicimos las mediciones que precisamos e iniciamos el software de CAD SolidWorks. Estas son las piezas que diseñamos:

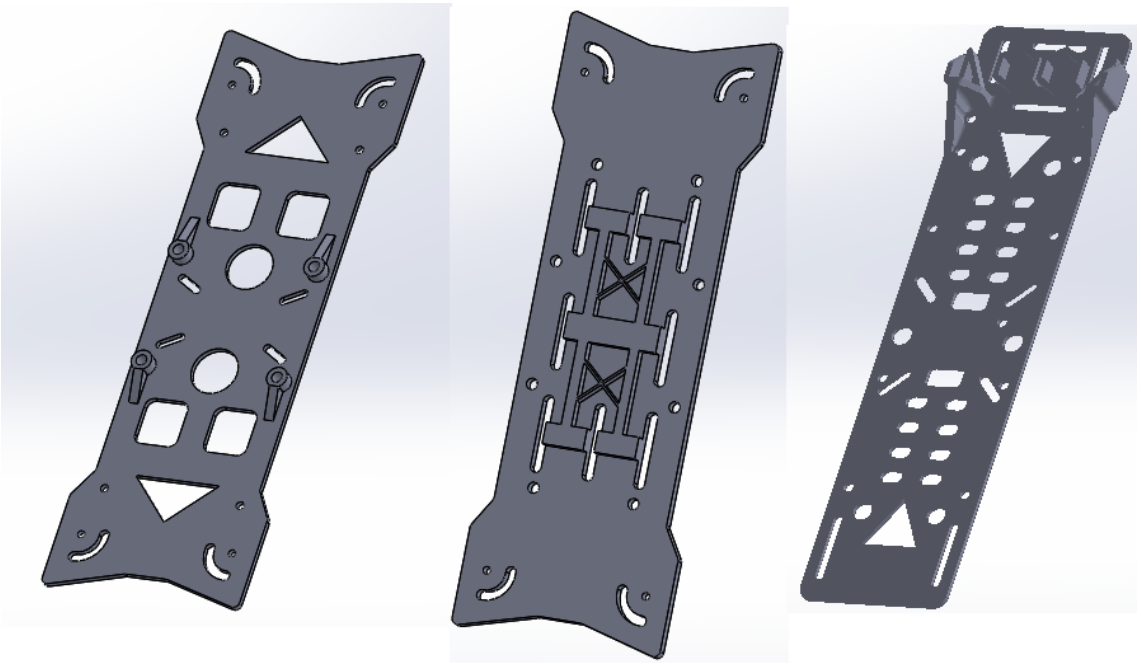


Figura 31. Base.

"Fuente: propia"

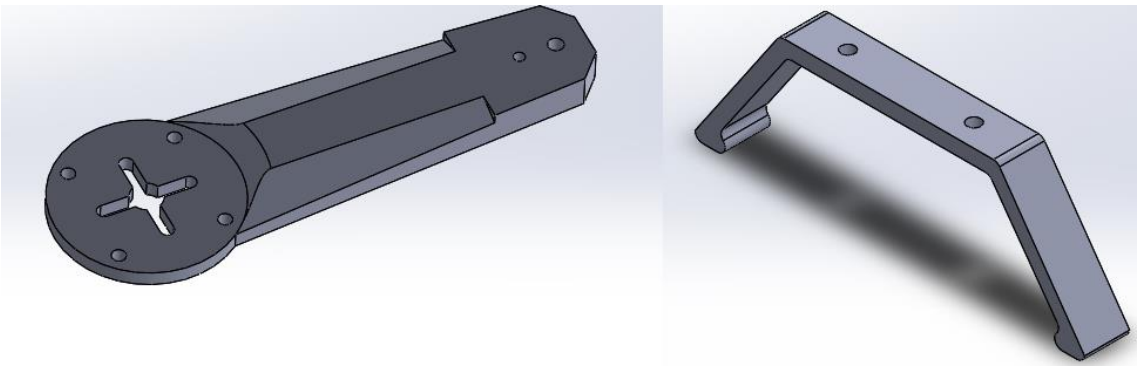


Figura 32. Brazo soporte motor y patas soporte estructura.

"Fuente: propia"

Estas figuras representan las piezas sólidas diseñadas e impresas en 3D. Nos gustó la idea de que tuviese distintos niveles entre superficie, de tal forma podemos distribuir con más facilidad los componentes que no queremos tocar ni modificar sin que sea urgente; nos referimos a la PDB. Incluyendo la protección que ofrece al no estar expuestos.

Como se ha dicho, el chasis está hecho íntegramente del elemento plástico usado en la impresión 3D. La impresora ha sido proporcionada por el laboratorio de impresión 3D de la ETSID, al servicio de todos los alumnos de la escuela. Impresora marca Zotrax modelo M200, y como material de impresión Z-ULTRAT. Este tiene las características ideales para ser la base de una máquina como lo es un cuadricóptero; durabilidad, flexibilidad, aptitud para el prototipo de piezas mecánicas.

Tabla 5. Características impresora Zotrax M200

Tecnología	Laser Phosphor Display (LPD)
Volumen de impresión	200 x 200 x 180 mm
Resolución	90-400 micrometros (microns)
Precisión de posicionamiento (X/Y)	1.5 microns
Precisión posicionamiento eje Z	1.25 microns
Conectividad	Tarjeta SD
Materiales que soporta	Z-ABS, Z-ULTRAT, Z-HIPS, Z-GLASS, Z-PCABS, Z-PETG
Software	Z-SUITE

Figura 33. Impresora 3D Zotrax m200.

"Fuente: https://zortrax.com"

Los resultados no han decepcionado; lo contrario más bien. El *frame* es resistente y robusto. Puede aguantar perfectamente el peso añadido de los motores, ESCs, batería y dispositivos electrónicos.

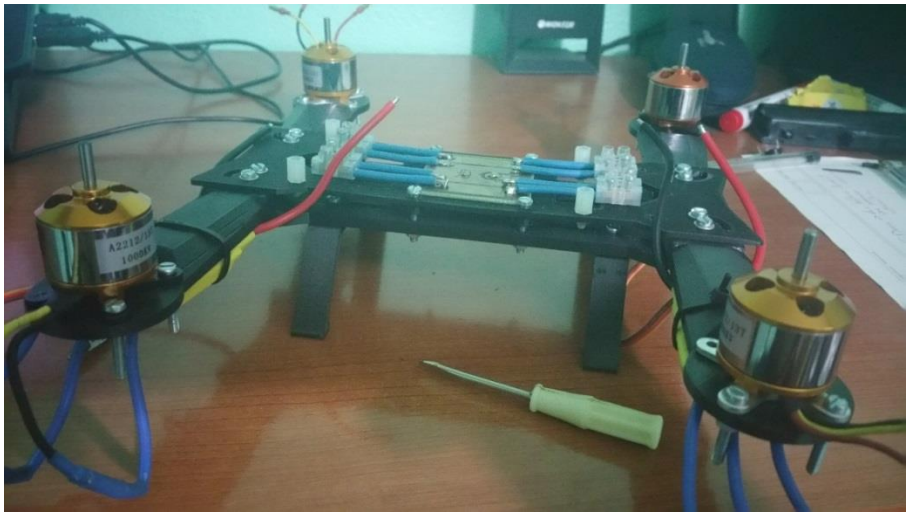


Ilustración 34 Estructura Dron.

"Fuente: propia"

5.2 Diseño electrónico

En este apartado, focalizaremos nuestra atención en las partes relacionadas con la electrónica a nivel de potencia. Esto es, el diseño de circuito impreso para distribución de corriente, la asignación de pines según su utilidad, conexión entre ESCs y motores y el esquema general del sistema.

En cuanto a lo que se refiere por diseño electrónico, comprende la distribución de pines entre cada dispositivo y el esquematizado del circuito impreso orientado a distribuir energía entre los cuatro ESC. En las siguientes tablas están ordenados cada pin con su respectiva conexión entre dispositivos:

Tabla 6. Conexión pines entre Arduino y MPU-6050

Arduino Nano	MPU-6050
3V3	VCC
GND	GND
A4	SDA
A5	SCL
D2	INT

Tabla 7. Conexión pines entre Arduino y HC-05

Arduino Nano	HC-05
5V	VCC
GND	GND
D11	RXD
D12	TRX
D13	EN

Tabla 8. Conexión pines entre Arduino y ESCs

Arduino Nano	ESCs
D7	ESC 1
D6	ESC 2
D5	ESC 3
D4	ESC 4

Tabla 9. Alimentación Arduino Nano

Arduino Nano	UBEC
VIN	+VCC
GND	-VCC

Tabla 10.. Conexión entre batería y UBEC

Batería Li-po	UBEC
VCC	+VIN
-VCC	-VIN

Estos enlaces se han representado esquemáticamente con el software de diseño Proteus. No se han realizado simulaciones con este programa, ya que disponemos de todos los dispositivos. Las esquematizaciones se dividen en dos partes, primero el sistema programable, que incluye el microcontrolador Arduino Nano, el módulo comunicador *bluetooth* HC-05 y el sensor acelerómetro y giróscopo MPU-6050.

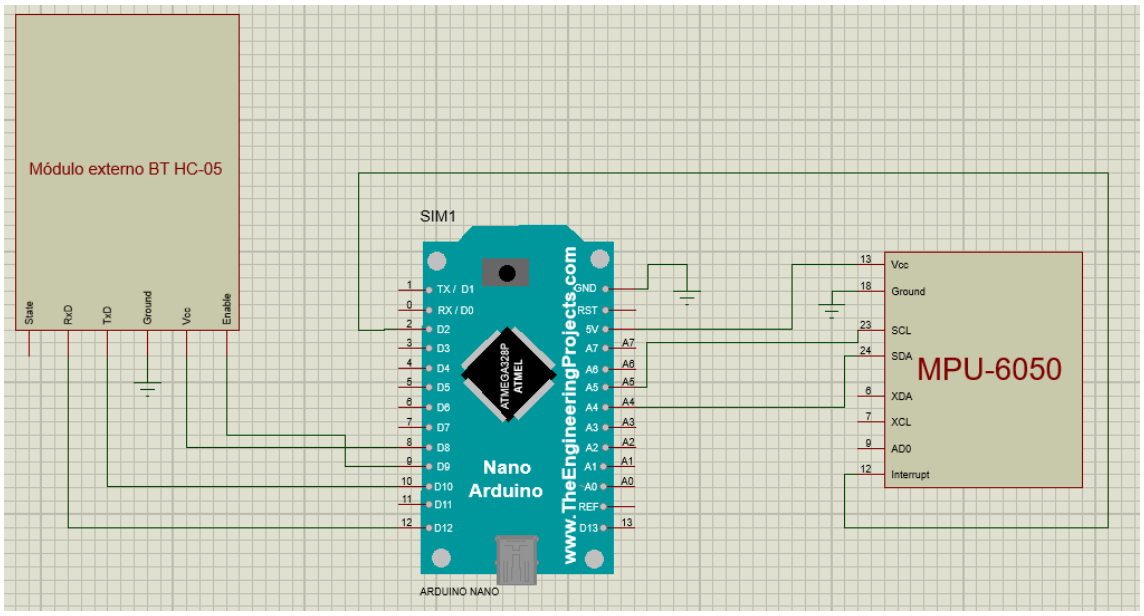


Figura 35. Esquema sistema programable.

"Fuente: propia"

Y segundo, el sistema actuador, que incorpora el microcontrolador conectado a los cuatro reguladores de velocidad. En la imagen de abajo se observa como las conexiones entre los motores y los ESC están alternadas en dos tipos. Esto se debe al sentido de giro que deben aplicar los ESC alimentando los motores. Por suerte, estas uniones son sencillas, ya que los dos cables externos del variador son los responsables de la alimentación eléctrica del motor y, por consiguiente, de su sentido de giro. Mientras que la conexión restante se conecta al cable amarillo, el cual recibe los pulsos velocidad de giro.

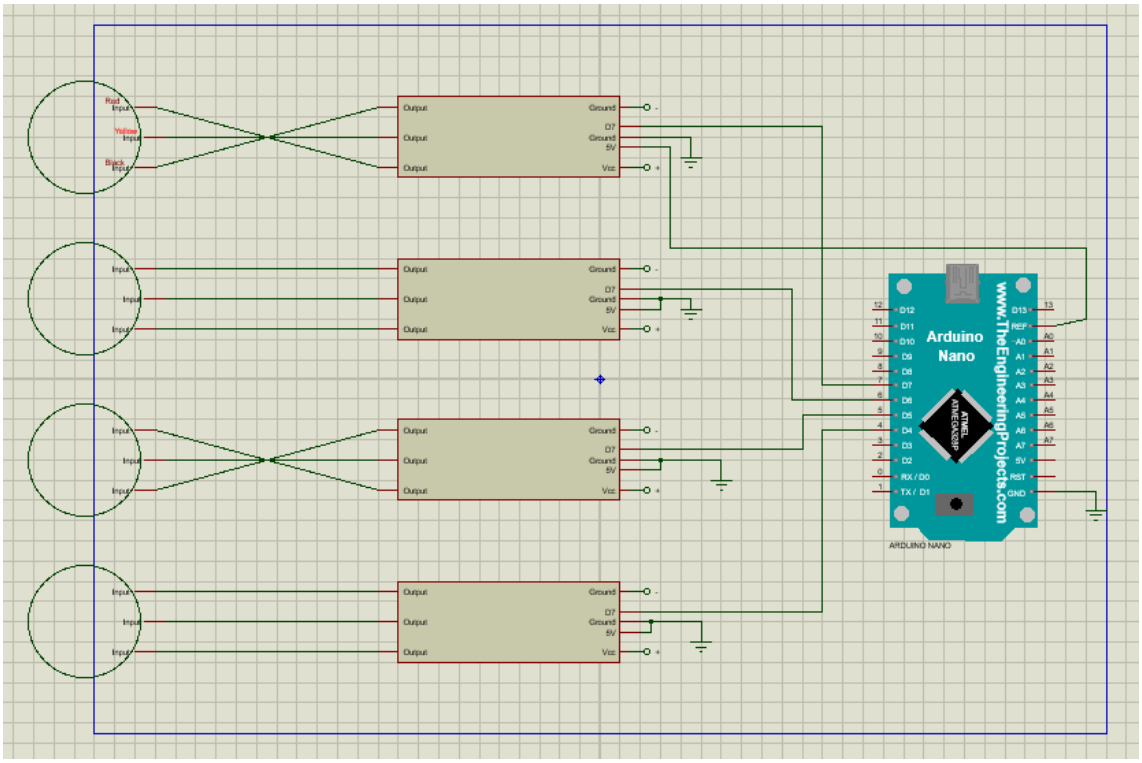


Figura 36. Sistema actuador.

"Fuente: propia"

Por último, en este sub-apartado, mostramos el diseño esquemático de la placa distribuidora de potencia a los variadores de velocidad. Su función es la de unir de manera simple y cómoda los cuatro ESC en un mismo punto desde donde se alimentará todo el sistema.

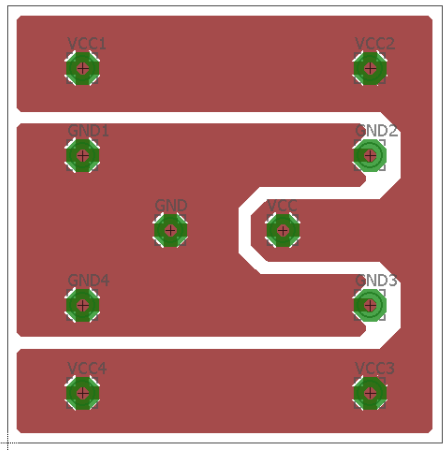


Figura 37. Power distributor board.

"Fuente: propia"

Esta ha sido impresa en una baquelita de tamaño 4x4cm mediante la técnica de planchado. Una vez estamos seguros del diseño definitivo de la placa a imprimir, exportamos el archivo a formato PDF e invertimos la imagen. Después, imprimimos el objeto sobre una hoja transparente del tipo diapositiva, limpiamos el trozo de baquelita y colocamos el circuito sobre ella. Es crucial, a la hora de imprimir el circuito sobre la baquelita, que hayamos invertido la

cara de la imagen impresa. A continuación, introducimos el conjunto en la prensa donde se realizará el planchado térmico. Posteriormente, la pieza debe pasar por un proceso de limpieza usando componentes químicos tales como ácidos y corrosivos para atacar al cobre excedente. Obtuvimos un resultado válido para su fin, que se muestra en la siguiente figura:

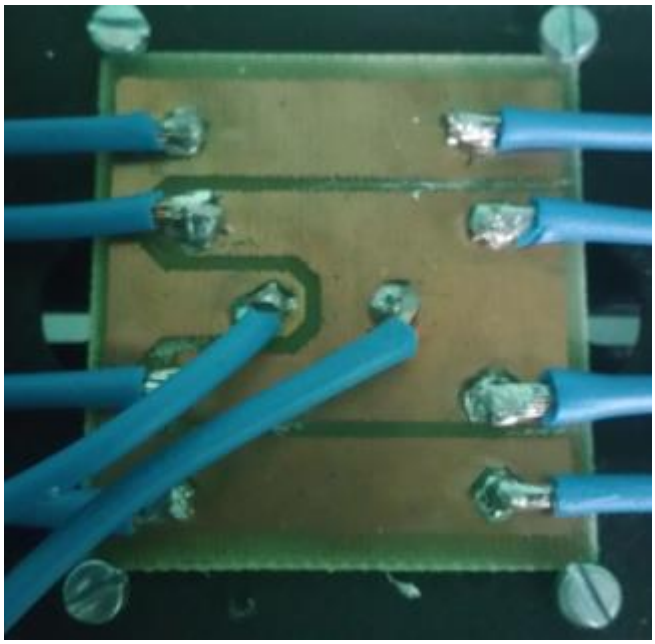


Figura 38. PDB soldada e instalada.

"Fuente: propia"

5.3 Diseño de software

En este sub-apartado incluimos todo el código implicado en el funcionamiento previo y final de los componentes, y del sistema que todos juntos forman. El proyecto está desarrollado por partes. Es decir, se han estudiado todos los módulos y dispositivos externos a la placa Arduino Nano. Esto es, que la programación para cada módulo se ha hecho independientemente. Una vez se ha reunido todo lo necesario, se ha creado un único código. A continuación, se redacta cada una de las funciones y códigos tratados a través de la Arduino IDE; incluido el *software* Android Studio para crear nuestra app de mando radiocontrol.

5.3.1 Calibración ESCs

En apartados anteriores se ha hecho mención de los ESC y algunas de sus características más importantes ligadas a la actividad del proyecto. De modo que procedemos a terminar de explicar su funcionamiento y su uso en el sistema.

Un *Electronic Speed Controller* es un circuito electrónico con la finalidad de variar la velocidad de giro de un motor eléctrico. Se suelen utilizar en modelos de radiocontrol, generando una salida de señal trifásica de bajo voltaje al motor; el ESC es alimentado directamente desde la batería. El control de estos dispositivos se realiza mediante modulación

por ancho de pulsos desde la unidad de control; Arduino Nano. El concepto de PWM, es una señal de onda cuadrada que ocurre en dos términos, HIGH y LOW. Respectivamente, son señales de 5 y 0 voltios de cierta duración. En esta figura podemos ver las distintas formas de la señal cuadrada dependientes de la duración del pulso; también llamado *duty cycle* o ciclo de trabajo.

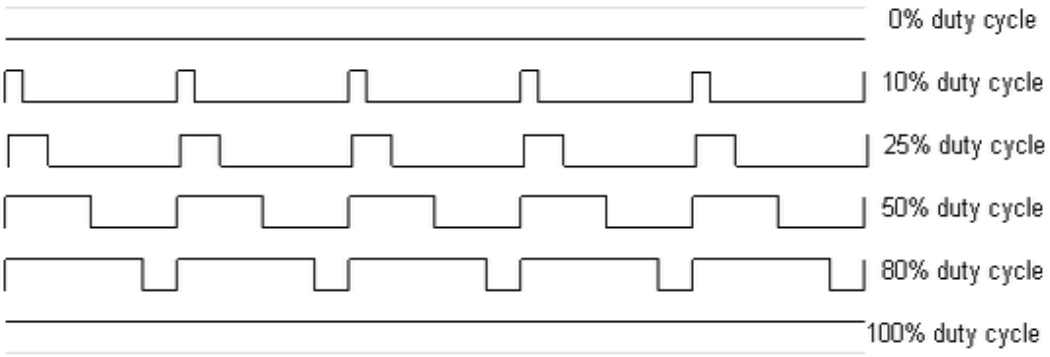


Figura 39. Duty Cycle PWM.

