Trabajo de Final De Máster
Máster en Ingeniería Industrial

Control de posición y trayectoria de una pelota en un sistema tipo Ball And Plate

Autor: Ramón Iglesias Bayo

Tutor: Ignacio Alvarado Aldea

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Proyecto Fin de Máster
Máster en Ingeniería Industrial

Control de posición y trayectoria de una pelota en un sistema tipo Ball And Plate

Autor:

Ramón Iglesias Bayo

Tutor:

Ignacio Alvarado Aldea

Profesor titular

Dpto. de Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2019

Proyecto Fin de Carrera: Control de posición y trayectoria de una pelota en un sistema tipo Ball And Plate

Autor: Ramón Iglesias Bayo
Tutor: Ignacio Alvarado Aldea

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

Agradecimientos

Este proyecto supone el fin de una etapa muy importante en mi vida, y una despedida de mi ciudad, Sevilla. Por ello, me gustaría dedicarlo a todas aquellas personas que me rodean y me han apoyado durante toda mi etapa universitaria.

A mis compañeros de aventuras en la Escuela, con los que he compartido alegrías y suspensos, y que siempre me han ayudado en todo lo que he necesitado.

A mis padres, que siempre me han inculcado que con esfuerzo y constancia todo es posible.

A Ignacio, sin su motivación y ayuda nada de esto hubiese sido posible. Su implicación para que los alumnos conozcan, aprendan y se animen a realizar sus propios proyectos es admirable.

Resumen

En este proyecto se ha llevado a cabo la construcción y definición de la arquitectura de control de un sistema tipo Ball And Plate, un sistema consistente en el control de la posición y trayectoria de una pelota que se desplaza sobre una superficie plana. El objetivo principal es crear un equipo demostrativo con varias trayectorias predefinidas por defecto, así como que éste sirva de punto de partida para la creación de un equipo de carácter permanente que sirva para la prueba de distintos tipos de controladores.

Para obtención de datos en tiempo real del sistema se utiliza una pantalla táctil y una unidad de medidas inerciales, mientras que como actuadores se han utilizado dos motores paso a paso. Toda la información es gestionada por un Arduino Uno, en el cual se han implementado los controladores.

Se ha desarrollado también una interfaz gráfica en Python, en la que se puede observar en tiempo real la referencia y la posición actual de la pelota sobre la superficie de la pantalla.

Abstract

In this project, the construction and the definition of control architecture of a Ball and Plate system were carried out, where position and trajectory of a ball which rolls in a plate can be controlled. The aim of this project is to create a demonstration Ball and Plate system with several predefined trajectories, as well as a starting point of the design of a permanent device, to test different control algorithms.

In order to obtain real time data from the system, a four-wire touchscreen and an Inertial Measurement Unit (IMU) were used whereas as actuators, two stepper motors were used. This real time data is acquired by an Arduino Uno, where PID controllers were implemented.

A Graphical User Interface (GUI) was also developed in Python, where the real-time position and the reference of the ball is plot.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

A^*	Conjugado
c.t.p.	En casi todos los puntos
c.q.d.	Como queríamos demostrar
■	Como queríamos demostrar
e.o.c.	En cualquier otro caso
e	número e
IRe	Parte real
IIm	Parte imaginaria
sen	Función seno
tg	Función tangente
arctg	Función arco tangente
sen	Función seno
$\sin^x y$	Función seno de x elevado a y
$\cos^x y$	Función coseno de x elevado a y
Sa	Función sampling
sgn	Función signo
rect	Función rectángulo
Sinc	Función sinc
∂y ∂x	Derivada parcial de y respecto
$x_\circ$	Notación de grado, x grados.
Pr(A)	Probabilidad del suceso A
SNR	Signal-to-noise ratio
MSE	Minimum square error
:	Tal que
<	Menor o igual
>	Mayor o igual
\	Backslash
⇔	Si y sólo si

1 DESCRIPCIÓN DEL SISTEMA

En primer lugar, en esta sección se realizará una descripción del sistema elegido, conocido popularmente como un *Ball and Plate*.
Un sistema *Ball and Plate* es aquel en el que se desea controlar la posición o trayectoria de una pelota que se desplaza libremente sobre una superficie. Con el fin de actuar sobre el estado del sistema, la plataforma puede ser inclinada mediante el uso de dos o más motores. La posición de la pelota en cada instante es usada como realimentación del sistema, de forma que se pueda calcular el error entre la posición actual de la pelota y la referencia.
Existen diversas opciones para la implementación de este sistema. La posición de la pelota, por ejemplo, puede obtenerse usando distintos métodos como una pantalla táctil (Figura 1 izquierda) o una cámara y un procesado de la imagen capturada (Figura 1 derecha). A su vez, existen también distintos actuadores que pueden ser usados: motores DC con encoders, servomotores, motores paso a paso, etc.

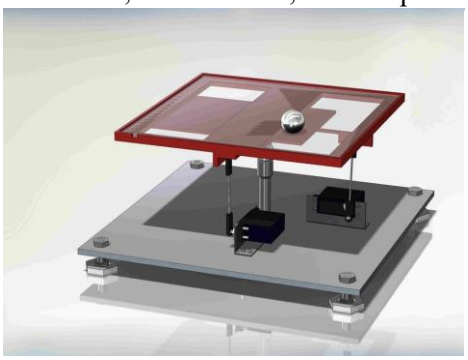


Figura 1 Distintas implementaciones del Ball And Plate [1] [2].

1.1 Descripción del sistema construido

El sistema construido está formado por una pantalla táctil unida a un tablero mediante una rótula esférica, la cual permite giros en torno a 3 ejes (impidiendo el desplazamiento relativo entre el tablero y la superficie), y por ello, eliminando 3 grados de libertad. Además de estas restricciones de desplazamiento, también está restringido el giro en torno al eje Z mediante una pequeña barra fijada en la pantalla (Figura 3). Esta barra fijada a la superficie tiene permitido el movimiento respecto a la base en el eje vertical, reduciendo los grados de libertad del sistema a sólo dos: la rotación de la pantalla en torno al eje X y el eje Y.
En el sistema, existen dos elementos de tomas de datos:

- Una pantalla táctil resistiva de 4 hilos, la cual hace de superficie sobre la que se desplazará la pelota, permitiendo a su vez conocer la posición de la pelota en cada instante.
- El uso de una unidad de medidas inerciales (IMU), la cual permite conocer el ángulo actual de la superficie del sistema.

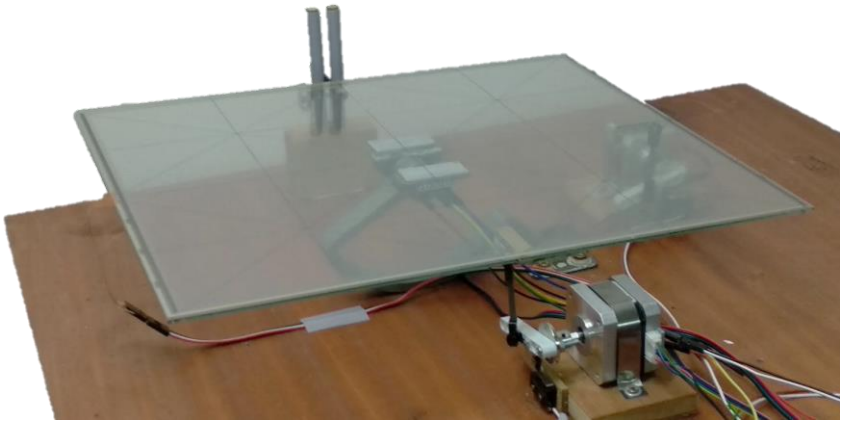


Figura 2 Sistema Ball And Plate construido.

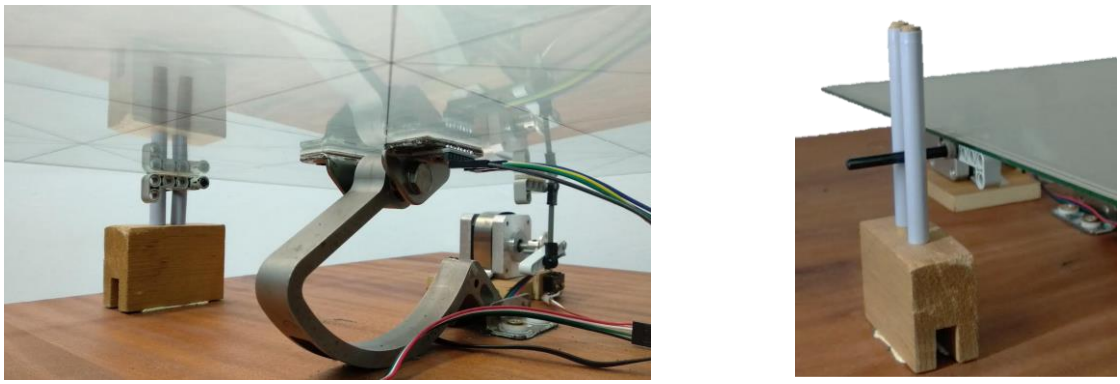


Figura 3 Detalle de la rótula esférica y la restricción de giro en Z.

Como actuadores del sistema se encuentran dos motores paso a paso, usados para inclinar la pantalla en dos ejes. En los siguientes subapartados se realiza una descripción más detallada de los distintos elementos usados en su construcción. La descripción realizada de algunos elementos como la unidad de medidas inerciales, el microcontrolador o los motores paso a paso se encuentran basados en la descripción ya realizada en el trabajo de final de grado "Desarrollo de un robot autoequilibrado basado en Arduino con motores paso a paso" realizado por José Antonio Borja Conde [3].

1.1.1 Pantalla táctil resistiva

La bandeja sobre la que se desplaza la pelota es una pantalla resistiva de 4 hilos del fabricante Haina Touch technology Co., Ltd, con un tamaño de 17 pulgadas de diagonal. En Figura 4 se puede encontrar una descripción técnica más detallada proporcionada por el fabricante.

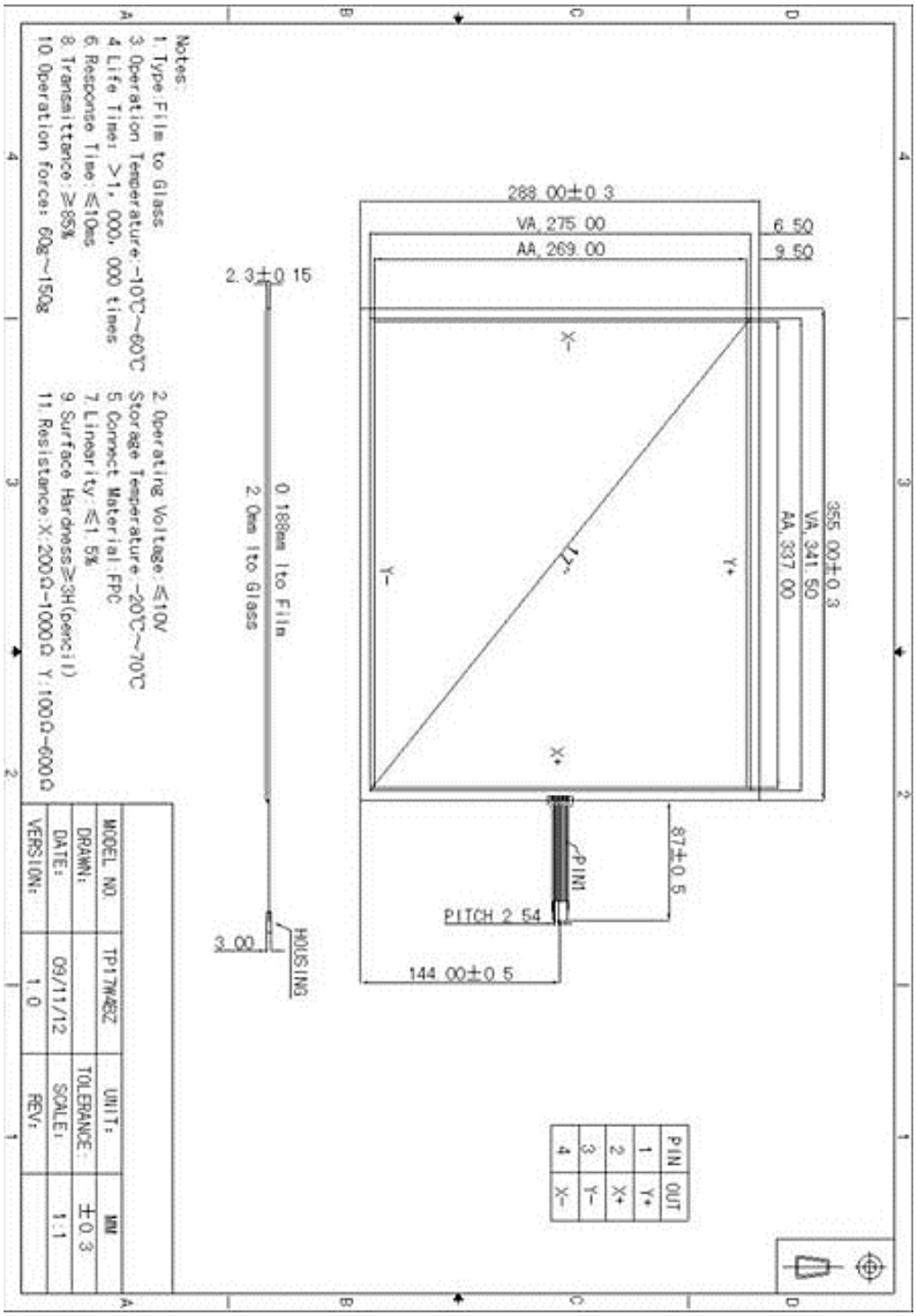


Figura 4 Descripción técnica de la pantalla táctil resistiva procedente del fabricante [4].

Como se explica en [5], una pantalla táctil resistiva está formada habitualmente por un soporte de cristal, que hace de capa fija de base, y una capa flexible de polietileno sobre ella. Estas dos capas están revestidas de forma uniforme de un material resistivo, habitualmente de óxido de indio y estaño. Encima de la capa fija de cristal, y sobre el material resistivo, se disponen una serie de elementos separadores en forma de puntos cuya función es mantener separación entre la capa inferior y la capa superior. Cuando se realiza un toque en la pantalla táctil, la capa flexible se flexiona, produciendo un contacto con la capa inferior.

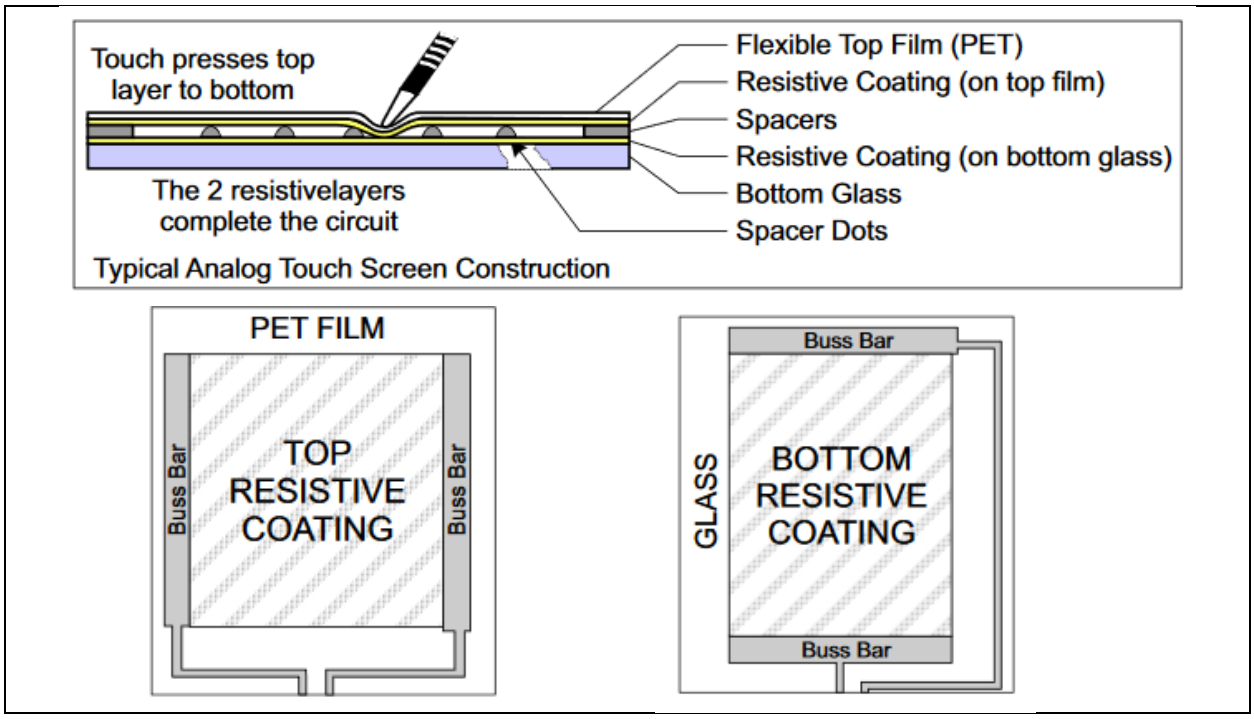


Figura 5 Figura explicativa de los elementos que conforman una pantalla táctil [5].

Esta posición del toque puede ser leída fácilmente con un controlador Arduino si se considera cada eje de la pantalla como un divisor resistivo. Para ello, si por ejemplo se quiere realizar la lectura de la posición del toque en el eje X, se pondría 5V en el extremo izquierdo, y el eje derecho conectado a tierra. Cuando se produzca un contacto entre ambas capas de la pantalla táctil, la tensión resultante del divisor resistivo puede ser leída desde uno de los extremos del otro eje (en este caso el eje Y), usando una función de tipo *analogRead()*.

En capítulos posteriores (3.4.1) se tratará de una forma más extensa la forma en la que se realizan las medidas de la pantalla.

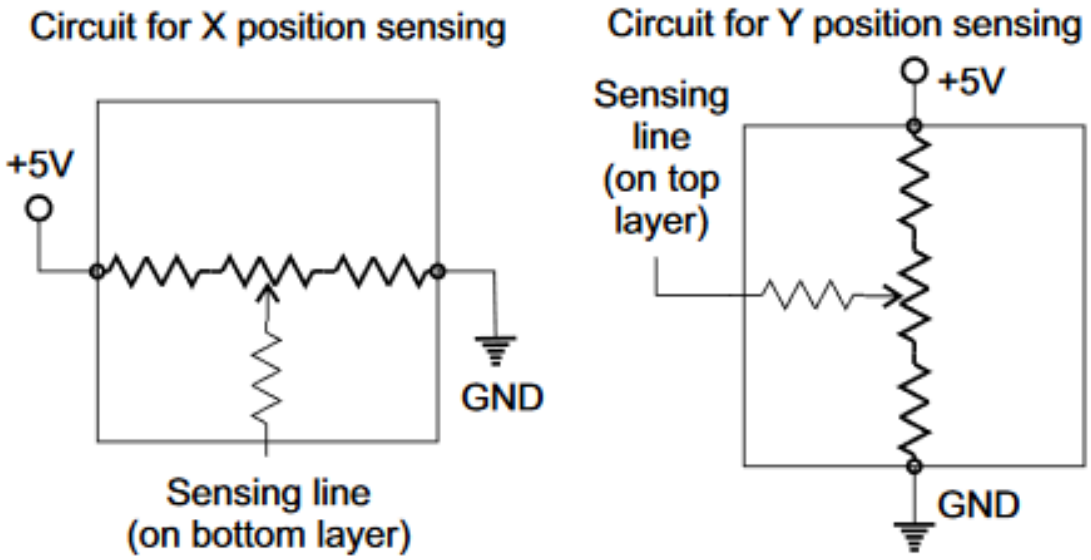


Figura 6 Divisor resistivo equivalente cuando se realiza un toque en la pantalla resistiva [5].

1.1.2 Motores paso a paso

Como actuadores del sistema se han usado dos motores paso a paso, con el fin de controlar el ángulo de la pantalla respecto a la horizontal. La elección de este tipo de motores se debe a los siguientes factores:

- Bajo coste.
- Facilidad de control con trenes de pulsos desde un microcontrolador, mediante el uso de drivers comerciales. Esto hace que se reduzca al mínimo el uso de pines del microcontrolador.
- Posibilidad de realizar movimientos muy precisos de amplitud constante, realizados cada vez que se emite un pulso al driver.

En este proyecto se ha usado el motor paso a paso Nema 17 17HD34008-22B de 2 fases y cuatro hilos, el cual es muy usado para aplicaciones de impresión 3D. Un resumen de las características técnicas extraído de [3] se puede encontrar a continuación:

Características	Medidas y unidades
Corriente nominal	1.5 A DC
Tensión nominal	3.45 V
Ángulo de avance por paso	1.8 ± 5%
Pasos completos por vuelta	200
Par de fricción	12mN.m
Par de mantenimiento	≥300mN.m (I = 1.5 A)
Máx. Frecuencia de arranque sin carga	≥1500pps
Máx. Frecuencia de funcionamiento sin carga	≥8000pps
Peso del motor	230 g
Tamaño (L x W x H)	42 x 42 x 34 (mm)

Tabla 1 Resumen de las especificaciones técnicas de Nema 17 17HD34008-22B.

Los motores de dos fases (como el usado en este proyecto) tienen dos bobinas en su interior, que al ser energizadas de forma secuencial y apropiada consiguen generar movimientos angulares discretos del rotor. Gracias a que los movimientos son discretos, también es posible por lo tanto conocer la situación del rotor del motor en cada instante, siempre que se tome una posición de referencia como 0, y se considere que no se pierden pulsos en ningún momento. Esta posición de referencia viene determinada por un final de carrera en uno de los extremos del rango de movimiento del motor, como se puede apreciar en Figura 4.

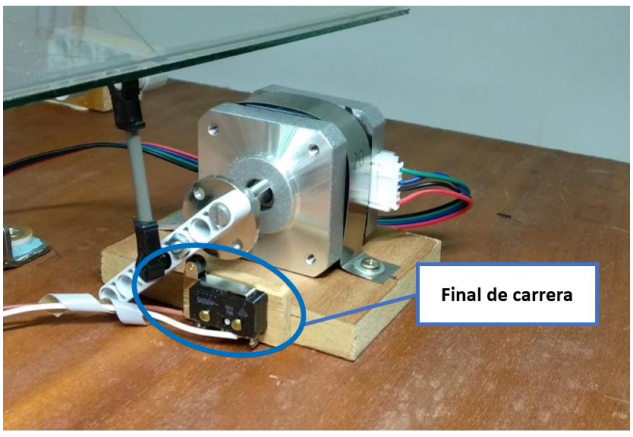


Figura 7 Final de carrera en la posición de inicio de los motores paso a paso.

Cada bobina tiene conectada dos de los cuatro hilos del motor, de forma que la energización necesaria puede ser controlada desde un driver o un microcontrolador.

A continuación, se presenta un ejemplo ilustrativo [6] en el que se explica la secuencia seguida para producir el movimiento correspondiente a un paso o un semipaso.

Ejemplo 1–1. *Secuencia de fases necesaria para el caso concreto de un motor con paso angular de 90º y un semipaso de 45º para girar 180º.*

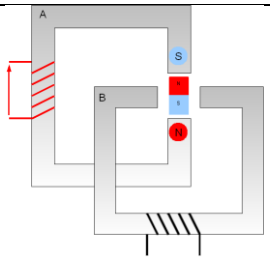
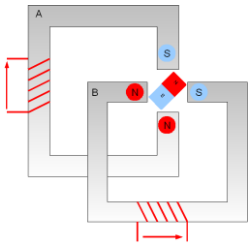
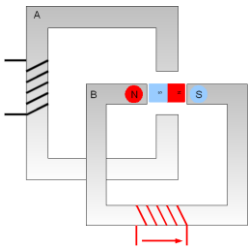
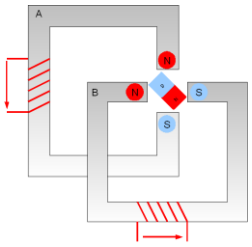
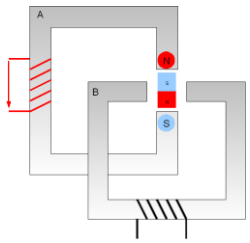
Paso	Terminal 1 Bobina A	Terminal 2 Bobina A	Terminal 1 Bobina B	Terminal 2 Bobina B	Imagen
Paso 1	+Vcc	-Vcc	-	-	
(Semi-)Paso 2	+Vcc	-Vcc	+Vcc	-Vcc	
Paso 3	-	-	+Vcc	-Vcc	
(Semi-)Paso 4	-Vcc	+Vcc	+Vcc	-Vcc	
Paso 5	-Vcc	+Vcc	-	-	

Tabla 2 Secuencia de fases correspondiente al ejemplo 1–2.

Usando la estrategia anteriormente explicada también podríamos conseguir realizar unidades más pequeñas a la mitad de un paso, como ¼ de paso o 1/8 de paso. Para ello, es necesario usar la técnica del microstepping, en la que en vez de encender o apagar las bobinas por completo, se puede variar la corriente aplicada a cada bobina aplicando una función senoidal discretizada, como explica Luis Llamas en [7]:

"Si pudiéramos aplicar a ambas bobinas dos señales eléctricas senoidal perfectas desfasadas 90º conseguiríamos un campo magnético rotatorio perfecto en el interior del motor. Por supuesto el controlador digital no genera valores analógicos perfectos, sino valores discretizados («a saltos»), por lo que la señal eléctrica que aplica es igualmente una función senoidal discretizada.

El resultado es un campo magnético un campo magnético rotativo con un paso inferior al paso nominal, que

depende del número de niveles de discretos que podemos emplear en las señales de excitación de la bobina."

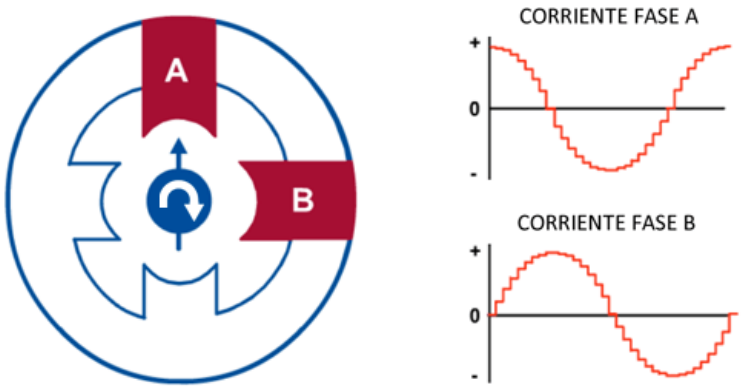


Figura 8 Función senoidal discretizada aplicada a las bobinas, extraído de [7].

En este proyecto la excitación de los motores será llevada a cabo con el driver comercial EasyDriver, el cual cuenta con la posibilidad de realizar la técnica del microstepping. A continuación, en el siguiente apartado se explican los aspectos más destacados de éste.

1.1.3 EasyDriver

Para el manejo de los motores paso a paso de forma sencilla desde el microcontrolador se usa el driver diseñado por Brian Schmalz conocido como EasyDriver v4.4 [8], el cual es una implementación del encapsulado A3967SLB de Allegro [9] que contiene toda la electrónica necesaria para su uso de forma sencilla.

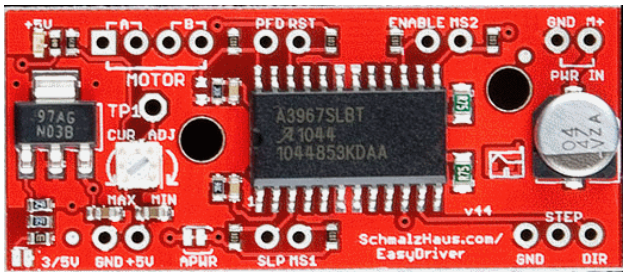


Figura 9 Easy Driver v4.4.

Características	Medidas y unidades
Tensión de alimentación de carga	Hasta 30 V
Corriente de salida continua	±750 mA
Corriente de salida de pico	±850 mA
Rango de temperatura de operación	-20°C a +85°C
Rango de tensión de alimentación de la lógica	3.0 a 5.5 V
Frecuencia máxima en el pin STEP	500kHz

Tabla 3 Especificaciones técnicas del driver A3967SLB de Allegro [9]

Este encapsulado permite operar motores bipolares en modo de pasos completos o 1/2, 1/4 o 1/8 de paso. Su característica principal es el traductor implementado en el encapsulado que, con solo realizar un pulso en la entrada STEP, realiza la energización necesaria en las bobinas para que el motor paso a paso avance un pulso (o 1/2, 1/4 o 1/8 de paso, dependiendo de otras dos entradas que fijan el modo de funcionamiento del driver). De forma muy general, se podría decir que es este traductor el que establece a cada convertidor DAC (digital to analogic converter) la tensión con la que el puente H debe energizar cada bobina (Figura 10), consiguiendo en

ambas bobinas dos señales senoidales discretizadas y desfasadas 90º entre ellas, como se describió en la Figura 8.

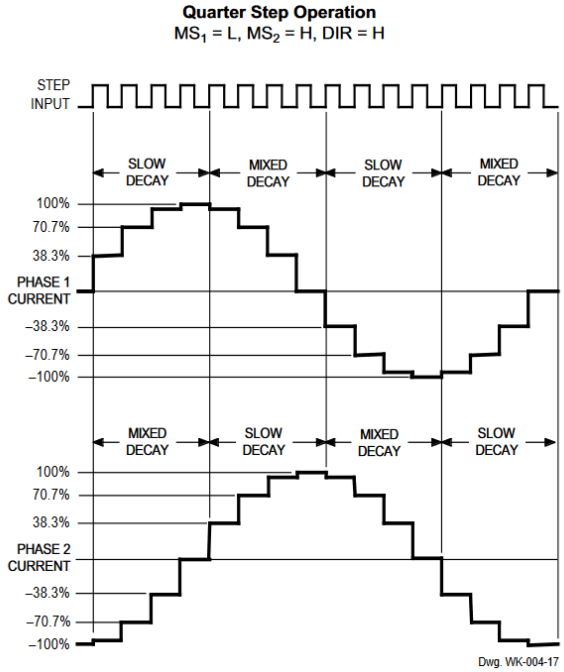


Figura 10 Energización de cada bobina usando el modo de 1/8 de paso por cada pulso en STEP.

A continuación, se explica brevemente el uso de los pines en el EasyDriver v4.4 que son interesantes para el proyecto, pudiendo encontrar más información sobre el resto de pines en [8] y [9].

MS1	MS2	Resolución
Low	Low	Pasos completos
High	Low	½ de paso
Low	High	¼ de paso
High	High	1/8 de paso

Tabla 4 Resolución de los micropasos [9]

Pin	Descripción
MS_1 y MS_2	Permiten configurar el avance por cada flanco de subida en el pin STEP, en función de la Tabla 4.
DIR	Determina el sentido de giro del motor.
STEP	Un flanco de subida produce el incremento de un pulso en el motor.
ENABLE	Entrada con lógica negada. Si el valor es *high* las salidas del driver son desactivadas.
GND – M+	Alimentación del driver, pudiendo ir hasta los 30 V.
SLP (sleep)	Entrada con lógica negada. Si el valor es *high* se minimiza el consumo de energía, pensado para cuando el driver no se encuentre en uso.

Tabla 5 Descripción de los pines para la implementación.

En el data sheet se encuentra más información sobre las restricciones temporales existentes en el encapsulado, las cuales vienen descritas en Figura 11:

- La frecuencia máxima de cambio en el pin STEP está fijada a 500 KHz, siendo esto igual a un periodo de cambio de 2 µs (teniendo en cuenta el flanco de subida y bajada).
- En el pin DIR, su cambio de valor debe realizarse como mínimo 200ns antes del pulso en STEP.
- Las restricciones de tiempo en los pines MS_1 y MS_2 no aplican, pues los valores de estos pines se fijan constantes con el fin de tener la mayor resolución posible en el control del movimiento (1/8 de paso por cada pulso en step).
- La restricción de tiempo en el pin SLEEP tampoco aplica, al no ser usado en la aplicación.

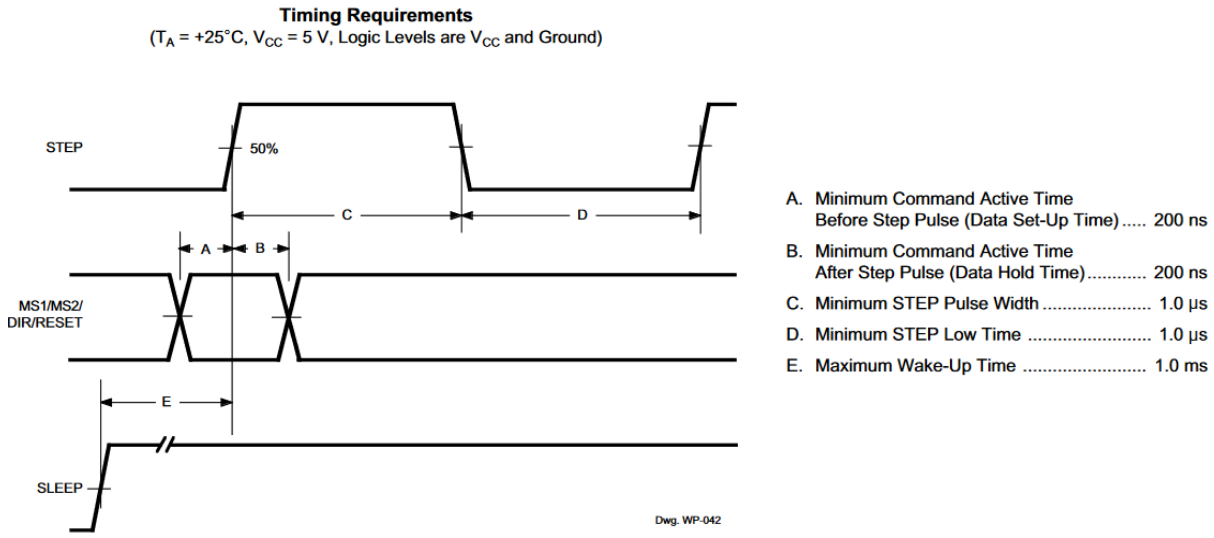


Figura 11 Restricciones temporales del driver A3967SLB de Allegro [9].

1.1.4 Arduino Uno

Con el fin de realizar el control en bucle cerrado de todo el sistema se ha implementado un Arduino Uno [10], el cual es una placa basada en un microcontrolador ATmega328P. Ésta contiene todo lo necesario para poder usar el microcontrolador de forma sencilla, incluyendo entre otros 14 pines de entrada/salida, 6 pines de entradas analógicas, conexión USB, una entrada ICSP que permite una programación sencilla del microcontrolador desde el ordenador, un botón de reset o una conexión para alimentar la placa con un adaptador AC-DC.

Figura 12 Arduino Uno revisión 3 [10].

