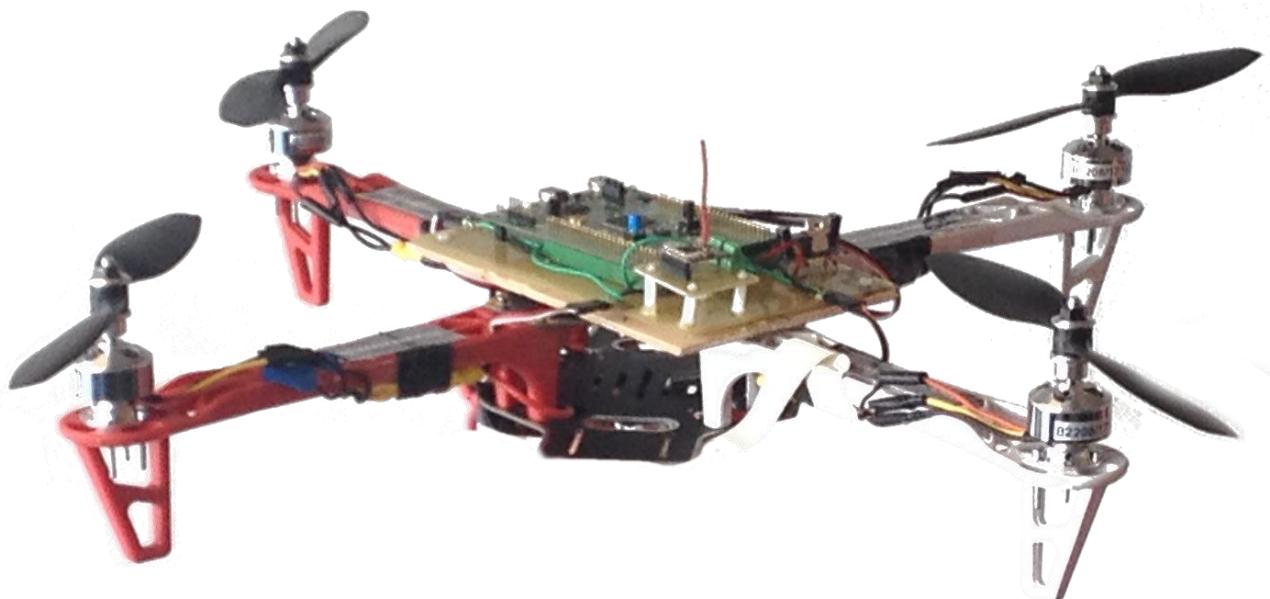


# Proyecto Tecnología Electrónica:

**ELECTROCOPTERO®**



*Alegre Usach, Roberto*

*Lluesma Rodríguez, Federico*

*Sellés Tomás, José Manuel*

*Grado en Ingeniería Aeroespacial*

# ÍNDICE

Agradecimientos.....	4
1 Introducción .....	5
2 Descripción del proyecto .....	6
2.1 Base teórica.....	6
2.2 Diagrama de Bloques .....	7
3 Desarrollo del Proyecto .....	9
3.1 Placa Discovery .....	9
3.1.1 Tabla de conexiones de la placa.....	9
3.1.2 Sistema de Referencia.....	10
3.1.3 Distribución LEDs de la placa.....	11
3.2 Comunicación inalámbrica.....	12
3.2.1 Consideraciones previas.....	13
3.2.2 Comandos empleados.....	13
3.2.3 Corte de Transmisión .....	19
3.2.4 USART .....	19
3.3 Comunicación por cable .....	20
3.3.1 Conexiones internas.....	20
3.3.2 Conexiones externas .....	29
3.4 IMU .....	32
3.4.1 Definición de Ángulos.....	33
3.4.2 Sistema de Referencia de Ángulos .....	34
3.4.3 Obtención de los ángulos del acelerómetro .....	35
3.4.4 Obtención Ángulos Giróscopo.....	36
3.4.5 Calibración de los sensores .....	37
3.4.6 Obtención de los ángulos en MatLab.....	42
3.5 Filtro Kalman.....	45
3.5.1 Diferencia con el Filtro Complementario .....	46
3.5.2 Funcionamiento del filtro.....	46
3.5.3 Consideraciones en la implementación del filtro.....	46
3.5.4 Parámetros del Filtro.....	47
3.5.5 Variables del Filtro.....	47
3.5.6 Ecuaciones del Filtro.....	48

3.5.7	Entradas y salidas del sistema.....	50
3.5.8	Calibración del filtro y estimación de constantes .....	51
3.6	Interrupciones.....	54
3.6.1	Interrupción TIMER 4 .....	54
3.6.2	Interrupción de la USART3 .....	59
3.7	Motores.....	60
3.7.1	Linealización de los motores .....	61
3.7.2	Control de los Motores .....	66
3.8	Dispositivo sonar.....	67
3.9	Sistema de control .....	68
3.9.1	Implementación .....	70
3.9.2	Ajuste constantes PID.....	75
3.9.3	Gráficas con ajuste PID.....	79
4	Detalles del Código Final .....	81
4.1	Tipos de Variables Usadas .....	81
4.2	CooCox .....	81
4.2.1	Llamadas a librerías.....	82
4.2.2	Definición de constantes.....	82
4.2.3	Definición de variables globales.....	83
4.2.4	Función de calibrar los sensores y los filtros.....	83
4.2.5	Función de calibrar el PID.....	84
4.2.6	Función de codificación de los ángulos .....	84
4.2.7	Programa principal .....	84
4.3	GUI de MatLab .....	87
4.3.1	Mensajes de la GUI de MatLab .....	88
4.3.2	Exportación de datos.....	91
5	Pruebas experimentales .....	93
5.1	Materiales empleados .....	93
5.2	Cuaderno diario .....	95
5.3	Recomendaciones.....	101
6	Conclusiones.....	104
7	Bibliografía.....	106

## Agradecimientos

---

Antes de comenzar la explicación de nuestro trabajo, no queremos dejar de dar las gracias a las personas sin las cuales este proyecto no habría salido adelante:

*A Miguel Alcañiz Fillol y Rafael Masot Peris, profesores de la asignatura Tecnología Electrónica, por permitirnos realizar este ambicioso proyecto y proporcionarnos toda la ayuda necesaria para el mismo sin importar el día ni la hora;*

*Al Departamento de Ingeniería Electrónica de la Escuela Técnica Superior de Ingeniería del Diseño y, en especial, a Pepe y María, técnicos del laboratorio de Tramuntana, por proporcionarnos todo el material necesario, desde un simple conector a un osciloscopio, y permitirnos sembrar el caos en su propio lugar de trabajo;*

*A Jaime Riera Guasp, profesor del Departamento de Física de la Escuela Técnica Superior de Ingeniería del Diseño, por prestarnos la balanza y el dinamómetro para los ensayos experimentales;*

*A Sergio García-Nieto Rodríguez y Xavier Blasco Ferragud, profesores del departamento de Ingeniería de Sistemas y Automática, por su ayuda a la hora de implementar y calibrar el controlador;*

*A nuestros compañeros de clase, por ayudarnos a realizar las pruebas experimentales en los días de más trabajo, sacrificando su propio físico y parte del tiempo para sus proyectos;*

*Y finalmente, a nuestras familias, por apoyarnos con el proyecto y darnos todas las facilidades posibles.*

# 1 Introducción

---

En la presente memoria se describe el proyecto que se ha llevado a cabo para la asignatura de Tecnología Electrónica, el cual consiste en crear un vehículo aéreo no tripulado (UAV), capaz de permanecer en posición estable en el aire ante cualquier perturbación de carácter débil que le pueda afectar.

Queremos recalcar desde el comienzo de este documento que el principal objetivo del proyecto es diseñar la electrónica completa de un vehículo de estas características. Obviamente, el control del mismo es un aspecto interesante y a la vez llamativo, aunque no se trata de un objetivo primordial, pues no es vital para el desarrollo de esta asignatura.

En las siguientes páginas trataremos de describir cómo hemos llevado a cabo el proyecto, intentando detallar al máximo todas las partes sobre las que hemos ido desarrollando, haciendo así que cualquier persona ajena al mismo y con ciertos conocimientos de aeronáutica y electrónica sea capaz de comprender el funcionamiento del mismo. Además, los códigos no se incluyen en su totalidad en esta memoria, puesto que se adjuntan en la misma carpeta que este documento. Nos referimos a los códigos *main.c*, realizado utilizando el software *CooCox*, y el *Control\_Electrocoptero.m*, realizado con *MatLab*, que nos ejecuta una interfaz gráfica de fácil manejo como veremos posteriormente.

## 2 Descripción del proyecto

### 2.1 Base teórica

*Electrocóptero* es un cuadricóptero que ha sido diseñado, como ya hemos mencionado anteriormente, con el fin de poder volar de forma controlada. Para llevarlo a cabo, comenzamos hace ya cuatro meses realizando un informe de viabilidad, en el que nos marcábamos ambiciosos objetivos y planteábamos el presupuesto y las especificaciones del proyecto.

En un primer momento se puede pensar que el control de un vehículo de este tipo puede conseguirse proporcionando a cada uno de los cuatro motores un empuje tal que lo equilibre en el aire. No obstante, esto se acerca poco a la realidad.

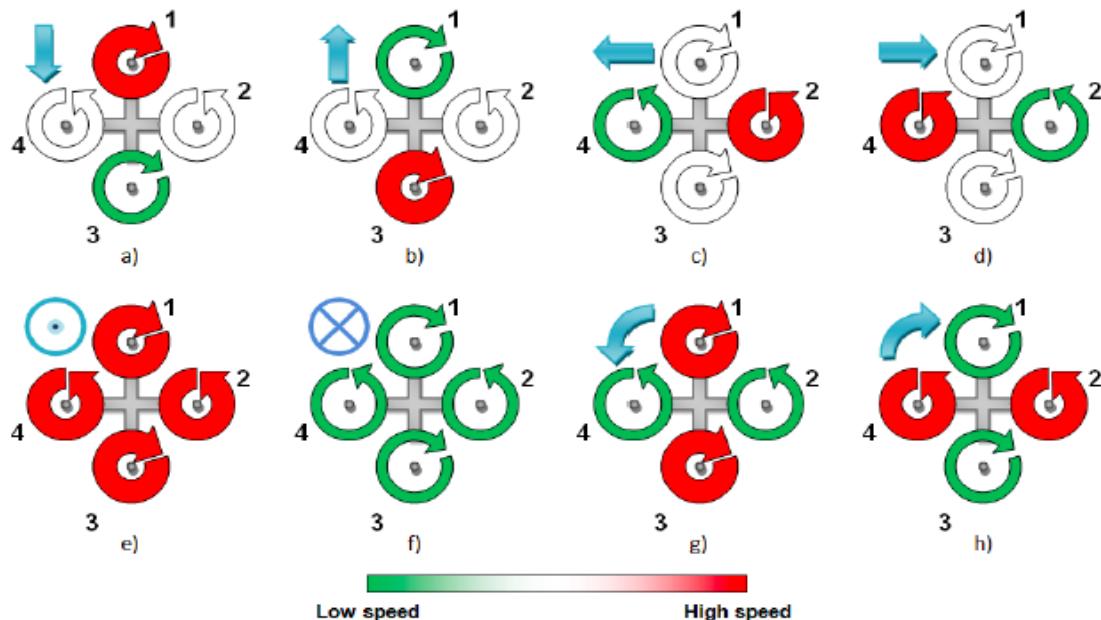
Controlar un cuadricóptero requiere conocer constantemente la inclinación del mismo, con el fin de modificar el empuje de cada motor y tratar de llegar al punto de equilibrio, ya que un sistema de este tipo es altamente inestable.

Para ello necesitamos un microcontrolador que recoja estos datos de inclinación de los sensores y controle el empuje de los motores. En nuestro proyecto hemos optado por la placa *STM32F3 Discovery*, que incorpora sensores como un acelerómetro, un magnetómetro y un giroscopio, que nos pueden ser útiles para conocer la actitud como veremos posteriormente sin tener que utilizar una Unidad de Medida Inercial (IMU) externa. A partir de estos datos aplicaremos un controlador de tipo PID que se encargue de dar el empuje correspondiente a cada motor para llegar al equilibrio.

No obstante, podemos estar en vuelo nivelado en torno al punto de equilibrio, pero, a su vez, estar girando en torno al eje vertical, pues los motores generan un fuerte par que lo hace rotar. Para evitar este fenómeno hacemos girar dos motores en un sentido y los otros dos motores en sentido opuesto (se requieren hélices distintas para cada sentido de giro) tratando de crear una estructura antipar. Éste es el fenómeno que en un helicóptero se corrige mediante el rotor de cola. Asimismo, utilizando el magnetómetro de la placa podemos llegar a saber si el cuadricóptero está rotando o no, pero tiene ciertos inconvenientes que detallaremos más adelante, por lo que no lo hemos utilizado.

Por otra parte, en cualquier dispositivo que se encuentre en sustentación en el aire, como es nuestro cuadricóptero, es imprescindible conocer la distancia que lo separa de la superficie, ya sea agua o tierra. Para ello, incorporamos un sonar que, leído correctamente, nos informa de la distancia a la que rebotan las ondas, tal y como se explica en el apartado correspondiente.

Si además de la estabilidad en el aire queremos controlar el movimiento del dispositivo en el espacio, debemos modificar el empuje de los motores tal y como muestra la siguiente imagen:



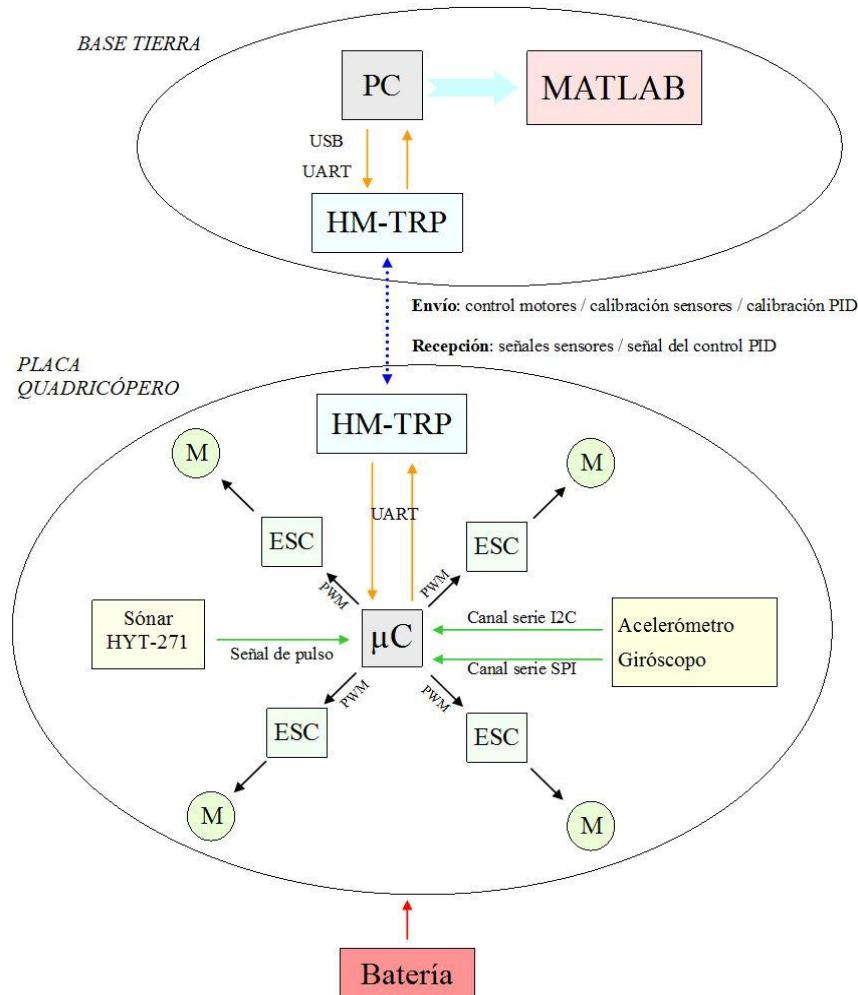
Las cuatro primeras situaciones corresponden a vuelo lateral. Se trata de reducir la sustentación del motor hacia dónde queremos ir y aumentar el régimen de giro del opuesto. Las dos siguientes hacen referencia al ascenso y descenso. Tal y como podemos pensar, se trata directamente de proporcionar más o menos potencia. Por último, se incluyen los casos en que buscamos la rotación horaria (h) o antihoraria (g), en los que buscamos modificar el par resultante.

También se ha de tener en cuenta el efecto de tener una superficie cercana a los motores. Si el chorro de aire que expulsado encuentra el suelo a pequeña distancia, el empuje generado es más grande: todo el flujo va directo a crear sustentación. No obstante, el mismo régimen de giro a una altura considerable genera mucha menos elevación, por lo que necesitaremos incrementar la potencia.

Para controlar el modo de funcionamiento, así como para hacer más ágil el ajuste de las constantes del controlador PID se diseñó una interfaz gráfica mediante *MatLab*, que comunicamos con el microcontrolador por medio de un módulo inalámbrico HM-TRP.

## 2.2 Diagrama de Bloques

Toda la interacción entre los diferentes dispositivos de nuestro proyecto que describiremos a continuación se encuentra detallada de forma esquemática en el siguiente diagrama de bloques:



Como podemos observar en el diagrama de bloques, el PC enviará diferentes tramas (STX) al microcontrolador por medio de los módulos inalámbricos. En función de qué trama se haya transmitido, el microcontrolador realizará una acción programada, entre las que se incluyen: modificación del pulso de los motores (manual o automático), modo de funcionamiento (ascenso, vuelo nivelado, descenso y modo *Hovercraft*), parada de emergencia, envío de datos de los sensores, pulso de los motores, constantes PID... Todos estos modos están explicados en el apartado de comunicación inalámbrica.

Por otra parte, el microcontrolador recibe las señales del acelerómetro (I2C), el giróscopo (SPI) y el sonar (modo captura) y modifica el empuje de los motores por medio de los *Electronic Speed Controllers* (ESC) que, en función del pulso recibido, modifican el régimen de giro del motor al cual van conectados.

## 3 Desarrollo del Proyecto

---

A continuación se muestra una serie de apartados que servirán para detallar todo lo referente al cuadricóptero y constituyen la parte importante de esta memoria.

### 3.1 Placa Discovery

---

Como bien decíamos antes, nuestro microcontrolador y cerebro del proyecto ha sido la placa *STM32-F3Discovery*, un micro de córtex ARM. Por experiencias previas, sabíamos que el controlador que se empleaba en esta asignatura, el *PIC18F4520*, es mucho menos potente en todos los aspectos, inclusive en cuanto a potencia de cálculo e instrucciones por segundo que puede realizar. Por dicho motivo creímos que las prestaciones del mismo se quedaban cortas y este año hemos decidido emplear la placa *STM32-F3Discovery*.

Una de las características comparativas que más nos ha llamado la atención entre ambos controladores es que una multiplicación en coma flotante con nuestra placa tarda del orden de microsegundos, mientras que la placa usada en años anteriores realiza esta operación en orden de magnitud de milisegundos.

Otra de las características es que la estructura con la que trabaja la *Discovery* es de 32 bits, mientras que el *PIC* lo hace a 16 bits. Por otra parte, la configuración de frecuencia del oscilador interno de la placa *STM32-F3Discovery* es de 72 MHz, mientras que en el caso del *PIC18F4520* se suele trabajar a una frecuencia de 4 MHz con oscilador interno, 18 veces menor velocidad.

Respecto a las memorias de datos, la de córtex ARM consta de una memoria RAM de 40 KBytes, mientras que el de la familia PIC tiene una capacidad de 1,5 KBytes, lo que nos muestra la mayor potencia de la placa empleada en nuestro proyecto.

#### 3.1.1 Tabla de conexiones de la placa

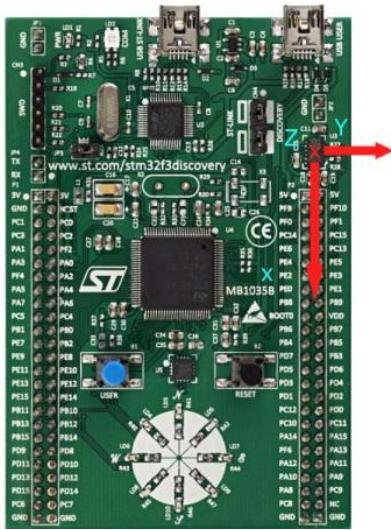
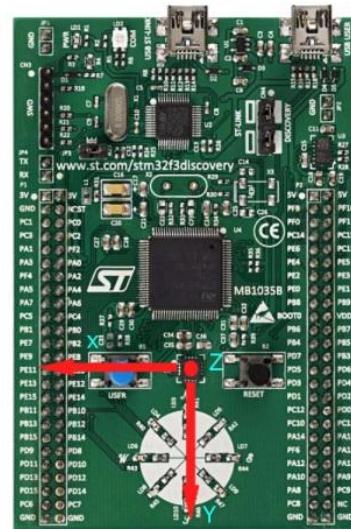
En la siguiente tabla se muestra la configuración que hemos dado a cada uno de los pines de la placa, con la función que desempeñan en el código. Es fundamental tener en cuenta el *Datasheet* de la placa puesto que no todos los pines cumplen con las especificaciones que requieren cada una de las funciones que debemos implementar. De este modo, la configuración final de los mismos es la siguiente:

PIN	Periférico	Función
<i>PB4</i>	TIMER 3	ESC 1
<i>PA4</i>	TIMER 3	ESC 2
<i>PC8</i>	TIMER 3	ESC 3
<i>PC9</i>	TIMER 3	ESC 4
<i>PD8</i>	USART 3	Transmisión inalámbrica
<i>PD9</i>	USART 3	Recepción inalámbrica
<i>PE3</i>	SPI	Línea de CS
<i>PA5</i>	SPI	Línea SCK
<i>PA6</i>	SPI	Línea MI
<i>PA7</i>	SPI	Línea MO
-	TIMER 4	Interrupción lectura datos
<i>PA0</i>	Botón B1	Calibración del giróscopo
<i>PB6</i>	I2C	SCL
<i>PB7</i>	I2C	SDA
<i>PA15</i>	TIMER 2	Sonar (captura pulso)
<i>PB9</i>	TIMER 17	Sonar (salida PWM)
-	LEDs	Ver <a href="#">Tabla de LEDs</a>

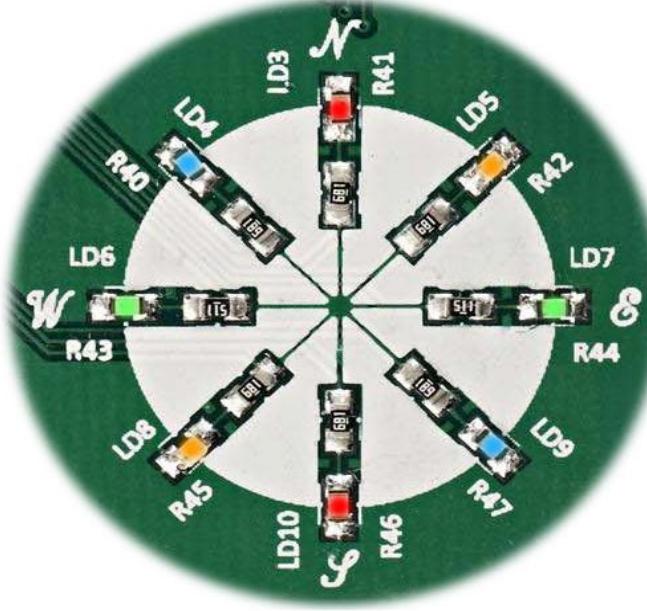
### 3.1.2 Sistema de Referencia

A continuación se muestran los sistemas de referencia utilizados a lo largo del proyecto y que son fundamentales sobre todo a la hora del tratamiento de las ecuaciones, así como del convenio de signos que podamos tomar para implementar dichas ecuaciones.

Los sistemas de referencia asociados a los sensores de lectura de aceleraciones o fuerzas específicas asociadas a la estructura fija del acelerómetro, y velocidades angulares de rotación del giróscopo son los siguientes:

**Sistema del acelerómetro****Sistema del giróscopo****3.1.3 Distribución LEDs de la placa**

Por otro lado, tenemos una burbuja de LEDs en la placa que nos sirve de mucha ayuda a la hora de señalizar cualquier movimiento de la aeronave no tripulada, bien cualquier transmisión y recepción de datos o también a la hora de calibrar la estructura respecto de una posición de referencia establecida por el usuario. A continuación se muestra la burbuja de LEDs con los colores que cada LED posee:



También podemos ver una tabla con la función de cada uno de los LEDs anteriores, que en ciertas ocasiones se saturan debido a las numerosas funciones que tienen asociadas, como es el caso del LED4:

LED	Posición	PIN	Color	Función
LD3	Norte	PE9	Rojo	Señal de pitch negativo
LD5	Noreste	PE10	Naranja	Indica ascenso
LD7	Este	PE11	Verde	Señal de roll negativo
LD9	Sureste	PE12	Azul	Fallo con transmisión Giróscopo
LD10	Sur	PE13	Rojo	Señal de pitch positivo
LD8 <sup>1</sup>	Suroeste	PE14	Naranja	Fallo con transmisión Acelerómetro
LD6	Oeste	PE15	Verde	Señal de roll positivo
LD4	Noroeste	PE8	Azul	Indica que el dispositivo se está calibrando Recopilando datos para la USART Modo control manual de motores Indica descenso Modo Hovercraft

En algunas ocasiones, tenemos combinación de varios LEDs como pueden ser las siguientes:

- Opción NSEO (LD3 + LD6 + LD7 + LD10): placa bloqueada por parada de emergencia o enviando datos de sensores por la USART.
- Opción NONE (LD4 + LD5) encendidos: indica pulso equilibrio.

Los LEDs verdes y rojos (NSEO) forman una combinación especial dado sus funciones. Ésta será detallada en el apartado de [LEDs "burbuja"](#).

### 3.2 Comunicación inalámbrica

Una de las bases de nuestro proyecto es la comunicación inalámbrica. Para ello empleamos el protocolo UART, tanto desde la placa como desde el PC. La transmisión UART es un protocolo serie, a partir del cual nos comunicamos desde ambos extremos a un módulo inalámbrico, el cual se encarga de modular la transmisión y enviar los bits que recibe mediante ondas de radio. El diagrama de esta conexión sería el siguiente:



<sup>1</sup> Este LED creemos que nunca funcionó, nunca se llegó a encender. Ver apartado 5.3.

Para nosotros, el proceso de comunicación inalámbrica es transparente. Nos hemos limitado a escribir y leer de una línea de transmisión o recepción siguiendo el protocolo UART, considerando que el módulo inalámbrico se encarga de gestionar por él mismo la transmisión.

### 3.2.1 Consideraciones previas

A pesar de haber dicho con anterioridad que no es relevante para nuestro proyecto la conversión de comunicación serie a radiofrecuencia, existen algunas consideraciones y limitaciones operativas que conlleva este sistema sin cables:

- La probabilidad de perder un dato es mucho mayor y por tanto para transmisiones largas hemos de incorporar un sistema de control de trama. Éstos consistirán en un byte inicial de inicio de trama, llamado STX, y un byte final de comprobación llamado CHECKSUM.
- El PC no es tan rápido para recibir datos como lo es de transmitir el micro, por tanto, es necesario incorporar retrasos en la transmisión del micro entre trama y trama.
- La transmisión es únicamente unidireccional. Si ambos dispositivos intentan transmitir a la vez, los datos colisionarán. Por ello hemos decidido trabajar en un sistema de jerarquía maestro-esclavo, para evitar que ambos dispositivos intenten transmitir simultáneamente.
- Siguiendo la conclusión del punto anterior trabajamos de la siguiente forma: *MatLab* actúa como maestro, de tal manera que cuando quiere algo de la placa le envía un comando asociado a la función que pretende. El micro, como esclavo, debe decir que ha entendido el comando y actuar en consecuencia, dejando *MatLab* a la espera de nuevos datos, o esperando él mismo la recepción de nuevas instrucciones.
- El alcance es limitado a algunos metros, así como la velocidad. Si intentamos transmitir a mayor distancia o a mayor velocidad de 9600 Hz, la pérdida de bytes será considerable.
- Trabajamos con transmisiones de datos byte a byte, es decir, cada paquete enviado o recibido debe ser un número de 8 bits, comprendido entre 0 y 255.
- Como trabajamos a 9600 bytes/s, cada byte tarda aproximadamente 1 ms en transmitirse. Ello imposibilita transmitir tramas demasiado largas si la placa está tomando medidas, pues la interrupción se da cada 10 ms. Además, el tiempo total de la transmisión no se aproxima al tiempo que tarda cada byte (1 ms) por el número de bytes, sino que hay que añadir retrasos intrínsecos a la gestión de interrupciones del programa y la transmisión inalámbrica.

### 3.2.2 Comandos empleados

La GUI de MatLab, que mostraremos más adelante, permite realizar 10 conexiones distintas con la placa, que hemos definido con el término "comando". Todas ellas empiezan de la misma forma: *MatLab* como maestro envía el STX asociado a ese

comando, que será 1 byte inconfundible. La placa recibirá este byte y, como se supone que se encuentra a la espera de instrucciones, salta la interrupción por recepción de la USART y ejecuta la instrucción determinada.

### Comando 0: inicio de la comunicación

Partimos de una situación en la que la comunicación se encuentra cerrada, por tanto ésta se debe iniciar.

- 1) *MatLab* abre el puerto y envía STX0.
- 2) La placa lo recibe y devuelve STX0.
- 3) Si llega correctamente de nuevo al PC, se deja el puerto abierto y en la GUI se activan todas las funciones. Si no, se cierra el puerto y se dejan sin activar los botones de los otros paneles.

Para cerrar la comunicación no es necesario volver a comunicar con la placa. Simplemente *MatLab* cierra el puerto.

### Comando 1: transmisión de datos de los sensores sin calibrar

- 1) *MatLab* envía una trama de 2 bytes:

STX1	N
------	---

N: número de datos a muestrear (tras escalar el rango de 10-2560 a 0-255)

- 2) La placa los recibe y cuando tiene los 2 bytes devuelve STX1.
- 3) *MatLab* vuelve a recibir STX1 y queda a la espera.
- 4) La placa guarda los N datos internamente, 1 por cada 10 ms que se ejecuta la interrupción, y enciende un LED azul para indicar que está guardando los siguientes datos (sin calibrar):
  - a. Aceleración en X del acelerómetro (AccX).
  - b. Aceleración en Y del acelerómetro (AccY).
  - c. Aceleración en Z del acelerómetro (AccZ).
  - d. Velocidad angular en X del giroscopio (GirX).
  - e. Velocidad angular en Y del giroscopio (GirY).
  - f. Velocidad angular en Z del giroscopio (GirZ).
- 5) Cuando ha terminado de tomar medidas, la placa realiza los siguientes pasos:
  - a. Envía un dato simbólico, 33, para iniciar la interrupción por transmisión.
  - b. Desactiva la interrupción para tomar medidas de los sensores.
  - c. Apaga el LED azul
  - d. Enciende los LEDs rojos y verdes.
  - e. Envía byte a byte cada una de las N tramas de 16 bytes, conformadas de la siguiente forma:

STX1	AccX	AccY	AccZ	GirX	GirY	GirZ	n	CHK
------	------	------	------	------	------	------	---	-----

n: nº de dato enviado (empezando por el 0), lo cual permite saber cuándo se ha tomado para representar gráficamente el tiempo en el eje horizontal.

- 6) *MatLab* recibe una a una las N tramas de 16 bytes. Sin embargo, para filtrar errores en la transmisión y no descoordinarse, realizará una lectura byte a byte. *MatLab* no empezará a construir internamente la trama hasta que no lea una

STX1, a partir del cual toma 15 bytes más. Cuando lea los 15, no volverá a construir otra trama hasta que lea otro STX1, por si se ha perdido algún byte por mitad.

- 7) Cuando la placa ha enviado todas las tramas, vuelve a activar la interrupción para tomar datos de los sensores y apaga los LEDs verdes y rojos.
- 8) Cuando *MatLab* ha terminado de descargar los datos, realiza una validación de los CHECKSUM y procesa los bytes llegados para ofrecer las aceleraciones y velocidades angulares en unidades con sentido físico. Llegados a este punto, el usuario puede representar gráficamente o guardar los datos.

### Comando 2: calibración de la placa

- 1) *MatLab* envía STX2.
- 2) La placa lo devuelve, enciende un LED azul y se calibra.
- 3) *MatLab* recibe el STX2 y entiende que la placa procede a calibrarse.

### Comando 3: control de los motores

- 1) *MatLab* envía STX3.
- 2) La placa lo devuelve y para los motores.
- 3) *MatLab* lo recibe de nuevo y activa en la GUI los comandos que permiten controlar los motores.
- 4) La placa se pone a la espera de recibir comandos de control de los motores.
- 5) Si la placa recibe otro STX3 corta el control de los motores por parte de *MatLab* y apaga el LED azul.

Durante el control (paso 3) *MatLab* envía tramas de 1 bytes, cuya interpretación por parte del microcontrolador es la siguiente:

M1	M0	+/-	D5	D4	D3	D2	D1	D0
----	----	-----	----	----	----	----	----	----

- M1M0: Motor a cambiar el pulso (0 a 3)
- +/-: Aumentar (1) o disminuir (0) la potencia
- D5...D0: Cantidad a aumentar o disminuir siguiendo la siguiente tabla

D5...D0	Cantidad	D5...D0	Cantidad
1	1 µs	7	25 µs
2	2 µs	8	30 µs
3	5 µs	9	50 µs
4	10 µs	10	100 µs
5	15 µs	11	150 µs
6	20 µs	12	200 µs

Existen unos bytes especiales que realizan otro tipo de acciones. Éstos no deben entrar en conflicto con los anteriores:

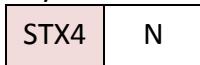
- xx000000: Parar el motor correspondiente: pulso de 910 µs.
- xx100000: Arrancar el motor correspondiente: pulso de 1000 µs.
- xx111111: Dar información sobre el pulso del motor correspondiente, sin modificarlo.
- 10011111: STX3, cortar control manual de los motores.
- 01011110: Parada de emergencia (comando 6).

En caso que se aumente o disminuya el pulso, o se mande la petición de información de pulso, la placa devolverá una trama de 1 byte con el valor del pulso con precisión de 1 µs partiendo de 1000 e integrando un sistema de escalas que *MatLab* deberá interpretar en base a datos anteriores:

1000µs: 000	1256 µs: 000	1512 µs: 000
...	...	...
1255µs: 255	1511 µs: 255	1767 µs: 255

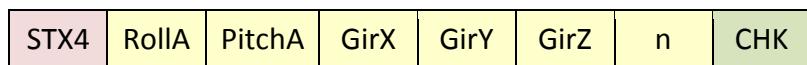
#### Comando 4: transmisión de datos de los sensores calibrados

- 1) *MatLab* envía una trama de 2 bytes:



N: número de datos a muestrear (tras escalar el rango de 10-2560 a 0-255)

- 2) La placa los recibe y cuando tiene los 2 bytes devuelve STX4.
- 3) *MatLab* vuelve a recibir STX4 y queda a la espera.
- 4) La placa guarda los N datos internamente, 1 por cada 10 ms que se ejecuta la interrupción, y enciende un LED azul para indicar que está guardo los siguientes datos (sin calibrar):
  - a. Roll calibrado del acelerómetro (*RollA*).
  - b. Pitch calibrado del acelerómetro (*PitchA*).
  - c. Velocidad angular calibrada en X del giróscopo (*GirX*).
  - d. Velocidad angular calibrada en Y del giróscopo (*GirY*).
  - e. Velocidad angular calibrada en Z del giróscopo (*GirZ*).
- 5) Cuando ha terminado de tomar medidas, la placa realiza los siguientes pasos:
  - a. Envía un dato simbólico, 33, para iniciar la interrupción por transmisión.
  - b. Desactiva la interrupción para tomar medidas de los sensores.
  - c. Apaga el LED azul
  - d. Enciende los LEDs rojos y verdes.
  - e. Envía byte a byte cada una de las N tramas de 16 bytes, conformadas de la siguiente forma:



n: nº de dato enviado (empezando por el 0), lo cual permite saber cuándo se ha tomado para representar gráficamente el tiempo en el eje horizontal.