"Fuente: https://www.arduino.cc/en/Tutorial/SecretsOfArduinoPWM"

Para manejar los ESC, es necesario seguir un protocolo. En primer lugar, el ancho de los pulsos está prefijado; en este caso, ancho de 1 a 2 milisegundos(ms). De forma que necesitan de un calibrado previo. Consiste en establecer el máximo y mínimo ancho de pulso, que se traduce en la velocidad del motor proveniente de la señal PWM enviada desde el microcontrolador. Calibrar envuelve la programación de los ESC para asimilar las ondas de modulación por pulsos que corresponden a los valores de máxima y mínima velocidad del mecanismo motriz.

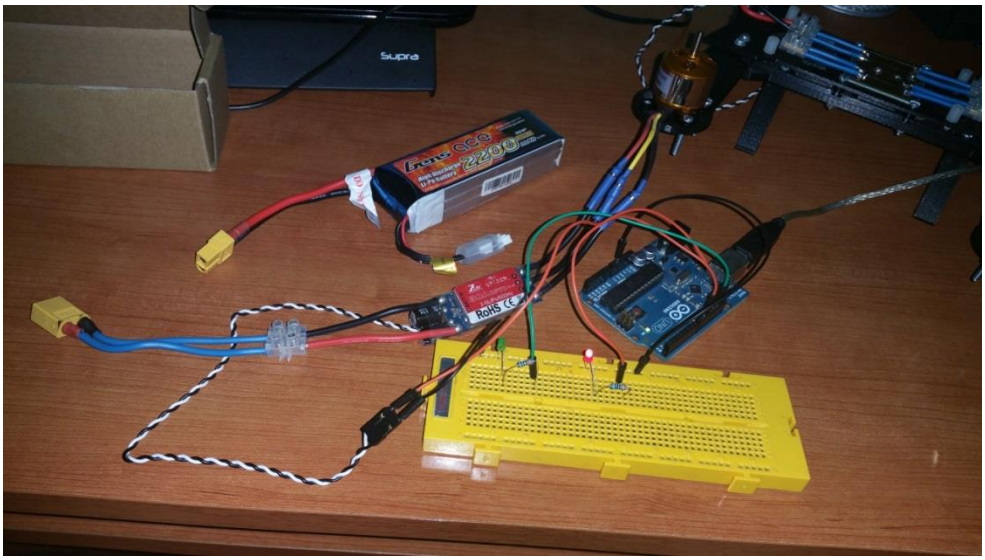


Figura 40. Conexiones para calibración ESC.

"Fuente: propia"

La señal que el ESC lee, es del mismo tipo que la utilizada por un Servo. En otras palabras, la librería Servo.h que incluye el IDE Arduino se puede usar para el calibrado y control de los controladores de velocidad, mediante la función writeMicroseconds de dicha librería. Abajo se expone el código de calibración empleando estos recursos.

```
#include <Servo.h>
Servo ESC;  //Objeto de Servo.h llamado ESC
int throttle = 0;  //Variable para la adjudicación de pulsos
const int verde = 12;
const int rojo = 13;


void setup()
{
  pinMode(verde,OUTPUT);
  pinMode(rojo,OUTPUT);
  ESC.attach(9,1000,2000); /*adjunta la salida de la señal al pin 9.
  Y establecemos los rangos de pulsos 1000 ms y 2000 ms*/

  throttle = 180;  //Pulso al máximo. Son grados para los servos, que ya están delimitados
  //de 0 a 180 entre 1000 a 2000 ms por la función anterior attach()
  ESC.write(throttle); //Adjudica el valor de throttle a la salida
  //A partir de este punto, se conecta la batería
  delay(10000);  //Dejamos un margen de 10 segundos para conectar la batería
  redBlink();  //Pasa a la función de parpadeo del led rojo
  throttle = 0;  //Pulso igual al mínimo --> 1000
  ESC.write(throttle);
  delay(10000);
  throttle = 90;  //Posición neutra
  ESC.write(throttle);
  delay(10000);   //El ESC debería estar calibrado
}
```

Figura 41. Sketch calibrado ESC.

"Fuente: propia"

El funcionamiento del código es bastante simple. Primero, la señal máxima es enviada al ESC desde el microcontrolador sin tenerlo alimentado. Damos corriente manualmente, se oirán una especie de pitidos desde el dispositivo y el programa cambia a la señal mínima. Seguido de los pitidos de confirmación, el *sketch* envía un pulso en posición neutra. Finalmente, el regulador de velocidad emite una serie de pitidos de confirmación, entonces sabemos que está calibrado. Sin embargo, nosotros añadimos unos leds de colores distintos para determinar en qué fase del calibrado se encuentra el proceso. Se muestran los componentes descritos en la Figura 38.

5.3.2 Configuración y función hc-05

El módulo *bluetooth* HC-05 es el que nos permite comunicar desde la Android app las órdenes establecidas para los movimientos del *quadcopter*. Aunque de él depende, en gran parte, que el VANT se mueva a nuestro antojo, su programación es bastante sencilla.

```
//Libreria de comunicacion serie
#include <SoftwareSerial.h>
//HC-05 setup
#define RxD 10
#define TxD 11
#define RST 8
#define KEY 9

//Comunicacion serie con el bluetooth
SoftwareSerial BT(RxD,TxD); //10 RX, 11 TX.*/

void setup() {
 //Iniciar lectura Bluetooth y puertos

    pinMode(RST, OUTPUT);
    pinMode(KEY, OUTPUT);
    digitalWrite(RST, LOW);
    //Modo comunicacion
    digitalWrite(KEY, LOW);
    //Enciendido Módulo pin 8
    digitalWrite(RST, HIGH);
    //Configuramos el puerto serie por software para comunicar con el HC-05
    BT.begin(38400);
    Serial.flush();
    delay(500);
    //Configuramos puerto serie para debug
    Serial.begin(38400);
```

Figura 42. Sketch configuración HC-05.

"Fuente: propia"

En la imagen, está escrito el código correspondiente a la configuración del módulo. Esta estructura sirve para fijar en el dispositivo mediante comandos AT un nombre con el que reconocerlo, una contraseña que permita vincular un dispositivo Android al módulo, establecer un modo de conexión, (esclavo o maestro), la velocidad de comunicación entre dispositivos, y otros comandos que pueden ser de utilidad. Cada comando AT se envía por mediación de monitor serie de la IDE Arduino. Primero insertamos la librería SoftwareSerial.h, incluida en la IDE Arduino, y definimos qué pines juegan el papel de comunicador y receptor (RX y TX). Segundo, alimentamos el módulo externo usando un pin digital e inicializamos un puerto serie correspondiente a la vinculación *bluetooth* para la recepción de datos desde el dispositivo Android.

Los comandos que soporta son:

- Prueba de funcionamiento:
  - **Envíar:** AT
  - **Recibe:** OK
- Configurar el Baudrate:
  - **Envíar:** AT+BAUD<Numero>
  - El parámetro número es un caracter hexadecimal de '1' a 'c' que corresponden a los siguientes Baud Rates: 1=1200, 2=2400, 3=4800, 4=9600, 5=19200, 6=38400, 7=57600, 8=115200, 9=230400, A=460800, B=921600, C=1382400
  - **Recibe:** OK<baudrate>
- Configurar el Nombre de dispositivo Bluetooth:
  - **Envíar:** AT+NAME<Nombre>
  - **Recibe:** OKsetname
- Configurar el código PIN de emparejamiento:
  - **Envíar:** AT+PIN<pin de 4 digitos>
  - **Recibe:** OK<pin de 4 digitos>
- Obtener la version del firmware:
  - **Envíar:** AT+VERSION
  - **Recibe:** Linvor1.8

Figura 43. Listado comandos AT.

"Fuente: datasheet módulo HC-05"

El programa utilizado en la secuencia siguiente de código, corresponde a la adquisición de datos y su clasificación según carácter recibido. La idea central es recibir datos, y con esa información que realice las acciones programadas para dicha referencia. Debido a la dualidad del control de vuelo que ofrece nuestro sistema, se condicionan *setpoints* con el fin de obtener la actuación deseada. Primero, el modo Manual o Acrobático. Su dato de entrada en el control PID es la velocidad angular actual del robot, su *setpoint* se trata de la velocidad angular que mandemos desde la app radiocontrol, y la salida es el pulso que se aplicará a su respectivo eje. Segundo, y por último, el modo Estable. Este control de vuelo, es el que usaremos con más frecuencia debido a la estabilidad en el vuelo que ofrece. Aquí es donde entra nuestro control en cascada, ya que se necesita de un refinamiento de los datos para la acción de control, con el fin de alcanzar una estabilidad excelente en el cuadricóptero. Su entrada es la inclinación, en grados, actual del sistema, el punto objetivo a alcanzar viene dado por en el comando RC, y por último la salida ofrece un valor correspondiente a la velocidad angular. Por supuesto, esta salida se trata de la entrada del PID consecutivo que ha sido implementado, cuya función sigue los mismos pasos que el control de vuelo Acrobático.

Como podemos observar en el código localizado en el anexo, se hace uso de las funciones *'if'*, de condición, permitiendo al programa seguir o pasar a otra secuencia si no hay información obtenida desde el módulo. En caso de tomar un dato, las condiciones deben elegir

qué movimiento va a realizar el dron según el valor del mismo y el modo de vuelo seleccionado en la etapa de inicialización. Por consiguiente, pasa a la llamada de las funciones del control PID con su respectivo ángulo deseado, o *setpoint*, mediante la función actualizar_control. Los *setpoint* previos a la condición *'if'* son enviados al control PID en caso que no se reciba carácter alguno desde la app Android. Esto se traduce en una substitución a las palancas de un mando RC, ya que mantiene el punto objetivo fijo hasta que se quiera cambiar, y así controlar con mejor fluidez su desplazamiento. El punto objetivo es el último que se haya llamado, generando que el PID calcule la manera de mantenerlo en ese punto hasta actualizar la orden. En la figura siguiente, se muestra un ejemplo en el que, si el caracter 'b' es recibido, el movimiento a ejecutar implica una variación en la inclinación del eje Pitch. Según qué modo de vuelo haya sido escogido aplicará un *setpoint* u otro; además de dejar los pertenecientes a los otros ejes a cero, y evitar futuras confusiones al controlador.

```
if(comando == 'b'){ //Pitch +

  if(modo_vuelo == ESTABLE){
  AccelP_Setpoint = 30.0;
  last_accelSetpointP = 30.0;
  last_accelSetpointR = 0.0;
  last_yawSetpoint = 0.0;
  }
  else if(modo_vuelo == ACROBATICO){
  GyroP_Setpoint = 50.0;
  last_gyroSetpointP = 50.0;
  last_gyroSetpointR = 0.0;
  last_yawSetpoint = 0.0;
}
```

Figura 44. Ejemplo setpoints y condición modo de vuelo.

"Fuente: propia"

La función encargada del modo de vuelo, se encuentra la última en la etapa de inicialización del sistema. Para esta selección, enviamos un carácter desde nuestra app Android que adjudica un entero guardado en las constantes ESTABLE y MANUAL, cuyo valor determina cuál de los dos modos nombrados debe proceder. El programa queda en bucle hasta que el valor entero de la variable modo_vuelo es mayor de cero imprimiendo en el monitor serial el modo establecido.

5.3.3 Inicialización y transformación MPU6050

Este dispositivo es el más complicado de programar e inicializar de todos los utilizados, aunque curiosamente sencillo de conectar al microcontrolador. Como se ha explicado con anterioridad, esta IMU mide aceleración y velocidad angular en tres ejes distintos; X, Y y Z.

Estas mediciones son utilizadas para que el control PID haga los cálculos necesarios con la finalidad de estabilizar el sistema en pleno vuelo.

Procedemos a la explicación íntegra de la programación del MPU-6050 para nuestro propósito. Medir cambios en la inclinación del *quadcopter*, transformar esos datos y usar un filtro complementario para conseguir eliminar el ruido causado por las mediciones del acelerómetro y el *drift*, que se traduce como una acumulación de error del giroscopio, el cual no mantiene su punto de partida como sí hace la medición del acelerómetro. Este filtrado es fácil de definir, tiene un bajo coste de procesamiento y ofrece una alta precisión en las medidas. Consiste en la unión de dos filtros, un *High-pass filter(*filtro paso-alto*)* para el giroscopio y un *Low-pass filter*(filtro paso-bajo) para el acelerómetro. El primero deja pasar únicamente valores por encima de cierto límite, al contrario que el filtro paso-bajo, que solo permite a los que están por debajo del límite. Dicho esto, empecemos con la configuración para la lectura predeterminada que realizan los sensores.

Antes de empezar, cabe destacar que hemos hecho uso de librerías de terceros, ya que la configuración de este dispositivo es ardua y carecemos del tiempo para dedicarnos a crear una propia. La primera línea incluye la librería MPU6050.h, necesaria para la obtención y conversión de datos, seguida de la librería I2cdev.h, importante si deseamos una correcta comunicación a la placa Arduino. Y por último, incluimos la librería Wire.h, responsable del éxito en la interacción mediante el protocolo de comunicación I2C. Creamos una instancia de la librería MPU6050.h para su uso en la obtención y adjudicación de dato. Seguido se definen los ratios de conversión, que también establece la documentación. Estas constantes dividen los valores que nos den el Acelerómetro y el Giroscopio para conseguir unos datos coherentes en la medición. RAD_A_DEG es la conversión de radianes a grados. La IMU entrega valores en enteros de 16 bits, como Arduino los guarda en menos bits, hay que declarar las variables que almacenan los enteros provenientes del MPU-6050 como el tipo de enteros int16_t. Las declaradas en este tipo de entero son los valores sin tratar (raw) de los sensores.

En la etapa de *setup* referente a la IMU, se inicia la comunicación I2C con el dispositivo, ya que la librería I2cdev.h no lo hace automáticamente. Entonces, inicializa la comunicación I2C, la verifica, y configura la escala de medición de los dos sensores según nuestros intereses. MPU6050_GYRO_FS_2000 significa que el máximo valor de velocidad angular que lee es de ±2000 grados/segundo, que equivalen a una resolución de 16 bits, siendo ±32768 en valores de medidas *raw*. MPU6050_ACCEL_FS_4 es el rango de escala del acelerómetro, configurado a ±16g.

Seguido encendemos el *Digital Low Pass-Filter* (DLPF) con la idea de remover parte del ruido de la señal del sensor. Elegimos la configuración 0x05, para que el DLPF filtre y el

ratio de salida de datos sea de 1kHz. Esta filtración está ligada al nivel de rotación de los motores del dron. La siguiente configuración se trata de la frecuencia usada en la eliminación del ruido del filtrado, con la cual podemos impedir cualquier tipo de vibración y el ruido que conlleva para la obtención de medidas.

Para finalizar con la etapa del *setup*, establecemos los *offset* obtenidos mediante la suma de distintas medidas y calculando la desviación media que tiene el sensor. Estos *offset* mejoran la calidad de la medición, de manera que cuando el dispositivo esté estable y paralelo al suelo, las medidas de los sensores marquen cero.

Una vez entendido el método de comunicación, configuración y obtención de datos desde el MPU-6050, solo quedan los cálculos y transformaciones que eventualmente usará el control PID para cumplir su función.

```
//-------------------Lectura y acondicionamiento lecturas MPU-6050--------------//
void leerMPU()
{
   mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

    //De valores raw a grados
    float accYangle_raw = atan((ax/A_R)/sqrt(pow((ay/A_R),2) + pow((az/A_R),2)))*RAD_TO_DEG;
    float accXangle_raw = atan((ay/A_R)/sqrt(pow((ax/A_R),2) + pow((az/A_R),2)))*RAD_TO_DEG;

    //Obtencion valores de inclinacion del Gyro para su uso en la funciones PID
    rollGyro = (gx)/GYRO_ESCALA;
    pitchGyro = (gy)/GYRO_ESCALA;
    yawGyro = (gz)/GYRO_ESCALA;

    accPitch = (0.1 * -accXangle_raw) + (0.9*(accPitch + (pitchGyro*tiempoCiclo)));
    accRoll =  (0.1 * accYangle_raw)  + (0.9*(accRoll  + (rollGyro*tiempoCiclo)));
}
```

Figura 45. Obtención y transformación medidas MPU-6050.

"Fuente: propia"

Aquí pasamos a la función leerMPU, llamada al inicio del bucle, donde se transforman las mediciones *raw* en las unidades correspondientes a la aceleración y la velocidad angular. Esta función empieza con el cálculo de los ángulos X e Y del acelerómetro en grados, pero sin pasar por el filtrado. Estas son las variables donde quedan guardados, accYangle_raw y accXangle_raw. Seguido, se obtienen las cifras del giroscopio con un cálculo basado en los *offset*s determinados en cada eje y la división de la constante GYRO_ESCALA. Que se encarga de escalar el dato en *raw*, ya que como hemos dicho con anterioridad el giroscopio está configurado en su máxima resolución y lecturas de 16 bits. Resultando en una lectura escalada en grados por segundo óptima para su uso en el control del PID en cascada. Y por último, el Filtro Complementario, en el cual intervienen las lecturas acondicionadas de ambos sensores. Debido a que el acelerómetro sufre alteraciones en las medidas por el ruido y las vibraciones del sistema. El resultado de la ecuación se aplica en el sistema de control PID.

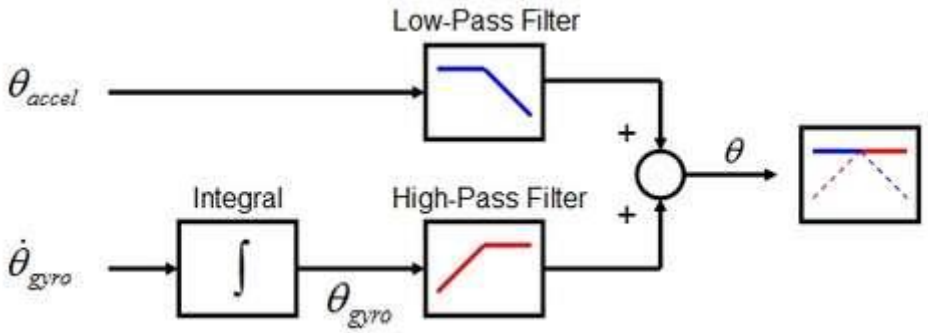


Figura 46. Esquema filtro complementario.

"Fuente: http://robotics.stackexchange.com/questions/1717/how-to-determine-the-parameter-of-a-complementary-filter"

5.3.4 Control PID

En este proyecto, la función principal del algoritmo de control a implementar es la de corregir el error de inclinación resultante en el desplazamiento y rotación del dron. Dicha actividad ocurre en los ejes X e Y; respectivamente, Roll y Pitch. Al mismo tiempo, la rotación sucede en el eje Z, que es el Yaw. Tal control es posible gracias a las medidas transformadas y filtradas del sensor MPU-6050. En la siguiente figura se muestran gráficamente estos conceptos.

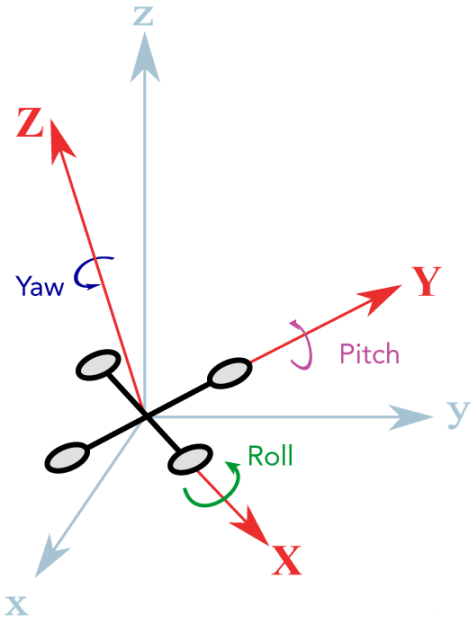


Figura 47. Ejes Yaw-Pitch-Roll.