A continuación, se presentan una serie de especificaciones técnicas del microcontrolador ATM328P [11] implementado en el Arduino:

- Microcontrolador de alto rendimiento de 8 bits basado en RISC[1]

[1] Acrónimo de Reduced Instruction Set Computer

- Memoria Flash ISP de 32KB con capacidad de escritura y lectura conjunta
- 1024 bytes de memoria EEPROM
- 2048 bytes de memoria SRAM
- 23 pines de entrada y salida de propósito general
- 32 registros de propósito general
- 3 timers con posibilidad de modos de comparación e interrupciones internas y externas.
- Posibilidad de comunicaciones UART[2]
- Interfaz serie SPI[3]
- 6 canales para convertidores analógicos/digitales de 10 bits

En posteriores secciones (sección 3.1 Introducción a las interrupciones en Arduino) se tratará de forma más extendida el uso de las interrupciones para la programación del controlador.

1.1.5 IMU MPU-6050

La IMU MPU-6050 [12] es una unidad de medidas inerciales para aplicaciones que requieran bajo consumo y alto rendimiento a un bajo coste. Ésta consta de un acelerómetro y un giroscopio de 3 ejes, permitiendo obtener medidas de velocidad y aceleración en 3 ejes (6 grados de libertad). Esta IMU también consta de un procesador digital de movimiento (DMP, acrónimo de Digital Motion Processor), el cual incorpora el firmware *InvenSense's MotionFusion™*. Este firmware incluye algoritmos capaces de procesar los datos del acelerómetro y giroscopio, así como herramientas de calibración de los sensores o incluso detección de gestos.

En principio, se puede llegar a pensar que una unidad de medidas inerciales como ésta no sería necesario al contar con los motores paso a paso, ya que éstos pueden ser usados para conocer el ángulo de la superficie sobre la que desliza la pelota respecto a la gravedad. Para ello, sólo sería necesario tener en cuenta el número de pasos dados (incrementos angulares fijos) respecto a una posición de inicio.

En la primera versión de este proyecto se realizó el control de la trayectoria usando el ángulo de la superficie calculada de esta forma. No obstante, el control de la trayectoria de la pelota resultante dependía en gran medida de la inclinación de la superficie donde se encontrara apoyado el sistema. Es por ello que se optó por incluir un control de bajo nivel más rapido que controlara el ángulo de inclinación de la superficie por la que se desplaza la pelota. Este control de bajo nivel usa como medida del sistema real las medidas obtenidas por la IMU.

La primera versión del esquema de control puede ser encontrado en 2.1 Esquema de control del sistema, donde se trata con mayor profundidad.

Algunas especificaciones técnicas de la IMU usada que se han considerado de utilidad se presentan a continuación [13]:

- Comunicación I^2C con una velocidad máxima de 400kHz
- Giroscopio con rango programable por el usuario (±250, ±500, ±1000 y ±2000 °/s)
- Acelerómetro con rango programable por el usuario (±2g, ±4g, ±8g y ±16g)
- 6 convertidores analógicos digitales de 16 bits para las salidas del giroscopio y acelerómetro.

[2] Acrónimo de Universal Asynchronous Receiver-Transmitter
[3] Acrónimo de Serial Peripherical Inteerface

- Buffer de 1024 bytes para una recogida de datos de forma conjunta en el caso de un modo de bajo consumo del sistema
- DMP (Digital Motion Processor) con algoritmos de procesamiento de datos, los cuales permiten reducir la carga de procesamiento al sistema.

Es habitual que la IMU se encuentre implementada en un módulo comercial (como el GY-521) en el cual se incorpora toda la electrónica necesaria para que sea fácilmente usado desde plataformas como Arduino. Este módulo también permite conectividad por bus I2C, lo cual permite obtener de forma sencilla los datos desde el Arduino.

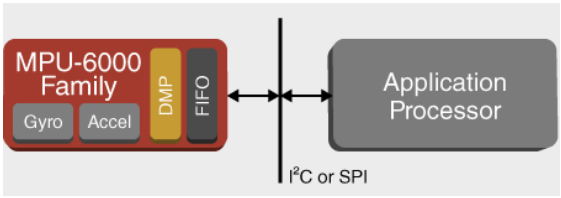


Figura 13 Diagrama de bloques de la comunicación del MPU6050 con el procesador [12].

1.1.5.1 Introducción a la comunicación I^2C

A continuación, se realiza una sencilla introducción a la comunicación I^2C en Arduino, la cual ha sido basada en las ideas planteadas por Luis Llamas en [14].

El estándar I2C (inter-Integrated Circuit) es un estándar de comunicaciones desarrollado por Philips en sus inicios, aunque posteriormente fue adoptado por el resto de los fabricantes. Su principal característica es la comunicación de datos a través de dos señales: CLK (señal de reloj que coordina el envío de información) y SDA (señal que se encarga de transmitir los datos). Es por ello un sistema síncrono, en el que durante la comunicación todos los dispositivos conectados al bus comparten la misma señal de reloj.

La arquitectura de la comunicación es de tipo maestro-esclavo, en la que el maestro comienza la comunicación, y realiza escrituras o lecturas en cualquiera de los esclavos. Cada uno de ellos dispone de una dirección en el bus única, que identifica al esclavo involucrado en la comunicación con el maestro.

Con el fin de llevar a cabo la comunicación entre el maestro y el esclavo de 1 byte, se usa una trama definida en el estándar (Figura 14).

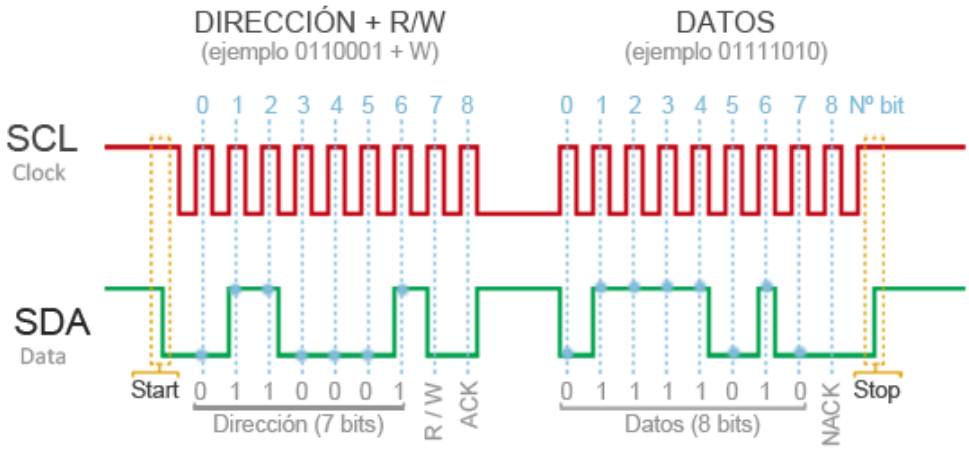


Figura 14 Trama de comunicación de 18 bits para la transmisión de un dato de 8 bits.

Ésta consta de:

- Dirección del esclavo de 7 bits.

- Bit que indica si se trata de una escritura o lectura.

- Bit de validación.

- Dato de 8 bytes que es enviado o recibido.

- Bit de validación.

Como ventajas de este estándar de comunicación se encuentran el número reducido de pines necesarios (sólo 2) y la existencia de mecanismos para saber si el mensaje ha llegado al destinatario (bit de validación). No obstante, como desventajas se encuentra su velocidad de transmisión de datos (media – baja), así como que no existen mecanismos para comprobar la integridad de los datos recibidos (por ejemplo, un mecanismo de tipo CRC).

1.1.5.2 Cálculo de la inclinación de la pantalla

A partir de las medidas obtenidas por el acelerómetro y el giroscopio de la IMU es posible obtener la inclinación del sensor; esto permite que sea usado para conocer la inclinación de la superficie donde se desliza la pelota. Para ello, se ha situado el sensor en la parte inferior de la pantalla, estando apoyado sobre la junta entre la articulación que une la base con la pantalla y la propia pantalla.

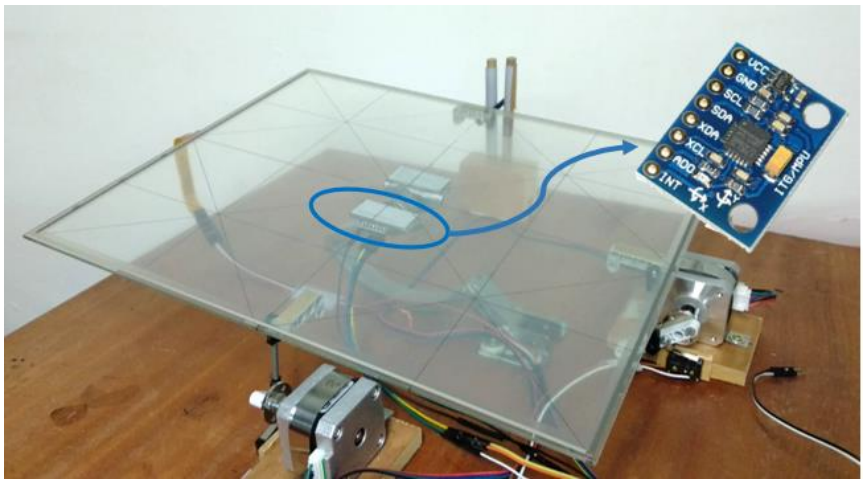


Figura 15 Módulo GY-521 ubicado en la unión de la pantalla con la articulación.

Existen dos formas distintas de realizar el cálculo de los ángulos de orientación de la IMU, los cuales corresponden con la inclinación de la superficie en el eje X y eje Y. El primer método que se explica es mediante el acelerómetro, mientras que el segundo usa las medidas obtenidas del giroscopio. Por último, se explican otros métodos que permiten obtener a partir de ambas unas medidas filtradas y con menor ruido. Todas las explicaciones que se presentan han sido extraidas de [15].

1.1.5.2.1 Obtención de ángulos a partir del acelerómetro

Con el fin de obtener la inclinación de la pantalla se pueden usar las medidas del acelerómetro, bastando sólo unas sencillas relaciones de trigonométricas para el cálculo. Se sabe que la medida de la gravedad es de $9.8\frac{m^2}{s}$ de forma perpendicular al suelo. Por ello, si se alinea por ejemplo el eje Z del MPU-6050 con la perpendicular del suelo, se obtendrán como medidas 9.8 en el eje Z, mientras que en el resto de ejes se obtendrá un valor cercano a 0. De la misma forma, si se coloca en una posición de 90º el sensor respecto a la perpendicular al suelo, se obtendrá de forma equivalente $\frac{9.8m^2}{s}$ en el eje alineado (en el caso de la Figura 16 en el eje X).

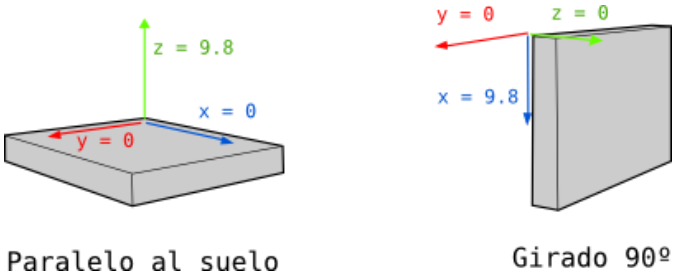


Figura 16 Medidas obtenidas cuando se coloca el IMU de forma paralela y perpendicular al suelo [15].

Siguiendo esta lógica, si se sitúa la IMU en una posición intermedia entre ambas posiciones, se puede deducir que el ángulo de la IMU con respecto a la gravedad está relacionado con las medidas del acelerómetro en el resto de los ejes. Estos ángulos corresponderían en este proyecto a la inclinación de la pantalla en los ejes X e Y.

$$Ángulo\ X = \text{atan}\left(\frac{x}{\sqrt{y^2 + z^2}}\right) \tag{1-1}$$

$$Ángulo\ Y = \text{atan}\left(\frac{y}{\sqrt{x^2 + z^2}}\right) \tag{1-2}$$

No obstante, estas medidas obtenidas a través de estas expresiones no pueden ser usadas en los algoritmos de control, ya que es habitual que las medidas obtenidas por el acelerómetro presenten bastante ruido y tengan grandes márgenes de error. Además de ello, debido a que las medidas son aceleraciones en cada eje, hay muchos casos en los que se podría llegar a pensar que se ha producido un giro, cuando en realidad lo que ha habido es un desplazamiento con aceleración en un eje concreto.

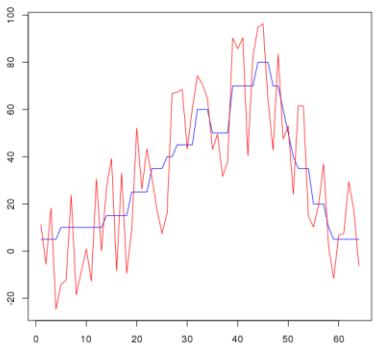


Figura 17 Apariencia típica de medidas obtenidas con un acelerómetro [15].

Por ejemplo, si se desplaza la IMU de un punto a otro con una aceleración determinada, esta medida de aceleración que se obtiene podría considerarse de forma errónea como un giro del dispositivo.

1.1.5.2.2 Obtención de ángulos a partir del giroscopio

También es posible obtener una medida estimada de la variación de los ángulos usando las medidas de los giroscopios. Éstos miden la velocidad angular ($°/s$), de forma que, estableciendo en un instante inicial una posición de referencia como 0º, es posible conocer el ángulo actual si se realizan medidas ($GiroscopioY$) cada cierto intervalo de tiempo Δt.

$$Ángulo\ Y = ÁnguloY_{anterior} + GiroscopioY \ \ \Delta t \tag{1-3}$$

A diferencia de las medidas obtenidas con el acelerómetro, éstas suelen ser bastante precisas. No obstante, debido a que el cálculo del ángulo se basa en una suma de incrementos, a largo plazo la suma de los pequeños errores implícitos en la medida produce la deriva de ésta, dando lugar a que el valor medido se aleje cada vez más de la realidad.

1.1.5.3 Filtrado de las medidas

Como se ha visto, en la práctica estas medidas están lejos de poder ser usadas como entradas para los controladores implementados en el Arduino (cada uno de los métodos explicados anteriormente presentan algunos problemas que no las hacen adecuadas para su uso). Con el fin de conseguir una medida filtrada, existe un gran número de estrategias de filtrado que pueden ser usadas.

Entre los filtros que se podrían aplicar, se contemplaron en un principio el filtro de Kalman y el filtro complementario, al ser los más utilizados en proyectos implementados en Arduino, así como en otras aplicaciones como sistemas de navegación de UAVs [16]. No obstante, de forma posterior, se decidió usar el propio filtrado que incluye el propio DMP de la IMU, de forma que se libera al Arduino de la carga computacional.

1.1.5.3.1.1 *Filtro de Kalman.*

Este filtro es uno de los más conocidos en la ingeniería, debido a sus usos en sistemas de navegación de aeronaves y UAVs[4], aunque también es usado en otros ámbitos como la estimación demográfica, procesamiento de imágenes, etc. Como se puede encontrar en [17], consiste en la estimación de variables de estado de un sistema a través de las salidas observables del mismo, las cuales presentan ruido blanco. De forma general, se puede decir que este filtro realiza una predicción del vector de variables de estado en función de la dinámica del sistema, y en cada instante la corrige mediante un análisis estadístico de las variables observables.

El filtro de Kalman proporciona buenos resultados, aunque su implementación es algo más costosa que la del filtro complementario. A diferencia de éste, el filtro complementario presenta una implementación más sencilla, unos resultados muy parecidos en términos de error, y una velocidad de convergencia mayor como se concluye en [16].

1.1.5.3.1.2 *Filtro complementario*

El filtro complementario se plantea como una simplificación al Filtro de Kalman, de forma que mantiene la idea principal, aunque se elimina el análisis estadístico para facilitar su implementación.

$$\theta = A\left(\theta_{prev} + \theta_{giroscopio}\right) + B\,\theta_{acelerómetro} \tag{1-4}$$

Siendo

θ Ángulo $filtrado$

θ_{prev} Ángulo $previo$

$\theta_{giroscopio}$ Ángulo $calculado\ con\ el\ giroscopio$

$\theta_{acelerómetro}$ Ángulo $calculado\ a\ partir\ del\ acelerómetro$

Siendo A y B unas constantes, las cuales se encargan de ponderar la influencia del acelerómetro y del giroscopio. La idea principal es obtener la tendencia del sistema a partir del dato con mayor precisión (giroscopio) aunque corrigiendo la deriva de éste con los datos obtenidos con el acelerómetro. Por ello, unos valores típicos para este filtro suelen ser en torno a $A \sim 0.98$ y $B \sim 0.02$.

El uso de este filtro puede entenderse mejor con el ejemplo planteado por Luis Llamas (véase Figura 18) en [18].

[4] Unmanned aerial vehicle, aeronaves no tripuladas.

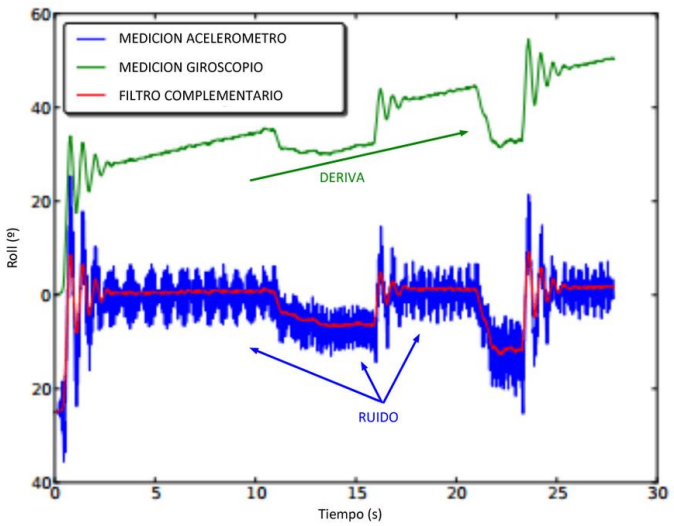


Figura 18 Medidas obtenidas con el acelerómetro, con el giroscopio, y su combinación en el filtro complementario [18].

1.1.5.3.1.3 *Uso del DMP*

Como se explicó al principio de este apartado, la IMU MPU-6050 posee un procesador digital de movimiento (DMP). Este procesador tiene implementados unos algoritmos capaces de fusionar los datos obtenidos por el acelerómetro y el giroscopio, con el fin de reducir los errores inherentes de cada sensor. Esto, por ello, permite realizar medidas de ángulos sin necesidad de realizar costosas implementaciones en términos computacionales de filtros de ningún tipo.

Con el fin de acceder a los datos del DMP, se ha usado una librería creada por Jeff Rowberg llamada i2cdevlib [19]. En secciones posteriores (3.5), se tratará cómo se ha usado esta librería para la lectura de los datos procesados por el DMP.

1.2 Conexionado de los elementos

Con el fin de realizar el conexionado de los elementos, se ha llevado a cabo el desarrollo de un escudo de Arduino, con el cual se reduce al mínimo el número de conexiones entre elementos con cables. El esquema de conexionado de cada uno de los elementos se puede encontrar en Figura 21.

Se destacan una serie de puntos en el conexionado de los elementos con el sistema:

- Los pines digitales correspondientes al Step y DIR llevan asociado un diodo LED, el cual aporta información como método de depuración cuando se está trabajando con trenes de pulsos: en función del parpadeo o de la luminosidad del LED se pueden detectar problemas en los trenes de pulsos.
- Se ha incluido un botón de Reset en el escudo, ya que el propio pulsador del Arduino quedará oculto.
- Se ha incluido un pulsador asociado a un pin con interrupciones, el cual es usado para cambiar de trayectoria predefinida.

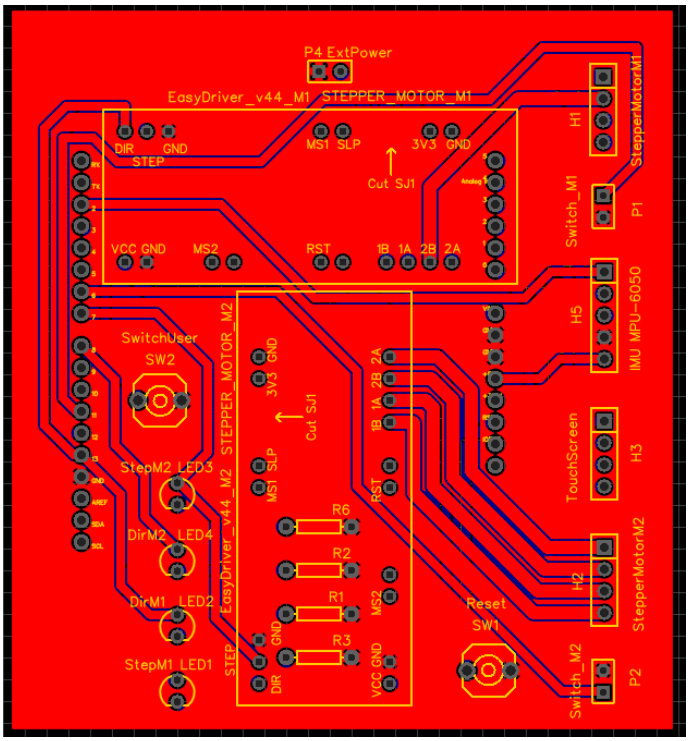


Figura 19 Parte superior del escudo diseñado.

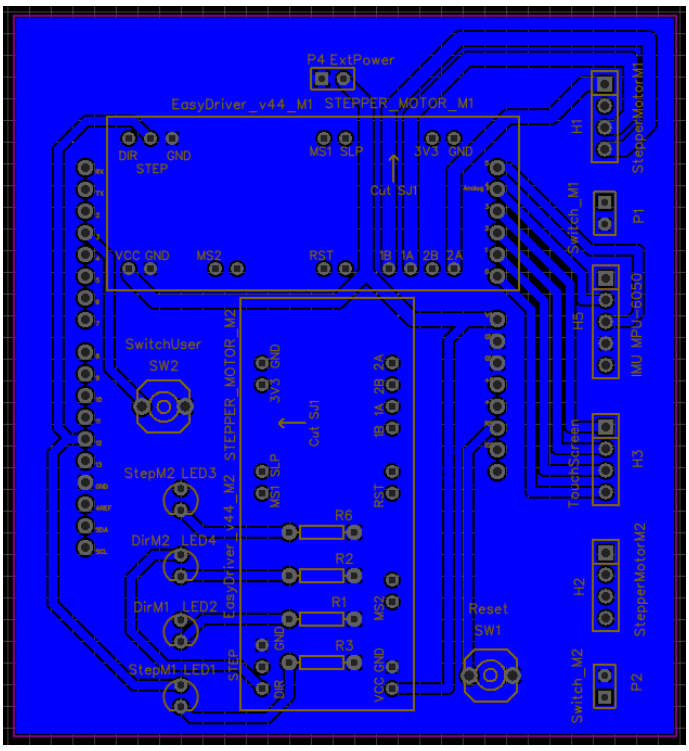


Figura 20 Parte inferior del escudo diseñado.

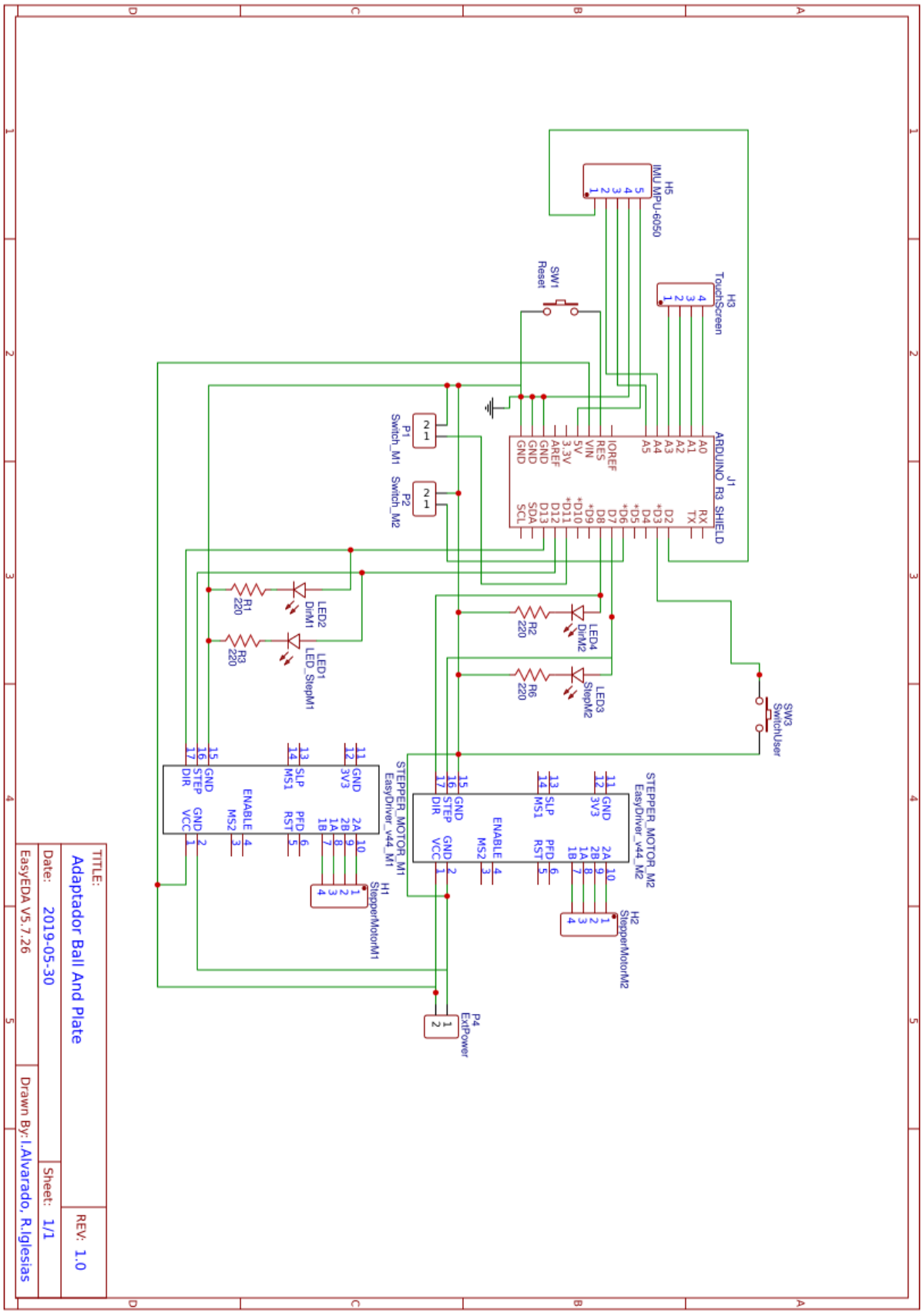


Figura 21 Esquema eléctrico del escudo realizado para el Arduino.

1.3 Descripción matemática del modelo

En esta sección se obtienen las ecuaciones matemáticas que rigen el comportamiento del sistema. El desarrollo completo de este apartado ha sido extraído del modelado matemático realizado en [20].

Con el fin de simplificar la obtención de las ecuaciones se usa la ecuación de Euler-Lagrange, la cual es expresada a continuación:

$$\frac{d}{dt}\frac{\partial T}{\partial \dot{q}_i} - \frac{\partial T}{\partial q_i} + \frac{\partial V}{\partial q_i} = Q_i \tag{1-5}$$

Siendo:

$$T: Energía\ cinética$$

$$V: Energía\ potencial$$

$$q_i\ coordenada\ generalizada\ i$$

$$Q_i\ Fuerza\ generalizada\ i$$

Se realizan una serie de hipótesis en los siguientes cálculos:

- La pelota tiene rodadura sin deslizamiento
- La pelota es completamente simétrica y homogénea
- Se desprecia la fricción en los balances de energía
- La pelota y la superficie se encuentran siempre en contacto

Las coordenadas generalizadas son desplazamiento de la pelota en X (x_b) e Y (y_b), y ángulo de inclinación de la superficie respecto a la horizontal en el eje X (α) e Y (β). Las coordenadas x_b e y_b tienen su origen en el centro de la superficie, como se aprecia en la imagen siguiente.

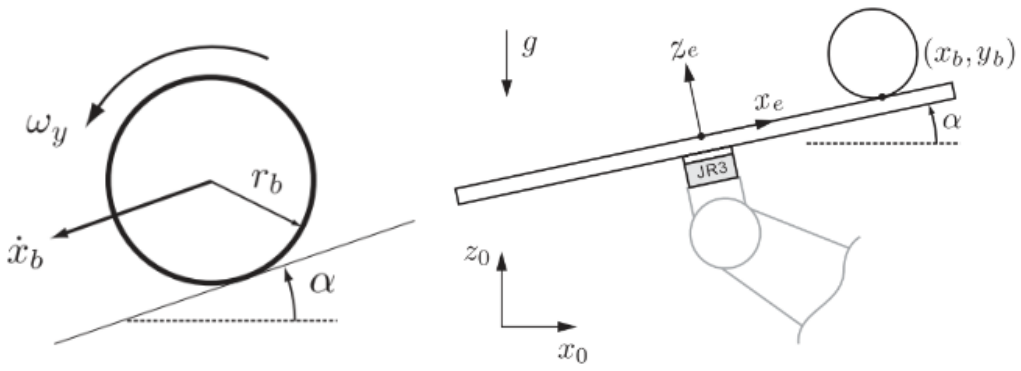


Figura 22 Grados de libertad usados en el desarrollo del modelado.

Se calcula la energía cinética de la pelota, suma de la energía cinética rotacional respecto a su centro de masa y traslacional.

$$T = \frac{1}{2} m_b \left(\dot{x_b}^2 + \dot{y_b}^2 \right) + \frac{1}{2} I_b \left(\omega_x^2 + \omega_y^2 \right) \quad (1\text{-}6)$$

m_b *masa de la pelota*
I_b *momento de inercia*
ω_x *velocidad angular eje X*
ω_y *velocidad angular eje Y*

La velocidad de la pelota, a su vez puede relacionarse con la velocidad angular y el radio:

$$\dot{x_b} = R\, \omega_x \quad \rightarrow \quad \omega_x = \frac{\dot{x_b}}{r_b} \quad (1\text{-}7)$$

$$\dot{y_b} = R\, \omega_y \quad \rightarrow \quad \omega_y = \frac{\dot{y_b}}{r_b} \quad (1\text{-}8)$$

$$T_b = \frac{1}{2} m_b \left(\dot{x_b}^2 + \dot{y_b}^2 \right) + \frac{1}{2} I_b \left(\left(\frac{\dot{x_b}}{r_b} \right)^2 + \left(\frac{\dot{y_b}}{r_b} \right)^2 \right) \quad (1\text{-}9)$$

Operando, la energía cinética de la pelota es

$$T_b = \frac{1}{2} \left(m_b + \frac{I_b}{r_b^2} \right) \left(\dot{x_b}^2 + \dot{y_b}^2 \right) \quad (1\text{-}10)$$

Se calcula ahora la energía potencial, la cual es calculada respecto al origen de coordenadas establecido.

$$V_b = m_b\, g\, (x_b sin\, \alpha + y_b sin\, \beta) \quad (1\text{-}11)$$

Por último, se usa la ecuación de Euler-Lagrange, siendo la lagrangiana:

$$L = T_b - V_b \quad (1\text{-}12)$$

En el caso de las ecuaciones correspondientes a la coordenada generalizada x_b e y_b, se considera que sólo actúa la gravedad.

$$\frac{d}{dt} \frac{\partial T}{\partial \dot{x_b}} - \frac{\partial L}{\partial x_b} = \left(m_b + \frac{I_b}{r_b^2} \right) \ddot{x_b} + m_b g \sin \alpha = 0 \quad (1\text{-}13)$$

$$\frac{d}{dt} \frac{\partial T}{\partial \dot{y_b}} - \frac{\partial L}{\partial y_b} = \left(m_b + \frac{I_b}{r_b^2} \right) \ddot{y_b} + m_b g \sin \beta = 0 \quad (1\text{-}14)$$

Agrupando términos, las ecuaciones diferenciales no lineales del sistema son:

$$\left(m_b + \frac{I_b}{r_b^2} \right) \ddot{x_b} + m_b g \sin \alpha = 0 \quad (1\text{-}15)$$

$$\left(m_b + \frac{I_b}{r_b^2} \right) \ddot{y_b} + m_b g \sin \beta = 0 \quad (1\text{-}16)$$

Las ecuaciones (1-15) y (1-16) relacionan la posición, velocidad y aceleración de la pelota con la inclinación de la superficie.

1.3.1 Linealización del sistema

Con el fin de llevar a cabo la linealización del modelo, se realiza la hipótesis de que los valores de inclinación

de la superficie (β y α) son muy pequeñas, variando como máximo ±5º.

$$\alpha \ll 1 \rightarrow \sin\alpha \simeq \alpha, \quad \beta \ll 1 \rightarrow \sin\beta \simeq \beta$$

Teniendo en cuenta lo anterior, y sabiendo que el momento de inercia de la pelota sólida es $I_b = \frac{2}{5} m_b r_b^2$, se llega a las ecuaciones del modelo linealizado del sistema.

$$\frac{5}{7}\ddot{x_b} + g\alpha = 0 \tag{1-17}$$

$$\frac{5}{7}\ddot{y_b} + g\beta = 0 \tag{1-18}$$

Usando las ecuaciones (1-17) y (1-18), y realizando la transformada de Laplace, se obtienen las funciones de transferencia, pudiendo ser tratado cada eje como un sistema independiente. Para la función de transferencia, se considera que la entrada del sistema es la inclinación de la superficie, y que la salida es la coordenada correspondiente de la pelota sobre la superficie.

$$\frac{X_b(s)}{\alpha(s)} = \frac{g}{\frac{5}{7}s^2} \tag{1-19}$$

$$\frac{Y_b(s)}{\beta(s)} = \frac{g}{\frac{5}{7}s^2} \tag{1-20}$$

Se concluye por ello, que el modelo linealizado depende exclusivamente de la gravedad.

1.3.2 Simulaciones

En este apartado se va a llevar a cabo la simulación del sistema con un controlador LQR (Linear Quadratic Regulator).

En primer lugar, se representa el modelo linealizado del sistema expresado en espacio de estados para tiempo discreto

$$\ddot{x}_{k+1} = A\, x_k + B\, u_k \tag{1-21}$$

siendo

x_k: Vector de estados del instante actual, formado por variables que describen la evolución del sistema.

u_k: Vector de entradas del sistema, es decir, las variables de actuación del sistema (ángulos α y β).

A: Matriz de estados.

B: Matriz de entrada.

Para el sistema modelado, queda:

$$\begin{bmatrix}\ddot{x}_1\\ \ddot{x}_2\\ \ddot{x}_3\\ \ddot{x}_4\end{bmatrix} = \begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\\ x_4\end{bmatrix} + \begin{bmatrix}-\frac{7}{5}\,g & 0\\ 0 & -\frac{7}{5}\,g\\ 0 & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}u_1\\ u_2\end{bmatrix} \tag{1-22}$$

Siendo:

$$x_1 = \dot{x}$$

$$x_2 = \dot{y}$$

$$x_3 = x$$

$$x_4 = y$$

Un controlador LQR definido para un sistema expresado en espacio de estados es aquel que con una acción de control de tipo $u_k = -K\,x_k$ minimiza la función de coste cuadrático

$$J(u) = \sum_{k=1}^{\infty} [(x_k)^T Q\, x_k + (u_k)^T R\, u_k] \qquad (1\text{-}23)$$

Para la obtención de la ganancia K para realizar las simulaciones del control del sistema en tiempo discreto se puede usar la función *dlqr* incluida en el paquete Control System Toolbox de Matlab.

