INGENIERÍA MECATRÓNICA

DI_CERO

DIEGO CERVANTES RODRÍGUEZ

ELECTRÓNICA DIGITAL: CIRCUITOS LÓGICOS, LENGUAJE VHDL Y VERILOG

XILINX (64-BIT PROJECT NAVIGATOR) & ADEPT

Control PID y Control Lazo Abierto con Arduino, VHDL y Verilog

Contenido

Introducción Teórica: Control PID

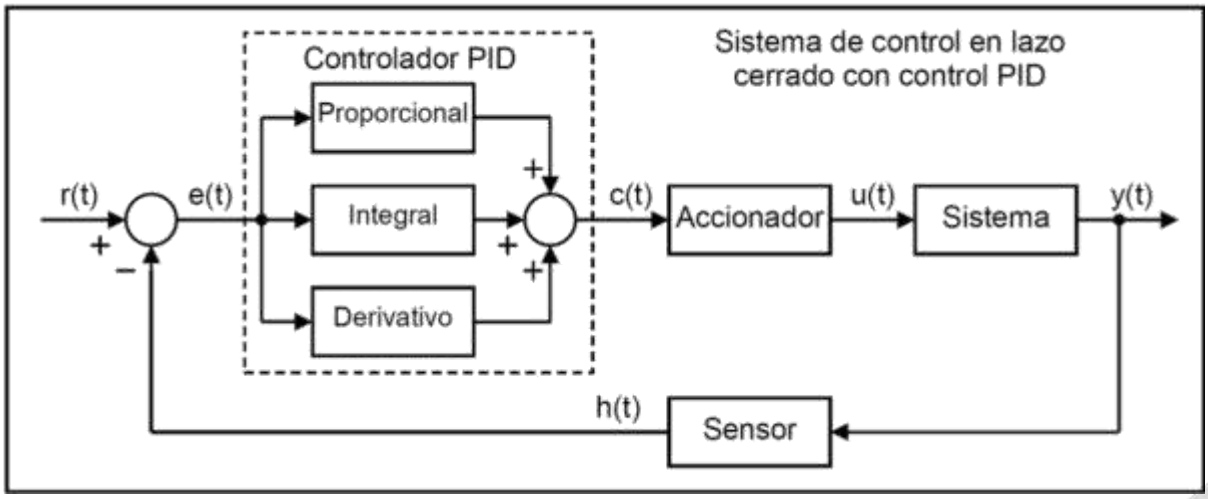
El **control PID** es un algoritmo utilizado en el **código de sistemas embebidos**, su objetivo es **lograr que alguna medición en un robot o sistema se alcance y mantenga en un valor deseado o punto de referencia (setpoint) con una mínima desviación o error**; como lo puede ser una posición, distancia, temperatura, flujo, orientación, entre otros.

Para que un **controlador PID** pueda trabajar **necesita contar con** al menos **un sensor**, **un actuador** y **un sistema embebido** que realice el análisis del sistema (**microcontrolador**, **FPGA**, **Raspberry Pi**, **etc.**); los pasos para llevar a cabo el control son los siguientes:

1. Primero **el sensor** **medirá el valor real del** **setpoint**.
2. El dato medido por el sensor será introducido al **sistema embebido**, donde **se realizará la resta del valor deseado** **menos** **el verdadero** en el sistema **para calcular el error:**

$$Error = e(t) = setPoint - valorMedidoPorElSensor = r(t) - Sensor$$

   a. **Dicho dato será transmitido al controlador PID**, que es un pedazo de código del **sistema embebido** encargado de realizar el análisis del **sistema** (**también llamado planta**) en el **presente**, **pasado** y **futuro**.
3. **El resultado será transferido del sistema embebido a un accionador** (que casi siempre es un motor), el cual **hará lo posible para corregir el error** y alcanzar el valor del **setpoint**.
4. Finalmente, **la respuesta será leída nuevamente** **por el sensor** **y por medio de una entrada de retroalimentación** se reiniciará el proceso. A esto se le llama control de bucle cerrado.



Las siglas de **PID** significan **Proporcional** **Integral** **y** **Derivativo**, donde el controlador analizará el sistema en el **presente, pasado y futuro** a través de 3 constantes **kp**, **ki** y **kd**:

- **Proporcional (kp):** La acción proporcional se encarga de **manejar el error del sistema en el presente**:
  - Cuando el **control P** **perciba que existe una diferencia no tan pequeña entre el valor medido actual y el valor deseado del setpoint**, se lo indicará al **actuador**, que **ejecutará una acción para reducir el error**. Cuanto mayor sea el error, mayor será la señal de control enviada al accionador para corregirlo. Pero **esto lo que ocasionará es una oscilación**, ya

que **el sistema solo reacciona cuando percibe el error,** por lo que **nunca alcanzará a eliminarlo completamente debido a la misma inercia de la mayoría de los sistemas** (que se opone al cambio). Además, uno de los problemas de esta fase es que, si el error es muy pequeño, no será identificado por el **control P**.