"Fuente: http://theuav.net/blog/2014/11/19/quadcopter-dynamics-math-p2.html"

Este sistema de control constituye un elemento de regulación para sí mismo y otros sistemas. Como sí tiene en consideración la señal de salida del sistema, es un control en lazo cerrado. Y debido a la inestabilidad del sistema a controlar, necesitaremos este tipo de control,

es decir, aquel en que se tiene en cuenta la señal de salida mediante una realimentación. Esta característica entrega una mayor estabilidad. Los sistemas de control en lazo cerrado se encargan de actuar en función a una entrada de referencia, también llamada *setpoint*. Este punto deseado, se compara con una señal de entrada y el resultado es el error que queremos minimizar. En función de esa señal de error se elabora una acción que enviar a los actuadores con el fin de obtener la respuesta adecuada. Esto es, lograr el equilibrio del robot.

El control PID se divide en tres conceptos, Proporcional, Integral y Derivativo. Dependiendo de la modalidad del controlador algunos de estos valores pueden ser cero. Cada uno de estos actúan en mayor medida para mejorar las características de la salida, pero no es posible obtener un control perfecto. Dicho de otro modo, por mucho que ajustemos las constantes no podremos reducir a cero el tiempo de establecimiento, ni la sobre-oscilación, ni el error, etc. Sino, que se trata de ajustarlo a un término medio cumpliendo las especificaciones mandadas. Se procede a la explicación de cada una de las constantes y del mismo PID:

Proporcional

La acción proporcional es la base de los tres modos de control, los otros (acción integral y acción derivativa) pueden ser sumados a la respuesta proporcional. En otras palabras, los modos de control solo pueden funcionar entre las combinaciones P, PI, PD, o PID. La salida que proporciona un controlador de este tipo, es proporcional a la señal de error. Esta es su ecuación:

$$u(t) = Kp \cdot e(t)$$

Donde u(t) es la acción de control, e(t) el error calculado ente la diferencia de la medición actual y el punto deseado, y Kp es una ganancia proporcional ajustable. Un controlador P puede controlar cualquier planta estable, pero posee un alcance limitado y error en régimen permanente.

Integral

La acción integral da una respuesta proporcional al error acumulado por el controlador P. Elimina el error en régimen permanente nombrado en la acción proporcional. Por el contrario, se alcanza un mayor tiempo de establecimiento, una respuesta más lenta y un mayor período de oscilación que en el caso anterior. Esta es su ecuación:

$$u(t) = Ki \int_0^t e(\tau) d\tau$$

La señal de control u(t) tiene un valor distinto a cero cuando la señal de error e(t) es cero. Por lo que se concluye que, dada una referencia constante, el error en régimen permanente es cero.

Derivativo

La acción derivativa da una respuesta proporcional a la derivada del error, es decir, a la velocidad de variación de la señal del error actuante. Si se incluye esta acción de control a las anteriores se disminuye el exceso de sobre-oscilaciones. Además de aumentar la velocidad de respuesta ante el error.

$$u(t) = Kd\frac{de(t)}{dt}$$

PID

Esta acción combinada reúne las ventajas de las tres acciones de control individuales. Esta es su ecuación:

$$u(t) = Kp \cdot e(t) + \frac{Kp}{Ti}\int_0^t e(\tau)d\tau + Kp \cdot Td\frac{de(t)}{dt}$$

Existen diversos criterios de calibrado para los controladores PID pero ninguno de ellos nos garantiza que encontremos unos valores que sirvan para estabilizar el sistema. Por lo que la mejor opción es el método de prueba y error. Probando parámetros, y en función de la respuesta obtenida variar estos parámetros; esto forma parte de otro tópico que se explica más adelante en la redacción. Con el control PID obtenemos finalmente un sistema que actúa para corregir los errores del sistema. De esta forma, así sería un control PID escrito en lenguaje C/C++:

```
PID(float input, float setpoint, float P, float I, float D){

  error = setpoint - input;
  errInt += error;
  errDer = error - errorAnterior;
  P = Kp*error;
  I = Ki*errInt;
  D = Kd*errDer;
  salida = P + I + D;
  errorAnterior = error;

  }
```

Figura 48. Algoritmo PID básico.

"Fuente: propia"

Este es un código perteneciente a un control PID estándar. El nuestro es vagamente distinto, ya que existen alteraciones en la salida a tener en consideración. Siendo estos, el conocido efecto *Windup* y el denominado *Derivative kick*. El primero, provocado por la parte de Integral del algoritmo PID. Esta alteración tiene como origen la limitación de los actuadores o su saturación, ya que el control PID no conoce qué límite puede alcanzar su respuesta y genera

saturación en la salida. Este efecto, se produce cuando el cálculo integral alcanza cifras muy altas generadas por su acumulación de error. De manera que, para errores menores la actuación sobre estos es mayor a la que debería.

```
AccelP_iErr += Ki * AccelP_error;
AccelP_iErr = constrain(AccelP_iErr, PID_MIN, PID_MAX);
double derInput = angulo - accelP_antInput;
double out = Kp*AccelP_error + AccelP_iErr - Kd*derInput;
out = constrain(out, PID_MIN, PID_MAX);
```

Figura 49. Antiwindup PID.

"Fuente: propia"

Para corregir este efecto en el sistema, como se muestra, se ha añadido en el código PID dos funciones *constrain*, incluida en la IDE Arduino, limitando la salida y la acumulación del error Integral para eliminar la posible saturación. La primera afecta al error integral, que estará limitado a los valores que consideramos máximos para la saturación de los actuadores, ya que la salida de este control va a ser la variable que determine la corrección de la velocidad de giro de cada motor respecto al desplazamiento elegido. Y la segunda actúa sobre los valores máximos del valor de la acción de control, que también están limitados al valor de saturación conveniente.

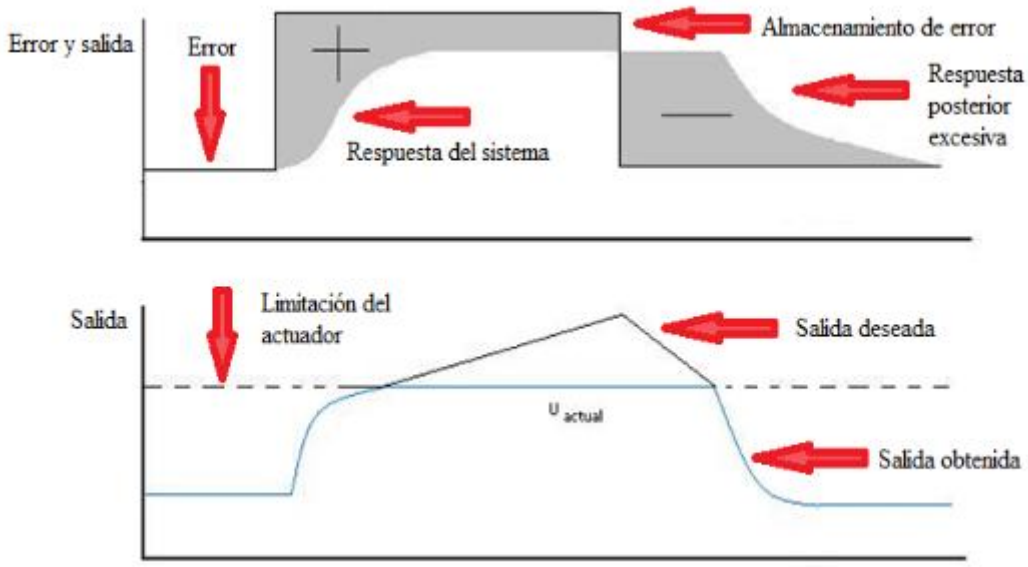


Figura 50. Efecto Windup PID.

"Fuente: https://controls.engin.umich.edu/wiki/index.php/PIDDownside"

En segundo lugar, debemos eliminar el fenómeno llamado *Derivative kick*. Este efecto tiene que ver con el cálculo de la acción derivativa cuando el sistema se somete a cambios bruscos en la referencia, como entradas tipo escalón por la fuerte señal de

corrección que genera. Específicamente, un error grande entre un intervalo de tiempo muy pequeño hace que tienda a infinito durante un breve periodo de tiempo. Los picos resultantes, tienen efectos negativos que provocan desgaste en los componentes mecánicos.

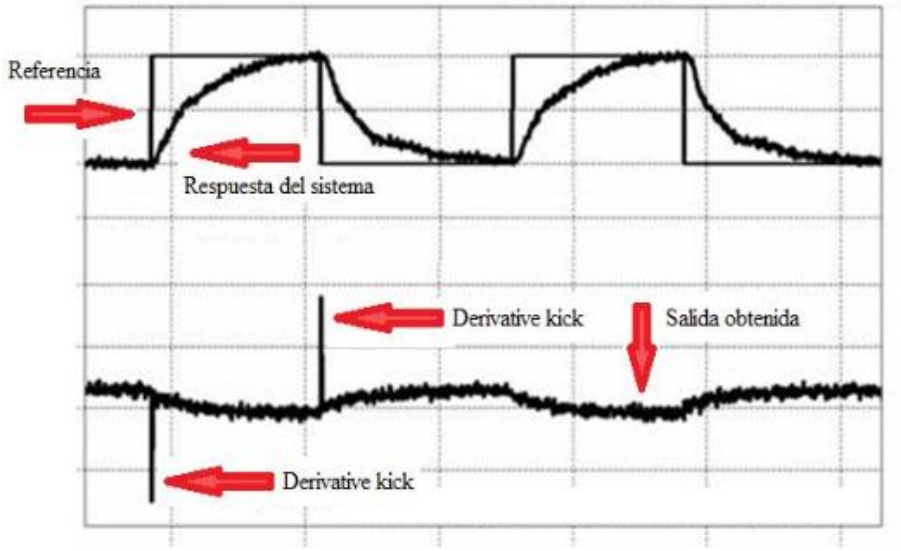


Figura 51. Derivative kick.

"Fuente: http://controlguru.com/pid-control-and-derivative-on-measurement/"

Después de entender el porqué de evitar este comportamiento indeseado, procedemos a ponerlo en práctica. Como el error es igual a la diferencia entre el punto que queremos alcanzar y la entrada del sistema, cualquier cambio en este punto objetivo causa un cambio instantáneo en el error. La derivada de este cambio tiende a infinito, ya que la derivada del tiempo no es cero y esto genera un escalado a un número mucho mayor. Este número se obtiene en el cálculo de la acción PID, que resulta en un pico alto en la salida; que tiende a infinito. De modo que, para deshacernos de este resultado del todo indeseado, realizaremos un cambio en las variables que intervienen en el cálculo de la acción derivativa.

$$\frac{dError}{dt} = \frac{dSetpoint}{dt} - \frac{dInput}{dt}$$

$$\frac{dError}{dt} = -\frac{dInput}{dt}$$

Esta ecuación representa la obtención del error. En nuestro sistema, el *setpoint* solo cambia cuando comandamos la orden de desplazar el dron, siendo constante la mayor parte del tiempo. La solución, es la que se muestra en la segunda ecuación mostrada, excepto cuando el objetivo a alcanzar cambia. El término derivativo es calculado con la multiplicación de la diferencia entre la entrada actual y la anterior, y la constante derivativa. En la siguiente figura se

muestra nuestro código PID con las líneas correspondientes a la corrección del fenómeno comentado:

```
float PID_AccelP(float angulo, float setpoint, float Kp, float Ki, float Kd){

    float AccelP_error = setpoint - angulo;
    AccelP_iErr += Ki * AccelP_error;
    AccelP_iErr = constrain(AccelP_iErr, PID_MIN, PID_MAX);
    double derInput = angulo - accelP_antInput;
    double out = Kp*AccelP_error + AccelP_iErr - Kd*derInput;
    out = constrain(out, PID_MIN, PID_MAX);
    accelP_antInput = angulo;
    return out;
  }
```

Figura 52. Algoritmo de control PID Eje Pitch.

"Fuente: propia"

La función PID llamada depende del eje y del desplazamiento comandado, en este caso se habría ordenado un movimiento en el eje Pitch, o sea, el eje Y. La salida calculada, es la referencia para el PID en cascada. Esto se define como la configuración donde la salida de un controlador de realimentación es el *setpoint* para otro controlador de realimentación; al menos en este caso solo hay uno. El control en cascada involucra sistemas ordenados uno dentro del otro. El principal propósito es la eliminación de algunas perturbaciones generando una acción de control del conjunto más estable y más rápida. En el primer PID, manejamos los valores del filtro complementario como entrada y la inclinación deseada como punto de ajuste, es decir, las órdenes del mando RC. Establece una acción de control, en este caso es la velocidad angular que los actuadores deben alcanzar para llegar al punto deseado. Aunque no es la corrección que se aplica, sino el punto objetivo del PID en cascada. Esto es, eliminar el error sobre la velocidad. Por consiguiente, el sistema de control nos ofrece la acción de control a aplicar en el eje que esté funcionando el desplazamiento del *quadcopter*. El resultado se almacena en una de las variables Roll, Pitch o Yaw según la acción requerida, para ser implementado en una serie de ecuaciones que decretan el ancho de pulso que aplicara cada ESCs en su respectivo motor, con el fin de alcanzar el punto deseado.

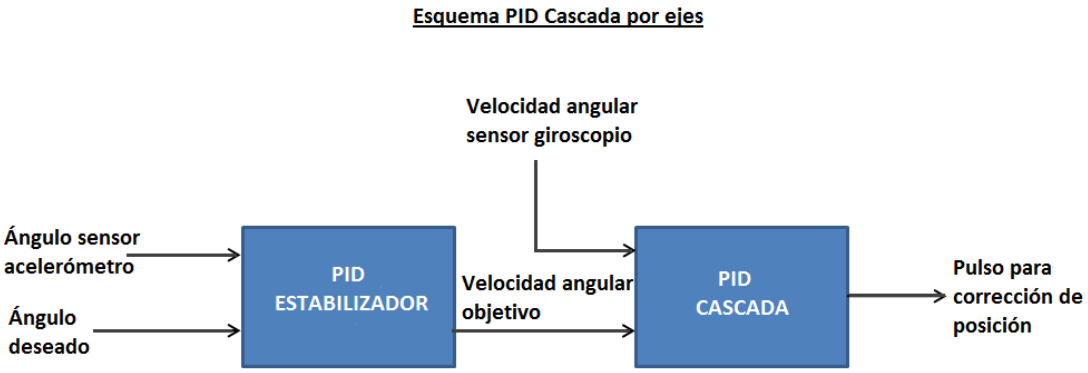


Figura 53.. Esquema PID en Cascada.

"Fuente: propia"

En la imagen consecutiva se encuentra el código referente a la llamada a las funciones PID correspondientes a cada eje y datos del sensor MPU-6050, además que se incluye una condición que elige la acción PID a implementar según el tipo de control a radiocontrol escogido. Consecutivamente se llama a estas funciones con el fin de obtener las salidas del control que se añaden a las ecuaciones que se muestran en la figura número 66.

```
//---------------Actualización valores de pulso adjudicados a cada ESC---------------//

void actualizar_control(){

  //Modo ESTABLE y ACROBATICO

    if(modo_vuelo == ESTABLE){
    //Llamamos las funciones del primer PID
      GyroP_Setpoint = PID_AccelP(accPitch, AccelP_Setpoint, P_Paccel, I_Paccel, D_Paccel);
      GyroR_Setpoint = PID_AccelR(accRoll, AccelR_Setpoint, P_Raccel,I_Raccel, D_Raccel);


    //Seguido llamamos a las funciones PID del control en cascada
      roll = PID_GyroR(rollGyro, GyroR_Setpoint, P_Rgyro, I_Rgyro, D_Rgyro);
      pitch = PID_GyroP(pitchGyro, GyroP_Setpoint, P_Pgyro, I_Pgyro, D_Pgyro);
      //Llamamos al control PID del eje Yaw
      yaw = PID_Yaw(yawGyro, YawSetpoint, P_yaw, I_yaw, D_yaw);
    }
    if(modo_vuelo == ACROBATICO){

      //este modo necesita una adaptación de los setpoint del modo estable al
      // modo acro, es decir, valores de velocidad angular como setpoint
      roll = PID_GyroR(rollGyro, GyroR_Setpoint, P_Rgyro, I_Rgyro, D_Rgyro);
      pitch = PID_GyroP(pitchGyro, GyroP_Setpoint, P_Pgyro, I_Pgyro, D_Pgyro);
      //Llamamos al control PID del eje Yaw
      yaw = PID_Yaw(yawGyro, YawSetpoint, P_yaw, I_yaw, D_yaw);
    }
```

Figura 54. Llamada funciones PID.

"Fuente: propia"

También se necesita de una inicialización para las variables de error utilizadas en el control automático. Esta función solo se llama una vez, ya que si los valores de error fuesen cero todo el tiempo no habría error del que preocuparse, y el control PID quedaría inservible.

Como se ha explicado, el sensor acelerómetro en el eje Z solo es capaz de obtener la aceleración de la tierra, aunque este valor nos permita calcular la inclinación en los ejes X e Y. Sin embargo, el sensor giróscopo sí es competente a la hora de medir la velocidad de rotación del eje Yaw. Con el fin de mover el sistema correctamente en los distintos ejes, el PID que controla el eje Yaw solo trata con los datos provenientes del giroscopio.

```
float PID_Yaw(float gyro, float RC, float Kp, float Ki, float Kd){

    float Yaw_error = RC - gyro;
    iErr_Yaw += Ki * Yaw_error;
    iErr_Yaw = constrain(iErr_Yaw, YAW_PID_MIN, YAW_PID_MAX);
    double derInput = gyro - yaw_antInput;
    double out = Kp*Yaw_error + iErr_Yaw - Kd*derInput;
    out = constrain(out, YAW_PID_MIN, YAW_PID_MAX);
    yaw_antInput = gyro;
    return out;
  }
```

Figura 55. PID Yaw.

"Fuente: propia"

El robot se mantiene estable con el control de los ejes Pitch y Roll, ya que para generar una rotación sobre el eje Yaw no se necesita más que incrementar las revoluciones de cualquiera de las dos parejas de motores que giren en el mismo sentido. Su *setpoint* viene determinado desde la app de radiocontrol. Es el único eje sin un PID en cascada, dado que la posición que alcance no alterará la estabilidad del robot.

5.3.4.1 Configuración constantes PID

Nombradas en el sub-apartado anterior, la variación de cada uno de los valores de estos parámetros influye en la efectividad de la estabilización del VANT. En un sistema como el nuestro, estas constantes son responsables de los siguientes comportamientos:

Coeficiente de la acción Proporcional, Kp:

Este es el valor que priorizamos en el ajuste de parámetros PID. Para ello, dejaremos a cero los dos restantes y alteraremos valores hasta encontrar el óptimo. Cuanto más alto es, más brusca es la reacción ante la corrección de equilibrio. El sistema se vuelve más sensible, y genera sobre oscilaciones a alta frecuencia con el objetivo de eliminar el error. La manera en

que hemos elegido la cifra óptima para esta constante es, aumentar el valor de Kp hasta que empiece a sobre oscilar, entonces lo reducimos un poco.

Coeficiente de la acción Integral, Ki:

Aquí necesitamos tener presente la función integradora, que es la de integrar el error en el tiempo; cuanto más persista el error, mayor es la fuerza aplicada para eliminarlo. Cuando el término Ki no es cero, a veces el error puede persistir durante un largo tiempo. Dicho de otro modo, el *quadcopter* intenta reducir el error de posición, pero el error no disminuye, entonces sigue intentándolo. Para ajustar este coeficiente seguimos el mismo procedimiento que con el anterior. Con valores demasiado altos, empieza a oscilar a baja frecuencia con una alta aceleración. Mientras que, si es menor, al reducir la aceleración sufre tambaleos.

Coeficiente de la acción Derivativa, Kd:

Kd puede ser ignorado en el ajuste de coeficientes, aunque sin él la corrección llevada a cabo por el término proporcional puede llegar a ser demasiado brusca. Este, altera la fuerza de la acción realizada para disminuir el error de posición cuando observa un cambio creciente o decreciente en este error. Actúa como un modo de absorción de brusquedad en el desplazamiento. Un valor alto en este coeficiente genera oscilaciones. La manera implementada de conseguir la cifra correcta para esta constante, ha sido aumentar el valor de Kd cuando se han manifestado vibraciones. Así no existe necesidad de reducir Kp. En esencia, cuanto más rudo sea el movimiento realizado en la rectificación, mayor será el suavizado de este. La contra es, que esta reacción puede volver al sistema lento y flojo a la hora de aplicar la corrección del error de posición.