Esta función tiene como entradas:

- Matrices A y B, que definen la evolución del estado del sistema
- Matrices Q y R, que ponderan el error del estado, y el esfuerzo de control respectivamente.

Se realizan simulaciones para el seguimiento de una trayectoria circular de radio 170 mm para un tiempo de muestreo de 0.064 ms con los valores de Q y R especificados a continuación, dando como resultante la matriz K usada en la acción de control del controlador.

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 80 \end{bmatrix} \quad R = \begin{bmatrix} 30 & 0 \\ 0 & 30 \end{bmatrix} \quad \rightarrow \quad K = \begin{bmatrix} -0.0027 & 0 & -0.0155 & 0 \\ 0 & -0.0027 & 0 & -0.0155 \end{bmatrix}$$

La evolución del espacio de estados que se obtiene con esta función de costo es el siguiente:

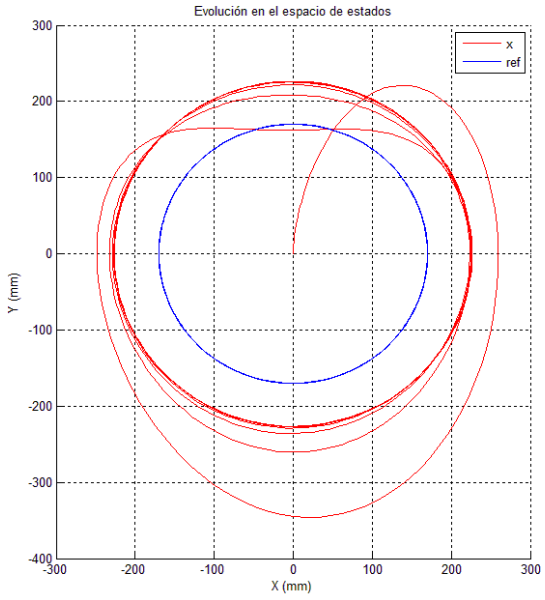


Figura 23 Evolución del espacio de estados con controlador LQR.

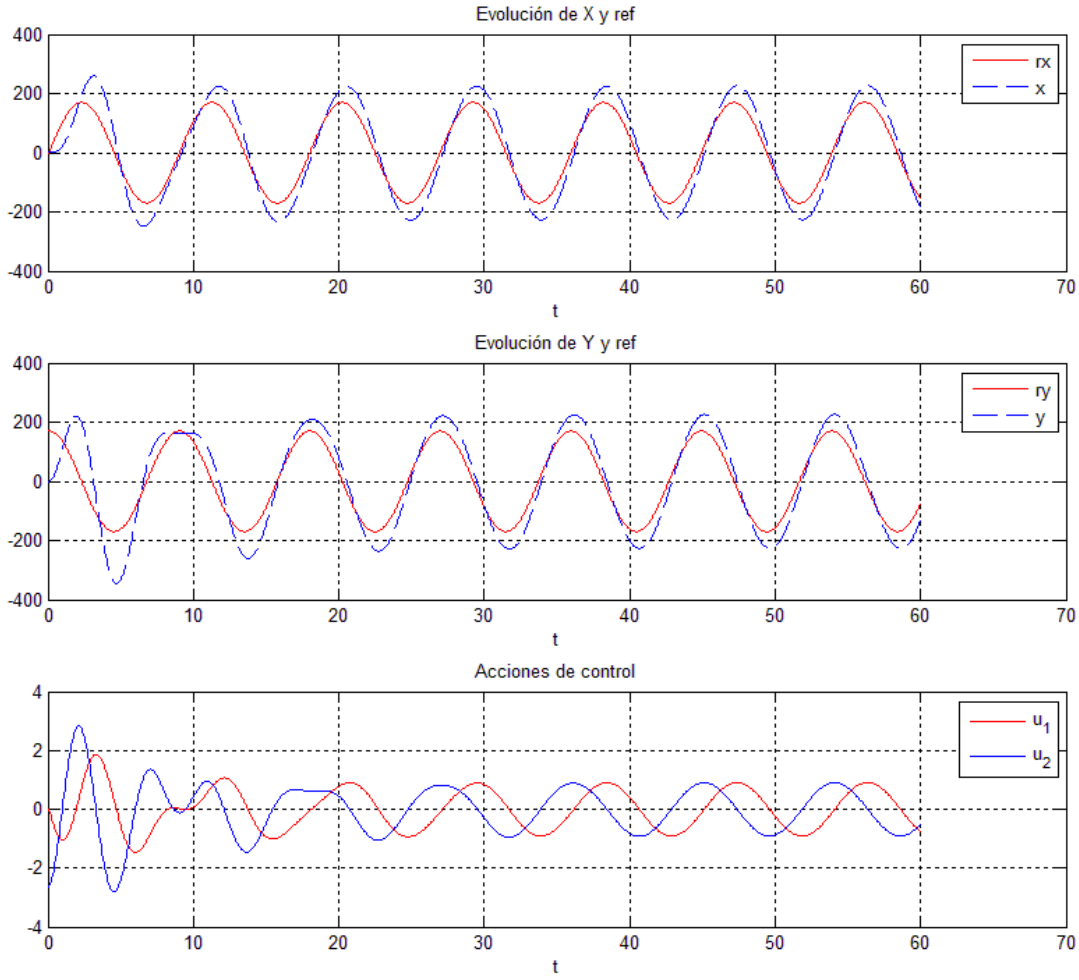


Figura 24 Evolución del eje X e Y, y de las acciones de control en el tiempo.

También es posible simular un offset de 0.5 grados en las medidas del ángulo de la superficie donde desliza la pelota, pudiendo obtener el comportamiento del sistema con ese supuesto.

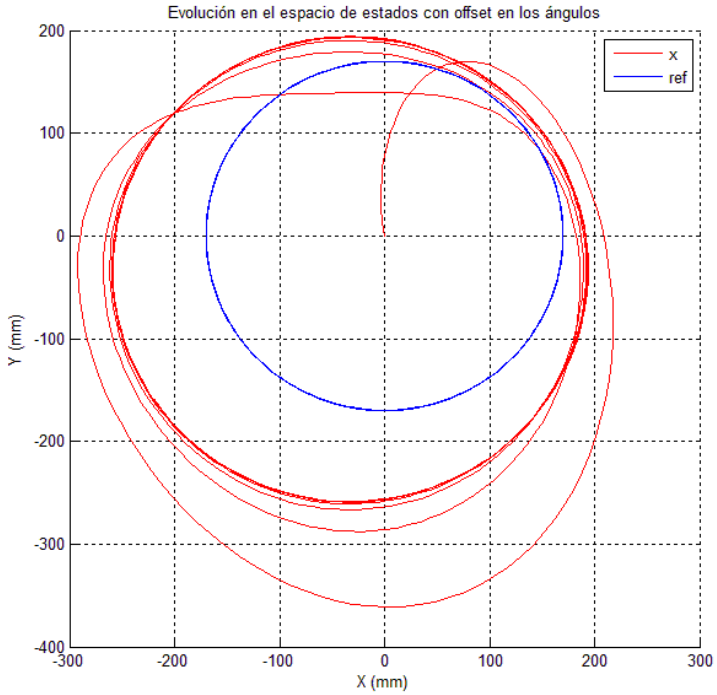


Figura 25 Evolución del espacio de estados con offset de 0.5 grados en ángulos.

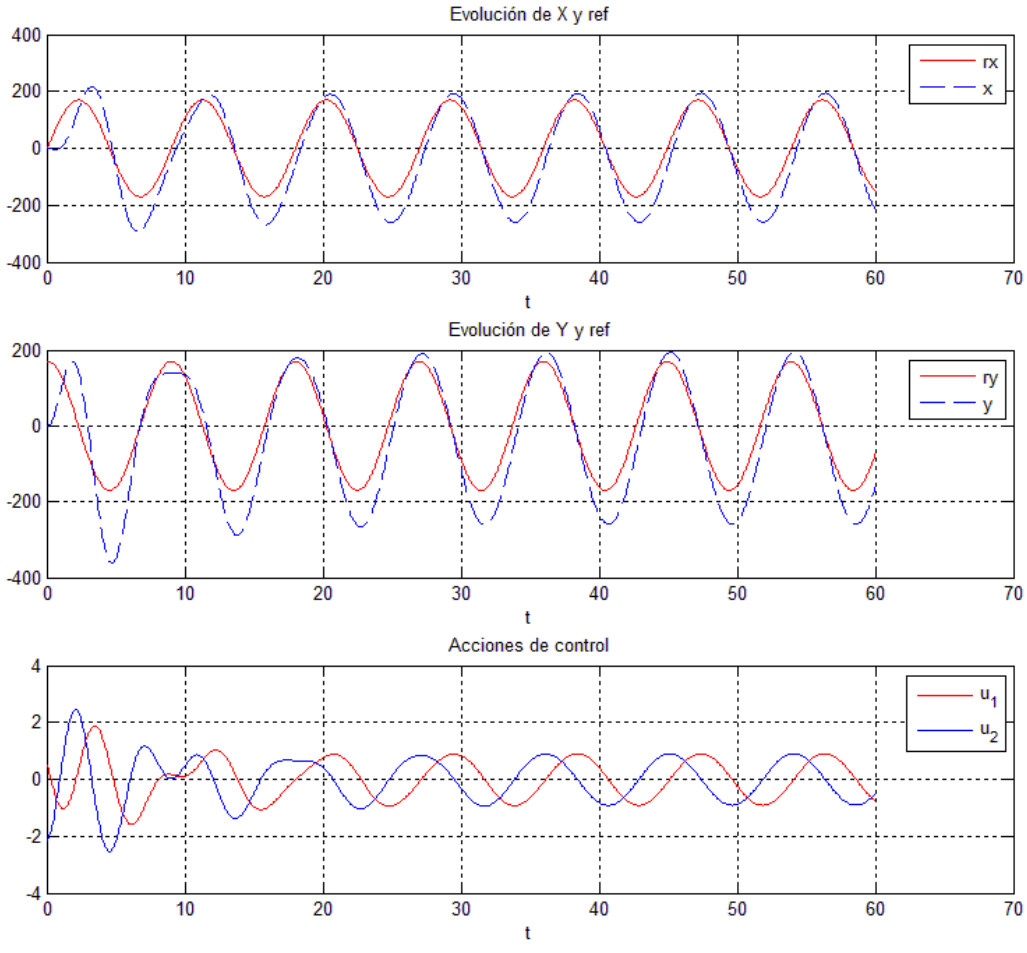


Figura 26 Evolución de X e Y, así como de acciones de control con offset de 0.5 grados en ángulos.

2 ESTRATEGIA DE CONTROL

En esta sección, se lleva a cabo una descripción de la estrategia de control realizada para el control de la trayectoria de la pelota en el sistema planteado.

2.1 Esquema de control del sistema

En un primer lugar, se llevó a cabo una estrategia de control básico, en el que la pantalla táctil era la única entrada de información real del sistema para conocer la posición actual de la pelota. El ángulo de la superficie sobre la que se desplaza la pelota con respecto a la base del sistema era estimado mediante los motores paso a paso, suponiendo que no hay pérdidas de pasos.

2.1.1 Primera implementación de control

En esta estrategia de control se usa la referencia y la posición actual de la pelota (x_b, y_b) para calcular el error en posición, siendo este error la entrada al controlador PID. Este controlador tiene como salida la variación de ángulo necesaria a aplicar al motor paso a paso correspondiente, el cual varia la inclinación de la pantalla.

A continuación, se presenta un esquema de la primera estrategia descrita, presentando a efectos de claridad el control de la pelota en el eje X, siendo el control de la posición en el eje Y equivalente.

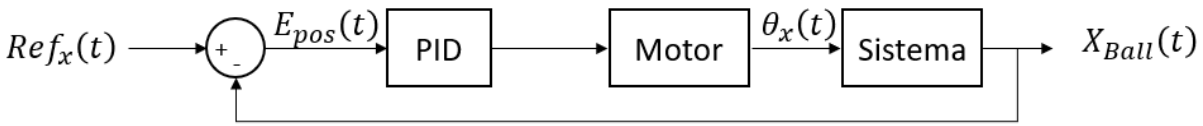


Figura 27 Estrategia de control inicial.

No obstante, esta estrategia de control supone que el ángulo de la superficie por la que se desplaza la pelota con respecto a la gravedad es el determinado por los motores paso a paso, lo cual no cierto. En la realidad, este ángulo es determinado con la posición en ese instante de los motores paso a paso más el ángulo que tenga la superficie en la que se encuentra apoyado el sistema completo. Es decir, si el sistema completo es apoyado en una mesa cuya superficie presenta una inclinación determinada con respecto al suelo, el error en posición de la pelota en régimen permanente aumentará.

Con el fin de corregir este offset constante en el ángulo de la pantalla, se podría incluir una calibración del sistema antes de su funcionamiento. Sin embargo, el sistema se seguiría viendo afectado si este offset siguiera variando con el tiempo. Un ejemplo de offset variante con el tiempo podría ser sostener el sistema en brazos durante su funcionamiento, produciendo variaciones de inclinación de su base.

2.1.2 Segunda implementación de control

Con el fin de conseguir un buen rechazo a perturbaciones de este tipo se modifica la estrategia de control incluyendo una realimentación de la inclinación real de la superficie, de forma que los resultados de seguimiento de trayectoria fueran en cierta medida independientes a agentes externos como la inclinación de la superficie

sobre la que se apoya el sistema.

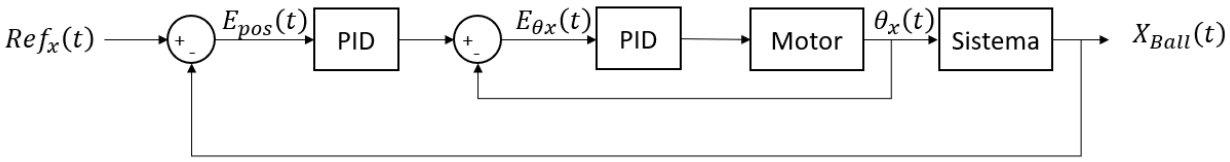


Figura 28 Estrategia de control corregida.

Si se observa el anterior esquema, se ve cómo existen dos controladores acoplados entre ellos, encontrándose en el nivel más bajo el control de la inclinación de la superficie, y estando en el nivel superior el control de la posición. Con el fin de que esta estrategia funcione de forma esperada, el controlador inferior debe ser ejecutado al menos 10 veces más rápido que el superior (el controlador inferior debe tener el tiempo necesario para alcanzar la referencia establecida).

No obstante, debido a las limitaciones temporales por la propia implementación en Arduino, finalmente los mejores tiempos de ejecución que se han conseguido son la ejecución del control de alto nivel cada 64 ms y la ejecución del controlador de bajo nivel cada 8ms. Esto da lugar a que el control de bajo nivel sea sólo 8 veces más rápido que el de alto nivel. A pesar de que no se llegue a una ejecución de 10 veces más rápido, se ha obtenido un resultado bastante satisfactorio en el control.

A continuación, se describe de forma esquemática el control, con el fin de facilitar la comprensión de la estructura al lector:

Controlador alto nivel: Control de posición de la pelota

- *Entrada*: Error de posición de la pelota, siendo la diferencia entre la posición leída por la pantalla táctil y la referencia en el eje correspondiente dependiente de la trayectoria seguida.
- *Salida*: Referencias de inclinación de la pantalla que tiene que ser alcanzado por el controlador de bajo nivel.

Controlador bajo nivel: Control de ángulo de la superficie

- *Entrada*: Error de inclinación de la superficie, siendo la diferencia entre la referencia dada por el controlador de alto nivel y los ángulos de inclinación de la superficie calculados en función del IMU.
- *Salida*: Número de pulsos a generar para mover los motores paso a paso que modifican el ángulo de la pantalla.

3 IMPLEMENTACIÓN DEL CONTROLADOR

Como se describió en la sección 1, el sistema es controlado haciendo uso de un microcontrolador Arduino Uno. Éste actúa sobre el sistema mediante el movimiento de dos motores paso a paso que varían el ángulo de la superficie respecto a la horizontal, controlados a su vez por dos drivers. Como datos de entrada al controlador se encuentran la posición de la pelota en la superficie obtenida por la pantalla táctil y una unidad de medición inercial (IMU).

Una vez se ha descrito el sistema a controlar y la estrategia de control planteada, en esta sección se explica en primer lugar la coordinación temporal entre cada uno de los elementos del sistema dentro de la programación, y luego el control de cada uno de ellos de forma aislada. Este apartado es de vital importancia, ya que es el aspecto que más tiempo ha consumido durante la ejecución del proyecto.

Usando sólo un Arduino Uno, hay diversos requisitos a cumplir:

- Lectura de la pantalla táctil, así como la gestión del filtrado de datos.
- Lectura de la inclinación actual de la superficie a partir de la comunicación con la unidad de medidas inerciales (IMU).
- Gestión de los trenes de pulsos de cada motor de forma independiente, con el fin de conseguir un movimiento adecuado de los motores.
- Ejecución del control de bajo nivel (control del ángulo de la superficie).
- Ejecución del control de alto nivel (control de la posición de la pelota).
- Envío de forma periódica de datos de supervisión a través del puerto serie a la aplicación en Python creada para ello.
- Todos los anteriores puntos además están sujetos a una ejecución periódica distinta en cada caso y a dependencias de datos entre ellas, lo cual dificulta aún más la implementación conjunta.

Con el fin del cumplimiento de todas ellas, la única solución viable es una implementación en Arduino basada en interrupciones. Por ello, se presenta en primer lugar una introducción a las interrupciones en Arduino.

3.1 Introducción a las interrupciones en Arduino

José Antonio Borja Conde en [21] realiza una introducción a las interrupciones, en las que se encuentra una síntesis muy completa sobre rutinas de interrupciones, temporizadores y usos de registros asociados a cada uno. Es por ello que se ha decidido realizar una cita de forma textual a estos apartados, los cuales fueron usados durante la integración del controlador en Arduino para una correcta comprensión de las interrupciones.

> Cuando se está ejecutando un programa, suelen producirse ciertas condiciones o acontecimientos por las que hay que realizar una operación concreta. A estas circunstancias se les denomina evento.

La detección del evento es un problema común, que se resuelve normalmente de dos formas posibles.

La primera forma es la denominada “poll”, que consiste en estar continuamente cada un tiempo fijo comprobando si se ha producido o no. Esta forma es más sencilla, pero tiene muchas desventajas, las cuales hacen que en numerosas ocasiones sea imposible de implementar:

1. Supone un alto coste de procesamiento. Ya que se está continuamente comprobando el estado, y en la mayoría de las ocasiones la respuesta no será útil (no se cumple el evento).

2. El hecho de tener que estar continuamente realizando la búsqueda, hace que otros procesos no puedan ser ejecutados de forma paralela. Al menos si el problema no es muy simple.

3. Si la respuesta al evento debe ser inmediata, este sistema no puede asegurarlo. De hecho, la probabilidad de que se responda al instante es prácticamente nula.

4. Si la duración del evento es inferior al tiempo de espera, puede que se produzca y desaparezca antes de ser detectado.

Por tanto, es evidente que este método solo se puede utilizar en tareas muy sencillas y con poca importancia. Por ello, los microprocesadores incorporan las “interrupciones”.

Una interrupción es una parada temporal de la ejecución de un evento para ejecutar una subrutina que corresponde al evento que ha generado la interrupción. Cuando se ha ejecutado la subrutina, se continúa ejecutando el proceso interrumpido por el mismo punto en que lo dejó.

Por ejemplo, la activación de un sensor o un pulsador, que se produzca un error, la activación de un temporizador, etc., podrían ser considerados eventos para generar una interrupción. La función correspondiente a una interrupción tiene la siguiente nomenclatura en Arduino:

```
ISR(“Nombre del registro correspondiente”)
{
// Proceso
}
```

En este proyecto, el tipo de interrupciones que se usarán fundamentalmente serán las interrupciones temporales, las cuales van asociadas al uso de temporizadores. Estos permiten producir eventos en instantes determinados de tiempo. Los hay de dos tipos:

- Temporizadores de un solo disparo: Avisan únicamente una vez desde que se activan hasta que se cumple el tiempo especificado. No tienen mucho interés para la programación en Arduino.
- Temporizadores periódicos: Serán muy útiles en este proyecto. Se caracterizan por permitir generar eventos con un periodo especificado.

A los temporizadores periódicos normalmente se les llama (y será el término utilizado en este proyecto) “timers”.

El funcionamiento de un timer, muy simplificadamente, consiste en un contador (un registro de un número determinado de bits), que va incrementándose una unidad cada vez que se produce un flanco en la señal de reloj del microprocesador. La señal de reloj es siempre fija, y se debe conocer su frecuencia, lo que permite tener noción del tiempo. Cuando el contador alcance un valor determinado, se producirá el evento.

Timers en Arduino UNO

Introducción a los registros

Para poder explicar el funcionamiento de los timers, se hará continuamente referencia a sus registros, por tanto, aquí se hace una introducción a estos. No se entra muy detalladamente en para qué sirve cada uno. Se recomienda tener este apartado accesible en la lectura de las siguientes secciones, ya que clarifica el entendimiento.
Para empezar, un registro es un conjunto de datos estructurados. Cada dato tiene su propia dirección, a través de la cual se puede consultar o modificar su valor. Para modificar su valor, en este proyecto, se usarán los llamados registros de desplazamiento. Esto se explicará a partir del siguiente ejemplo.

Ejemplo 3–1. Supóngase que se tiene un registro llamado REGEJEM, que consta de 4 bits. Normalmente, a cada posición se le asigna un identificador. Suponiendo que estas serían: POS0, POS1, POS2 y POS 3, y que sus valores iniciales son 0, 1, 1 y 0, queda el siguiente esquema final:

REGEJEM	POS3	POS2	POS1	POS0
Valor	0	1	1	0

Tabla 6 Prescaladores Timer 0 y 1

Para crear un registro de desplazamiento habría que escribir, por ejemplo, "1<<POS3", que es igual que escribir "1000". Haciendo uso de esto, se pueden modificar los valores de la siguiente forma:

- Poner POS3 = 1:
  REGEJEMP |= 1<<POS3.
  Si nos fijamos es igual que hacer REGEJEM = 0110 OR 1000 = 1110.
- Poner POS2 = 0:
  REGEJEM &= ~(1<<POS2).
  Si nos fijamos es igual que hacer: REGEJEM = 0110 AND NOT(0100) = 0110 AND 1011= 0010.

Una vez se tiene claro qué es y cómo modificar un registro de desplazamiento, se presentan los registros de los timers que se usarán en este proyecto, con una breve explicación que se puede complementar con los apartados posteriores, los cuales se indican. El carácter "x" se refiere al timer, que puede ser 0, 1 o 2:

- TCNTx: Valor del contador asociado al timer. Es un entero comprendido entre 0 y 255 para el timer 0 y 2, y entre 0 y 65535 para el timer 1. Cada flanco de reloj lo incrementa una unidad, de forma que este valor sirve de referencia para referencias temporales. Su valor vuelve a ser 0 cuando se alcanza el máximo o cuando se resetea. Más información en el apartado "Características principales"
- OCRxA y OCRxB: Valores para que salten las interrupciones A o B, respectivamente. Cuando TCNTx se iguale a alguna de estas variables, se activará la interrupción correspondiente. En modo normal, si están activadas, se ejecutará la función de interrupción. En modo CTC sirven para resetear el valor del contador cuando se iguala a cualquiera de las dos, en el PWM para activar y desactivar la señal de salida, etc. Más información en los apartados "Modos de temporización" y "Interrupciones Timers".
- TIMSKx: Se denomina máscara del timer. Sirve para activar o desactivar las interrupciones asociadas. Sus posiciones OCIExA y OCIExB activan y desactivan las interrupciones A y B del timer, las cuales se producen cuando TCNTx = OCIExA/B. La posición TOIEx activa y desactiva la interrupción por OverFlow del timer. Más información en el apartado "Interrupciones Timers".
- TCCRxA y TCCRxB: Permiten variar las características de los timers. Según las posiciones a las que se le varía el valor se modificará una característica u otra:

  - CSx0, CSx1 y CSx2: Posiciones de los registros TCCRxA/B para el prescalado del timer, de forma que TCNTx se incrementa cada tantos flancos de reloj como prescalado haya especificado. Más información en el apartado "Prescalado Timers".

  - WGMx3, WGMx2, WGMx1 y WGMx0: Posiciones de los registros TCCRxA/B para cambiar su modo de funcionamiento. Más información en el apartado "Modos de temporización".

Características principales

Para explicar los timer en Arduino, es necesario primero conocer algunas características básicas del Arduino.

El Arduino UNO tiene una frecuencia de reloj de 16Mhz. Esto significa que cada 62.5 ns se produce un flanco en la señal de reloj.

El Arduino UNO, consta de 3 timers programables:

- Timer 0: Tiene 8 bits de resolución, lo que quiere decir que el contador, TCNT0, puede alcanzar un valor máximo de $(2^8-1) = 255$ (de 0 a 255). Este temporizador es usado por funciones de tiempo muy comunes como millis(), micros(), delay()...Por lo que si se varía su modo de funcionamiento quedarán

inhabilitadas. No obstante, es posible hacer uso de sus interrupciones, aunque el periodo por defecto asociado no pueda ser modificado. Tiene asociado los pines 5 y 6 para generar señales PWM.

- Timer 1: Tiene 16 bits de resolución, lo que quiere decir que el contador, TCNT1, puede alcanzar un valor máximo de (2_{16}-1) = 65335 (de 0 a 65335). Tiene los pines 9 y 10 asociados a la señal PWM.
- Timer 2: Es equivalente al Timer0, con la única diferencia de que no está asociado a las funciones de tiempo típicas y los pines asociados a la señal PWM son 3 y 11.

Prescalado Timers

Como se ha podido ver en apartados anteriores, el fin de los timers es generar señales con un periodo determinado.

Con la frecuencia de reloj y la resolución de los timers se pueden conocer los periodos:

- El reloj genera un pulso cada (1/16 MHz) = 62.5 nseg.
- El timer 1 permite un máximo de $(2^{16} - 1)$ = 65535 incrementos. El timer 0 y 2 permite 255.
- Periodo máximo con timer 1: 62.5nseg*65535 = 4.096 ms.
- Periodo máximo con timer 0 y 2: 62.5nseg*255 = 15.94 µs.

Los preescaladores permiten aumentar estos periodos de forma que en vez de incrementarse TCNTx cada pulso de reloj, lo haga cada un número de veces definido por el usuario.

Para ello habría que modificar los registros de la siguiente forma:

CSx2	CSx1	CSx0	Descripción
0	0	1	Prescala = 1
0	1	1	Prescala = 64
0	1	0	Prescala = 256
1	0	1	Prescala = 1024

Tabla 7 Prescaladores Timers 0 y 1

CSx2	CSx1	CSx0	Descripción
0	0	1	Prescala = 1
0	1	0	Prescala = 8
0	1	1	Prescala = 32
1	0	0	Prescala = 64
1	0	1	Prescala = 128
1	1	0	Prescala = 256
1	1	1	Prescala =1024

Tabla 8 Prescaladores Timer 2

Por tanto, se pueden conseguir los siguientes periodos:

Prescala / Timer	1	8	32	64	128	256	1024
0	15.94 µs	127.52 µs	510.08 µs	1021 ms	15.94 ms	15.94 ms	15.94 ms

1	4.096 ms	---	---	262.14 ms	---	15.94 s	15.94 s
2	15.94 µs	---	---	1021 ms	---	15.94 ms	15.94 ms

Tabla 9 Periodos máximos con timers prescalados

Modos de temporización

Los temporizadores se pueden configurar en hasta dieciséis modos distintos. Como vemos en las siguientes tablas:

Modo	WGM13	WGM12	WGM11	WGM10	Modo de operación
0	0	0	0	0	Normal
1	0	0	0	1	PWM, Phase Correct, 8-bit
2	0	0	1	0	PWM, Phase Correct, 9-bit
3	0	0	1	1	PWM, Phase Correct, 10-bit
4	0	1	0	0	CTC
5	0	1	0	1	Fast PWM, 8-bit
6	0	1	1	0	Fast PWM, 9-bit
7	0	1	1	1	Fast PWM, 10-bit
8...15	...	...	...	...	...

Tabla 10 Registros para modos de operación Timer 1

Modo	WGM13	WGM12	WGM11	WGM10	Modo de operación
0	0	0	0	0	Normal
1	0	0	0	1	PWM, Phase Correct
2	0	0	1	0	CTC
3	0	0	1	1	Fast PWM
4	0	1	0	0	Reservado
5	0	1	0	1	PWM, Phase Correct
6	0	1	1	0	Reservado
7	0	1	1	1	Fast PWM
8...15	...	...	...	...	...

Tabla 11 Registros para modos de operación Timer 0 y 2

Básicamente destacan tres:

- Modo output compare (CTC): Este modo de operación consiste en fijar el valor máximo que debe alcanzar el contador del timer para resetarlo. Como se ha dicho anteriormente, se puede acceder al valor del contador a través del registro TCNTx. Para fijar la cota superior, se indica

en los registros OCRxA y OCRxB. Cuando TCNTx sea igual a cualquiera de estos dos valores saltará la interrupción correspondiente al valor que se haya alcanzado, se reseteará el contador y se repetirá el proceso. Es decir, si OCRxA es menor que OCRxB, TCNTx se reseteará cuando se iguale al valor OCRxA. Si de lo contrario OCRxB es menor que OCRxA ¸se reseteará cuando se iguale al valor OCRxB.
Para el timer 1 habría que poner: WGM1[3:0] = 0100, y para el timer 0 o 2: WGMx[3:0] = 0010.
Es importante tener en cuenta que el valor de OCRxA/B no puede sobrepasar el valor de resolución del timer (255 para el 0 y 2 y 65535 para el 1). Además, si su valor es nulo, no habrá interrupción.

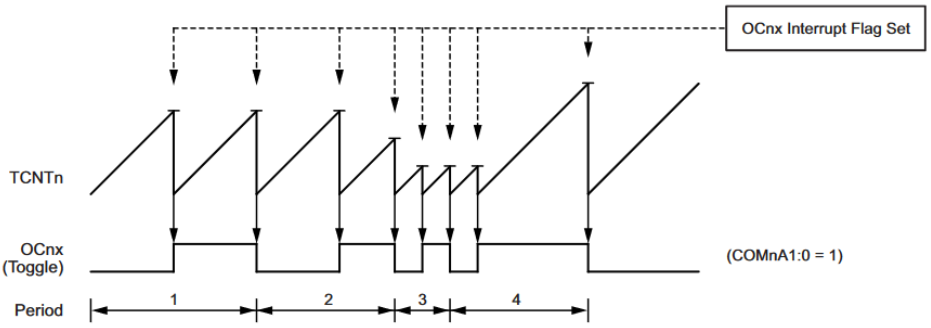


Figura 29 Funcionamiento del modo CTC

- Modo Fast PWM: Este modo de operación permite generar una señal por ancho de pulso. El valor TCNTx siempre llega al máximo. Hay que asignar a OCRxA o OCRxB el valor correspondiente para producir el ancho del pulso deseado según la siguiente fórmula:

$$OCRxy = Porcentaje\ PWM * Valor\frac{máximoTCNTx}{100}$$

De esta forma, mientras TCNTx sea inferior a OCRxy, la salida será positiva y en caso contrario nula.
No obstante, esto es válido cuando se tiene que la señal de salida no esté negada. En caso contrario, mientras TCNTx sea inferior a OCRxy, la salida será nula y en caso contrario positiva.
Para que la salida no esté negada hay que poner las posiciones del registro COMx1 = 1 y COMx2 = 0, y para negarla COMx1 = 1 y COMx2 = 1.
Según las Tabla 8 y Tabla 9, para conseguir este modo en el timer 1 habría que poner: WGM1[3:0] = 0101, y para el timer 0 o 2: WGMx[3:0] = 0011.
Es importante tener en cuenta que el valor de OCRxA/B no pude sobrepasar el valor de resolución del timer (255 para el 0 y 2 y 65535 para el 1).

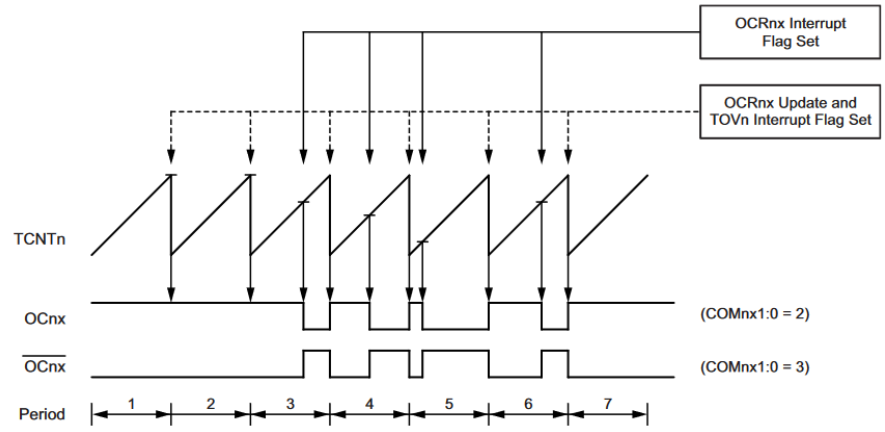


Figura 30 Funcionamiento del modo Fast PWM

- Modo normal: Es el modo más importante en este proyecto, ya que será el único que se use. Esto se debe a que, aunque es el más básico, es el que más libertad de programación permite, y con esta se pueden conseguir los demás modos de operación.

La forma de utilizarlo es modificar el valor OCRxA/B. TCNTx irá incrementándose continuamente del valor mínimo al máximo. Cuando su valor se iguale a alguno de los registros previamente indicados, saltará la interrupción correspondiente, y TCNTx seguirá aumentando.
Habría que activar los avisos de estas interrupciones (modificando los registro de TIMSKx) y crear sus procesos correspondientes según la siguiente tabla:

Registro	Función asociada	Interrupción
OCIExA	ISR(TIMERx_COMPA_vect)	TCNTx = OCRxA
OCIExB	ISR(TIMERx_COMPB_vect)	TCNTx = OCRxB
TOIEx	ISR(TIMERx_ OVF_vect)	TCNTx OverFlow

Tabla 12 Interrupciones Timers

Interrupciones Timers

Como se ha dicho anteriormente, cada timer consta de tres interrupciones. Las interrupciones A y B se producen cuando el valor OCRxA/B es igual a TCNTx, y la interrupción por OverFlow se produce cuando TCNTx alcanza su máximo valor posible (255 o 65535, según el timer).