- 6) *MatLab* recibe una a una las N tramas de 14 bytes. Sin embargo, para filtrar errores en la transmisión y no descoordinarse, realizará una lectura byte a byte. *MatLab* no empezará a construir internamente la trama hasta que no lea una

STX4, a partir del cual toma 13 bytes más. Cuando lea los 13, no volverá a construir otra trama hasta que lea otro STX4, por si se ha perdido algún byte por mitad.

- 7) Cuando la placa ha enviado todas las tramas, vuelve a activar la interrupción para tomar datos de los sensores y apaga los LEDs verdes y rojos.
- 8) Cuando *MatLab* ha terminado de descargar los datos, realiza una validación de los CHECKSUM y procesa los bytes llegados para ofrecer las aceleraciones y velocidades angulares en unidades con sentido físico. Llegados a este punto, el usuario puede representar gráficamente o guardar los datos.

#### **Comando 5: envío datos del sónar**

- 1) *MatLab* envía a la placa STX5.
- 2) La placa lo recibe y devuelve una trama de 2 bytes con el valor leído del registro del sonar, en forma de parte alta y baja.
- 3) *MatLab* la recibe, la vuelve a combinar, y aplica el factor de escala dividiendo por 58 para obtener una medida en centímetros.

#### **Comando 6: parada de emergencia**

- 1) *MatLab* envía STX6<sup>2</sup>.
- 2) La placa lo recibe y se bloquea:
  - a. Para los 4 motores,
  - b. Sale del modo de control manual de los motores en caso de que esté.
  - c. Apaga el LED azul.
  - d. Enciendo los dos LEDs verdes y rojos.
  - e. Deja de tomar medidas de los sensores, parando dicha interrupción.
- 3) *MatLab* vuelve a enviar STX6 y la placa se reinicia:
  - a. Apaga los LEDs rojos y verdes.
  - b. Se calibra.
  - c. Activa la interrupción para tomar medias.

#### **Comando 7: Arranque de motores**

- 1) *MatLab* envía a la placa STX7.
- 2) La placa lo recibe:
  - a. Devuelve STX7.
  - b. Aumenta el pulso de los motores hasta el valor nominal de arranque, cada 10 ms se aumenta el pulso de cada motor en 1  $\mu$ s.
  - c. Devuelve a *MatLab* el valor STX7+1.
  - d. La placa entra en modo control automático de motores.
- 3) *MatLab* lo recibe y permite realizar al usuario 4 nuevas acciones:
  - a. Parada de motores.
    - i. *MatLab* envía STX8.
    - ii. La placa lo recibe y lo devuelve.
    - iii. La placa pone directamente el pulso de parado (870  $\mu$ s) y para el PID.

---

<sup>2</sup> No puede coincidir con ningún otro byte enviado desde *MatLab*, aunque estemos dentro de control manual o automático.

- iv. La placa sale del modo de control automático de los motores.
- b. Ascenso.
  - i. *MatLab* envía 1.
  - ii. La placa lo recibe y activa el PID con el punto de equilibrio 1.
  - iii. La placa enciende el LED correspondiente.
- c. Mantener altura.
  - i. *MatLab* envía 2.
  - ii. La placa lo recibe y activa el PID con el punto de equilibrio 2.
  - iii. La placa enciende el LED correspondiente.
- d. Descenso.
  - i. *MatLab* envía 3.
  - ii. La placa lo recibe y activa el PID con el punto de equilibrio 3.
  - iii. La placa enciende el LED correspondiente.
- e. Modo hovercraft
  - i. *MatLab* envía 4.
  - ii. La placa lo recibe y para el PID con el punto de equilibrio 4.
  - iii. Se calibra el PID.
  - iv. La placa enciende el LED correspondiente.

### Comando 8: Parada de motores

- 1) *MatLab* envía STX8.
- 2) La placa lo recibe y devuelve STX8.
- 3) En caso de que esté en modo control automático de motores, la placa sale de él y se calibra el PID.

### Comando 9: Envío de constantes del PID

- 1) *MatLab* envía la trama con las constantes (13 bytes).

STX9	Kc <sub>x</sub>	Td <sub>x</sub>	Ti <sub>x</sub>	Kc <sub>y</sub>	Td <sub>y</sub>	Ti <sub>y</sub>
------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

- 2) La placa lo recibe y devuelve el valor de las constantes.
- 3) La placa actualiza el valor de las constantes.
- 4) *MatLab* comprueba la coincidencia de la trama enviada y recibida e informa al usuario.

#### 3.2.2.1 *Explicación de las tramas*

En el diseño de las tramas explicado anteriormente, se ha de tener en cuenta el significado de los colores de fondo empleados:

- Rojo: indica que ese byte es un STX.
  - STX0 = 166
  - STX1 = 202
  - STX3 = 177
  - STX4 = 132
  - STX5 = 231
  - STX6 = 094
  - STX7 = 080
  - STX8 = 211
  - STX9 = 052

- Amarillo: el dato en particular consta de 2 bytes y es enviado como parte alta y parte baja.
- Verde: el byte es un CHECKSUM. Para asegurarnos que su valor máximo es de 255, pues tan sólo se transmite en un byte, éste es calculado como el promedio de los bytes de la trama enviada (sin contar el STX).
- Gris: 1 bit.

### 3.2.3 Corte de Transmisión

Cada vez que *MatLab* espera recibir algún dato de la placa pone en marcha un temporizador ajustado a 2 segundos. Si durante ese periodo de tiempo no se activa la interrupción por recepción de la USART en la GUI, se cortará la comunicación o cancelará el comando correspondiente, avisando al usuario en la lista de mensajes.

De esta forma se evita que el programa quede bloqueado esperando datos que no vayan a llegar porque se han perdido, cosa que ocurre con frecuencia a la hora de pedir datos de los sensores a la placa con los comandos 1 y 4.

### 3.2.4 USART

Este es el código empleado para configurar el protocolo USART, el cual es realmente UART porque trabajamos en modo asíncrono. Al igual que el resto de líneas que hemos incluido en la memoria, se encuentra comentada la función que realiza la mayoría de ellas. No obstante, bajo el código incluimos varios aspectos a tener en cuenta:

```
////*** Configuración de la USART ***////
RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART3, ENABLE); // Habilitación del reloj
para la USART3

// Configuración de los parámetros de la USART3
USART_InitStructure.USART_BaudRate = 9600;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
USART_Init(USART3, &USART_InitStructure);
USART_Cmd(USART3, ENABLE);

// Configuración de la interrupción de la USART3
NVIC_InitStructure.NVIC_IRQChannel=USART3_IRQn;      // Canal USART3
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0; // Nivel de prioridad 0 (el
más alto)
NVIC_InitStructure.NVIC_IRQChannelSubPriority=0; // Nivel de subprioridad 0 (el más
alto, aquí no aplica
NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE; // Interrupción habilitada
NVIC_Init(&NVIC_InitStructure); // Se inicializa la interrupción de la USART3 con
la información de la estructura
USART_ITConfig(USART3, USART_IT_RXNE, DISABLE); // Se deshabilita la interrupción
por recepción de un dato hasta que calibremos
USART_ITConfig(USART3, USART_IT_TC, DISABLE); // Se deshabilita la interrupción por
trasmisión finalizada
```

Como se puede ver con cada comentario anterior, primero configuramos la *USART* con una velocidad de 9600 bytes/s, con una trama de 8 bits y sin bit de paridad. Además, deshabilitamos la interrupción por recepción hasta que calibremos, para que así no nos moleste ningún dato que pudiese llegar.

También se encuentra deshabilitada la transmisión con el fin que no se interrumpa el proceso si no se está transmitiendo ningún dato, ya que saltaría la interrupción cada vez que el *buffer* de salida se encontrase vacío, aunque se borrase el flag correspondiente.

### 3.3 Comunicación por cable

---

Además de la comunicación inalámbrica, nuestro proyecto consta de una importante circuitería física que detallamos a continuación:

#### 3.3.1 Conexiones internas

Para realizar de forma más sencilla la configuración y lectura del acelerómetro y del giróscopo se han creado funciones auxiliares con las que escribir y leer los registros de estos sensores.

Como podemos ver a continuación, en la configuración de ambos sensores utilizamos un LED de la placa de modo que si se enciende indica que se ha producido un fallo y se bloquea la misma.

##### 3.3.1.1 I2C

En nuestra placa *Discovery* establecemos una configuración de 2 líneas mediante el protocolo I2C entre el microcontrolador y el acelerómetro<sup>3</sup>, que son las siguientes:

Puerto	Línea	Función
PB6	SCL	Señal de reloj
PB7	SDA	Señal de datos

Comenzamos la configuración de la interfaz I2C2 habilitando la señal de reloj y estableciendo los parámetros de esta interfaz para que trabaje en modo esclavo y habilitando el ACK. El *Timing* del I2C es generado por el archivo Excel proporcionado:

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
I2C_InitStructure.I2C_AnalogFilter = I2C_AnalogFilter_Enable;
I2C_InitStructure.I2C_DigitalFilter = 0x00;
I2C_InitStructure.I2C_OwnAddress1 = 0x00;
I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
I2C_InitStructure.I2C_Timing = 0x00201D2B4;
```

---

<sup>3</sup> Empleamos el modelo LSM303DLHC.

```
I2C_Init(I2C1, &I2C_InitStructure);
```

A continuación activamos el interfaz I2C1 y configuramos el acelerómetro, lo cual implica que esta función escribe por el canal:

```
I2C_Cmd(I2C1, ENABLE); // Habilitación del interfaz I2C1

error = 1;
while ((error == 1) && (nAcc <= LIM_ACC_CONFIG))
{
    nAcc++;
    error = Config_Acc();
}
if (error == 1) // No se ha podido configurar correctamente, no avanzamos
{
    GPIO_SetBits(GPIOE, GPIO_Pin_14); // Se enciende el LED de aviso
    while(1);
}
```

Como ya hemos comentado anteriormente, intentamos configurar el acelerómetro cierto número de veces (LIM\_ACC\_CONFIG). Si no lo conseguimos, se enciende el LED PE14 y se queda bloqueada la placa en un *while* infinito.

La función completa utilizada para configurar el acelerómetro (Config\_Acc) puede encontrarse en el código del proyecto. No obstante, detallamos los aspectos más importantes de la misma.

En primer lugar comprobamos que el bus está libre, con una especie de temporizador que saltaría en caso de no obtener respuesta tras un cierto número de intentos (TIMEOUT\_CONST\_ACC):

```
TimeOut = TIMEOUT_CONST_ACC;
while((I2C_GetFlagStatus(I2C1, I2C_ISR_BUSY) != RESET) && (ErrorAcc==0))
{
    if((TimeOut--) == 0) ErrorAcc=1;
}
if (ErrorAcc == 1) // Hay error, no seguimos
{
    return 1; // Salimos de la función
}
```

Seguidamente configuramos la dirección del esclavo (0x32), el número de bytes a escribir (5), la generación automática de STOP y el modo de inicio por escritura:

```
I2C_TransferHandling(I2C1, 0x32, 5, I2C_AutoEnd_Mode, I2C_Generate_Start_Write);

// Se espera a que la dirección de dispositivo se haya enviado y confirmado
TimeOut = TIMEOUT_CONST_ACC;
while((I2C_GetFlagStatus(I2C1, I2C_ISR_TXIS) == RESET) && (ErrorAcc==0))
{
    if((TimeOut--) == 0) ErrorAcc=1;
}
if (ErrorAcc == 1) // Hay error, no seguimos
{
```

---

<sup>4</sup> Valor generado por el archivo Excel.

```

    return 1; // Salimos de la función
}

```

Una vez hemos enviado la anterior trama, si no ha habido errores, pasamos a configurar propiamente el acelerómetro, localizado en la dirección (`DirAcc_Config`<sup>5</sup> = `0x20` (primer registro a configurar)+ `0x80` = `0xA0`). Colocamos el primer bit a 1 con el objetivo de que los siguientes bytes enviados no sobrescriban el primer registro, es decir, que se desplace el puntero.

```
I2C_SendData(I2C1, (uint8_t) DirAcc_Config);
```

En caso de que el envío haya sido correcto continuamos configurando los siguientes registros recorriendo el vector `Conf_Acc[i]`, de acuerdo a la siguiente tabla:

Registro	Valor	Función
<code>0x20</code>	<code>0x47</code>	Modo 3 ejes Modo de consumo normal <code>ODR</code> = 95 Hz <code>Cut-off</code> = 25
<code>0x21</code>	<code>0x00</code>	Filtro de paso alto deshabilitado
<code>0x22</code>	<code>0x00</code>	Deshabilitamos interrupciones
<code>0x23</code>	<code>0x18</code>	Escala de $\pm 4g$ <sup>6</sup> Alta resolución activada

```

for (i=0; i<4; i++)
{
    I2C_SendData(I2C1, Conf_Acc[i]); //Configuración registro i
    TimeOut = TIMEOUT_CONST_ACC;
    while(((I2C_GetFlagStatus(I2C1, I2C_ISR_TXIS) == RESET) && (ErrorAcc==0))&&(i<3))
    {
        if((TimeOut--) == 0) ErrorAcc=1;
    }
    if (ErrorAcc == 1) // Hay error, no seguimos
    {
        return 1; // Salimos de la función
    }
}

```

En el último caso ya no verificamos si el flag de transmisión está a 1, sino que una vez acaba la secuencia de STOP (`I2C_GetFlagStatus(I2C1, I2C_ISR_STOPF)`), borramos el flag de final de secuencia y devolvemos `ErrorAcc = 0` pues no ha habido ningún error.

```
I2C_ClearFlag(I2C1, I2C_ICR_STOPCF);
return 0;
```

Una vez configurado correctamente, podemos leer del sensor. El proceso para obtener la señal de cada uno de los ejes del acelerómetro viene recogido en la función

---

<sup>5</sup> Ver *datasheet* acelerómetro página 25

<sup>6</sup> Escogimos esta escala porque era suficiente para el cálculo posterior de *roll* y *pitch*.

Lee\_Acc. Estas medidas las almacenamos en el vector tipo *charLect\_Acc*, de tamaño 6 (dos bytes por eje).

Comenzamos, al igual que en el proceso de escritura, comprobando si el bus está libre. Seguidamente indicamos la dirección del dispositivo, el número de bytes a escribir (únicamente la dirección del dispositivo) y los modos de inicio y finalización (escritura y manual, respectivamente):

```
I2C_TransferHandling(I2C1, 0x32, 1, I2C_SoftEnd_Mode, I2C_Generate_Start_Write);
```

A continuación, si no ha habido error, enviamos la dirección de dentro del dispositivo, al igual que al configurar el acelerómetro:

```
I2C_SendData(I2C1, (uint8_t) DirAcc_Config);
```

En caso de no haber habido error volvemos a configurar la transferencia con el objetivo de leer 6 bytes:

```
I2C_TransferHandling(I2C1, 0x32, 6, I2C_AutoEnd_Mode, I2C_Generate_Start_Read);
```

Finalmente procedemos a leer uno a uno los datos de cada registro, almacenándolos en el vector descrito anteriormente. Según la configuración empleada, la lectura<sup>7</sup> será la siguiente:

Registro	Señal	Registro	Señal	Registro	Señal
0x28	Parte baja eje X	0x2A	Parte baja eje Y	0x2C	Parte baja eje Z
0x29	Parte alta eje X	0x2B	Parte alta eje Y	0x2D	Parte alta eje Z

```
for (i=0;i<6;i++)
{
    Lect_Acc[i] = I2C_ReceiveData(I2C1);
    TimeOut = TIMEOUT_CONST_ACC;
    while(((I2C_GetFlagStatus(I2C1,I2C_ISR_RXNE) == RESET) && (ErrorAcc==0))&&(i<5))
    {
        if((TimeOut --) == 0) ErrorAcc=1;
    }
    if (ErrorAcc == 1) // Hay error, no seguimos
    {
        return 1; // Salimos de la función
    }
}
```

En el último caso, al igual que con la transmisión, no entramos esperamos a que se ponga el flag de recepción a 1, sino que verificamos si el flag de stop está a 1 y finalmente lo borramos, devolviendo ErrorAcc = 0:

```
TimeOut = TIMEOUT_CONST_ACC;
while(((I2C_GetFlagStatus(I2C1,I2C_ISR_STOPF) == RESET) && (ErrorAcc==0))
{
```

---

<sup>7</sup> Este valor viene dado en complemento a 2. La conversión la realizamos en el apartado 3.6.1.

```

        if((TimeOut --) == 0) ErrorAcc=1;
    }
    if (ErrorAcc == 1) // Hay error, no seguimos
    {
        return 1; // Salimos de la función
    }

I2C_ClearFlag(I2C1, I2C_ICR_STOPCF);
return 0;

```

### 3.3.1.2 SPI

En este caso se establece una configuración de 4 líneas entre el microcontrolador y el giróscopo<sup>8</sup> mediante el uso del protocolo serie SPI. Estas líneas son las siguientes:

Puerto	Línea	Función
PA5	SCK	Señal de reloj
PA7	MO	Escritura
PA6	MI	Lectura
PE3	CS	Señal de control

En primer lugar habilitamos la señal de reloj para el canal SPI1 de la placa *Discovery*.

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
```

Y luego configuramos la línea con diferentes parámetros: comunicación full dúplex donde la placa es el maestro, con un tamaño de trama de 8 bits con el primero el bit más significativo.

```

SPI_I2S_DeInit(SPI1);
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8;
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
SPI_InitStructure.SPI_CRCPolynomial = 7;
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_Init(SPI1, &SPI_InitStructure);

```

Por último configuramos la FIFO de recepción para que genere el evento cuando se haya recibido 8 bits, y habilitamos las interfaz SPI.

```

SPI_RxFIFOThresholdConfig(SPI1, SPI_RxFIFOThreshold_QF);
SPI_Cmd(SPI1, ENABLE);

```

---

<sup>8</sup> Empleamos el modelo L3GD20.

Ahora ya estamos en disposición de configurar el giróscopo mediante la línea SPI. Para ello empleamos la función L3GD20\_Config() que cambiará el valor de ciertos registros del sensor para buscar que nos proporcione las medidas deseadas. Esta función está incluida dentro de un bucle, donde en cada baso se lee la salida de la función, que es una variable que indica si se ha producido un fallo al configurar (error = 1) o no (error = 0). Si tras LIM\_GIR\_CONFIG veces que se intente configurar el giróscopo se produce un error, encenderemos el LED12 y dejaremos bloqueada la placa dentro de un *while* infinito.

```
error = 1;
while ((error == 1) && (nGir <= LIM_GIR_CONFIG))
{
    nGir++;
    error = L3GD20_Config();
}
if (error == 1) // No se ha podido configurar correctamente, no avanzamos
{
    GPIO_SetBits(GPIOE, GPIO_Pin_12); // Se enciende el LED de aviso
    while(1);
}
nGir = 0; // Limpiamos contador para otras funciones
```

La función L3GD20\_Config() viene a continuación:

```
intL3GD20_Config()
{
int i, DatoRx;
int ErrorGir = 0;
```

Ponemos a 0 la línea CS para indicar que vamos a transmitir.

```
GPIO_ResetBits(GPIOE, GPIO_Pin_3);
```

Esperamos cierto tiempo a que el registro de transmisión esté vacío, comprobando el flag SPI\_I2S\_FLAG\_TXE. En caso que pase TIMEOUT\_CONST\_GIR veces que intentemos comprobar si el flag está desactivado y no lo esté, salimos de la función con error a 1, poniendo antes la línea CS a 1 para finalizar la comunicación.

```
TimeOut = TIMEOUT_CONST_GIR;
while ((SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET) && (ErrorGir==0))
{
if((TimeOut--) == 0) ErrorGir=1;
}
if (ErrorGir == 1) // Hay error, no seguimos
{
    GPIO_SetBits(GPIOE, GPIO_Pin_3); // Paramos comunicación
    return 1; // Salimos de la función
}
```

Lo siguiente sería enviar al esclavo (giróscopo) en qué registro queremos que escriba los datos que le vamos a mandar posteriormente. La dirección del primer registro donde vamos a escribir está guardada en la constante DirGir\_Config, que es 0x20

correspondiente al primer registro de control del sensor (CTRL\_REG1)<sup>9</sup>. No obstante, y considerando la dirección como un número de 6 bits, hemos de añadir al principio (bit más significativo) un 0 para indicar que vamos a escribir, y un 1 en el segundo bit más significativo para indicar que se va a enviar más de un dato.

```
// Se envía la dirección dentro del dispositivo en la que se quiere escribir
SPI_SendData8(SPI1, 0x40|DirGir_Config); // Se le suma 01 al principio
// Se espera a que el registro de datos esté vacío
TimeOut = TIMEOUT_CONST_GIR;
while ((SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET) && (ErrorGir==0))
{
    if((TimeOut--) == 0) ErrorGir=1;
}
if (ErrorGir == 1) // Hay error, no seguimos
{
    GPIO_SetBits(GPIOE, GPIO_Pin_3); // Paramos comunicación
    return 1; // Salimos de la función
}
```

De nuevo esperamos a que el registro esté vacío con la misma gestión de error que realizamos previamente.

Dada la configuración de la comunicación SPI, tras enviar el dato el esclavo nos devolverá otro, aunque éste no es deseado y no contiene información relevante. De todas formas, para eliminarlo de la FIFO de recepción y que podamos leer posteriores datos y no queden en cola, hemos de leerlo por obligación:

```
DatoRx = SPI_ReceiveData8(SPI1); // Recibimos dato basura
```

Ya estamos en disposición de enviar los 5 datos. Recordemos que tras enviar cada byte, deberemos eliminar el dato basura recibido por el sensor.

La variable Conf\_Gir es un vector de 5 elementos que contiene los valores que se deben guardar en los 5 registros de configuración del giróscopo. Como hemos configurado la línea para transmitir más de 1 dato, automáticamente el esclavo entiende que debe sumar 1 al registro donde debe guardar un dato tras recibir el anterior.

Este proceso está automatizado en un bucle *for*:

```
// Vamos a transmitir los 5 datos al giroscopio
for (i=0;i<=4;i++)
{
    // Se envía el valor que se quiere escribir en dicha dirección
    SPI_SendData8(SPI1,Conf_Gir[i]);

// Se espera a que finalice la transmisión
TimeOut = TIMEOUT_CONST_GIR;
while ((SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET) && (ErrorGir==0))
{
    if((TimeOut--) == 0) ErrorGir=1;
}
```

---

<sup>9</sup> Ver *datasheet* giróscopo página 29.

```

if (ErrorGir == 1) // Hay error, no seguimos
{
    GPIO_SetBits(GPIOE, GPIO_Pin_3); // Paramos comunicación
    return 1; // Salimos de la función
}
DatoRx = SPI_ReceiveData8(SPI1); // Recibimos dato basura
}

```

De nuevo realizamos la gestión de posibles errores cada vez que se envía un dato.

Por último, finalizamos la comunicación poniendo la línea de CS a 1.

```

// Fin de la comunicación
GPIO_SetBits(GPIOE, GPIO_Pin_3); // Bit de CS a '1'
return 0; // Si hemos llegado aquí, no hay errores
}

```

Por último, cabe explicar qué valor hemos dado a cada registro del giróscopo:

Registro	Valor	Función
0x20	0x3F	Modo 3 ejes Modo de consumo normal ODR = 95 Hz Cut-off = 25
0x21	0x00	Filtro deshabilitado <sup>10</sup>
0x22	0x00	Deshabilitamos interrupciones
0x23	0x00	Actualización continua Bit menos significativo en la dirección más baja Resolución de ±250 dps Configuración de SPI de 4 líneas
0x24	0x00	Modo boot normal FIFO desactivada HPF desactivada

Por otra parte, para leer los datos del giróscopo empleamos la función L3GD20\_Read, cuyo primer parámetro es la primera dirección donde vamos a leer y el segundo es el número de bytes a leer, cada uno correspondiente a un registro 1 unidad superior.

Dada la configuración que hemos realizado del sensor, los registros que vamos a leer son los siguientes:

Registro	Señal	Registro	Señal	Registro	Señal
0x28	Parte baja eje X	0x2A	Parte baja eje Y	0x2C	Parte baja eje Z
0x29	Parte alta eje X	0x2B	Parte alta eje Y	0x2D	Parte alta eje Z

```

intL3GD20_Read(int Dir0, unsignedint numByte)
{

```

<sup>10</sup> No necesitamos este filtro pues empleamos posteriormente el filtro Kalman

```
int i, DatoRx;
int ErrorGir = 0;
```

Ponemos a 0 la línea CS para iniciar la comunicación.

```
GPIO_ResetBits(GPIOE, GPIO_Pin_3);
```

Antes de enviar ningún dato, esperamos a que se vacíe el registro al igual que hacíamos al configurar.

```
TimeOut = TIMEOUT_CONST_GIR;
while ((SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET) && (ErrorGir==0))
{
    if((TimeOut--) == 0) ErrorGir=1;
}
if (ErrorGir == 1) // Hay error, no seguimos
{
    GPIO_SetBits(GPIOE, GPIO_Pin_3); // Paramos comunicación
    return 1; // Salimos de la función
}
```

Todo y que queremos leer, lo primero que vamos a hacer es escribir en la línea el primer registro del que queremos recibir información, que será el 0x28 (parte baja de la señal del eje X). Ahora sumamos 11 a la parte más significativa antes de enviar el byte (es decir, sumamos 0xC0), para indicar que vamos a realizar un proceso de lectura de varias tramas.

```
SPI_SendData8(SPI1,Dir0|0xC0);
```

De nuevo esperamos a que se vacíe el registro, controlamos los errores y leemos el dato enviado por el sensor.

```
TimeOut = TIMEOUT_CONST_GIR;
while ((SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET) && (ErrorGir==0))
{
    if((TimeOut--) == 0) ErrorGir=1;
}
if (ErrorGir == 1) // Hay error, no seguimos
{
    GPIO_SetBits(GPIOE, GPIO_Pin_3); // Paramos comunicación
    return 1; // Salimos de la función
}
DatoRx = SPI_ReceiveData8(SPI1); // Dato basura
```

Y ya entramos en el proceso de lectura, donde antes de leer hemos de enviar un dato simbólico dado el funcionamiento del canal SPI. Éste dato no aportará ninguna información al giróscopo (al igual que el dato que leímos anteriormente), pero su envío es esencial para que continúe la comunicación.

Los datos que leamos de cada uno de los ejes se almacenaran en el vector de 6 elementos Lect\_Gir, que estará disponible fuera de la función de lectura junto con el vector que habíamos obtenido al leer el acelerómetro.

Cada vez que esperemos recibir un dato, esperaremos mediante un *while* a que el registro de recepción se llene, y mientras esté vacío no avanzaremos en el código.

```
// Leemos los numByte que nos han dicho
for (i=0;i<=numByte-1;i++)
{
    SPI_SendData8(SPI1, 0x00); // Envío dato basura

    // Se espera a que el dato se haya recibido
    TimeOut = TIMEOUT_CONST_GIR;
    while ((SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_BSY) == RESET) &&
(ErrorGir==0))
    {
        if((TimeOut--) == 0) ErrorGir=1;
    }
    if (ErrorGir == 1) // Hay error, no seguimos
    {
        GPIO_SetBits(GPIOE, GPIO_Pin_3); // Paramos comunicación
        return 1; // Salimos de la función
    }
    // Se lee el dato recibido
    Lect_Gir[i] = SPI_ReceiveData8(SPI1);
}

Tras leer los 6 bytes correspondientes a la parte baja y alta de cada uno de los 3 ejes, paramos la comunicación poniendo la línea de CS a 1 y devolviendo ausencia de errores.

GPIO_SetBits(GPIOE, GPIO_Pin_3);
return 0; // Si hemos llegado aquí, no hay errores
}
```

### 3.3.2 Conexiones externas

Además de las conexiones internas que trae de serie la placa *Discovery* entre el microcontrolador ST y los sensores anteriormente nombrados, hemos cableado una serie de conexiones externas indispensables para la construcción de un cuadricóptero a partir de la placa base. Dentro de dicho conjunto empezaremos con la batería.

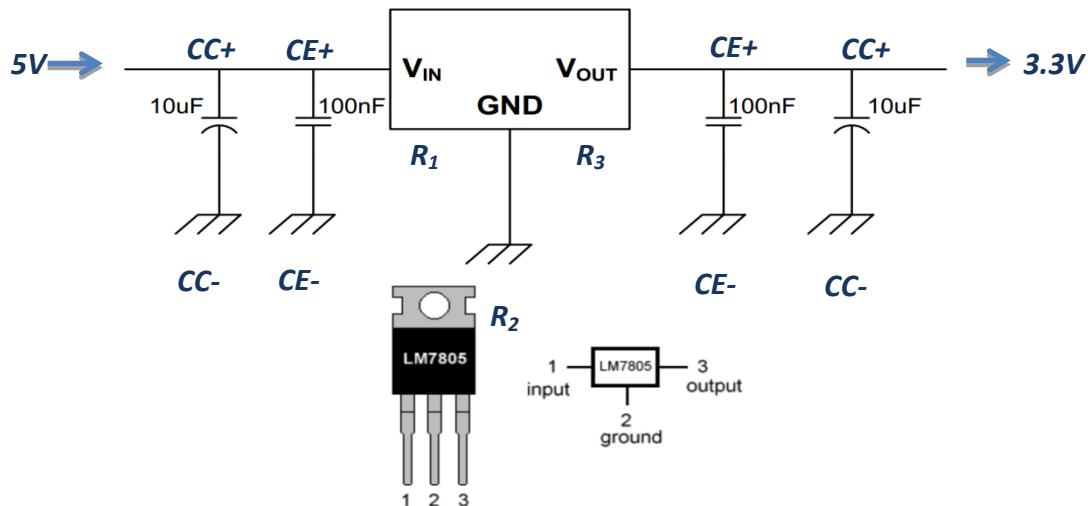
Este dispositivo nos da una tensión de alimentación 12V<sup>11</sup>. No obstante, tanto los reguladores *ESC*, el sonar y la placa *Discovery*, así como el módulo inalámbrico, funcionan a menor voltaje. Los tres primeros trabajan a 5V, mientras que el módulo inalámbrico lo hace a 3,3V.

En primer lugar, alimentamos un *ESC* (en este caso el *ESC4*) directamente con la tensión de la batería. Asimismo, este *ESC* actúa como regulador ya que la tensión de salida de dicho componente es de 5V, que es la que alimenta a todos los otros elementos que funcionan a dicho voltaje.

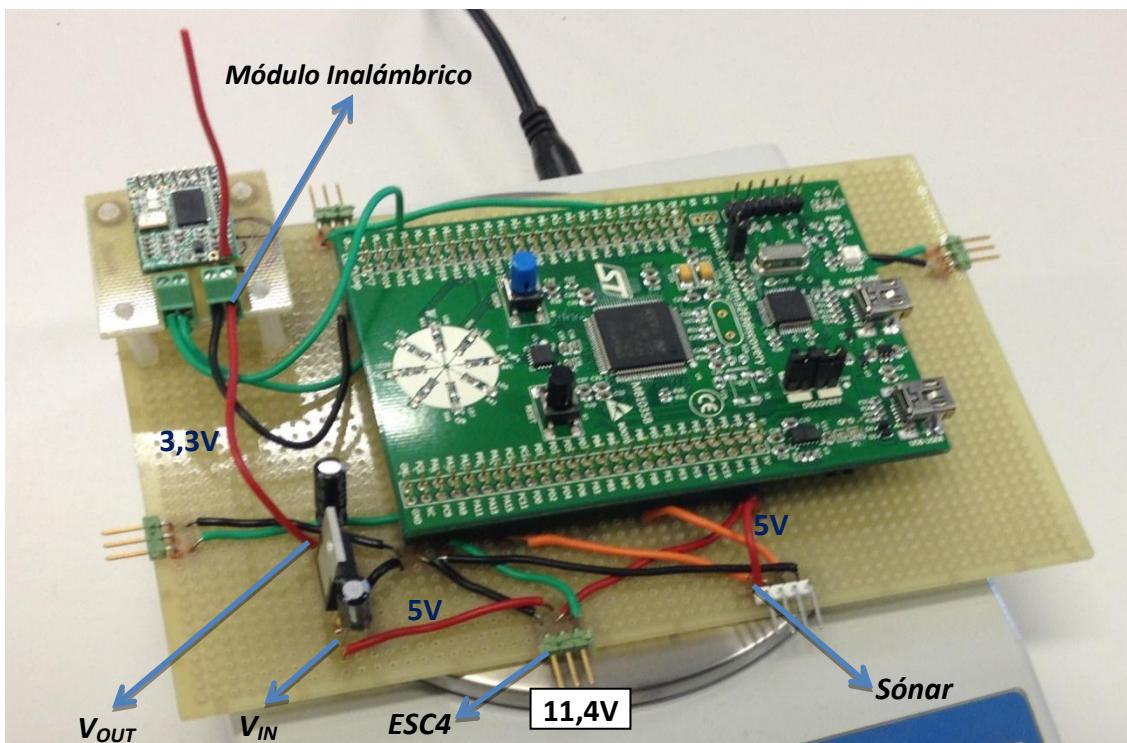
---

<sup>11</sup> En teoría la batería debería proporcionar 11,4 V.