$$PID_P = Kp * Error = kp * e(t)$$

- **Integral (ki):** La acción integral se encarga de **manejar el error del sistema en el pasado**:
  - Lo que hace el **control I es considerar el historial de errores acumulados y actuar gradualmente para eliminar el error oscilante a lo largo del tiempo**, esto más que nada **toma en cuenta las fluctuaciones causadas por la corrección del control P, analizando el área bajo la curva del error y tratando de reducirla**, ayudando así al sistema a **alcanzar una estabilidad al reducir sus oscilaciones**. Por esta razón es que **el control I casi nunca se utiliza solo, mínimo se combina con el control P**. El tipo de **control PI** reconoce todo tipo de errores, aunque este sea muy pequeño, eliminándolo poco a poco de forma gradual hasta acercarlo lo más posible a ser cero.

$$PID_{PI} = PID_{PI} + (Kp * Error + Ki * Error) = PID_{PI} + \big((kp + ki) * e(t)\big)$$

- **Derivativo (kd):** La acción derivativa se encarga de **manejar el error del sistema en el futuro**:
  - El **control D considera la tasa de cambio de la curva del error** (el cambio que ha tenido a través del tiempo)**, si esto se observa en un solo punto de la gráfica se representará como una recta tangente,** así es como **la parte derivativa puede predecir hacia dónde se dirigirá el error en el futuro y prevenir oscilaciones excesivas o sobrecompensaciones**, alterando **la magnitud de la corrección aplicada por la parte proporcional para ajustarla a la velocidad de cambio del error, logrando así hacer cambios rápidos en la señal mandada hacia el actuador para alcanzar el valor deseado del setpoint**.

$$PID_{PD} = Kp * Error + Kd * \frac{Error - ErrorAnterior}{Intervalo\ de\ Tiempo} = kp * e(t) + kd * \frac{e(t) - e(t-1)}{t}$$

Ecuación PID en el código:

$$PID_{PID} = kp * e(t) + PID_{PI} + Ki * e(t) + kd * \frac{e(t) - e(t-1)}{t}$$

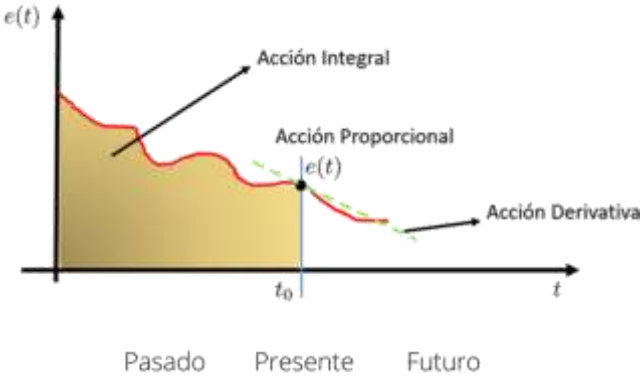
Ecuación PID diferencial:

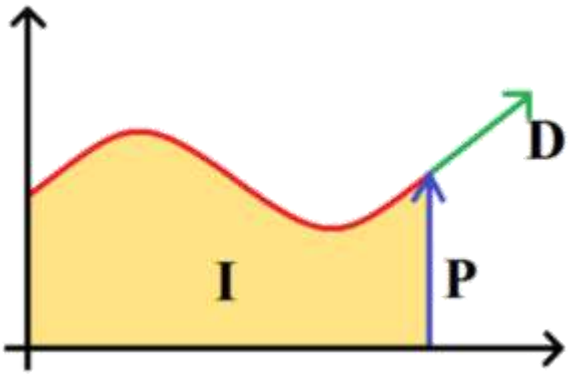
$$PID = kp * e(t) + Ki * \int_0^t e(\tau)d\tau + kd * \frac{de(t)}{dt} = kp\left(e(t) + \frac{1}{Ti} * \int_0^t e(t)dt + Td * \frac{de(t)}{dt}\right)$$

El **control PID** es aplicado de forma diferente para cada sistema, por lo que se deben probar varios valores en las constantes **kp**, **ki** y **kd** para su calibración. Usualmente primero se calibra **kp** para obtener una oscilación no muy grande alrededor **del valor deseado**, luego se calibra **kd** para obtener respuestas rápidas que alcancen el valor del **setpoint** y finalmente se calibra **ki** para reducir las oscilaciones producidas en el sistema antes de lograr igualar al **punto de referencia**.

El control PID es ampliamente utilizado en diversas aplicaciones; como control de temperatura en sistemas de calefacción y refrigeración, control de velocidad en motores eléctricos, sistemas de

navegación y control de posición en robótica, entre muchas otras. Su eficacia, versatilidad y capacidad para adaptarse a diferentes sistemas lo convierten en uno de los algoritmos de control más utilizados en la ingeniería y automatización.