Coeficientes de corrección para PID del eje Yaw

Se aplican las mismas instrucciones a seguir en el logro de averiguar qué valor es el conveniente para los parámetros de cada acción de control.

```
//------PID------//

#define pitchPID_KP  1.0
#define pitchPID_KI  0.02
#define pitchPID_KD  0.2
#define PID_MIN  -300.0
#define PID_MAX  300.0

#define rollPID_KP  0.68
#define rollPID_KI  0.025
#define rollPID_KD  0.4
#define PID_MIN  -300.0
#define PID_MAX  300.0

#define pitchCASC_PID_KP  1.3
#define pitchCASC_PID_KI  0.001
#define pitchCASC_PID_KD  0.5
#define CASC_PID_MIN  -300.0
#define CASC_PID_MAX  300.0

#define rollCASC_PID_KP  1.0
#define rollCASC_PID_KI  0.001
#define rollCASC_PID_KD  0.2
#define CASC_PID_MIN  -300.0
#define CASC_PID_MAX  300.0
```

Figura 56. Coeficientes PID.

"Fuente: propia"

Como hemos explicado en las líneas anteriores, se requiere de unos valores para cada coeficiente afines al comportamiento del dron. Esta imagen muestra las ganancias que se aplican en el control PID del eje Pitch tanto en su control en modo Estable como para el modo Acrobático. La razón de esto, es que, aunque el VANT sea simétrico no podemos adjudicar los mismos valores a los dos PID. El control que implementamos en la estabilidad del robot tiene como objetivo mantenerse firme con la menor de las oscilaciones.

5.3.4.2 Test PID

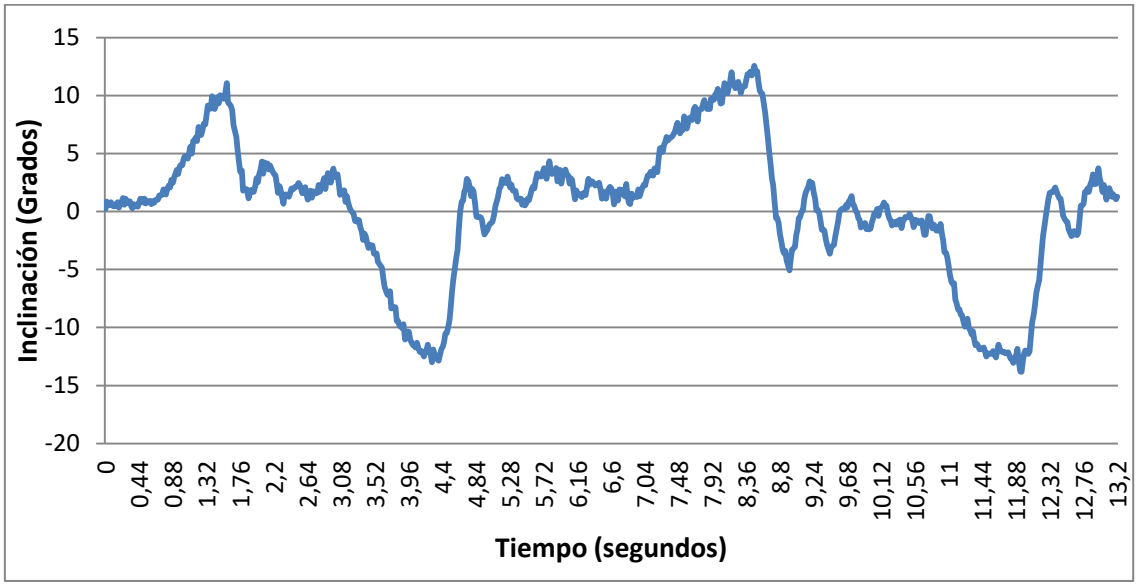
Este tema es primordial a causa del impacto negativo que puede tener en el sistema un control automático mal diseñado y mal configurado. En el código principal se ha añadido una función *debug* con el fin de ver en tiempo real cómo responde el algoritmo ante ciertas entradas mediante el monitor serie del Arduino IDE. En él incluimos las variables que hemos considerado fundamental en la comprobación del buen, o mal, funcionamiento del robot. Éstas son, valores calculados con el filtro Complementario, datos del giroscopio, acción de control computada en cada PID y los pulsos a aplicar en los actuadores.

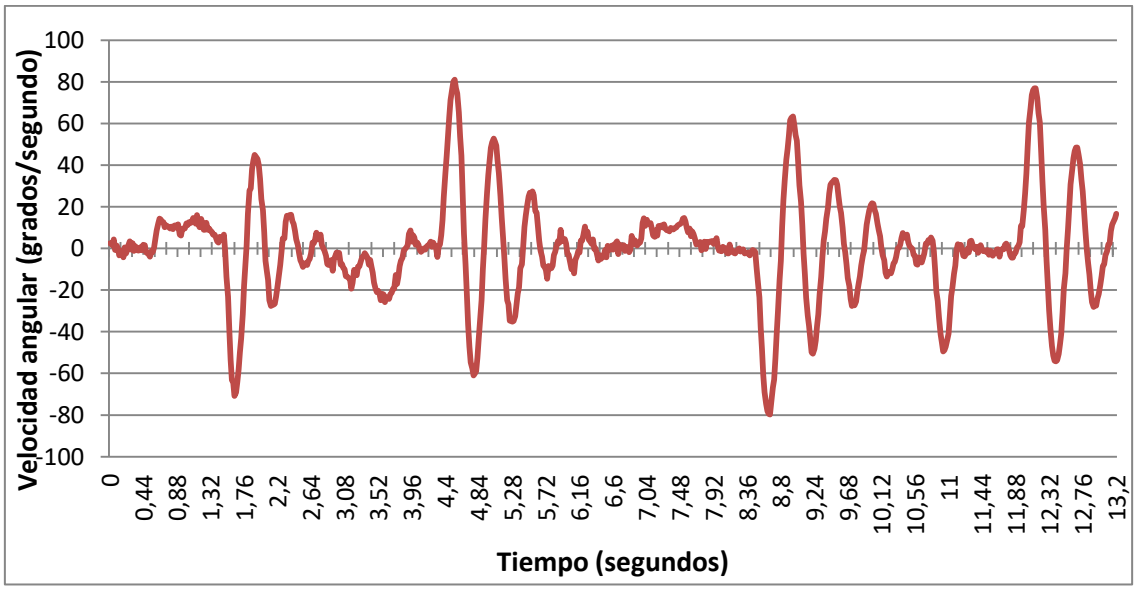
Como muestra de los resultados, se han guardado estas variables en una hoja de cálculo para crear gráficas temporales y poder estudiar las distintas conclusiones, ya sean positivas o

negativas. Para un procesado más sencillo, hemos obtenido datos de los distintos ejes por separado y de los distintos modos de vuelo. Seguido, se presentan las distintas gráficas temporales.

Eje Y, control Pitch:

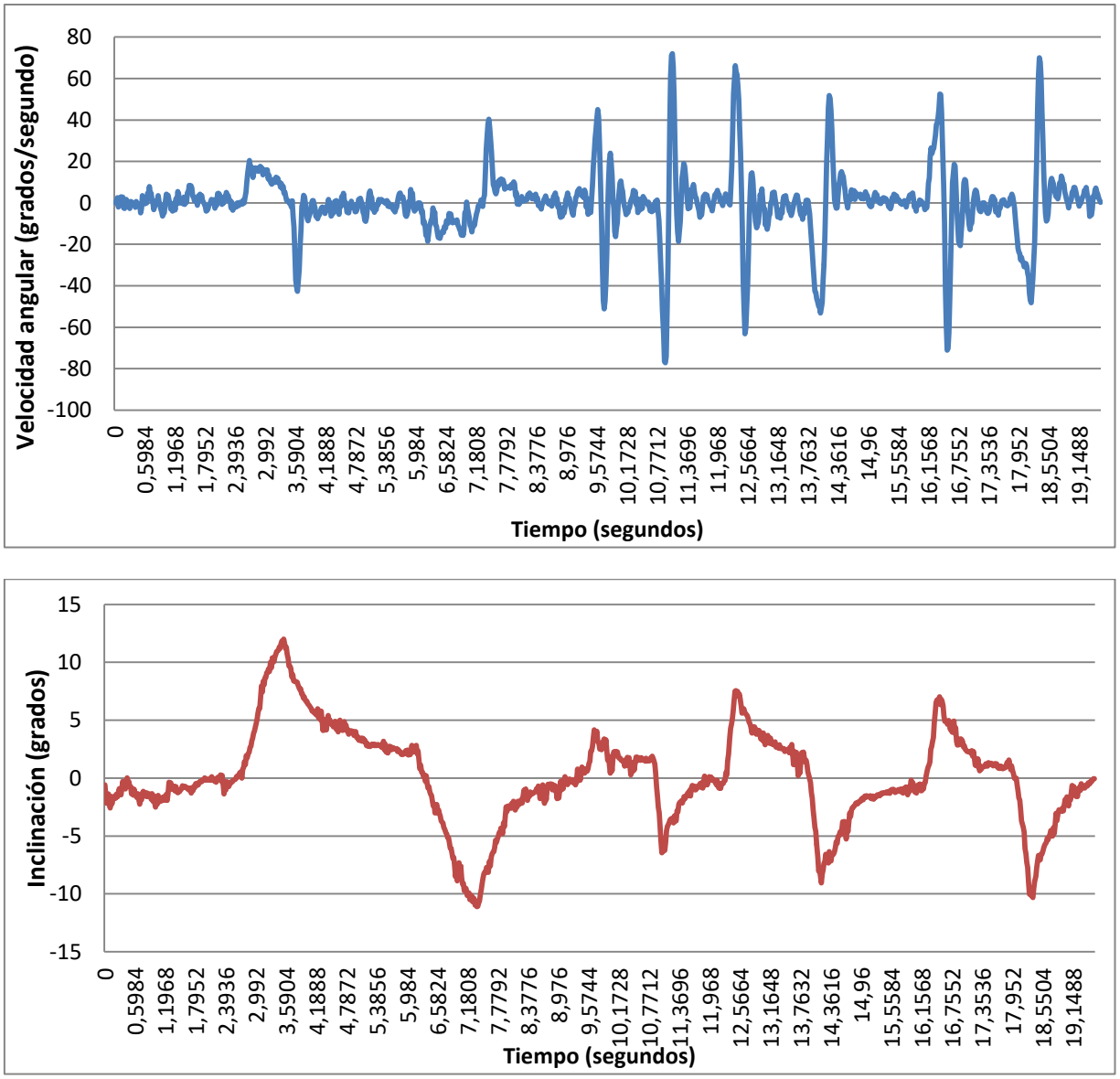
Tabla 11. Gráfica temporal eje Pitch, modo Estable





Recordemos que, el sistema en modo estable fija el error de posición en la medida de la inclinación; mesurada en ángulos. Para obtener los valores representados en esta gráfica, inclinamos el dron, primero hacia delante y esperamos a que se equilibrara en su eje. Y repetimos hacia atrás. De esta forma, se puede ver la rápida corrección llevada cabo.

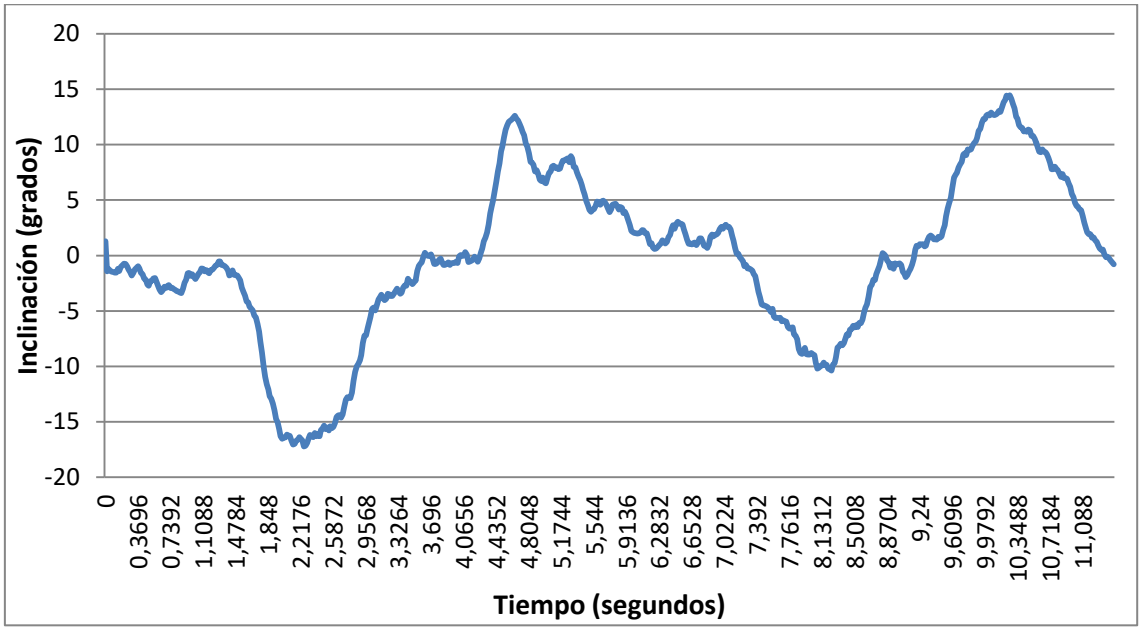
Tabla 12. Gráficas temporales eje Pitch, modo Manual

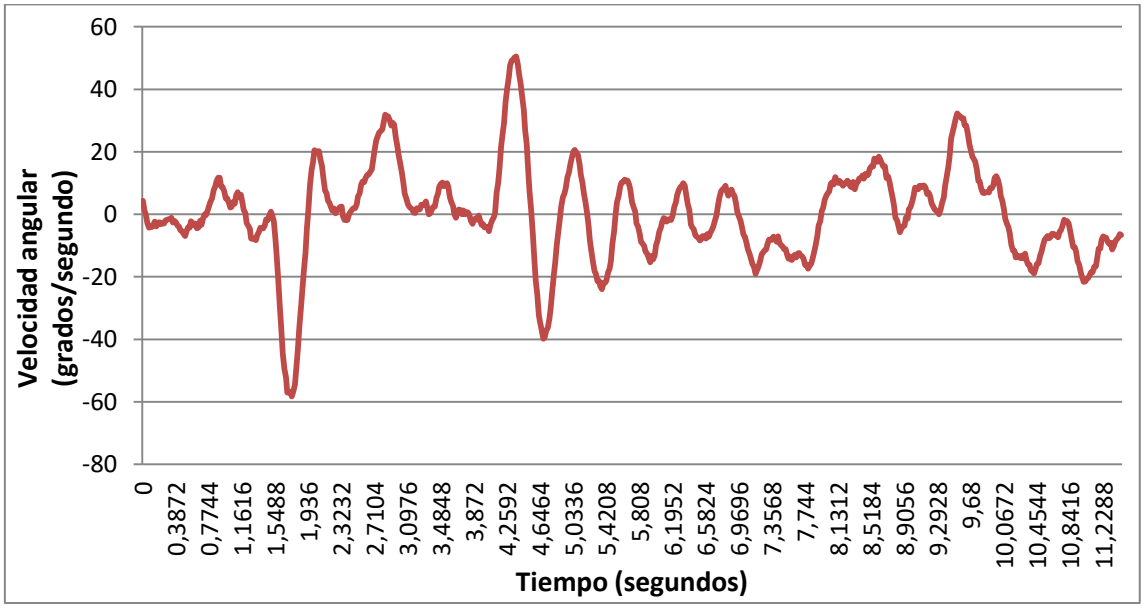


En el estilo de vuelo Manual, recordemos que el error a corregir está medido en ángulos por segundo, es decir, la velocidad angular alcanzada. La primera gráfica, muestra esos mismos valores. Se puede observar la rápida acción de control efectuada, cuando se ha inclinado a la fuerza el VANT sobre el eje Y repetidas veces. En la ilustración siguiente, se aprecia la inclinación en ángulos del robot. Esto es una demostración del funcionamiento visible de este modo, el cual se trata de evitar cualquier cambio forzado por un factor externo en la posición.

Eje X, control Roll:

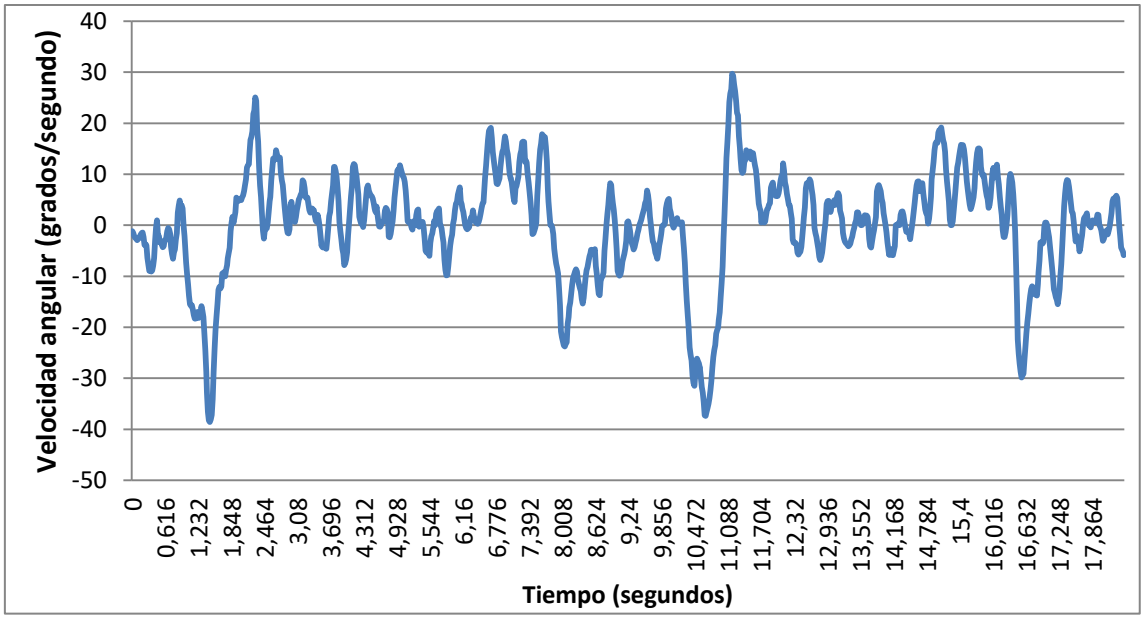
Tabla 13. Gráfica temporal eje Roll, modo Estable

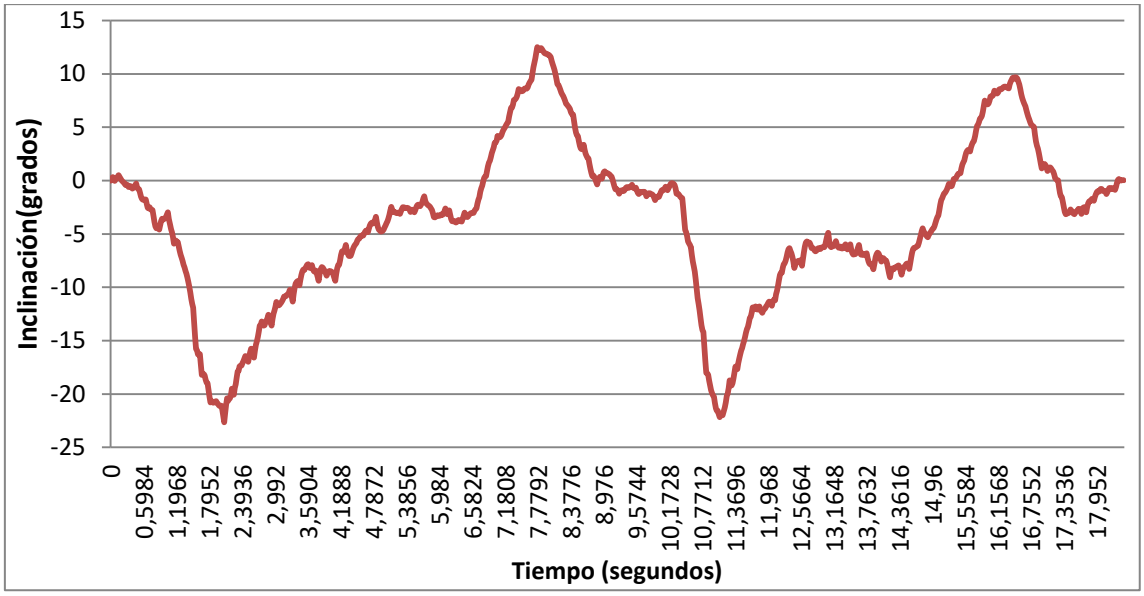




Aquí mostramos la misma información que en la anterior gráfica dedicada al eje Y, pero en el eje X. En nuestra experiencia con el funcionamiento exterior y visible, hemos notado una velocidad de ajuste más rápida del eje Pitch en ambos modos. Aunque esto no ha afectado en absoluto al correcto trabajo del robot.