De esta forma, si está programado el modo normal y teniendo en cuenta la prescala y la resolución del timer, se pueden generar interrupciones separadas un periodo deseado, con las siguientes ecuaciones:

$$T_A = OCRxA * \frac{Prescala}{16000000}$$

$$T_B = OCRxB * \frac{Prescala}{16000000}$$

$$T_{OVF} = Máx(TCNT_x) * \frac{Prescala}{16000000}$$

Para activarlas hay que modificar la máscara de los timers de la siguiente forma (se hace para el caso del timer 1, pero es exactamente igual para los demás):

```
TIMSK1 = 0; //Limpiar la máscara del timer
TIMSK1 |= (1 << OCIE1A); //Activar interrupción A
TIMSK1 |= (1 << OCIE1B); //Activar interrupción B
TIMSK1 |= (1 << TOIE1); //Activar interrupción OverFlow
```

Por último, habría que crear la función que se ejecutará cuando se produzca el evento:

```
ISR(TIMER1_OVF_vect)
{
// Proceso cuando se genere OverFlow en el timer (TCNT1 = 65535)
}
ISR(TIMER1_COMPA_vect)
{
//Proceso cuando se alcance el valor de A (TCNT1 = OCR1A)
}
ISR(TIMER1_COMPB_vect)
{
//Proceso cuando se alcance el valor de B (TCNT1 = OCR1B)
}
```

Los procesos citados deben ser lo más breve posible, ya que si son densos habrá problemas con la ejecución de las distintas interrupciones que se produzcan y el programa dejará de funcionar.

En el caso de que sea muy necesario realizar un proceso largo dentro de una interrupción, puede usarse la orden interrupts(), que permite que las demás interrupciones se ejecuten sin haber acabado la que llamó a la función, y el programa seguirá funcionando.

3.2 Integración temporal de los elementos en la implementación de Arduino

Una vez presentadas las interrupciones en Arduino, se procede a resumir a continuación el uso de todas las interrupciones temporales en este proyecto a través de la Figura 31.

A efectos de clarificar el esquema, no se ha incluido la gestión de las comunicaciones con la unidad de medidas inerciales (MPU-6050), pues se gestiona a través de interrupciones asociadas a pines digitales. También se ha incluido en el timer 2 las interrupciones de un sólo registro de comparación (para la generación de un tren de pulsos), siendo la gestión de interrupciones del otro registro de comparación de forma análoga.

La configuración de todos los timers usados es la siguiente:

- Timer 1 (16 bits): Configurado en modo normal. Prescalador 8, generando overflows cada 32 ms.
    - Habilitada 1 interrupción por registro de comparación (A).
    - Habilitada interrupción por overflow.
- Timer 2 (8 bits): Configurado en modo normal. Prescalador 256, generando overflows cada 4 ms.
    - Habilitadas 2 interrupciones por registro de comparación (A y B), siendo una para cada tren de pulsos de los motores.
    - Habilitada interrupción por overflow.

Los tiempos usados son los siguientes:

- Actualización de valores para la trayectoria: Cada 64 ms (cada dos overflows del Timer 1).
- Ejecución controlador PID del nivel superior correspondiente a cada eje: Cada 64 ms, desfasados entre ellos 32 ms.
- Ejecución controlador PID del nivel inferior correspondiente a cada eje: Cada 8ms (cada dos overflows del Timer 2).

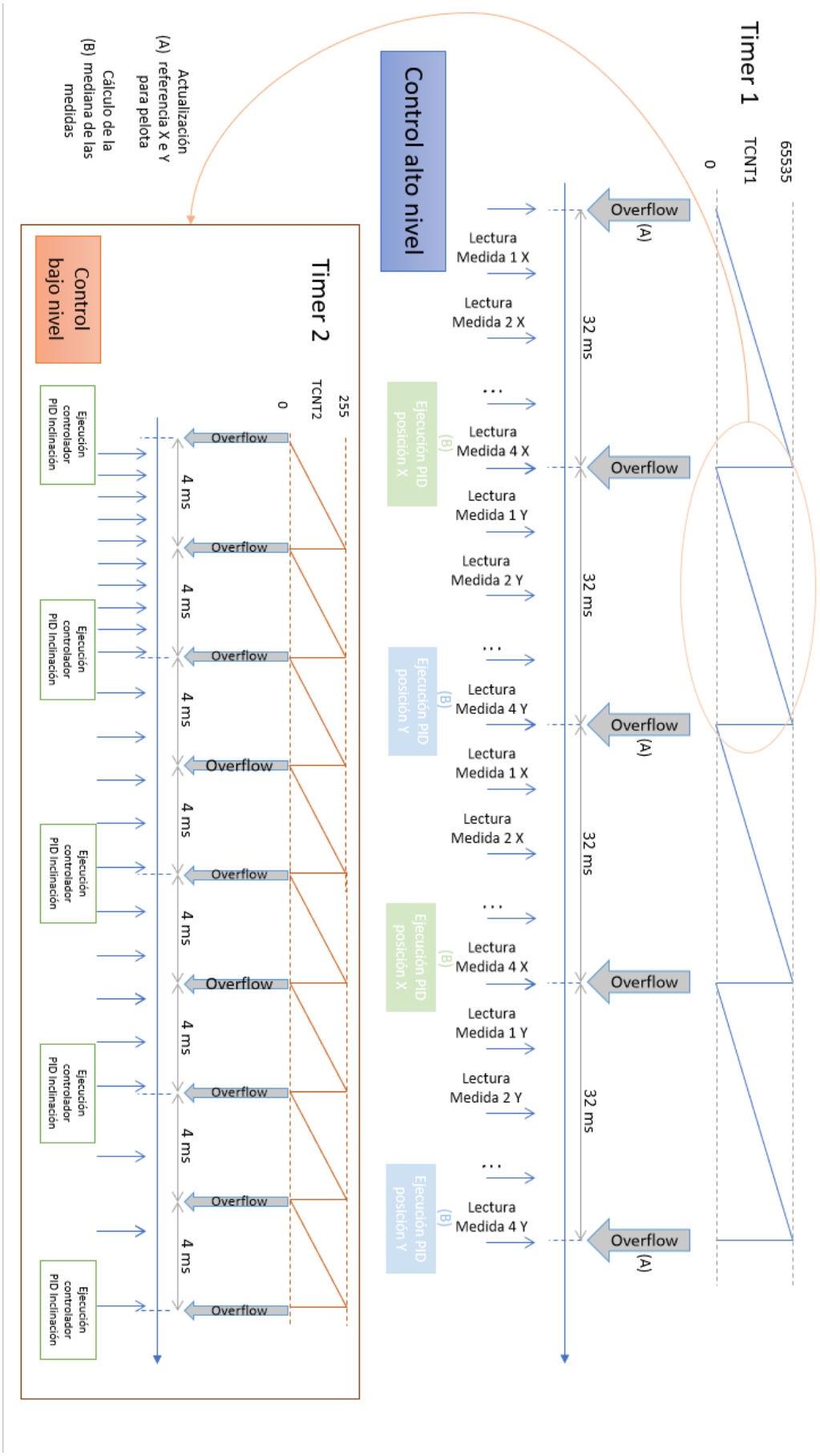


Figura 31 Coordinación temporal de la implementación en Arduino.

3.3 Movimiento de motores paso a paso

Como se vio anteriormente en las secciones 1.1.2 Motores paso a paso y 1.1.3 EasyDriver, para el control de los motores se usa un driver que hace de interfaz entre el Arduino y el motor paso a paso, de forma que con sólo dos pines (STEP y DIR) es posible controlar el avance de pasos y el sentido del giro del motor.

Con el fin de conseguir la mayor resolución posible, el driver se configura a 1/8 de paso por cada flanco de subida en el pin STEP (estado por defecto cuando no se encuentran conectados los pines MS1 y MS2 del EasyDriver). Para realizar el control del movimiento, sólo es necesario generar un tren de pulsos para cada motor, siendo el número de pulsos dados igual al número de fracciones de 1/8 de pasos que el motor avanzará.

Para esta generación de pulsos existen librerías ya incluidas en Arduino que permiten su gestión con sólo incluir el número de pasos por vuelta y el número de grados del que se desea el movimiento. Un ejemplo de ello es la librería Stepper, incluida en las librerías por defecto de Arduino. No obstante, las funciones de esta librería son bastante básicas y su ejecución no está basada en interrupciones; es por ello, que no permite, por ejemplo, el movimiento simultáneo de dos motores paso a paso. También hay otras como AccelStepper, que sí que permiten su uso sin que bloqueen la ejecución del resto del código.

En este proyecto se ha decidido realizar una implementación propia de la generación de trenes de pulsos, de forma que se pueda llevar un mejor control de todos los timers e interrupciones usados en la programación de forma global, sin estar limitado por código externo. Para ello, se ha creado una clase *Motor*, la cual al instanciarla se definen parámetros como los pines usados, o la configuración del motor. Esta clase desarrollada se explica más adelante en el apartado 3.3.2 y puede ser encontrada en Anexo A: Programación en Arduino.

3.3.1 Generación de trenes de pulsos

El objetivo de esta implantación es conseguir un movimiento rápido, pero a la vez suave. Por ello, en la implementación se busca que los pulsos se encuentren siempre equidistantes en el mismo periodo de control de los motores.

Para la generación de cada tren de pulsos del motor sería lógico pensar que se necesitara un timer por cada uno de los motores del sistema; no obstante, como ya se mencionó anteriormente, el Arduino Uno solo dispone de 3 timers, por lo que realizando esta estrategia no se podría llevar a cabo la implementación del controlador del Ball And Plate.

Esta limitación ya fue detectada y tratada por otros compañeros en proyectos anteriores relacionados con Arduino. José Antonio Borja Conde en su TFG "Desarrollo de un robot autoequilibrado basado en Arduino con motores paso a paso" aporta un buen enfoque a este problema. Como él mismo explicaba para el robot autoequilibrado en [21]:

> Los motores tendrán distintas velocidades, para poder permitir el giro del cuerpo, por lo que se necesitarán dos señales independientes, es decir, un tren de pulsos para cada motor.
>
> El problema principal de esto era que el Arduino UNO dispone de tres timers, y el hecho de necesitar distintas señales para los motores hacía pensar en un principio que ocuparía el uso de dos de ellos. […]
>
> Tras varias pruebas, se descubrió que había una forma de utilizar el timer 1 para mover los dos motores sin necesidad de que siempre tuviesen la misma velocidad. Para ello hay que hacer uso de las interrupciones A y B del timer, una para cada señal.
>
> La estrategia consiste en que el programa decide el número de incrementos del contador del timer que corresponde al periodo del tren de pulsos, teniendo en cuenta que el timer 1 tiene una resolución de 65536 y se prescala a 8 (32.77ms máximo periodo). De esta forma se tendría npulsA y npulsB, cada una correspondiente a uno de los motores. El timer estará programado en modo normal, por lo que siempre cuenta hasta su máximo valor. Seguidamente se realiza el siguiente proceso:

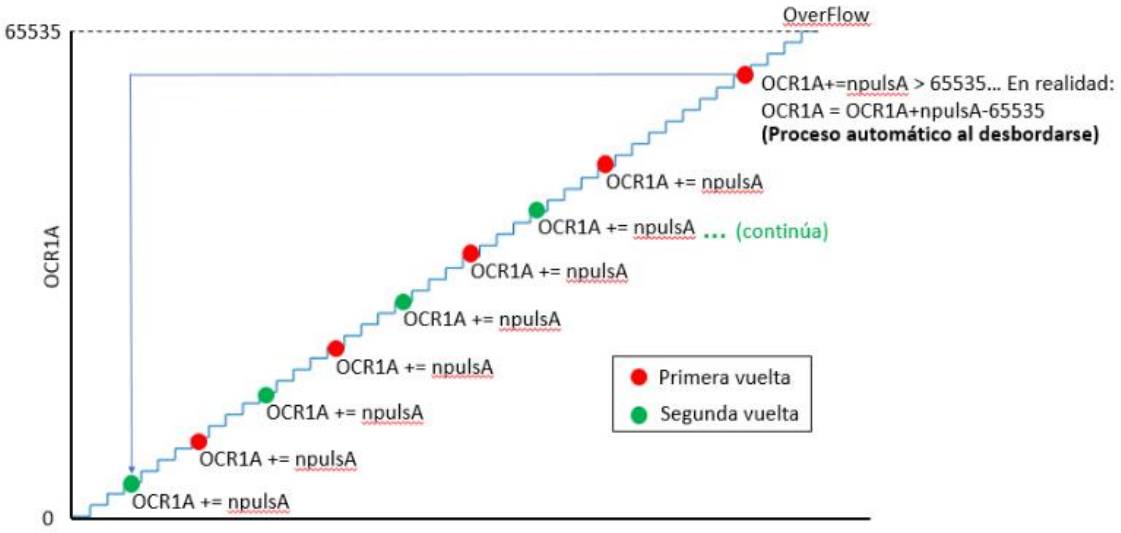


Figura 32 Estrategia tren de pulsos usando una interrupción y timer a OverFlow [21]

Como se puede ver, en la figura solo está representada la interrupción A del timer 1, pero este cuenta con dos, por lo que la B realizaría el mismo proceso.

[…]

Resumiendo, el funcionamiento consiste en autoincrementar el valor con el que salta la interrupción cuando esta se produce. Así se pueden tener hasta 65535 interrupciones A o B desde que el contador del timer es nulo hasta que se realiza el OverFlow.

[…]

Esto soluciona notablemente el problema de generar dos trenes de pulsos distintos para el movimiento de los motores. La única consideración a tener en cuenta es que NpulsA/B no sea mayor que 65535, y que no sea menor que la velocidad que permitan los motores o drivers por sus características físicas.

Tras los buenos resultados que obtuvo José Antonio en el funcionamiento de los motores, en este proyecto se decidió seguir la misma estrategia para la generación de pulsos, aunque adaptándolo a las necesidades concretas de éste.

3.3.1.1 Estrategia seguida

Siguiendo como ejemplo la implementación de José Antonio Borja Conde, para la generación de estos trenes de pulsos con pulsos equidistantes en cada periodo de control se ha usado el timer 2, el cual es de 8 bits, configurado en modo normal (por es por ello que cuenta de 0 a 255). Este timer se ha prescalado a 256, obteniendo el overflow del contador cada 4.086 ms. El cálculo de la velocidad de cada tramo (intervalo entre pulsos) se hará de forma periódica (señalado en Figura 33 como *Ejecución controlador generador de pulsos)*, siendo el intervalo de control de una duración de 8 ms, es decir, cada dos overflows del timer 2.

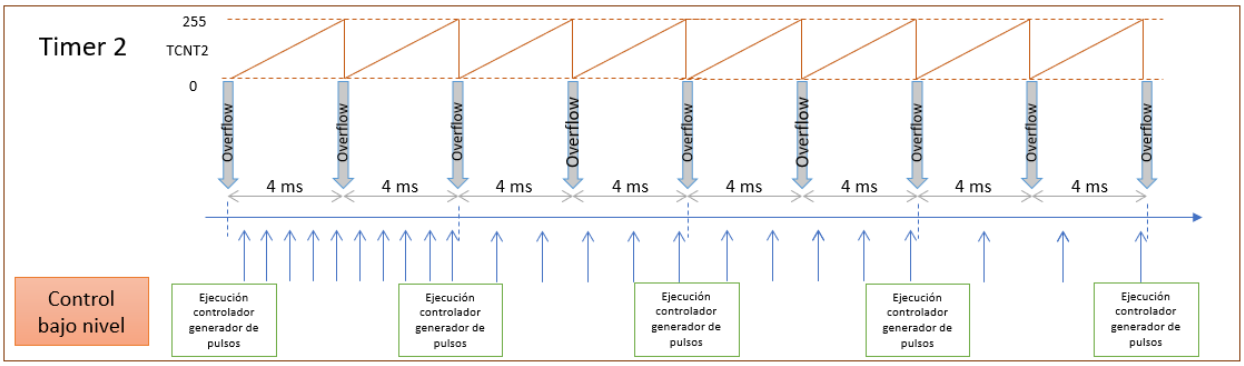


Figura 33 Gestión de interrupciones del Timer 2 para la generación de trenes de pulsos.

La elección de este valor de prescalado y de timer se debe a varias razones:

- El control de los motores tiene que ser muy rápido, de forma que el sistema pueda responder rápido a variaciones en el error de inclinación del ángulo (por ejemplo, cuando hay un cambio en el ángulo de la superficie sobre la que el sistema se apoya, afectando por ello al ángulo de la superficie respecto a la gravedad).
- Al ser el muy pequeño el intervalo de control considerado, ante la pérdida de una interrupción no se ve penalizado el movimiento del motor de forma aparente.
- Aparentemente, el timer 1 nos permitiría una resolución bastante mayor de interrupciones; no obstante, con esta configuración del contador, el timer 2 ya nos permitiría interrupciones de hasta 16 µs, lo cual ya sería suficiente para los requisitos del sistema (se ha comprobado experimentalmente que con interrupciones cada menos de 200 µs se producen una gran pérdida de interrupciones una vez se integran todos los elementos). Es por ello, que usar el timer 1 sería desaprovechar sus posibilidades.

Con el fin de generar los trenes de pulsos, se habilitan las dos interrupciones de comparación A y B, de forma que cuando la cuenta del timer coincida con OCR2A/B se ejecute una subrutina que cambie el estado del pin conectado a la entrada STEP de cada driver.

Para generar las interrupciones que generan los trenes de pulsos, es necesario realizar el cálculo de cuánto habrá que incrementar el valor OCR2A/B de forma repetida. Este cálculo se lleva a cabo cada 2 overflows en el Timer 2 (señalado en Figura 33 como *Ejecución controlador generador de pulsos)*. El procedimiento para calcular este valor se presenta a continuación.

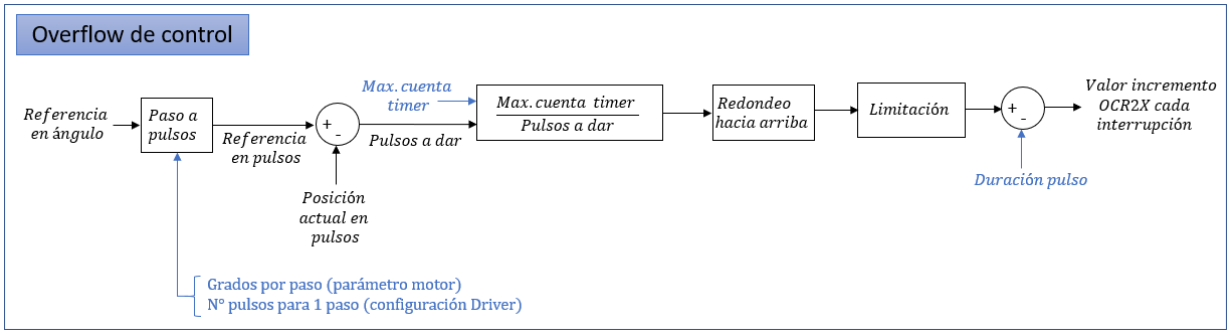


Figura 34 Algoritmo de cálculo del incremento necesario en OCR2X en cada intervalo de control.

En primer lugar, la referencia en ángulo se convierte a número de pulsos del motor usando la configuración del driver y el número de grados por paso completo (parámetro del motor). Tras esto, se resta la posición actual en pulsos del motor, la cual es actualizada de forma interna desde la inicialización del motor (considerando como 0 el final de carrera que posee cada motor en el extremo inferior como se observa en Figura 7). Esto nos daría como resultado el número de pasos que separa al motor de su posición deseada.

Este número de pasos a dar resultante es luego usado para conocer cuál debe ser el valor que habría que incrementar OCR2A/B para crear las interrupciones necesarias hasta llegar a la posición final, coincidiendo además con la ejecución del overflow de control. Para ello, primero se divide el valor máximo del contador (en este caso 255) entre el número de pulsos que habría que dar, obteniendo cada cuantas cuentas del timer 2 se debería dar un pulso.

Hay determinados valores de incrementos de pulsos que no son posibles llevar a cabo en el intervalo de control debido a restricciones temporales del driver o a las propias limitaciones de la programación por interrupciones realizada. Con el fin de impedir una pérdida de pulsos por parte del motor y minimizar la pérdida de interrupciones, el número de cuentas de reloj por pulso obtenido se limita con valores experimentales determinados tras diversas pruebas. En este caso, este valor son 16 cuentas del timer 2, equivalentes a 256 µs.

Por último, a este valor limitado se le resta la duración del pulso configurada (en este caso son 8 cuentas del

temporizador, equivalente a 128 µs). Esto es debido a que, para la creación de un pulso, es necesario primero activar el pin, y luego generar otra interrupción para realizar la desactivación.

Con el fin de facilitar el entendimiento de la generación de las interrupciones, en la siguiente figura se presenta un intervalo de control del motor, en el cual se observa cómo se distribuyen las interrupciones necesarias de forma equidistante una vez realizado el cálculo en el overflow de control. En él se aprecia cómo un intervalo de control se define como el intervalo entre dos cuentas completas de timer 2 (dos overflows). También se incluye el tren de pulsos generado, el cual es entrada del driver del motor paso a paso.

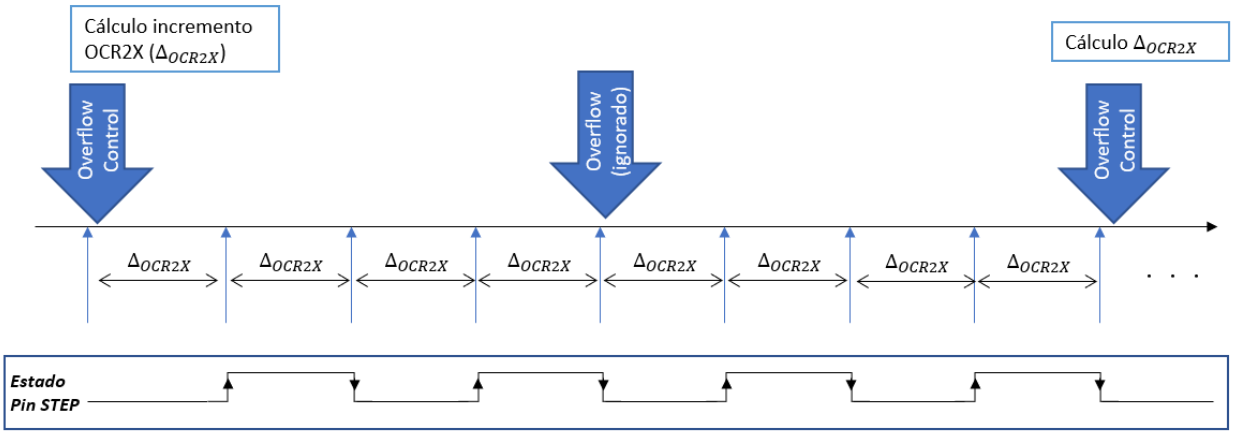


Figura 35 Figura explicativa de la distribución temporal de las interrupciones durante un intervalo completo de control del motor.

Ejemplo 3–2. *Se conoce que la configuración del driver es de 1/8 de paso por pulso y cada 200 pasos el motor da una vuelta completa. El intervalo de control se considera de 2 overflows del timer 2. Se tiene como restricción que el valor mínimo entre pulsos tiene que ser de 16 cuentas de timer 2. Se desea mover el motor 10 grados siguiendo el algoritmo explicado en Figura 34. Se supone el motor en posición inicial.*

Con este ejemplo se pretende ilustrar cómo se calculan los tiempos entre las interrupciones necesarias para generar el tren de pulsos. El objetivo es conseguir realizar el movimiento deseado en el intervalo de control (2 overflows).

En primer lugar, se calcula la referencia en pulsos del motor.

$$Referencia\ en\ pulsos = 10\ grados\ x \frac{200\ pasos}{360\ grados} x \frac{8\ pulsos}{1\ paso} = 44.44\ pulsos\ \cong\ 44\ pulsos$$

Al ser el intervalo de control de $\frac{255\ cuentas\ Timer\ 2}{overflow}\ x\ 2\ overflows$, la cuenta del timer en el intervalo de control será equivalente a 255 cuentas del temporizador. Ahora se calcula el valor que habría entre cada pulso en el tren de pulsos.

$$\frac{\frac{255\ cuentas\ Timer\ 2}{overflow}\ x\ 2\ overflows}{44\ pulsos} = 11.59 \frac{cuentas\ T2}{pulso} \cong 12 \frac{cuentas\ T2}{pulso} \rightarrow No\ cumple\ restricción \rightarrow 16 \frac{cuentas}{pulso}$$

Como se ve, la restricción impuesta en el enunciado es más limitante, por lo que se aumenta el valor a 16 cuentas por pulso dado (el mínimo). Suponiendo la duración del pulso a nivel alto la mitad (8 cuentas), se llega a que el incremento necesario en el registro OCR2X a realizar en cada interrupción es de 8 cuentas del temporizador (una interrupción para poner a nivel alto la señal STEP, y otra para ponerlo a nivel bajo).

Una vez completado el primer intervalo de control, se volverían a ejecutar los cálculos anteriores. Suponiendo que no ha cambiado la referencia de posición del motor, si se vuelven a realizar los cálculos se llegaría a que en el segundo intervalo de control habría que dar los 13 pulsos restantes:

$$Segundo\ intervalo\text{: } 44\ pulsos\ a\ dar - 31\ pulsos\ dados\ en\ primer\ intervalo\ (posición\ actual) = 13\ pulsos$$

$$\frac{\frac{255\ cuentas\ Timer\ 2}{overflow}\ x\ 2\ overflows}{13\ pulsos} = 39.23\frac{cuentas}{pulso} \cong 39\frac{cuentas}{pulso}$$

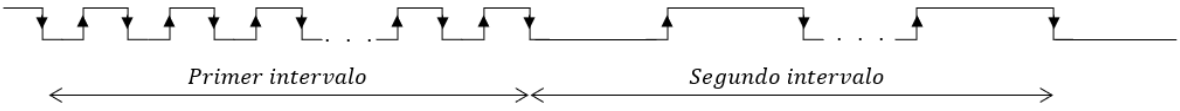


Figura 36 Tren de pulsos generado en la solución del ejemplo propuesto.

Conclusiones

- Se ha comprobado que el ángulo dado no se puede realizar en un sólo intervalo de control, sino que harían falta dos intervalos.
- En el primer intervalo de control como máximo se dan 31 pulsos completos, mientras que en segundo se dan los 13 restantes. En caso de no haber podido realizar los 31 pasos en el primer intervalo (por ejemplo, por pérdida de una interrupción), estos se tendrían en cuenta para el segundo intervalo.
- El intervalo mínimo entre pulsos es de 31 cuentas del timer 2, usando la restricción de 16 cuentas del timer 2 por pulso generado.

$$Primer\ intervalo\text{: } \frac{\frac{255 cuentas}{overflow}\ x\ 2\ overflows}{16\frac{cuentas}{pulso}} = 31.85\ pulsos\ \rightarrow 31\ pulsos\ dados\ (restricción)$$

$$Giro\ máximo\ en\ 8\ ms\ (intervalo\ de\ control)\text{: } 31\ pulsos\ x\frac{1\ pasos}{8\ pulsos}x\ \frac{360\ grados}{200\ pasos} = 6.98\ grados$$

$$Velocidad\ máxima\text{: } \frac{6.98\ grados}{8\ ms}x\frac{1000\ ms}{1\ s}x\ \frac{1\ revolución}{360\ grados} = 2{,}42\ revoluciones\ por\ segundo$$

3.3.2 Implementación en Arduino usando clases

Con el fin de gestionar de la forma más modular posible la generación de los trenes de pulsos para los motores paso a paso se ha creado una clase de tipo Motor, de forma que en ella se encuentran todos los datos y funciones relacionadas con su gestión.

A continuación, se explicarán de forma breve las distintas funciones y datos que están disponibles desde el exterior de una instancia de tipo Motor.

3.3.2.1 Variables accesibles

- `int _RefPulsePosition`: Indica la posición actual del motor en pulsos.
- `unsigned long _CountMotorInterval`: Longitud del intervalo de control del movimiento de los motores, expresado en número de cuentas del Timer usado. En esta implementación, como se usa el timer 2 y el intervalo de control son 2 overflows, entonces este valor es 255 x 2 = 510 cuentas del timer 2.
- `int _PulseWidth`: Ancho de cada pulso del tren de pulsos generado. Se encuentra expresado en cuentas del timer 2.
- `int _BetweenPulses`: Distancia entre el final de un pulso y el inicio del pulso siguiente generado. Se encuentra expresado en cuentas del timer 2.

- `float _StepAngle`: Parámetro del motor paso a paso usado. Indica el número de grados que gira el motor por paso dado.
- `float _MicroStep`: Indica el número de micropasos que conforman el avance de un paso. Depende de la configuración del Driver del motor.
- `int _PulseCycle`: Valor que ha de incrementarse el Compare Match Register correspondiente al timer 2 que se esté usando (OCR2A/B). Este dato se encuentra como público para poder depurar durante su ejecución.

3.3.2.2 Funciones principales accesibles

- `void setPinConfig(int input_StepPin, int input_DirectionPin, int input_LimitSwitchPin)`

Función usada para configurar los pines a los que está conectado el driver del motor paso a paso usado. Como entradas se encuentran los pines de Arduino a los que se encuentran conectados el pin STEP y DIR del EasyDriver, así como el interruptor de final de carrera que se encuentra situado en el motor paso a paso.

- `unsigned int SetAnglePosition(float reference)`

Función que debe ser ejecutada en cada Overflow de control. La función tiene como entrada la referencia de posición del motor, y la convierte a referencia en pulsos teniendo en cuanta los parámetros del driver usado y del motor concreto (MicroStep y StepAngle respectivamente). Esta referencia es luego enviada a la función SetPulsePosition. Como salida, devuelve el valor que es necesario asignar al Compare Match Register (OCR) correspondiente del timer que se esté usando para generar la interrupción (en este caso es OCR2A/B, en función de si es el motor del eje X o Y).

- `unsigned int SetPulsePosition(int reference)`

Realiza el cálculo del incremento de valor que hay que hacer en el Compare Match Register (OCR) en cada interrupción generada. Para ello, tiene en cuenta entre otros la posición actual del motor o la referencia en pulsos (entrada de la función). El procedimiento seguido ya se explicó en Figura 34 Algoritmo de cálculo del incremento necesario en OCR2X en cada intervalo de control. Como salida, devuelve el valor calculado.

- `int PulseGenerator_Vcte(int OCRXX)`

Función que debe ser ejecutada en cada interrupción del comparador correspondiente al motor que se esté controlando. Gestiona el control de los pines STEP y DIR en cada interrupción generada. Tiene como entrada el valor del Compare Match Register correspondiente cuando se ha producido la ejecución de la subrutina (OCRXX) y devuelve como salida el valor que hay que asignar a OCRXX para generar la siguiente interrupción.

- `void InitPosition(float PosInit)`

Inicializa el motor paso a paso. Para ello, el motor retrocede hasta que encuentra la posición de inicio, marcada con el interruptor de final de carrera. Tras esto, avanza hasta la posición inicial en pulsos especificada con la variable de entrada PosInit (en pulsos), y una vez allí, establece en cero las variables de referencia y posición del motor.

3.4 Lectura de posición usando la pantalla táctil

Para la explicación de cómo realizar medidas de la posición de la pelota, se partirá de la información descrita en la sección 1.1.1 Pantalla táctil resistiva, en la cual se describían las características básicas de la pantalla resistiva, así como de la composición y el fundamento de su funcionamiento.

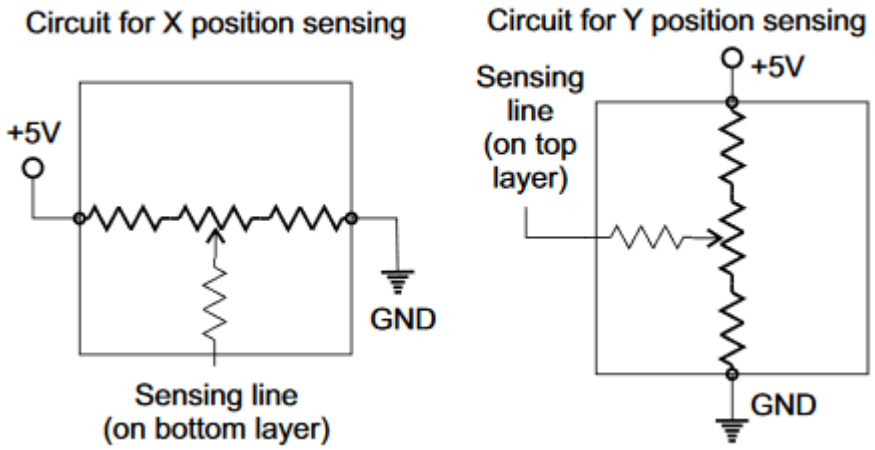


Figura 37 Divisor resistivo equivalente cuando se realiza un toque en la pantalla resistiva [5].

Como se explicó previamente, a efectos prácticos las pantallas táctiles resistivas son divisores resistivos cuya resistencia varía de forma proporcional a la distancia entre el toque realizado y el extremo de las capas impregnadas en el resistivo.

A continuación, primero se explicará la forma de realizar la lectura de ambos ejes con Arduino. Tras esto, se presentan los problemas surgidos en las lecturas y las acciones tomadas para obtener unos datos correctos. Por último, se presenta la implementación final desarrollada en Arduino, incluyendo el uso de las interrupciones del Timer 2.

3.4.1 Lectura de X e Y usando Arduino

El procedimiento que hay que seguir para la lectura de unas coordenadas X e Y está descrito a continuación, con ayuda de la Figura 38. Los terminales X-, X+, Y-, Y+ (respectivamente equivalente en código a X1, X2, Y1, Y2) corresponden a cada uno de los hilos que posee la pantalla, apareciendo indicados en Figura 4 de forma explícita.

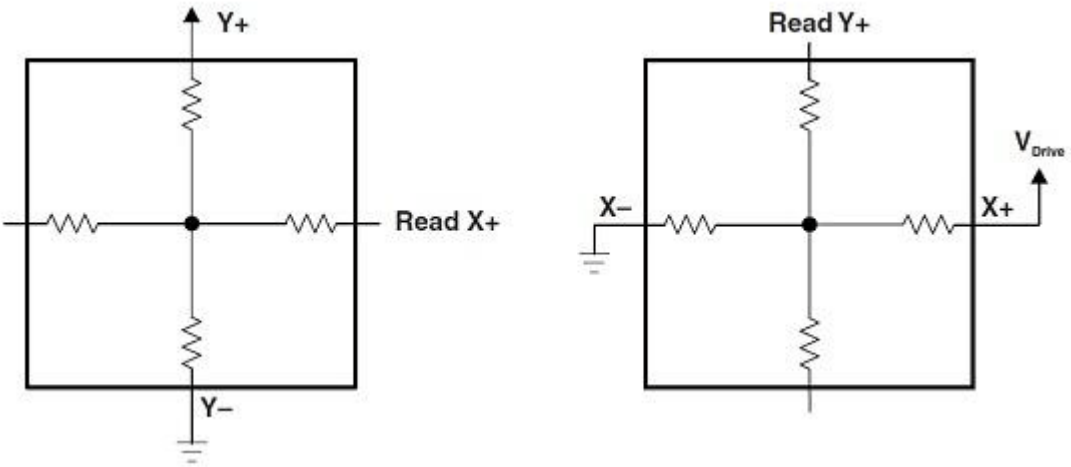


Figura 38 Figura ilustrativa para la lectura de las coordenadas en una pantalla táctil resistiva de 4 hilos.