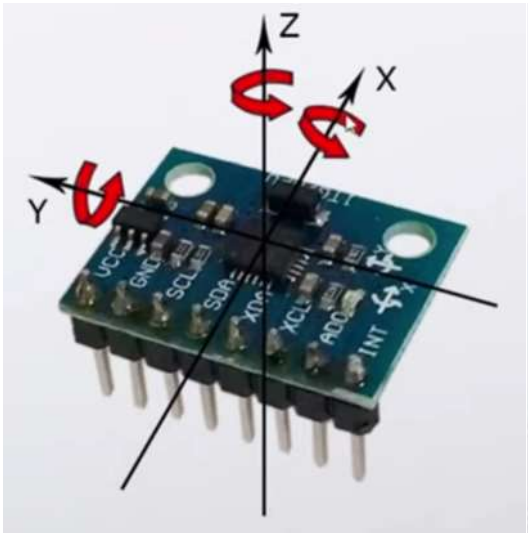


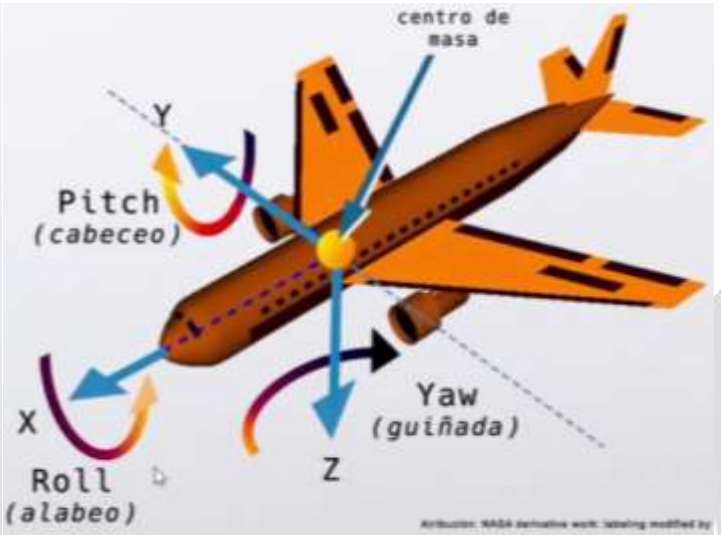


Ejemplo de Controlador PID para Drone: Sensor de Orientación MPU6050 y BLDC

Lo que se busca lograr con este **controlador PID (Proporcional Integral y Derivativo)** es que durante el inicio del vuelo de un drone se mantenga una posición recta para este que se eleve sin chocar, para ello recordemos que se **necesita contar con** al menos **un sensor**, **un actuador** y **un sistema embebido** que realice el análisis del sistema (**microcontrolador**, **FPGA**, **Raspberry Pi**, **etc.**). En este sistema o planta se cuenta con los siguientes componentes:

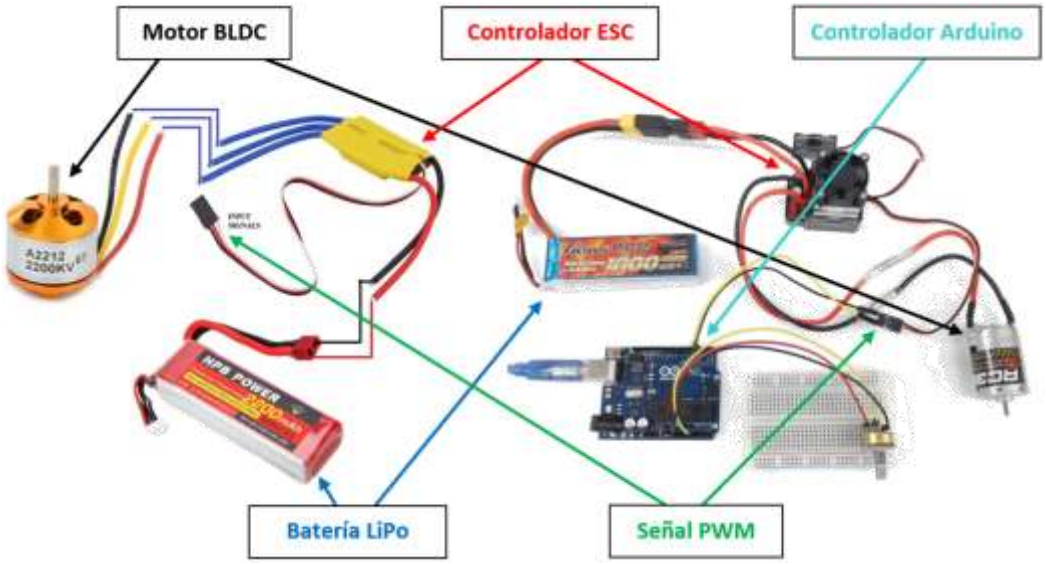
- **Sensor**: Para medir la posición del dispositivo se utilizará un sensor **acelerómetro** y **giróscopo MPU6050** para describir a través de una **comunicación serial con protocolo I2C** la orientación de un objeto tridimensional a través de 3 ángulos de Euler llamados Roll, Pitch y Yaw.