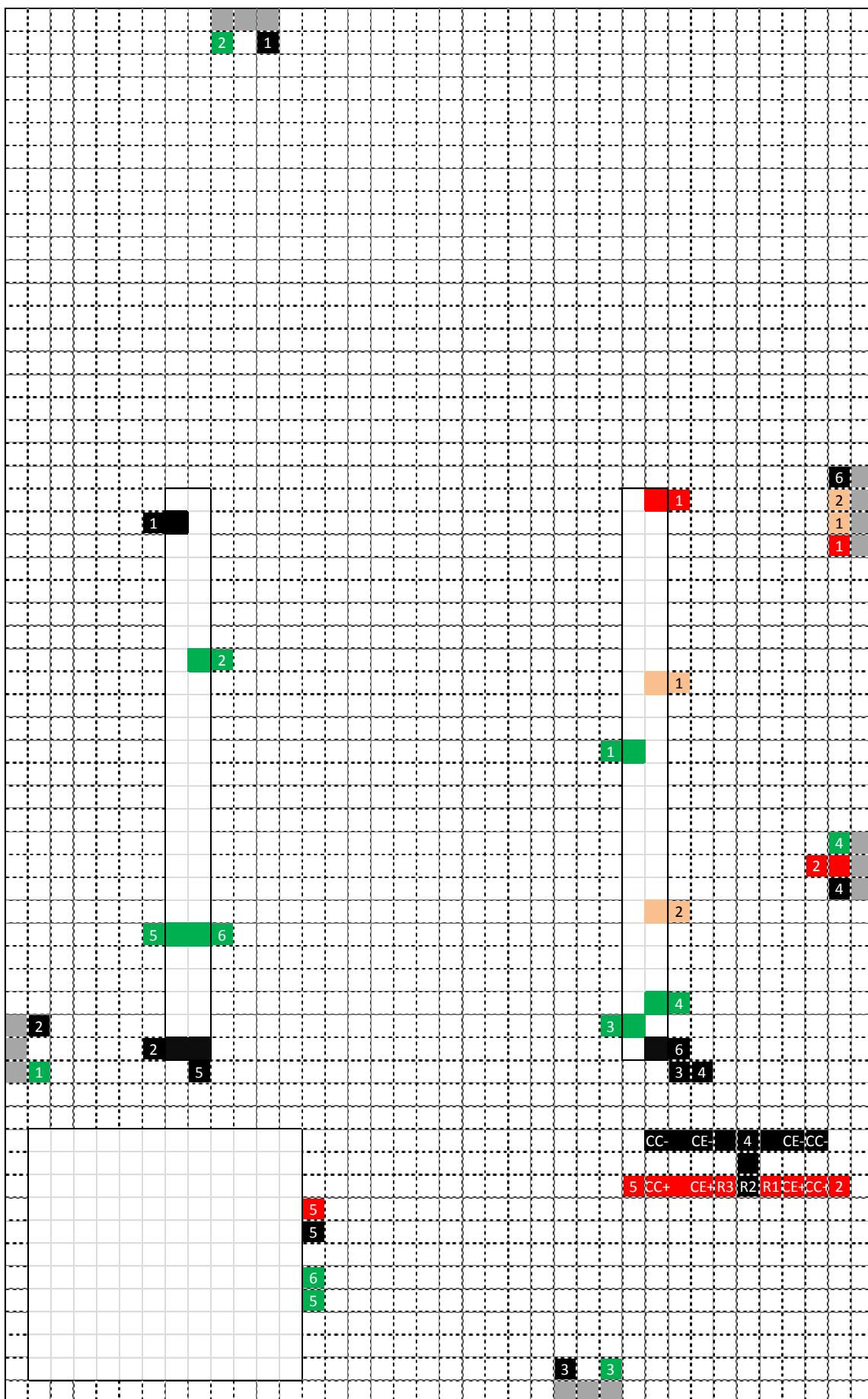
Por otro lado, es esencial que comentemos cómo se realiza esta transformación de voltaje para conseguir los 3,3V que necesita el módulo de comunicación inalámbrica. Esto se puede ver de manera muy sencilla en el siguiente esquema:



Como vemos, el voltaje ( $V_{IN}$ ) que entra es la salida del *ESC* es de un total de 5V y se filtra a través de dos condensadores, de 10 y 0,1 micro faradios respectivamente. El voltaje que llega es rectificado a través de un regulador modelo *LM7805*, disminuyendo la tensión a 3,3V. A continuación, el voltaje que sale ( $V_{OUT}$ ) de nuevo es filtrado a través de dos condensadores de 0,1 y 10 micro faradios. Con este voltaje ya podemos operar con el dispositivo inalámbrico. Si nos fijamos en los cables rojos:



Todo lo anteriormente comentado y más detalles misceláneos se pueden ver de forma más detallada y esquematizada en la siguiente distribución de pines, que adjuntamos en el archivo Excel [Distribución Pines.xlsx](#):



Cada cuadrado es una posición concreta definida por 2 coordenadas de nuestra placa de plástico sobre la cual hemos soldado los diferentes componentes electrónicos. Las uniones de pin a pin se visualizan con el mismo número, mientras que los colores están referidos a lo siguiente:

Color	Referencia
Rojo	Alimentación
Negro	Masa
Verde	Señal ESC + Inalámbrica
Rosa	Señal sónar
Azul	Conectores

Por su parte, las conexiones de los condensadores y de los reguladores se encuentran igualmente referenciadas en la tabla anterior y en el esquema de conexiones que vimos también con anterioridad.

### 3.4 IMU

---

Como sabemos, la IMU o Unidad de Medidas Iniciales, es un dispositivo electrónico que mide y nos da información acerca de la velocidad, orientación y fuerzas gravitacionales de nuestro cuadricóptero, usando combinación de los dispositivos con los que contamos integrados en la *Discovery*, es decir, acelerómetro, giróscopo y magnetómetro.

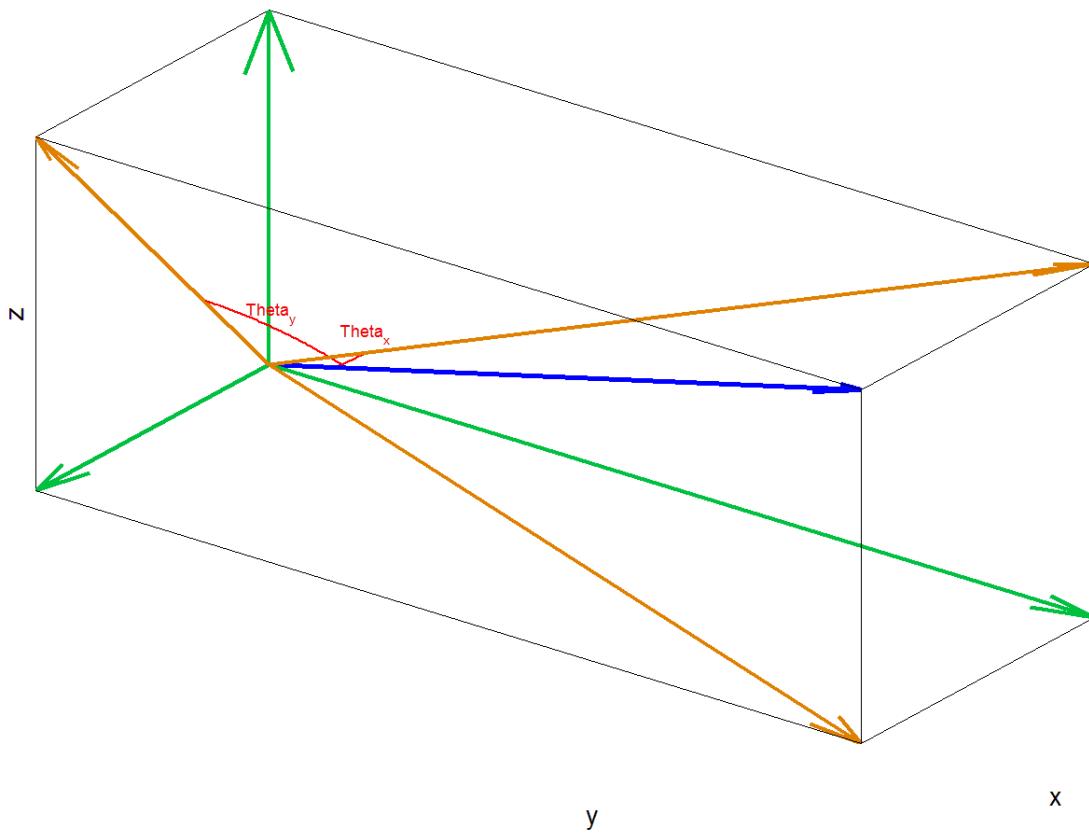
Por tanto, una vez realizada con éxito la comunicación con los sensores de la placa, hemos de interpretar estas señales y extraer información útil a partir de ellas: el *roll* y el *pitch*. Como ya hemos comentado anteriormente, éstos serán los ángulos que emplearemos para estabilizar el cuadricóptero y, por dicho motivo, es de vital importancia conocer cómo extraerlos a partir de las aceleraciones y las velocidades angulares que obtenemos del acelerómetro y del giróscopo respectivamente.

Lo que disponemos en realidad, combinando sendos transductores, es una IMU de 6 grados de libertad: 3 por cada sensor (1 por cada lectura en un eje independiente). En realidad la placa *Discovery* dispone también de un magnetómetro de 3 ejes, pudiendo llegar a formar una IMU de 9 grados de libertad. Sin embargo, en navegación inercial los más empleados son los dos primeros, pues para poder usar el último es necesario conocer el campo magnético del cuadricóptero (que es generado principalmente por la batería y los motores) y restarlo a las medidas. Ésta tarea no es sencilla en absoluto, y además no queríamos sobreexplotar los recursos de la placa. Por ello decidimos seguir adelante con los 6 grados de libertad.

### 3.4.1 Definición de Ángulos

En primer lugar, hemos de definir cómo hemos denominado cada ángulo, pues existen numerosas formas de caracterizarlos. Con el siguiente esquema que simula la placa inclinada en 3D con la presencia del vector gravedad, definimos los siguientes 4 ángulos que empleamos en los cálculos:  $\theta_x$ ,  $\theta_y$ ,  $\theta_{xx}$  y  $\theta_{yy}$ . El vector gravedad es representado en azul, su proyección sobre los planos coordinados en naranja y sobre los ejes en verde.

Para comprender mejor la dinámica del vehículo y, en el apartado de control, poder desacoplar las ecuaciones de los ejes X e Y, vamos a trabajar con los ángulos  $\theta_x$  y  $\theta_y$ , la magnitud de los cuales se puede ver en la siguiente imagen:



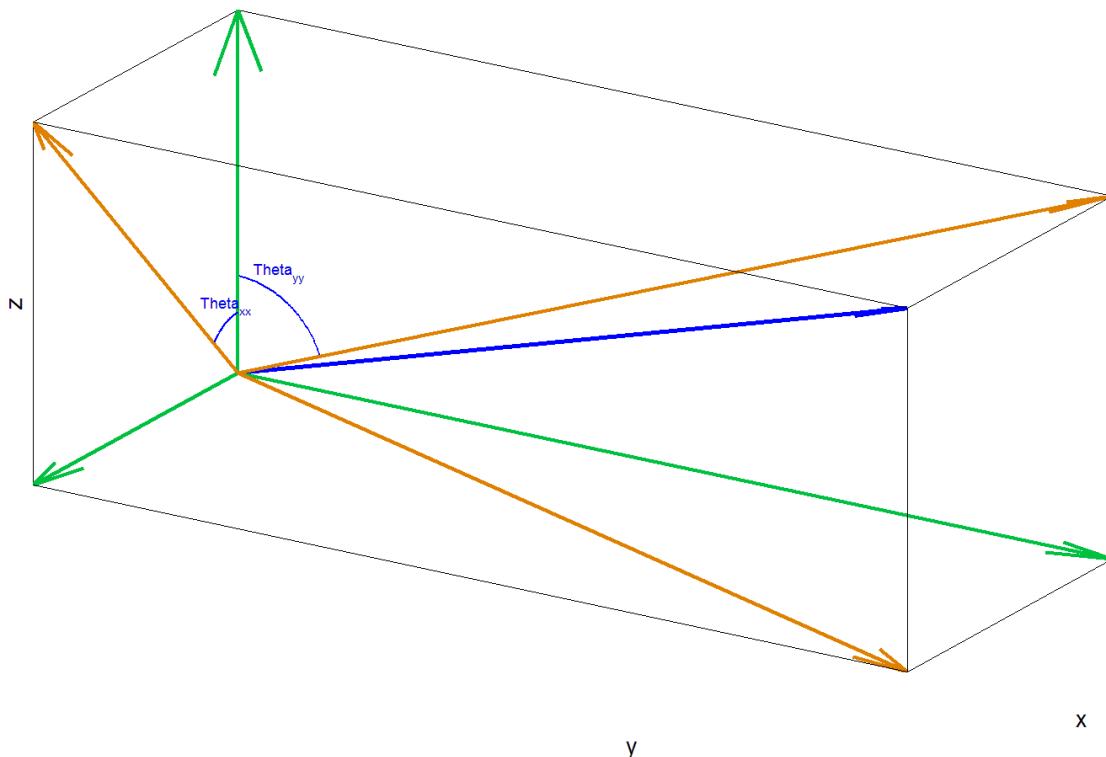
Éstos serán nuestros objetivos principales<sup>12</sup> a lo largo de nuestro proyecto y por eso los llamaremos *Roll* y *Pitch* respectivamente.  $\theta_{xx}$  y  $\theta_{yy}$  simplemente serán ángulos de apoyo para ciertos cálculos relacionados con el giroscopio. Las relaciones entre ambos ángulos se pueden obtener fácilmente por trigonometría:

$$\sin \theta_{xx} = \frac{\sin \theta_x}{\cos \theta_y}; \quad \sin \theta_{yy} = \frac{\sin \theta_y}{\cos \theta_x}$$

También podemos ver la representación de  $\theta_{xx}$  y  $\theta_{yy}$ :

---

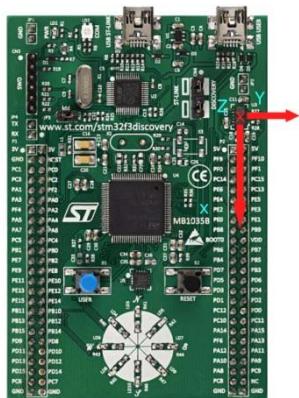
<sup>12</sup> Comúnmente en navegación inercial se emplea la siguiente definición:  $\theta_x = pitch$  y  $\theta_{yy} = roll$ .



### 3.4.2 Sistema de Referencia de Ángulos

Al igual que definíamos antes, y como recordatorio, mirando los *Datasheet* del acelerómetro y del giróscopo nos damos cuenta de que cada uno emplea su propio sistema de referencia. Por ello nosotros empleamos otro sistema de referencia en las ecuaciones, que nos sirve para uniformizar los cálculos y representa el sistema global. Todos los sistemas se resumen a continuación:

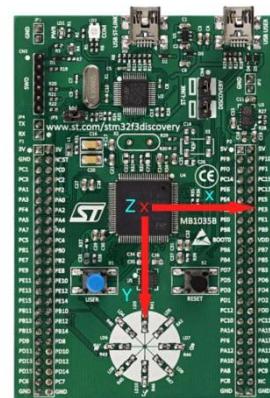
**Sistema usado por el acelerómetro<sup>13</sup>**



**Sistema usado por el giróscopo**



**Sistema empleado en las ecuaciones**



Nuestro sistema posee el eje X está orientado hacia la derecha, el eje Y hacia abajo y el Z entra al papel. Otra característica fundamental es que este sistema está ligado a la

<sup>13</sup> Este sistema nos dimos cuenta experimentalmente que todos los ejes están cambiados respecto a lo que dice el *Datasheet*, probablemente debido a que toma la aceleración como negativa.

placa, es decir, se mueve solidario con ella. Además, su origen es el centro de gravedad del cuadricóptero.

Emplear este sistema de referencia implica que a la hora de leer los sensores hemos de tener en cuenta las permutaciones de los ejes correspondientes:

$$\begin{array}{ll} \omega_x^0 = -\omega_x^1 & a_x^0 = a_y^1 \\ \omega_y^0 = \omega_y^1 & a_y^0 = a_x^1 \\ \omega_z^0 = -\omega_z^1 & a_z^0 = a_z^1 \end{array}$$

Donde el superíndice 0 indica el sistema de referencia general, y el 1 el particular de cada sensor.

### 3.4.3 Obtención de los ángulos del acelerómetro

El acelerómetro es un sensor que proporciona las aceleraciones a las que está sometido el dispositivo, sumando además las acciones gravitatorias de la siguiente forma:

$$\vec{a}_{sensor} = \vec{a}_{real} - \vec{g}$$

Si suponemos que nuestro dispositivo nos presenta aceleraciones reales, pues tratamos que se mantenga en equilibrio estático, o estas son despreciables, el acelerómetro directamente nos proporcionará las proyecciones del vector gravedad en los 3 ejes del cuadricóptero, siendo las medidas normales 0 g en los ejes X e Y, y 1 g en el eje Z.

En consecuencia, *roll* y *pitch* definidos anteriormente como el complementario del ángulo que forman los ejes X e Y con el vector gravedad se pueden obtener de la figura vista con anterioridad con trigonometría básica:

$$\theta_x = \text{atan} \frac{a_x}{\sqrt{a_y^2 + a_z^2}}; \quad \theta_y = \text{atan} \frac{a_y}{\sqrt{a_x^2 + a_z^2}}$$

Es posible observar que el rango de estos valores es de -90º (si  $a_i < 0$ ) a 90º (si  $a_i > 0$ ) siendo 0º el valor cuando la placa está situada sobre el plano horizontal, es decir, con el vector gravedad coincidente con el eje Z.

Como mejora de la interpretación del acelerómetro, podríamos considerar la diferencia de aceleraciones debida a que éste sensor no se encuentra sobre el centro de gravedad de nuestro aparato<sup>14</sup>. Ello hace que, si se está rotando, la aceleración lineal no sea la misma la que existe en el centro de nuestro sistema de referencia con el particular del sensor. Mediante la velocidad angular proporcionada por el giróscopo se puede suprimir este problema, pero introduce demasiados calculos intermedios y decidimos no incorporarlo en el algoritmo final.

---

<sup>14</sup> Las ecuaciones correspondientes se encuentran implementadas en el fichero *MathematicaDiferencia de aceleraciones.nb*

### 3.4.4 Obtención Ángulos Giróscopo

El giróscopo es un sensor que proporciona velocidades angulares, es decir, variación del ángulo con respecto al tiempo. Por ello, la obtención de los ángulos a partir de los datos del sensor requiere un paso de integración:

$$\theta = \int \omega dt + \theta_0$$

Donde  $\omega$  es la velocidad angular.

Aplicando el método de los trapecios, dadas medidas equidistantes en el tiempo un  $\Delta t$ , podemos aproximar numéricamente la integral anterior como:

$$\theta_{i-1} = \theta_i + \frac{\omega_{i-1} + \omega_i}{2} \Delta t, \quad i \geq 1$$

Sin embargo, la integración de la velocidad angular en ningún eje del sensor proporciona directamente el ángulo que habíamos definido para el acelerómetro. Realmente lo que nos está proporcionando el giróscopo es la velocidad angular del sólido en sus ejes fijos. Para obtener la variación con respecto al tiempo de los ángulos *roll* y *pitch* se han de considerar las relaciones entre la velocidad angular de un sólido rígido y los ángulos de Euler. Ésta relación depende de la actitud que tenga en apartado en cada momento, lo cual hace de este un proceso iterativo. No obstante, bajo una iteración, para no cargar mucho de cálculos a la placa, se puede obtener la siguiente relación:

$$\begin{aligned}\dot{\theta}_y &= \omega_x \cdot \cos \theta_{xx} - \omega_z \cdot \cos \theta_{yy} \\ \dot{\theta}_x &= -\omega_y \cdot \cos \theta_{yy} + \omega_z \cdot \cos \theta_{xx}\end{aligned}$$

Implementado en nuestro código con las siguientes instrucciones:

```
Qyp = ratio_GirX*cos(Qxx) - ratio_GirZ*sin(Qxx);
Qxp = -ratio_GirY*cos(Qyy) + ratio_GirZ*sin(Qyy);
```

Los ratios del *roll* y *pitch* serían respectivamente  $\dot{\theta}_x$  y  $\dot{\theta}_y$ <sup>15</sup>, la velocidad angular proporcionada por el giróscopo sería  $\omega_x$ ,  $\omega_y$  y  $\omega_z$ <sup>16</sup>, y los ángulos  $\theta_{xx}$  y  $\theta_{yy}$ <sup>17</sup> ya han sido enunciados anteriormente, y definen la actitud el cuadricóptero en ese instante.

Como vemos, el ratio del *roll* depende de la velocidad angular en los otros dos ejes, y únicamente es independiente del valor en su mismo eje. Para el ratio del *pitch* ocurre exactamente lo mismo.

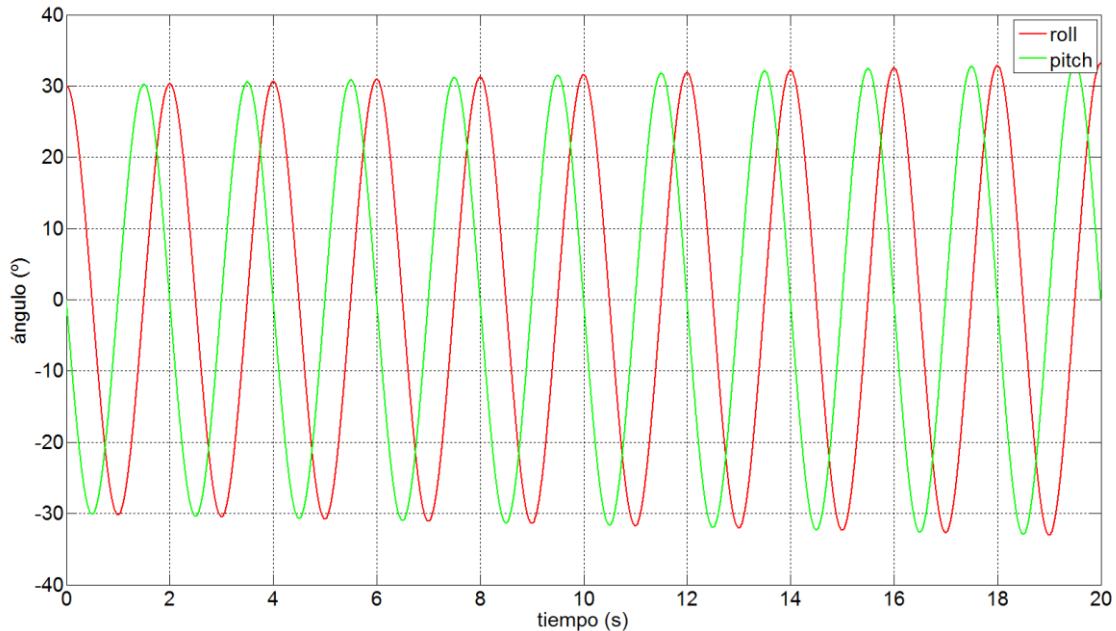
Este acople del problema tiene sentido, pues si pensamos un caso en el que el cuadricóptero esté inclinado 30º sobre el eje X, presentando inicialmente un *roll* de 30º y un *pitch* de 0º, y se produzca una variación angular en el eje Z (perpendicular al

<sup>15</sup>Qyp y Qxp en el código.

<sup>16</sup>ratio\_GirX, ratio\_GirY y ratio\_GirZ en el código.

<sup>17</sup>Qxx y Qyy en el código.

plano del aparato), vamos a llegar a un fenómeno que tanto el *roll* como el *pitch* cambian con el tiempo, como se observa en la siguiente gráfica<sup>18</sup>:



En conclusión, para obtener el *roll* y el *pitch* con el giróscopo no podemos integrar directamente la tasa de variación proporcionada por el giróscopo, si no que hemos de realizar una conversión previa entre ratios angulares. Una vez realizada, para cada paso de integración es posible aplicar trapecios de la siguiente forma:

$$\theta_x^i = \theta_x^{i-1} + \frac{\dot{\theta}_x^{i-1} + \dot{\theta}_x^i}{2} \Delta t; i > 0$$

$$\theta_y^i = \theta_y^{i-1} + \frac{\dot{\theta}_y^{i-1} + \dot{\theta}_y^i}{2} \Delta t; i > 0$$

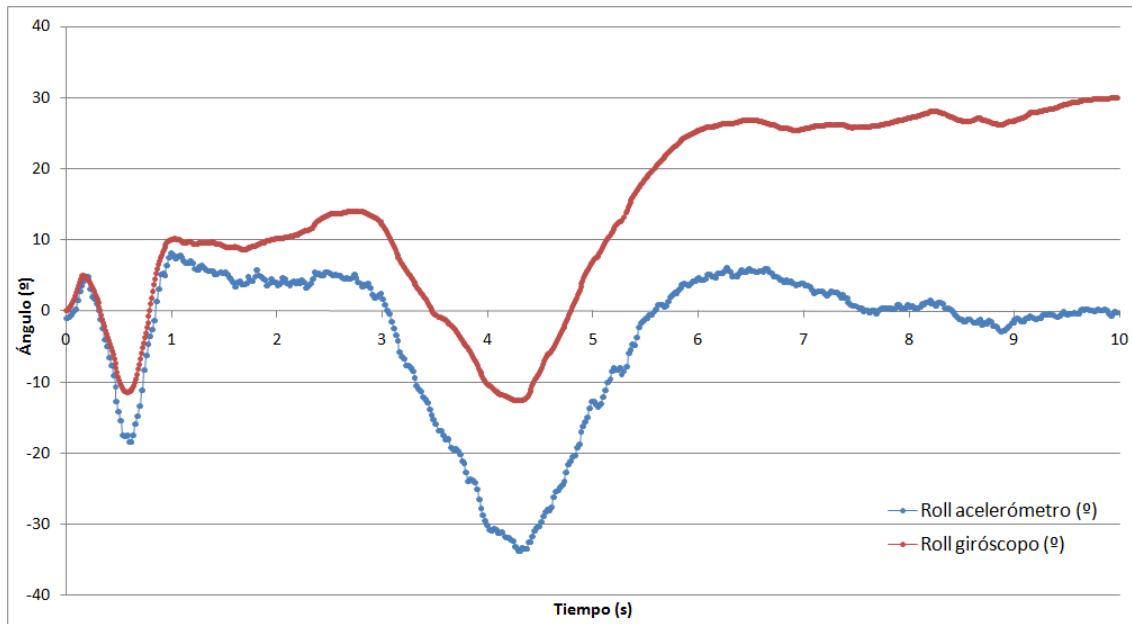
Sin embargo, como ya veremos posteriormente, este paso de integración se realiza dentro del filtro *Kalman*.

### 3.4.5 Calibración de los sensores

Una vez hemos configurado el acelerómetro y el giróscopo y se ha explicado el proceso para obtener *roll* y *pitch* a partir de ellos, pasamos a calibrarlos, pues los datos brutos de los sensores no ofrecen buenos resultados. Esto se puede ver en la siguiente gráfica, en la que se han calculado datos de *roll* de ambos sensores durante 10 segundos a partir de los datos obtenidos por los sensores empleando la comunicación inalámbrica.

---

<sup>18</sup> Los programas de *MatLabSimulacion\_roll\_pitch\_x.m* donde x va de 1 a 4 generan estas simulaciones

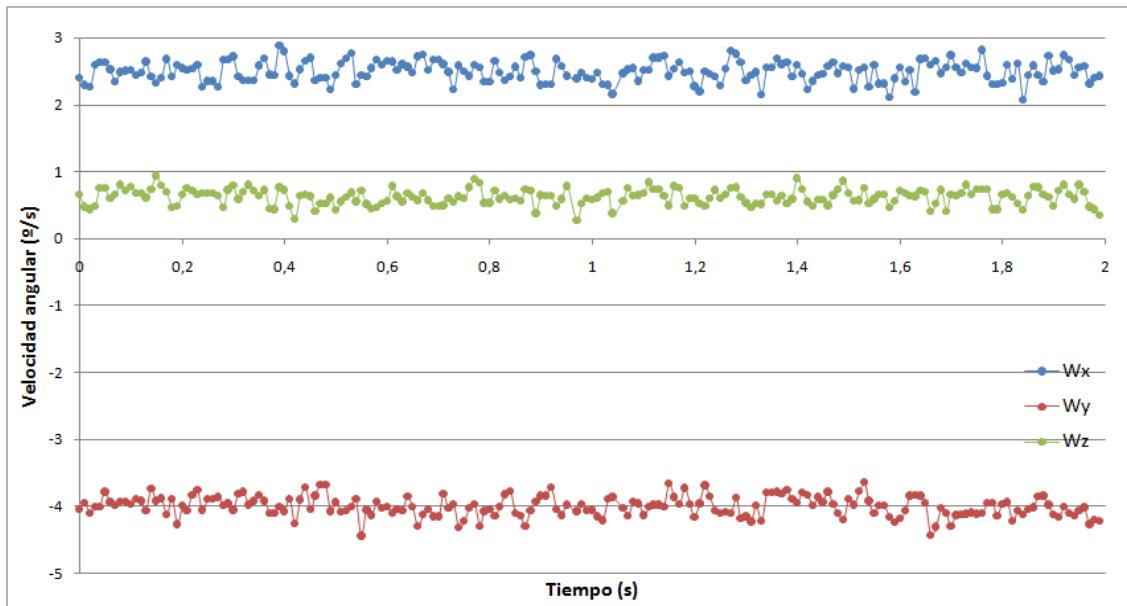


Como vemos las tendencias de los sensores son totalmente análogas, sin embargo, se precisa de cierta calibración, en especial para el giróscopo, para reducir su acumulación de error al integrar.

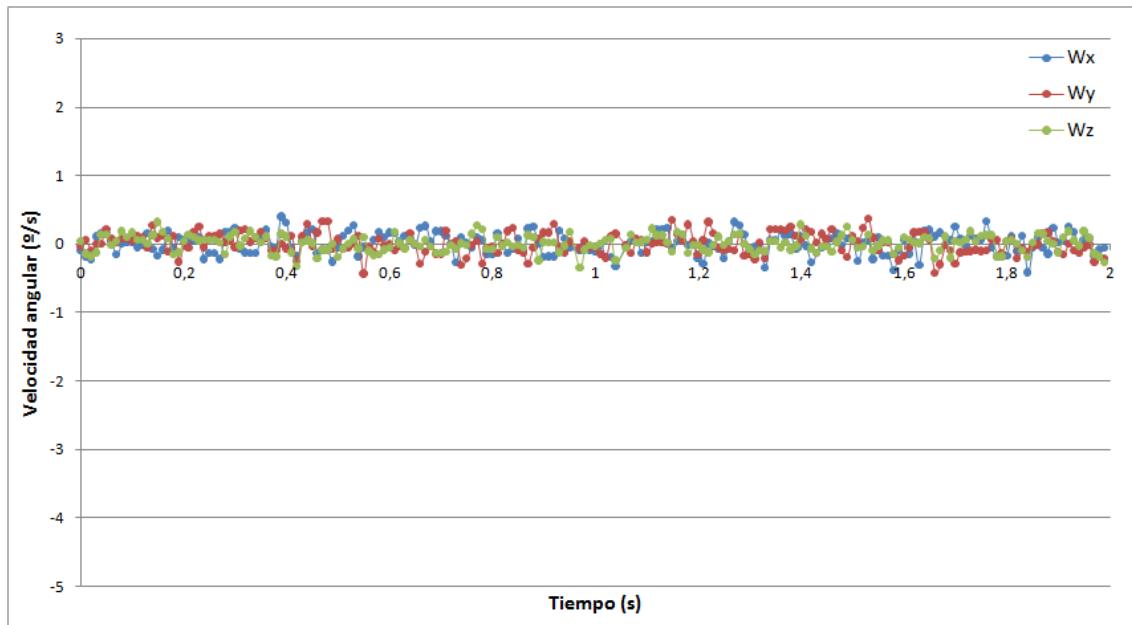
Por su parte, la característica propia del acelerómetro es más ruidosa que la del giróscopo, que representa una tendencia mucho más suave. En el apartado del filtro analizaremos mejor esta tendencia.

Estudiemos los sensores por separado, ahora en condiciones de placa situada sobre plano horizontal de tal manera que busquemos ángulos igual a 0 en el tiempo:

Tras demandas a la placa los datos de los sensores, en una posición estable, observamos la siguiente tendencia:



Donde se ve claramente que el giróscopo presenta cierto error intrínseco. Por tanto es necesario calibrar las medidas que, como vemos, presentan un offset constante. Restándole este desplazamiento obtenemos una medida mucho más limpia del sensor:



Ahora ya tenemos una señal mucho más próxima a 0 (no se desvía más de 0,5 °/s).

Su varianza, que emplearemos más tarde en la calibración del filtro Kalman es la siguiente:

- $\text{Var}(\omega_x) = 0,0232 \text{ } (\text{°}/\text{s})^2 = 0,0004 \text{ } (\text{rad}/\text{s})^2 = Q_{\text{giroBias\_pitch}}$
- $\text{Var}(\omega_y) = 0,0229 \text{ } (\text{°}/\text{s})^2 = 0,0004 \text{ } (\text{rad}/\text{s})^2 = Q_{\text{giroBias\_roll}}$
- $\text{Var}(\omega_z) = 0,0142 \text{ } (\text{°}/\text{s})^2$

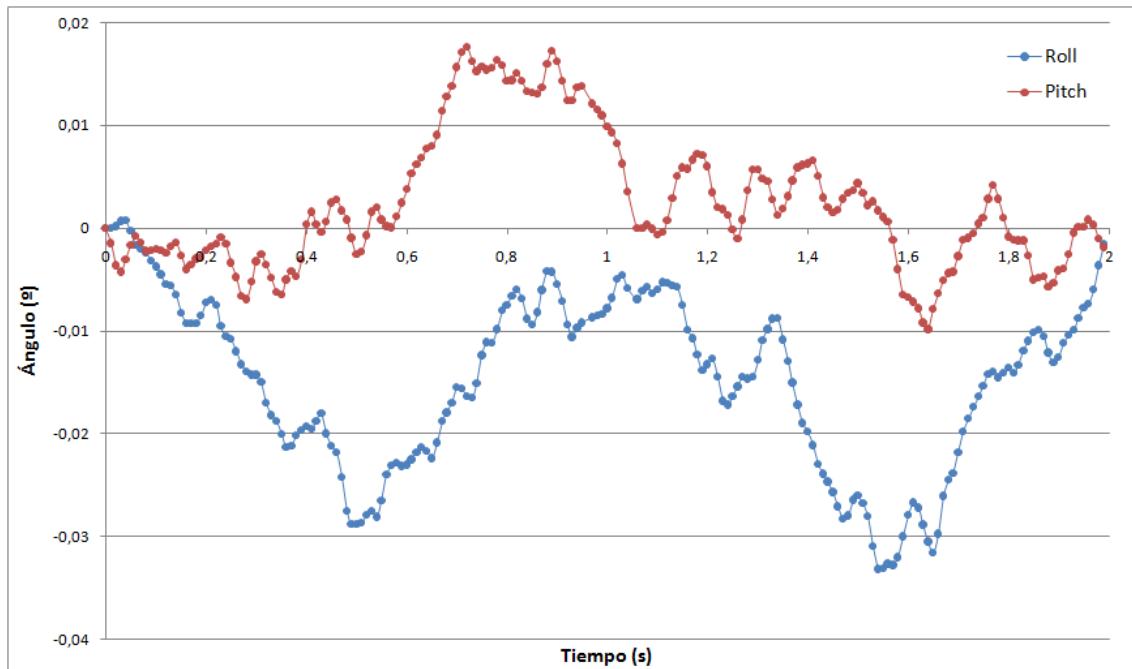
Donde `Q_giroBias_pitch` y `Q_giroBias_roll` son las variables que empleamos en el código del controlador.