Tabla 14. Gráficas temporales eje Roll, modo Manual





Para finalizar con las muestras, exponemos las gráficas temporales convenientes al modo Manual del eje X. El comportamiento, en comparación con el del eje Pitch, es similar en los aspectos de inclinación y corrección del error. Forzamos la desnivelación hacia un lado hasta observar estabilidad, y procedemos a ladear el lado opuesto hasta volverse estable. Como se ha explicado antes, aquí las medidas del declive en grados se aprecian de forma más lenta al retorno a su posición original. Sin embargo, es un comportamiento normal, ya que este estilo de vuelo no tiene como función esa característica de buscar una inclinación de cero grados, sino de una actuación firme y sin oscilaciones.

5.3.5 Ecuaciones y envío de pulsos a ESCs

Una vez cosechadas las cifras refinadas desde el sistema de control automático, se aplican al siguiente tramo del código. Aquí se completan las expresiones que se aplican desde el ESC al motor en forma de pulsos de entre 1 y 2 ms; como se ha explicado en el apartado de los reguladores de velocidad.

```
m1 = throttle - pitch + roll - yaw;
m2 = throttle - pitch - roll + yaw;
m3 = throttle + pitch - roll - yaw;
m4 = throttle + pitch + roll + yaw;

if (m1 < MOTOR_MIN) m1 = MOTOR_MIN;  //1200 es la velocidad minima (
if (m2 < MOTOR_MIN) m2 = MOTOR_MIN;
if (m3 < MOTOR_MIN) m3 = MOTOR_MIN;
if (m4 < MOTOR_MIN) m4 = MOTOR_MIN;

if(m1 > MOTOR_MAX) m1 = MOTOR_MAX; //Velocidad maxima 2000.
if(m2 > MOTOR_MAX) m2 = MOTOR_MAX;
if(m3 > MOTOR_MAX) m3 = MOTOR_MAX;
if(m4 > MOTOR_MAX) m4 = MOTOR_MAX;
```

Figura 57. Expresión pulsos ESC y condiciones.

"Fuente: propia"

Como puede apreciarse en el tramo representado en la figura, después de otorgar a cada regulador el pulso que entregar al motor se muestran unas condiciones en caso que los valores sean demasiado bajos o demasiado altos. En otras palabras, para que no sature, ya que nuestros ESC está calibrado con unos límites de pulso. De esta forma, ocurre la activación de los distintos puertos que se encargan de proporcionar una entrada a los motores según el pulso de entrada.

Una vez se han establecido las cifras pertenecientes a los pulsos, se convoca a la función y esta reescribe los microsegundos del control de rotación. Este es el tramo final de un ciclo del programa. Se ha hecho uso de la función incluida en la librería Servo.h. Esta envía los pulsos en microsegundos al regulador de velocidad, de esta forma no es necesaria una transformación a escala para la función que se encarga de escribir el ángulo en los pines de salida. Ya que la librería Servo.h, es utilizada en el control de giro de servomotores, que tiene como rango de 0 a 180 grados, que se traducen en mínimo y máximo pulso en un ESC. La citada librería, funciona mediante interrupciones, es decir que genera pulsos a 50Hz de frecuencia. Esto se traduce en 20ms entre pulsos HIGH, lo que significa que proporciona una salida de modulación por ancho de pulsos en cualquiera de los pines digitales de la tarjeta; solución perfecta para aplicar a la perfección el ancho de pulso en la corrección de la velocidad de rotación de los motores.

Representar una gráfica con pulsos sería repetir el tópico, en la figura 33 se muestran de forma visual el concepto de PWM.

5.3.6 Android Studio

El último punto que atañe al código escrito se encuentra la programación de la app Android usada como mando radiocontrol con el fin de definir qué actividad efectuará nuestro vehículo aéreo no tripulado. Esta, se puede instalar en cualquier dispositivo de sistema operativo Android con pantalla táctil. Se han explorado dos opciones y decidido por una para desarrollar esta aplicación para *smartphones* y *tablets*, AndroidStudio y App Inventor. Dos *software* completamente gratuitos y de libre acceso, pero con distinto modo de programación. Primero, App Inventor posee una interfaz gráfica, esto es debido a su programación por bloques y la paleta de donde añadir elementos al conjunto sin necesidad de declarar librerías. Tan solo se programa mediante bloques.

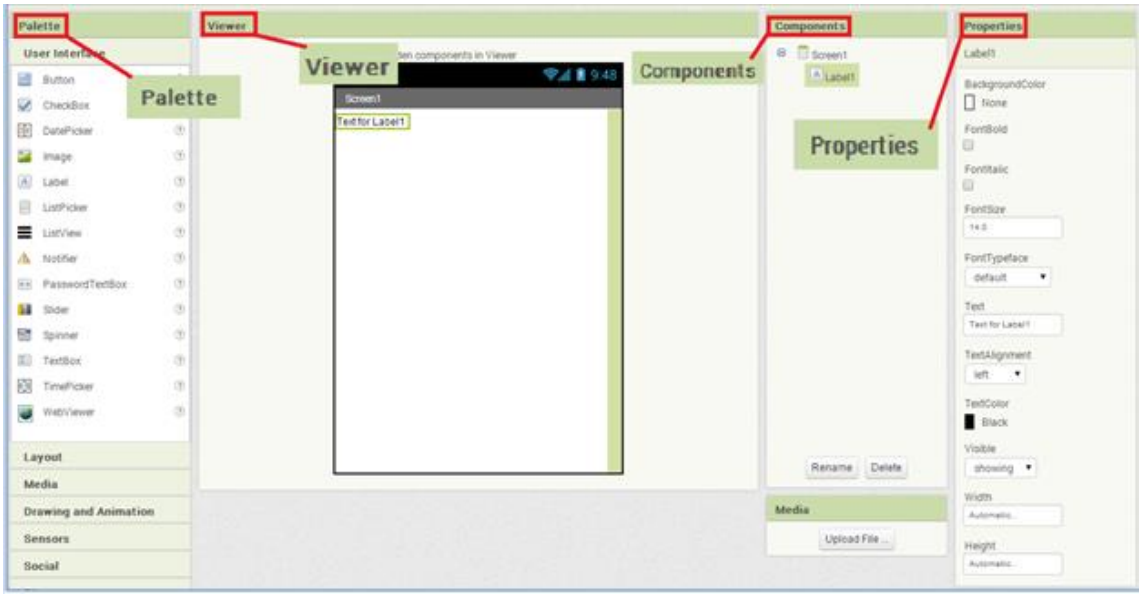


Figura 58. Interfaz App Inventor.

"Fuente: propia"

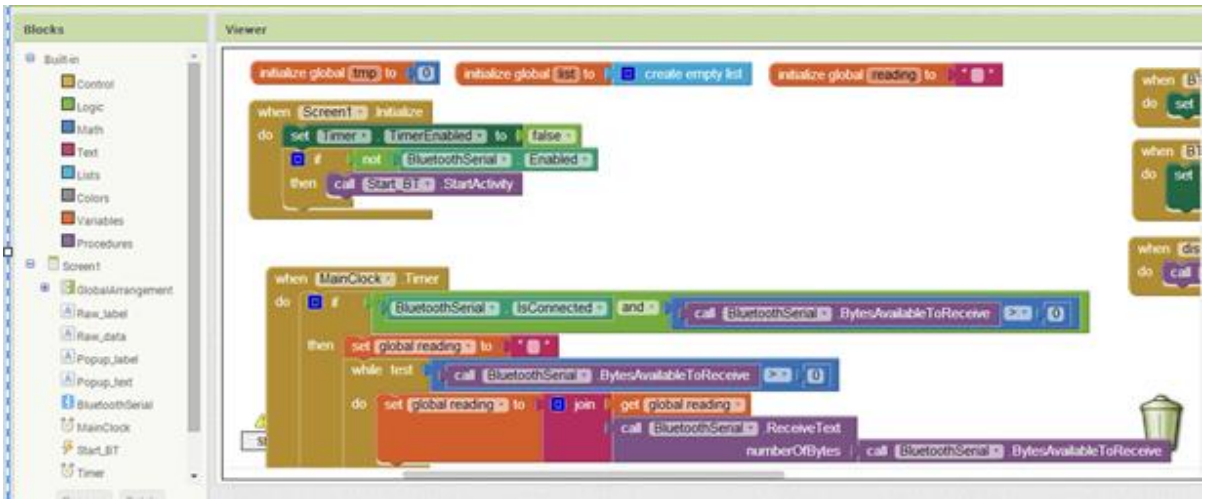


Figura 59. Programación por bloques App Inventor.

"Fuente: propia"

Mientras que, con el *software* de Android Studio se programa en lenguaje Java. Nuestros conocimientos de este lenguaje de programación son limitados, pero en red se puede encontrar abundante información para la tarea tan asequible que realiza esta aplicación.

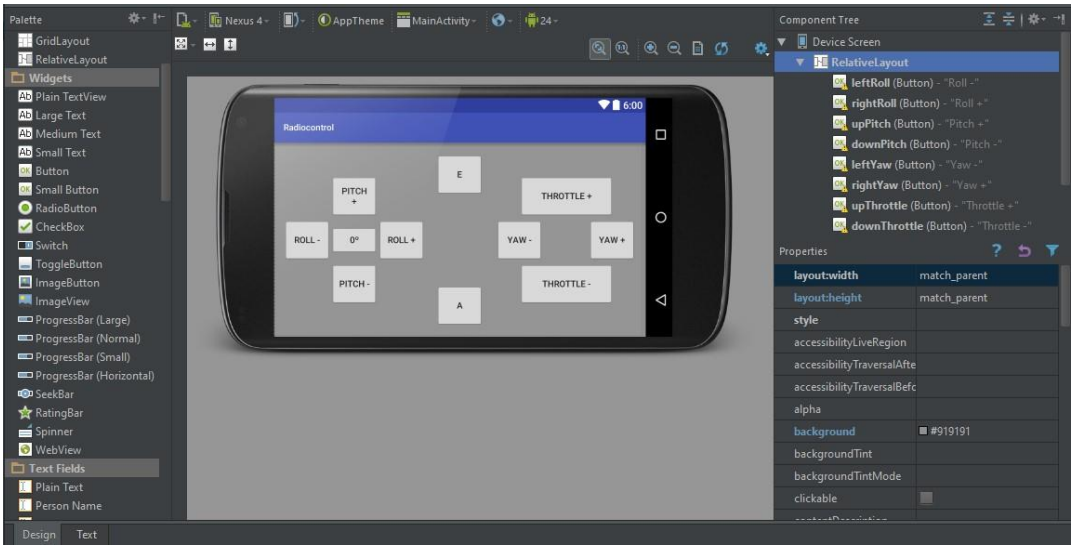


Figura 60. Interfaz de programación Android Studio.

"Fuente: propia"

Al final nos decidimos por este último. Con App Inventor descubrimos limitaciones y dificultades a la hora de colocar los botones del mando RC, por eso arriesgamos con la programación en Java, con el objeto de tener control directo sobre la distribución del diseño gráfico. La tarea a cumplir, no es ni más ni menos que el envío de caracteres al módulo *bluetooth*, los cuales son interpretados por el *sketch* Arduino.

```
//Creacion de la activity principal del programa
public class MainActivity extends AppCompatActivity {

    private static final String TAG = "bluetooth2";

    Button leftYaw, rightYaw, upThrottle, downThrottle, leftRoll, rightRoll, upPitch, downPitch;
    TextView txtArduino;
    Handler h;

    final int RECIEVE_MESSAGE = 1;      // Status  for Handler
    //Variable donde se almacena el adaptador Bluetooth del movil
    private BluetoothAdapter btAdapter = null;
    //Variable donde se almacena el adaptador Bluetooth remoto(Arduino)-HC05
    private BluetoothSocket btSocket = null;

    private StringBuilder sb = new StringBuilder();

    private ConnectedThread mConnectedThread;

    // SPP UUID service: ID de la conexion entre los dos dispositivos
    private static final UUID MY_UUID = UUID.fromString("00001101-0000-1000-8000-00805F9B34FB");

    // MAC del dispositivo remoto(HC05)
    private static String address = "98:D3:31:F4:11:71";
```

Figura 61. Declaración de variables Android Studio

Aquí se crea la actividad principal del programa. La cual, declara las variables de los distintos botones y del almacenamiento de información de los distintos módulos *bluetooth* que intervienen; como son el *Smartphone* y el módulo HC-05. Además, necesita incluir la dirección MAC del módulo HC-05 para un correcto funcionamiento, es decir, para cada dispositivo habría que modificar la dirección expuesta en el *sketch*.

```
@Override
//Funcion que se ejecuta al abrirse la aplicacion
//Todas las aplicaciones Android lo tienen.
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);

    leftYaw = (Button) findViewById(R.id.left_yaw);
    rightYaw = (Button) findViewById(R.id.right_yaw);
    upThrottle = (Button) findViewById(R.id.up_throttle);
    downThrottle = (Button) findViewById(R.id.down_throttle);

    leftRoll = (Button) findViewById(R.id.left_roll);
    rightRoll = (Button) findViewById(R.id.right_roll);
    upPitch = (Button) findViewById(R.id.forward_pitch);
    downPitch = (Button) findViewById(R.id.backwards_pitch);
```

Figura 62. Asignación botones Android Studio

En la imagen mostrada se aprecia la declaración de los distintos pulsadores denominándolos por la acción que debe efectuar el dron.

```
//Guarda el dispositivo Bluetooth del movil en esta variable.
btAdapter = BluetoothAdapter.getDefaultAdapter();       // get Bluetooth adapter
checkBTState();

//Cada una de estas funciones envía un caracter.
leftYaw.setOnClickListener((v) → {
        mConnectedThread.write("i");
        Toast.makeText(getBaseContext(), "Left Yaw", Toast.LENGTH_SHORT).show();
});

rightYaw.setOnClickListener((v) → {
        mConnectedThread.write("h");
        Toast.makeText(getBaseContext(), "Right Yaw", Toast.LENGTH_SHORT).show();
});

upThrottle.setOnClickListener((v) → {
        mConnectedThread.write("d");
        Toast.makeText(getBaseContext(), "Up Throttle", Toast.LENGTH_SHORT).show();
});

downThrottle.setOnClickListener((v) → {
        mConnectedThread.write("");
        Toast.makeText(getBaseContext(), "Down Throttle", Toast.LENGTH_SHORT).show();
});

leftRoll.setOnClickListener((v) → {
        mConnectedThread.write("g");
        Toast.makeText(getBaseContext(), "Left Roll", Toast.LENGTH_SHORT).show();
});
```

Figura 63. Guardar bluetooth del dispositivo local y funciones de envío de caracteres Android Studio

En lo alto del código se guarda la información del adaptador *bluetooth* integrado en el *Smartphone*. Y debajo, están las funciones llamadas cuando pulsamos en el botón adjudicado a cada uno de los distintos caracteres, cuya función es la de enviar un carácter al ser llamada. Aunque el dispositivo HC-05 pueda vincularse al *Smartphone* sin necesidad de programa alguno, la app sí necesita sincronizarse con el dispositivo móvil.

Se ha programado de manera que, al abrir la aplicación, esta averigua si la conexión *bluetooth* está encendida. En caso negativo, pedirá al usuario activar la comunicación *bluetooth* en pantalla. Como se muestra en la siguiente captura de pantalla desde nuestro *Smartphone*.

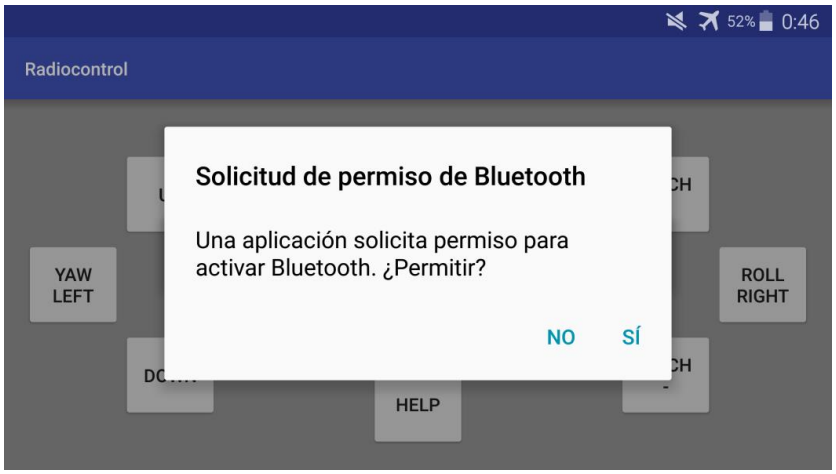


Figura 64. Petición para encender bluetooth App Android.

"Fuente: propia"

```
//Recibe el canal/dispositivo input y el output
private class ConnectedThread extends Thread {
    private final InputStream mmInStream;
    private final OutputStream mmOutStream;

    public ConnectedThread(BluetoothSocket socket) {
        InputStream tmpIn = null;
        OutputStream tmpOut = null;

        // Get the input and output streams, using temp objects because
        // member streams are final
        try {
            tmpIn = socket.getInputStream();
            tmpOut = socket.getOutputStream();
        } catch (IOException e) { }

        mmInStream = tmpIn;
        mmOutStream = tmpOut;
    }
```

Figura 65. Proceso Thread Android Studio

El proceso *Thread* se familiariza con el proceso en bucle de la IDE Arduino. Se trata de un bucle con variables y procesos propios, en el cual se pueden añadir otros. Una vez la app está conectada al *bluetooth* del dispositivo externo, espera a que se produzca una interacción en los botones de la interfaz para leer el carácter que es enviado al *handler*, donde se encuentra la función que enviará el dato al módulo externo HC-05.

Tal y como se describe en el comentario del código, la figura de abajo corresponde al diseño de cada botón que forma la interfaz gráfica de la aplicación Android. Su posición en pantalla es colocada manualmente sin necesidad de programación alguna, sin embargo, las medidas sí están delimitadas en estas líneas. Además, la velocidad de envío de datos fue comprobada mediante el monitor serial de la IDE Arduino, resultando en un rápido envío desde que se pulsa hasta aparecer en pantalla; de retraso despreciable al ojo humano.

```
<!--Cada etiqueta es un boton de la app y en esta parte del codigo es donde
la parte de diseño grafico se lleva a cabo-->

<Button
    android:layout_width="75dp"
    android:layout_height="70dp"
    android:id="@+id/left_yaw"
    android:text="Yaw left"
    android:layout_centerVertical="true"
    android:layout_alignParentLeft="true" />

<Button
    android:layout_width="75dp"
    android:layout_height="70dp"
    android:id="@+id/right_yaw"
    android:text="Yaw right"
    android:layout_below="@+id/up_throttle"
    android:layout_toRightOf="@+id/down_throttle"
    android:layout_toEndOf="@+id/down_throttle"/>

<Button
    android:layout_width="75dp"
    android:layout_height="70dp"
    android:id="@+id/up_throttle"
    android:text="Up"
    android:layout_above="@+id/left_yaw"
    android:layout_toRightOf="@+id/left_yaw"
    android:layout_toEndOf="@+id/left_yaw" />
```

Figura 66. Diseño XML botones Android Studio

Para finalizar, mostramos la sencilla interfaz de nuestra app instalada e iniciada. En última instancia se ha añadido un botón en el centro de la cruceta izquierda, para el dron volver al punto de equilibrio sin complicaciones.

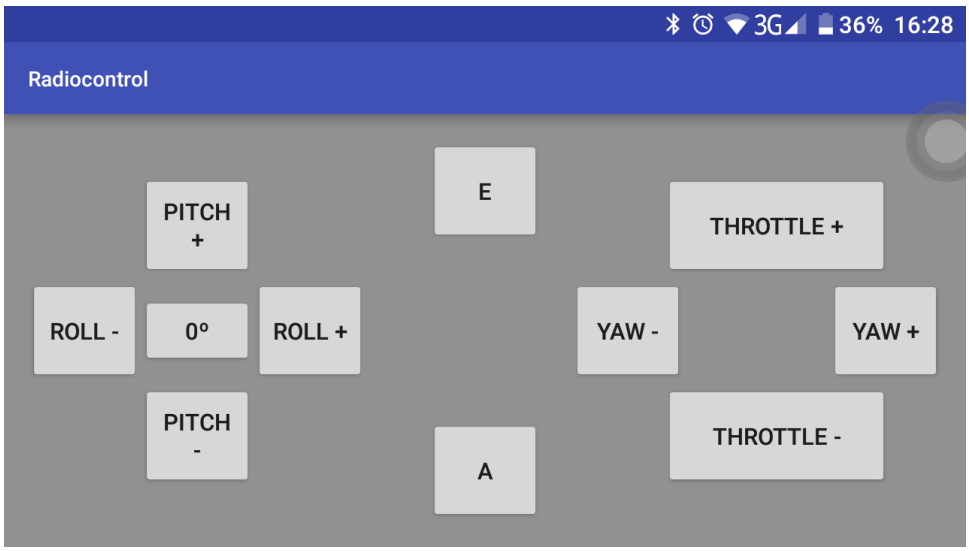


Figura 67. Interfaz App Radiocontrol Android Studio.