- **Actuador**: **Para corregir el error de orientación medido por el sensor** se utilizará un motor sin escobillas, también llamado brushless o BLDC, **el cual se maneja por medio de dos controladores, uno externo llamado ESC (Electronic Speed Controller)** que es alimentado por una **batería LiPo** y

un segundo controlador que es el mismo sistema embebido, el cual **por medio de una señal PWM de 50Hz cuyo duty cycle varía de 1 a 2ms** controlará su velocidad de rotación, pudiendo elegir velocidades de 0 a $KV * V_{ESC} = KV * V_{LiPo}\ [rpm]$; dicha señal se dirige al **ESC** y finalmente **la salida trifásica** obtenida se conecta a las 3 bobinas **A**, **B** y **C** del BLDC. **Los pines positivo y negativo del puerto que recibe la señal PWM llamado BEC (Battery Eliminator Circuit) proporcionan 5V para alimentar cualquier otro dispositivo,** que puede ser el mismo **Arduino**. Además, cabe mencionar que al cambiar el orden de conexión de sus cables **A**, **B** y **C**, se invierte el sentido de giro del motor.

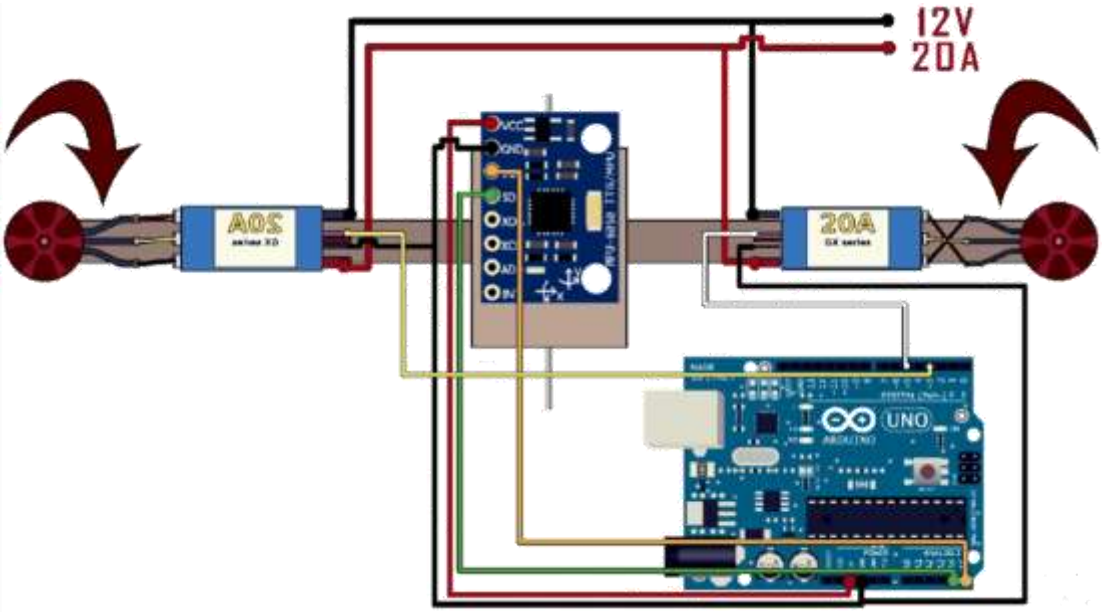


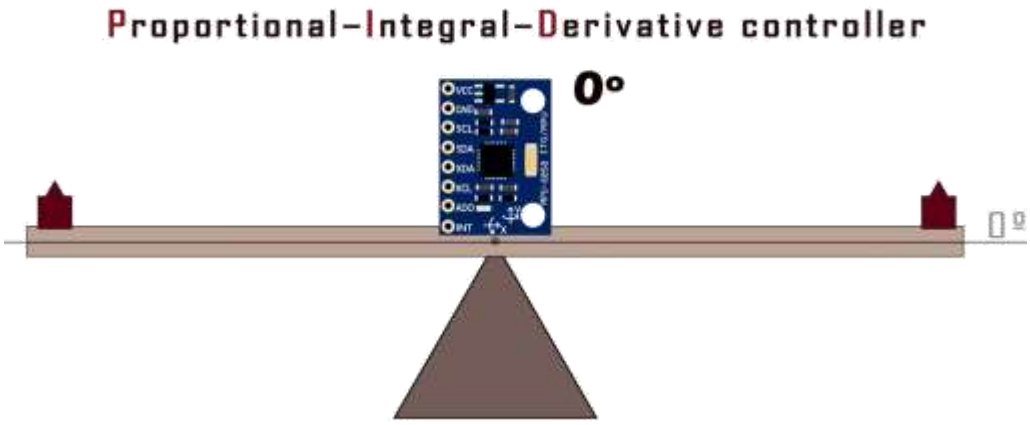
- **Sistema embebido**: El sistema embebido será un **microcontrolador Atmega328P**, el cual se incluye en la placa de desarrollo **Arduino UNO**, de este se utilizará el **Pin 11** para mandar la **señal PWM** hacia el **ESC** que controla la **velocidad de rotación del BLDC** (**actuador**) y los pines analógicos **A4** y **A5** se conectarán a los pines **SDA (Serial Data)** y **SCL (Serial Clock)** del sensor **MPU6050** para habilitar la comunicación serial I2C, también el pin digital 2 se conectará al pin **INT** del mismo sensor para habilitar las interrupciones externas.
    - Los datos de orientación obtenidos del sensor MPU6050 se podrán obtener a través de alguna de las siguientes dos librerías:
        - **Librería Wire:** No se conecta el Pin 2 de interrupciones externas (**INT**) y los cálculos se realizan de forma manual.
        - **librería i2cdevlib para establecer la comunicación I2C y librería Simple_MPU6050 para obtener las señales de orientación:** Al utilizar estas dos librerías, si se conecta el Pin 2 de interrupciones externas (**INT**) al microcontrolador y los resultados se obtienen de forma automática a través del **DMP (Digital Movement Processor)** incluido en el sensor MPU6050.