Como estamos realizando un ensayo con  $\theta_x$  y  $\theta_y$  nulos,  $\theta_{xx}$  y  $\theta_{yy}$  también lo serán y por tanto, las ecuaciones que relacionan las variaciones temporales se reducen a:

$$\dot{\theta}_x = -\omega_y; \quad \dot{\theta}_y = \omega_x$$

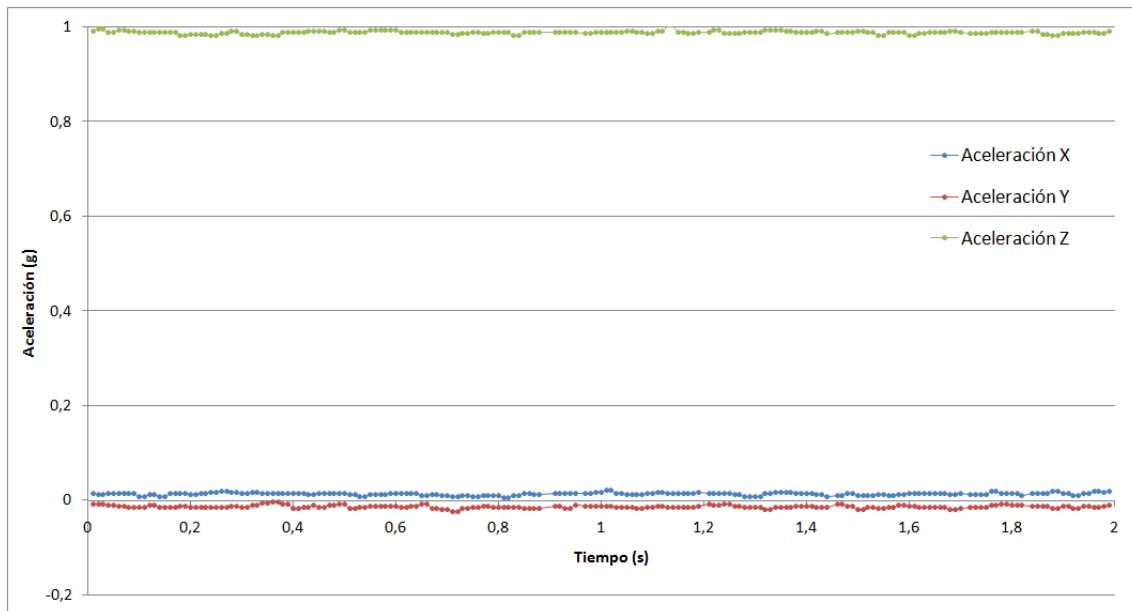
Por tanto la variancia del ratio del *roll* será directamente la varianza de la velocidad angular en Y del giróscopo, y análogo para el *pitch*.

Integrando esta señal sin el offset obtenemos unos datos muy claros de la actitud del aparato:



Apenas hay variación de los ángulos respecto al valor de equilibrio. Concluimos que es necesario implementar en la placa un algoritmo de calibración para los datos del giróscopo.

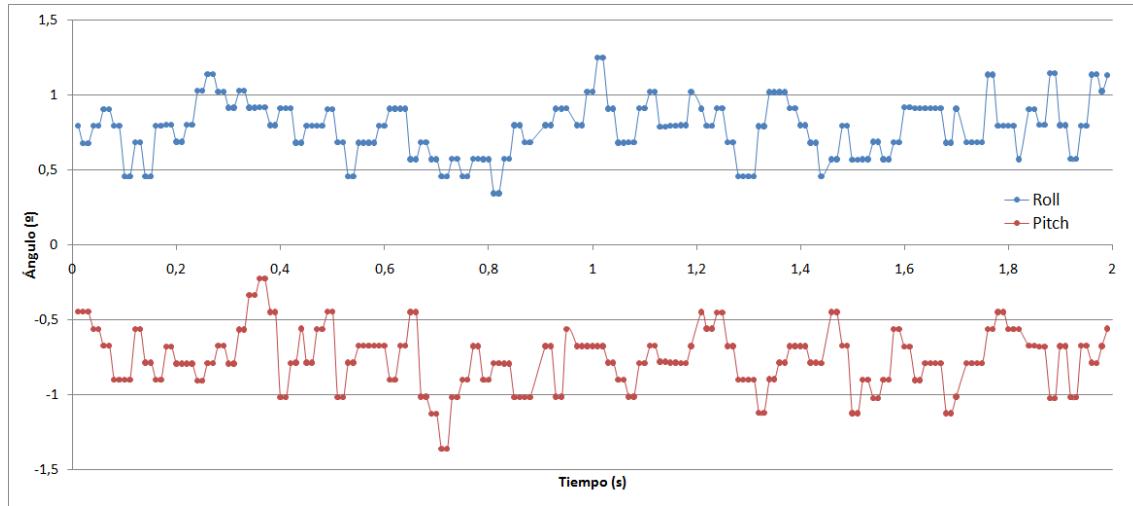
Por otra parte, estudiando los datos brutos sin calibrar del acelerómetro:



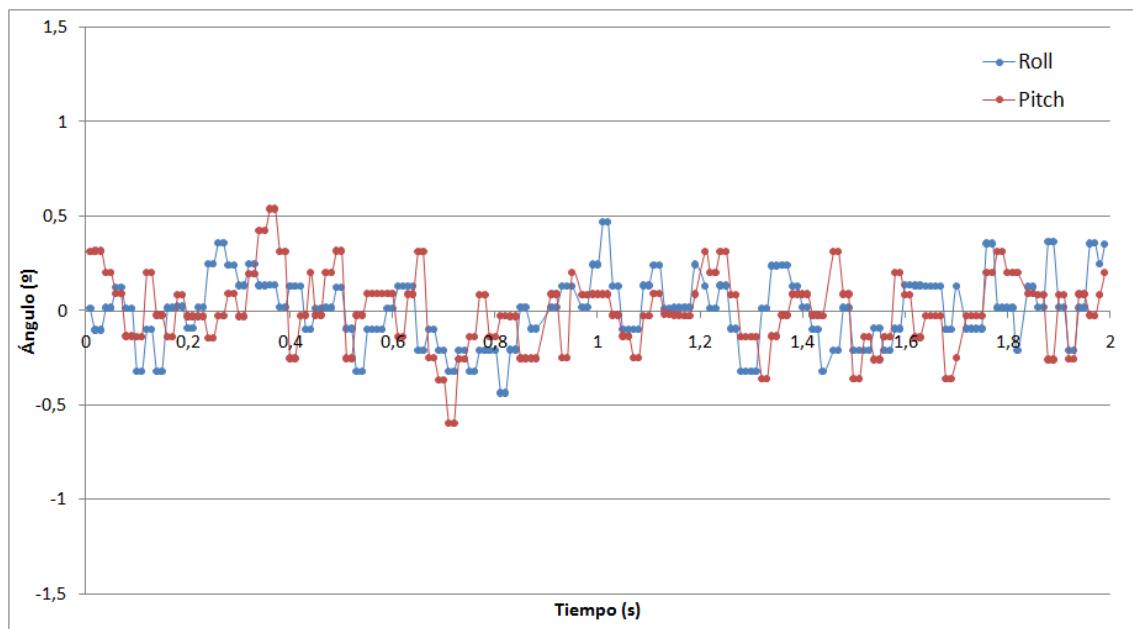
Vemos como las medidas sí se aproximan a los objetivos: 1g en el eje Z y 0 g en los otros dos. Los promedios serían los siguientes:

- $a_x = 0,0134 \text{ g}$
- $a_y = -0,0132 \text{ g}$
- $a_z = 0,9890 \text{ g}$

Sin embargo, si calculamos los ángulos a partir de estas medidas, observamos lo siguiente:



Tenemos medidas en torno a  $0,75^\circ$  con una media constante. Por tanto, podemos restar este *offset* tal y como hacíamos sobre los datos del giróscopo, pero esta vez sobre el ángulo calculado tras aplicar el arco tangente de tal manera que las dos líneas colapsen de la siguiente forma:



Obteniendo un resultado que no excede el medio grado.

Por su parte, la varianza de estas medidas es la siguiente:

- $\text{Var}(\theta_x^{\text{Acc}}) = 0.0341 \text{ } {}^\circ{}^2 = 0.00060 \text{ rad}^2 = \text{Q\_angle\_roll}$
- $\text{Var}(\theta_y^{\text{Acc}}) = 0.0392 \text{ } {}^\circ{}^2 = 0.00068 \text{ rad}^2 = \text{Q\_angle\_pitch}$

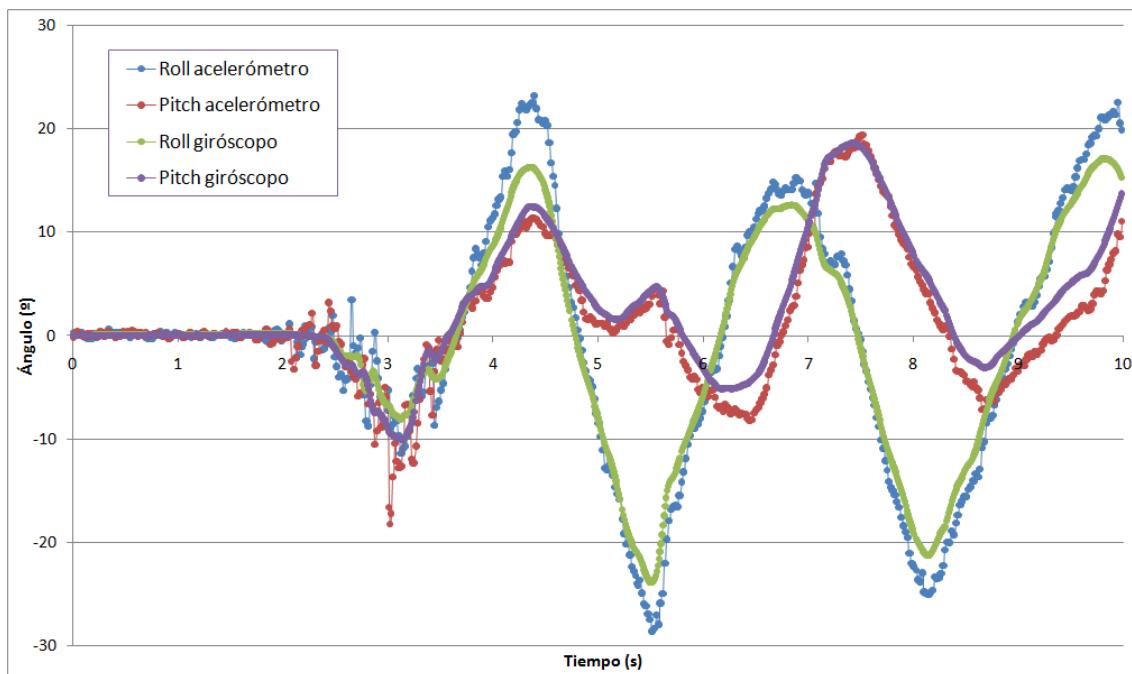
Donde Q\_angle\_roll y Q\_angle\_pitch son las variables empleadas en la calibración del filtro Kalman.

Si calculamos la varianza de la diferencia entre las medidas de ambos sensores, ya calibrados, sobre una experiencia estática para medir únicamente el ruido, obtenemos los siguientes parámetros:

- $\text{Var}(\theta_x) = 0,00063 \text{ rad}^2 = \text{R\_angle\_roll}$
- $\text{Var}(\theta_y) = 0,00104 \text{ rad}^2 = \text{R\_angle\_pitch}$

Donde `R_angle_roll` y `R_angle_pitch` son los últimos parámetros requeridos para calibrar el filtro.

Si mandamos las señales calibradas de ambos sensores podemos obtener unos resultados que se superponen, pero manteniendo en cierta medida la desviación del giróscopo y el ruido del acelerómetro:



Por ello se hace necesario incluir un filtro digital que nos dé una señal más limpia. Éste será el filtro Kalman, cuya implementación se explica [posteriormente](#).

### 3.4.6 Obtención de los ángulos en MatLab

Para realizar todo el trabajo descrito anteriormente fue necesario haber configurado previamente la comunicación inalámbrica, teniendo en cuenta las consideraciones descritas con [anterioridad](#).

En primer lugar, puesto que solo podíamos leer los datos desde el modo *debug* de *CooCox*, buscamos la forma de poder enviar la lectura bruta de los sensores a *MatLab*. Para ello, configuramos la interrupción de la USART con el comando STX1, de modo que al recibirla el microcontrolador se prepara para almacenar ángulos en una matriz

que hemos denominado *datos* en la cual rellenará una fila cada vez que se interrumpe con el temporizador 4 para leer los sensores.

Es virtualmente imposible enviar la trama requerida de 12 bytes entre cada interrupción, pues éstas se dan cada 10 ms y, trabajando la comunicación a 9600, la trama tarda 12,5 ms teóricos en transmitirse. Por ello es por lo que optamos por dividir el proceso en dos fases: almacenar los datos (el TIM4 trabaja de forma normal) y envío de datos (el TIM4 no interrumpe, solo lo hace la USART).

El código que realiza dichas funciones viene especificado a continuación:

```
DatoRx = USART_ReceiveData(USART3);
if (recepccion == 0) // Debemos recibir un STX e identificar el comando
{
    switch (DatoRx)
    {
        case 202: // Recopilar datos de los datos de los sensores
            recepcion = 1; // Sí esperamos algo más
            comando = 1;
            break;
    }
}
```

Una vez recibido el STX (202), la placa se queda a la espera de más órdenes (recepccion = 1), que se trata del número de datos a almacenar dividido por 10:

```
else// Recepción = 1
{
    switch (comando)
    {
        case 1: // Recibimos nº de datos a muestrear
            NumDatosAlmacenar = DatoRx+1; // Número de datos/10
            USART_SendData(USART3, STX1); // Enviamos confirmación
            USART_ITConfig(USART3, USART_IT_RXNE, DISABLE);
            l = 0; // Inicializamos contador de muestreo
            recepcion = 0; // No esperamos recibir más datos
            accion = 1; // Interrupción del TIM4 que debe almacenar datos
            GPIO_SetBits(GPIOE, GPIO_Pin_8); // LED almacenar datos
            break;
    }
}
```

Seguidamente, al poner accion = 1, en la interrupción del TIMER 4 se procederá a guardar datos en la matriz datos hasta que llegamos al número deseado (NumDatosAlmacenar):

```
if (accion == 1 && l < NumDatosAlmacenar*10)
{
    if(comando == 1)
    {
        ax_USART = ax + 32768;
        ay_USART = ay + 32768;
        az_USART = az + 32768;
        wx_USART = wx + 32768;
        wy_USART = wy + 32768;
        wz_USART = wz + 32768;

        datos[1][0] = ax_USART >> 8;
```

```

    datos[1][1] = ax_USART%256;
    datos[1][2] = ay_USART >> 8;
    datos[1][3] = ay_USART%256;
    datos[1][4] = az_USART >> 8;
    datos[1][5] = az_USART%256;
    datos[1][6] = wx_USART >> 8;
    datos[1][7] = wx_USART%256;
    datos[1][8] = wy_USART >> 8;
    datos[1][9] = wy_USART%256;
    datos[1][10] = wz_USART >> 8;
    datos[1][11] = wz_USART%256;
}

```

Al haber guardado ya todos los datos, enviamos un dato basura, con el objetivo de entrar en la interrupción por transmisión y construir ahí las tramas completas. Asimismo, deshabilitamos la interrupción del TIMER 4, pues no nos interesa que salte mientras estamos enviando datos al PC:

```

if(1 == NumDatosAlmacenar*10)
{
    accion = 2; // Para entrar en la interrupción por transmisión
    GPIO_ResetBits(GPIOE, GPIO_Pin_8); // Apagamos LED recopilación datos
    GPIO_SetBits(GPIOE, GPIO_Pin_9); // Encendemos LEDs envío de datos
    GPIO_SetBits(GPIOE, GPIO_Pin_11);
    GPIO_SetBits(GPIOE, GPIO_Pin_13);
    GPIO_SetBits(GPIOE, GPIO_Pin_15);
    cont = 0; // Indica qué trama estamos indicando (en el tiempo)
    cont_Buf = 0; // Indica qué dato estamos enviando (en la trama)
    l = 0; // Para futuras intervenciones
    TIM_ITConfig(TIM4, TIM_IT_Update, DISABLE);
    USART_ITConfig(USART3, USART_IT_TC, ENABLE);
    USART_SendData(USART3,33); // Dato basura transmisión
    return; // Salimos de la interrupción. No hace falta aplicar el filtro
}

```

La primera vez que entra en la interrupción enviamos el STX1 y construimos la trama<sup>19</sup> byte a byte, que enviamos en las siguientes interrupciones:

```

if (accion == 2) // Queremos recopilar datos
{
    switch (comando)
    {
        case 1:
            if (cont_Buf == 0)
            {
                USART_SendData(USART3,STX1); // Se envía el STX
                // Se construye la trama
                datos[cont][12]=cont >> 8;
                datos[cont][13]=cont%256;
                suma = 0;
                for (j=0;j<14;j++)

```

---

<sup>19</sup> Ver [apartado 3.2.2 Comandos empleados](#)

```

        suma = datos[cont][j] + suma;
    }
    suma=suma/14;
    cont_Buf=1;
}
else if (cont_Buf > 0 && cont_Buf < 15)
{
    USART_SendData(USART3,datos[cont][cont_Buf-1]);
    cont_Buf++;
}

```

Una vez se ha enviado toda la trama enviamos el CHECKSUM y añadimos un retardo con el objetivo de que no se satura el *buffer* de *MatLab*:

```

else // cont_Buf al máximo
{
    USART_SendData(USART3,suma);
    cont++;
    cont_Buf=0; // Lo siguiente es un STX
    retardo=0; // Añadimos un retardo
    while (retardo<40000)
    {
        retardo++;
    }
}

```

Finalmente, repetimos el proceso hasta que se han enviado todos los datos, volviendo a activar la interrupción del TIMER 4 y la recepción de datos, y deshabilitando la interrupción por transmisión:

```

if (cont == NumDatosAlmacenar*10)
{
    accion = 0; // Dejamos de enviar
    USART_ITConfig(USART3, USART_IT_TC, DISABLE);
    TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
    USART_ITConfig(USART3, USART_IT_RXNE, ENABLE);
}
break;

```

Posteriormente, una vez realizado el tratamiento de datos explicado en las secciones anteriores, añadimos la opción STX4 que tiene un formato similar a la descrita anteriormente, pero transmite datos ya calibrados y por ello carece de sentido repetir el código en esta memoria.

### 3.5 Filtro Kalman

---

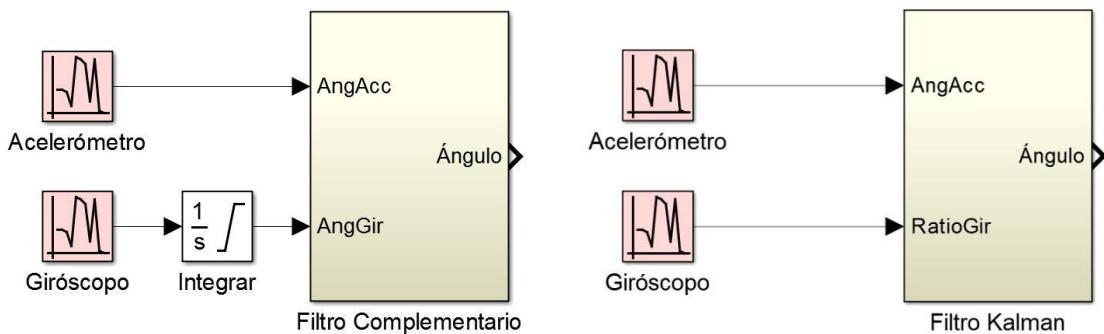
Tras las conclusiones anteriores es necesario aplicar un filtro de algún tipo. Analizando las capacidades del filtro complementario, en primer lugar con memorias de equipos anteriores y posteriormente con nuestros propios datos, nos dimos cuenta que no era capaz de solucionar el problema de la desviación del giróscopo sin acumular

demasiado ruido del acelerómetro. Además, su rango de validez temporal no es muy grande, y enseguida acumula errores y deja de ser válido.

Por ello decidimos implementar un filtro de nivel superior, muy empleado en navegación inercial: el filtro Kalman.

### 3.5.1 Diferencia con el Filtro Complementario

Como sistema en general, la principal diferencia entre ambos filtros es la entrada del mismo, como se puede observar en los siguientes diagramas:



Mientras que el filtro complementario toma el ángulo dado por el acelerómetro y la integración dada por el giróscopo, el filtro Kalman toma directamente la medida del giróscopo, que es una velocidad angular, y realiza la integración dentro del sistema.

Las ecuaciones internas del filtro avanzado le permiten calcular y por tanto eliminar el *offset* del giróscopo en cada paso de integración, de tal manera que se autocalibra con el tiempo y es por ello que realiza unos cálculos bastante correctos durante mayor periodo de tiempo.

Dicho proceso no es posible mediante el filtro complementario, pues la integración de la velocidad angular se realiza fuera el mismo y éste internamente no es capaz de estimar cuánto vale el *bias*.

### 3.5.2 Funcionamiento del filtro

El filtro Kalman se basa en, a partir de la varianza de las medidas, saber cuánto debe confiar en todo momento en las señales de cada sensor. El resultado es, como ya hemos enunciado anteriormente, que no sólo es capaz de seguir con precisión la media sin ruido del acelerómetro, sino que también es inmune al *offset* que presenta el giróscopo.

### 3.5.3 Consideraciones en la implementación del filtro

A la hora de implementar el filtro, como estamos ante un problema que es en principio 3D y por tanto con 6 grados de libertad (las 3 señales de los dos sensores), deberíamos implementar el filtro Kalman Extendido. Sin embargo, éste es muy complejo, presenta demasiados cálculos y requiere un tiempo de procesamiento más largo, además de consumir mucha más memoria. Como nos hemos visto limitados en cuanto a recursos

de almacenamiento del microprocesador, decidimos emplear las ecuaciones del filtro 1D, mucho más sencillas, y suponer el problema 2D desacoplado parcialmente del siguiente modo:

- Implementaremos de forma completamente equivalente las ecuaciones para el *roll* y para el *pitch*, como si de dos problemas 1D se tratase.
- Acoplamos las tres velocidades angulares que nos da el giróscopo para obtener los dos ratios temporales, el de variación del *roll* y del *pitch*, empleando las ecuaciones que nombramos en la obtención de los ángulos del giróscopo. Sobre ellos, restaremos los correspondientes *bias* y realizaremos la integración.

De esta forma asignamos dos *bias* ficticios, uno para el *roll* y otro para el *pitch*. Aquí estamos aproximando la realidad y desacoplando parte del problema como anunciamos previamente, pues lo normal sería asignar un *bias* a cada grado de libertad independiente del giróscopo, que en efecto son tres (eje X, eje Y y eje Z).

### 3.5.4 Parámetros del Filtro

Así como en el filtro complementario teníamos un único parámetro para definirlo, el tiempo experimental ( $\tau$ ) que permitía hacer más caso a la medida de un sensor y evadirse de la medida del otro, para implementar el código del filtro Kalman, es necesario definir algunos parámetros estadísticos:

- Varianza del ruido del proceso (Q):
  - Varianza de *roll* y *pitch* que calculamos del acelerómetro.
  - Varianza del *bias* del ratio del *roll* y ratio del *pitch*.
  - Covarianza de los dos parámetros anteriores, que suelen ser nulos.
- Varianza del ruido de las medidas (R) del *roll* y del *pitch*.

Estos parámetros nos llevan a otra diferencia con el filtro Complementario. Como veremos más tarde, el filtro Kalman da más peso a la medida de un sensor con respecto al otro de forma dinámica, y no fija como con el primer filtro.

A priori, las ecuaciones del filtro admiten que estos datos estadísticos dependan del tiempo. Sin embargo, para simplificar las ecuaciones, los supondremos constantes.

Por tanto, vamos a trabajar con un total de 6 parámetros, que vienen definidos en las ecuaciones por los siguientes nombre:

- Q del *roll*: Q\_angle\_roll.
- Q del *pitch*: Q\_angle\_pitch.
- Q del *bias* del *roll*: Q\_gyroBias\_roll.
- Q del *bias* del *pitch*: Q\_gyroBias\_pitch.
- R del *roll*: R\_angle\_roll.
- R del *pitch*: R\_angle\_pitch.

### 3.5.5 Variables del Filtro

En la implementación, hemos empleado distintas variables para las diferentes magnitudes que intervienen en el problema. Éstas son las siguientes:

- Qxp: Tasa de variación del *roll* (rad/s).

- $Q_{xp}$ : Tasa de variación del *pitch* (rad/s).
- $\text{ratio\_GirX}$ : señal calibrada del giróscopo en el eje X (rad/s).
- $\text{ratio\_GirY}$ : señal calibrada del giróscopo en el eje Y (rad/s).
- $\text{ratio\_GirZ}$ : señal calibrada del giróscopo en el eje Z (rad/s).
- $Q_{xx} = \theta_{xx}$  (rad).
- $Q_{yy} = \theta_{yy}$  (rad).
- $\text{rate\_Qx}$  = Tasa de variación del *roll*, sin el *bias* (rad/s).
- $\text{rate\_Qy}$  = Tasa de variación del *pitch*, sin el *bias* (rad/s).
- $\text{rate0\_Qx}$  = Tasa de variación del *roll* en el paso anterior, sin el *bias* (rad/s).
- $\text{rate0\_Qy}$  = Tasa de variación del *pitch* en el paso anterior, sin el *bias* (rad/s).
- $dt$ : paso temporal de integración = 10 ms.
- $P_{Qx}$ : matriz de error de covarianzas del *roll*.
- $P_{Qy}$ : matriz de error de covarianzas del *pitch*.
- $S_{Qx}$ : covarianza de innovación del *roll*.
- $S_{Qy}$ : covarianza de innovación del *pitch*.
- $yy_{Qx}$ : innovación del *roll*.
- $yy_{Qy}$ : innovación del *pitch*.
- $K_{Qx}$ : ganancia Kalman del *roll*.
- $K_{Qy}$ : ganancia Kalman del *pitch*.
- $\text{bias\_Qx}$ : *bias* del *roll* (rad/s).
- $\text{bias\_Qy}$ : *bias* del *pitch* (rad/s).
- $\text{roll}$ : *roll* salida del filtro (rad).
- $\text{pitch}$ : *pitch* salida del filtro (rad).
- $\text{roll\_Acc}$ : *roll* del acelerómetro.
- $\text{pitch\_Acc}$ : *pitch* del acelerómetro.

### 3.5.6 Ecuaciones del Filtro

El código implementado en el microcontrolador referente al filtro es el siguiente:

- 1) Cálculo de la tasa de variación del *roll* y del *pitch* a partir de los datos del giróscopo.

```
Qyp = ratio_GirX*cos(Qxx) - ratio_GirZ*sin(Qxx);
Qxp = -ratio_GirY*cos(Qyy) + ratio_GirZ*sin(Qyy);
```

- 2) Restamos a las tasas anteriores el *bias* que nos proporciona el filtro, calculado del paso anterior.

```
rate_Qx = Qxp - bias_Qx;
rate_Qy = Qyp - bias_Qy;
```

- 3) Realizamos la primera estimación del ángulo únicamente con los datos del giróscopo, integrando aplicando el método de los trapecios.

```
roll += dt * (rate_Qx + rate0_Qx)/2;
pitch += dt * (rate_Qy + rate0_Qy)/2;
```

- 4) Calculamos una primera estimación de la matriz de error de covarianzas  $P$ , que nos indica cuánto vamos a confiar en la predicción realizada anteriormente.

Cuanto menor sea su valor, mayor peso va a tener en el dato final la estimación ya realizada.

```
P_Qx[0][0] += dt * (dt*P_Qx[1][1] - P_Qx[0][1] - P_Qx[1][0] + Q_angle_roll);
P_Qx[0][1] -= dt * P_Qx[1][1];
P_Qx[1][0] -= dt * P_Qx[1][1];
P_Qx[1][1] += Q_gyroBias_pitch * dt;

P_Qy[0][0] += dt * (dt*P_Qy[1][1] - P_Qy[0][1] - P_Qy[1][0] + Q_angle_pitch);
P_Qy[0][1] -= dt * P_Qy[1][1];
P_Qy[1][0] -= dt * P_Qy[1][1];
P_Qy[1][1] += Q_gyroBias_pitch * dt;
```

- 5) Calculamos la innovación (yy) como diferencia entre la estimación del ángulo realizado y la señal de acelerómetro.

```
yy_Qx = roll_Acc - roll;
yy_Qy = pitch_Acc - pitch;
```

- 6) Calculamos el parámetro S, o covarianza de innovación. Cuanto mayor sea S, debido a que P o la varianza R es más grande, menos confiamos en la medida del acelerómetro.

```
S_Qx = P_Qx[0][0] + R_angle_roll;
S_Qy = P_Qy[0][0] + R_angle_pitch;
```

- 7) Obtenemos la ganancia Kalman, K, que es más pequeña cuanto mayor sea S o menor sea P, que se produce cuando no confiamos mucho en el acelerómetro.

```
K_Qx[0] = P_Qx[0][0] / S_Qx;
K_Qx[1] = P_Qx[1][0] / S_Qx;
```

```
K_Qy[0] = P_Qy[0][0] / S_Qy;
K_Qy[1] = P_Qy[1][0] / S_Qy;
```

- 8) Añadimos al ángulo que habías calculado previamente con el giróscopo cierta cantidad aportada por la innovación ponderada con la ganancia Kalman.

```
roll += K_Qx[0] * yy_Qx;
pitch += K_Qy[0] * yy_Qy;
```

- 9) Actualizamos el *bias* en base también a la ganancia Kalman, para el siguiente paso.

```
bias_Qx += K_Qx[1] * yy_Qx;
bias_Qy += K_Qy[1] * yy_Qy;
```

- 10) Actualizamos la matriz de error de covarianzas P para el siguiente paso, de nuevo empleado K.

```
P_Qx[0][0] -= K_Qx[0] * P_Qx[0][0];
P_Qx[0][1] -= K_Qx[0] * P_Qx[0][1];
P_Qx[1][0] -= K_Qx[1] * P_Qx[0][0];
P_Qx[1][1] -= K_Qx[1] * P_Qx[0][1];

P_Qy[0][0] -= K_Qy[0] * P_Qy[0][0];
P_Qy[0][1] -= K_Qy[0] * P_Qy[0][1];
P_Qy[1][0] -= K_Qy[1] * P_Qy[0][0];
P_Qy[1][1] -= K_Qy[1] * P_Qy[0][1];
```

11) Calculamos el ratio del *roll* y del *pitch* con el nuevo *bias*, para poder integrar en el paso posterior.

```
rate0_Qx = Qxp - bias_Qx;
rate0_Qy = Qyp - bias_Qy;
```

12) Los ángulos  $\theta_{xx}$  y  $\theta_{yy}$  serán los siguientes:

```
Qxx = asin(sin(roll )/cos(pitch));
Qyy = asin(sin(pitch)/cos(roll) );
```

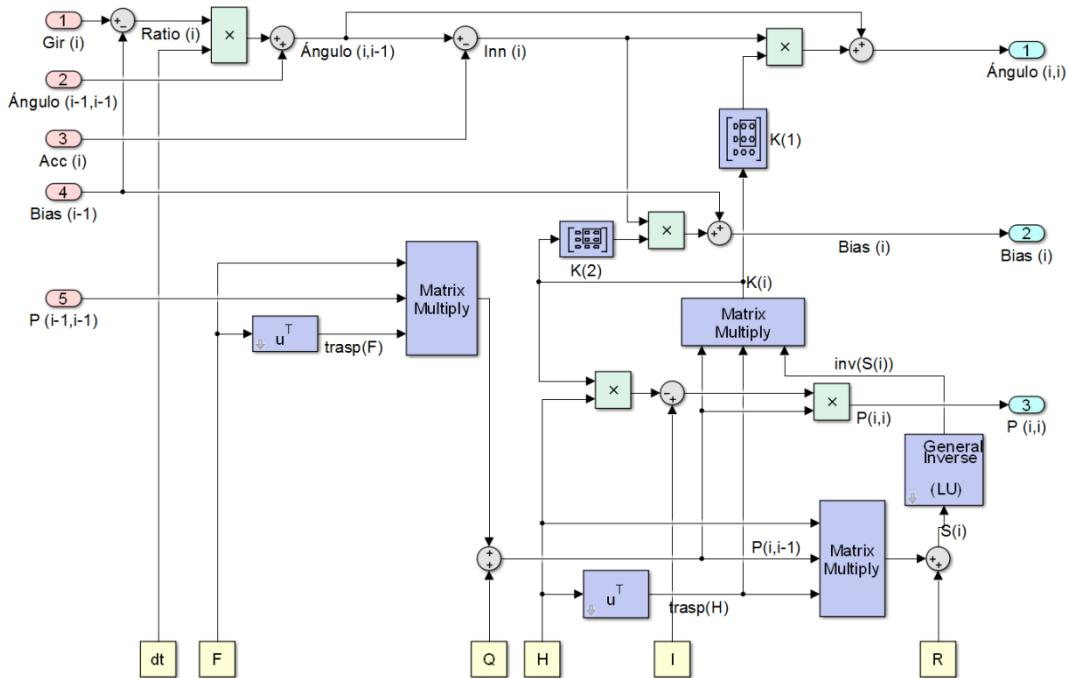
### 3.5.7 Entradas y salidas del sistema

Para acoplar el filtro con otros elementos, es necesario verlo como un sistema con ciertas entradas y salidas.

En primer lugar, si realizamos las siguientes definiciones:

$$H = [1 \quad 0]; F = \begin{bmatrix} 1 & -\Delta t \\ 0 & 1 \end{bmatrix}; Q = \begin{bmatrix} Q_{\text{ángulo}} & 0 \\ 0 & Q_{\text{bias}} \end{bmatrix}$$

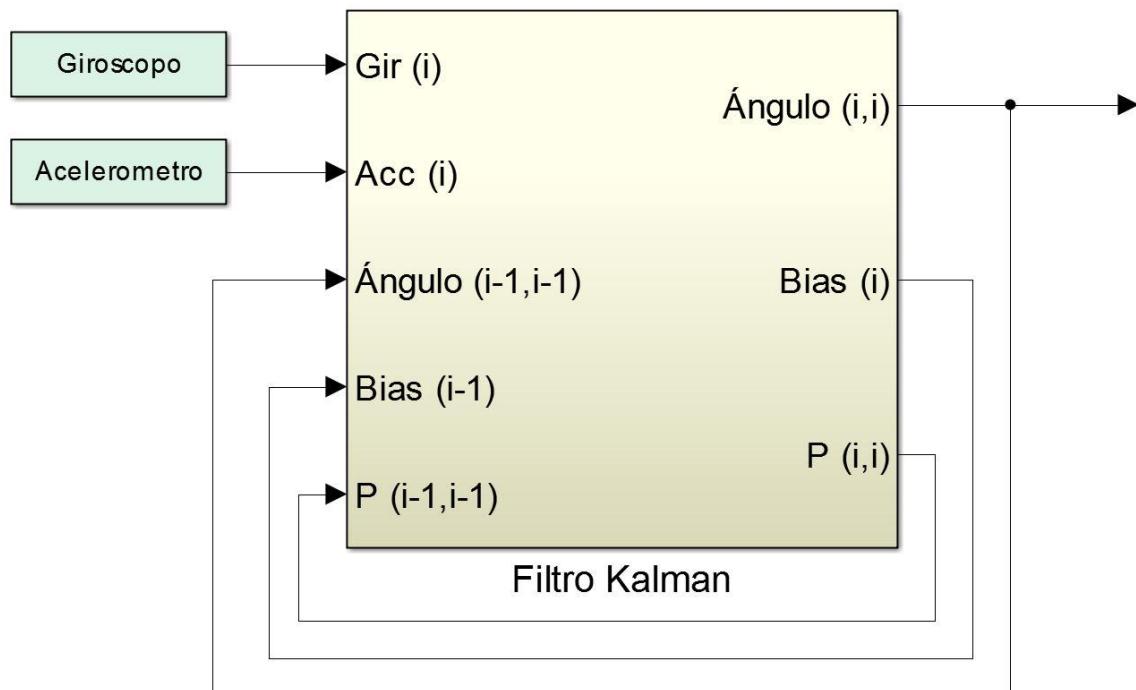
Podemos describir el filtro 1D para un ángulo cualquiera con el siguiente esquema:



Los colores de dicho esquema indican diferentes operaciones o funciones:

- Rojo: entradas.
- Azul: salidas.
- Gris: sumadores.
- Verde: productos escalares.
- Violeta: operaciones matriciales.
- Amarillo: constantes del problema.