Las medidas procedentes de la pantalla son necesarias a intervalos constantes, por lo que la ejecución de su lectura se plantea en forma de interrupción, permitiendo realizar otras tareas en los intervalos entre medidas.

Lectura coordenada X

```
//Set X1 as input
pinMode(X1, INPUT);
 //Set X2 as tristate
pinMode(X2, INPUT);
digitalWrite(X2, LOW);
//voltage divider in Y1(+5V) and Y2(GND)
pinMode(Y1, OUTPUT);
digitalWrite(Y1, HIGH);
pinMode(Y2, OUTPUT);
digitalWrite(Y2, LOW);

X_sample = analogRead(Y1)
```

Lectura coordenada Y

```
//Set Y1 as input
pinMode(Y1, INPUT);
//Set Y2 as tristate
pinMode(Y2, INPUT);
digitalWrite(Y2, LOW);
//voltage divider in X1(+5V) and X2(GND)
pinMode(X1, OUTPUT);
digitalWrite(X1, HIGH);
pinMode(X2, OUTPUT);
digitalWrite(X2, LOW);

Y_sample = analogRead(X1)
```

Siguiendo este procedimiento, se obtendría la lectura de la posición de la pelota (X, Y) en un instante, estando los valores recogidos en la lectura entre 0 y 1023.

3.4.2 Problemas en las lecturas

El procedimiento explicado anteriormente es teóricamente correcto, y es la base del sistema de medidas implementado. No obstante, en la práctica se observa que cuando se intentan llevar a cabo las lecturas de ambos ejes de forma secuencial sin ningún tipo de intervalo de retardo entre ellas se comienzan a obtener medidas ruidosas, incluso dándose el caso de medidas invariantes al movimiento de la pelota en la pantalla.

Tras realizar distintas pruebas, se comprueba de forma experimental que este fenómeno está relacionado con el cambio de modo de los pines, siendo necesario un intervalo entre el cambio del modo de funcionamiento de pines y la medida realizada con la función analogRead.

Con el fin de solucionar este problema, se plantea una nueva estrategia para realizar las medidas, la cual consiste en primero cambiar la configuración de los pines, y realizar la lectura del eje en la siguiente interrupción, como se explica en la siguiente Figura 39.

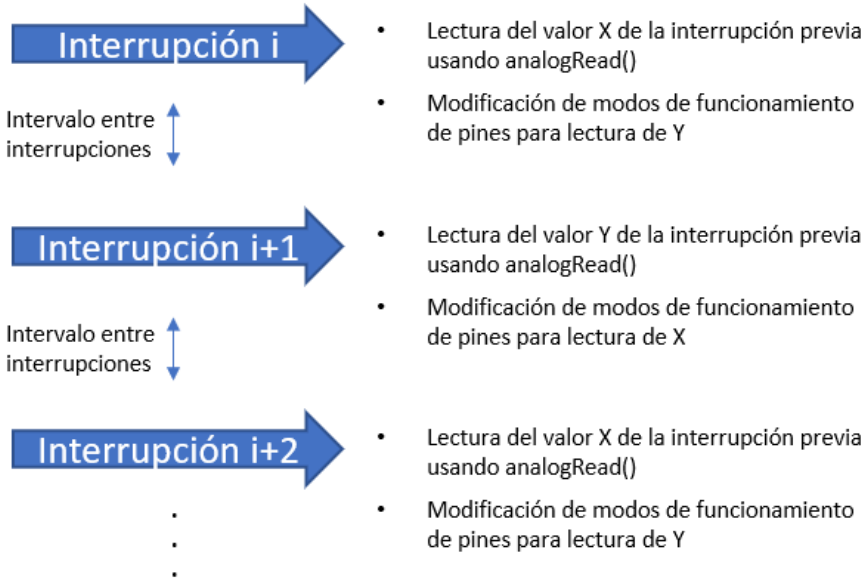


Figura 39 Primera implementación planteada para toma de medidas mediante interrupciones.

No obstante, se siguen apreciando otros fenómenos como la toma de algunas medidas erróneas de forma aleatoria, las cuales son producidas debido a que en ocasiones el movimiento de la pelota impide que el contacto entre las dos capas de la pantalla no se haga correctamente. Estas medidas erróneas afectan de forma negativa al control global del sistema. Como solución a esto se han implementado distintos métodos de filtrado, que permiten omitir estas medidas falsas en el control, o incluso en el peor de los casos, reducirla.

3.4.2.1 Filtrado por mediana

Consiste en realizar en el intervalo de medida un número N de medidas, y tomar como medida filtrada la mediana de ellas. Con esto se consigue un suavizado general de la señal. No obstante, se incluye un retraso en la tendencia de la señal, además de que su implementación es costosa a efectos computacionales al tener que implementar bucles para la ordenación de las medidas.

Una comparación experimental del filtrado por mediana frente a los datos sin filtrar puede ser encontrado en la sección 3.4.4 Resultados experimentales del filtrado en Arduino.

3.4.2.2 Filtro paso bajo de primer orden

Como explica Luis Llamas en [22], consiste en obtener una medida filtrada en función del valor medido, el valor filtrado anterior, y un coeficiente α. Este filtro da como resultado un suavizado de la medida, el cual depende del valor dado al coeficiente α.

$$X_k = \alpha\, M + (1-\alpha\,)X_{k-1} \qquad \begin{matrix} X_k\ \ Valor\ filtrado \\ X_{k-1}\ \ Valor\ filtrado\ anterior \\ M\ \ Valor\ de\ la\ medida\ tomada \\ \alpha\ \ Coeficiente\ que\ varía\ de\ 0\ a\ 1 \end{matrix} \qquad (3\text{-}1)$$

El comportamiento de este filtro dependerá del valor asignado al coeficiente α:

- Si el valor del coeficiente es alto (en torno a 1), la señal entonces presentará poco filtrado, teniendo un peso mayor el valor actual de la señal que el valor filtrado anterior.
- Si el valor del coeficiente es bajo (en torno a 0), el efecto de suavizado será mayor, teniendo más peso el valor filtrado que el valor medido. No obstante, esto también conlleva ralentizar el tiempo

de respuesta del sistema, ya que introduce un retraso entre la señal original y la filtrada.

3.4.2.3 Fundamentos de la programación en Arduino

Tras haber comentado cómo se realiza la medida de la posición de la pelota en el sistema con la pantalla táctil y los distintos filtrados que se van a usar, en este apartado se explica la implementación realizada en código teniendo en cuenta todos los detalles mencionados anteriormente. Ésta también sigue la misma estrategia usada en la parte de la generación de los trenes de pulsos de los motores.

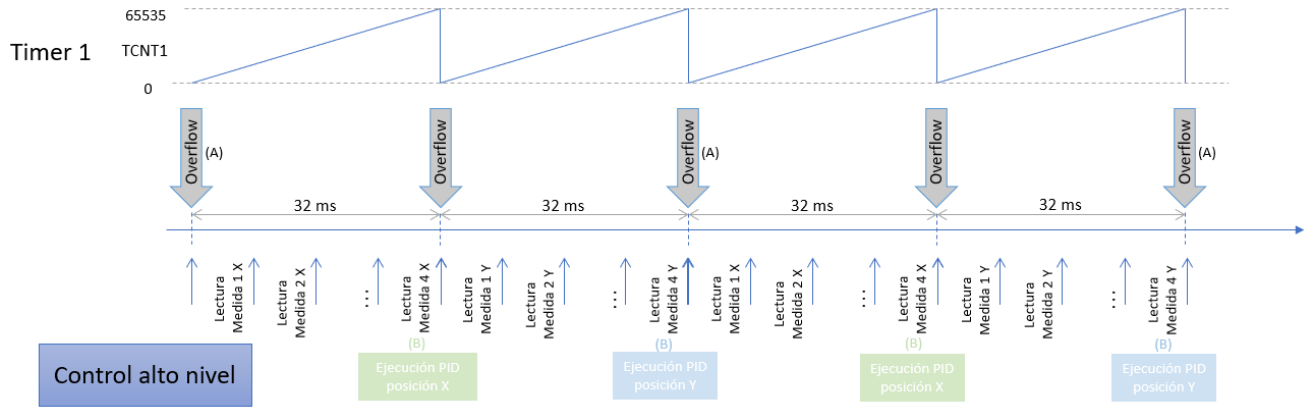


Figura 40 Implementación de lectura de la pantalla táctil usando Timer 1.

Para la implementación de las interrupciones usadas para tomar medidas se hace uso del Timer 1 con prescalador 8 y configurado en modo normal con una interrupción de comparación habilitada. Por ello, cuando la cuenta del timer 1 coincida con OCR1A se ejecutará una subrutina que tomará la medida correspondiente, y modificará el registro OCR1A de nuevo con el fin de preparar la ejecución de la siguiente interrupción.

También tendrá habilitada una interrupción por Overflow, en la cual se llevará a cabo la ejecución del controlador de posición de alto nivel con la medida generada en ese caso. Estos overflows se generan cada 32 ms, y en ellos se alterna la ejecución del controlador de alto nivel del eje X o eje Y; es decir, cada 64 ms se ejecuta el control de cada eje, estando entre ellos desfasados 32 ms.

Para una mejor comprensión de lo explicado anteriormente y la Figura 40 se usa de apoyo la Figura 41, en la que se presenta la implementación real esquematizada.

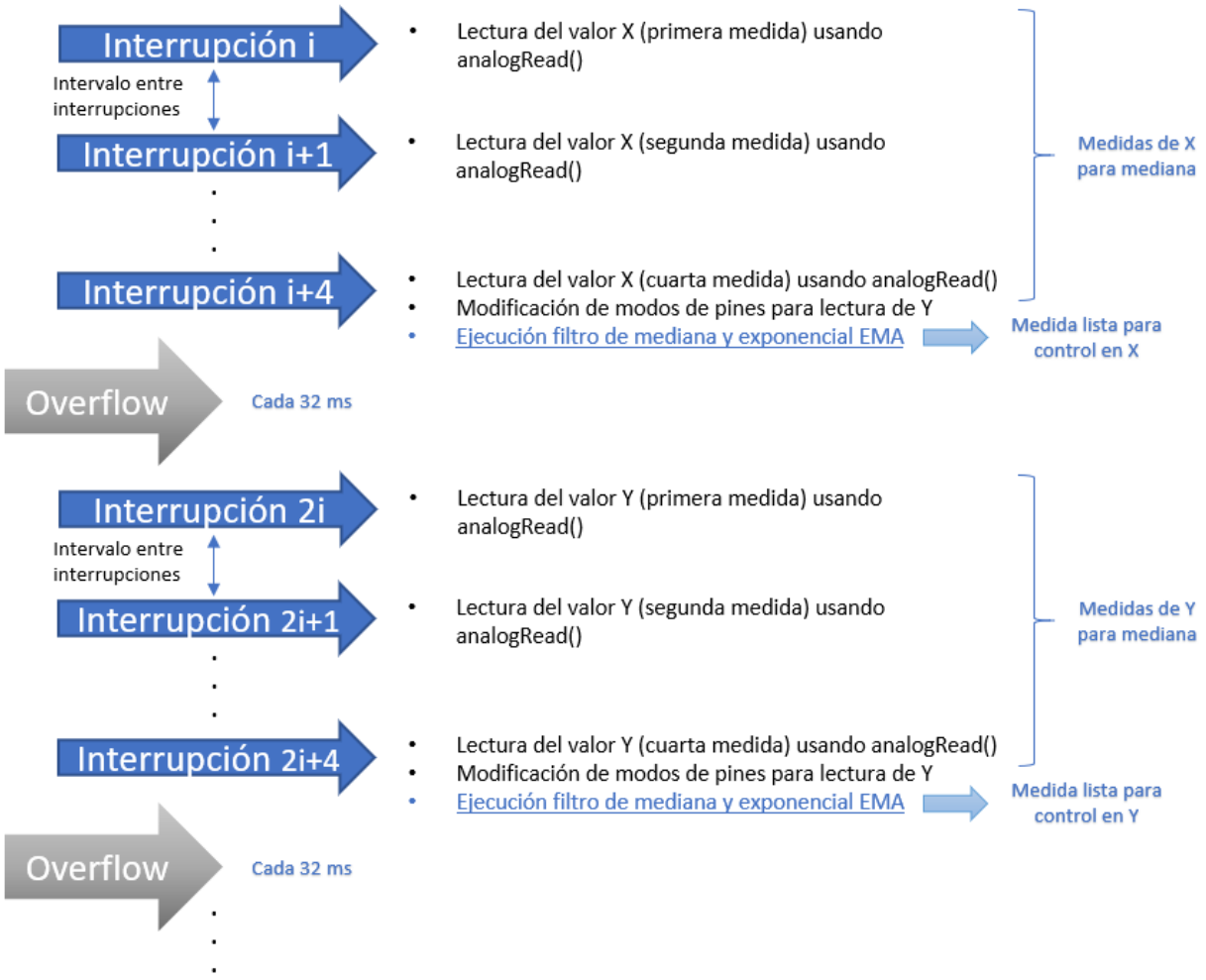


Figura 41 Implementación final de la medida de la posición de la pelota en la programación de Arduino.

Se destacan varios puntos:

- Con el objetivo de evitar el cambio del modo de funcionamiento de los pines en cada interrupción, se realizan primero el conjunto de las lecturas de posición del eje X.
- El tiempo entre interrupciones se calcula dividiendo el tiempo existente entre overflows (en este ejemplo 65535 cuentas del temporizador) entre las medidas a tomar (se ha escogido el valor de 4 medidas). Este tiempo resultante es usado para luego ir aumentando el valor del registro de los comparadores del timer correspondiente en cada ejecución de la subrutina de interrupción.

$$\text{OCIE1A} = \text{OCIE1A} + \frac{65535}{\text{Nmedidas}}$$

  El cálculo del incremento que hay que realizar al registro OCIE1A en cada interrupción se hace cada dos overflows (aquel que coincida después de la lectura de Y).
- Cada vez que se han realizado todas las medidas necesarias de un eje concreto (en el caso de esta implementación son 4) se realiza la mediana, y sobre el valor resultante se aplica el filtro de paso bajo de primer orden. Tras esto, esta medida es usada para el control de alto nivel del eje correspondiente.

De esta forma, se tendrían las medidas listas para ser usadas en el control superior cada 64ms, estando los controladores del eje X y eje Y desfasados 32 ms.

3.4.3 Implementación en Arduino usando clases

Al igual que con la gestión del movimiento de los motores paso a paso, se ha creado una clase de tipo TouchScreen, de forma que en ella se encuentran todos los datos y funciones relacionadas con su gestión.

A continuación, se explicará de forma breve los distintas funciones y datos que están disponibles desde el exterior de una instancia de tipo TouchScreen.

3.4.3.1 Variables principales accesibles

- `bool _EnaLowPassFilter`: Flag que activa el filtro paso bajo de primer orden implementado.
- `float _alphaX, _alphaY`: Valor de alfa, que define la actuación del filtro paso bajo.
- `float _Y_sample, _X_sample`: Valores de la posición en el eje X y en el eje Y, los cuales son usados para el control. Está expresado en mm multiplicados por 100, con el fin de no perder precisión.
- `int _N_Overflow`: Número de overflows que componen un intervalo de medida de la pantalla táctil. Por ejemplo, en este proyecto se usa un intervalo de medida de 2 overflows del Timer 1 (64 ms).

3.4.3.2 Funciones principales accesibles

- `TouchScreen(const uint8_t input_X1, const uint8_t input_X2, const uint8_t input_Y1, const uint8_t input_Y2, volatile uint16_t* OCRXX)`

  Constructor de la clase. Como entrada se incluyen los pines de Arduino que se encuentran conectados a los 4 hilos de la pantalla táctil, así como el puntero a los registros Compare Match Register del timer de 16 bits.

- `void setConfig(int NumberMeas, int N_Overflow, unsigned long MeasInterval, float ScaleFactor_X, float ScaleFactor_Y)`

  Función usada para configurar todo lo relacionado con las medidas que se tomarán en cada intervalo. Como entradas se encuentra el número de medidas a realizar para el filtro de la mediana, la duración del intervalo de medida en overflows, el número máximo de cuentas del Timer que se use (65535) y el factor de escala necesario para pasar las medidas a mm x 100.

- `void EnableLowPassFilter(bool enable, float alphaX, float alphaY)`

  Habilita el filtro paso bajo con los valores de alfa especificados en las entradas.

- `void SetNumberMeas()`

  Función que debe ser ejecutada en cada Overflow de control. Realiza de forma interna el cálculo del incremento de valor que hay que hacer en el Compare Match Register (OCR) para generar las interrupciones necesarias, y asigna el valor necesario en el registro OCR para generar la primera interrupción.

- `int MeasInterruptGenerator()`

  Función que debe ser ejecutada en cada interrupción del comparador correspondiente a la pantalla táctil. Gestiona la toma de lectura de la pantalla táctil, así como el cambio de modo de los pines. Una vez toma todos los valores necesarios de las muestras, también aplica el filtro de la mediana y paso bajo en caso de que esté activo.
  El valor de retorno de la función además indica el control que hay que ejecutar en la interrupción:

  - 0: No hay que ejecutar control de posición de la pelota.
  - 1: Lectura filtrada del eje X está lista para realizar el control de posición.
  - 2: Lectura filtrada del eje Y está lista para realizar el control de posición.

3.4.4 Resultados experimentales del filtrado en Arduino

3.4.4.1 Filtro usando la mediana

Se ha llevado a cabo una comparación entre la medida obtenida de la lectura de la pantalla táctil y la medida resultante del filtrado de la mediana de forma experimental en el sistema. Para ello:

- Se ha usado la implementación explicada en Figura 40, y se ha representado en tiempo real la última medida en el eje Y (la cuarta medida del intervalo, la cual es tomada antes de la ejecución del PID de control del eje Y) como medida sin filtrar y la medida obtenida tras aplicar el filtro de la mediana.
- Con el fin de provocar una lectura más ruidosa, se ha realizado la prueba con una pelota más ligera a la usada finalmente en el funcionamiento del sistema. Esto hace que haya momentos en los que no se produzcan contactos entre las capas de la que está constituida la pantalla, y se provoquen lecturas erróneas.

A continuación, se presenta el resultado de la prueba, siendo cuatro el número de medidas por intervalo.

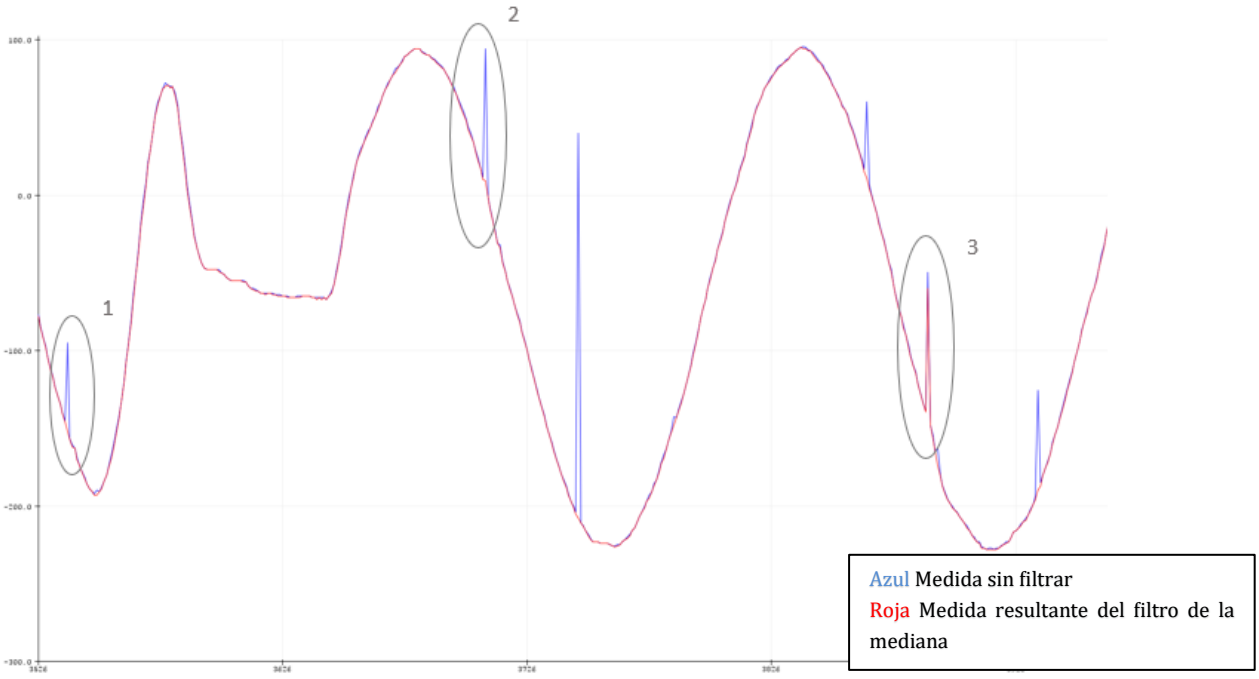


Figura 42 Comparativa de la medida usando el filtro de la mediana.

- Se aprecia como se producen lecturas con valores falsos en la medida sin filtrar, los cuales en la mayoría de los casos son corregidos tras el filtrado de la mediana (casos 1 y 2).
- Hay casos en los que todas las medidas realizadas en el intervalo son de valor erróneo. En estos casos, el filtrado de los datos por mediana no aporta una mejora real, como en el caso 3.

3.4.4.2 Filtro paso bajo sobre la mediana

Se ha llevado a cabo una comparación entre la medida obtenida en el apartado anterior y la medida resultante tras aplicarle un filtro paso bajo de primer orden a la medida ya filtrada con la mediana. Esta prueba se ha realizado bajo las mismas condiciones que en la prueba anterior.

- Se ha usado la implementación explicada en Figura 40, y se ha representado en tiempo real la última medida en el eje Y (la cuarta medida del intervalo, la cual es tomada antes de la ejecución del PID de control del eje Y) como medida sin filtrar y la medida obtenida tras aplicar el filtro de la mediana.

- Con el fin de provocar una lectura más ruidosa, se ha realizado la prueba con una pelota más ligera a la usada finalmente en el funcionamiento del sistema.
- El valor α del filtro paso bajo de primer orden es igual a 0.60.

A continuación, se muestra el resultado de la prueba.

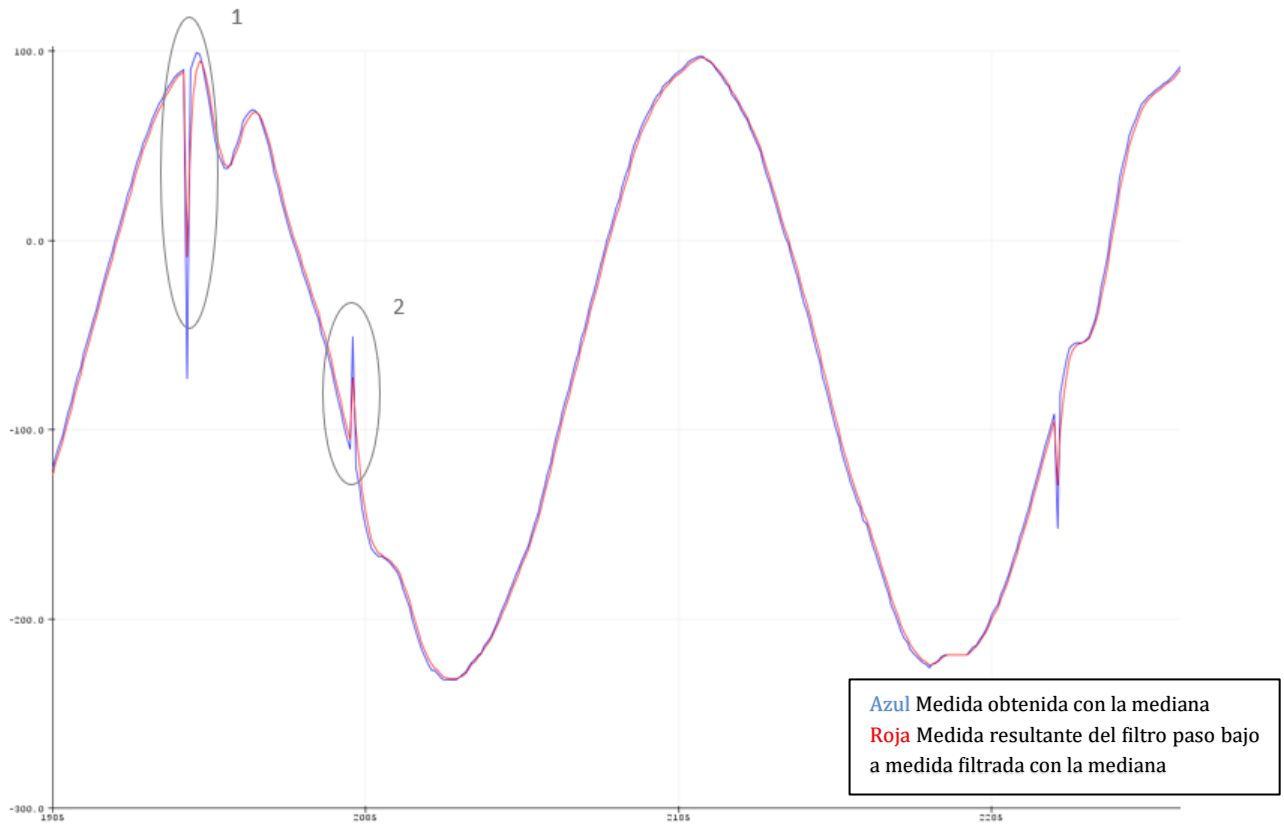


Figura 43 Influencia del filtrado paso bajo con α=0.60 sobre el filtrado de la mediana.

- Se observa como los picos que no pudieron evitarse en el filtro usando el valor de la mediana se suavizan (como en los casos 1 y 2), reduciendo su valor. No obstante, seguirán provocando perturbaciones al sistema.
- Se observa un retardo en la medida de la posición de la medida, como se comentó en el apartado 3.4.2.2.

3.5 Lectura de ángulos de la IMU MPU-6050

Como en las anteriores secciones, se toma como punto de partida la explicación de las distintas formas de lectura posibles del ángulo de la superficie, localizada en la sección 1.1.5 IMU MPU-6050.

Como se explicó anteriormente, la medida de la IMU MPU-6050 puede ser obtenida partiendo de los datos del acelerómetro, con los datos del giroscopio, o mediante una combinación de ambas, con el fin de reducir los errores intrínsecos asociados a cada método de medida.

En este proyecto se ha decidido llevar a cabo la lectura del ángulo a través del DMP integrado (procesador digital de movimiento) debido a que proporciona mejores resultados que un filtro complementario y libera al Arduino de la tarea del filtrado, como explica Luis Llamas en [18].

Para la comunicación con la IMU MPU-6050 y el DMP se usará la comunicación I2C, por el cual se recibirán los datos requeridos. Para esto, se han usado librerías desarrolladas por Jeff Rowberg, entre las que se encuentra una librería específica para el uso de MPU-6050 [23], así como la librería i2cdevlib, la cual mejora la librería

I2C [24]. Más información acerca del proyecto de Jeff Rowberg puede ser encontrada en [25].

Con el fin de leer los datos generados por el DMP es necesario basar las lecturas en interrupciones asociadas a pines del Arduino. El DMP genera el dato del ángulo de la superficie en la que se desplaza la pelota respecto a la gravedad en función de un procesado de los datos obtenidos por el acelerómetro y el giroscopio. Tras esto, estos datos son enviados por el DMP mediante I2C, y la MPU-6050 notifica la disponibilidad de la lectura de los datos usando el pin INT.

Con el fin de conocer cuándo hay un nuevo dato disponible, se asocia una interrupción de un pin de entrada de Arduino a esta señal generada por la IMU, con el fin de realizar la gestión de la lectura de datos de la forma más óptima posible (se realiza sólo la espera de datos cuando la MPU así lo ha notificado).

3.5.1 Explicación del código

En primer lugar, en la parte del setup del código, primero se abre la comunicación I2C como máster, se fija la señal de reloj que rige las comunicaciones, y se comprueba la conexión. Una vez realizado esto, se envían los valores de calibración y se activa el DMP.

Una vez activado, se hace una petición al DMP para obtener la longitud de los paquetes de datos que se recibirán por comunicaciones, el cual será usado durante la ejecución del programa para conocer si han llegado todos los paquetes al buffer.

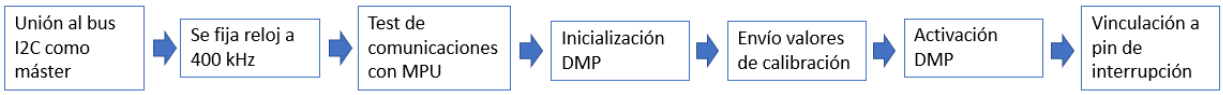


Figura 44 Inicialización necesaria en setup() de la programación.

Por último, se vincula la interrupción del pin conectado físicamente a la IMU a la ejecución de una subrutina consistente en una función sencilla, la cual sólo activa una variable bandera (mpuInterrupt) que indica que existen datos para leer en el buffer.

El DMP actualiza los datos que genera por defecto a una frecuencia de 100Hz, lo cual significa que cada 10ms la IMU generará una interrupción para indicar que existen datos disponibles para el envío .

La estructura de los paquetes recibidos por I2C son mostrados a continuación. En ella se encuentran la lectura de los ángulos de la IMU, que están expresados en quaternions (QUAT W, QUAT X, QUAT Y y QUAT Z); las medidas correspondientes al giroscopio y las medidas del acelerómetro. De forma interna, estos valores son luego pasados a unidades del sistema internacional.

```
/* ============================================================================================== *
 | Default MotionApps v2.0 42-byte FIFO packet structure:                                         |
 |                                                                                                |
 | [QUAT W][      ][QUAT X][      ][QUAT Y][      ][QUAT Z][      ][GYRO X][      ][GYRO Y][      ] |
 |   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  |
 |                                                                                                |
 | [GYRO Z][      ][ACC X ][      ][ACC Y ][      ][ACC Z ][      ][      ]                       |
 |  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41                        |
 * ============================================================================================== */
```

3.5.1.1 Gestión de las interrupciones

En este subapartado se explica cómo se realiza la gestión de las interrupciones asociadas a los pines del Arduino para la posterior lectura del dato. Existen dos pasos en la gestión de la interrupción:

- Interrupción asociada al pin del Arduino: Cuando se produce el evento que genera la interrupción (en este caso, el flanco de subida generado por el DMP de la IMU), se ejecuta la correspondiente subrutina, la cual sólo fija a TRUE la variable global *mpuInterrupt*. Esta es usada para indicar que existen datos disponibles para la lectura.

- Lectura del buffer de las comunicaciones: Esta segunda parte no se lleva a cabo durante la interrupción, sino en la parte del main loop() de la programación del Arduino. En ella se espera a que se recepcione el mensaje completo, y tras esto los datos recibidos se almacenan y se realiza la conversión a ángulos.

En la Figura 45 se presenta a continuación un esquema de la programación seguida en el loop() para la lectura del buffer de comunicaciones, con la finalidad de clarificar el procedimiento seguido.

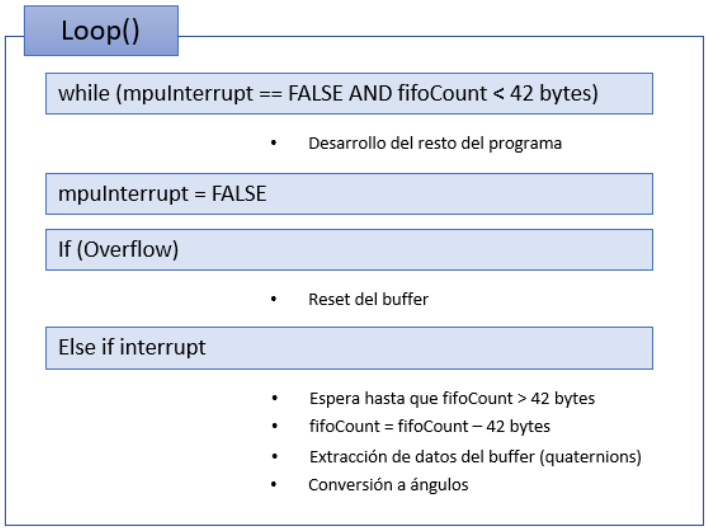


Figura 45 Gestión de las interrupciones durante el loop() de la programación.

Primero, en cada iteración hay que comprobar si hay nuevos datos disponibles generados por el DMP (a través de la variable bandera *mpuInterrupt*) o si hay aún paquetes completos pendientes por leer (siendo fifoCount el número de bytes sin leer en el buffer). En caso afirmativo, entonces se procede a la lectura de los datos y a su conversión en ángulos. En caso contrario, se sigue ejecutando el resto del programa.

Es necesario también destacar la gestión que se debe de hacer con el buffer en las comunicaciones, de forma que no se produzcan pérdidas de datos debido a overflows del buffer. Para evitar esto, es importante que el DMP tenga configurada una frecuencia de actualización de datos acorde a las posibilidades del Arduino Uno.

3.5.2 Aumento de velocidad de lectura

Durante las primeras integraciones realizadas con la IMU MPU-6050 se hizo evidente que era necesario una mayor tasa de actualización de las medidas de los ángulos de inclinación X e Y de la superficie del sistema.

Como se explicó anteriormente en la sección 2.1 Esquema de control del sistema, el sistema consta de dos niveles de control jerárquicos. En el nivel superior, se encuentra el control de la posición de la pelota y su trayectoria en la superficie, mientras que en el nivel inferior se encuentra el controlador de inclinación de la superficie en función de las referencias fijadas por el nivel superior. Con el fin de conseguir una buena respuesta general del sistema, el nivel inferior debe tener una frecuencia de ejecución del control mayor al nivel superior. Para ello, por tanto, es también necesario una mayor frecuencia en la obtención de los datos.

Dentro de la librería usada *MPU6050_6axis_MotionApps20.h* creada por Jeff Rowberg [26] se fijan los valores de configuración de funcionamiento para el DMP de la IMU MPU-6050, como se puede ver en Figura 46. Estos valores definidos en este archivo son luego usados para inicializar las funcionalidades del DMP con la función correspondiente de la librería.

Dentro de esta estructura de configuración del DMP, se encuentra la tasa de datos del buffer FIFO (*MPU6050_DMP_FIFO_RATE_DIVISOR,* cuyo define está remarcado en la parte superior de Figura 46). Ésta se encuentra con valor por defecto 0x01, equivalente a 100Hz como se explica en los comentarios de la librería remarcadas en la parte inferior de la Figura 46.

La frecuencia de actualización de los datos de 100 Hz resulta en una recepción de datos alrededor de 10 ms, lo cual es una tasa de refresco baja para poder ejecutar el control de los motores cada 8 ms como se planteó en el apartado 3.3 Movimiento de motores paso a paso. Como solución, y siguiendo las recomendaciones del creador de la librería, se aumenta la tasa de refresco de los datos hasta los 200Hz, ya que aumentar más de este valor puede resultar en la obtención de datos con ruido. Para ello, se establece el define de *MPU6050_DMP_FIFO_RATE_DIVISOR* con valor 0x00.