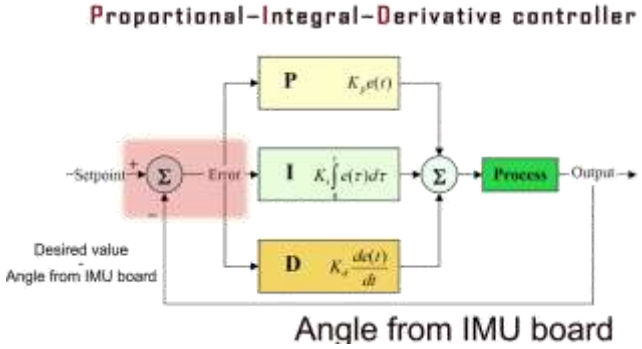
Con estos 3 componentes y conociendo la variable a controlar: $setpoint = 0°\ de\ Roll\ y\ 0°\ de\ Pitch$, se podrá ejecutar la **ecuación de control PID en el código del sistema embebido:**

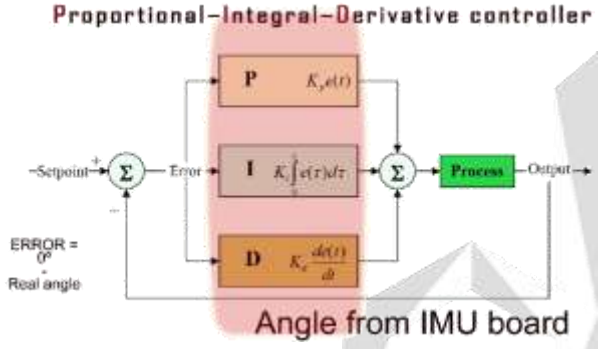
$$PID_{PID} = kp * e(t) + PID_{PI} + Ki * e(t) + kd * \frac{e(t) - e(t-1)}{t}$$

Diagrama de conexión del control PID aplicado al balanceo de un drone: Incluye un sensor de orientación MPU6050, un actuador BLDC y el sistema embebido Arduino UNO