Ahora ya somos capaces de definir el filtro como un sistema de caja negra, con ciertas retroalimentaciones, lo cual nos será útil para explicar posterior apartados:



Donde las entradas serían:

- Velocidades angulares del giróscopo.
- Ángulos del acelerómetro.

Y las retroalimentaciones:

- Ángulo del filtro.
- *Bias* de los ángulos.
- Matriz *P* de error de covarianzas.

La salida del ángulo del filtro será recogido por nuestro controlador PID, cuya implementación y esquematización veremos el [apartado 3.9](#).

### 3.5.8 Calibración del filtro y estimación de constantes

Una vez funciona correctamente la comunicación y además hemos calibrado correctamente los sensores, el siguiente paso es calibrar los parámetros experimentales del filtro Kalman.

Para tal fin podemos pedir tandas de 1.000 medidas calibradas a la placa mediante nuestra GUI y, empleando una función de MatLab que implementa el filtro Kalman de forma idéntica a cómo está programa en la placa<sup>20</sup>, ver cómo funciona el filtro con distintos valores de las constantes, a las cuales les daremos posteriormente un valor en el programa de la *Discovery*.

<sup>20</sup> Hablamos del script *Filtro\_Kalman\_2D.m*.

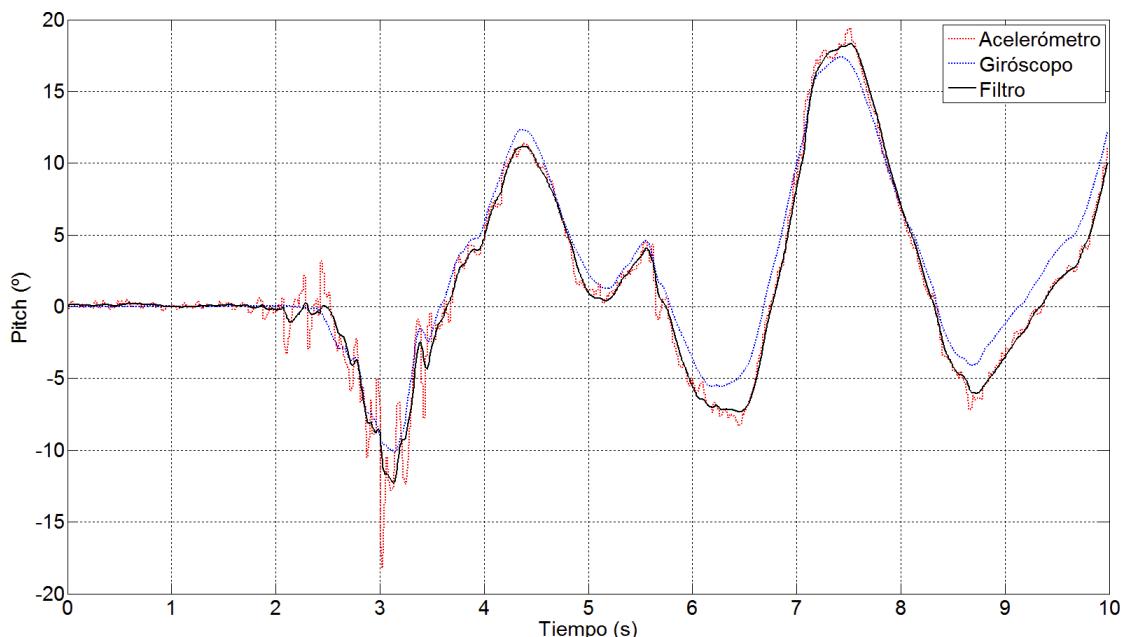
Para realizar una buena comparación, además programamos en el script anterior que generase una gráfica con las siguientes líneas, para el caso del *roll* y del *pitch* por igual:

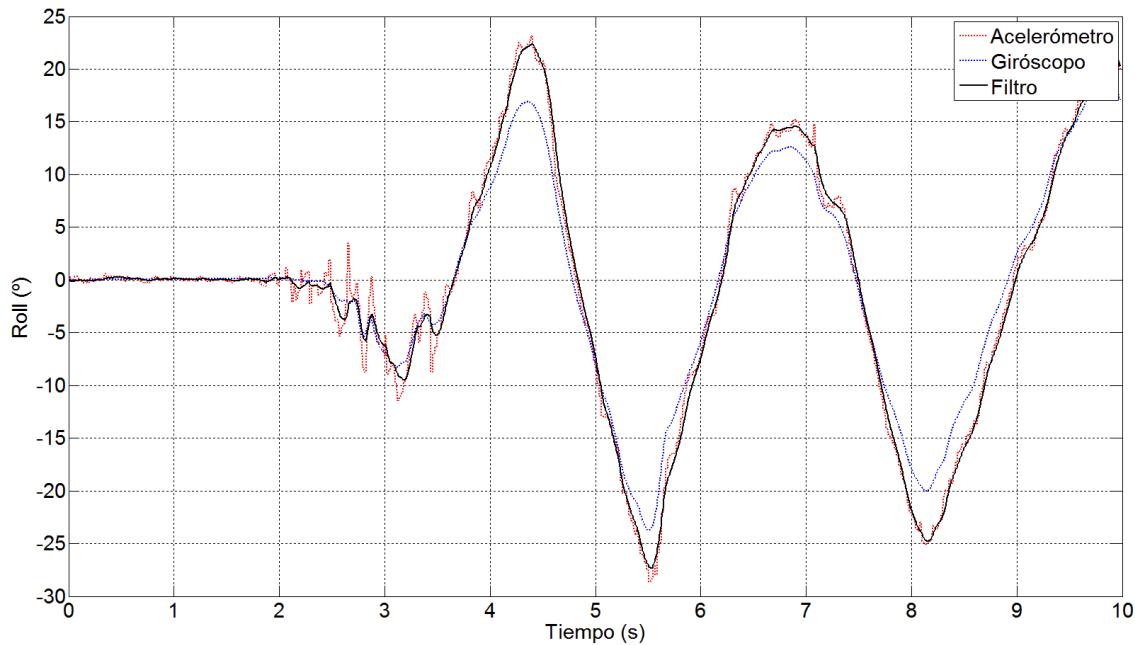
- Ángulo calculado a partir del acelerómetro (mismo programa que está en la placa).
- Ángulo integrado únicamente a partir de los datos del giróscopo. Esta función realmente no es implementada en la placa, pues es intrínseca al filtro Kalman como ya se vio en su apartado.
- Ángulo integrado y filtrado mediante el filtro Kalman (mismo programa que en la placa).

Con los datos obtenidos en el apartado de calibración de los sensores, los cuales nos daban los datos estadísticos a partir de los que se definen las constantes del filtro, podemos establecer una primera estimación de las parámetros:

- Q\_angle\_roll: varianza del *roll* del acelerómetro  $0,00060 \text{ rad}^2$
- Q\_angle\_pitch: varianza del *pitch* del acelerómetro  $0,00068 \text{ rad}^2$
- Q\_gyroBias\_roll: varianza de la velocidad angular en X del giróscopo:  $0,0004 (\text{rad/s})^2$
- Q\_gyroBias\_pitch: varianza de la velocidad angular en Y del giróscopo:  $0,0004 (\text{rad/s})^2$
- R\_angle\_roll: varianza de la diferencia entre los ángulos calculados de ambos sensores, una vez calibrados, para el *roll*:  $0,00063 \text{ rad}^2$ .
- R\_angle\_pitch: varianza de la diferencia entre los ángulos calculados de ambos sensores, una vez calibrados, para el *pitch*:  $0,00104 \text{ rad}^2$ .

Cabe recordar que estos datos han sido obtenidos mediante ensayos estáticos sobre el punto de equilibrio del cuadricóptero (*roll* y *pitch* nulos). Ahora hemos de comprobar si se comportan bien ante la dinámica del vehículo. Para ello pedimos datos a la placa, realizando breves movimientos sobre la misma en torno al punto de equilibrio durante el periodo de toma de medias, obteniendo el siguiente ajuste sobre los mismos:



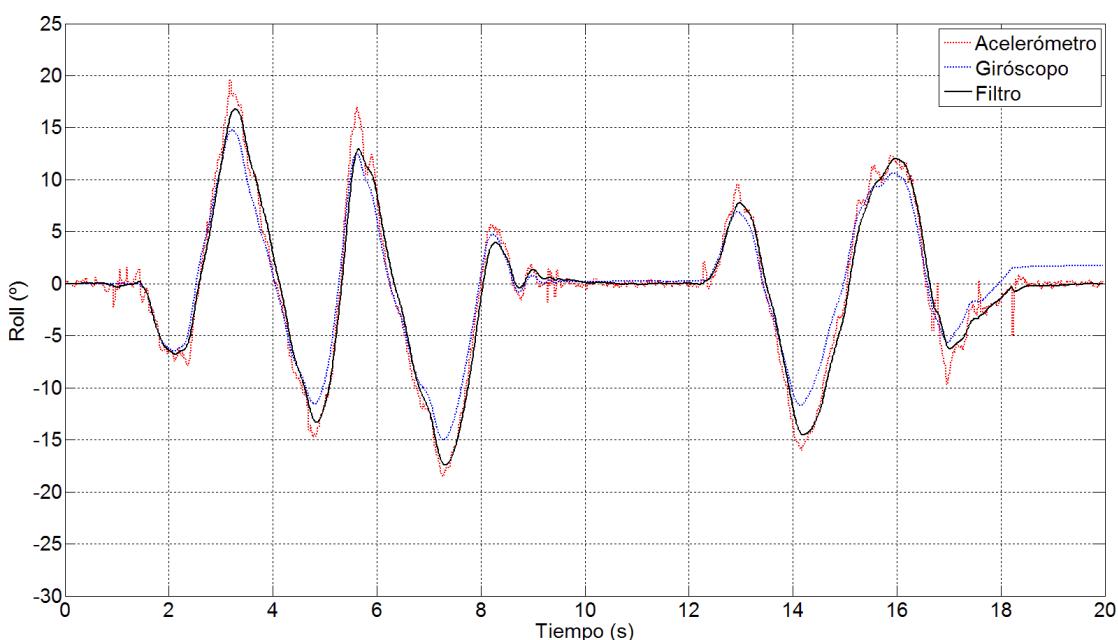


Como podemos observar, la señal negra se ajusta muy bien a los sensores, sin embargo presenta más ruido del que esperábamos, en especial en el caso del *pitch*.

Reajustando las constantes experimentales, realizando ensayos similares a los del apartado anterior para probar diferentes dinámicas del vehículo, llegamos a este sexteto de valores que se acopla bien a cualquier tendencia del vehículo.

Q_angle_roll	$0,001\text{rad}^2$	Q_gyroBias_roll	$0,003(\text{rad/s})^2$	R_angle_roll	$0,03\text{rad}^2$
Q_angle_pitch	$0,001\text{rad}^2$	Q_gyroBias_pitch	$0,003(\text{rad/s})^2$	R_angle_pitch	$0,03\text{rad}^2$

Podemos ver los resultados de estas constantes sobre unos datos cuya duración total es de 20 segundos:



Únicamente representamos el caso del *roll*, donde ya se observa muy bien que la salida del filtro con las nuevas constantes es una señal muy limpia, sin ruido y sin *bias*, aunque se mueva la placa casi 20°. Podemos ver al final como el giróscopo sí se ha desplazado.

## 3.6 Interrupciones

---

A lo largo de la rutina del microcontrolador empleamos dos interrupciones para llevar a cabo tareas fundamentales. Estas interrupciones son la generada para la transmisión USART y la interrupción generada por el temporizador 4, que comentamos a continuación:

### 3.6.1 Interrupción TIMER 4

El objetivo principal del TIMER 4 es ejecutar la rutina de lectura de ángulos, aplicar el filtro Kalman, el algoritmo de control y finalmente modificar el pulso de los motores. Para ello configuramos este temporizador en modo base de tiempos.

#### 3.6.1.1 Configuración TIMER 4

Comenzamos habilitando la señal de reloj para el TIMER4:

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);
```

A continuación escogemos el preescalar y el periodo del temporizador con la siguiente expresión:

$$\Delta T = \frac{(Periodo\ TIM4 + 1) \cdot (Prescalar\ TIM4 + 1)}{Frecuencia\ TIM4}$$

Como nuestro objetivo es conseguir una interrupción de  $\Delta T = 10ms$  y la *Frecuencia TIM4 = 72 MHz*, debemos conseguir un producto tal que  $(Periodo\ TIM4 + 1) \cdot (Prescalar\ TIM4 + 1) = 720000$ . Como siempre buscamos un prescalar bastante alto, decidimos fijar *Prescalar TIM4 = 7199*, de modo que *Periodo TIM4 = 99*. Asimismo, configuramos el modo de cuenta ascendente y activamos el temporizador:

```
TIM_TimeBaseStructure.TIM_Period = (Freq_T4*T_calibracion_ms)/7200-1;
TIM_TimeBaseStructure.TIM_Prescaler = 7199;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);
TIM_Cmd(TIM4, ENABLE);
```

Seguidamente seleccionamos el canal IRQ y establecemos un nivel de prioridad 1. Esto se debe a que esta interrupción en un principio saltaba en medio de la recepción y transmisión de la USART, provocando que esta comunicación se quedase bloqueada

y/o enviase datos erróneos<sup>21</sup>. Dándole a esta interrupción una prioridad más baja conseguimos que la interrupción de la USART se lleve a cabo sin problemas.

```
NVIC_InitStructure.NVIC_IRQChannel=TIM4_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority=0;
NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
NVIC_Init(&NVIC_InitStructure);
TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
```

No obstante, hay que prestar atención a este hecho, pues si se produce esta interrupción en un intervalo distinto de 10 ms la integración del giróscopo dará un valor erróneo<sup>22</sup>, por lo que es recomendable calibrar después de largas transmisiones de datos. En cualquier caso, como el envío de datos desde *MatLab* se suele hacer en reposo, aunque el intervalo de tiempo real no sea 10 ms no afecta al resultado, pues la velocidad angular sería nula.

Cuando salta esta interrupción y tras borrar el flag de interrupción, el microcontrolador ejecuta la rutina más importante de todo el código que se describe paso a paso en los siguientes puntos.

### 3.6.1.2 Lectura de los sensores

En primer lugar leemos el valor de los registros de acelerómetro y giróscopo. Como ambos nos proporcionan el valor en parte alta y parte baja y en complemento a 2 realizamos la conversión a un único valor con el signo correspondiente. Si la lectura ha dado error, mantenemos los valores de aceleración y velocidad angular del paso anterior. Si se supera un cierto número de fallos encendemos un LED para avisar de que algo falla:

```
if (L3GD20_Read(DirGir_Leer,6) == 0) // Leemos el valor del giróscopo y comprobamos
si se ha producido un error
{
    // Convertimos a un único valor con signo
    wx = -(int)(Lect_Gir[0] + (Lect_Gir[1] << 8));
    wy = (int)(Lect_Gir[2] + (Lect_Gir[3] << 8));
    wz = -(int)(Lect_Gir[4] + (Lect_Gir[5] << 8));
    // Contadores
    nGir = 1; // Reiniciamos contador de errores consecutivos
    GPIO_ResetBits(GPIOE, GPIO_Pin_12); // Apagamos LED
}
else// Ha dado error
{
    // No actualizamos valores de w, suponemos que se mantienen constantes: Otra
    opción sería omitir en esta iteración la aportación del giróscopo
    nGir++;
    if (nGir >= LIM_GIR_LECTURA) // Hemos alcanzado el límite
    {
```

---

<sup>21</sup> Si llega un dato a la USART mientras hay otro en el buffer de entrada y estamos en la interrupción del TIM4 (por lo que no leemos el dato), se activa el bit de *Overrun* que hace que se entre constantemente en la interrupción por transmisión de la USART.

<sup>22</sup> Integrar numéricamente implica conocer con exactitud el paso de tiempo de integración.

```

        GPIO_SetBits(GPIOE, GPIO_Pin_12); // Encendemos LED para avisar al
usuario
    }
}

// Leemos el acelerómetro
if (Lee_Acc(6) == 0) // Leemos el valor del acelerómetro y comprobamos si se ha
producido un error
{
    // Convertimos a un único valor con signo
    ay = (int)(Lect_Acc[0] + (Lect_Acc[1] << 8));
    ax = (int)(Lect_Acc[2] + (Lect_Acc[3] << 8));
    az = -(int)(Lect_Acc[4] + (Lect_Acc[5] << 8));
    // Contadores
    nAcc = 1; // Reiniciamos contador de errores consecutivos
    GPIO_ResetBits(GPIOE, GPIO_Pin_14); // Apagamos LED
}
else// Ha dado error
{
    // No actualizamos valores de a, suponemos que se mantienen constantes: Otra
opción sería omitir en esta iteración la aportación del acelerómetro
    nAcc++;
    if (nAcc >= LIM_ACC_LECTURA) // Hemos alcanzado el límite
    {
        GPIO_SetBits(GPIOE, GPIO_Pin_14); // Encendemos LED para avisar al
usuario
    }
}

```

Una vez tenemos los valores de aceleración lineal y velocidad angular en el sistema de referencia global escogido pasamos a obtener el roll y el pitch a partir del acelerómetro:

```

roll_Acc = atan(ax/sqrt(ay*ay+az*az));
pitch_Acc = atan(ay/sqrt(ax*ax+az*az));

```

Si hemos de almacenar datos para la calibración (calibrar = 1) utilizamos este valor para obtener el *offset*. En caso contrario, le restamos este *offset* para obtener el valor definitivo y calculamos el ratio de giro del giróscopo:

```

pitch_Acc -= pitch_Acc0;
roll_Acc -= roll_Acc0;
ratio_GirX = (wx - wx0)*w2rps;
ratio_GirY = (wy - wy0)*w2rps;
ratio_GirZ = (wz - wz0)*w2rps;

```

A continuación aplicamos el código del filtro Kalman explicado en su correspondiente apartado para obtener la inclinación definitiva del *Electrocóptero*. No obstante, puesto que necesitamos un valor previo para el giróscopo, la primera iteración del filtro se corresponde a lo siguiente, ya que no hay *bias*:

```

if (cont == 0)
{
    rate0_Qx = ratio_GirX;
    rate0_Qy = -ratio_GirY;
}

```

Una vez aplicado el filtro incrementamos el contador de muestras (`cont++`) y pasamos a encender los LEDs “burbuja”:

### 3.6.1.3 LEDs “burbuja”

Estos LEDs son los correspondientes a las orientaciones N, S, E, W de la tabla del [apartado 3.1.3](#). La función de encendido o apagado de cada uno de ellos depende del *roll* y el *pitch*, aplicando el siguiente código:

```
if (roll > 0)
{
    GPIO_SetBits(GPIOE, GPIO_Pin_15);
    GPIO_ResetBits(GPIOE, GPIO_Pin_11);
}
else
{
    GPIO_SetBits(GPIOE, GPIO_Pin_11);
    GPIO_ResetBits(GPIOE, GPIO_Pin_15);
}
if (pitch > 0)
{
    GPIO_SetBits(GPIOE, GPIO_Pin_9);
    GPIO_ResetBits(GPIOE, GPIO_Pin_13);
}
else
{
    GPIO_SetBits(GPIOE, GPIO_Pin_13);
    GPIO_ResetBits(GPIOE, GPIO_Pin_9);
}
```

De este modo, podemos conocer desde fuera si las variables *pitch* y *roll* se comportan coherentemente, funcionando igual que un nivel burbuja. Además se puede concluir de los LEDs si la placa se ha bloqueado en algún punto, o sigue entrando en la interrupción del Temporizador 4.

### 3.6.1.4 Función de control

Una vez hemos actualizado la posición de los LEDs pasamos a aplicar la función de control que describimos en el [apartado 3.9](#).

### 3.6.1.5 Calibración de la placa

Toda la secuencia anterior desde que aplicamos el filtro Kalman en adelante tiene lugar si no hemos de calibrar la placa. Como ya hemos indicado en el [apartado 3.4.6](#), esto es importante, tanto por el hecho de que el acelerómetro no está perfectamente horizontal encima del electrocóptero, como porque el giróscopo puede indicar cierta rotación cuando realmente no la hay, como veímos en las gráficas de ese apartado.

Por este motivo, cuando deseamos calibrar debemos poner a cero ciertos parámetros y almacenar datos de los sensores un cierto número de veces para promediar posteriormente. Para ello, llamamos a la función `config_calibrar`, que realiza los siguientes pasos:

- 1) Apagamos los LEDs burbuja:

```
GPIO_ResetBits(GPIOE, GPIO_Pin_9);
GPIO_ResetBits(GPIOE, GPIO_Pin_11);
GPIO_ResetBits(GPIOE, GPIO_Pin_13);
GPIO_ResetBits(GPIOE, GPIO_Pin_15);
```

- 2) Ponemos a 0 los parámetros del filtro Kalman, así como las velocidades angulares y la señal del acelerómetro:

```
bias_Qx = 0;                                P_Qy[1][1] = 0;
bias_Qy = 0;                                K_Qx[0] = 0;
roll = 0;                                    K_Qx[1] = 0;
pitch = 0;                                   K_Qy[0] = 0;
Qxx = 0;                                     K_Qy[1] = 0;
Qyy = 0;                                     wx0 = 0;
P_Qx[0][0] = 0;                             wy0 = 0;
P_Qx[0][1] = 0;                             wz0 = 0;
P_Qx[1][0] = 0;                             roll_Acc0 = 0;
P_Qx[1][1] = 0;                             pitch_Acc0 = 0;
P_Qy[0][0] = 0;
P_Qy[0][1] = 0;
P_Qy[1][0] = 0;
cont = 0;
```

- 3) Modificamos el periodo del TIMER 4 para que la calibración se lleve a cabo con un periodo de 5 ms y deshabilitamos la interrupción por recepción de datos de la USART:

```
TIM_SetAutoreload(TIM4,(Freq_T4*T_calibracion_ms)/7200-1);
USART_ITConfig(USART3, USART_IT_RXNE, DISABLE);
```

A esta función se accede al iniciar la placa, al enviar el STX2 desde MatLab o al pulsar el botón azul de usuario.

Cuando salta la interrupción del TIMER 4 descrita en el apartado anterior, como ahora `calibrar = 1`, realizaremos el siguiente código:

```
if (calibrar == 1) // Si hay que calibrar no calculamos nada, buscamos el offset
```

- 1) Encendemos el LED azul que nos indica que se está calibrando:

```
GPIO_SetBits(GPIOE, GPIO_Pin_8);
```

- 2) Vamos acumulando los datos de las variables de velocidades angulares y aceleraciones, e incrementamos el contador:

```
wx0 = wx0 + wx;
wy0 = wy0 + wy;
wz0 = wz0 + wz;
roll_Acc0 = roll_Acc0 + roll_Acc;
pitch_Acc0 = pitch_Acc0 + pitch_Acc;
cont++;
```

- 3) Cuando hemos entrado en la interrupción `NumCalibrar`, ya que hemos acabado de calibrar, ponemos el contador a 0 y apagamos el LED azul, puesto que ya no está calibrando:

```
calibrar = 0;
cont = 0;
GPIO_ResetBits(GPIOE, GPIO_Pin_8);
```

- 4) Despues, realizamos el promedio y devolvemos al TIMER 4 el período de 10 ms, volviendo a habilitar la recepción de la USART:

```
wx0 /= NumCalibrar;
wy0 /= NumCalibrar;
wz0 /= NumCalibrar;
roll_Acc0 /= NumCalibrar;
pitch_Acc0 /= NumCalibrar;
TIM_SetAutoreload(TIM4,(Freq_T4*T_ms)/7200-1);
USART_ITConfig(USART3, USART_IT_RXNE, ENABLE);
```

### *3.6.1.6 Resumen de la interrupción del Temporizador 4*

Podemos resumir las diferentes tareas que realiza el Temporizador 4 en la siguiente tabla de líneas:

Líneas	Función
<b>0879</b>	Borrado del flag de interrupción
<b>0881 - 0900</b>	Lectura del giróscopo
<b>0902 - 0921</b>	Lectura del acelerómetro
<b>0924 - 0926</b>	Obtención de los ángulos brutos a partir de los datos del acelerómetro
<b>0929 - 0953</b>	Calibrar
<b>0958 - 1020</b>	Almacenar datos para enviar por la USART
<b>1023 - 1025</b>	Obtención de los ángulos calibrados del acelerómetro
<b>1028 - 1031</b>	Obtención de las velocidad angulares calibradas del giróscopo
<b>1033 - 1094</b>	Filtro Kalman
<b>1097 - 1118</b>	LEDs burbuja
<b>1121 - 1221</b>	Controlador PID

### *3.6.2 Interrupción de la USART3*

La otra interrupción de nuestro programa es la que se da relacionada con la comunicación inalámbrica del PC. No entraremos en mucho detalle en la implementación del código, pues es un algoritmo muy sencillo, pero con muchas condiciones y variables de control que no aportan ningún sentido, salvo la gestión de flujos del programa.

Lo importante de esta sección es la función que realiza la placa para cada comando enviado desde *MatLab*, que ya ha sido explicado en el [apartado correspondiente](#).

Aquí lo único que nombraremos será que la interrupción de la USART se divide en dos partes, la de recepción y la de transmisión. En la de transmisión solo se entrará cuando se vaya a enviar más de un dato, si no, ni siquiera se habilitará. Recordemos que es muy importante desactivarla cuando ya no vayamos a transmitir más información.

Por su parte, la de recepción se activa cada vez que llega un byte al *buffer* del controlador, es decir, que leemos byte por byte.

### 3.7 Motores

---

Los motores que componen nuestra estructura son cuatro motores *Brushless* que van acompañados cada uno de ellos con su respectivo variador *ESC* (*Electronic Speed Controller*). Como hemos visto con la práctica, realmente cada uno de nuestros motores consume alrededor de unos 6A, y no los 4A que en un primer momento estimábamos. Esta es la razón por la cual la batería dura poco más de diez minutos.

A continuación vemos la configuración que presentan ESC + Motor *Brushless*:



En primer lugar trataremos de conocer cómo es el motor y a qué rangos trabaja, determinaremos el comportamiento en particular de cada uno de ellos, para poder ver así si podemos configurarlos en torno a un punto de equilibrio. Para ello debemos caracterizar los motores.

Una vez estén caracterizados podremos aplicar una función de control a cada uno de ellos, para lo que necesitaremos saber cómo enviamos la señal de los motores desde la placa a cada ESC.

Cabe tener en cuenta que no todos los motores giran en el mismo sentido, tal y como describíamos en la [base teórica](#) del proyecto. A tal efecto, conectamos los cables del ESC al motor en un orden u otro, consiguiendo que los motores giren en el sentido deseado. Así, el motor 1 y el 4 (opuestos entre sí) giran en sentido horario, mientras que el 2 y el 3 lo hacen en el sentido contrario de las agujas del reloj.

### 3.7.1 Linealización de los motores

A parte de tomar unas buenas medidas de los ángulos, para el sistema de control el otro aspecto fundamental es conocer la fuerza que nos proporciona el motor en función del pulso que le damos al ESC.

En el código del microcontrolador disponemos de un vector de 4 elementos, llamado *vel*, que contiene exactamente el pulso que le estamos dando al motor. Para que esta variable pueda adquirir unidades de fuerza es necesario realizar un ensayo para caracterizar cada uno de los 4 motores.

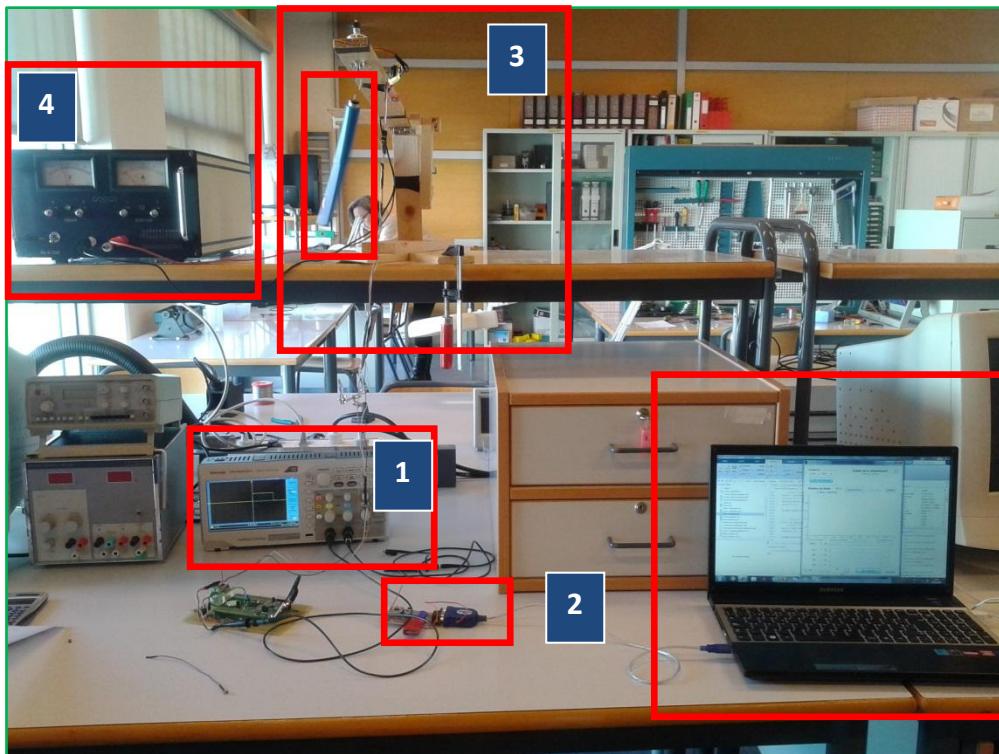
Para tal fin, y sabiendo que los motores nos deben dar en torno a 2,5 N cada uno, realizamos 4 ensayos, uno con cada motor de forma independiente durante los cuales aprovechamos la GUI de MatLab que nos permite controlar los motores de forma manual. Como resultado final obtendremos una gráfica en la cual podemos representar la fuerza de los motores en función del pulso proporcionado, para así poder modelizar el comportamiento.

#### *3.7.1.1 Equipo Empleado*

- Palanca de madera.
- Dinamómetro.
- Motor + hélice + ESC.
- Fuente de alimentación.
- Sondas y osciloscopio.
- Placa *Discovery*.
- Módulo inalámbrico.
- Gatos mecánicos.
- Portátil con *MatLab*.

#### *3.7.1.2 Montaje del Ensayo*

El esquema global del ensayo se puede mostrar en una imagen de la siguiente forma, indicando más abajo que significan cada uno de los números de montaje y comprobando que se utilizan los equipos anotados más arriba:



A pesar de que parezca muy complejo el montaje, se puede dividir en 4 partes claramente diferenciadas:

### **Montaje 1**

Conectamos la placa al osciloscopio para ver qué pulso verdaderamente se está transmitiendo por el pin que lleva a la señal de control del ESC.

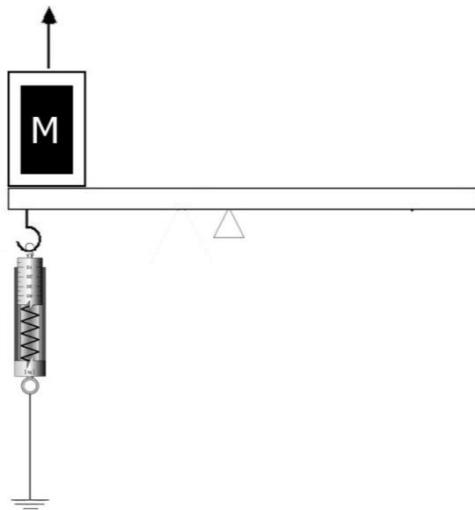
### **Montaje 2**

Comunicamos la placa con *MatLab*, lo cual nos permite conocer de forma más rápida y directa el pulso que da el motor en comparación a la medición en el osciloscopio. Esta comunicación nos permite además enviar los comandos de aumentar y disminuir el pulso al controlador. Ello es posible gracias a la conexión del módulo inalámbrico.

### **Montaje 3**

Empleando una palanca apoyada sobre su centro, situamos el motor sobre uno de sus extremos por encima del tablón de madera horizontal que forma el mecanismo. Por debajo enganchamos el dinamómetro, de escala graduada 0,01 N, de tal modo que mida el empuje vertical que está proporcionando el motor.

Podemos esquematizar el montaje de la siguiente forma:

**Montaje 4**

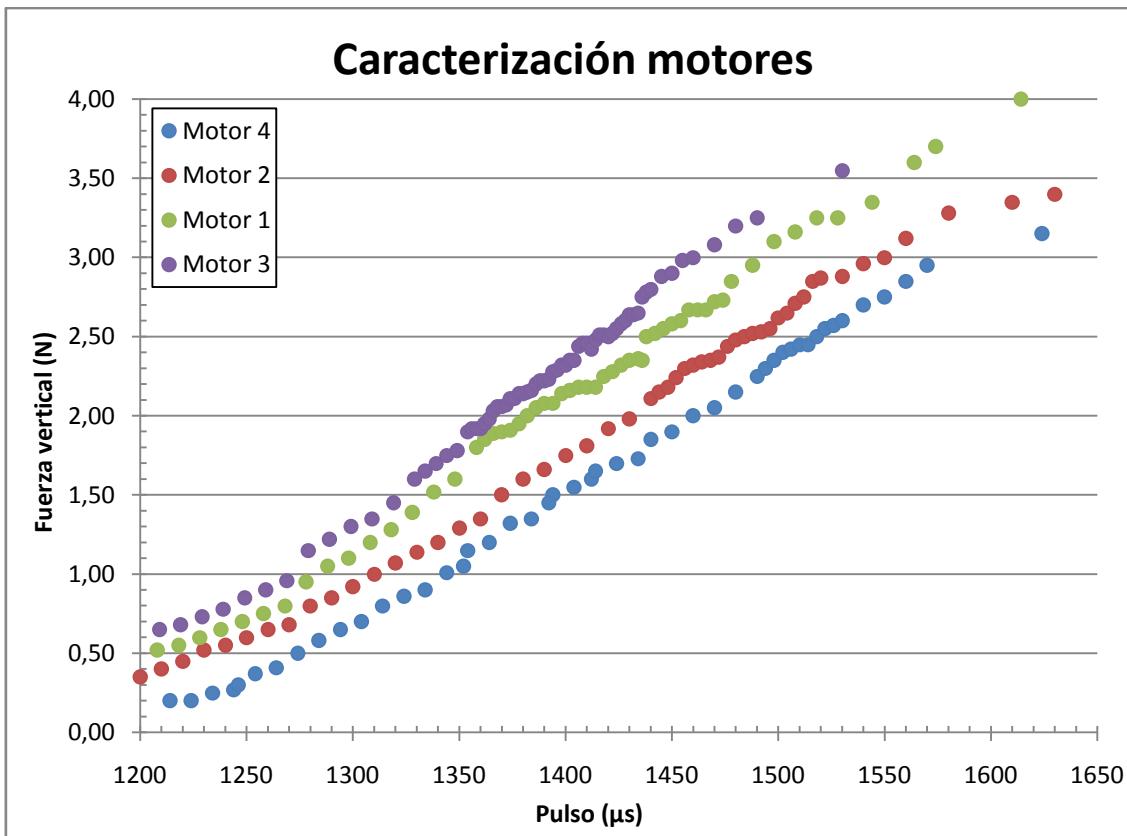
Alimentamos la placa conectándola a la fuente de alimentación a una tensión de 11,4 V, mediante un cable banana-cocodrilo rojo y otro negro.

***3.7.1.3 Realización del Ensayo***

- 1) Iniciamos la comunicación inalámbrica con el PC y arrancamos el motor proporcionando un pulso de 1000  $\mu$ s.
- 2) Subimos el pulso hasta el punto en que el motor se equilibra solo en el aire, y anotamos este valor.
- 3) Enganchamos el dinamómetro y volvemos a buscar el punto de equilibrio con el medidor colgando.
- 4) Subimos el pulso de 10 en 10  $\mu$ s y vamos anotando la medida de fuerza para cada paso.
- 5) Al acercarnos a la zona buscada, en torno a 2 N, y hasta haberla pasado en un margen prudencial (hasta 3 N aproximadamente), tomamos medidas únicamente distanciadas 2  $\mu$ s para tener más precisión en la zona donde buscamos modelizar.
- 6) Finalmente aumentamos el pulso abruptamente hasta ver qué fuerza pueden darnos los motores, y el pulso máximo al que deberíamos llegar.
- 7) Desconectamos el motor.

***3.7.1.4 Resultados Obtenidos***

Con las medidas tomadas de los 4 motores, podemos representar la siguiente gráfica:



Lo primero que nos llama la atención es que la tendencia de las 4 curvas es análoga. Por suerte, en la zona de equilibrio todas siguen una tendencia bastante lineal. Por su parte, se puede observar la presencia de una zona cóncava para pulsos bajos y una zona convexa para los altos, aunque esta parte va a ser irrelevante para el control y, por tanto, para la modelización.

Sin embargo, las curvas no colapsan, lo cual implica que cada motor proporciona fuerzas distintas para el mismo pulso. Como son más o menos paralelas, este *offset* puede deberse a errores sistemáticos y por tanto corregibles posteriormente.

Por su parte, si queremos buscar unos límites operativos para el pulso que le demos a los motores desde el control, observando la gráfica vemos que un pulso mínimo de funcionamiento podría ser de 1300  $\mu$ s y un máximo operativo de 1600  $\mu$ s, estando la zona de modelización buscada entre 1400 y 1500  $\mu$ s.

### 3.7.1.5 Modelización

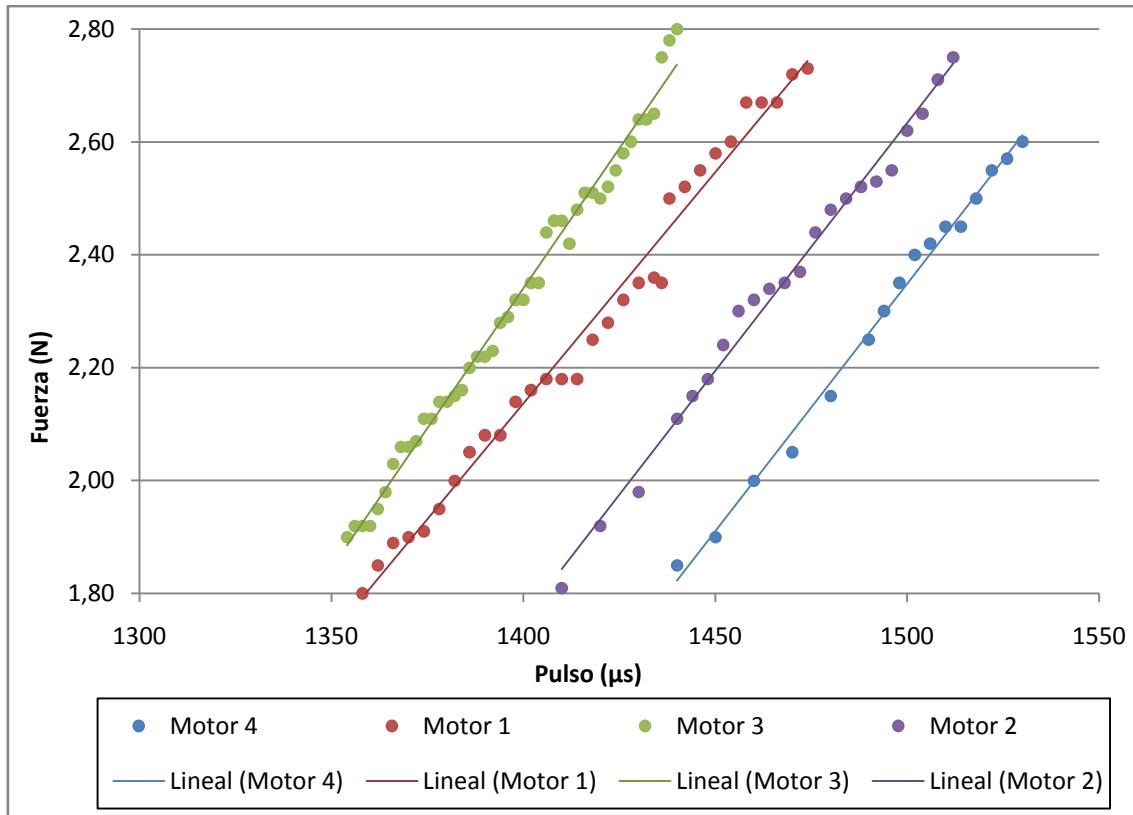
El objetivo de esta experiencia es obtener un modelo experimental que nos relacione el pulso de los motores con la fuerza que son capaces de generar, en torno a un punto de equilibrio de 2,5 N. Como se desprende de las curvas vistas en el apartado anterior, un buen modelo es un modelo lineal en esa área, y por tanto procedemos a linealizar los motores buscando una expresión analítica de la forma:

$$F_i = b_i \cdot p_i + a_i$$

Donde los diferentes parámetros son:

- $F_i$  es la fuerza que nos da el motor en concreto.
- $b_i$  es la tasa de variación de la fuerza por cada  $\mu\text{s}$  de pulso que aumentamos.
- $p_i$  es el pulso que le damos a cada motor.
- $a_i$  es la fuerza proporcionada por el motor cuando no hay pulso. En teoría debería ser cero.

Si tomamos únicamente valores próximos a 2,5N, donde además tenemos medidas más próximas entre sí, podemos representar la siguiente gráfica:



Agregando una línea de tendencia lineal y aplicando el método de mínimos cuadrados, obtenemos los siguientes valores que caracterizan los motores:

Motor	Pendiente (N/ $\mu\text{s}$ )	Ordenada en origen (N)	$\rho^2$
1	0,008189	-9,32803642	0,985
2	0,008774	-10,5273903	0,990
3	0,009906	-11,5274334	0,989
4	0,008762	-10,7946273	0,994

Vemos como todos los coeficiente de correlación están en torno a 0,99, por lo que el ajuste lineal es bastante correcto en torno a la zona de equilibrio.

En cuanto a la ordenada en origen, ésta es poco representativa, pues buscaremos un mejor ajuste en una experiencia posterior. Como ya hemos dicho, pensamos que el

offset de estas curvas puede no ser correcto debido a errores en el ensayo y por tanto descartamos la validez de éste parámetro.

La pendiente de las curvas sí es un dato significativo, y es la conclusión principal que vamos a extraer de este ensayo, pues nos proporciona información de cuánto aumenta la fuerza cuando aumentamos el pulso, es decir, el objetivo de esta experiencia.

$$b_1 = 0,008189 \frac{N}{\mu s}; \quad b_2 = 0,008774 \frac{N}{\mu s}$$

$$b_3 = 0,009906 \frac{N}{\mu s}; \quad b_4 = 0,008762 \frac{N}{\mu s}$$

Como vemos, todos los motores nos dan más o menos la misma tasa. El motor 3 se destaca sobre el resto, quizás debido a que va asignado con una hélice distinta a las demás ya que la tuvimos que sustituir al dañarse la anterior durante un ensayo.

Cabe decir que todos los datos obtenidos durante la experiencia no sólo corresponden a un motor en concreto, sino que han sido asignados al trío motor-hélice-ESC para evitar discrepancias entre cualquiera de estos dispositivos y así minimizar errores.

### 3.7.1.6 Fuentes de Error

Los distintos errores que podemos encontrar en la realización del ensayo y que pueden afectar, directa o indirectamente, a los resultados obtenidos son los siguientes:

- Es posible que durante el montaje del motor éste no se situara de forma completamente vertical.
- Despreciamos el peso del tablón horizontal de la balanza.
- Suponemos la histéresis despreciable y que el motor presenta la misma tendencia al aumentar el pulso y al disminuirlo.
- Errores visuales a la hora de leer el dinamómetro, aumentados además por la vibración transmitida desde el motor.
- Discontinuidades en el rendimiento de los motores los cuales no presentaban un régimen de funcionamiento estable para un pulso dado. Esto fue especialmente representativo en el motor 1.
- A la hora de caracterizar, el motor 3 empleaba una hélice más grande respecto a la que está equipada con los otros tres motores. Posteriormente colocamos 4 hélices iguales, por lo que habría que repetir la caracterización. Sin embargo, debido a la falta de tiempo, decidimos mantener los datos obtenidos anteriormente, siendo conscientes del error que estamos cometiendo.

### 3.7.2 Control de los Motores

El control de los motores *Brushless* se realiza mediante los *ESC*. En función del pulso que recibe el *ESC* el motor gira a más o menos revoluciones, generando mayor o menor sustentación de las hélices. Para generar las señales *PWM* (*Pulse Width Modulation*)

utilizamos los cuatro canales del TIMER 3. Para ello lo configuramos de la siguiente forma:

En primer lugar inicializamos la base de tiempos y la configuramos, configurando en modo de cuenta ascendente y un preescalar de 71 y un periodo tal que el periodo completo de la señal es de 20 ms<sup>23</sup>:

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0x0000;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
TIM_TimeBaseStructure.TIM_Prescaler = 71;
TIM_TimeBaseStructure.TIM_Period = T_MOTOR_ms*1000 - 1;
TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
```