Se comprueba experimentalmente con la realización de pruebas con temporizadores, que el periodo de refresco antes del cambio se encontraba en torno a los $10 \pm 0.4\ ms$, mientras que, tras el cambio, el periodo de refresco se reduce a $5 \pm 0.4\ ms$. Con este cambio, por ello, se permite que ahora sí se puede llevar a cabo la ejecución del control del ángulo de la pantalla en periodos de 8ms.

```
#ifndef MPU6050_DMP_FIFO_RATE_DIVISOR
#define MPU6050_DMP_FIFO_RATE_DIVISOR 0x01
#endif

// thanks to Noah Zerkin for piecing this stuff together!
const unsigned char dmpConfig[MPU6050_DMP_CONFIG_SIZE] PROGMEM = {
//  BANK    OFFSET  LENGTH  [DATA]
    0x03,   0x7B,   0x03,   0x4C, 0xCD, 0x6C,         // FCFG_1 inv_set_gyro_calibration
    0x03,   0xAB,   0x03,   0x36, 0x56, 0x76,         // FCFG_3 inv_set_gyro_calibration
    0x00,   0x68,   0x04,   0x02, 0xCB, 0x47, 0xA2,   // D_0_104 inv_set_gyro_calibration
    0x02,   0x18,   0x04,   0x00, 0x05, 0x8B, 0xC1,   // D_0_24 inv_set_gyro_calibration
    0x01,   0x0C,   0x04,   0x00, 0x00, 0x00, 0x00,   // D_1_152 inv_set_accel_calibration
    0x03,   0x7F,   0x06,   0x0C, 0xC9, 0x2C, 0x97, 0x97, 0x97, // FCFG_2 inv_set_accel_calibration
    0x03,   0x89,   0x03,   0x26, 0x46, 0x66,         // FCFG_7 inv_set_accel_calibration
    0x00,   0x6C,   0x02,   0x20, 0x00,               // D_0_108 inv_set_accel_calibration
    0x02,   0x40,   0x04,   0x00, 0x00, 0x00, 0x00,   // CPASS_MTX_00 inv_set_compass_calibration
    0x02,   0x44,   0x04,   0x00, 0x00, 0x00, 0x00,   // CPASS_MTX_01
    0x02,   0x48,   0x04,   0x00, 0x00, 0x00, 0x00,   // CPASS_MTX_02
    0x02,   0x4C,   0x04,   0x00, 0x00, 0x00, 0x00,   // CPASS_MTX_10
    0x02,   0x50,   0x04,   0x00, 0x00, 0x00, 0x00,   // CPASS_MTX_11
    0x02,   0x54,   0x04,   0x00, 0x00, 0x00, 0x00,   // CPASS_MTX_12
    0x02,   0x58,   0x04,   0x00, 0x00, 0x00, 0x00,   // CPASS_MTX_20
    0x02,   0x5C,   0x04,   0x00, 0x00, 0x00, 0x00,   // CPASS_MTX_21
    0x02,   0xBC,   0x04,   0x00, 0x00, 0x00, 0x00,   // CPASS_MTX_22
    0x01,   0xEC,   0x04,   0x00, 0x00, 0x40, 0x00,   // D_1_236 inv_apply_endian_accel
    0x03,   0x7F,   0x06,   0x0C, 0xC9, 0x2C, 0x97, 0x97, 0x97, // FCFG_2 inv_set_mpu_sensors
    0x04,   0x02,   0x03,   0x0D, 0x35, 0x5D,         // CFG_MOTION_BIAS inv_turn_on_bias_from_no_motion
    0x04,   0x09,   0x04,   0x87, 0x2D, 0x35, 0x3D,   // FCFG_5 inv_set_bias_update
    0x00,   0xA3,   0x01,   0x00,                     // D_0_163 inv_set_dead_zone
                  // SPECIAL 0x01 = enable interrupts
    0x00,   0x00,   0x00,   0x01, // SET INT_ENABLE at i=22, SPECIAL INSTRUCTION
    0x07,   0x86,   0x01,   0xFE,                     // CFG_6 inv_set_fifo_interupt
    0x07,   0x41,   0x05,   0xF1, 0x20, 0x28, 0x30, 0x38, // CFG_8 inv_send_quaternion
    0x07,   0x7E,   0x01,   0x30,                     // CFG_16 inv_set_footer
    0x07,   0x46,   0x01,   0x9A,                     // CFG_GYRO_SOURCE inv_send_gyro
    0x07,   0x47,   0x04,   0xF1, 0x28, 0x30, 0x38,   // CFG_9 inv_send_gyro -> inv_construct3_fifo
    0x07,   0x6C,   0x04,   0xF1, 0x28, 0x30, 0x38,   // CFG_12 inv_send_accel -> inv_construct3_fifo
    0x02,   0x16,   0x02,   0x00, MPU6050_DMP_FIFO_RATE_DIVISOR // D_0_22 inv_set_fifo_rate

    // This very last 0x01 WAS a 0x09, which drops the FIFO rate down to 20 Hz. 0x07 is 25 Hz,
    // 0x01 is 100Hz. Going faster than 100Hz (0x00=200Hz) tends to result in very noisy data.
    // DMP output frequency is calculated easily using this equation: (200Hz / (1 + value))

    // It is important to make sure the host processor can keep up with reading and processing
    // the FIFO output at the desired rate. Handling FIFO overflow cleanly is also a good idea.
};
```

Figura 46 Parámetros de funcionamiento en MPU6050_6Axis_MotionApps20.h [26]

3.6 Resultados de la implementación

En este apartado se presentan algunos resultados de seguimientos de posición de la pelota en el sistema Ball And Plate. Los valores de las constantes de los controladores PID han sido hallados de forma experimental, siendo del mismo valor en ambos ejes.

Controlador PID Superior - Posición de la pelota	Controlador PID Inferior – Ángulo de la pantalla
$K_p = 0.044 \quad K_i = 0 \quad K_d = 0.55$	$K_p = 0.6 \quad K_i = 0 \quad K_d = 0.2$

Antes de presentar los resultados, es necesario comentar que el controlador no es capaz de detectar cuándo se encuentra la pelota encima de la superficie, ya que la medida de la presión no se ha implementado en este sistema (conlleva la gestión de aún más interrupciones).

Cuando la pelota no se encuentra sobre la pantalla táctil, los datos leídos de la pantalla muestran una posición variante alrededor de cero. Es por ello que, si existiese término integral, este término tendría mucho peso en el cálculo de la acción de control. Este provocaría una saturación en la acción de control, haciendo que el sistema no respondiese durante un periodo una vez la pelota se encuentre sobre la pantalla. Como se puede comprobar en los resultados de la implementación que se verán a continuación, el seguimiento de posición en régimen permanente es bastante bueno, por lo que no se considera necesario incluir un término integral en la acción de control.

A continuación, se presentan los resultados de algunas pruebas realizadas, con el fin de comprobar el comportamiento del sistema controlado ante perturbaciones externas.

Cambio de posición fija a seguimiento de trayectoria

Se parte inicialmente con el sistema controlando la posición de la pelota en la posición central de la pantalla. En un instante concreto, se cambia la referencia al seguimiento de una trayectoria circular.

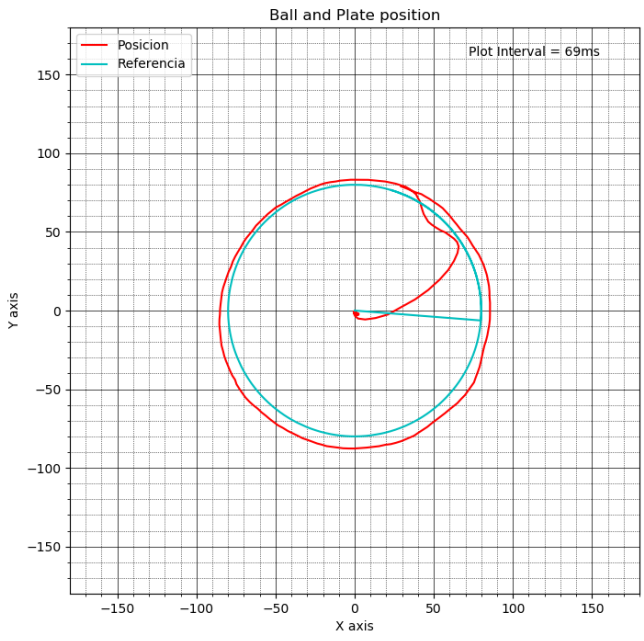


Figura 47 Comportamiento del sistema ante cambio de referencia.

Cambio de seguimiento de trayectoria circular a lemniscata

Se parte inicialmente del sistema siguiendo una trayectoria circular. A continuación, se cambia la trayectoria al seguimiento de una lemniscata.

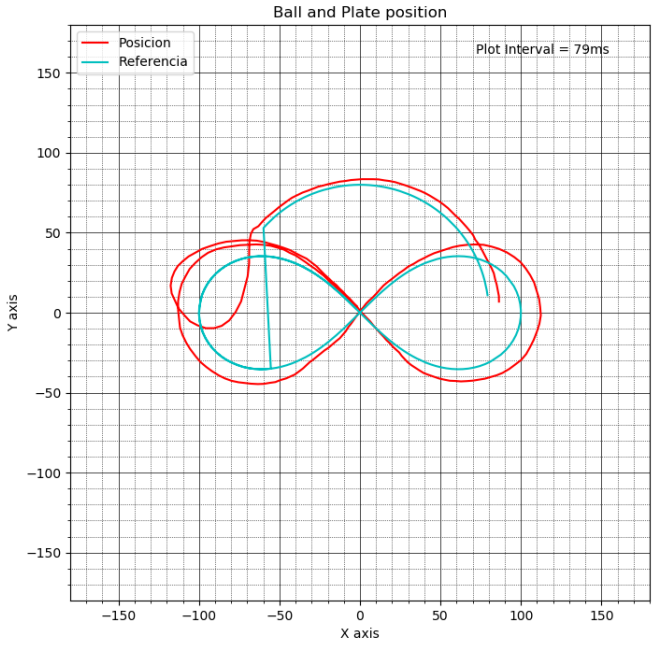


Figura 48 Comportamiento del sistema ante cambio de trayectoria.

Rechazo a perturbaciones controlador de bajo nivel

Se parte inicialmente del sistema siguiendo una trayectoria circular. En un instante concreto, se eleva de forma repentina uno de los extremos del tablero (correspondiente al eje Y) en el que se encuentra apoyado el sistema. En el sistema se aprecia como se produce una perturbación (señalada en la figura), la cual es corregida gracias al controlador de bajo nivel que se encarga de mantener el ángulo ordenado por el controlador de alto nivel.

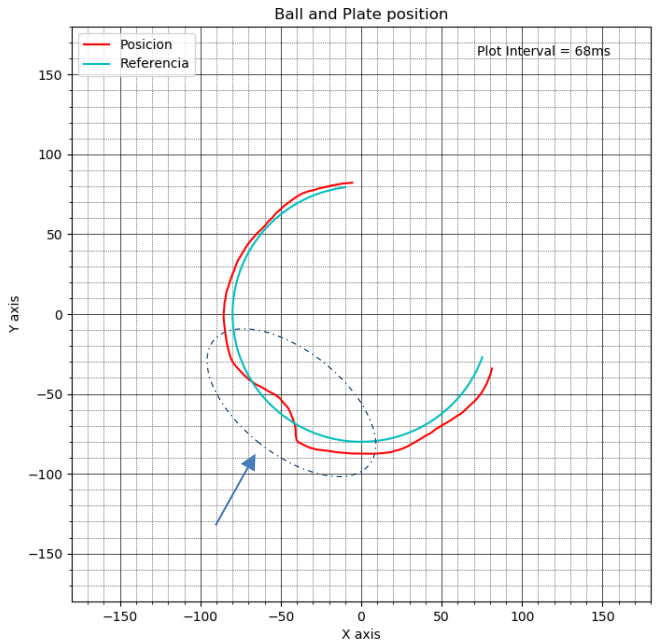


Figura 49 Comportamiento del sistema ante cambio en la base del sistema.

Rechazo a perturbaciones controlador alto nivel

Se parte inicialmente del sistema siguiendo una trayectoria circular. En un instante concreto, se golpea la pelota,

produciendo una perturbación en el sistema y alejándolo de la referencia. Se aprecia como la acción del controlador corrige la trayectoria de la pelota, y la lleva de nuevo a la trayectoria circular.

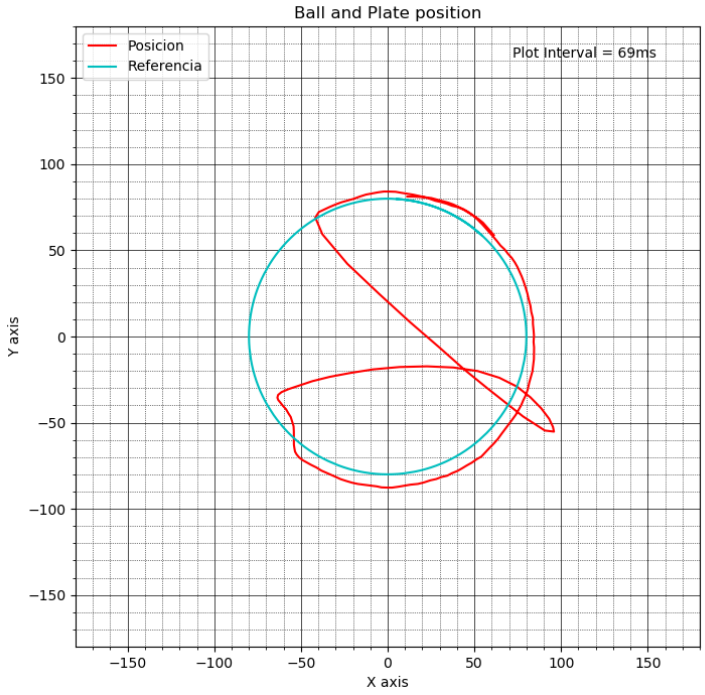


Figura 50 Comportamiento del sistema ante perturbación externa.

4 VISUALIZACIÓN EN TIEMPO REAL

Junto a la implementación que se ha realizado en el controlador Arduino, se ha llevado a cabo la creación de una aplicación de monitorización en la que es posible observar la posición de la pelota en el sistema en tiempo real, así como de la referencia.

Esta aplicación se ha desarrollado en Python, el cual es un lenguaje de programación interpretado y orientado a objetos. Los principales motivos por las que se ha decidido usar este lenguaje son:

- Gran diversidad de librerías, paquetes y APIs de código libre disponibles en la red, los cuales permiten abstraerse de las capas inferiores de programación.
- Gran soporte de la comunidad en el desarrollo de aplicaciones y proyectos de tipo DIY (Do It Yourself).

En primer lugar, se lleva a cabo una breve explicación de la aplicación realizada. Tras ésta, se procede a explicar los fundamentos de la programación orientada a objetos realizada en Python, incluyendo los módulos y librerías usadas, así como la forma de comunicación entre el Arduino y la interfaz de Python.

4.1 Ventana desarrollada

La aplicación desarrollada consiste en un script de Python el cual al ejecutarse realiza la apertura del puerto serie con el Arduino Uno, y tras esto, representa en la interfaz gráfica los datos de monitorización recepcionados procedentes de Arduino. La programación ha sido basada en dos scripts de Python ya existentes en [27] y [28], en los cuales se representaban las medidas en tiempo real de una unidad de medidas inerciales (como la que usa el propio sistema Ball And Plate) en una interfaz gráfica.

A continuación, se procede a realizar la presentación de la ventana desarrollada en Python.

1. Representación de la trayectoria de la pelota en la superficie frente a la consigna de trayectoria en tiempo real del sistema Ball And Plate. La trayectoria está formada por los últimos 50 valores de referencia y posición usados en el control y enviados por el Arduino. El número de datos almacenados para su representación también puede ser configurado.
2. En estas gráficas es posible encontrar la evolución de la posición del eje X o Y en los últimos valores recibidos por comunicaciones.
3. Selección de trayectoria. Es posible en ella cambiar la trayectoria del sistema Ball And Plate entre una de las predefinidas por defecto pulsando en el pulsador correspondiente. Las trayectorias definidas son circunferencia, lemniscata o control en el centro de origen de la superficie.

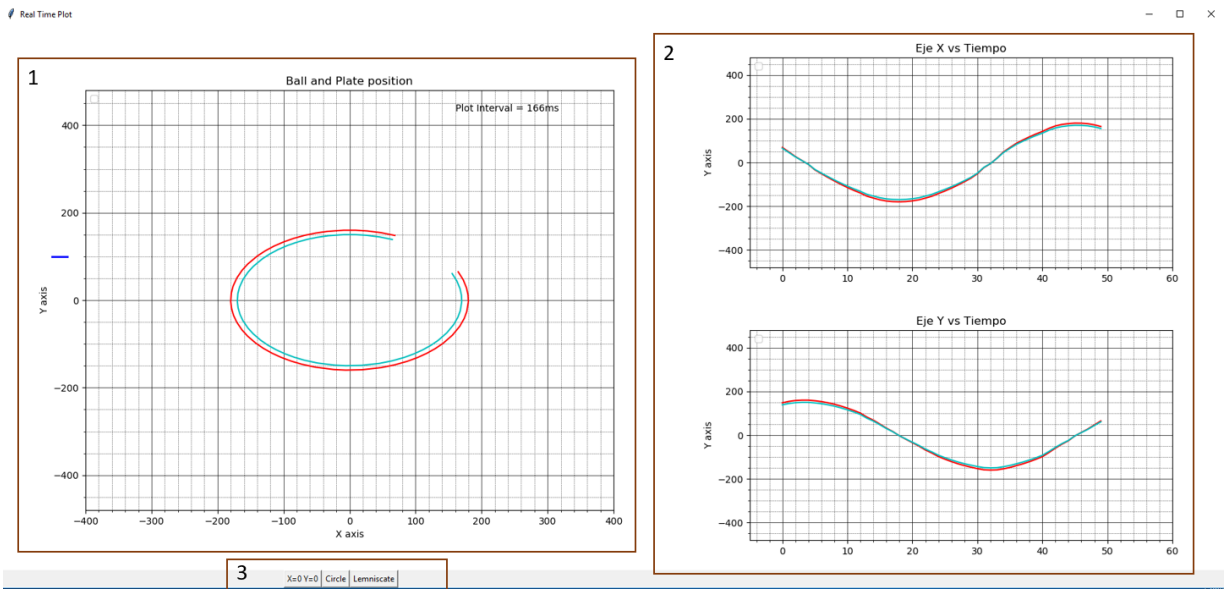


Figura 51 Aspecto de la ventana de la aplicación de monitorización.

4.2 Programación de la interfaz

En esta sección, primero se presentan los módulos usados y la idea principal seguida para plantear la estructura de la aplicación. Tras esto, se explican algunos detalles más concretos como la descripción de las clases que se han definido para la gestión de la interfaz gráfica o cómo se ha realizado la comunicación entre el Arduino y Python.

4.2.1 Módulos y librerías usadas

Para llevar a cabo el desarrollo de la programación, los principales módulos que se usan son las siguientes.

- Threading [29]: Módulo que permite la gestión de hilos haciendo de interfaz del módulo _thread incluido en Python, el cual gestiona los hilos en bajo nivel.
- MatPlotLib [30]: Librería que permite representaciones de datos en forma de diversos formatos: gráficas, histogramas, diagramas de barras, imágenes, representaciones de campos escalares y campos vectoriales, etc. Se sitúa como una alternativa libre a programas como Matlab en la representación de datos.

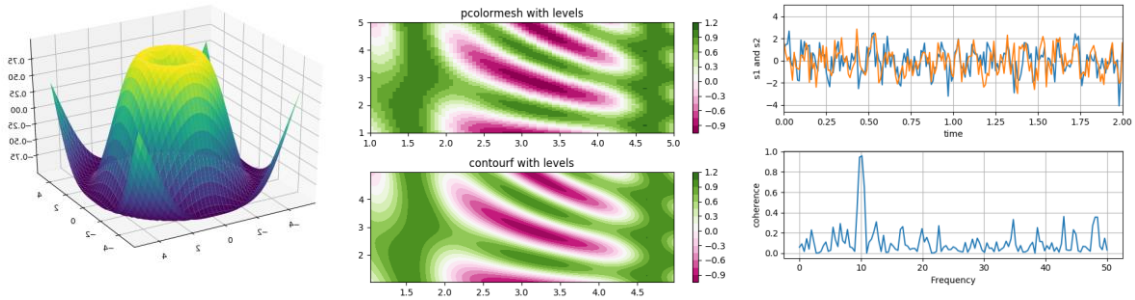


Figura 52 Ejemplos de representaciones con MatPlotLib

- Pandas [31]: Librería que permite el tratamiento y posterior análisis de estructuras de datos llamadas DataFrames (similares a matrices). Esta librería es usada en un gran número de aplicaciones: neurociencia, estadísticas, algoritmos de Machine Learning, etc.
- Tkinter [32]: Paquete que hace de interfaz entre Python y la herramienta Tk, la cual es una herramienta gratuita y multiplataforma que proporciona elementos básicos necesarios para crear interfaces gráficas

de usuario (GUI). .

- PySerial [33]: Este módulo contiene todo lo requerido para el acceso al puerto serie. Ésta soporta distintos tamaños de bytes, bits de paridad y distintos tipos de control de flujo.

4.2.2 Fundamentos de la programación

En primer lugar, se intenta realizar la apertura del puerto serie a una velocidad de 2000000 bauds, con el fin de que las transmisiones afecten el menor tiempo posible en la ejecución de las interrupciones de Arduino. Tras esto, se comienza un hilo secundario (en la figura aparece como hilo Background), el cual se usará únicamente para gestionar la recepción de datos desde el Arduino. Cada vez que se reciba un paquete de datos, este hilo lo almacenará a un buffer del tamaño de los datos recibidos, de forma que sólo se almacena el último paquete de datos de monitorización recibido.

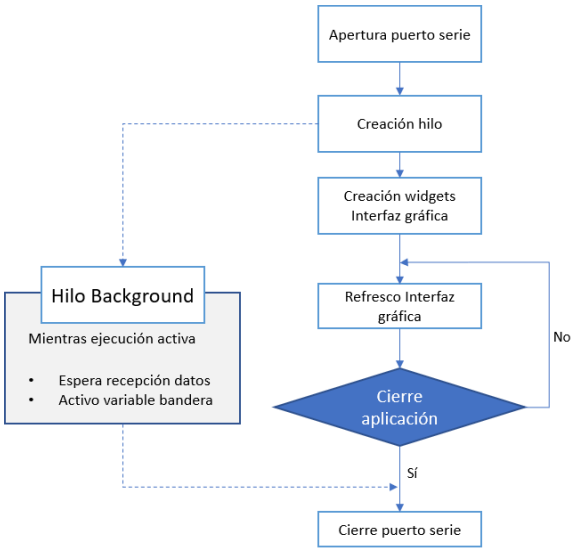


Figura 53 Diagrama simplificado del funcionamiento de la aplicación.

De forma paralela, una vez se ha intentado abrir el puerto serie, se crean los distintos widgets[5] que componen la interfaz gráfica, y se asocian a la ventana creada usando el paquete Tkinter. Cada elemento va asociado además a una función que aporta funcionalidades a la interfaz:

- En el caso de las gráficas que muestran los últimos datos recibidos, cada una de ellas lleva asociada una función que se encarga de generar los datos a representar en cada instante.
- En el caso de los pulsadores de las trayectorias, están asociada a una función que se ejecutará sólo cuando sean pulsados.

Tras la definición de estas asociaciones, la programación queda bloqueada en la ejecución de la actualización de la interfaz, hasta que se cierre la ventana creada. En el caso de que la ventana sea cerrada, se notificará al hilo que se la ejecución ha sido finalizada, y se esperará a su cierre mediante un join() en la ejecución principal del programa. Por último, una vez el hilo se encuentre cerrado, se cierra la conexión por el puerto serie, y se da por finalizada la ejecución.

4.2.3 Uso de clases para la gestión de la interfaz

Como se explica en [34], las clases en Python permiten agrupar un conjunto de datos y distintas funcionalidades de forma conjunta. En ellas se pueden agrupar distintos tipos de datos y funciones, las cuales pueden ser accedidas desde fuera de la propia clase (en función de si están definidas como públicas o privadas). Cuando se

[5] Elementos que componen una interfaz gráfica, y que permiten al usuario interactuar con el sistema operativo y la aplicación. Algunos widgets frecuentemente usados en las interfaces gráficas son botones, iconos, menús desplegables, etc.

define una nueva clase, también es necesario definir el constructor de la clase, que define la forma en la que se crean los objetos de datos de esa clase y suelen incluir la definición de los datos que serán usados posteriormente.

Las clases proporcionan a Python las características propias de la programación orientada a objetos. La creación de una clase implica a su vez la creación de un nuevo tipo de objeto, permitiendo a su vez la creación de nuevas instancias de ese tipo de objeto.

Para el desarrollo de la programación de esta aplicación se ha realizado una programación basada en clases. Estas clases se describen con brevedad con el fin de dar una idea general al lector de la programación, que también puede ser consultada en Anexo B: Programación en Python.

4.2.3.1 Clase SerialPlot

Esta clase se encarga de almacenar todos los datos relacionados con la conexión puerto serie abierta con el Arduino para la recepción de datos. En ella, se definen distintas funciones:

- ConnectSerialPort: Se encarga de gestionar la apertura del puerto serie. En caso de éxito, llama a su vez a la función readSerialStart(), explicada más adelante.
- CloseSerialPort: Se encarga de gestionar el cierre del puerto serie.
- readSerialStart: Se encarga de la apertura del hilo que se ejecutará en segundo plano, el cual se encargará de gestionar la recepción de los datos por el puerto serie.
- getSerialData: Función que se vincula a la representación gráfica de la posición de la pelota frente a la referencia. Para ello, extrae los datos del buffer de las comunicaciones, lo convierte al tipo de dato configurado (en este caso float de 4 bytes) y lo almacena en listas[6] definidas en la clase. Tras esto, prepara los datos para el refresco de las gráficas.
- getPlotXData/ getPlotYData. Al ser representados los mismos datos en la gráfica central y en las gráficas laterales, esta función sólo accede a los datos almacenados en listas y prepara los datos para la gráfica correspondiente.
- backgroundThread. Función que se encarga de realizar las lecturas de los datos recibidos por comunicaciones, y los almacena en un vector de datos (serán convertidos más adelante con getSerialData).
- Close: Primero ordena el cierre del hilo usando una variable compartida entre ambos, y realiza la espera del cierre del hilo hijo. Una vez cerrado, realiza el cierre del puerto serie.

4.2.3.2 Clase MainApp

Esta clase se encarga de la creación del layout usado en la aplicación, es decir, de la disposición de los pulsadores y las gráficas usadas en la visualización, así como la configuración del estilo de la gráfica, de la leyenda o de las líneas de cuadrícula. Dentro de esta clase, se hace uso a su vez de la clase window, la cual asocia las gráficas a la visualización de la pantalla.

Las funciones que se encuentran en esta clase son aquellas que se encuentran vinculadas a la pulsación de los pulsadores:

- TrajCircle, TrajCenter, TrajLemniscate: Se encargan de enviar el comando correspondiente al botón pulsado. Esto consiste en el envío de una cadena concreta de caracteres, siendo ‘TRA0’, ‘TRA1’ o ‘TRA2’ las cadenas usadas para cambiar a la trayectoria de círculo, al centro o la lemniscata respectivamente.

[6] Las listas en Python son un tipo de dato muy usado. Permite almacenar una secuencia de datos diversos, en la que se pueden incluir de forma simultánea valores numéricos, objetos, etc.

4.2.3.3 Clase window

Esta clase se encarga de asociar las distintas ventanas de representaciones gráficas con la ventana principal de la aplicación, así como la ubicación de cada uno. Como entradas básicas al constructor de la clase se tienen las figuras que se deseen usar y la ubicación de cada figura. Con el fin de realizar esto, incluye dos funciones que son usadas para la gestión de las tres figuras de representación:

- initWindow: Inicializa la figura principal, que representa la posición en tiempo real de la pelota frente a su referencia.
- initWindowSmall: Inicializa las dos figuras ubicadas en la izquierda de la pantalla.

4.2.4 Comunicación por puerto serie entre Arduino y Python

En este apartado se explica cómo se ha realizado la interacción entre el Arduino y la interfaz de usuario gráfica creada en Python a través del puerto serie.

Las comunicaciones entre ambos dispositivos se han planteado de forma distinta en función de quién es el emisor y el receptor.

4.2.4.1 Comunicación Arduino → Python

El objetivo de la interfaz web es la supervisión de la posición y la referencia del sistema Ball And Plate. Este envío periódico se realiza cada vez que se produce lo que se ha llamado como un 'Overflow de control' del controlador de la posición de la pelota (redondeados en Figura 54).

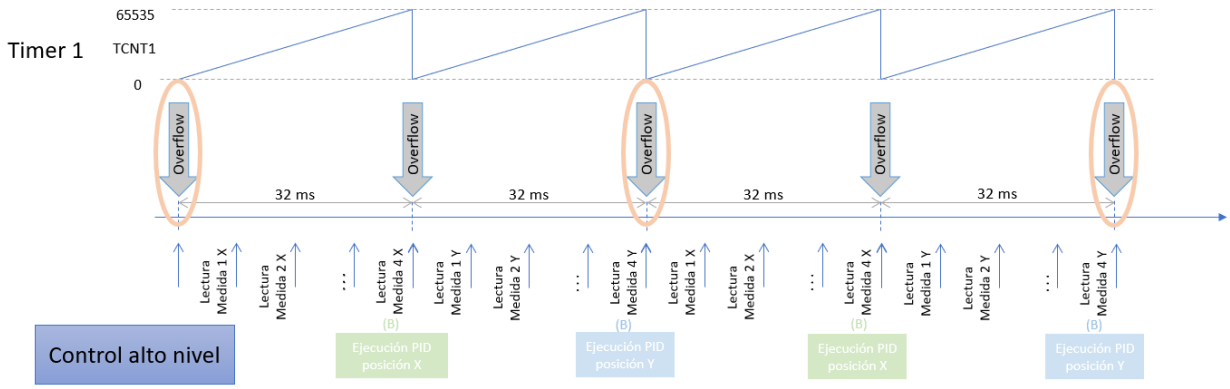


Figura 54 Ubicación del envío de datos de monitorización por puerto serie.

Como el rango de estos datos es muy amplio, así como luego es necesaria una posterior transformación, se decidió realizar este envío de datos de monitorización en bytes, de forma que el envío de cualquier variable de tipo float (definida en el Arduino Uno de tamaño 4 bytes) puede realizarse con sólo el envío de 4 bytes. Esto permite conseguir un envío mucho más óptimo en términos computacionales.

Este envío de datos es realizado con unas sencillas funciones que se encontraban definidas en los scripts que se usaron como base para esta aplicación [27]:

```
  /************************************/
  /*       Datalogger Function        */
  /************************************/


void sendToPC_uint( volatile uint16_t* data)
```

```
{
  byte* byteData = (byte*)(data);     // Casting to a byte pointer
  Serial.write(byteData, 2);          // Send through Serial to the PC
}

void sendToPC_int( int* data)
{
  byte* byteData = (byte*)(data);     // Casting to a byte pointer
  Serial.write(byteData, 2);          // Send through Serial to the PC
}

void sendToPC_float(float* data)
{
  byte* byteData = (byte*)(data);     // Casting to a byte pointer
  Serial.write(byteData, 4);          // Send through Serial to the PC
}
```

Para el caso de la recepción de los datos en Python desde el hilo *Background* (encargado de la gestión de las comunicaciones), los datos recibidos son almacenados en la estructura rawData, la cual puede ser luego accedida por el padre.

```
def backgroundThread(self):    # retrieve data
    time.sleep(1.0)  # give some buffer time for retrieving data

    self.serialConnection.reset_input_buffer()
    while (self.isRun):
        self.serialConnection.readinto(self.rawData)
        self.isReceiving = True
```

4.2.4.2 Comunicación Python → Arduino

Como se ha comentado en anteriores apartados, la aplicación también permite la modificación de la trayectoria seguida por la pelota sobre la superficie. Debido a que esta comunicación va a ser muy poco frecuente, se ha decidido plantear mediante el envío de comandos en forma de cadenas de caracteres.

El envío de caracteres se hace usando una función ya definida en la librería pySerial, la cual permite el envío de caracteres de forma muy sencilla. Los comandos implementados en la aplicación son sólo 3, aunque en la programación de Arduino se han añadido más con el objetivo de poder realizar depuración (entre ellos, se encuentra la posibilidad de enviar las referencias de ángulos de la superficie al controlador de bajo nivel).

- TRA0: Control de posición en la posición X=0, Y=0.
- TRA1: Trayectoria circular.
- TRA2: Trayectoria en forma de Lemniscata.

```
def TrajCircle(self): # Funcion del boton
    self.serial.serialConnection.write(b"TRA1");
    time.sleep(2)

def TrajCenter(self): # Funcion del boton
    self.serial.serialConnection.write(b"TRA0");
    time.sleep(2)

def TrajLemniscate(self): # Funcion del boton
    self.serial.serialConnection.write(b"TRA2");
    time.sleep(2)
```

En el caso del Arduino, cuando llega algún dato al buffer del puerto serie, se realiza la lectura de los bytes, los cuales se vuelcan al vector de caracteres declarado como `buffer` y se obtiene en una variable de tipo entero *index* la longitud de los datos leídos.

Tras esto, las tres primeras letras del comando son usadas para identificar el comando, mientras que el resto de datos son información requerida por el comando concreto; en este caso, equivale a la identificación de la trayectoria.

```
/**************************/
 /*  Serial Port Reception */
 /**************************/
 while (Serial.available())
 {
   //newline or max of 9 chars
   int index = Serial.readBytesUntil('\n', buffer, 9);
   buffer[index] = '\0';

   if(APP_MODE!=2) Serial.println(buffer);

   /* The string is clasified depending on the command */

   if(buffer[0] == 'T' && buffer[1] == 'R'&& buffer[2] == 'A')
   {
     /* First 3 characters are deleted */
     for (int i=3; i!= index ; i++)  buffer[i-3]= buffer[i];
     buffer[index-3]= '\0';
     TrajGUI = atoi(buffer);  //convert readString into a number
   }
```

ANEXO A: PROGRAMACIÓN EN ARDUINO

En este anexo se presenta la programación realizada en el Arduino. La programación en el IDE de Arduino se ha llevado a cabo mediante el uso de Tabs, las cuales se presentan a continuación.

Es necesario tener en cuenta que para el correcto funcionamiento de la apliación, también es necesario realizar la modificación explicada en el apartado 3.5.2 Aumento de velocidad de lectura, en la que se modificaba la librería correspondiente de la IMU MPU-6050.