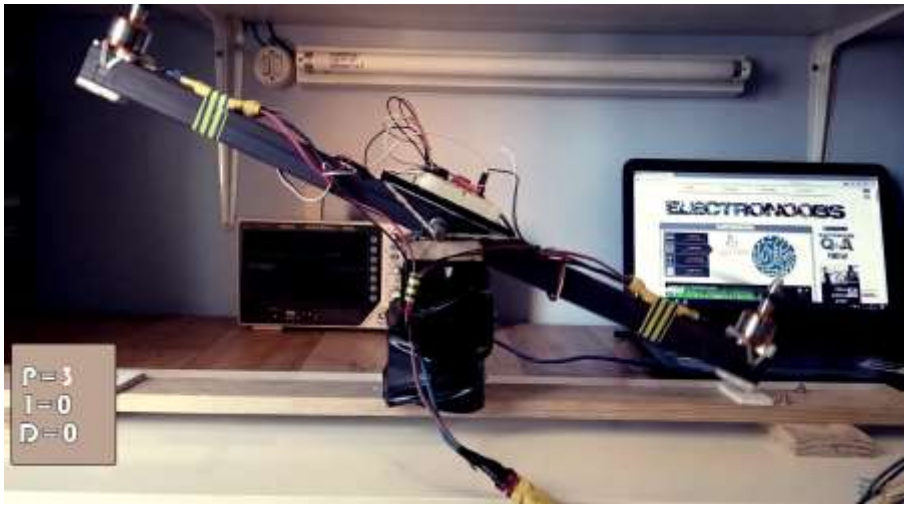


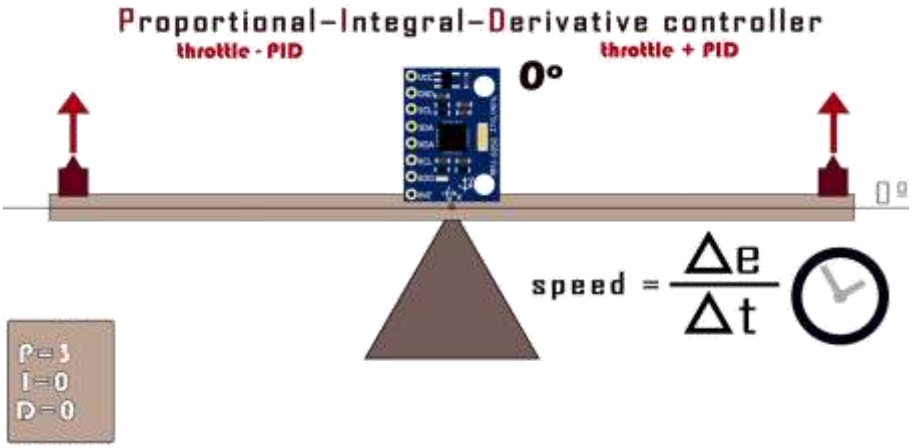


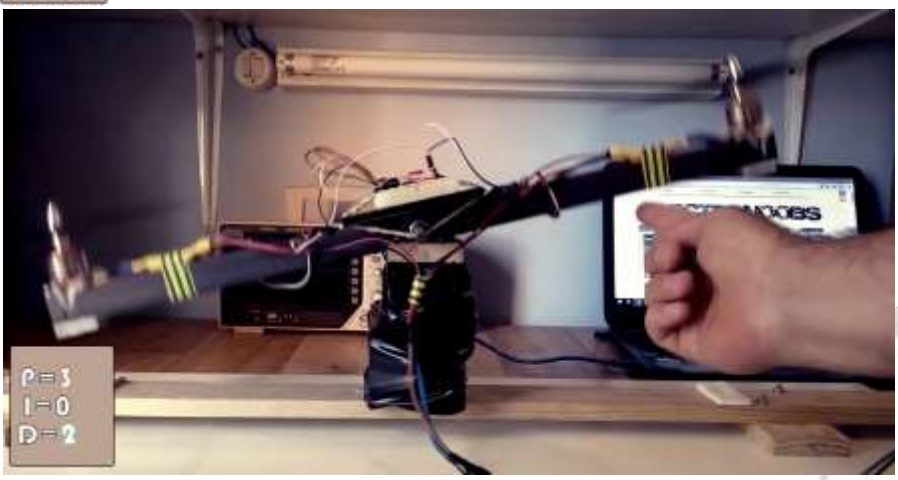
$$PID_{PID} = kp * e(t) + PID_{PI} + Ki * e(t) + kd * \frac{e(t)}{t}$$

Calibración de las Constantes PID: Kp, Kd Y Ki

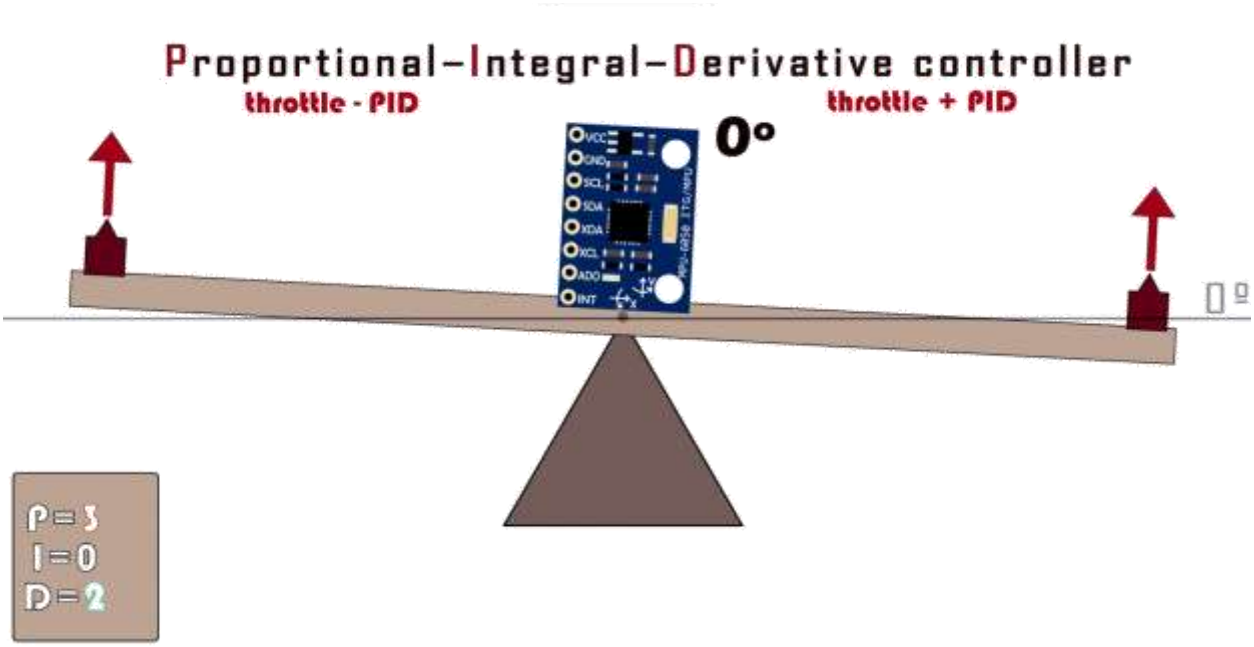
Recordemos que al aplicar un **controlador PID** a un sistema o planta, este se debe calibrar para optimizar su función, para ello se deben probar varios valores en las constantes **kp**, **ki** y **kd**. Usualmente primero se calibra **kp** para obtener una oscilación no muy grande alrededor **del valor deseado**, luego se calibra **kd** para obtener respuestas rápidas que alcancen el valor del **setpoint** y finalmente se calibra **ki** para reducir las oscilaciones producidas en el sistema antes de lograr igualar al **punto de referencia** o para detectar errores muy pequeños. Al final se obtiene un PID que responda bien a alteraciones externas inesperadas.