A continuación inicializamos el modo de funcionamiento *PWM*, incorporando a cada señal un ancho de pulso de 870 µs (*VelMin\_us*), que se corresponde con el pulso de motor parado:

```
TIM_OCStructInit(&TIM_OCInitStruct);
TIM_OCInitStruct.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStruct.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStruct.TIM_Pulse = VelMin_us;
TIM_OC1Init(TIM3, &TIM_OCInitStruct);
TIM_OC2Init(TIM3, &TIM_OCInitStruct);
TIM_OC3Init(TIM3, &TIM_OCInitStruct);
TIM_OC4Init(TIM3, &TIM_OCInitStruct);
TIM_Cmd(TIM3, ENABLE);
```

Cuando queremos modificar el ancho de pulso de esta señal, ya sea manualmente o por medio del *PID*, utilizamos la función *TIM\_SetCompare<sub>i</sub>*:

Por ejemplo, si queremos poner un ancho de pulso de 1300 al motor 1, escribiremos *TIM\_SetCompare1(TIM3,1300)*.

### *3.8 Dispositivo sonar*

---

Como ya hemos comentado al comienzo de este documento, uno de los dispositivos externos que hemos incorporado en nuestro proyecto ha sido el sónar, que es simplemente un equipo empleado para generar y recibir el sonido de carácter infrasonoro, que nos convierte en distancia.

Como es obvio, en cualquier dispositivo que se encuentre en sustentación en el aire, como es el electrocóptero, es imprescindible conocer la distancia que lo separa de la superficie, ya sea agua o tierra.

En nuestro caso, a pesar de que podemos controlar el roll y pitch con el acelerómetro y el giróscopo, no hay nada que nos indique si el electrocóptero está subiendo, bajando, o su altura es constante. Por ello, y teniendo en cuenta que el funcionamiento de este elemento no es complejo desde el punto de vista externo, consideramos conveniente

---

<sup>23</sup> Se ha aplicado la misma fórmula que en el [apartado 3.6.1](#).

incorporar un sonar, concretamente, el modelo HC-SR04. A continuación se describe su funcionamiento así como varios aspectos a tener en cuenta en su implementación con el microcontrolador de nuestro proyecto.

Por el puerto PB9 generamos una señal PWM de periodo 20 ms y ancho de pulso (duty) igual a 20 µs utilizando el canal 1 del TIMER 17, que configuramos de manera similar que los canales del TIMER 3. Cada vez que el sónar recibe el pulso alto de esta señal envía una onda ultrasónica por el emisor que es recibida más tarde por el receptor cuando esta rebota en cualquier superficie. En el tiempo comprendido entre la emisión y la recepción, el sonar pone a nivel alto su otra línea. Este tiempo es directamente proporcional a la distancia recorrida por la onda, pudiendo obtener la siguiente expresión sabiendo la velocidad del sonido (344 m/s):

$$\text{distancia (cm)} = \frac{\text{tiempo}(\mu\text{s})}{58}$$

Para calcular este tiempo se utiliza el modo captura del canal 1 del TIMER 2 de nuestro microcontrolador. Dado que el sonar trabaja a 5 V, es necesario encontrar un puerto de la placa que admita tensiones de entrada de esta magnitud, como el PA15 y que incorpore, a su vez, el canal 1 del TIMER 2.

Internamente, es necesario configurar el modo captura con las siguientes consideraciones:

- Periodo suficiente para poder capturar el intervalo máximo:  
`TIM_TimeBaseStructure.TIM_Period = 0xFFFF`
- Prescalar para que cada cuenta equivalga a 1 µs. Como la frecuencia es 72 MHz, seleccionamos un prescalar de 71:  
`TIM_TimeBaseStructure.TIM_Prescaler = 71`
- Captura por flanco de bajada:  
`TIM_ICInitStruct.TIM_ICPolarity=TIM_ICPolarity_Falling`
- TIMER en modo esclavo con reinicio provocado por ambos flancos:  
`TIM_SelectSlaveMode(TIM2,TIM_SlaveMode_Reset)`  
`TIM_SelectInputTrigger(TIM2,TIM_TS_TI1F_ED)`

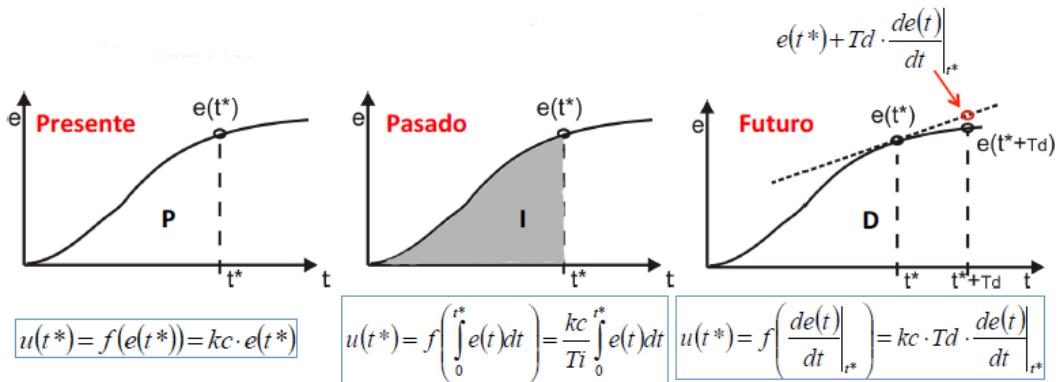
De este modo, con leer el registro de captura y dividirlo entre 58, obtenemos la distancia entre el cuadricóptero y el suelo.

### 3.9 Sistema de control

---

Como ya hemos mencionado anteriormente, el control de la estabilidad del electrocóptero es gestionado en nuestro proyecto por medio de un *PID*. Este tipo de controlador se basa en aplicar tres acciones distintas, una proporcional, una integral y una derivativa:

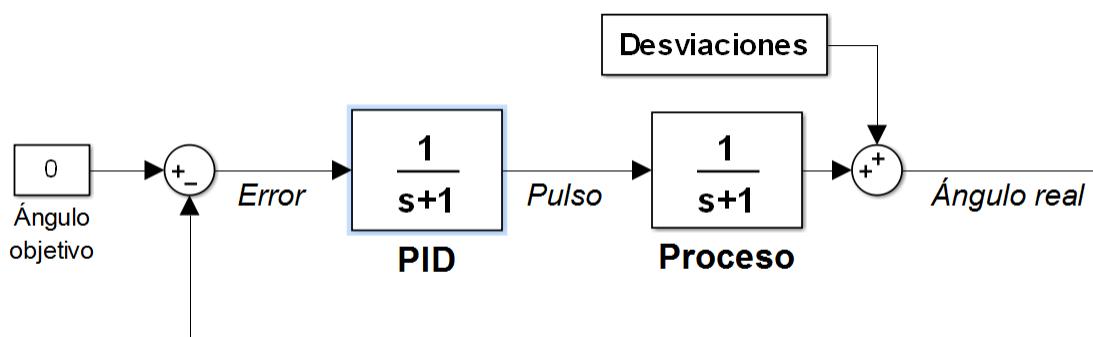
- **P:** Proporcional. Es instantánea, depende del presente. Amplifica el error.
- **I:** Integral. Depende del pasado. Acumula el error anterior.
- **D:** Derivada. Predice el error futuro.



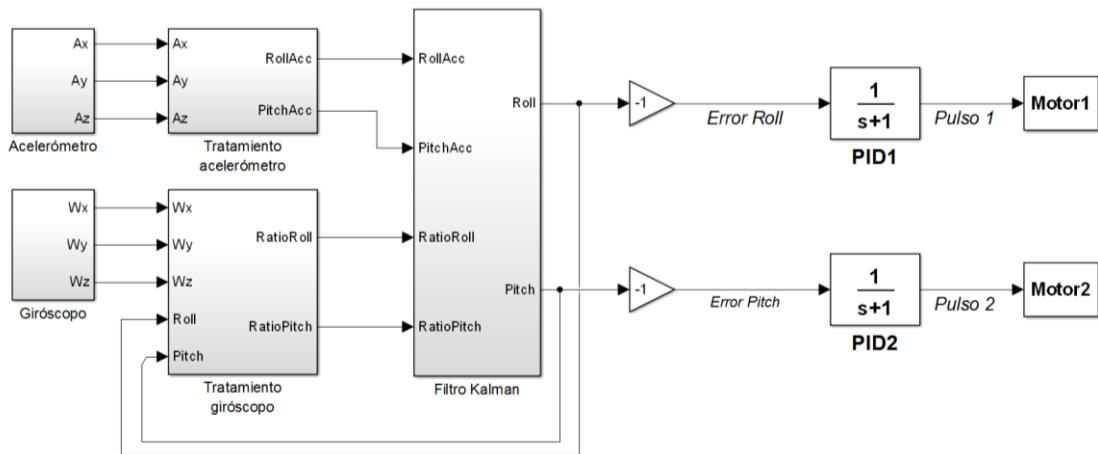
En primer lugar implementaremos en la placa estas tres acciones y más adelante trataremos de hallar la constante propia de cada acción, ya sea teóricamente o experimentalmente.

Aunque en principio nos planteamos integrar un PID para cada motor, finalmente consideramos más conveniente fijar el pulso de dos motores (uno de cada eje) e incorporar un controlador en los dos restantes. De este modo simplificamos el proceso, pues estamos reduciendo el número de constantes a ajustar de 12 a tan solo 6 (3 por eje), aunque no estaríamos controlando completamente la altura del sistema, pues en cada ocasión se generará más o menos sustentación. Esta situación se puede mejorar leyendo el sonar y aumentando o disminuyendo el pulso de los motores sin PID en función de la lectura.

Nuestro principal objetivo es conseguir que el ángulo que miden los sensores sea lo más próximo a  $0^\circ$ . No obstante, es prácticamente imposible conseguir un pulso que, aplicado continuamente, garantice la estabilidad, debido a las posibles perturbaciones externas y a nuestro modelo. Para disminuir este efecto, medimos continuamente el ángulo real que obtenemos del filtro Kalman. A partir de él aplicamos la función de control para el motor en cuestión y obtenemos un nuevo pulso, que afectará al proceso como podemos ver a continuación:



Integrando ambos controladores el esquema final de control es el siguiente:



### 3.9.1 Implementación

En la rutina del TIMER 4 explicada anteriormente hay una parte reservada al PID que se ejecuta si activarPID ==1, lo cual sucede únicamente cuando desde la interfaz gráfica ordenamos ascender, mantener altura o descender:

En primer lugar comenzamos actualizando el error que le llega a cada PID, que se corresponde, como veímos en los anteriores diagramas, con el *roll* y el *pitch*, cuyo valor es positivo debido al sistema de referencia utilizado y los motores sobre los que actuamos. Realmente el valor a considerar sería: 0 (ángulo objetivo) – (–*roll*) = *roll*.

```
error_PID[0] = roll;
error_PID[1] = pitch;
```

Cada acción del control generará un pulso que trate de devolver el sistema a la posición deseada de equilibrio, ya sea multiplicando directamente por el error (parte P), integrando por el método de los trapecios (parte I) o derivando (parte D):

```
// Parte Proporcional
pulso_P[0] = Kc[0]*error_PID[0];
pulso_P[1] = Kc[1]*error_PID[1];
// Parte Integral
pulso_I[0] += Ti[0]*(error_PID[0] + error0_PID[0])/2*Ts;
pulso_I[1] += Ti[1]*(error_PID[1] + error0_PID[1])/2*Ts;
// Parte Derivada24
pulso_D[0] = Td[0]*rate0_Qx;
pulso_D[1] = Td[1]*rate0_Qy;
```

Cada parte va ponderada por una constante, que son las que deberemos ajustar para el correcto funcionamiento del sistema de control. La parte proporcional va multiplicada por  $K_c$ , la integral por  $T_i$  y la derivada por  $T_d$ .

<sup>24</sup> Ver apartado 3.9.1.1 Filtro acción derivativa

Seguidamente sumamos la acción de cada parte y comprobamos si se encuentra en un rango coherente, ya que se podría producir una variación brusca que no nos interese. Si se excede el máximo o se baja por debajo del mínimo, es decir, si se *satura*<sup>25</sup>, damos el valor máximo o el mínimo al pulso del PID y a la parte integral de damos el valor que tenía en el paso anterior, para que no siga ni subiendo ni bajando.

```
for (i=0;i<2;i++)
{
    pulso_PID[i] = pulso_P[i] + pulso_I[i] + pulso_D[i];
    if (pulso_PID[i] > pulso_max_PID)
    {
        pulso_PID[i] = pulso_max_PID;
        pulso_I[i] = pulso_I0[i];
    }
    elseif (pulso_PID[i] < pulso_min_PID)
    {
        pulso_PID[i] = pulso_min_PID;
        pulso_I[i] = pulso_I0[i];
    }
}
```

Este pulso realmente se trata de un incremento tomado a partir del pulso de equilibrio (*pulso0*), que se define al elegir el modo de funcionamiento (ascender, mantener altura o descender). Por lo tanto:

```
vel[0] = pulso0[0] + pulso_PID[0];
vel[1] = pulso0[1] + pulso_PID[1];
```

Además es necesario actualizar el error para poder reutilizarlo posteriormente en el siguiente paso, para la acción integral:

```
error0_PID[0] = error_PID[0];
error0_PID[1] = error_PID[1];
pulso_I0[0] = pulso_I[0];
pulso_I0[1] = pulso_I[1];
```

Finalmente actualizamos el pulso de los motores e incrementamos una unidad el contador:

```
TIM_SetCompare1(TIM3,vel[0]);
TIM_SetCompare2(TIM3,vel[1]);
contador_PID++;
```

Cuando el contador llega al valor 23, enviamos el pulso del motor 1 y realizamos lo propio para el motor 2 al llegar a 24. Para sincronizar el proceso, cuando el contador toma el valor 25 enviamos STX10 para que así *MatLab* interprete correctamente los datos:

```
if (contador_PID == 23)          //Enviamos pulso del motor 1
{
    pulso_MatLab = pulso_PID[0] - pulso_min_PID;
    USART_SendData(USART3,pulso_MatLab);
}
```

---

<sup>25</sup> Los límites de saturación vienen dados por ±100 µs sobre el punto de equilibrio.

```

elseif (contador_PID == 24)      //Enviamos pulso del motor 2
{
    pulso_MatLab = pulso_PID[1] - pulso_min_PID;
    USART_SendData(USART3,pulso_MatLab);
}
elseif (contador_PID == 25)      //Sincronizamos con MatLab por si hay error
{
    USART_SendData(USART3,STX10);
    contador_PID = 0;
}

```

Cabe tener en cuenta que, como ya hemos comentado en el apartado correspondiente, la comunicación inalámbrica no es bidireccional, de modo que al enviar datos no podemos recibir. Por tanto, si estamos enviando el pulso de los motores a *MatLab*, puede que la orden que queremos enviar a la placa no se envíe correctamente, pues ha interferido con el dato que va en sentido contrario, haciendo que tengamos que volver a probar suerte. Para reducir la probabilidad de que se dé este caso enviamos los tres datos seguidos, de modo que solo habría problemas en el 12% de las ocasiones. Al no haber observado experimentalmente ningún problema, hemos mantenido esta probabilidad, pero podría reducirse haciendo que se envíen datos con menor frecuencia (por defecto está a 4 tramas de 3 bytes por segundo).

Por otra parte, como ya hemos comentado anteriormente, existe la posibilidad de modificar estas constantes desde la interfaz gráfica. Para ello, el microcontrolador espera una trama de 13 bytes que contiene, además del STX9, el valor de las constantes en parte alta y baja, con una precisión de dos decimales en el caso de  $Kc_i$ , y de tres decimales para  $Td_i$  y  $Ti_i$ . Para ello se decodifica el mensaje de la siguiente manera después de reenviar el mensaje a *MatLab*, que compara ambas tramas e informa de si la transmisión se ha realizado con éxito:

```

Kc[0] = (256*constantes_USART[0]+constantes_USART[1])/100;
Kc[1] = (256*constantes_USART[6]+constantes_USART[7])/100;
Td[0] = (256*constantes_USART[2]+constantes_USART[3])/1000;
Td[1] = (256*constantes_USART[8]+constantes_USART[9])/1000;
Ti[0] = (256*constantes_USART[4]+constantes_USART[5])/1000;
Ti[1] = (256*constantes_USART[10]+constantes_USART[11])/1000;

```

### 3.9.1.1 Filtro acción derivativa

Habitualmente la acción derivativa debe implementarse con su propio filtro, dado que la variable error (o ángulo en este proyecto) suele presentar bastante ruido y derivar magnitudes con ruido hace que lleguemos a señales indefinidas.

El filtro de la derivada consiste en un filtro digital de paso alto, en el cual la función de transferencia pasa de ser:

$$D(s) = T_d s$$

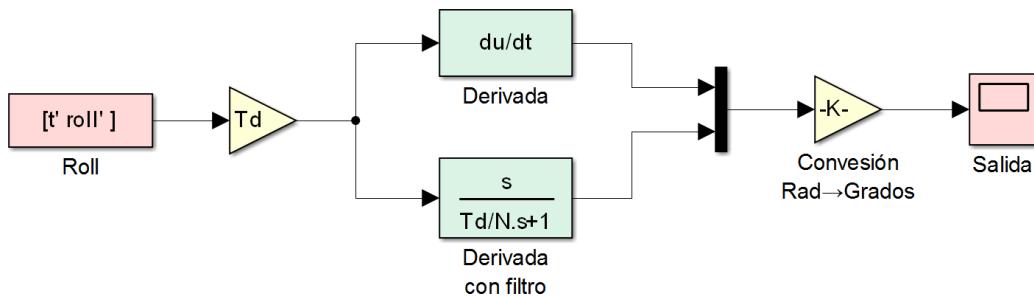
a la siguiente, aplicando una pequeña corrección:

$$D(s) = T_d \frac{s}{1 + s \frac{T_d}{N}}$$

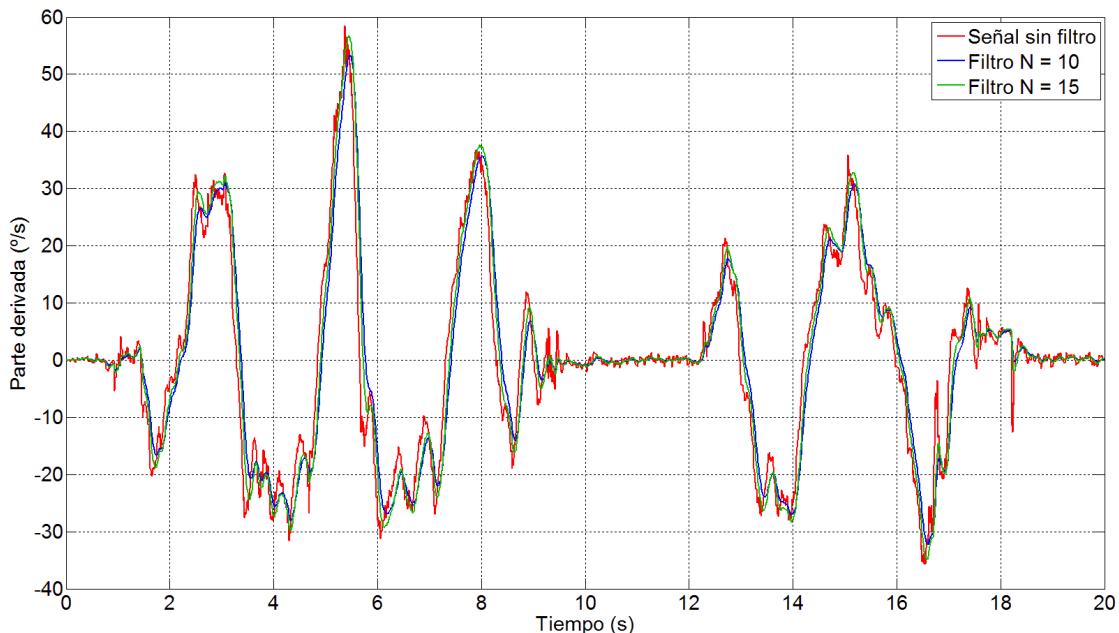
donde N es un parámetro experimental, que define la cantidad de filtrado.

Como podemos ver en la fórmula, si N tiende a valores muy altos, la función de transferencia con filtro y sin filtro coinciden. Este parámetro suele variar entre 5 y 20.

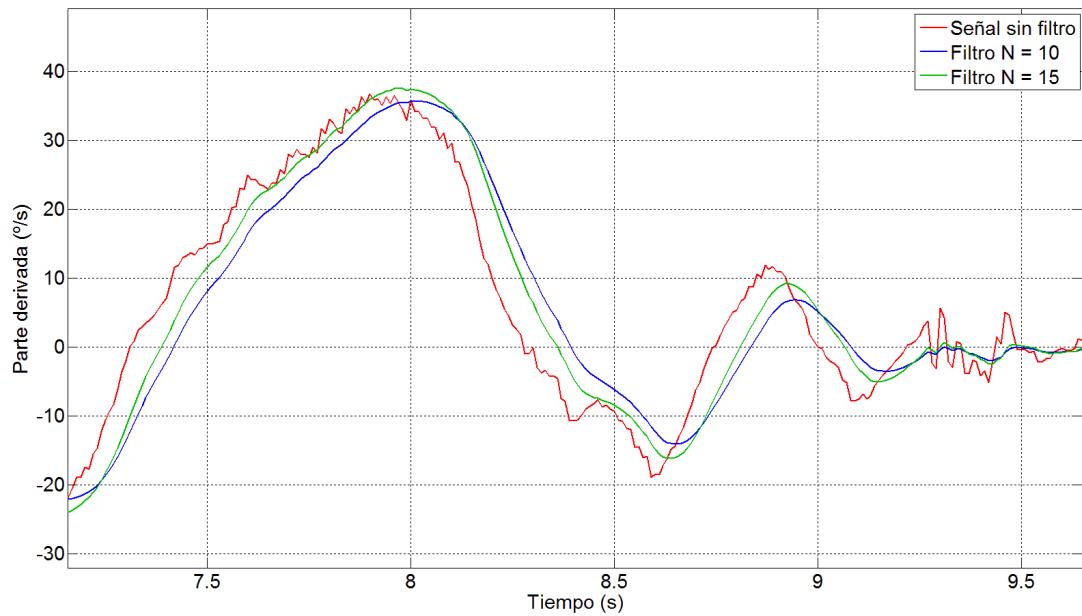
Tras implementarlo en *Simulink*, se lanzó una simulación cuyos datos de entrada eran ángulos reales tomados de la placa, para ver como respondían ambas derivadas. El esquema implementado fue el siguiente:



A partir de tal diagrama, mediante el  $T_d$  teórico que vimos en el apartado anterior cuyo valor era de 1, y probando dos valores de N diferentes, obtuvimos los siguientes resultados:



La primera impresión que nos deja el filtro es que consigue eliminar el ruido de la señal derivada (roja). Sin embargo, vamos a analizar los resultados con más detenimiento haciendo un zoom de la gráfica.



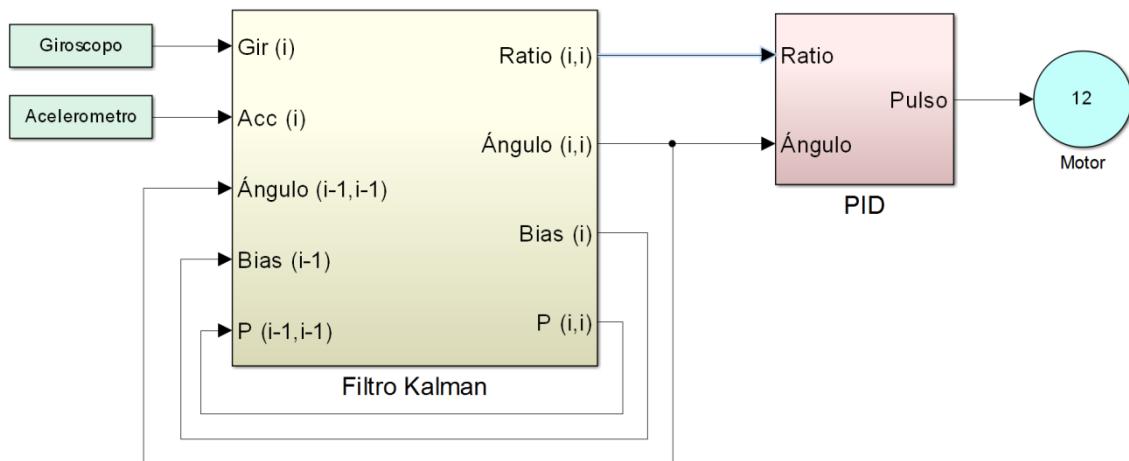
Comprobamos que sí es verdad como las señales filtradas (en verde y azul) no presentan los picos de la derivada habitual. Sin embargo existe un detalle que nos alejó de implementar este algoritmo en el código final, y fue el hecho del retardo que presenta el filtro. Aunque se ve que es de algunas décimas de segundo, recordemos que la placa actúa sobre los motores cada centésima de segundo, lo cual implica que puede realizar 10 correcciones en cada décima. Si estas correcciones actúan sobre una señal que viene retardada, pongamos 2 décimas, existen 20 acciones de control sobre una señal falsa, lo cual hace inviable el sistema.