Tab 1: Define

```
/*****************************************/
/********* Ball And Plate Control *********/
/*********     #define section     *********/
/*****************************************/

/* Code written by Ramon Iglesias, 2019 */

/*********************************************/
/* External Libraries for IMU MPU6050 device  */
/*********************************************/
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#include "Wire.h"

/*********************************************/
/*         Arduino Pinout Definition          */
/*********************************************/

/* External Interruptions  */
#define MPUInterrupt_PIN      2 /* read data from MPU-6050 */
#define ButtonInterrupt_PIN   3 /* User button - Trajectory */

/* EasyDriver Stepper Motor 1  */
#define StepM1_PIN            12
#define DirM1_PIN             13
#define LimitSwM1_PIN         11

/* EasyDriver Stepper Motor 2  */
#define StepM2_PIN            7
#define DirM2_PIN             8
#define LimitSwM2_PIN         6

/* 4-Wire TouchScreen  */
#define X1                    A1
#define X2                    A3
#define Y1                    A0
#define Y2                    A2

/*********************************************/
/* Data Output (Serial Port)  Mode selection  */
/*********************************************/

/* Data output can be chosen depending on the value:
 *  0. Debug Mode
```

```
 *  1. Arduino Plotter: Real time position is sent in
 *     characters (Serial Plotter).
 *  2. Python Plotter: References and real time position
 *     is sent in bytes to the Python App.
 */
#define APP_MODE              0

/* Debug Mode Options */

/* If MEAS_MPU =1, it shows data taken from MPU and time
 *  between received measures */
#define MEAS_MPU              0

/* If MEAS_TOUCHSCREEN = 1, it shows data taken from
 *  TouchScreen (from -450 to 450 aprox.) */
#define MEAS_TOUCHSCREEN      0

/* If DEBUG_TOUCHSCREEN = 1, it shows Timer1 counter
 *  in each interruption performed, and the output of
 *  median filter.
 */
#define DEBUG_TOUCHSCREEN     0



/* If COM_FILTER_TOUCHSCREEN = 1, it shows Timer1 counter
 *  in each interruption performed, and the output of
 *  median filter.
 */
#define COM_FILTER_TOUCHSCREEN     0




/* If DEBUG = 1, it shows lower-level controller's inputs
 *  and outputs (angles measured by MPU6050 and references
 *  in angules to be sent to stepper motors.
 */
#define DEBUG                 0

/* If DEBUG_PULSES = 1, it shows debug information
 *  of stepper motor pulses: Timer2 count between
 *  pulses (calculated each 2 Timer2 overflows = 255 x 2
 *  counts). Note: In Motor Class, "pulse cycle"
 *  - Letter A represents an interrupt in Motor 1
 *   to change STEP state
 *  - Letter B represents an interrupt in Motor 1
 *   to change STEP state
 */
#define DEBUG_PULSES          0

/*******************************************************/
/*High-level Controller: Ball position with Touchscreen */
/*******************************************************/

/*    Interrupt&Control configuration     */
/*****************************************/

/* NOTE: A measurement is defined as obtaining X
```

```
 *  and Y ball position  */

/* NUMSAMPLES. Number of complete measurements to be
 *  taken in a measurement interval to execute the
 *  median filter. This value has to be at least equal
 *  or greater to NUM_OV1_PER_CYCLE/2
 */
#define NUMSAMPLES              4

/* NUM_OV1_PER_CYCLE: Defines the length of the
 *  measurement period in Timer 1 Overflows
 */
#define NUM_OV1_PER_CYCLE      2

/* TRAJECTORY: Defines the trajectory by default
 *  0. X=0, Y=0
 *  1. Circle
 *  2. Lemniscate
 */
#define TRAJECTORY             2

/* INCR_TIME: Defines the speed of the trajectory
 */
#define INCR_TIME              0.02

/* UPPER_CONTROL_DISABLED: Disables hihg-level
 *  controller.Angles references of the Plate
 *  can be set by sending them through Serial Port.
 *  Disabled = 1, Enabled = 0
 */
#define UPPER_CONTROL_DISABLED 0
```

Tab 2: StepperMotorClass

```
/*****************************************/
/*********  Stepper Motor Class  **********/
/*****************************************/

/* Code written by Ramon Iglesias, 2019 */



class Motor{

  /************************/
  /* Variable declaration */
  /************************/

  private:

  /* Interval variables involved in control */

    /* Position reference in pulses */
    int _RefPulsePosition;
    /* Value used to generate the pulse train */
    bool _val;
    /* Timer Count value to be assigned to the
     *  Compare Match Register (OCR) */
```

```
  int _NextCountValue;
  /* Rotation Direction.
   *  HIGH: Forward (step increasing)
   *  LOW: Backards (step decreasing) */
  bool _Direction;

/* Arduino-EasyDriver Pinout */

  int _StepPin; /* Pin connected to Step */
  int _DirectionPin; /* Defines initial position */
  int _LimitSwitchPin; /* Defines initial position */

public:

  /* Current Position in pulses - referred to
   *  limit switch */
  int _PulsePosition;
  /* Timer count between pulses calculated in the
   *  current control interval. This is the value
   *  that OCRXX has to be incremented in each
   *  interrupt */
  int _PulseCycle;

/* Interval parameters involved in control */

  /* CountMotorInterval: Number of timer counts between
   *  overflows where the stepper motor has to carry out
   *  the movement */
  unsigned long _CountMotorInterval= 255*2;
  /* Pulse length in Timer Counts */
  int _PulseWidth;
  /* Timer Counts between pulses in Train pulse */
  int _BetweenPulses;
  /* Stepper Motor Parameters */
  float _StepAngle; /* Degrees per step */
  float _MicroStep; /* Microsteps per step */

/*********************/
/* Class Constructor */
/*********************/

Motor()
  {
    /* initial values - Parameters */
    _StepAngle= 1.8; /* Degrees */
    _MicroStep= 8; /* Microsteps per step */
    _PulseWidth= 8; /* 256 microseconds */
    _BetweenPulses= 8; /* 256 microseconds */
  }


/*********************/
/*   setPinConfig    */
/*********************/

/* It assigns inputs to the corresponding Pin
 *  variable and configures pin mode */

void setPinConfig(int input_StepPin, int input_DirectionPin,
```

```
                    int input_LimitSwitchPin)
  {
    _StepPin= input_StepPin;
    _DirectionPin= input_DirectionPin;
    _LimitSwitchPin= input_LimitSwitchPin;

    pinMode(_StepPin, OUTPUT);
    pinMode(_DirectionPin, OUTPUT);
    pinMode(_LimitSwitchPin, INPUT_PULLUP);
    digitalWrite(_StepPin, LOW);
    digitalWrite(_DirectionPin, LOW);
  }

/*****************************/
/*    ResetPulsePosition     */
/*****************************/

/* Set current position as initial position  */

void ResetPulsePosition()
  {
    _PulsePosition= 0;
    _RefPulsePosition= 0;
  }

/*****************************/
/*     SetAnglePosition      */
/*****************************/

/* It sets position reference in angles. It returns
 *  the value to be assigned in the corresponding
 *  Compare Match Register (OCRXX) */

 unsigned int SetAnglePosition(float reference)
  {
    float RefAnglePosition= reference;
    /* The reference is converted into pulses */
    int PulseReference= (RefAnglePosition/_StepAngle) * _MicroStep;
    /* PulseReference is set as PulseReference */
    int res= SetPulsePosition(PulseReference);
    return res;
  }

 /*****************************/
 /*     SetPulsePosition      */
 /*****************************/

 /* It sets position reference in pulses. It returns
 *  the value to be assigned in the corresponding
 *  Compare Match Register (OCRXX) */

 unsigned int SetPulsePosition(int reference)
  {
    /* Reference in pulses is set */
    _RefPulsePosition= reference;
    /* Number of pulses to generate to get the desired
     *  position is calculated */
    int PulseIncrement = (_RefPulsePosition - _PulsePosition);
    PulseIncrement = abs(PulseIncrement);
```

```
    /* Avoid dividing by 0 */
    if (PulseIncrement == 0)  PulseIncrement= 1;
    /* OCRXX Increment per interruption is calculated */
    int res = _CountMotorInterval/PulseIncrement;
    /* Number is rounded */
    int remain = _CountMotorInterval%PulseIncrement;
    if (remain > (PulseIncrement/2))  res= res+1;
    /* Time restrictions are applied */
    if (res < (_BetweenPulses + _PulseWidth))
    {
      res= (_BetweenPulses + _PulseWidth);
    }
    /* Time needed to change the state from True to False
     *  is also taken into consideration */
    res = res - _PulseWidth;
    /* Pulse Cycle is assigned */
    _PulseCycle = res;
    /* It is also returned, in order to be assigned in OCRXX */
    return res;
 }

/***************************/
/*      PulseGenerator     */
/***************************/

/* Function in charge of manage the state of the STEP
 *  and DIR pins every time the interruption occurs */

int PulseGenerator_Vcte(int OCRXX)
 {
    /* If desired position has not been reached or val is HIGH*/
    if (_PulsePosition!=_RefPulsePosition || _val!=LOW)
     {
       /* Direction is set */
       if (_PulsePosition > _RefPulsePosition) _Direction= LOW;
       else _Direction=HIGH;
       /* Pulse generation and position update  */
       if (_val==LOW)
        {
          OCRXX = OCRXX + _PulseWidth;
          _val=HIGH;
          if (_Direction==HIGH)  _PulsePosition++;
          else _PulsePosition--;
        }
       else /* Val is HIGH */
        {
          _val=LOW;
          if (_PulsePosition!=_RefPulsePosition)
            {
              /* OCRXX is increased */
              OCRXX = OCRXX + _PulseCycle;
            }
          else /* Position Reached */
            {
               OCRXX = OCRXX + _CountMotorInterval;
            }
         }
        digitalWrite(_DirectionPin, _Direction);
        digitalWrite(_StepPin, _val);
```

```
          /* Only for debug purposes
          Serial.print(" Val : ");
          Serial.print(" ");
          Serial.print(val);
          Serial.print(" Ref : ");
          Serial.print(" ");
          Serial.print(RefPulsePosition);
          Serial.print("  Current Pos : ");
          Serial.print(" ");
          Serial.print(PulsePosition);
          Serial.print(" ");*/
       }
      return(OCRXX);
    }

   /****************************/
   /*      InitPosition()      */
   /****************************/

   /* It initialize the stepper motor. It will move
    *  backwards until the limit switch is reached.
    *  Then it is moved to PosInit (input in pulses)
    *  and then this position is set as initial
    *  position */

   void InitPosition(float PosInit)
    {
      bool LimitSwitchState= digitalRead(_LimitSwitchPin);
      while(LimitSwitchState == 0)
         {
             SetAnglePosition(-180);
             LimitSwitchState= digitalRead(_LimitSwitchPin);
         }
      /* Reset the position to 0 */
      ResetPulsePosition();
      /* Reset the reference to PosInit */
      SetAnglePosition(PosInit);
      /* Wait until position is reached */
      while(_RefPulsePosition!=_PulsePosition) delay(1000);
      delay(1000);
      /* Current position is set as initial position */
      ResetPulsePosition();
    }
}; /* End Motor Class Definition */
```

Tab 3: TouchScreenClass

```
/*****************************************/
/**********  Touchscreen Class  **********/
/*****************************************/

/* Code written by Ramon Iglesias, 2019 */


class TouchScreen{

  /***********************/
  /* Variable declaration */
```

```
/************************/

private:

/* Interval variables involved in interruptions */

  /* Timer Count value to be assigned to the
   *  Output Compare register (OCR) */
  int _NextCountValue;
  /* Timer count between pulses calculated in the
   *  current control interval. This is the value
   *  that OCRXX has to be incremented in each
   *  interrupt */
  int _PulseCycle;
  /* _CurrentMeas: Number of the current measurement */
  int  _CurrentMeas;
  /* Pointer to 16 bit Output Compare Register used */
  volatile uint16_t* _OCRXX; /* Only for Timer 1 */

/* Interval variables involved in measurements */

  /* Arrays where measurement data is stored to perform
   *  the median filter */
  float  _X_Samples_Array[10];
  float  _Y_Samples_Array[10];
  /* Variables used in Low Pass filter */
  int _Y_sample_ant;
  int _X_sample_ant;

/* TouchScreen-Arduino Pinout */
  int _X1;
  int _X2;
  int _Y1;
  int _Y2;

/* Interval parameters involved in control */

  /* Number of complete measures to be taken in the
   *  measurement interval, in order to carry out
   *  the median filter. A complete measure includes
   *  X and Y position */
  int _NumberMeas = 6;
  /* _MeasInterval: Length of the measument interval.
   *  Defined in Number of timer counts  */
  unsigned long _MeasInterval = 65535 * 2;
  /* _MsInterval: 1ms in number of timer counts
   *  of the Timer1 used with prescaler 1024 */
  int _MsInterval = 16;
  /* TouchSreen Offsets */
  const int _X_0= 518;
  const int _Y_0= 508;
  /* Scale factor in X and Y axis */
  float _ScaleFactor_Y = 14300/441.8;
  float _ScaleFactor_X = 30000/810.0;

public:

  /* Low Pass Filter enable Flag */
  bool _EnaLowPassFilter = HIGH;
```

```
    /* Alpha values used in Low Pass Filter */
    float _alphaX = 0.80;
    float _alphaY = 0.80;
    /* X and Y position to be used in control */
    float _Y_sample;
    float _X_sample;
    /* No Overflows in a measurement interval*/
    int _N_Overflow = 2;

    /* Enable Flags for debug and test purposes */
    bool _debug=LOW;
    bool _debugMeasures=LOW;
    bool _ShowMeasure=LOW;
    /* Filter comparation Plots */
    bool _CompareFilterMedian=LOW;
    bool _CompareFilterLowPass=LOW;

  /*********************/
  /* Class Constructor */
  /*********************/

 /* It assigns inputs to the corresponding Pin
  *  variables, and stores OCR Pointer */

  TouchScreen(const uint8_t input_X1, const uint8_t input_X2,
              const uint8_t input_Y1, const uint8_t input_Y2,
              volatile uint16_t* OCRXX)
    {
      _X1= input_X1;
      _X2= input_X2;
      _Y1= input_Y1;
      _Y2= input_Y2;
      _OCRXX= OCRXX;
    }

  /*********************/
  /*     setConfig     */
  /*********************/

   /* It set internal parameters related to interruptions and
    *  touchscreen resolution */

    void setConfig(int NumberMeas, int N_Overflow,
                   unsigned long MeasInterval,
                   float ScaleFactor_X, float ScaleFactor_Y)
    {
      /* Interruption configuration */
      if (NumberMeas < (N_Overflow/2))
      {
        NumberMeas = (N_Overflow/2) + 1;
      }
      _NumberMeas= NumberMeas;
      _N_Overflow= N_Overflow;
      _MeasInterval= (MeasInterval)*N_Overflow;
      /* TouchScreen Resolution */
      _ScaleFactor_X = ScaleFactor_X;
      _ScaleFactor_Y = ScaleFactor_Y;

      if(_debug)
```

```
    {
      Serial.println(" SetConfig() inputs ");
      Serial.print(" >>>>>NumberMeas ");
      Serial.println(NumberMeas);
      Serial.print(" >>>>>N_Overflow ");
      Serial.println(N_Overflow);
      Serial.print(" >>>>>MeasInterval ");
      Serial.println(MeasInterval);

      Serial.println(" SetConfig() after calculation ");
      Serial.print(" >>>>>NumberMeas ");
      Serial.println(_NumberMeas);
      Serial.print(" >>>>>N_Overflow ");
      Serial.println(_N_Overflow);
      Serial.print(" >>>>>MeasInterval ");
      Serial.println(_MeasInterval);
    }

  }

/*******************************/
/*     EnableLowPassFilter     */
/*******************************/

 /* It activates and set configuration of Low
  *  Pass Filter */

  void EnableLowPassFilter(bool enable, float alphaX, float alphaY)
  {
    _EnaLowPassFilter= enable;
    _alphaX= alphaX;
    _alphaX= alphaY;
  }

/************************/
/*     insert_sort      */
/************************/

 /* It sorts a vector from highest value to
  *  lowest value  */

  void insert_sort(float array[], uint8_t size)
  {
    uint8_t j;
    int save;

    for (int i = 1; i < size; i++)
    {
      save = array[i];
      for (j = i; j >= 1 && save < array[j - 1]; j--)
        array[j] = array[j - 1];
      array[j] = save;
    }
  }

/************************/
/*     SetNumberMeas    */
/************************/
 /* It has to be executed in the control overflow.
```

```
    *  It calculates the count increment to be added
    *  in Output Compare register in order to carry out
    *  the different interruptions.
    */
    void SetNumberMeas()
    {
      /* Current meas is set to 0 - this function is executed
       *  at the beginning of hte measurement interval */
      _CurrentMeas=0;

      if(_debug) Serial.print("\n StartMeas");

      /* Avoid dividing by 0 */
      if (_NumberMeas == 0)  _NumberMeas= 1;

      /* OCRXX Increment per interruption is calculated */
      int res = _MeasInterval/(2*_NumberMeas);
      //La medida con la division es truncada por defecto
      if(_debug)
      {
        Serial.print("_res_");
        Serial.print(res);
        Serial.print(" ");
      }
      /* Time restriction in order not to lose steps */
      if (res< 2 * _MsInterval) res= 2 * _MsInterval;

      /* Result is stored in PulseCycle */
      _PulseCycle= res;
      /* The first interruption is set in OCR register*/
      *_OCRXX= res;
    }

  /*************************/
  /*     PulseGenerator    */
  /*************************/
   /* It manages the OCR register in order to perform
    *  the measurement interruptions. It also carries out
    *  measurements, changes pin modes and indicates to
    *  high-level controller when control in X or Y
    *  axis has to be carried out. The ControltoBeExecuted
    *  is the output of the function:
    *  - 0. No action
    *  - 1. Data needed to control X axis is ready
    *  - 2. Data needed to control Y axis is ready
    */

    int MeasInterruptGenerator()
    {
       int ControltoBeExecuted=0;
       _CurrentMeas++;
       if(_debug)
       {
        Serial.print(" CompA_");
        Serial.print(TCNT1);
        Serial.print("_");
        Serial.print("M");
        Serial.print(_CurrentMeas);
       }
```

```
      /* If there are X or Y measurements to be measured */
      if (_CurrentMeas<=(2*_NumberMeas))
      {
         /* Next interruption is set in OCR register*/
         *_OCRXX = *_OCRXX + _PulseCycle;

         /* If there are X measurements to be measured */
         if (_CurrentMeas <= (_NumberMeas) )
         {
           if(_debug) Serial.print("_X");

           /* Previous measurement is read */
           float aux_X = - analogRead(Y1) + _X_0;
           _X_Samples_Array[_CurrentMeas-1] =  aux_X * _ScaleFactor_X;


           if(_debugMeasures)
           {
             Serial.print(" X=");
             Serial.print(_X_Samples_Array[_CurrentMeas-1]);
           }

           /* If measurement in X axis is finished, X axis
            *  Control can be executed */
           if (_CurrentMeas == (_NumberMeas) )
           {
             if(_debug) Serial.print("_ControlX");

             /* Output assignment */
             ControltoBeExecuted=1;

           /* Get ready to read Y in the next interrupt */
             /* Set X1 as input */
             pinMode(X1, INPUT);
             /* Set X2 as tristate */
             pinMode(X2, INPUT);
             digitalWrite(X2, LOW);
             /* voltage divider in Y1(+5V) and Y2(GND) */
             pinMode(Y1, OUTPUT);
             digitalWrite(Y1, HIGH);
             pinMode(Y2, OUTPUT);
             digitalWrite(Y2, LOW);

             /* Median filter is performed */
             insert_sort(_X_Samples_Array, _NumberMeas);
             _X_sample= _X_Samples_Array[(_NumberMeas-1)/2];

              /* Low Pass filter is performed */
             if (_EnaLowPassFilter==HIGH)
             {
               if(_debug) Serial.print("_LP_Filter");
               _X_sample= _X_sample * _alphaX + (1-_alphaX) *
_X_sample_ant;
             }

             /* Previous values are stored for the next execution */
             _X_sample_ant= _X_sample;
```

```
              if(_debug)
              {
                Serial.print("_X=");
                Serial.print(_X_sample);
              }
              else if(_ShowMeasure)
              {
                Serial.print(_X_sample);
                Serial.print("\t");
              }

              if(_debugMeasures)
              {
                Serial.print("\n");
                Serial.print(_X_Samples_Array[0]);
                Serial.print("\t");
                Serial.print(_X_Samples_Array[1]);
                Serial.print("\t");
                Serial.print(_X_Samples_Array[2]);
                Serial.print("\t");
                Serial.print(_X_Samples_Array[3]);
                Serial.print("\t");
                Serial.print(_X_Samples_Array[4]);
                Serial.print("\t");
                Serial.print(_X_Samples_Array[5]);
                Serial.print("\t");
                Serial.print(_X_Samples_Array[6]);
                Serial.print("\n");
              }
            }
          }
          /* If there are Y measurements to be measured */
          else if (_CurrentMeas > (_NumberMeas) )
          {
            if(_debug) Serial.print("_Y");

            /* Previous measurement is read. Y measurement is
             *  scaled with the TouchScreen resolution */
            float aux_Y = - analogRead(X1) + _Y_0;
            _Y_Samples_Array[_CurrentMeas-_NumberMeas-1] =  aux_Y *
_ScaleFactor_Y;

            /* If measurement in Y axis is finished, Y axis
             *  Control can be executed */
             if (_CurrentMeas == (2*_NumberMeas) )
            {
              if(_debug) Serial.print("_ControlY");

              /* Output Assignment */
              ControltoBeExecuted=2;

              /* Get ready to read X in the next interrupt */
              /* Set Y1 as input */
              pinMode(Y1, INPUT);
              /* Set Y2 as tristate */
              pinMode(Y2, INPUT);
              digitalWrite(Y2, LOW);
              /* voltage divider in X1(+5V) and X2(GND) */
              pinMode(X1, OUTPUT);
```

```
                digitalWrite(X1, HIGH);
                pinMode(X2, OUTPUT);
                digitalWrite(X2, LOW);

                /* Median filter is performed */
                insert_sort(_Y_Samples_Array, _NumberMeas);
                _Y_sample= _Y_Samples_Array[(_NumberMeas-1)/2];

                /* Filters vs rawData */
                 if (_CompareFilterMedian)
                 {
                    Serial.print(_Y_Samples_Array[_NumberMeas-1]);
                    Serial.print("\t");
                    Serial.println(_Y_sample);
                 }

                 if (_CompareFilterLowPass)
                 {
                    Serial.print(_Y_sample);
                    Serial.print("\t");
                 }

                /* Median filter is performed */
                 if (_EnaLowPassFilter==HIGH)
                 {
                  if(_debug) Serial.print("_Filter");
                  _Y_sample= _Y_sample * _alphaY + (1-_alphaY) *
_Y_sample_ant;
                 }

                 if (_CompareFilterLowPass) Serial.println(_Y_sample);

                 /* Previous values are stored for the next execution */
                 _Y_sample_ant= _Y_sample;

                if(_debug)
                {
                  Serial.print("_Y=");
                  Serial.print(_Y_sample);
                }
                else if(_ShowMeasure)
                {
                  Serial.println(_Y_sample);
                }

                if(_debugMeasures)
                {
                  Serial.print("\n");
                  Serial.print(_Y_Samples_Array[0]);
                  Serial.print("\t");
                  Serial.print(_Y_Samples_Array[1]);
                  Serial.print("\t");
                  Serial.print(_Y_Samples_Array[2]);
                  Serial.print("\t");
                  Serial.print(_Y_Samples_Array[3]);
                  Serial.print("\t");
                  Serial.print(_Y_Samples_Array[4]);
                  Serial.print("\t");
                  Serial.print(_Y_Samples_Array[5]);
```

```
                Serial.print("\t");
                Serial.print(_Y_Samples_Array[6]);
                Serial.print("\n");
              }
            }
          }
        }
        else /* If all measurements have been carried out */
        {
         if(_debug) Serial.println("_End");
         /* It assigns a complete MeasInterval, in order
          *  to wait until next overflow.
         */
         *_OCRXX = *_OCRXX + _MeasInterval;
        }
    if(_debug)
    {

      Serial.print("_");
      Serial.print(ControltoBeExecuted);
    }
    return ControltoBeExecuted;
    }
  };/* End TouchScreen Class Definition */
```