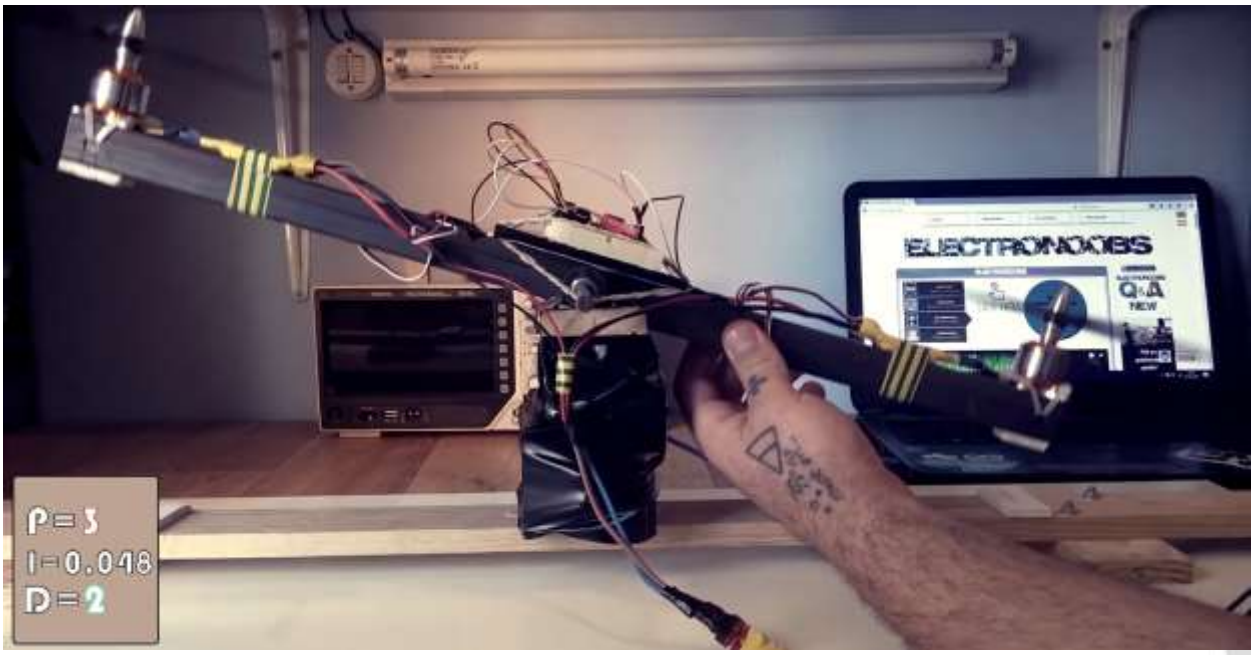






La **constante Ki** que se agrega al final se debe calibrar para **manejar errores mínimos u oscilaciones indeseadas en el sistema** al querer alcanzar el valor del **setpoint**. **El valor de esta suele** ser decimal y **encontrarse entre 0 y 1**, además de que **es acumulativa**, por lo que, **si el error dura mucho tiempo**, **el control integral irá aumentando gradualmente para contrarrestar al error con una fuerza acumulativa**.





Las constantes obtenidas cambiarán dependiendo de la forma, el peso, la función y tamaño del sistema, por lo cual, siempre deben ser calibradas.