En la práctica, implementar este filtro constaría de añadir una ecuación más, concretamente una ecuación para aplicar la corrección que vimos en la función de transferencia a la variable error, para así obtener el nuevo parámetro, llamado error del filtro, y derivar sobre este último. El hecho de tener que aumentar el número de ecuaciones y de variables fue otro detalle que nos llevó a dejar de lado el filtro de la parte derivativa y proseguir sin él.

La solución que aplicamos para minimizar el principal problema de no trabajar con el filtro (el ruido de la derivada), fue no aplicar la señal de control D sobre la derivada del ángulo, sino ya extraer la derivada del ángulo ( $\dot{\theta}_x$  y  $\dot{\theta}_y$ ) de dentro del filtro Kalman, en la parte en que al ratio obtenido del giróscopo le restamos el *bias* que nos proporciona el filtro en cada paso.

Como sabemos de apartados anteriores, la señal del giróscopo es libre de error. Si le descontamos el offset sabemos que tenemos una magnitud muy limpia sobre la cual aplicar la señal de control derivada. Este es el motivo por el cual en la ecuación de la parte D del PID empleamos las variables `rate_Qx` y `rate_Qy`, que provienen directamente de Kalman.

Si recordamos la caja negra de dicho filtro, lo que estamos realizando es lo siguiente:



Donde hemos añadido sobre el esquema original la unión del PID y Kalman mediante el *Ratio*, es decir, la derivada del ángulo.

### 3.9.2 Ajuste constantes PID

Para poder ajustar el controlador se debe realizar un modelo del sistema, como describimos a continuación, aunque también podemos tratar de ajustar las constantes experimentalmente:

#### 3.9.2.1 Ajuste teórico

El modelo creado para simular el comportamiento del electrocóptero está basado en el equilibrio de momentos alrededor de los ejes X e Y. Para ello, necesitamos hallar las inercias de cada componente respecto de estos ejes. Utilizando la tabla de las masas del apartado 5.1 y sabiendo las distancias aproximadas al centro del electrocóptero podemos realizar una estimación:

	Masa (kg)	Altura (m)	Base (m)	Ancho (m)	$I_x$ local ( $\text{kg}\cdot\text{m}^2$ )	$I_y$ local ( $\text{kg}\cdot\text{m}^2$ )	$I_z$ local ( $\text{kg}\cdot\text{m}^2$ )	$d_x$ (m)	$d_y$ (m)	$I_x$ global ( $\text{kg}\cdot\text{m}^2$ )	$I_y$ global ( $\text{kg}\cdot\text{m}^2$ )
<b>Motor</b>	0,051	0,015	0,020	0,020	6,009E-06	6,009E-06	-	0,230	0,230	2,683E-03	2,683E-03
<b>Biela</b>	0,054	0,010	0,210	0,025	1,982E-04	1,982E-04	-	0,145	0,145	1,329E-03	1,329E-03
<b>ESC</b>	0,013	0,010	0,050	-	2,730E-06	2,730E-06	-	0,120	0,120	1,842E-04	1,842E-04
<b>Batería+ Centro Base</b>	0,425	0,020	0,145	0,005	7,448E-04	7,581E-04	1,50E-05	0,000	0,000	3,865E-04	3,865E-04
<b>Discovery</b>	0,010	0,005	0,100	0,170	2,425E-05	8,404E-06	-	0,000	0,000	2,425E-05	8,404E-06
										Total: <b>0,008803</b>	<b>0,008787</b>

Con estos valores de inercia podemos crear un modelo dinámico que se compone de las siguientes ecuaciones:

$$\text{Eje } X: I_y \cdot \ddot{\theta}_x = F_1 \cdot r - F_4 \cdot r$$

$$Eje\ Y: I_x \cdot \ddot{\theta}_y = F_2 \cdot r - F_3 \cdot r$$

Estas ecuaciones las podemos linealizar teniendo en cuenta que las fuerzas 3 y 4 permanecen constantes ( $F_3$  y  $F_4$ ):

$$Eje\ X: I_y \cdot \Delta\ddot{\theta}_x = \Delta F_1 \cdot r$$

$$Eje\ Y: I_x \cdot \Delta\ddot{\theta}_y = \Delta F_2 \cdot r$$

Como hemos modelizado la fuerza de cada motor, podemos escribir los incrementos de fuerza en función de los incrementos de pulsos, con la ayuda de la pendiente de la anterior recta:

$$Eje\ X: I_y \cdot \Delta\ddot{\theta}_x = b_1 \cdot \Delta p_1 \cdot r$$

$$Eje\ Y: I_x \cdot \Delta\ddot{\theta}_y = b_2 \cdot \Delta p_2 \cdot r$$

A continuación aplicamos transformadas de Laplace, teniendo en cuenta que  $\Delta\ddot{\theta} = s^2 \cdot \Delta\theta$ . Por lo tanto, llegamos al siguiente par de ecuaciones:

$$Eje\ X: I_y \cdot s^2 \cdot \Delta\theta_x = b_1 \cdot \Delta p_1 \cdot r$$

$$Eje\ Y: I_x \cdot s^2 \cdot \Delta\theta_y = b_2 \cdot \Delta p_2 \cdot r$$

Despejando las incógnitas que buscamos, queda:

$$Eje\ X: \Delta\theta_x = \frac{b_1 \cdot r}{I_y} \cdot \frac{1}{s^2} \cdot \Delta p_1$$

$$Eje\ Y: \Delta\theta_y = \frac{b_2 \cdot r}{I_x} \cdot \frac{1}{s^2} \cdot \Delta p_2$$

Hemos obtenido un modelo con un doble integrador ( $\frac{1}{s^2}$ ). Esto no es realmente cierto pues existe una cierta constante experimental que se opone al giro, por lo que el modelo pasaría a ser de tipo 1, es decir, con un solo integrador ( $\frac{1}{s \cdot (s+k)}$ ). En cualquier caso, supondremos que esta constante es nula.

El modelo obtenido indica que el proceso es inestable (no tiene polos con parte real negativa) por lo que se hace necesario incorporar la acción derivativa, además de la proporcional ya establecida. Así mismo, teniendo en cuenta un mal diseño, nos será de gran ayuda también la acción integral.

Para ajustar los valores de las constantes debemos imponer un tiempo de establecimiento ( $\tau$ ), que se trata del tiempo en el que el sistema alcanzaría un 98% de la posición final y que veremos en el siguiente punto.

### Empleo de la herramienta *rltool* de MatLab

La herramienta *rltool* de *MatLab* resulta de utilidad para poder ajustar el controlador una vez tenemos el modelo teórico. Hemos de considerar que *rltool* debe trabajar con la función de transferencia en forma numérica, no simbólica, por tanto hemos de

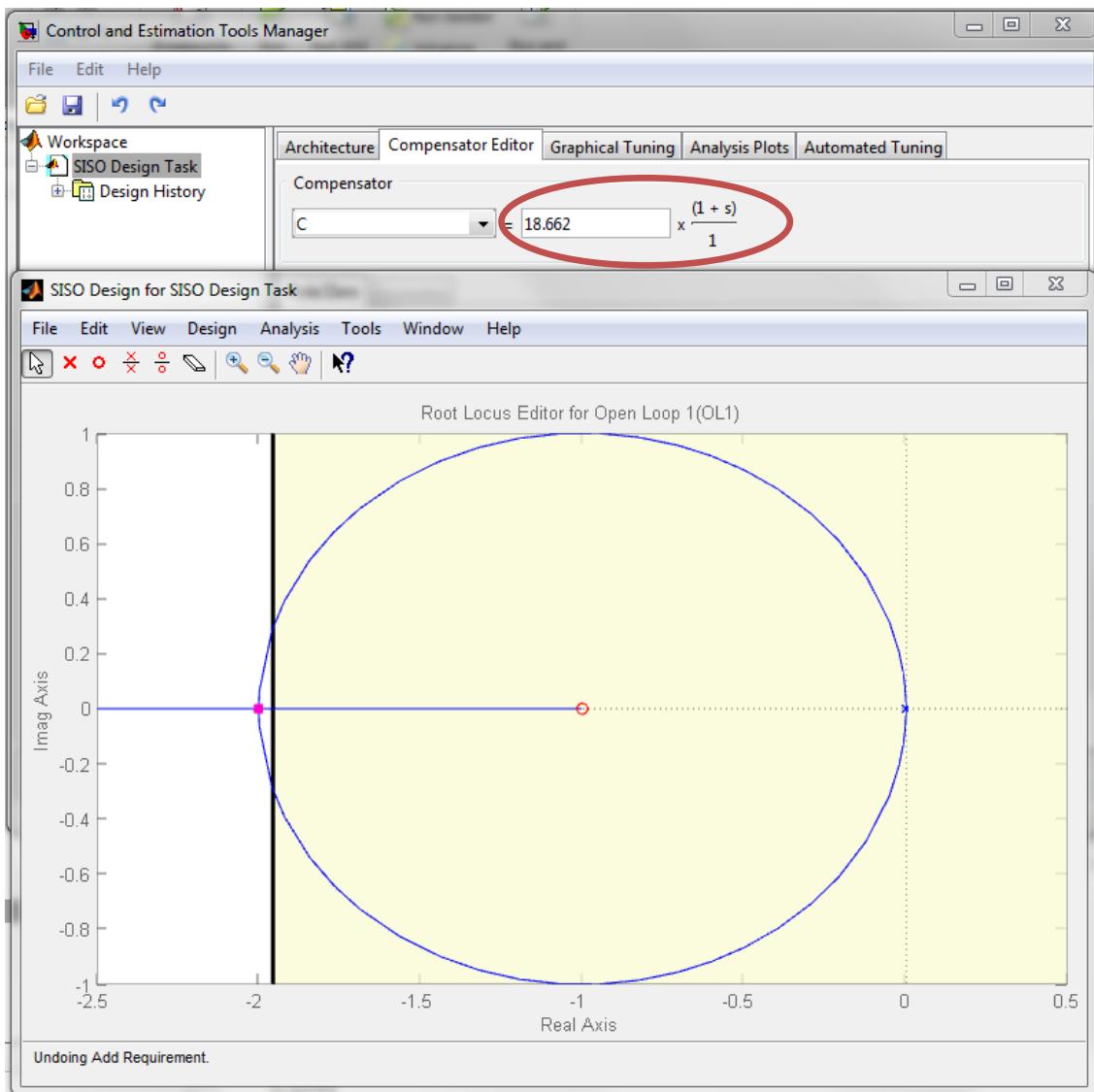
sustituir los valores de las inercias, los radios y las pendientes de los motores. Nos quedaría lo siguiente:

$$\Delta\theta_x = \frac{0,2143}{s^2} \cdot \Delta p_1; \Delta\theta_y = \frac{0,2292}{s^2} \cdot \Delta p_s$$

Como la base teórica que vamos a implementar nos dice que tenemos 2 integradores, no vamos a ajustar un controlador PID a priori, sino un controlador PD definido por los parámetros  $K_c$  y  $T_d$ . Como tenemos 2 grados de libertad hemos de fijar 1 parámetro, que será el tiempo de establecimiento  $\tau$  que definimos previamente. Le daremos un valor arbitrario de 2 segundos.

Podemos decirle a *rltool* que queremos dicho  $\tau$ , además de decirle que deseamos evitar sobreoscilaciones en nuestro sistema. Con dichos requerimientos podemos movernos sobre el lugar de las raíces de nuestra función de transferencia en bucle cerrado, para encontrar un punto en el cual se cumplan nuestras especificaciones y sea *MatLab* quien nos diga qué constantes corresponden a dicho punto.

Por ejemplo, para el eje X realizamos el siguiente ajuste:



Donde vemos que *MatLab* nos da una función de transferencia para el PID de la siguiente forma:

$$PD = C \cdot (s + 1)$$

donde C tiene un valor de 18,662.

Para las ecuaciones, el PD que hemos de implementar es el siguiente:

$$PD = K_c + T_d \cdot s$$

Por tanto, si identificamos términos obtenemos que

$$K_c = T_d = C$$

El siguiente paso sería repetir dicho proceso para el eje Y, en el cual obtenemos resultados análogos y por ello no los repetimos en esta memoria.

Finalmente, podemos resumir este primer ajuste teórico en la siguiente tabla:

	Eje X	Eje Y
$K_c$	18,66	17,45
$T_d$	18,66	17,45
$T_i$	0	0

Tras realizar diferentes ensayos experimentales nos dimos cuenta que estas constantes están muy alejadas de la realidad debido a las múltiples simplificaciones que hemos supuesto en la obtención de la función de transferencia. Por tanto es necesario realizar un ajuste experimental de las mismas.

### 3.9.2.2 Ajuste experimental

Así como anteriormente nos basábamos en datos teóricos y en modelos para hacer todo tipo de cálculos, para llevar a cabo la obtención de las constantes del PID en el desarrollo experimental procedemos de la siguiente forma:

1. Anulamos las acciones integral y derivativa y tratamos de obtener una constante proporcional ( $K_{c_x}$  y  $K_{c_y}$ ) tal que aumente el pulso de una forma coherente al desviarse del equilibrio. Este valor es aproximadamente de 120 pulsos/rad.
2. Seguidamente buscamos una acción derivativa tal que ayude a compensar los cambios bruscos del proporcional, pero que no sea muy agresiva. Esto se corresponde con un tiempo de  $T_d=3$  s·pulsos/rad, aunque no deja de ser un valor aproximado.
3. Finalmente, aplicamos la acción integral con el objetivo de que busque un pulso que mantenga el equilibrio. Se trata de una importante acción, pues puede llegar a corregir errores de modelado y de desequilibrio. Un buen valor puede ser  $T_i=15$  s·rad/pulsos, aunque de nuevo es un valor aproximado y puede ser ajustado en todo momento.

Para realizar de forma más sencilla este ajuste, y no estar cambiando en todo momento las constantes por código, programamos un panel<sup>26</sup> en la GUI de *MatLab* que nos permite ajustar dichas constantes, tanto para el eje X como para el eje Y, de forma mucho más dinámica y rápida. De este modo, el envío a la placa *Discovery* es mucho más compacto y unificado. El aspecto del mismo se muestra en la imagen del margen.



Aunque parezca trivial, la obtención experimental de dichas constantes es una de las tareas más complejas de este proyecto, y posiblemente en una de las que más tiempo se debe emplear. De hecho, este proceso sigue su curso, ya que aún no hemos obtenido tres valores que conjuntamente garanticen la estabilidad del electrocóptero en cada eje<sup>27</sup>.

Para más detalle del proceso de obtención de las constantes, en el [apartado 5](#), ya hacia mediados de diciembre, se describen de forma más detallada las experiencias que hemos llevado a cabo para tratar de obtener estos valores, en muchos casos, sin obtener una respuesta positiva.

### 3.9.3 *Gráficas con ajuste PID*

Dentro del apartado del control automático de los motores, cuando arrancamos motores y pulsamos el botón de ascender, la GUI de *MatLab* nos permite registrar la potencia que nos suministra cada uno de los dos motores sobre los que está actuando el *PID*. Esto es muy útil porque podemos ver de forma muy intuitiva, mediante los indicadores de potencia de los motores, cómo se comporta el *PID* ante cualquier perturbación externa. A continuación podemos ver el cambio sobre los motores 1 y 2, así como el aviso de que *PID* está activado (*PID verde*):

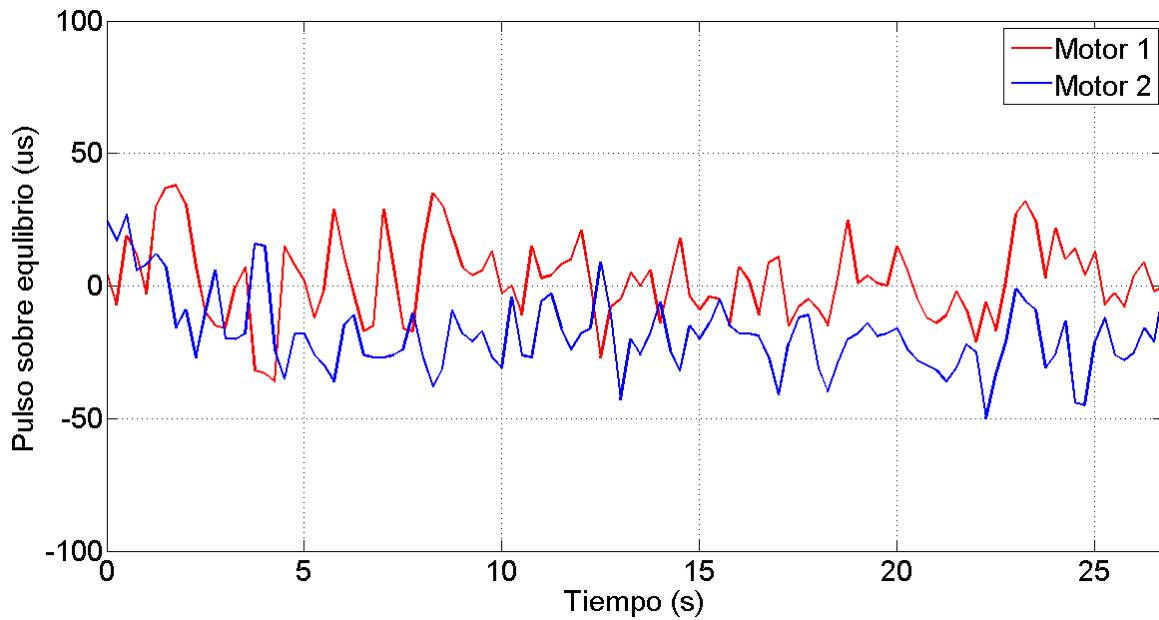
---

<sup>26</sup> Los paneles de la GUI de *MatLab* se comentan de forma extendida en el [apartado 4.4](#).

<sup>27</sup> Ver el [apartado 5.4](#) en el que se dan algunas recomendaciones.



Asimismo, tenemos la opción de graficar dichos valores registrados, y ver cómo es el comportamiento de los motores respecto al pulso de equilibrio a lo largo del tiempo. Aquí podremos observar cuando un motor ha debido suministrar más potencia o menos, indicando en ambos casos si necesitaba más *pitch* o menos, más *roll* o menos, o una combinación de ambas. Se visualiza de forma muy coherente en la siguiente gráfica, la cual se obtuvo de una prueba real realizada en el laboratorio posterior al día de la presentación del proyecto:



## 4 Detalles del Código Final

---

En los siguientes apartados trataremos de estructurar como hemos realizado los principales códigos del proyecto.

### 4.1 Tipos de Variables Usadas

---

Es fundamental saber de las variables de las que disponemos y de las que hacemos uso. He aquí un pequeño detalle de las variables que emplea el compilador que hemos empleado:

- Unsigned char: 1 byte [0, 255]
- Char: 1 byte [-128, 127]
- Unsigned short: 2 byte [0, 65535]
- Short: 2 byte [-32768, 32767]
- Unsigned int/long: 4 byte [0, 4.294.967.295]
- Int/long: 4 byte [-2.147.483.648, 2.147.483.647]
- Float: 4 byte (7 dígitos de precisión)
- Double: 8 byte (15 dígitos de precisión)

### 4.2 CooCox

---

A lo largo de la memoria hemos explicado diferentes aspectos del código implementado en nuestro microcontrolador STMF3, en paralelo con las funciones que realizan diferentes partes del mismo. En este apartado aportaremos una visión general

del mismo, y explicaremos las partes que no han sido ni vayan a ser nombradas en otros apartados.

Antes de analizar el código detenidamente, vamos a resumir las 1746 líneas en la siguiente tabla, que agrupa las funciones que realiza nuestro programa en C:

Líneas	Función	Nombre
<b>0001 - 0010</b>	Llamadas a librerías	-
<b>0012 - 0031</b>	Definición de constantes	-
<b>0033 - 0157</b>	Definición de variables globales	-
<b>0160 - 0166</b>	Codificación de ángulos	Codifica
<b>0169 - 0229</b>	Configuración del giróscopo	L3GD20_Config
<b>0232 - 0296</b>	Lectura del giróscopo	L3GD20_Read
<b>0299 - 0378</b>	Configuración del acelerómetro	Config_Acc
<b>0381 - 0471</b>	Lectura del acelerómetro	Lee_Acc
<b>0474 - 0512</b>	Calibración de los sensores y del filtro	Config_calibrar
<b>0515 - 0524</b>	Calibración de los parámetros del PID	calibrar_PID
<b>0527 - 0873</b>	Programa principal	main
<b>0876 - 1221</b>	Interrupción del Temporizador 4	TIM4_IRQHandler
<b>1225 - 1746</b>	Interrupción de la USART	USART3_IRQHandler

#### 4.2.1 Llamadas a librerías

El código emplea el soporte de 9 librerías de funciones, que es lo primero que aparece en el código:

```
#include "stm32f30x.h" // Librería la ST
#include "cmsis_lib/include/stm32f30x_gpio.h" // Librería de entrada/salida
#include "cmsis_lib/include/stm32f30x_rcc.h" // Librería de señal de reloj
#include "cmsis_lib/include/stm32f30x_tim.h" // Librería de los temporizadores
#include "cmsis_lib/include/stm32f30x_spi.h" // Librería del SPI
#include "cmsis_lib/include/stm32f30x_misc.h" // Librería de misceláneos
#include "cmsis_lib/include/stm32f30x_i2c.h" // Librería del I2C
#include "cmsis_lib/include/stm32f30x_usart.h" // Librería de la UART
#include "math.h" // Librería matemática
```

#### 4.2.2 Definición de constantes

Hemos definido un total de 19 constantes, todas comentadas en el mismo código, careciendo de sentido situarlas en esta memoria.

#### 4.2.3 Definición de variables globales

Las variables globales empleadas en nuestro código (que son las más importantes y numerosas) están enumeradas en la siguiente tabla:

Tipo	Número	Número total <sup>28</sup>	Tamaño total (Bytes)
Unsigned char	47	35924	35924
Char	0	0	0
Unsigned short	10	31	62
Short	8	8	16
Unsignedint	6	6	24
Int	8	8	32
float	5	9	36
double	44	58	464
<b>Total</b>	<b>128</b>	<b>36044</b>	<b>35,7 KBytes</b>

A partir de ellas hemos podido hacer una estimación de la memoria RAM que requiere nuestro código, alrededor de 35,7 KBytes. Recordemos que la RAM de la STM3F3 es de 40 KBytes, así que estamos cerca del límite. De hecho, en algunas ocasiones el compilador nos dio error por sobrepasar la RAM del microcontrolador.

La mayor parte de la memoria está empleada en la definición de la variable *datos*, que permite almacenar hasta 2.560 datos de los sensores que se guardan en tiempo real, pero se envían a posteriori(y por ello es necesario almacenarlos) como se explicó en el apartado comunicación inalámbrica.

Por su parte, dentro de cada función se encuentran definidas funciones específicas para las mismas. En cualquier caso, el empleo de variables globales, todo y que consume más memoria, es esencial debido a que permite que su valor pueda ser reconocido por distintas funciones, por ejemplo por la interrupción del Temporizador 4 y por la de la USART.

#### 4.2.4 Función de calibrar los sensores y los filtros

Esta función declarada como **voidconfig\_calibrar(void)** realiza las siguientes funciones:

- Apaga los LEDs burbuja.
- Pone a 0 las variables del filtro complementario.
- Pone a 0 las variables de offset de los sensores.
- Cambiamos la interrupción del Temporizador 4 de 10 ms a 5ms.

---

<sup>28</sup> El número no coincide con el número total debido a que algunas variables son vectores o matrices de varios elementos.

- Deshabilitamos la interrupción por recepción de la USART (se habilitará de nuevo cuando se termine de calibrar).

#### 4.2.5 Función de calibrar el PID

Esta función declarada como `voidcalibrar_PID(void)` vuelve a poner a 0 los parámetros del PID. Es de especial importancia reiniciar el valor de la acción integral debido a que ésta va acumulando error conforme va actuando el controlador.

#### 4.2.6 Función de codificación de los ángulos

Para enviar datos del acelerómetro calibrados por la por la USART ([comando 4](#)) empleamos la función `intCodifica(doubleangulo)`, cuyo parámetro de entrada es un ángulo en coma flotante que puede variar entre -90° y 90° y la salida es un entero de 2 bytes sin signo, entre 0 y 65535 realizando una transformación lineal. Más tarde, esta salida se divide en parte alta y parte baja para enviar 2 datos al PC.

#### 4.2.7 Programa principal

Vamos a explicar los aspectos más relevantes del programa principal que no han sido tratados en otros apartados de la memoria.

Empezamos el *main* con un retardo, para dejar que se establece la corriente en todos los componentes de la placa, y no empezar a actuar sobre los distintos elementos antes que alcancen su régimen de funcionamiento nominal:

```
retardo = 0;
while (retardo < 4000000)
{
    retardo++;
}
```

Continuamos definiendo las diferentes estructuras que vamos a emplear a lo largo del código:

<code>GPIO_InitTypeDefGPIO_InitStructure;</code>	//	Estructura	para	la
<code>configuración de los Pines de Entrada/Salida</code>				
<code>SPI_InitTypeDefSPI_InitStructure;</code>	//	Estructura	para	la
<code>configuración del SPI</code>				
<code>I2C_InitTypeDef I2C_InitStructure;</code>	//	Estructura	para	la
<code>configuración del I2C</code>				
<code>TIM_TimeBaseInitTypeDefTIM_TimeBaseStructure;</code>	//	Estructura	para	la
<code>configuración de los Temporizadores Base de tiempos</code>				
<code>TIM_OCInitTypeDefTIM_OCInitStruct;</code>	//	Estructura	para	la
<code>configuración de los Temporizadores Modo PWM</code>				
<code>TIM_ICInitTypeDefTIM_ICInitStruct;</code>	// Para el modo captura			
<code>USART_InitTypeDefUSART_InitStructure;</code>	//	Estructura	para	la
<code>comunicación USART</code>				
<code>NVIC_InitTypeDefNVIC_InitStructure;</code>	//	Estructura	para	la
<code>configuración de las Interrupciones</code>				
<code>NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);</code>	// Se establece que solo existen			
<code>niveles de prioridad (no hay subniveles)</code>				

Lo siguiente es inicializar los valores de configuración de los registros del giróscopo y del acelerómetro, que toman los valores que vimos en el apartado de conexiones internas.

A continuación configuraremos los pines de entrada y salida, habilitando en primer lugar la señal de reloj para todos los puertos.

```
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE); // Puerto A (SPI/ESC 1/PWM Sonar)
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOB, ENABLE); // Puerto B (I2C)
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOC, ENABLE); // Puerto C (ESC 2-4)
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOD, ENABLE); // Puerto D (USART)
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOE, ENABLE); // Puerto E (LEDS y SPI)
```

En las siguientes líneas definimos para cada pin que vamos a emplear las características de la estructura `GPIO_InitStructure` para que funcione correctamente. No vamos a poner el código de esta parte, pero sí lo resumimos en una tabla:

Pin	Función	Modo	Tipo de salida
PA5	SCK SPI	AF5	Pulldown
PA7	MO SPI	AF5	Pulldown
PA6	MI SPI	AF5	Pulldown
PE3	CS SPI	OUT	Pull up
PB6	SCL I2C	AF4	Pulldown
PB7	SDA I2C	AF4	Pulldown
PD8	TX USART	AF 7	Pull up
PD9	RX USART	AF 7	Pull up
PB4	ESC 1	AF 2	Pull up
PA4	ESC 2	AF 2	Pull up
PC8	ESC 3	AF 2	Pull up
PC9	ESC 4	AF 2	Pull up
PB9	PWM Sónar	AF 1	Pull up
PA15	Captura Sónar	AF 1	Pulldown
PE9 - PE15	LEDs	OUT	Pull up
PA0	Calibrar	IN	No pull

Cabe decir además que empleamos todos los pines con una velocidad de 50 MHz y con modo de salida Pull/Push.

Proseguimos el código configurando todos los canales que hemos ido explicando a lo a lo largo de la memoria, como son los del giróscopo, acelerómetro, USART, sonar y temporizadores 2, 4 y 17. Todo ello ha sido explicado anteriormente.

Por último, dentro de la función *main* nos encontramos el bucle principal, cuya única función es leer si se está pulsando el botón de usuario de la placa. Cuando detecta la línea activada, es decir, que se ha presionado el botón, la placa procede a calibrarse:

```
while(1)
{
if (GPIOA->DATA[0] == 1) // Calibraremos al pulsar el botón
{
calibrar = 1;
config_calibrar();
}
}
```

Como conclusión, podemos resumir las diferentes fases del código principal en la siguiente tabla:

Líneas	Función
<b>530 - 547</b>	Declaración de variables y estructuras
<b>549 - 560</b>	Inicialización de variables
<b>563 - 687</b>	Inicialización de pines de entrada / salida y alternativos
<b>690 - 721</b>	Configuración del giroscopio
<b>724 - 751</b>	Configuración del acelerómetro
<b>754 - 771</b>	Configuración del Temporizador 4 para el PID (Modo base de tiempos)
<b>774 - 794</b>	Configuración de la USART 3
<b>797 - 818</b>	Configuración del Temporizador 3 para los ESC (Modo PWM)
<b>821 - 837</b>	Configuración del Temporizador 17 para el sónar (Modo base de tiempos)
<b>840 - 862</b>	Configuración del Temporizador 2 para el sónar (Modo Captura)
<b>864 - 872</b>	Bucle principal

### 4.3 GUI de MatLab

La interfaz gráfica usada para nuestro proyecto ha sido la GUI de *MatLab*. La siguiente imagen muestra visualmente la interfaz gráfica que hemos empleado en el PC para comunicarnos con la placa, y enviar y recibir instrucciones y datos:



Podemos dividir la interfaz en diferentes paneles, asociados a los comandos que veremos en el siguiente apartado:

- **Panel de comunicación:** permite seleccionar el puerto de comunicación y la velocidad, así como abrir o cerrar la comunicación (comando 0).
- **Panel de transmisión de datos:** permite decir a la placa que se recojan datos de los sensores y la cantidad (múltiplo de 10), sin calibrar (comando 1) o calibrados (comando 4). Cuando se han recibido todos los datos es posible graficarlos con el botón *Representar*, o guardarlos en formato texto, Excel y variable de *MatLab* con el botón *Guardar datos*. En este panel también se informa de la fecha y hora de la última recepción correcta de datos<sup>29</sup>.
- **Panel del sonar:** permite recibir la altura medida por el sonar y mostrarla en pantalla (comando 5).

<sup>29</sup> Ver el apartado 4.4.2

- **Panel de otras opciones:** permite decirle a la placa que se calibre (comando 2) o que realice una parada de emergencia de los motores y la lectura de los sensores (comando 6).
- **Panel de control de los motores manual:** permite controlar manualmente los motores desde el PC (comando 5). Podemos enviar el pulso que queremos a cada motor. Cuenta además con un botón para arrancarlos y pararlos.
- **Panel de control de los motores automático:** permite controlar de forma automática un proceso previamente definido, de arranque, ascenso, mantener altura, descender y parar. Además, tiene una opción extra que son los que hacen que la aeronave no tripulada vuele en modo *Hovercraft*.
- **Panel de envío de constantes PID:** para no estar cambiando en todo momento las constantes del PID que queremos enviar al cuadricóptero, generamos un panel en el que poder configurar las constantes del PID ( $K_c$ ,  $T_i$ ,  $T_d$ ) de forma manual pero rápida, de los ejes X e Y de nuestra plataforma.
- **Panel de información:** indica cual es el estado actual de la comunicación para cada comando, por ejemplo, cuando se piden datos a la placa y los descarga, muestra el porcentaje de datos que le han llegado en tiempo real. También muestra cualquier tipo de error en la comunicación para informar al usuario.
- **Panel de recepción del PID:** consta de 4 gráficas que indican la carga en tiempo real de los motores cuando se entra en modo automático, así como un texto que se ilumina cuando se activa el PID en el micro. Además permite registrar los datos recibidos de la salida del PID en tiempo real, para graficarlo a posteriori como vimos en el apartado anterior.

### 4.3.1 Mensajes de la GUI de MatLab

Los mensajes de la GUI de *MatLab* nos permiten saber en todo momento el estado de comunicación entre la placa y *MatLab*. Por ello es esencial conocer con detalle los diferentes mensajes, pues además del simple hecho de informar si ha funcionado o no una llamada, permiten conocer detalles más técnicos cuando existe un problema. Según el tipo de mensaje podemos saber si el fallo es debido a que no se ha recibido o enviado ningún dato, o si lo recibido no ha coincidido con lo que se esperaba obtener, entre otras cosas.

El conocimiento de este apartado es esencial para que el usuario pueda decir a priori si un error se ha debido a *MatLab* o a la placa.

#### 4.3.1.1 *Mensajes del comando 0 (inicio de la comunicación)*

*Éxito al comunicar.* Se ha recibido correctamente el STX0.

*Esperando confirmación...* Se ha enviado STX0 y se espera la réplica de la placa.

*Error al comunicar.* Vuelva a abrir la comunicación. Se ha recibido un dato distinto de STX0.

*Error al comunicar. Tiempo agotado. Vuelva a enviar la petición.* Ha saltado el temporizador de *MatLab* y no se ha recibido ningún dato.

*Comunicación cerrada por el usuario.* Se ha cerrado el puerto COM, sin informar a la placa de ello.

#### 4.3.1.2 Mensajes del comando 1 (sensores sin calibrar)

*El número de datos debe estar comprendido entre 10 y 2560.* Se ha introducido un número de datos erróneos para recibir por la placa.

*Esperando mensaje de confirmación...* Se ha enviado STX1 y se espera que sea devuelto por la placa.

*Éxito al comunicar. Esperando la recepción de los datos...* La placa ha devuelto correctamente el STX1.

*Error al comunicar. Vuelva a enviar la petición.* Se ha recibido un dato distinto de STX1.

*Tiempo restante estimado: 10 s.* Tiempo que se espera que tarde la placa en almacenar los datos. Se actualiza a tiempo real.

*Recibiendo datos: 88%...* Indica el porcentaje de datos que se ha recibido hasta el momento. Se actualiza en tiempo real.

*Datos procesados. 0 datos perdidos. 0 dato/s erróneos.* Indica los datos perdidos durante la transmisión por coordinar las tramas y el número de datos cuyo CHECKSUM no se ha cumplido.

*Error al procesar los datos. Ninguno correcto.* No ha llegado ningún dato correcto, en todos ha fallado el CHECKSUM.

*Error al comunicar. Tiempo agotado. Vuelva a enviar la petición.* Ha saltado el temporizador de MatLab y no se ha recibido ningún dato.

*Error al recibir los datos, no se ha recibido ninguno.* Ha saltado el temporizador de MatLab y no se ha recibido ningún dato.

#### 4.3.1.3 Mensajes del comando 2 (calibración de la placa)

*Enviando comando de calibración...* Se ha enviado STX2 y se espera su réplica.

*Confirmación recibida. El controlador procederá a calibrarse.* La placa ha devuelto correctamente el STX2.

*Error al comunicar. Vuelva a enviar la petición.* Se ha recibido un dato distinto de STX2.

*Error al comunicar. Tiempo agotado. Vuelva a enviar la petición.* Ha saltado el temporizador de MatLab y no se ha recibido ningún dato.