Tab 4: _Main

```
/*****************************************/
/********* Ball And Plate Control *********/
/*********      main section      *********/
/*****************************************/

  /************************/
  /*       Instances      */
  /************************/

/* Motor Class */
Motor M1; //Creacion objeto M1 X
Motor M2; //Creacion objeto M2 Y

/* TouchScreen Class */
TouchScreen Ts= TouchScreen(X1, X2, Y1, Y2, &OCR1A);

/* MPU-5060 Class */
const int mpuAddress = 0x68;  // It can be 0x68 or 0x69
MPU6050 mpu(mpuAddress);

  /************************/
  /*   Control variables  */
  /************************/

//Incrementos para trayectorias circulares, etc
float incr_ref;
int CountInterval=0;

/*   HIGH LEVEL PID Controller - Ball Position */
/*********************************************/
```

```
/* X Axis */
float K_pX=0.044;
float K_iX=0;
float K_dX=0.55;

float RefX=0.0;
float errSumX, lastErrX;

/* Y Axis */
float K_pY=0.044;
float K_iY=0;
float K_dY=0.55;

float RefY=0.0;
float errSumY, lastErrY;

/*   LOW LEVEL PID Controller - Plate Angle */
/*****************************************/

/* X Axis */
float RefThetaX=0.0;
float OutputMotorX=0;
float errSumThetaX, lastErrThetaX;
float K_pThetaX=0.6;//0.5;
float K_iThetaX=0;
float K_dThetaX=0.2;//0;

/* Y Axis */
float RefThetaY=0;
float OutputMotorY=0.0;
float errSumThetaY, lastErrThetaY;
float K_pThetaY=0.6;//0.5;
float K_iThetaY=0;
float K_dThetaY=0.2;//0;


  /********************/
  /*   MPU variables  */
  /********************/

/* MPU control/status vars */

bool dmpReady = false;  /*  set true if DMP init was successful */
uint8_t mpuIntStatus;   /*  holds actual interrupt status byte from MPU */
uint8_t devStatus;      /*  return status after each device
                         *  operation (0 = success, !0 = error) */
uint16_t packetSize;    /*  expected DMP packet size (default is 42 bytes)
*/
uint16_t fifoCount;     /*  count of all bytes currently in FIFO */
uint8_t fifoBuffer[64]; /*  FIFO storage buffer */

/* Vars used in debug mode */
float TimeMeas_0;
float TimeMeas_1;

/* var to manage interrupts */
volatile bool mpuInterrupt = false;

/* MPU data management */
```

```
Quaternion q;           /*  [w, x, y, z] */
VectorInt16 aa;         /*  [x, y, z] */
VectorInt16 aaReal;     /*  [x, y, z] */
VectorInt16 aaWorld;    /*  [x, y, z] */
VectorFloat gravity;    /*  [x, y, z] */
float ypr[3];           /*  [yaw, pitch, roll] */

float ang_x, ang_y;

/* MPU Angle Offsets */
float offsetY= -3.75; //-1.59;
float offsetX= -2.7; //-3.0;//-2.94

  /**********************/
  /*  TouchScreen Data  */
  /**********************/

/* Low Pass Filter */
float alphaX=0.7;
float alphaY=0.7;

/* 4-Wire TouchScreen  */
float const ScaleFactor_Y = 14300/441.8;
float const ScaleFactor_X = 30000/810.0;

  /**************************/
  /*  Python App Interface  */
  /**************************/

/* Serial Port buffer */
char buffer[10];

/* Ball and Plate Trajectory */
int TrajGUI = TRAJECTORY;

  /************************/
  /*  Auxiliary functions */
  /************************/

/* TrajectoryButton */

void TrajectoryButton() {
    TrajGUI++;
    if (TrajGUI>2) TrajGUI=0;
}

/* TrajectoryButton */

void dmpDataReady() {
    mpuInterrupt = true;
}


  /*********************************/
  /*       Ball And Plate Setup      */
  /*********************************/

void setup()
{
```

```
  /* Serial Port init */
  Serial.begin(2000000);

  /*********************/
  /* Setup Touchscreen */

  Ts._debugMeasures=LOW;
  if(DEBUG_TOUCHSCREEN==1) Ts._debug=HIGH;
  else Ts._debug=LOW;
  if(MEAS_TOUCHSCREEN==1) Ts._ShowMeasure=HIGH;
  else Ts._ShowMeasure=LOW;

  /* Set configuration */
  Ts.setConfig(NUMSAMPLES, NUM_OV1_PER_CYCLE, 65525,
               ScaleFactor_X, ScaleFactor_Y );

  /*********************/
  /*   Setup MPU6050   */
  /*********************/

  #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
      Wire.begin();
      /* 400kHz I2C clock. Comment this line if having
       *  compilation difficulties */
      Wire.setClock(400000);
  #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
      Fastwire::setup(400, true);
  #endif

  mpu.initialize();
  if(APP_MODE!=2) Serial.println(mpu.testConnection() ? F("IMU iniciado
correctamente") : F("Error al iniciar IMU"));

  pinMode(MPUInterrupt_PIN, INPUT);
  /* Connection is checked */
  if(APP_MODE!=2) Serial.println(F("Testing device connections..."));
  if(APP_MODE!=2) Serial.println(mpu.testConnection() ? F("MPU6050
connection successful") : F("MPU6050 connection failed"));

  /* DMP is initialized */
  if(APP_MODE!=2) Serial.println(F("Initializing DMP..."));
  devStatus = mpu.dmpInitialize();

  /* Calibration Values */
  mpu.setXGyroOffset(158);
  mpu.setYGyroOffset(-107);
  mpu.setZGyroOffset(35);
  mpu.setXAccelOffset(-1660);
  mpu.setYAccelOffset(-1440);
  mpu.setZAccelOffset(1509);

  /* Activate DMP */
  if (devStatus == 0)
  {
    if(APP_MODE!=2) Serial.println(F("Enabling DMP..."));
    mpu.setDMPEnabled(true);

    /* An interrupt is attached to Digital pin */
    attachInterrupt(digitalPinToInterrupt(MPUInterrupt_PIN),
```

```
                    dmpDataReady, RISING);

    mpuIntStatus = mpu.getIntStatus();

    if(APP_MODE!=2) Serial.println(F("DMP ready! Waiting for first
interrupt..."));
    dmpReady = true;

        // get expected DMP packet size for later comparison
        packetSize = mpu.dmpGetFIFOPacketSize();
  }
  else
  {
    /* ERROR!
     * 1 = initial memory load failed
     * 2 = DMP configuration updates failed
     * (if it's going to break, usually the code will be 1)
     */
    if(APP_MODE!=2) Serial.print(F("DMP Initialization failed (code "));
    if(APP_MODE!=2) Serial.print(devStatus);
    if(APP_MODE!=2) Serial.println(F(")"));
}

  /*********************/
  /*   Button setup    */
  /*********************/

  pinMode(ButtonInterrupt_PIN, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(ButtonInterrupt_PIN),
TrajectoryButton, FALLING);

  /***************************/
  /*   Interruption Setups   */
  /***************************/
  noInterrupts(); // disable all interrupts

  /************/
  /*  Timer 1 */
  /************/

 /* Ball position controller in Overflow */

  TCCR1A = 0;
  TCCR1B = 0;

  OCR1A = 65535; /* compare match register */

  TCCR1B |= (1 << CS11); /* prescaler 8, overflows each 32 ms */
  TIMSK1 |= (1 << OCIE1A); /* enable timer compare interrupt A */

  /************/
  /*  Timer 2 */
  /************/

 /* Plate Angle controller in Overflow */

  TCCR2A = 0;
  TCCR2B = 0;
```

```
  TIMSK2 |= (1 << OCIE2A); /* enable timer compare interrupt A */
  TIMSK2 |= (1 << OCIE2B); /* enable timer compare interrupt B */

  OCR2A = 255; /* compare match register */
  OCR2B = 255; /* compare match register */

  TCCR2B |= (1 << CS22); /* prescaler 256, overflows each 4 ms */
  TCCR2B |= (1 << CS21); /* prescaler */
  //TCCR2B |= (1 << CS20); /* prescaler */

  interrupts(); // enable all interrupts

  /****************************/
  /*   Stepper Motor Setups   */
  /****************************/

  /* Motor initalization */
  M1.setPinConfig(StepM1_PIN, DirM1_PIN, LimitSwM1_PIN);
  M2.setPinConfig(StepM2_PIN, DirM2_PIN, LimitSwM2_PIN);
  M1.InitPosition(3.0);
  M2.InitPosition(3.0);

  noInterrupts(); /* Disable all interrupts */

  TIMSK2 |= (1 << TOIE2); /* Enable Overflow T2 Motor Control */
  TIMSK1 |= (1 << TOIE1); /* Enable Overflow T1 TouchScreen */

  interrupts(); // enable all interrupts//MPU

  if(APP_MODE!=2) Serial.print("Fin inicializacion");
}

  /**********************************/
  /*       Ball And Plate Loop      */
  /**********************************/

void loop()
{
  /* If an error occurs, program is stopped */
  if (!dmpReady) return;

  /* While there is no interruption */
  while (!mpuInterrupt && fifoCount < packetSize)
  {
      /**************************/
      /*  Serial Port Reception */
      /**************************/
      while (Serial.available())
      {
        //newline or max of 9 chars
        int index = Serial.readBytesUntil('\n', buffer, 9);
        buffer[index] = '\0';

        if(APP_MODE!=2) Serial.println(buffer);

        /* The string is clasified depending on the command */

        if(buffer[0] == 'R' && buffer[1] == 'Y')
        {
```

```
      /* First 2 characters are deleted */
      for (int i=2; i!= index ; i++)  buffer[i-2]= buffer[i];
      buffer[index-2]= '\0';
      Serial.print("Valor leido RefY ");
      Serial.println(buffer);  //so you can see the captured String
      Serial.print(" ");
      RefThetaY = atoi(buffer);  //convert readString into a number
      Serial.println(RefThetaY); //so you can see the integer
    }
    else if(buffer[0] == 'R' && buffer[1] == 'X')
    {
      /* First 2 characters are deleted */
      for (int i=2; i!= index ; i++)  buffer[i-2]= buffer[i];
      buffer[index-2]= '\0';
      Serial.print("Valor leido RefX ");
      Serial.println(buffer);  //so you can see the captured String
      Serial.print(" ");
      RefThetaX = atoi(buffer);  //convert readString into a number
      Serial.println(RefThetaX); //so you can see the integer
    }
    else if(buffer[0] == 'K' && buffer[1] == 'P'&& buffer[2] == 'Y')
    {
      /* First 3 characters are deleted */
      for (int i=3; i!= index ; i++)  buffer[i-3]= buffer[i];
      buffer[index-3]= '\0';
      Serial.print("Valor leido K_pY ");
      Serial.println(buffer);  //so you can see the captured String
      Serial.print(" ");
      float aux = atoi(buffer);  //convert readString into a number
      K_pY = aux*0.001;
      Serial.println(aux);  //so you can see the captured int
      Serial.print(" ");
      Serial.println(K_pY); //so you can see the integer
    }
    else if(buffer[0] == 'K' && buffer[1] == 'P'&& buffer[2] == 'X')
    {
      /* First 3 characters are deleted */
      for (int i=3; i!= index ; i++)  buffer[i-3]= buffer[i];
      buffer[index-3]= '\0';
      Serial.print("Valor leido K_pX ");
      Serial.println(buffer);  //so you can see the captured String
      Serial.print(" ");
      float aux = atoi(buffer);  //convert readString into a number
      K_pX = aux*0.001;
      Serial.println(aux);  //so you can see the captured int
      Serial.print(" ");
      Serial.println(K_pX); //so you can see the integer
    }
    else if(buffer[0] == 'K' && buffer[1] == 'D'&& buffer[2] == 'Y')
    {
      /* First 3 characters are deleted */
      for (int i=3; i!= index ; i++)  buffer[i-3]= buffer[i];
      buffer[index-3]= '\0';
      Serial.print("Valor leido K_dY ");
      Serial.println(buffer);  //so you can see the captured String
      Serial.print(" ");
      float aux = atoi(buffer);  //convert readString into a number
      K_dY = aux*0.001;
      Serial.println(aux);  //so you can see the captured int
```

```
          Serial.print(" ");
          Serial.println(K_dY); //so you can see the integer
        }
        else if(buffer[0] == 'K' && buffer[1] == 'D'&& buffer[2] == 'X')
        {
          /* First 3 characters are deleted */
          for (int i=3; i!= index ; i++)  buffer[i-3]= buffer[i];
          buffer[index-3]= '\0';
          Serial.print("Valor leido K_dX ");
          Serial.println(buffer);  //so you can see the captured String
          Serial.print(" ");
          float aux = atoi(buffer);  //convert readString into a number
          K_dX = aux*0.001;
          Serial.println(aux);  //so you can see the captured int
          Serial.print(" ");
          Serial.println(K_dX); //so you can see the integer
        }
        else if(buffer[0] == 'T' && buffer[1] == 'R'&& buffer[2] == 'A')
        {
          /* First 3 characters are deleted */
          for (int i=3; i!= index ; i++)  buffer[i-3]= buffer[i];
          buffer[index-3]= '\0';
          TrajGUI = atoi(buffer);  //convert readString into a number
        }
        buffer[0] = '\0';
      }
   }
  // There is an interruption generated by MPU-6050
  mpuInterrupt = false; /* Interrupt flag is set to FALSE */
  mpuIntStatus = mpu.getIntStatus();

  /* FIFO state is controlled, making sure there is no overflow */
  fifoCount = mpu.getFIFOCount();

  if ((mpuIntStatus & 0x10) || fifoCount == 1024)
  {
    mpu.resetFIFO();
    Serial.println(F("FIFO overflow!"));
  }
  else if (mpuIntStatus & 0x02)
  {
    /* Wait for correct available data length, should be a VERY short wait
*/
    while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

    /* Read a packet from FIFO */
    float TimeGetfifo_0= micros();
    mpu.getFIFOBytes(fifoBuffer, packetSize);
    float TimeGetfifo_1= micros();

    /* track FIFO count here in case there is > 1 packet available
     *  (this lets us immediately read more without waiting for an
interrupt) */
    fifoCount -= packetSize;

   /* Get Yaw, Pitch, Roll */
   mpu.dmpGetQuaternion(&q, fifoBuffer);
   mpu.dmpGetGravity(&gravity, &q);
   mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
```

```
   /* Angle data is changed to degrees, and offset is added */
   if((ypr[2] * 180/M_PI)>0) ang_x = -(ypr[2] * 180/M_PI - 180 + offsetX);
   else ang_x = -(ypr[2] * 180/M_PI + 180 + offsetX);

   if((ypr[1] * 180/M_PI)>0) ang_y = ypr[1] * 180/M_PI - 180 + offsetY;
   else ang_y = ypr[1] * 180/M_PI + 180 + offsetY;

   if(MEAS_MPU)
   {
    Serial.print("\t");
    Serial.print(ang_x);
    Serial.print("\t");
    Serial.print(ang_y);
    Serial.print("\tFifo ");
    Serial.print(fifoCount);

    TimeMeas_0= micros();
    float TimeBetweenMeas= (TimeMeas_0 - TimeMeas_1)*0.001;
    Serial.print("\t");
    Serial.print(TimeBetweenMeas);
    Serial.println(" ms");
    TimeMeas_1= TimeMeas_0;
    }
  }


} /* End void loop */


  /**********************************/
  /*   Interrupt Service Routines    */
  /**********************************/


  /*************************/
  /*   Timer 1 Compare A    */
  /*************************/
  /* Timer 1 Compare Interrupt A Service Routine */

ISR(TIMER1_COMPA_vect)
{
  int ControlType= Ts.MeasInterruptGenerator();

  if(ControlType==1) /* X Control */
  {
    double errorX = (RefX - (Ts._X_sample/100));
    double dErrX = (errorX - lastErrX);
    errSumX += errorX;

    /* PID X Controller */
    double Incr_X;
    Incr_X = -(K_pX * errorX + K_iX * errSumX + K_dX * dErrX);

    /* X Saturation */
    if (Incr_X > 4) Incr_X= 4;
    else if (Incr_X < -4) Incr_X= -4;

    if(!UPPER_CONTROL_DISABLED) RefThetaX = Incr_X;
```

```
    /* Last error value is stored */
    lastErrX = errorX;
  }

    if(ControlType==2)  /* Y Control */
    {
      double errorY = (RefY - (Ts._Y_sample/100));

      double dErrY = (errorY - lastErrY);
      errSumY += errorY;

      /* PID Y Controller */
      double Incr_Y;

      Incr_Y = (K_pY * errorY + K_iY * errSumY + K_dY * dErrY);

      /* Y Saturation */
      if (Incr_Y > 4) Incr_Y= 4;
      else if (Incr_Y < -4) Incr_Y= -4;

      if(!UPPER_CONTROL_DISABLED) RefThetaY = Incr_Y;

      /* Last error value is stored */
      lastErrY = errorY;
    }
} /* End ISR(TIMER1_COMPA_vect) */


  /*************************/
  /*   Timer 1 Overflow    */
  /*************************/
  /* Timer 1 Overflow Service Routine */

int cont=0;

ISR(TIMER1_OVF_vect)
{
  cont++;
  if(Ts._N_Overflow == cont)
  {
    /* Touchscreen Control Overflow */
    Ts.SetNumberMeas();
    cont=0;

    if (APP_MODE == 1) /* Arduino Serial Plotter */
    {
      Serial.print(Ts._X_sample/100);
      Serial.print("\t");
      Serial.println(Ts._Y_sample/100);
    }
    else if (APP_MODE == 2) /* Python GUI Plotter */
    {
      float X_sample = Ts._X_sample/100;
      float Y_sample = Ts._Y_sample/100;

      sendToPC_float(&X_sample);
      sendToPC_float(&Y_sample);
      sendToPC_float(&RefX);
```

```
      sendToPC_float(&RefY);
    }
  }

  /* Trajectory Update */

  if (TrajGUI== 1) /* Circle */
  {
    RefX = 80*cos(incr_ref);
    RefY = 80*sin(incr_ref);
  }
  if (TrajGUI== 0) /* x=0 y=0 */
  {
    RefX = 0;
    RefY = 0;
  }
  if (TrajGUI== 2) /* Lenmiscate */
  {
    RefX = (100*cos(incr_ref))/(1+sin(incr_ref)*sin(incr_ref));
    RefY =
(100*sin(incr_ref)*cos(incr_ref))/(1+sin(incr_ref)*sin(incr_ref));
  }
  incr_ref=incr_ref+INCR_TIME;


} /* End ISR(TIMER1_OVF_vect) */


  /***************************/
  /*   Timer 2 Compare A     */
  /***************************/
  /* Timer 2 Compare Interrupt A Service Routine */

ISR(TIMER2_COMPA_vect) // timer compare interrupt service routine
{
  OCR2A = M1.PulseGenerator_Vcte(OCR2A);
  if (DEBUG_PULSES==1) Serial.print("B");

}

  /***************************/
  /*   Timer 2 Compare B     */
  /***************************/
  /* Timer 2 Compare Interrupt B Service Routine */

ISR(TIMER2_COMPB_vect) // timer compare interrupt service routine
{
  OCR2B = M2.PulseGenerator_Vcte(OCR2B);
  if (DEBUG_PULSES==1) Serial.print("A");
}



  /***************************/
  /*   Timer 2 Overflow      */
  /***************************/
  /* Timer 2 Overflow Service Routine */

int contT2=0;
ISR(TIMER2_OVF_vect)
```

```
{
  contT2++;

  if (contT2 > 1)
  {
    contT2=0;

    if(DEBUG == 1)
    {
      Serial.print("Co ");
      Serial.print(" COAngle");
      Serial.print(" X_");
      Serial.print(ang_x);
      Serial.print(" Y_");
      Serial.print(ang_y);
    }

    /* NOTE: Plate angle is read in loop() from MPU6050 */

   /* PID Y Controller (Low-level)*/
    double errorThetaX = (RefThetaX - (ang_x));
    double errorThetaY = (RefThetaY - (ang_y));

    double dErrThetaX = (errorThetaX - lastErrThetaX);
    errSumThetaX += errorThetaX;

    double dErrThetaY = (errorThetaY - lastErrThetaY);
    errSumThetaY += errorThetaY;

    /* Angle increment to be changed by Stepper Motors */
    double Incr_X = (K_pThetaX * errorThetaX + K_iThetaX * errSumThetaX +
K_dThetaX * dErrThetaX);
    double Incr_Y = (K_pThetaY * errorThetaY + K_iThetaY * errSumThetaY +
K_dThetaY * dErrThetaY);

    /* Increment Angle Saturation in X axis */
    if (Incr_X > 10) Incr_X= 10;
    else if (Incr_X < -10) Incr_X= -10;

    /* Increment Angle Saturation in Y axis */
    if (Incr_Y > 10) Incr_Y= 10;
    else if (Incr_Y < -10) Incr_Y= -10;

    OutputMotorX = OutputMotorX + Incr_X;
    OutputMotorY = OutputMotorY + Incr_Y;

    /* Last error value is stored */
    lastErrThetaX = errorThetaX;
    lastErrThetaY = errorThetaY;

    /* Plate Angle Saturation in X axis */
    if (OutputMotorX>90)
    {
      OutputMotorX=90;
    }
    else if (OutputMotorX<0)
    {
      OutputMotorX= 0;
    }
```

```
    /* Plate Angle Saturation in Y axis */
    if (OutputMotorY>90)
    {
      OutputMotorY=90;
    }
    else if (OutputMotorY<0)
    {
      OutputMotorY= 0;
    }

     M1.SetAnglePosition(OutputMotorX);
     M2.SetAnglePosition(OutputMotorY);

     if(DEBUG == 1)
     {
       Serial.print(" MotorX_");
       Serial.print(OutputMotorX);
       Serial.print(" MotorY_");
       Serial.println(OutputMotorY);
     }

     if (DEBUG_PULSES==1)
     {
      Serial.print("\n O ");
      Serial.print(M1._PulseCycle);
      Serial.print(" ");
      Serial.print(M2._PulseCycle);
      Serial.print(" ");
     }
  }
}


  /**********************************/
  /*       Datalogger Function      */
  /**********************************/

void sendToPC_uint( volatile uint16_t* data)
{
  byte* byteData = (byte*)(data);     // Casting to a byte pointer
  Serial.write(byteData, 2);          // Send through Serial to the PC
}

void sendToPC_int( int* data)
{
  byte* byteData = (byte*)(data);     // Casting to a byte pointer
  Serial.write(byteData, 2);          // Send through Serial to the PC
}

void sendToPC_float(float* data)
{
  byte* byteData = (byte*)(data);     // Casting to a byte pointer
  Serial.write(byteData, 4);          // Send through Serial to the PC
}
```

ANEXO B: PROGRAMACIÓN EN PYTHON

```
# -*- coding: utf-8 -*-
"""
/****************************************/
/********* Ball And Plate Monitor *********/
/*********      main script       *********/
/****************************************/

First version of Python Plotter for BallAndPlate System.

Based on https://www.thepoorengineer.com/en/arduino-python-plot/

Created by Ramon Iglesias, 2019


"""

from threading import Thread
import serial
import time
import collections
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import struct
import copy
import pandas as pd
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import tkinter as Tk
from tkinter.ttk import Frame

class Window(Frame):
    def __init__(self, figure1, column1, row1,
                           figure2, column2, row2,
                           figure3, column3, row3,
                           master, SerialReference):

        Frame.__init__(self, master)
        self.entries = []
        self.setPoint = None
        # a reference to the master window
        self.master = master
        # keep a reference to our serial connection so that we can use it for
        # bi-directional communicate from this class
        self.serialReference = SerialReference
        # initialize the main plot window with our settings
        self.initWindow(figure1, column1, row1)
        # initialize two small plot windows with our settings
        self.initWindowSmall(figure2, column2, row2)
        self.initWindowSmall(figure3, column3, row3)


    def initWindow(self, figure, column, row):
        self.master.title("Real Time Plot")
        canvas = FigureCanvasTkAgg(figure, master=self.master)
        canvas.get_tk_widget().grid(column=column,row=row, rowspan=2)

    def initWindowSmall(self, figure, column, row):
        self.master.title("Real Time Plot")
        canvas = FigureCanvasTkAgg(figure, master=self.master)
        canvas.get_tk_widget().grid(column=column,row=row)


class MainApp:
```

```
63.     def TrajCircle(self): # Button Functionality
64.         self.serial.serialConnection.write(b"TRA1");
65.         time.sleep(2)
66.
67.     def TrajCenter(self): # Button Functionality
68.         self.serial.serialConnection.write(b"TRA0");
69.         time.sleep(2)
70.
71.     def TrajLemniscate(self): # Button Functionality
72.         self.serial.serialConnection.write(b"TRA2");
73.         time.sleep(2)
74.
75.     def __init__(self, master, serial, maxPlotLength):
76.         self.master = master
77.         self.serial = serial
78.
79.         # plotting settings (main window)
80.         pltInterval = 25    # Period at which the plot animation updates [ms]
81.         xmin_plate = -180
82.         xmax_plate = 180
83.         ymin_plate = -(150)
84.         ymax_plate = 150
85.         # plotting settings (x vs time window)
86.         xmin_plotX = -0.10 * maxPlotLength
87.         xmax_plotX = maxPlotLength + 10
88.         ymin_plotX = -(180)
89.         ymax_plotX = 180
90.         # plotting settings (y vs time window)
91.         xmin_plotY = -0.10 * maxPlotLength
92.         xmax_plotY = maxPlotLength + 10
93.         ymin_plotY = -(150)
94.         ymax_plotY = 150
95.
96.         # Main window plot axis configuration
97.         fig_plate = plt.figure(figsize=(10, 8))
98.         ax_plate = plt.axes(xlim=(xmin_plate, xmax_plate),
99.                             ylim=(float(ymin_plate - (ymax_plate - ymin_plate) / 10),
100.                                 float(ymax_plate + (ymax_plate - ymin_plate) / 10)))
101.        ax_plate.set_title('Ball and Plate')
102.        ax_plate.set_xlabel("X axis (mm)")
103.        ax_plate.set_ylabel("Y axis (mm)")
104.        plt.minorticks_on()
105.        plt.legend(loc="upper left")
106.        # Customize the major grid
107.        plt.grid(which='major', linestyle='-', linewidth='0.5', color='black')
108.        # Customize the minor grid
109.        plt.grid(which='minor', linestyle=':', linewidth='0.5', color='black')
110.
111.        # x vs Time window plot axis configuration
112.        fig_plotX = plt.figure(figsize=(8, 4))
113.        ax_plotX = plt.axes(xlim=(xmin_plotX, xmax_plotX),
114.                            ylim=(float(ymin_plotX - (ymax_plotX - ymin_plotX) / 10),
115.                                  float(ymax_plotX + (ymax_plotX - ymin_plotX) / 10)))
116.        ax_plotX.set_title('Eje X vs Tiempo')
117.        ax_plotX.set_xlabel("")
118.        ax_plotX.set_ylabel("Y axis (mm)")
119.        plt.minorticks_on()
120.        plt.legend(loc="upper left")
121.        # Customize the major grid
122.        plt.grid(which='major', linestyle='-', linewidth='0.5', color='black')
123.        # Customize the minor grid
124.        plt.grid(which='minor', linestyle=':', linewidth='0.5', color='black')
125.
126.        # y vs Time window plot axis configuration
127.        fig_plotY = plt.figure(figsize=(8, 4))
128.        ax_plotY = plt.axes(xlim=(xmin_plotY, xmax_plotY),
129.                            ylim=(float(ymin_plotY - (ymax_plotY - ymin_plotY) / 10),
130.                                  float(ymax_plotY + (ymax_plotY - ymin_plotY) / 10)))
```

```
131.        ax_plotY.set_title('Eje Y vs Tiempo')
132.        ax_plotY.set_xlabel("")
133.        ax_plotY.set_ylabel("Y axis (mm)")
134.        plt.minorticks_on()
135.        plt.legend(loc="upper left")
136.        # Customize the major grid
137.        plt.grid(which='major', linestyle='-', linewidth='0.5', color='black')
138.        # Customize the minor grid
139.        plt.grid(which='minor', linestyle=':', linewidth='0.5', color='black')
140.
141.        # Location of each plot in the main window (row and column)
142.        column_plate=0
143.        row_plate=1
144.        column_plotX=1
145.        row_plotX=1
146.        column_plotY=1
147.        row_plotY=2
148.        # This class attaches figures to the tkinter Canvas
149.        app = Window(fig_plate, column_plate, row_plate,
150.                     fig_plotX, column_plotX, row_plotX,
151.                     fig_plotY, column_plotY, row_plotY,
152.                     master, serial)
153.
154.        # Main Plot is linked with a real-time update function
155.        lineLabel = ['X', 'Y', 'X_ref','Y_ref']
156.        # linestyles for the different plots
157.        style = ['r-', 'c-', 'b-', 'c-']
158.        #The coordinate system of the Axes; (0,0) is bottom left of the axes,
159.        # and (1,1) is top right of the axes.
160.        timeText = ax_plate.text(0.70, 0.95, '', transform=ax_plate.transAxes)
161.        lines = []
162.        lineValueText = []
163.        for i in range(4):
164.            lines.append(ax_plate.plot([], [], style[i], label=lineLabel[i])[0])
165.            lineValueText.append(ax_plate.text(0.70, 0.90-
    i*0.05, '', transform=ax_plate.transAxes))
166.        anim_plate= animation.FuncAnimation(fig_plate, self.serial.getSerialData,
167.                                            fargs=(lines, lineValueText, lineLabel, timeText),
168.                                            interval=pltInterval)    # fargs has to be a tuple
169.        # Regarding fargs: Additional arguments to pass to each call to func.(getSerialData)
170.
171.        # PLOT X AXIS is linked with a real-time update function
172.        lineLabel = ['X', 'Y', 'Z']
173.        style = ['r-', 'c-', 'b-']  # linestyles for the different plots
174.        #The coordinate system of the Axes; (0,0) is bottom left of the axes,
175.        # and (1,1) is top right of the axes.
176.        timeText = ax_plotX.text(0.70, 0.95, '', transform=ax_plotX.transAxes)
177.        lines = []
178.        lineValueText = []
179.        for i in range(2):
180.            lines.append(ax_plotX.plot([], [], style[i], label=lineLabel[i])[0])
181.            lineValueText.append(ax_plotX.text(0.70, 0.90-
    i*0.05, '', transform=ax_plotX.transAxes))
182.        anim_plotX= animation.FuncAnimation(fig_plotX, self.serial.getPlotXData,
183.                                            fargs=(lines, lineValueText, lineLabel, timeText),
184.                                            interval=pltInterval)    # fargs has to be a tuple
185.
186.        # PLOT Y AXIS is linked with a real-time update function
187.        lineLabel = ['X', 'Y', 'Z']
188.        style = ['r-', 'c-', 'b-']  # linestyles for the different plots
189.        #The coordinate system of the Axes; (0,0) is bottom left of the axes,
190.        # and (1,1) is top right of the axes.
191.        timeText = ax_plotY.text(0.70, 0.95, '', transform=ax_plotY.transAxes)
192.        lines = []
```

```
193.         lineValueText = []
194.         for i in range(2):
195.             lines.append(ax_plotY.plot([], [], style[i], label=lineLabel[i])[0])
196.             lineValueText.append(ax_plotY.text(0.70, 0.90-
    i*0.05, '', transform=ax_plotY.transAxes))
197.         anim_plotY= animation.FuncAnimation(fig_plotY, self.serial.getPlotYData,
198.                                             fargs=(lines, lineValueText, lineLabel, timeText),
199.                                             interval=pltInterval)    # fargs has to be a tuple
200.
201.
202.         ###### Trajectory Selection ######
203.
204.         # A frame with grey background is created
205.         self.TrajSelection = Tk.Frame(width=300, height=150, background='grey')
206.         # The frame is located in column 0, row 3
207.         self.TrajSelection.grid(column=0,row=3)
208.
209.         # A button for each trajectory is added in GUI
210.         self.Traj0 = Tk.Button(master=self.TrajSelection,
211.                                text="X=0 Y=0",
212.                                command=self.TrajCenter)
213.         self.Traj0.grid(column=1,row=1)
214.
215.         self.Traj1 = Tk.Button(master=self.TrajSelection,
216.                                text="Circle",
217.                                command=self.TrajCircle)
218.         self.Traj1.grid(column=2,row=1)
219.
220.         self.Traj2 = Tk.Button(master=self.TrajSelection,
221.                                text="Lemniscate",
222.                                command=self.TrajLemniscate)
223.         self.Traj2.grid(column=3,row=1)
224.
225.         ###### Connection retry Button ######
226.
227.         # A frame with grey background is created
228.         self.ConnectSelection = Tk.Frame(width=300, height=150, background='white')
229.         # The frame is located in column 0, row 3
230.         self.ConnectSelection.grid(column=1,row=3)
231.
232.         # Buttons for connection retry and close serial port are added
233.         self.ConnectButton = Tk.Button(master=self.ConnectSelection,
234.                                        text="Connection Retry",
235.                                        command=self.serial.ConnectSerialPort)
236.         self.ConnectButton.grid(column=6,row=1)
237.
238.
239.         # Keep this handle alive, or else figure will disappear
240.         master.mainloop()
241.
242.
243.
244.def main():
245.     # Parameters of Serial Port
246.     portName = 'COM3'
247.     baudRate = 2000000
248.     maxPlotLength = 50      # number of points in stored to plot real time trajectory
249.     dataNumBytes = 4        # number of bytes of 1 data point - If 2, int data. If 4, float dat
    a
250.     numPlots = 4            # number of data points read through Serial Port
251.
252.     # initializes all required variables to manage Serial Port
253.     s = serialPlot(portName, baudRate, maxPlotLength, dataNumBytes, numPlots)
254.
255.
256.     root = Tk.Tk()
```

```
257.    app= MainApp(root, s, maxPlotLength)
258.
259.    s.close()
260.
261.
262.class serialPlot:
263.    def __init__(self, serialPort='/dev/ttyUSB0', serialBaud=38400, plotLength=100,
264.                 dataNumBytes=4, numPlots=1):
265.
266.        self.port = serialPort
267.        self.baud = serialBaud
268.        self.plotMaxLength = plotLength
269.        self.dataNumBytes = dataNumBytes
270.        self.numPlots = numPlots
271.        self.rawData = bytearray(numPlots * dataNumBytes) # Where data is stored
272.        self.dataType = None
273.        if dataNumBytes == 2:
274.            self.dataType = 'h'     # 2 byte integer
275.        elif dataNumBytes == 4:
276.            self.dataType = 'f'     # 4 byte float
277.        self.data = []
278.        for i in range(numPlots):   # give an array for each type of data and store them in a l
    ist
279.            self.data.append(collections.deque([0] * plotLength, maxlen=plotLength))
280.        self.isRun = True
281.        self.SerialPortError = False
282.        self.isReceiving = False
283.        self.thread = None
284.        self.plotTimer = 0
285.        self.previousTimer = 0
286.        # self.csvData = []
287.        self.ConnectSerialPort()
288.
289.    def ConnectSerialPort(self):
290.        # Serial Port tries to be opened
291.        print('Trying to connect to: ' + str(self.port) + ' at ' + str(self.baud) + ' BAUD.')
292.        try:
293.            self.serialConnection = serial.Serial(self.port, self.baud, timeout=4)
294.            print('Connected to ' + str(self.port) + ' at ' + str(self.baud) + ' BAUD.')
295.            self.SerialPortError = False
296.        except:
297.            self.SerialPortError = True
298.            print("Failed to connect with " + str(self.port) + ' at ' + str(self.baud) + ' BAUD
    .')
299.
300.        if not self.SerialPortError:
301.                self.readSerialStart()  # starts background thread if serial Port could be open
    ed
302.
303.    def CloseSerialPort(self):
304.         self.close()
305.
306.
307.    def readSerialStart(self):
308.        if self.thread == None:
309.            self.thread = Thread(target=self.backgroundThread)
310.            self.thread.start()
311.            # Block till we start receiving values
312.            while self.isReceiving != True:
313.                time.sleep(0.05)
314.
315.
316.    def getSerialData(self, frame, lines, lineValueText, lineLabel, timeText):
317.        currentTimer = time.perf_counter()
318.        # the first reading will be erroneous
319.        self.plotTimer = int((currentTimer - self.previousTimer) * 1000)
320.        self.previousTimer = currentTimer
321.        timeText.set_text('Plot Interval = ' + str(self.plotTimer) + 'ms')
```

```
322.         # so that the 3 values in our plots will be synchronized to the same sample time
323.         privateData = copy.deepcopy(self.rawData[:])
324.         for i in range(self.numPlots):
325.             data = privateData[(i*self.dataNumBytes):(self.dataNumBytes + i*self.dataNumBytes)]

326.             value,  = struct.unpack(self.dataType, data) # Data type is changed
327.             self.data[i].append(value)    # we get the latest data point and append it to our a
    rray
328.             lineValueText[i].set_text('[' + lineLabel[i] + '] = ' + str(value))
329.
330.         lines[0].set_data(self.data[0], self.data[1])  #Set the data X and Y
331.         lines[1].set_data(self.data[2], self.data[3])  #Set the data X and Y
332.         # Data can be stored in a csv data
333.         # self.csvData.append([self.data[0][-1], self.data[1][-1], self.data[2][-
    1], self.data[3][-1]])
334.
335.     def getPlotYData(self, frame, lines, lineValueText, lineLabel, timeText):
336.         lines[0].set_data(range(self.plotMaxLength), self.data[1])  #Set the data X
337.         lines[1].set_data(range(self.plotMaxLength), self.data[3])  #Set the data X
338.         #print(self.data[1])
339.
340.     def getPlotXData(self, frame, lines, lineValueText, lineLabel, timeText):
341.         lines[0].set_data(range(self.plotMaxLength), self.data[0])  #Set the data Y
342.         lines[1].set_data(range(self.plotMaxLength), self.data[2])  #Set the data Y
343.
344.     def backgroundThread(self):    # retrieve data
345.         time.sleep(1.0)  # give some buffer time for retrieving data
346.
347.         self.serialConnection.reset_input_buffer()
348.         while (self.isRun):
349.             self.serialConnection.readinto(self.rawData)
350.             self.isReceiving = True
351.
352.     def close(self):
353.         self.isRun = False
354.         if not self.SerialPortError:
355.             self.thread.join()
356.             self.serialConnection.close()
357.             print('Disconnected...')
358.         print('Window has been closed...')
359.         # df = pd.DataFrame(self.csvData)
360.         # df.to_csv('directory/data.csv')
361.
362.if __name__ == '__main__':
363.     main()
364.
```

REFERENCIAS

[1] «Arduino Project Hub,» 21 Mayo 2016. [En línea]. Available: https://create.arduino.cc/projecthub/davidhamor/ball-and-plate-c48027. [Último acceso: 25 Mayo 2019].

[2] «LEYBOLD,» [En línea]. Available: https://www.leybold-shop.com/ball-and-plate-control-system-33-052.html. [Último acceso: 25 Mayo 2019].

[3] J. A. Borja Conde y I. Alvarado Aldea, «Sección 3. Cambios De Hardware,» de *Desarrollo de un robot autoequilibrado basado en Arduino con motores paso a paso*, Sevilla, Escuela Técnica Superiro de Ingeniería, Universidad de Sevilla, 2018, pp. 27-37.

[4] Haina Touch Technology Co., Ltd., «HAINA TOUCH Official Store,» 09 11 2012. [En línea]. Available: https://m.es.aliexpress.com/item/32809597549.html?spm=a2g0n.detail-cache.0.0.f6e9a97CMzdbY. [Último acceso: 27 Abril 04].

[5] HantouchUSA, «www.sparkfun.com,» [En línea]. Available: https://www.sparkfun.com/datasheets/LCD/HOW%20DOES%20IT%20WORK.pdf. [Último acceso: 27 Abril 2019].

[6] Wikipedia, «Motor paso a paso,» [En línea]. [Último acceso: 28 Abril 2019].

[7] L. Llamas, «www.luisllamas.es,» [En línea]. Available: https://www.luisllamas.es/motores-paso-paso-arduino-driver-a4988-drv8825/. [Último acceso: 28 Abril 2019].

[8] B. Schmalz, «Easy Driver Stepper Motor Driver,» [En línea]. Available: https://www.schmalzhaus.com/EasyDriver/. [Último acceso: 11 05 2019].

[9] Allegro MicroSystems, Inc., «Data Sheet A3967 MICROSTEPPING DRIVER WITH TRANSLATOR,» [En línea]. Available: https://www.sparkfun.com/datasheets/Robotics/A3967.pdf. [Último acceso: 11 Mayo 2019].

[10] Arduino, «Arduino Uno,» [En línea]. Available: https://store.arduino.cc/arduino-uno-rev3. [Último acceso: 11 Mayo 2019].

[11] Atmel, «ATmega328P Data Sheet,» Enero 2015. [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf. [Último acceso: 11 Mayo 2019].

[12] InvenSense, «MPU-6050,» [En línea]. Available: https://www.invensense.com/products/motion-tracking/6-axis/mpu-6050/. [Último acceso: 16 Mayo 2019].

[13] InvenSense, «MPU-6000 and MPU-6050 Product Specification Revision 3.4,» 8 Agosto 2013. [En línea]. Available: https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf. [Último acceso: 21 Mayo 2019].

[14] L. Llamas, «El bus I2C en Arduino,» 18 Mayo 2016. [En línea]. Available:

https://www.luisllamas.es/arduino-i2c/. [Último acceso: 20 Mayo 2019].

[15] Robologs, «Tutorial de Arduino y MPU-6050,» 15 Octubre 2014. [En línea]. Available: https://robologs.net/2014/10/15/tutorial-de-arduino-y-mpu-6050/. [Último acceso: 13 05 2019].

[16] C. Chérigo y H. Rodríguez, «Evaluación de algoritmos de fusión de datos para estimación de la orientación de vehículos aéreos no tripulados,» *RIDTEC,* vol. 13, nº 2, pp. 90-99, Julio-Diciembre 2017.

[17] J. A. Castañeda Cárdenas, M. A. Nieto Arias y V. A. Ortiz Bravo, «Análisis y aplicación del filtro de Kalman a una señal con ruido aleatorio,» *Scientia et Technica,* vol. 18, nº 1, pp. 267-274, 2013.

[18] L. Llamas, «Medir la inclinación con IMU, Arduino y filtro complementario,» 7 Septiembre 2016. [En línea]. Available: https://www.luisllamas.es/medir-la-inclinacion-imu-arduino-filtro-complementario/. [Último acceso: 16 05 2019].

[19] J. Rowberg, «GitHub de la librería i2cdevlib,» [En línea]. Available: https://github.com/jrowberg/i2cdevlib. [Último acceso: 16 Mayo 2019].

[20] M. Nokhbeh y D. Khashabi, «Mathematical Modelling,» de *Modelling and Control of Ball-Plate System*, Tehran , Amirkabir University of Technology, 2011, pp. 3-8.

[21] J. A. Borja Conde y I. Alvarado Aldea, «Sección 4.3 Estrategia para implementar tren de pulsos para velocidad de los motores,» de *Desarrollo de un robot autoequilibrado basado en Arduino con motores paso a paso*, Sevilla, Universidad de Sevilla, 2018, pp. 47-49.

[22] L. Llamas, «Filtro paso bajo y paso alto exponencial (EMA) en Arduino,» [En línea]. Available: https://www.luisllamas.es/arduino-paso-bajo-exponencial/. [Último acceso: 01 Mayo 2019].

[23] J. Rowberg, «Github de la librería MPU6050,» 29 Septiembre 2018. [En línea]. Available: https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050. [Último acceso: 19 05 2019].

[24] J. Rowberg, «Github de la librería i2cdevlib,» 15 Octubre 2018. [En línea]. Available: https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/I2Cdev. [Último acceso: 19 Mayo 2019].

[25] J. Rowberg, «I2Cdevlib,» [En línea]. Available: https://www.i2cdevlib.com. [Último acceso: 19 Mayo 2019].

[26] J. Rowberg, «Github de la librería MPU6050_6Axis_MotionApps20.h,» 29 Septiembre 2018. [En línea]. Available: https://github.com/jrowberg/i2cdevlib/blob/master/Arduino/MPU6050/MPU6050_6Axis_MotionApps20.h. [Último acceso: 19 Mayo 2019].

[27] ThePoorEngineer, «Plotting Serial Data from Arduino in Real Time with Python,» 4 Febrero 2018. [En línea]. Available: https://www.thepoorengineer.com/en/arduino-python-plot/#multiple. [Último acceso: 3 Junio 2019].

[28] ThePoorEngineer, «Creating a Graphic User Interface (GUI) with Python,» 24 Febrero 2018. [En línea]. Available: https://www.thepoorengineer.com/en/python-gui/. [Último acceso: 3 Junio 2019].

[29] Python Software Foundation, «Threading Documentation,» [En línea]. Available: https://docs.python.org/3/library/threading.html. [Último acceso: 4 Junio 2019].

[30] Matplotlib, «Matplotlib Version 3.1.0,» [En línea]. Available: https://matplotlib.org/. [Último acceso: 4 Junio 2019].

[31] Pandas, «Pandas Python Data Analysis Library,» [En línea]. Available: https://pandas.pydata.org/. [Último acceso: 04 Junio 2019].

[32] Python Software Foundation, «tkinter Python Module,» [En línea]. Available: https://docs.python.org/3/library/tkinter.html#tkinter-modules. [Último acceso: 04 Junio 2019].

[33] C. Liechti, «pySerial documentation,» [En línea]. Available: pythonHosted.org/pyserial/index.html. [Último acceso: 11 Junio 2019].

[34] Python Software Foundation, «Classes (The Python Tutorial),» [En línea]. Available: https://docs.python.org/3.7/tutorial/classes.html. [Último acceso: 10 Junio 2019].

[35] J. A. Borja Conde y I. Alvarado Aldea, «Sección 4. Implementación de aceleración,» de *Desarrollo de un robot autoequilibrado basadoen Arduino con motores paso a paso*, Sevilla, Universidad de Sevilla, 2018, pp. 39-47.

[36] C. A. Ávila Ramos y F. A. Suarez Cardenas, «Diseño de controladores basados en técnicas de control optimo LQR+i y H2 para un prototipo del péndulo invertido sobre ruedas,» *Revista Politécnica ,* nº 15, pp. 45-51, 2012.