"Fuente: propia"

6. Presupuesto

A continuación, se muestran las tablas correspondientes al coste de cada ítem que consideramos importante para la implementación de este proyecto.

Tabla 15. Presupuesto componentes principales

Componentes principales		
nº	**Descripción**	**Coste (€)**
1	Arduino Nano	2,64
1	Sensor MPU-6050	2,02
1	Módulo bluetooth HC-05	2,67
6	Electronic Speed Controller (ESC)	89,4
4	Motores brushless	54,4
2	Hélices CW	2,5
2	Hélices CCW	2,5
1	Batería Lipo	28,9
1	Chasis	47,97
1	PDB(Power Distributor Board)	1,65
1	Placa de prototipo soldable	1,34
Total	235,99	

Tabla 16. Herramientas y componentes de montaje

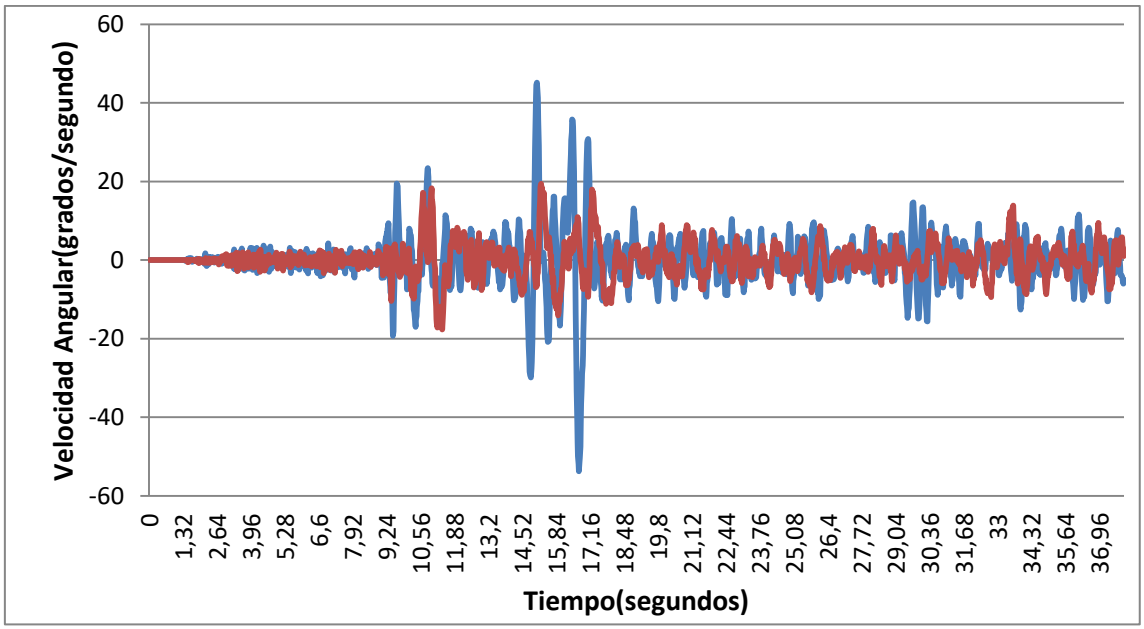
Herramientas		
nº	**Descripción**	**Coste (€)**
1	Tester	12,53
X	Tuercas y contratuercas M3	2
1	Soldador y estaño	15,9
X	Cableado	5
1	Portátil con sistema operativo Windows	0
1	Dispositivo Android	0
12	Bullet connectors (parejas)	12
1	Pack material termoretráctil	2,59
24	Pines RGB macho-macho	0,6
Total	50,62	

7. Resultados

Con el propósito de mostrar los datos cosechados en los vuelos efectuados para la adquisición de datos, se han creado unas gráficas temporales que incluyen las variables que hemos considerado importantes en el control automático de un *quadcopter*. A continuación, mostramos esta información.

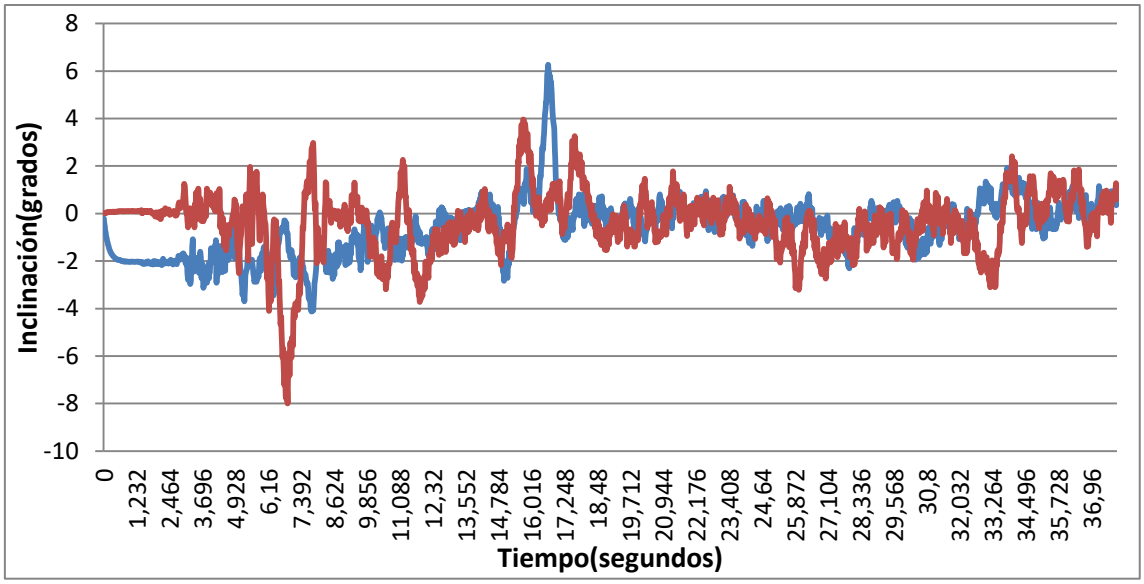
Modo Estable:

Tabla 17. Gráficas datos modo Estable íntegro. Giroscopio



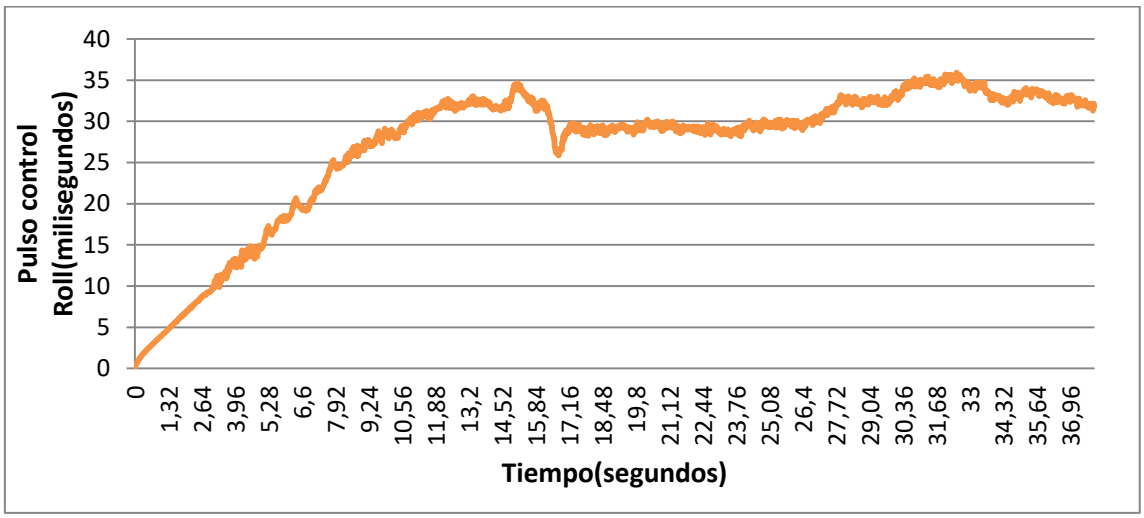
En la representación de los datos provenientes del giroscopio, podemos ver el comportamiento en ambos ejes del dron, X(azul) e Y(rojo), en el despegue del vehículo aéreo no tripulado y la cantidad de movimiento oscilatorio captado por el sensor. Los picos más altos representan el momento en que hemos forzado el dron hacia un lado u otro con el fin de demostrar el correcto funcionamiento en conjunto de los controladores asignados a cada eje.

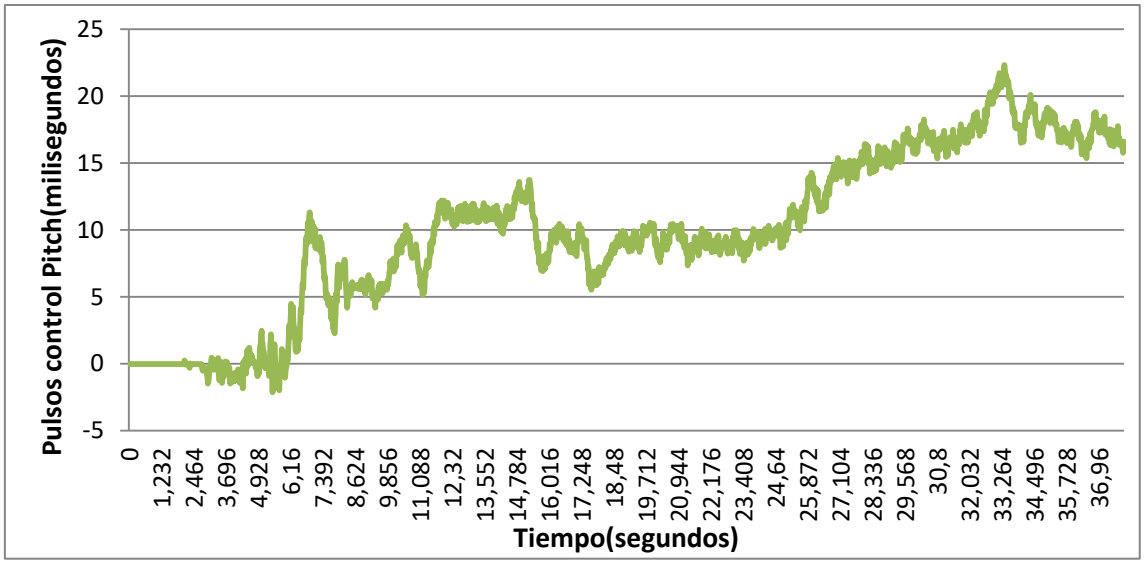
Tabla 18. Gráficas datos modo Estable íntegro. Filtro Complementario



Lo mismo pasa en los datos provenientes del cómputo del filtro Complementario encargado de refinar los valores del acelerómetro y giroscopio, con el fin de ofrecer unas medidas más exactas y menos propensas a errores causados por el ruido. Claramente, se aprecia la proximidad al punto deseado, cero, en la inclinación de cada eje. Y, tal como hemos establecido antes, los picos destacables pertenecen al momento en que forzamos manualmente el robot para poner a prueba su algoritmo de control.

Tabla 19. Gráficas datos modo Estable íntegro. Acciones de control Roll y Pitch

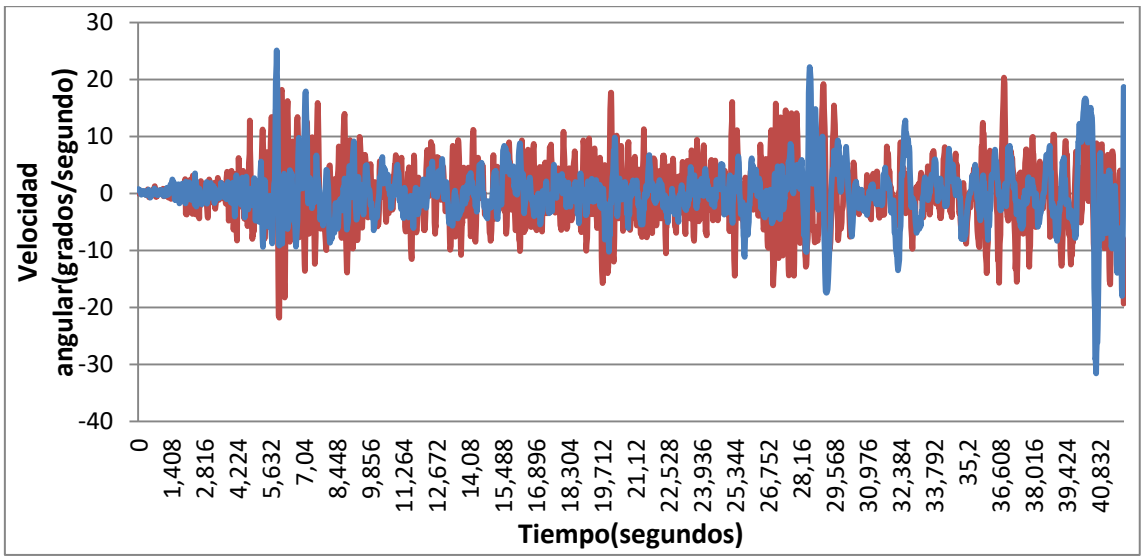




En cambio, en estas gráficas hemos añadido la salida de los PID encargados del control de estabilidad del modo Estable en loes ejes X e Y, o sea, Roll y Pitch respectivamente. Con estas salidas, se calculan los pulsos que serán, finalmente, enviados desde los pines Arduino a los reguladores de velocidad para el control de rotación de los motores.

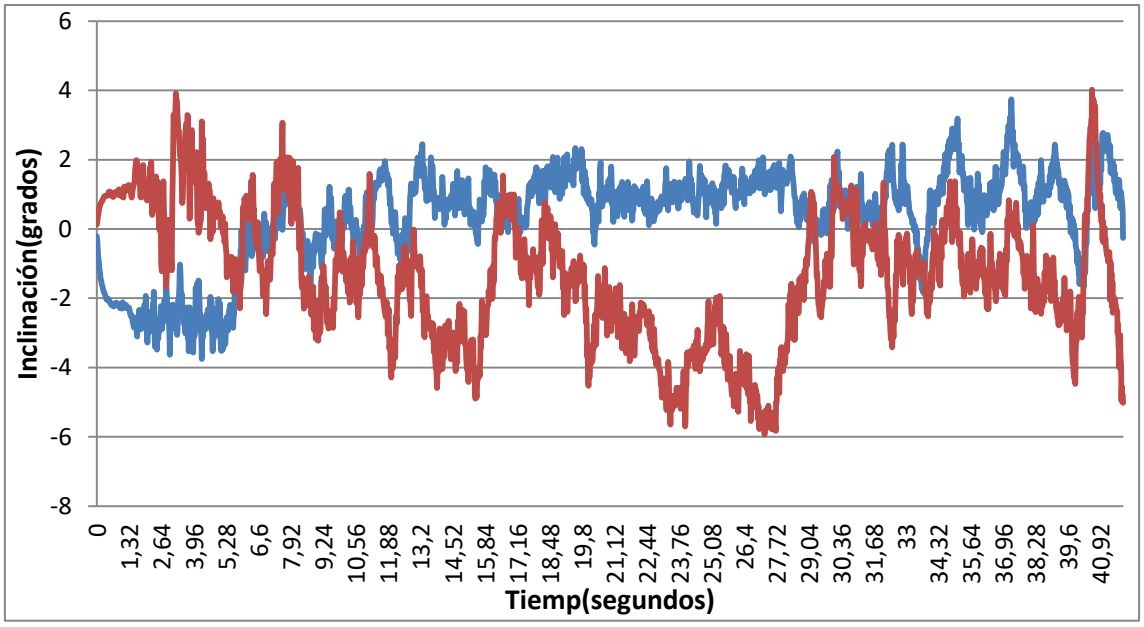
Modo Manual:

Tabla 20. Gráficas datos modo Manual íntegro. Giroscopio



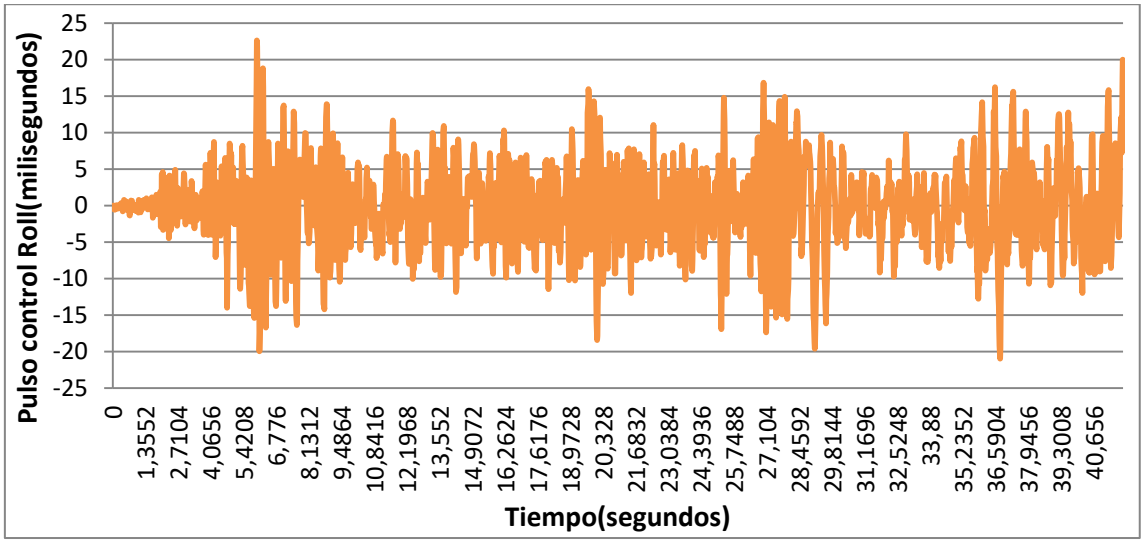
De la misma forma, hemos representado las gráficas correspondientes al modo Manual. E igual que con las anteriores, contienen picos adyacentes a los movimientos forzados para probar el correcto funcionamiento del control automático de estabilidad implementado.

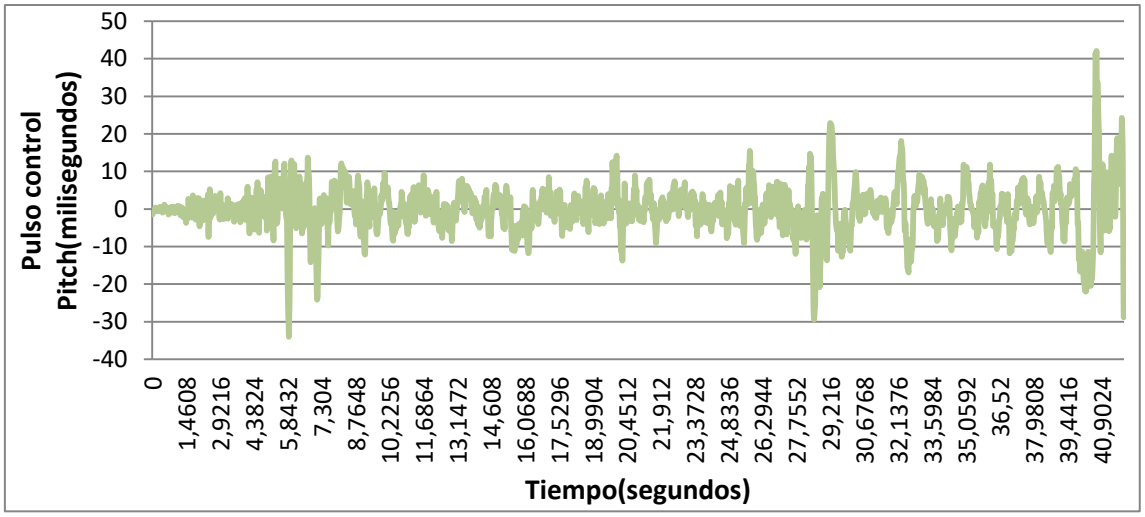
Tabla 21.Gráficas datos modo Manual íntegro. Filtro Complementario



De igual manera, incluimos la inclinación sufrida en este estilo de vuelo para mostrar una vez más la estabilidad de este *quadcopter*. En este caso, el eje Pitch sufrió más cambios, ya que inclinamos de forma deliberada cada extremo sucesivamente en distintas ocasiones. Aunque el eje Roll no haya padecido muchos cambios de posición en esta gráfica, tan solo con ver los datos proporcionados por el giroscopio, vemos un claro movimiento oscilatorio debido a los cambios bruscos de velocidad obligados por nosotros de manera externa.