#### 4.3.1.4 Mensajes del comando 3 (control de los motores)

*Esperando confirmación para controlar motores...* Se ha enviado STX3 y se espera su réplica.

*Controlando motores.* Se ha recibido correctamente el STX3.

*Error al comunicar. Vuelva a enviar la petición.* Se ha recibido un dato distinto del STX3

*Potencia del motor 1 recibida.* Se ha recibido un dato tras pedir el pulso de un motor.

*Error al comunicar. Tiempo agotado. Vuelva a enviar la petición.* Ha saltado el temporizador de MatLab y no se ha recibido ningún dato.

*No se ha recibido el pulso del motor 3.* Ha saltado el temporizador de MatLab sin recibir ningún dato tras pedir el pulso de un motor.

*Aumento de pulso del motor 4 enviado.* MatLab ha enviado la trama correspondiente para aumentar el pulso de un motor.

*No se envió el pulso porque excedía el máximo.* El usuario ha intentado enviar un pulso muy alto, así que no se ha completado la operación.

*Disminución de pulso del motor 2 enviada.* MatLab ha enviado la trama correspondiente para disminuir el pulso de un motor.

*No se envió el pulso porque era inferior al mínimo.* El usuario ha intentado enviar un pulso muy bajo, así que no se ha completado la operación.

*Comunicación cerrada con los motores.* MatLab ha enviado STX3 para indicar que se deja de controlar los motores. No se espera ninguna respuesta.

#### 4.3.1.5 Mensajes del comando 4 (sensores calibrados)

Mensajes idénticos a los del comando 1.

#### 4.3.1.6 Mensajes del comando 5 (envío altura del sonar)

*Esperando recepción de la altura...* Se ha enviado STX5 y se espera la recepción de los 2 bytes de la altura.

*Altura recibida con éxito.* Se han recibido 2 datos tras pedir la altura del sonar.

*Error al recibir la altura del sonar. Tiempo agotado.* Ha saltado el temporizador de MatLab y no se ha recibido ningún dato.

#### 4.3.1.7 Mensajes del comando 6 (parada de emergencia)

*Parada de emergencia enviada.* Se ha enviado STX6. No se espera respuesta puesto que la placa se bloquea.

#### 4.3.1.8 Mensajes del comando 7 (arranque de motores)

*Esperando confirmación de arrancar...* Se ha enviado STX7 y se espera la réplica de la placa.

*Arrancando motores...* La placa ha devuelto correctamente STX7.

*Error al comunicar con la placa.* Se ha recibido un dato distinto de STX7.

*Motores arrancados.* La placa ha devuelto correctamente STX7+1.

*Error al arrancar los motores.* Se ha recibido un dato distinto de STX7+1.

*Error al arrancar los motores. Tiempo agotado.* Ha saltado el temporizador de *MatLab* y no se ha recibido ningún dato.

*No se ha recibido confirmación sobre el arranque de los motores. Tiempo agotado.* Ha saltado el temporizador de *MatLab* y no se ha recibido ningún dato que indique que los motores hayan parado de arrancar.

#### 4.3.1.9 Mensajes del comando 8 (parada de motores)

*Parando motores...* Se ha enviado STX8 y se espera la réplica de la placa.

*Motores parados.* La placa ha devuelto correctamente STX8.

*Error al parar los motores.* Se ha recibido un dato distinto de STX7.

*No se ha recibido confirmación del paro de los motores. Tiempo agotado.* Ha saltado el temporizador de *MatLab* y no se ha recibido ningún dato.

#### 4.3.1.10 Mensajes del comando 9 (envío constantes del PID)

*Esperando confirmación de las constantes del PID.* Se ha enviado el STX9 y las 6 constantes y se espera la réplica de las mismas por parte de la placa.

*Recibidos 13 de 13 datos.* Número de parejas de 2 bytes que se han recibido tras enviar las constantes del PID, además del dato inicial para iniciar la transmisión.

*Información del PID recibida correctamente.* La placa ha devuelto correctamente las constantes del PID, porque coinciden con las enviadas.

*TdX KcY incorrectos.* Hay algunas constantes que no coinciden con las enviadas.

*Error al confirmar las constantes del PID. Tiempo agotado.* Ha saltado el temporizador de *MatLab* y no se ha recibido ningún dato.

### 4.3.2 Exportación de datos

Una vez hemos obtenido todos los parámetros de los ángulos de *pitch* y *roll*, así como las velocidades angulares en los tres ejes, podemos exportar los datos bien sea en un fichero *Excel* o bien sea un archivo de texto, ambos estructurados de la siguiente manera:

Tiempo (s)	Roll acelerómetro (º)	Pitch acelerómetro (º)	Roll giróscopo (º)	Pitch giróscopo (º)	Vel. ang. X (º/s)	Vel. ang. Y (º/s)	Vel. ang. Z (º/s)
0	-0,193636988	-0,02059968	0	0	0,110627909	0,011444266	0,095368887
0,01	-0,193636988	-0,02059968	-0,000114439	0,001106279	0,110627909	0,011444266	0,072480354
0,02	0,042572671	0,094758526	-0,000343311	0,001945527	0,057221332	0,034332799	0,072480354
0,03	0,042572671	0,094758526	0,000724834	0,002479594	0,049591821	-0,247959106	0,019073777
0,04	0,281528954	-0,026092927	0,002784819	0,003013651	0,057221332	-0,164034485	0,057221332
...	...	...	...	...	...	...	...

Con el proyecto se incluyen diferentes ficheros que incluyen datos obtenidos de los sensores, en los tres formatos citados anteriormente.

## 5 Pruebas experimentales

---

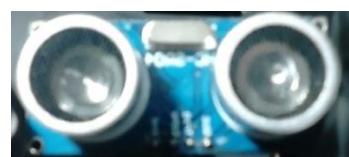
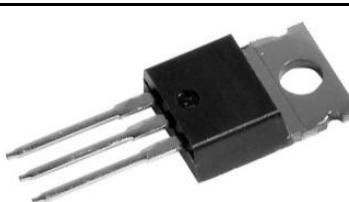
En este apartado comentaremos los aspectos menos relacionados con la programación que nos hemos encontrado a lo largo de nuestro trabajo.

### 5.1 Materiales empleados

---

Los principales materiales empleados en el desarrollo del presente proyecto se muestran en la siguiente tabla, en la que también indicamos el peso de los más notorios y de los cuales obtenemos las inercias, que vimos en el [apartado 3.9.2.1](#), así como las unidades utilizadas de cada uno de ellos y una imagen para la rápida identificación de los mismos:

Componente	Unidades	Imagen	Masa (g)
<i>Motor BLDC</i>	4		50,6
<i>ESC</i>	4		12,6
<i>Chasis FYT-450</i>	1		308,5
<i>Batería 4000 mAh</i>	1		331,3
<i>Placa STM32-F3 Discovery</i>	1		100,6

<i>Módulo inalámbrico HM-TRP</i>	2		-
<i>Hélices</i>	4		-
<i>Interruptor On/Off</i>	1		-
<i>Sónar HYT-271</i>	1		-
<i>Condensadores electrolíticos 10 µF</i>	2		-
<i>Condensadores 100 nF</i>	2		-
<i>Regulador de tensión LM7805</i>	1		-

A todos los componentes anteriores hay que añadir el material fungible que hemos ido utilizando y que hemos encontrado en el laboratorio, como pueden ser cables y cinta adhesiva de diferentes colores, estaño para soldar, silicona térmica, diversa tornillería o la madera de balsa empleada para la unión de la base de la placa *Discovery* con el chasis del electrocóptero con el fin de evitar contactos indeseados.

Apuntar también que el peso total de nuestro proyecto es de unos **1.027 gramos**, multiplicando el peso anterior por el número de unidades de cada componente y sumando a los de la tabla anterior la masa del material fungible al que antes hacíamos referencia, y que no se detallan en la tabla debido a su despreciable peso por separado.

En definitiva, como en todo proyecto, un conocimiento de los materiales a utilizares fundamental, bien sea durante el proyecto o en el estudio de viabilidad previo al mismo, ya que nos limitamos a un presupuesto del que no podemos excedernos y conocer las características de cada uno de ellos hace que podamos sacarles el máximo rendimiento posible.

## 5.2 Cuaderno diario

---

A continuación se comenta día tras día los distintos avances que íbamos realizando, así como los problemas que nos encontramos y que creemos que pueden ser comunes para posibles proyectos futuros de un cuadricóptero de estas características:

24/10/2014

- Estudio del control de los motores a través de las señales PWM.
- Configuración de los 4 canales del TIMER 3.
- Conocimiento del funcionamiento del filtro complementario, tanto nivel bajo como nivel alto.

31/10/2014

- Estudio del funcionamiento del acelerómetro y el giróscopo. Lectura de los correspondientes *DataSheet* y decidir los valores de los registros.
- Configuración de los registros de los dos sensores y medición de los mismos para comprobar que realmente escribimos los valores deseados.
- Integración de la señal del giróscopo, suponiendo que su valor inicial es nulo. Aplicamos el método de los trapecios para una mayor precisión.
- Configuración del TIMER 4 para tomar medidas cada 10 ms.

7/11/2014

- Cálculo de ángulos con el acelerómetro: *roll* y *pitch*.
- Primera previsualización de la distribución de pines. Generamos archivo *Excel* con los pines de la placa.

14/11/2014

- Configuración de calibrar giróscopo. Se activa al arrancar la placa o con el botón de usuario de la placa (azul).
- Estudio del dispositivo sonar y asignación de pines.
- Situación del sonar en la parte inferior del Electrocóptero
- Asignación de las funciones de los *LEDs* de la placa *Discovery*.
- Se decide la configuración óptima y se comienza a soldar los cables y los conectores, distribuyendo las señales de 5 V y de 0 V coherentemente.

19/11/2014

- Se finaliza la soldadura integrando la placa *Discovery* soldando pin a pin.
- Colocamos el dispositivo regulador de tensión y el módulo inalámbrico en la placa pegándolo con silicona térmica.
- Se comprueba que la soldadura es correcta utilizando un multímetro.

21/11/2014

- Se comprueba que el regulador de tensión funciona correctamente, al llegar los 3,3V que necesita el módulo inalámbrico para funcionar.
- Instalación de *drivers* y configuración en el ordenador del módulo inalámbrico utilizado para la recepción y envío de datos.

28/11/2014

- Decidimos los temporizadores a los que conectar el sonar. Necesitamos uno que genere una señal PWM y otro que disponga de modo captura. Después de varias pruebas, seleccionamos el canal 1 del TIMER 2 para el modo captura y el canal 1 del TIM 17 para generar la señal PWM.
- Probamos el envío y recepción de datos mediante la USART, configurándolo en el portátil y en la *Discovery*. Conseguimos enviar y recibir desde el ordenador un total de dos datos, y lo generalizamos para el conjunto de trece datos que recibimos en la aplicación real.

5/12/2014

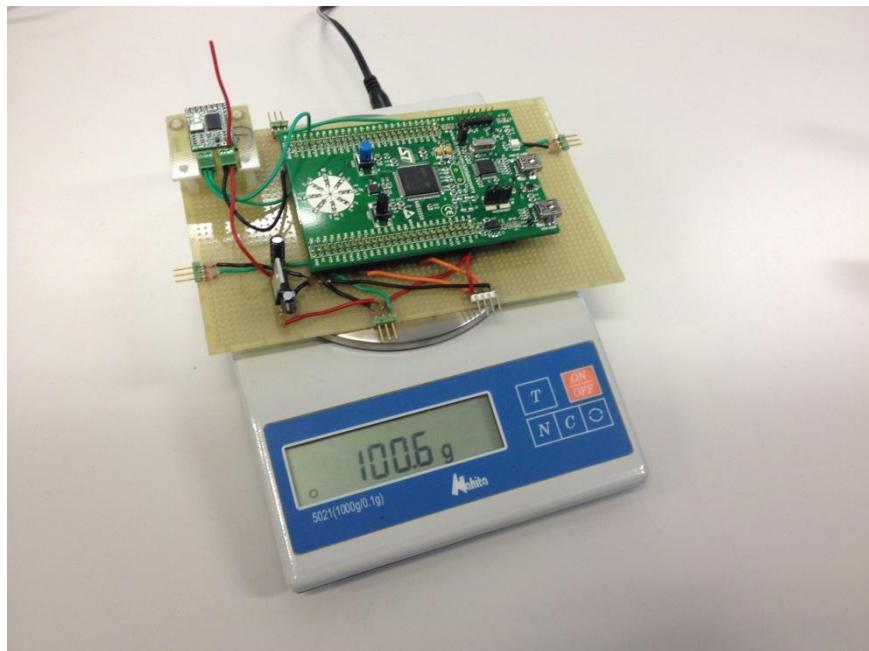
- Continuamos con el envío y recepción de datos desde *MatLab* al microcontrolador y viceversa.
- Además, comprobamos que variando el movimiento de la *Discovery* (simulando un movimiento del cuadricóptero), se modifican los datos recibidos sobre *MatLab*.

10/12/2014

- Sin previo aviso, el puerto PC6 con el que sacábamos la señal para el motor 1 por medio del canal 1 del TIMER 3 parece haber muerto, pues no obtenemos ninguna señal a diferencia de otros días. Buscando en el *DataSheet* otros puertos que comparten el canal 1 de este mismo TIMER, decidimos cambiar de puerto esta señal, desoldando el cable que unía el puerto con la conexión y reemplazándolo por otro que llegase al nuevo puerto (PB4).
- Primeros ensayos con el sonar al comprobar que la línea de entrada PA15 (captura) admite tensiones de 5V.

12/12/2014

- Una vez modificados los programas de *MatLab* y *Discovery*, hacemos una simulación mediante osciloscopio y *MatLab* de cómo variamos la potencia de los motores. En este sentido, vamos incrementando la señal de pulso en dos microsegundos, y vemos mediante el osciloscopio, cómo sería el comportamiento de los motores a dicha señal. Para ello creamos en la interfaz gráfica un *push button* que nos permite ir incrementando la señal en dos microsegundos, o decrementar en dos microsegundos la misma.
- Colocamos un *display* en forma de texto, para que, si el usuario lo desea, pueda ver numéricamente el valor del pulso que está enviando a los motores, ya que es más cómodo visualizarlos de este modo que con el osciloscopio.
- Pesamos los principales componentes del proyecto mediante una báscula electrónica.



- Realizamos el montaje para la linealización de cada uno de nuestros motores (cuatro en total), como describimos en el [apartado 3.7.1](#).
- Respecto de la linealización de los motores anterior, de forma teórica, una vez tengamos el pulso necesario de cada motor (diferente en cada caso, pues la caracterización es diferente en cada motor) para sustentar el peso de los componentes (hemos calculado de forma rápida que cada motor debe soportar aproximadamente unos 2.48 Newton de fuerza), significa que este es nuestro punto de equilibrio sobre el cual nuestro cuadricóptero conseguirá volar ascendentemente y estabilizarse.

16/12/2014

- Planeamos caracterizar los dos motores que nos faltan. Cuando uno de ellos estaba aproximadamente proporcionando la mitad de su empuje máximo, se suelta el tornillo que une la base del motor con el propio motor, provocando que se suelte de la base y caiga del balancín, deformándose la hélice.



- Como consecuencia, debemos cambiar el motor y la hélice y caracterizarlo de nuevo. Una vez caracterizado, lo montamos junto a los otros en la base, así como la instalación del sónar en la parte inferior.
- Por otro lado, empezamos a soldar los cables de positivo y negativo de cada *ESC* a la estructura del cuadricóptero, para así llevarlos todos a la batería.

19/12/2014

- En un primer momento, acabamos de realizar el montaje de todos los componentes de la base. Ajustamos los cuatro motores junto con los cuatro *ESCs*, uniéndolos de la forma más precisa a la base. Además, instalamos de la forma más centrada posible la batería en la parte inferior de la base, donde tiene su compartimento.
- Construimos un pequeña base aislante de madera que nos sirva para apoyar de forma estable y lo más plana posible la placa sobre la base del cuadricóptero, procurando además que los ejes del acelerómetro y el giróscopo queden alineados con los brazos de la base.
- Acabamos de soldar todos los *ESCs* a la base, haciendo así que todos queden alimentados por la batería (punto común base).
- Una vez tenemos todo construido y montado, lo probamos con un resultado negativo<sup>30</sup> (prueba con batería). El principal síntoma es que de uno de los *ESC*(del que alimenta a la placa) sale una pequeña columna de humo. Las posibles causas, según lo comentamos con el profesorado, es que el *ESC* posiblemente esté defectuoso, al estar a la intemperie durante todo un año.
- Una vez lo desmontamos para quitar el *ESC*, que presenta un olor muy fuerte, comprobamos con el multímetro que el voltaje que nos proporciona la batería es de 12.5V, que es un poco superior a lo que necesitan los motores, que es de

---

<sup>30</sup> Primera prueba del Electrocóptero: en este caso con batería.

unos 11.4V. Hablando con los profesores, descartamos que esta sea la causa del fallo, ya que este elemento aguanta hasta 15V o más.

- Por lo tanto, cambiamos el ESC y configuramos de nuevo la estructura. Volvemos a desoldar y soldar el cableado. Esta vez realizamos la prueba con la fuente de alimentación, para reafirmar la hipótesis de que la batería no era la causa del fallo. Conectando motor a motor, desde la GUI de MatLab, vemos que todos los motores funcionan perfectamente y tiene un funcionamiento óptimo, sin que se dispare la intensidad, pero llegamos al cuarto motor y vemos que deja de funcionar el proceso y se dispara la corriente hasta el límite de seguridad que le fijamos. Suponemos, por tanto, que esta fue la causa que generó el episodio anterior.
- Reemplazamos el ESC por uno nuevo y lo colocamos en la base. Volvemos a probar todo con la fuente de alimentación y ahora todos los motores se encienden perfectamente, sin ningún pico de intensidad que pueda dañar los componentes.
- Por último, una vez sabemos que funciona todo con la fuente de alimentación, solo nos queda comprobar que la batería no da ningún problema. Finalmente, funciona todo correctamente.

03/01/2015

- Prueba del funcionamiento<sup>31</sup>, buscando los pulsos que nos den el equilibrio, para así poder implementar un principio de filtro. Lo cogemos con siete cuerdas, cuatro desde la parte superior a la placa, y otros tres a las patas de la base.
- Vemos que no es un buen método, y decidimos posponer la prueba a otro día, con mejores condiciones de visibilidad, y con una sujeción únicamente de las patas de la base.
- Obtención de las inercias en 3D de todas las partes del electrocóptero, es decir, del motor, de los brazos, del ESC, de la batería y de la placa. Vemos que el motor y los brazos son los elementos que mayor inercia generan sobre el centro de gravedad. Para este punto ver con más detalle el [apartado 3.9.2.1](#).
- Mediante el uso de un tacómetro obtenemos que las revoluciones a las que giran los motores en el punto de equilibrio están alrededor de 5060 rpm, pero con grandes desviaciones.

07/01/2015

- Prueba de funcionamiento<sup>32</sup>. Esta vez, realizamos un montaje un poco más complejo, ya que sujetamos con cuatro cuerdas cada una de las patas de la base. Además, ponemos una protección sobre el suelo, que consta de una colchoneta y una manta, que nos sirve para amortiguar los golpes que, debido a la inestabilidad, se producen del cuadricóptero con la superficie.
- Realizamos pruebas, y vemos que sin la acción de control es muy difícil estabilizarlo. Aun así, obtenemos un supuesto pulso de equilibrio que hace que nuestro electrocóptero se intente estabilizar en torno a un punto, pero muy lejano a las expectativas que teníamos.

<sup>31</sup> Segunda prueba del Electrocóptero: batería + siete cuerdas. En el aire.

<sup>32</sup> Tercera prueba del Electrocóptero: batería + cuatro cuerdas + colchoneta. Efecto suelo.

- Implementamos el PID en el microcontrolador, configurando el pulso de equilibrio con el obtenido en la prueba.

08/01/2015

- Puesta a punto de la maqueta: cambio de las hélices de los motores, ya que es mejor que las cuatro sean del mismo diámetro y no presenten deformaciones, así como de las sujeciones de los mismos. Esto lo hacemos después de hablar con el profesor de control automático, que nos dice que si el pulso enviado al motor es mayor que los habituales, posiblemente la holgura que tienen nuestros motores con las gomas, harán que las hélices salgan disparadas. De hecho, esto nos ocurrió al realizar una pequeña prueba<sup>33</sup> con él.

09/01/2015

- Programamos el código de arranque progresivo, ascenso, estabilización, descenso y parada de motores con el siguiente objetivo: arrancamos y ascendemos activando el PID hasta el equilibrio; cambiamos el pulso del PID para mantener el equilibrio, y, cuando queramos descender lo volvemos a modificar. Configuramos además los LEDs para cada una de estas funciones.
- Implementación en *MatLab* y *CooCox* del envío y recepción de las constantes del PID para así poder controlarlas desde la GUI de *MatLab* de una forma más cómoda. Esto se comenta de forma detallada en el [apartado 3.9.2.2](#).

14/01/2015

- Implementación del nuevo PID, más o menos estable, y que nos aporta un efecto Hovercraft, el cual se nos ocurre incluirlo en un botón de la GUI de *MatLab* que active dicha opción y quede más vistoso de cara a la presentación del proyecto.



- Probamos<sup>34</sup> a coger cada una de las patas con una cuerda, pero esta vez lo probamos estirando cuatro personas de una cuerda atada a la base de la estructura, y lo sujetamos con firmeza. Hay que tener en cuenta que todos

<sup>33</sup> Cuarta prueba del Electrocóptero: batería. Pulso máximo → Cambio de hélices.

<sup>34</sup> Quinta prueba del Electrocóptero: batería + PID + 4 cuerdas en el aire, misma altura → Inestabilidad.

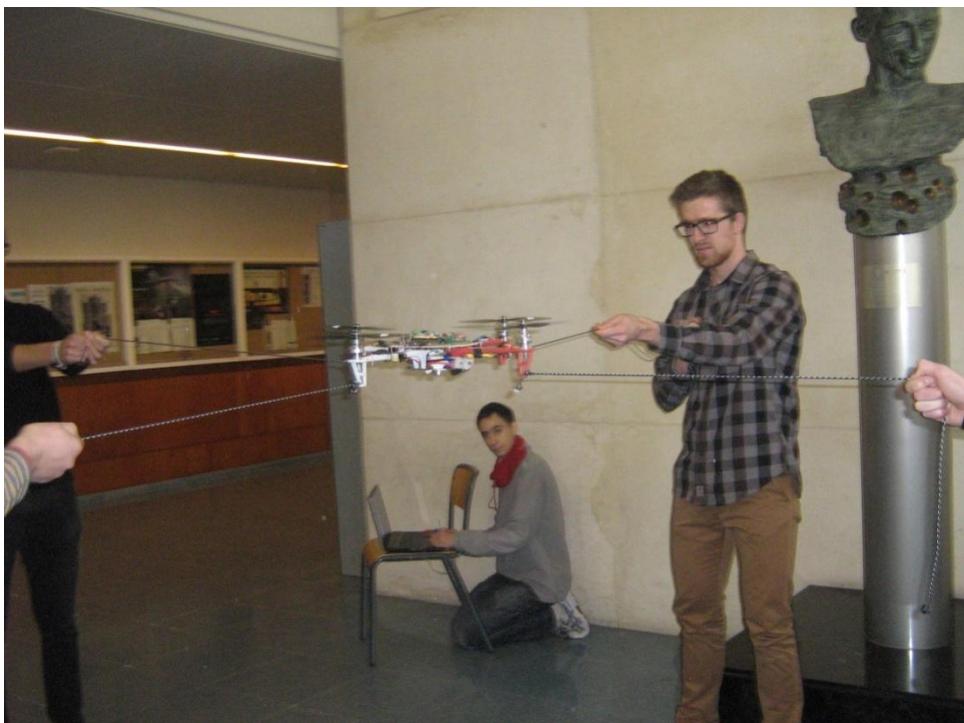
debemos mantener las cuerdas a la misma altura, con el fin de que no se produzca contacto entre la cuerda y las hélices de los motores. Vemos los primeros indicios de que el controlador PID funciona correctamente ya que no se producen golpes excesivos, y si empezamos a soltar poco a poco trata de autorregularse en torno al equilibrio.

15/01/2015

- Intentamos probar el PID implementado siguiendo el procedimiento del día anterior<sup>35</sup>, así como el modo Hovercraft.
- Acabado de la GUI de *MatLab*, incluyendo el indicador de potencia del PID.
- Impresión del póster en formato DIN-A1, para la presentación.

16/01/2015

- Últimos retoques.
- Exposición de Tecnología Electrónica. Hall de la Escuela Técnica Superior de Ingeniería del Diseño.



23/01/2015

- Configuramos la opción de graficar los pulsos del PID en la interfaz gráfica.

### 5.3 Recomendaciones

---

A continuación se nombran una serie de consejos que a nosotros como grupo de trabajo nos funcionaron y que de cara a afrontar un proyecto de estas dimensiones hay que tener en cuenta:

---

<sup>35</sup> Sexta prueba del Electrocóptero: batería + *PID* + Sujeción en el aire → Indicios de Estabilidad.

- Buena cooperación entre los componentes del grupo.
- Llevar un cuaderno donde apuntar todo lo que se avanza en un día. Aunque puede parecer costoso, finalmente resulta muy útil.
- No dejar para última hora las pruebas experimentales del cuadricóptero. Empezar con ellas al menos un mes antes de la presentación, pues pueden surgir problemas con el material no esperados, como relatamos en el anterior apartado.
- No limitarse a las horas de clase, sino quedar prácticamente todas las semanas una vez más como mínimo.
- Respecto a la estructura física del cuadricóptero:
  - Situar los *ESCs* debajo de las hélices del motor para tener buena refrigeración y evitar de este modo un sobrecalentamiento no deseado.
  - Contabilizar las revoluciones por minuto que dan cada una de las hélices con instrumentos apropiados, como puede ser un tacómetro digital, así como hacer una buena caracterización de cada uno de los motores y sus hélices. Conocer las revoluciones a las que giran los motores es fundamental, pues si se consiguen que giran todas al mismo régimen se evita la rotación sobre el eje vertical, que es lo que creemos que impide ajustar correctamente el cuadricóptero atándolo con cuerdas.
  - El anclaje de los motores no es muy efectivo. Se basa en un pequeño tornillo que debemos apretar con una llave *Allen*, por lo que recomendamos asegurarnos de que proporciona un buen apriete para evitar que salga disparado.
  - Sujetar las hélices a los ejes de los motores con unas sujetaciones diseñadas con este fin, pues, a diferencia de las gomas, evitan que haya movimiento relativo entre hélice y motor
  - En general, colocar el mínimo peso prescindible, es decir, no añadir demasiada cinta de sujeción, tornillería que sirva como contrapeso ni cualquier elemento que sobrecargue la estructura. Cualquier peso extra supone generar más fuerza con los motores, lo que se traduce en una disminución de la duración de la batería.
  - Buena sujeción del módulo inalámbrico a la placa de plástico, y buena sujeción de la placa de plástico a una de contrachapado que servirá para evitar la conducción eléctrica al chasis del cuadricóptero.
  - No dejar cables sueltos, sobre todo de cara a una prueba experimental.
- Respecto a la programación de la placa *STM32F3 Discovery*:
  - Guardar el código del proyecto y de la interfaz gráfica con varias copias de seguridad. Es posible que de un día para otro se modifiquen ciertos aspectos que más adelante cuestan de corregir si no se dispone de una copia de seguridad.

- Prestar atención al tamaño de variable utilizado en cada momento, puesto que la memoria del microcontrolador es limitada.
- No todos los temporizadores de la placa *Discovery* son iguales. Cada uno de ellos tiene diferentes modos que vienen explicados en el *datasheet* del microcontrolador:
  - TIM 1, TIM 2, TIM 3, TIM 4 y TIM 8 tienen 4 canales de comparación, captura y PWM.
  - TIM15 tiene 2 canales de comparación, captura y PWM.
  - TIM16 y TIM17 tienen 1 canal de comparación, captura y PWM.
- Por tanto, cada uno de ellos tiene sus propias funciones, como `RCC_APB1PeriphClockCmd`, que activa la señal de reloj de los temporizadores 2, 3, 4, 6 y 7, mientras que `RCC_APB1PeriphClockCmd` activa para los del bloque de periféricos 2 (1, 8, 15, 16 y 17).
- La circuitería de la placa saca sobre 3 V como estado alto de la señal, lo cual puede limitar alguna aplicación.
- La señal generada por el sonar saca 5 V, por lo que la línea de entrada al microcontrolador debe soportar esta tensión.
- Tener un muy buen conocimiento de la parte de control automático, focalizándose en la búsqueda de las constantes del *PID*. En este punto quizás lo más fundamental sea tener una buena ayuda del profesorado que domina dicha materia.
- No desesperarse nunca, aprender el porqué de los errores que se puedan dar e intentar siempre dar soluciones a los problemas que se presenten.

## 6 Conclusiones

---

Para finalizar esta memoria, nos gustaría señalar varios aspectos tanto de nuestro proyecto como de la asignatura en general.

Comenzamos hablando de la asignatura Tecnología Electrónica, una de las que más interés nos ha despertado a lo largo de estos dos años y medio del Grado en Ingeniería Aeroespacial. Ya desde el año pasado, cuando se nos explicó el funcionamiento de esta asignatura en Ingeniería Electrónica, tuvimos claro que nos íbamos a matricular en ella. Aquí hemos podido aplicar los conceptos que a lo largo del curso íbamos adquiriendo y pensamos que esto es fundamental en una asignatura que es totalmente práctica.

Esta asignatura se la recomendaríamos, como ya hemos hecho personalmente, a todo aquel con curiosidad por todos los dispositivos electrónicos que nos rodean en nuestro día a día. En ella se aplican tanto los conocimientos previos adquiridos en el anterior curso, pero, sobre todo, se aprenden interesantes conceptos nuevos, como puede ser la programación de *displays LCD* o la transmisión inalámbrica de datos. Después de lo vivido, volveríamos a elegir esta asignatura sin dudarlo.

Un aspecto negativo sería el hecho de haber comenzado con el proyecto prácticamente a finales de octubre. Aunque la teoría explicada en las primeras semanas del curso es necesaria, pensamos que en las primeras sesiones se podría dejar un cierto espacio de tiempo para comenzar ya el proyecto y poder dar los primeros pasos en esas primeras semanas. Asimismo, para decidir el proyecto, sería de ayuda contar con la experiencia de los alumnos del curso anterior, con el fin de orientar a proyectos similares de las dificultades encontradas y dar consejos.

Precisamente, la elección del proyecto fue uno de los aspectos más difíciles que se nos planteó. Aunque en un primer momento estuvimos dubitativos, la idea de construir un cuadricóptero siempre nos llamó la atención. Desde el momento en el que se nos dio el visto bueno comenzamos a documentarnos sobre los sensores, el controlador PID y, sobre todo, la placa *STM32F3 Discovery*, para la que fueron necesarias tres tardes extra de clase. Aunque consideramos que la incorporación de este nuevo microcontrolador ha permitido incluir notables mejoras respecto al proyecto realizado el anterior año, una posible pega es el hecho de no poder simular en el ordenador la programación, sino que es necesario implementarlo físicamente. No obstante, nos fue suficiente con el modo *debug* de *CooCox* y nos obligó a implementar la electrónica con una gran antelación al resto de proyectos.

No queremos olvidar que no hemos conseguido el objetivo de estabilizar el electrocóptero en el aire. No obstante, sí se ha cumplido la principal meta, que consistía en diseñar e implementar la electrónica completa que requiere un vehículo de estas características.

En cualquier caso, hubiésemos conseguido la estabilización o no, nuestros conocimientos de electrónica han aumentado en gran medida a lo largo de estos cuatro meses. Y no solo los de esta asignatura, sino que el proyecto nos ha ayudado a comprender el funcionamiento y la implementación de un controlador y mejorar nuestro nivel de programación, tanto en lenguaje en C como en *Matlab*, así como a trabajar en equipo.

Esta última característica, el trabajo en grupo, es imprescindible en proyectos de gran entidad como este puesto que el conocimiento que uno tiene ha de ser transmitido al resto de componentes del grupo y de este modo se agiliza muchísimo ante cualquier adversidad. Recordemos que “*trabajar en equipo divide el trabajo y multiplica los resultados*”.

Personalmente, hemos quedado satisfechos con el trabajo realizado y nos gustaría continuar con el proyecto, quizás con algo más de calma que a lo largo de este estresante último mes.

## 7 Bibliografía

- [1] ÁNGEL BENEDICTO & DANIEL ORIENT & NACHO VICIANO, Memoria del *Quadcopter 2.0*, Enero de 2014.
- [2] BRUCE BENEDICTUS & PEPIJN DE WINTER, *Quadcopter stabilization with complementary filter and fuzzy logic filter + PID motor control*, Enero de 2014.
- [3] RAFAEL MASOT PERÍS & MIGUEL ALCAÑIZ FILLOL, Transparencias de la asignatura *Tecnología Electrónica 2014-2015*, Recursos PoliformaT, Octubre de 2014.
- [4] RAFAEL MASOT PERÍS, Transparencias de la asignatura Ingeniería Electrónica, *Tema 08. PIC18F4520*, Recursos PoliformaT, Febrero de 2014.
- [5] Manual de usuario de la placa Discovery *UM1570 User Manual*, por STMicroelectronics, [www.st.com](http://www.st.com), 2013.
- [6] Manual de programa para STM32F3 y STM32F4 *PM0214 Programming manual*, por STMicroelectronics, [www.st.com](http://www.st.com), 2014.
- [7] Manual de referencia del microcontrolador STM32F303 *RM0316 Reference manual*, por STMicroelectronics, [www.st.com](http://www.st.com), 2014.
- [8] Hoja de datos del controlador STM32F303VB *Datasheet*, por STMicroelectronics, [www.st.com](http://www.st.com), 2014.
- [9] Hoja de datos del acelerómetro LSM303DLHC *Datasheet*, por STMicroelectronics, [www.st.com](http://www.st.com), 2013.
- [10] Hoja de datos del giróscopo L3GD20 *Datasheet*, por STMicroelectronics, [www.st.com](http://www.st.com), 2013.
- [11] Hoja de datos del sónar HC-SR04 *Datasheet*, CytronTechnologies.
- [12] RAFAEL MASOT PERÍS & MIGUEL ALCAÑIZ FILLOL, Teoría del seminario de la placa Discovery *Diseño de sistemas basados en microcontroladores con núcleo ARM Cortex*, Octubre de 2014.
- [13] *A Guide To using IMU (Accelerometer and Gyroscope Devices) in Embedded Applications*, [http://www.starlino.com/imu\\_guide.html](http://www.starlino.com/imu_guide.html), Diciembre 2009.
- [14] VICENTE MATA AMELA, Transparencias de Mecánica de 2º de Aeroespacial *Cinemática del Sólido Rígido*, Septiembre 2013.
- [15] JOAN VILA CARBÓ, Transparencias de Transporte, Navegación y Circulación Aérea de 3º de Aeroespacial *Técnicas e instrumentos de Navegación*, Noviembre 2014.
- [16] Interpretación de las señales de los sensores, CHRobotics LLC, <http://www.chrobotics.com>.
- [17] XAVIER BLASCO FERRAGUD & SERGIO GARCÍA-NIETO RODRÍGUEZ, Transparencias Control Automático, *Tema 07: El Regulador PID*, Recursos PoliformaT, Diciembre de 2014.