Tabla 22.Gráficas datos modo Manual íntegro. Acciones de control Roll y Pitch





Aquí podemos dar más detalle de la labor del algoritmo de control destinado al eje X(naranja), mostrando una actividad bastante continua y eficaz, como se aprecia en las gráficas anteriores. Al mismo tiempo, la participación del PID en el eje Y(verde) es persistente y eficiente interviniendo en la corrección del error en la velocidad angular del sistema. Antes de finalizar, es necesario aclarar por qué no hemos incluido gráficas referentes al eje Z ,o Yaw. Esto es, porque solo proporciona información de la velocidad angular a la que gira el sistema, sin ningún tipo de relevancia en la estabilidad horizontal y vertical.

8. Conclusiones

Con el propósito de sintetizar la serie de éxitos cosechados en el transcurso de este trabajo, procedemos a numerar y esclarecer cada uno de ellos.

En primer lugar, cabe señalar la parte positiva. Se han alcanzado los objetivos señalados al principio del documento. Como se ha demostrado en el apartado anterior, el algoritmo de control diseñado para cada estilo de vuelo funciona según lo esperado. Este paso, ha sido con diferencia el más costoso en todo el proceso, a causa de las variables y datos a procesar, de cuya salida depende la conducta íntegra del cuadricóptero. Así pues, la implementación de una app Android ha resultado satisfactoria, vinculándose sin problema al módulo *bluetooth* y cumpliendo tanto en diseño como en funcionalidad. Igualmente, la experiencia obtenida con la programación de nuestro propio código desde cero y funcional, resulta en sí mismo un triunfo indiscutible. Además, su estructura permite modificarlo de manera que puedan ser añadidos otros dispositivos como una cámara, una emisora de radio para comunicación RC, sensores infrarrojos, y un largo etc. Es decir, los pines no usados del microcontrolador Arduino Nano pueden ser utilizados para añadir otras funcionalidades; el código es compatible con otros modelos de microcontrolador de la marca Arduino, como el Arduino Uno.

En cuanto a la parte del hardware, del cual está compuesto nuestro *quadcopter*, nos ha brindado con la experiencia en la búsqueda de componentes compatibles entre ellos con el fin de ejecutar un trabajo optimizado y sin problemas técnicos imprevistos que puedan retrasar el avance del proyecto. Todo esto, alimenta nuestra experiencia en los diversos campos que ocupa cada cuestión. Con lo cual, tan solo podemos agradecer y abrazar cada dificultad que nos encuentra en el camino hacia la culminación de nuestros objetivos.

En último lugar, cabe señalar la parte negativa en la consecución de este proyecto, empezando por la problemática del eje Z. Este, como explicado en el apartado del sensor MPU-6050, solo puede ser mesurado mediante el giroscopio, que nos ofrece la velocidad angular en la que se desplaza el sistema en este eje. La contrariedad que subrayamos es la espontaneidad en la corrección del error de movimiento. Cosa que ha generado un laborioso calibrado de coeficientes PID mediante el método de prueba y error. La solución a este problema es la de añadir un magnetómetro externo al sensor, de manera que nos pueda informar del desfase angular sufrido desde su encendido, proporcionando un mayor conocimiento y control sobre este eje.

Figura 68. Magnetómetro marca Compass de tres ejes HMC5883L.

"Fuente: www.cetronic.es"

Como último punto en los contras de nuestro trabajo, tratamos el asunto del control radiocontrol. Esta experiencia nos ha brindado conocimientos básicos sobre aeromodelismo, entre ellos destacamos la importancia de incluir una emisora RC con su respectivo mando. Esto es a causa de la necesidad de una rápida intervención de los controles sobre el dron. La clave para el manejo de este tipo de sistemas es la habilidad, la cual se obtiene con la práctica. Además de las interrupciones por *hardware* que permiten al sistema reaccionar rápidamente ante las órdenes del mando RC, esta interrupción no afecta al bucle del programa subido al microcontrolador. En nuestro caso, no precisamos de una interrupción por hardware, sino una adquisición de datos en cada repetición del bucle, de forma que cuando la app Android envía el carácter afín al comando a realizar, este queda en el buffer del módulo *bluetooth* hasta que el bucle llega a la función de adquisición de datos. Aquí es donde se genera un retraso en la reacción del sistema, y poniendo en riesgo el robot en mitad del vuelo.

Figura 69. Emisora Saturn 2,4 Ghz de 5 canales, FHSS y receptor 2,4 Ghz FHSS de 6 canales.

"Fuente: www.kitsmodelismo.es"

Al final del trabajo se han logrado los siguientes puntos:

- Realizar un algoritmo de control que permita al dron mantenerse estable automáticamente, y flexible a cualquier cambio en el código.
- Implementar una app Android mediante el software AndroidStudio para el manejo del cuadricóptero a distancia mediante comunicación *bluetooth*
- Desarrollar una base sólida, ligera, flexible y desmontable.

Partiendo de la labor realizada a lo largo de este Trabajo de Fin de Grado, surgen las siguientes líneas de trabajo futuras:

- Control de estabilidad mejorado.
- Sistema radiocontrol basado en emisora de radiofrecuencia.
- Añadido de otros componentes.
- Proyectos relacionados con el diseño e impresión 3D.

9. Referencias bibliográficas y bibliografía

9.1 Referencias bibliográficas

IDE Arduino (Septiembre de 2015)
https://www.arduino.cc/en/Main/ArduinoBoardNano
http://www.righto.com/2009/07/secrets-of-arduino-pwm.html

IMU MPU-6050 y comunicación I2C (Septiembre de 2015)
http://robologs.net/2014/10/15/tutorial-de-arduino-y-mpu-6050/
http://www.i2cdevlib.com/forums/
https://www.cdiweb.com/datasheets/invensense/MPU-6050_DataSheet_V3%204.pdf

Módulo bluetooth HC-05 (Septiembre de 2015)
http://www.prometec.net/bt-hc05/
http://www.robotshop.com/media/files/pdf/rb-ite-12-bluetooth_hc05.pdf

UBEC (Septiembre de 2015)
https://oscarliang.com/what-is-esc-ubec-bec-quadcopter/
http://es.aliexpress.com/item/Bluelans-BEC-UBEC-3A-5V-Brushless-Receiver-Servo Power-Supply-for-RC-Airplane-Aircraft/

Control automático (Diciembre de 2016)
http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-direction/

Aeromodelismo(Enero de 2016)
http://aeroquad.com/showwiki.php?title=Flight-Modes#Rate-Mode
https://oscarliang.com/understanding-pid-for-quadcopter-rc-flight/

ESCs (Abril de 2016)
http://robots.dacloughb.com/project-2/esc-calibration-programming/
http://rcarduino.blogspot.com.es/2012/01/can-i-control-more-than-x-servos-with.html
https://rc-innovations.es/RCI-Spider-ZTW-30A-OPTO-Small-multicoptero

Android Studio/Java (Mayo de 2016)
http://stackoverflow.com/
http://www.codeproject.com

Diseño e impresión 3D (Mayo de 2016)
http://www.thingiverse.com/
https://zortrax.com/

9.2 Bibliografía

LAJARA VIZCAÍNO, José Rafael; PELEGRÍ SEBASTIÁ, **José.** *Sistemas integrados con Arduino*. Barcelona, Ed. Marcombo, Primera edición, 2014.

Anexo

Sketch_quadcopter_3.ino

```
#include "Constantes.h"
#include <Math.h>
#include <Servo.h>
//Añadimos la libreria MPU6050 para la obtencion y connversion de datos
#include "MPU6050.h"
//Libreria de comunicacion serie entre pines
#include <SoftwareSerial.h>
//Libreria de comunicacion entre el Arduino y el sensor
#include "I2Cdev.h"
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
  #include "Wire.h"
#endif

//Creamos la instancia de la libreria MPU6050 para obtencion y adjudicacion de datos
MPU6050 mpu;
//Modo de vuelo
int modo_vuelo = -1;

//Variables PID Roll, Pitch y Yaw

//Setpoints PIDs
float AccelP_Setpoint, AccelR_Setpoint;
float GyroP_Setpoint, GyroR_Setpoint;
float YawSetpoint;
double last_accelSetpointP, last_accelSetpointR, last_yawSetpoint;
double last_gyroSetpointP, last_gyroSetpointR;

//Input Anterior (sustituto del error derivativo
//para correción de derivative kick)
double accelP_antInput;
double accelR_antInput;
double gyroP_antInput;
double gyroR_antInput;
double yaw_antInput;

//Error Integral
float AccelP_iErr, AccelR_iErr;
float GyroP_iErr, GyroR_iErr;
float iErr_Yaw;

//Variables Ecuación y pulsos ESC --> Throttle-Yaw-Pitch-Roll

//Accion de control PID cascada
float roll, pitch, yaw;
int last_throttle = 1000;
//Aceleración del dron
int throttle = 1000;
```

```
//Pulso que se enviará a cada ESC
int m1, m2, m3, m4;

//Decalarión e inicialización parámetros PID
float P_Paccel = pitchPID_KP;
float P_Raccel = rollPID_KP;
float P_Pgyro = pitchCASC_PID_KP;
float P_Rgyro = rollCASC_PID_KP;
float P_yaw = YAW_PID_KP;
float I_Paccel = pitchPID_KI;
float I_Raccel = rollPID_KI;
float I_Pgyro = pitchCASC_PID_KI;
float I_Rgyro = rollCASC_PID_KI;
float I_yaw = YAW_PID_KI;
float D_Paccel = pitchPID_KD;
float D_Raccel = rollPID_KD;
float D_Pgyro = pitchCASC_PID_KD;
float D_Rgyro = rollCASC_PID_KD;
float D_yaw = YAW_PID_KD;

//Variables para la gestion y transformacion de datos del MPU6050: Offsets, valores filtrados y
datos obtenidos en tiempo real

float rollGyro, pitchGyro, yawGyro;
float accPitch, accRoll, accYaw;
float ciclo = CICLO;
float tiempoCiclo = ciclo/1000000;
int16_t ax, ay, az, gx, gy, gz;
double gyro_pitch_off, gyro_roll_off, gyro_yaw_off;
double acc_pitch_off, acc_roll_off, acc_yaw_off;

//Caracter recepción datos Bluetooth
unsigned char comando;

//Variable tiempo debug
unsigned long prev_time = 0;


//Comunicacion serie con el bluetooth
SoftwareSerial BT(RxD,TxD);  //10 RX, 11 TX

void setup() {

 //inicializar el I2C bus ( la librería I2Cdev no lo hace automáticamente)
   #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
      Wire.begin();
      TWBR = 24; // 400kHz I2C clock (200kHz if CPU is 8MHz)
   #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
      Fastwire::setup(400, true);
   #endif

   //Iniciar lectura Bluetooth y puertos
```

```
  pinMode(RST, OUTPUT);
  pinMode(EN, OUTPUT);
  //Estado incial
  digitalWrite(RST, LOW);
  //Modo comunicacion, en HIGH sería modo Configuración
  digitalWrite(EN, LOW);
  //Enciendido Módulo pin 8
  digitalWrite(RST, HIGH);

  //Configuramos el puerto serie por software para comunicar con el HC-05

  BT.begin(38400);
  Serial.flush();
  delay(500);
  //Configuramos puerto serie para debug
  Serial.begin(38400);

 //Inicializamos la comunicacion i2C y la configuracion del sensor MPU6050

 Serial.println("Inicializando dispositivos I2C");
 mpu.initialize();

 Serial.println("Verificando conexion MPU6050...");
 Serial.println(mpu.testConnection() ? "MPU6050 OK" : "MPU6050 ERROR!!!!!!!!!!!!!");

 Serial.println("Configurando MPU6050...");
 //Configura la salida a la máxima velocidad angular que puede leer, +-2000 deg/sec equivalen
a 16bits que son valores entre +-32768.
 mpu.setFullScaleGyroRange(MPU6050_GYRO_FS_2000);
 //Configura el rango de escala del acelerómetro a +-16g. Que es el máximo que se puede.
 mpu.setFullScaleAccelRange(MPU6050_ACCEL_FS_4);
 //Configuramos el DLPF, filtro paso-bajo digital que incorpora el dispositivo MPU-6050 para
 //reducir vibraciones causadas por los motores
 mpu.setDLPFMode(MPU6050_DLPF_BW_10);  //10,20,42,98,188
 //Sample rate de filtración de ruido
 mpu.setRate(4);   // 0=1khz 1=500hz, 2=333hz, 3=250hz 4=200hz

 //Añadimos los offset que nos ha proporcionado el programa de calibrado de Offset
 Serial.println("Obteniendo offsets MPU6050...");

 mpu.setXAccelOffset(-3253);
 mpu.setYAccelOffset(1689);
 mpu.setZAccelOffset(783);
 mpu.setXGyroOffset(63);
 mpu.setYGyroOffset(11);
 mpu.setZGyroOffset(8);

 Serial.println("Offsets calculados");

  //Inicializamos puerto serie para debug
  #ifdef DEBUG_DATOS
```

```
  while(!Serial);
  Serial.println("Adquisicion de variables ON");
 #endif

 //Llamada a las funciones de inicialización de motores y armado

 inic_motores();
 delay(1000);
 armar_motores();
 delay(1000);
 inicializar_VariablesErrorPID();
 delay(500);
 modo_RC();

}

void loop(){

 //Llamamos a la función MPU_6050
 leerMPU();

  //Setpoints a cero, si no hay comando, estos se mantienen. Y si hay comando, el bucle while
los cambia
  //en relacion al comando recibido

  AccelP_Setpoint = last_accelSetpointP;
  AccelR_Setpoint = last_accelSetpointR;
  YawSetpoint = last_yawSetpoint;
  throttle = last_throttle;

  if(modo_vuelo == ACROBATICO){
    GyroP_Setpoint = last_gyroSetpointP;
    GyroR_Setpoint = last_gyroSetpointR;
  }

  //Esperamos a recibir datos
 if (BT.available()){

  comando = BT.read();


  //A continuación las funciones de desplazamiento

  if(comando == 'd'){  //Aumentar altitud

    throttle += 25;
    throttle = constrain(throttle, THROTTLE_MIN, THROTTLE_MAX);
    last_throttle = throttle;
    last_accelSetpointP = 0.0;
    last_accelSetpointR = 0.0;
    last_gyroSetpointP = 0.0;
    last_gyroSetpointR = 0.0;
```

```
  last_yawSetpoint = 0.0;
 }
 if(comando == 'p'){ //Disminuir altitud

  throttle -= 50;
  throttle = constrain(throttle, THROTTLE_MIN, THROTTLE_MAX);
  last_throttle = throttle;
  last_accelSetpointP = 0.0;
  last_accelSetpointR = 0.0;
  last_gyroSetpointP = 0.0;
  last_gyroSetpointR = 0.0;
  last_yawSetpoint = 0.0;
 }

 if(comando == 'b'){ //Pitch -

  if(modo_vuelo == ESTABLE){
  AccelP_Setpoint = -20.0;
  last_accelSetpointP = -20.0;
  last_accelSetpointR = 0.0;
  last_yawSetpoint = 0.0;
  }
  else if(modo_vuelo == ACROBATICO){
  GyroP_Setpoint = -40.0;
  last_gyroSetpointP = -40.0;
  last_gyroSetpointR = 0.0;
  last_yawSetpoint = 0.0;
 }
    }

 if(comando == 'e'){  //Pitch +

  if(modo_vuelo == ESTABLE){
  AccelP_Setpoint = 20.0;
  last_accelSetpointP = 20.0;
  last_accelSetpointR = 0.0;
  last_yawSetpoint = 0.0;
  }
  else if(modo_vuelo == ACROBATICO){
  GyroP_Setpoint = 40.0;
  last_gyroSetpointP = 40.0;
  last_gyroSetpointR = 0.0;
  last_yawSetpoint = 0.0;
  }
 }



 if(comando == 'f'){  //Roll -

  if(modo_vuelo == ESTABLE){
  AccelR_Setpoint = -20.0;
```

```
    last_accelSetpointR = -20.0;
    last_accelSetpointP = 0.0;
    last_yawSetpoint = 0.0;
    }
    else if(modo_vuelo == ACROBATICO){
    GyroR_Setpoint = -40.0;
    last_gyroSetpointR = -40.0;
    last_gyroSetpointP = 0.0;
    last_yawSetpoint = 0.0;
    }
       }

  if(comando == 'g'){  //Roll +

    if(modo_vuelo == ESTABLE){
    AccelR_Setpoint = 20.0;
    last_accelSetpointR = 20.0;
    last_accelSetpointP = 0.0;
    last_yawSetpoint = 0.0;
    }
    else if(modo_vuelo == ACROBATICO){
    GyroR_Setpoint = 40.0;
    last_gyroSetpointR = 40.0;
    last_gyroSetpointP = 0.0;
    last_yawSetpoint = 0.0;
    }
       }

  if(comando == 'h'){  //Yaw +

    YawSetpoint = 50.0;
    last_yawSetpoint = 50.0;
    last_accelSetpointP = 0.0;
    last_accelSetpointR = 0.0;
    last_gyroSetpointR = 0.0;
    last_gyroSetpointP = 0.0;

  }

  if(comando == 'i'){  //Yaw -

    YawSetpoint = -50.0;
    last_yawSetpoint = -50.0;
    last_accelSetpointP = 0.0;
    last_accelSetpointR = 0.0;
    last_gyroSetpointR = 0.0;
    last_gyroSetpointP = 0.0;

  }

}
```

```
    BT.flush();

    actualizar_control();

  //Llama a debug
  #ifdef DEBUG_DATOS
   proceso_debug();
 #endif
 prev_time = micros();


}
```

Constantes.h

```
//------MPU-6050------//

//HC-05 setup
#define RxD 12
#define TxD 11
#define RST 8
#define EN 13

//Ratios de conversión segun documentacion
#define A_R 16384.0
#define G_R 131.0

//Ratios de escala
#define GYRO_ESCALA 16.4

//Definimos tiempo de ciclo
#define CICLO 5000

//------PID------//

#define pitchPID_KP  1.0
#define pitchPID_KI  0.02
#define pitchPID_KD  0.2
#define PID_MIN  -300.0
#define PID_MAX  300.0

#define rollPID_KP  0.68
#define rollPID_KI  0.025
#define rollPID_KD  0.4
#define PID_MIN  -300.0
#define PID_MAX  300.0

#define pitchCASC_PID_KP  1.3
#define pitchCASC_PID_KI  0.001
#define pitchCASC_PID_KD  0.5
#define CASC_PID_MIN  -300.0
#define CASC_PID_MAX  300.0

#define rollCASC_PID_KP  1.0
#define rollCASC_PID_KI  0.001
#define rollCASC_PID_KD  0.2
#define CASC_PID_MIN  -300.0
#define CASC_PID_MAX  300.0

#define YAW_PID_KP  0.5
#define YAW_PID_KI  0.008
#define YAW_PID_KD  0.0
#define YAW_PID_MIN  -250.0
#define YAW_PID_MAX  250.0
```

```
//-------MOTOR PINES------//

#define PIN_MOTOR1  7
#define PIN_MOTOR2  6
#define PIN_MOTOR3  5
#define PIN_MOTOR4  4

//------Rangos pulsos ESC------//
#define MOTOR_MIN  1000
#define MOTOR_MEDIO  1500
#define MOTOR_MAX  2000

//------Configuración RC------//

#define THROTTLE_MIN 1000
#define THROTTLE_MAX 2000
#define THROTTLE_HALF 1500

//-------Debug Config---------//
#define DEBUG_DATOS
#define DEBUG_ANGULOS
#define DEBUG_PID
#define DEBUG_MOTORES
#define DEBUG_TIEMPO_CICLO

//------Modo de Control RC-------//
#define ACROBATICO 1
#define ESTABLE 2
```

Control_Motores.ino

```
//----------Actualización valores de pulso adjudicados a cada ESC----------//

void actualizar_control(){

 //Modo ESTABLE y ACROBATICO

   if(modo_vuelo == ESTABLE){
   //Llamamos las funciones del primer PID
    //GyroP_Setpoint = PID_AccelP(accPitch, AccelP_Setpoint, P_Paccel, I_Paccel, D_Paccel);
    //GyroR_Setpoint = PID_AccelR(accRoll, AccelR_Setpoint, P_Raccel,I_Raccel, D_Raccel);

    pitch = PID_AccelP(accPitch, AccelP_Setpoint, P_Paccel, I_Paccel, D_Paccel);
    roll = PID_AccelR(accRoll, AccelR_Setpoint, P_Raccel,I_Raccel, D_Raccel);


   //Seguido llamamos a las funciones PID del control en cascada
    //roll = PID_GyroR(rollGyro, GyroR_Setpoint, P_Rgyro, I_Rgyro, D_Rgyro);
    //pitch = PID_GyroP(pitchGyro, GyroP_Setpoint, P_Pgyro, I_Pgyro, D_Pgyro);
    //Llamamos al control PID del eje Yaw
    yaw = PID_Yaw(yawGyro, YawSetpoint, P_yaw, I_yaw, D_yaw);
   }
   if(modo_vuelo == ACROBATICO){

    //este modo necesita una adaptación de los setpoint del modo estable al
    // modo acro, es decir, valores de velocidad angular como setpoint
    roll = PID_GyroR(rollGyro, GyroR_Setpoint, P_Rgyro, I_Rgyro, D_Rgyro);
    pitch = PID_GyroP(pitchGyro, GyroP_Setpoint, P_Pgyro, I_Pgyro, D_Pgyro);
    //Llamamos al control PID del eje Yaw
    yaw = PID_Yaw(yawGyro, YawSetpoint, P_yaw, I_yaw, D_yaw);
   }
    /*Aunque los valores de roll, pitch y yaw sean de tipo float, al sumar una variable int con
otra float,  automáticamente redondea los valores float y el resultado es una cifra de tipo int*/

   m1 = throttle + pitch + roll + yaw;
   m2 = throttle + pitch - roll - yaw;
   m3 = throttle - pitch - roll + yaw;
   m4 = throttle - pitch + roll - yaw;

   if (m1 < MOTOR_MIN) m1 = MOTOR_MIN;  //1000 es la velocidad mínima
   if (m2 < MOTOR_MIN) m2 = MOTOR_MIN;
   if (m3 < MOTOR_MIN) m3 = MOTOR_MIN;
   if (m4 < MOTOR_MIN) m4 = MOTOR_MIN;

   if(m1 > MOTOR_MAX) m1 = MOTOR_MAX; //Velocidad maxima 2000
   if(m2 > MOTOR_MAX) m2 = MOTOR_MAX;
   if(m3 > MOTOR_MAX) m3 = MOTOR_MAX;
   if(m4 > MOTOR_MAX) m4 = MOTOR_MAX;

   actualizar_motores(m1, m2, m3, m4);
}
```

Control_PID.ino

```
//--------------------------------Control PID--------------------------------//

float PID_AccelP(float angulo, float setpoint, float Kp, float Ki, float Kd){

   float AccelP_error = setpoint - angulo;
   AccelP_iErr += Ki * AccelP_error;
   AccelP_iErr = constrain(AccelP_iErr, PID_MIN, PID_MAX);
   double derInput = angulo - accelP_antInput;
   double out = (Kp*AccelP_error - 0.04) + AccelP_iErr - Kd*derInput;
   out = constrain(out, PID_MIN, PID_MAX);
   accelP_antInput = angulo;
   out = out;
   return out;
 }

float PID_AccelR(float angulo, float setpoint, float Kp, float Ki, float Kd){

   float AccelR_error = setpoint - angulo;
   AccelR_iErr += Ki * AccelR_error;
   AccelR_iErr = constrain(AccelR_iErr, PID_MIN, PID_MAX);
   double derInput = angulo - accelR_antInput;
   double out = (Kp*AccelR_error + 0.02) + AccelR_iErr - Kd*derInput;
   out = constrain(out, PID_MIN,PID_MAX);
   accelR_antInput = angulo;
   out = out;
   return out;
 }

float PID_GyroP(float gyro, float RC, float Kp, float Ki, float Kd){

   float GyroP_error = RC - gyro;
   GyroP_iErr += Ki * GyroP_error;
   GyroP_iErr = constrain(GyroP_iErr, CASC_PID_MIN, CASC_PID_MAX);
   double derInput = gyro - gyroP_antInput;
   double out = Kp*GyroP_error + GyroP_iErr - Kd*derInput;
   out = constrain(out, CASC_PID_MIN, CASC_PID_MAX);
   gyroP_antInput = gyro;
   out = out;
   return out;
 }
```

```
float PID_GyroR(float gyro, float RC, float Kp, float Ki, float Kd){

   float GyroR_error = RC - gyro;
      GyroR_iErr += Ki * GyroR_error;
   GyroR_iErr = constrain(GyroR_iErr, CASC_PID_MIN, CASC_PID_MAX);
   double derInput = gyro - gyroR_antInput;
   double out = Kp*GyroR_error + GyroR_iErr - Kd*derInput;
   out = constrain(out, CASC_PID_MIN, CASC_PID_MAX);
   gyroR_antInput = gyro;
   out = out;
   return out;
  }
float PID_Yaw(float gyro, float RC, float Kp, float Ki, float Kd){

   float Yaw_error = RC - gyro;
    iErr_Yaw += Ki * Yaw_error;
   iErr_Yaw = constrain(iErr_Yaw, YAW_PID_MIN, YAW_PID_MAX);
   double derInput = gyro - yaw_antInput;
   double out = (Kp * Yaw_error + 0.15) + iErr_Yaw - Kd*derInput;
   out = constrain(out, YAW_PID_MIN, YAW_PID_MAX);
   yaw_antInput = gyro;
   return out;
  }
```

Debug.ino

```
void proceso_debug(){
#ifdef DEBUG_DATOS

#ifdef DEBUG_ANGULOS
  Serial.print(F("Ángulo X:"));
  Serial.print((float)(accRoll));
  Serial.print('\t');
  Serial.print(F("Ángulo Y:"));
  Serial.print((float)(accPitch));
  Serial.print('\t');
  Serial.print(F("Roll Gyro:"));
  Serial.print(rollGyro);
  Serial.print('\t');
  Serial.print(F("Pitch Gyro:"));
  Serial.print(pitchGyro);
  Serial.print('\t');
  Serial.print(F("Yaw Gyro:"));
  Serial.print(yawGyro);
  Serial.print('\t');

#endif

#ifdef DEBUG_PID
  Serial.print(F("PID_R:"));
  Serial.print(GyroR_Setpoint);Serial.print(',');
  Serial.print(roll);//Serial.print(',');
  Serial.print('\t');

  Serial.print(F("PID_P:"));
  Serial.print(GyroP_Setpoint);Serial.print(',');
  Serial.print(pitch);
  Serial.print('\t');

  Serial.print(F("PID_Y:"));
  Serial.print(YawSetpoint);Serial.print(',');
  Serial.print(yaw);
  Serial.print('\t');
#endif

#ifdef DEBUG_MOTORES

  Serial.print('\t');
  Serial.print(F("M1:"));
  Serial.print(m1);
  Serial.print('\t');
  Serial.print(F("M2:"));
  Serial.print(m2);
  Serial.print('\t');
  Serial.print(F("M3:"));
  Serial.print(m3);
```

```
  Serial.print('\t');
  Serial.print(F("M4:"));
  Serial.print(m4);
  Serial.print('\t');
#endif

#ifdef DEBUG_TIEMPO_CICLO
  Serial.print('\t');
  unsigned long elapsed_time = micros() - prev_time;
  Serial.print(F("Time:"));
  Serial.print((float)elapsed_time/1000);
  Serial.print(F("ms"));
  Serial.print('\t');
#endif

  Serial.println();
#endif
}
```

Modos_vuelo.ino

```
//--------------------Establecimiento de los modos de control del Quad--------------------------//

void modo_RC(){
 //No funciona, buscar otra condicion para la adjudicacion del modo de vuelo
 while(modo_vuelo <= 0){

    if (BT.available()){

       unsigned char caract_modo = BT.read();
       if(caract_modo == 'z') modo_vuelo = ESTABLE;
       else if(caract_modo == 'y') modo_vuelo = ACROBATICO;
       BT.flush();
    }
 }

 if(modo_vuelo == ESTABLE) Serial.println("Modo Estable");
 else if(modo_vuelo = ACROBATICO) Serial.println("Modo Acrobatico");
 delay(1000);
}
```

Motores.ino

```
//-------------Envío de valores de velocidad a cada ESC, armado, inicialización y actualización-------//

//Creamos obbjetos Servo para cada motor
Servo motor1;
Servo motor2;
Servo motor3;
Servo motor4;

void inic_motores(){
 //Adjudicar pines con cada ESC y motor
 motor1.attach(PIN_MOTOR1);
 motor2.attach(PIN_MOTOR2);
 motor3.attach(PIN_MOTOR3);
 motor4.attach(PIN_MOTOR4);

 //Establecer velocidad mínima para inicializar motores
 motor1.writeMicroseconds(MOTOR_MIN);
 motor2.writeMicroseconds(MOTOR_MIN);
 motor3.writeMicroseconds(MOTOR_MIN);
 motor4.writeMicroseconds(MOTOR_MIN);
}

void armar_motores(){
 //Establece velocidad mínima después de inicializar los motores
 motor1.writeMicroseconds(MOTOR_MIN);
 motor2.writeMicroseconds(MOTOR_MIN);
 motor3.writeMicroseconds(MOTOR_MIN);
 motor4.writeMicroseconds(MOTOR_MIN);
}

void actualizar_motores(int m1, int m2, int m3,int m4){
 //Modifica en la entrada los pulsos calculados despues de la acción de control computada
 motor1.writeMicroseconds(m1);
 motor2.writeMicroseconds(m2);
 motor3.writeMicroseconds(m3);
 motor4.writeMicroseconds(m4);
}
```

MPU_6050.ino

```
//-------------------Lectura y acondicionamiento lecturas MPU-6050--------------//
void leerMPU()
{
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);


   //De valores raw a grados
   float accYangle_raw = atan(((ax)/A_R)/sqrt(pow(((ay)/A_R),2) +
pow(((az)/A_R),2)))*RAD_TO_DEG;
   float accXangle_raw = atan(((ay)/A_R)/sqrt(pow(((ax)/A_R),2) +
pow(((az)/A_R),2)))*RAD_TO_DEG;

   //Obtencion valores de inclinacion del Gyro para su uso en la funciones PID
   pitchGyro =  ((gx)/GYRO_ESCALA);
   rollGyro = ((-1) * (gy)/GYRO_ESCALA);
   yawGyro = ((-1) * (gz)/GYRO_ESCALA);

   accPitch = (0.1 * accXangle_raw) + (0.9*(accPitch + (pitchGyro*tiempoCiclo)));
   accRoll = (0.1 * accYangle_raw)  + (0.9*(accRoll  + (rollGyro*tiempoCiclo)));

}
```

Radiocontrol apk

```
package com.rc.radiocontrol;

import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.Intent;
import android.os.Build;
import android.os.Handler;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.lang.reflect.Method;
import java.util.UUID;

//Creacion de la activity principal del programa
public class MainActivity extends AppCompatActivity {

    private static final String TAG = "bluetooth2";

    Button leftYaw, rightYaw, upThrottle, downThrottle, leftRoll, rightRoll, upPitch,
downPitch, mBtn, zBtn, yBtn;
    TextView txtArduino;
    Handler h;

    final int RECIEVE_MESSAGE = 1;                  // Status  for Handler
    //Variable donde se almacena el adaptador Bluetooth del movil
    private BluetoothAdapter btAdapter = null;
    //Variable donde se almacena el adaptador Bluetooth remoto(Arduino)-HC05
    private BluetoothSocket btSocket = null;

    private StringBuilder sb = new StringBuilder();

    private ConnectedThread mConnectedThread;

    // SPP UUID service: ID de la conexion entre los dos dispositivos
    private static final UUID MY_UUID = UUID.fromString("00001101-0000-1000-
8000-00805F9B34FB");

    // MAC del dispositivo remoto(HC05)
    private static String address = "98:D3:31:F4:11:71";
```

```
    @Override
    //Funcion que se ejecuta al abrirse la aplicacion
    //Todas las aplicaciones Android lo tienen.
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        leftYaw = (Button) findViewById(R.id.leftYaw);
        rightYaw = (Button) findViewById(R.id.rightYaw);
        upThrottle = (Button) findViewById(R.id.upThrottle);
        downThrottle = (Button) findViewById(R.id.downThrottle);

        leftRoll = (Button) findViewById(R.id.leftRoll);
        rightRoll = (Button) findViewById(R.id.rightRoll);
        upPitch = (Button) findViewById(R.id.upPitch);
        downPitch = (Button) findViewById(R.id.downPitch);

        mBtn = (Button) findViewById(R.id.mBtn);
        zBtn = (Button) findViewById(R.id.zBtn);
        yBtn = (Button) findViewById(R.id.yBtn);

        h = new Handler() {
            public void handleMessage(android.os.Message msg) {
                switch (msg.what) {
                    case RECIEVE_MESSAGE:
                                                    // if receive massage
                        byte[] readBuf = (byte[]) msg.obj;
                        String strIncom = new String(readBuf, 0, msg.arg1);
                        // create string from bytes array
                        sb.append(strIncom);
                                    // append string
                        int endOfLineIndex = sb.indexOf("\r\n");
                        // determine the end-of-line
                        if (endOfLineIndex > 0) {
                                    // if end-of-line,
                            String sbprint = sb.substring(0, endOfLineIndex);
            // extract string
                            sb.delete(0, sb.length());
                                  // and clear
                            txtArduino.setText("Data from Arduino: " + sbprint);            // update
TextView
                        }
                        //Log.d(TAG, "...String:"+ sb.toString() +  "Byte:" + msg.arg1 + "...");
                        break;
                }
            };
        };
```

```
//Guarda el dispositivo Bluetooth del movil en esta variable.
      btAdapter = BluetoothAdapter.getDefaultAdapter();            // get Bluetooth
adapter
      checkBTState();

      //Cada una de estas funciones envía un carácter segú en botón pulsado
      leftYaw.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("i");
            Toast.makeText(getBaseContext(), "Left Yaw",
Toast.LENGTH_SHORT).show();
         }
      });

      rightYaw.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("p");
            Toast.makeText(getBaseContext(), "Right Yaw",
Toast.LENGTH_SHORT).show();
         }
      });

      upThrottle.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("d");
            Toast.makeText(getBaseContext(), "Up Throttle",
Toast.LENGTH_SHORT).show();
         }
      });

      downThrottle.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("h");
            Toast.makeText(getBaseContext(), "Down Throttle",
Toast.LENGTH_SHORT).show();
         }
      });

      leftRoll.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("g");
            Toast.makeText(getBaseContext(), "Left Roll",
Toast.LENGTH_SHORT).show();
         }
      });

      rightRoll.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("f");
```

```
            Toast.makeText(getBaseContext(), "Right Roll",
Toast.LENGTH_SHORT).show();
         }
      });

      upPitch.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("b");
            Toast.makeText(getBaseContext(), "Up Pitch",
Toast.LENGTH_SHORT).show();
         }
      });

      downPitch.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("e");
            Toast.makeText(getBaseContext(), "Down Pitch",
Toast.LENGTH_SHORT).show();
         }
      });

      mBtn.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("m");
            Toast.makeText(getBaseContext(), "0º", Toast.LENGTH_SHORT).show();
         }
      });

      zBtn.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("z");
            Toast.makeText(getBaseContext(), "E", Toast.LENGTH_SHORT).show();
         }
      });

      yBtn.setOnClickListener(new View.OnClickListener() {
         public void onClick(View v) {
            mConnectedThread.write("m");
            Toast.makeText(getBaseContext(), "A", Toast.LENGTH_SHORT).show();
         }
      });
   }

   //Crea la conexión entre los dos dispositivos Bluetooth
   private BluetoothSocket createBluetoothSocket(BluetoothDevice device) throws
IOException {
      if(Build.VERSION.SDK_INT >= 10){
         try {
```

```
            final Method  m =
device.getClass().getMethod("createInsecureRfcommSocketToServiceRecord", new
Class[] { UUID.class });
            return (BluetoothSocket) m.invoke(device, MY_UUID);
         } catch (Exception e) {
            Log.e(TAG, "Could not create Insecure RFComm Connection",e);
         }
      }
      return  device.createRfcommSocketToServiceRecord(MY_UUID);
   }

   @Override
   public void onResume() {
      super.onResume();

      Log.d(TAG, "...onResume - try connect...");

      // Set up a pointer to the remote node using it's address.
      BluetoothDevice device = btAdapter.getRemoteDevice(address);

      // Two things are needed to make a connection:
      //   A MAC address, which we got above.
      //   A Service ID or UUID.  In this case we are using the
      //     UUID for SPP.

      try {
         btSocket = createBluetoothSocket(device);
      } catch (IOException e) {
         errorExit("Fatal Error", "In onResume() and socket create failed: " +
e.getMessage() + ".");
      }

      // Discovery is resource intensive.  Make sure it isn't going on
      // when you attempt to connect and pass your message.
      btAdapter.cancelDiscovery();

      // Establece la conexión
      Log.d(TAG, "...Connecting...");
      try {
         btSocket.connect();
         Log.d(TAG, "....Connection ok...");
      } catch (IOException e) {
         try {
            btSocket.close();
         } catch (IOException e2) {
            errorExit("Fatal Error", "In onResume() and unable to close socket during
connection failure" + e2.getMessage() + ".");
         }
      }
```

```
        // Crea un stream de datos para comunicar
        Log.d(TAG, "...Create Socket...");

        mConnectedThread = new ConnectedThread(btSocket);
        mConnectedThread.start();
    }

    @Override
    public void onPause() {
        super.onPause();

        Log.d(TAG, "...In onPause()...");

        try     {
            btSocket.close();
        } catch (IOException e2) {
            errorExit("Fatal Error", "In onPause() and failed to close socket." +
e2.getMessage() + ".");
        }
    }

    //Comprueba que el adaptador Bluetooth está encendido
    private void checkBTState() {
        if(btAdapter==null) {
            errorExit("Fatal Error", "Bluetooth not support");
        } else {
            if (btAdapter.isEnabled()) {
                Log.d(TAG, "...Bluetooth ON...");
            } else {
                //Pide encendido bluetooth
                Intent enableBtIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
                startActivityForResult(enableBtIntent, 1);
            }
        }
    }

    //Cuando hay algún error sale de la aplicación
    private void errorExit(String title, String message){
        Toast.makeText(getBaseContext(), title + " - " + message,
Toast.LENGTH_LONG).show();
        finish();
    }

    //Recibe el canal/dispositivo input y el output
    private class ConnectedThread extends Thread {
        private final InputStream mmInStream;
        private final OutputStream mmOutStream;

        public ConnectedThread(BluetoothSocket socket) {
```

```
        InputStream tmpIn = null;
        OutputStream tmpOut = null;

        // Get the input and output streams, using temp objects because
        // member streams are final
        try {
            tmpIn = socket.getInputStream();
            tmpOut = socket.getOutputStream();
        } catch (IOException e) { }

        mmInStream = tmpIn;
        mmOutStream = tmpOut;
    }

    //Mientras este Thread exista ejecutara esto
    public void run() {
        byte[] buffer = new byte[256];  // buffer store for the stream
        int bytes; // bytes returned from read()

        // Keep listening to the InputStream until an exception occurs
        while (true) {
            try {
                // Read from the InputStream
                bytes = mmInStream.read(buffer);            // Get number of bytes and
message in "buffer"
                h.obtainMessage(RECIEVE_MESSAGE, bytes, -1,
buffer).sendToTarget();             // Send to message queue Handler
            } catch (IOException e) {
                break;
            }
        }
    }

    //Envía datos al dispositivo remoto
    public void write(String message) {
        Log.d(TAG, "...Data to send: " + message + "...");
        byte[] msgBuffer = message.getBytes();
        try {
            mmOutStream.write(msgBuffer);
        } catch (IOException e) {
            Log.d(TAG, "...Error data send: " + e.getMessage() + "...");
        }
    }
}
```