Código Arduino

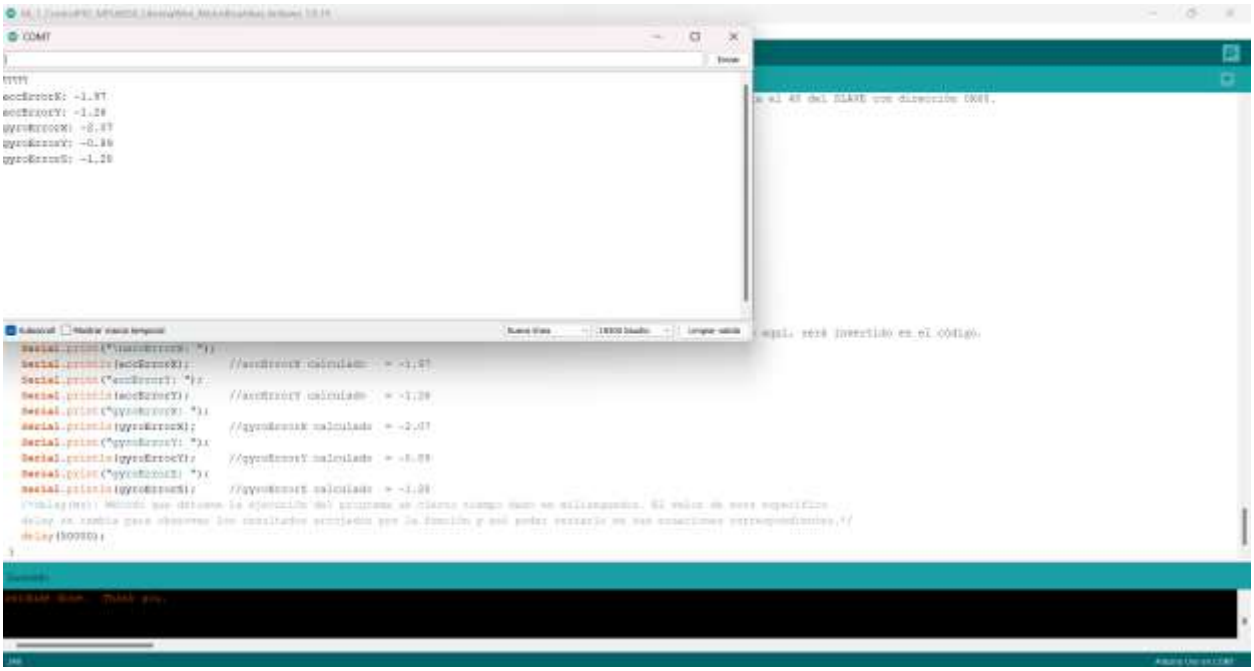
Control PID Drone: Sensor MPU6050 (**Librería Wire**)/Actuador BLDC:

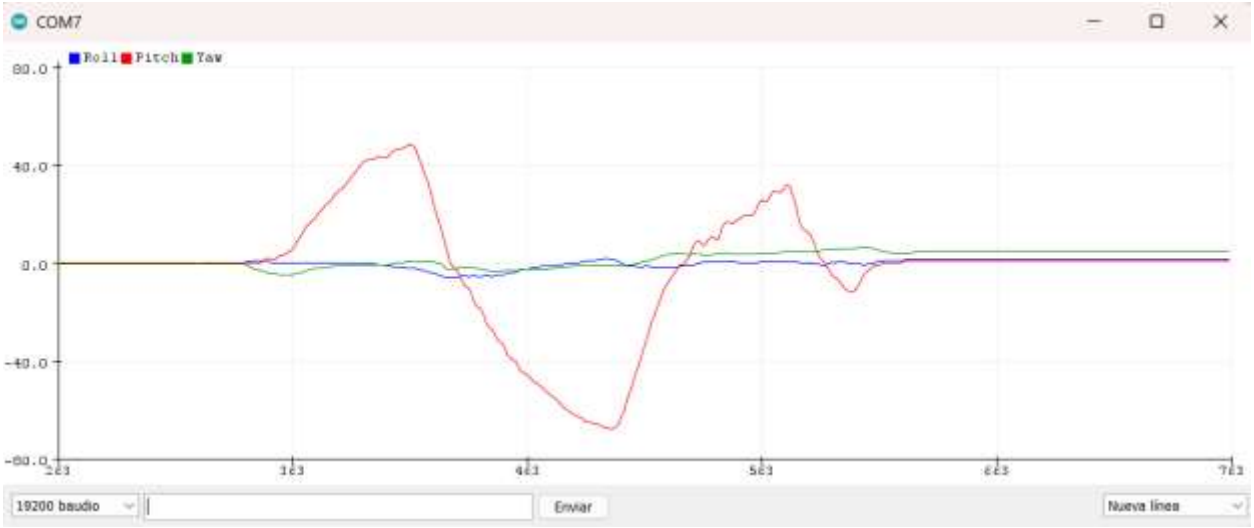
```
/*66.1.-El control PID es un algoritmo utilizado en el código de sistemas embebidos, su objetivo es lograr que alguna medición
en un robot o sistema se alcance y mantenga en un valor deseado o punto de referencia (setpoint) con una mínima desviación o
error; como lo puede ser una posición, distancia, temperatura, flujo, orientación, entre otros.
Para que un controlador PID pueda trabajar necesita contar con al menos un sensor, un actuador y un sistema embebido que realice
el análisis del sistema (microcontrolador, FPGA, Raspberry Pi, etc.).*/
//IMPORTACIÓN DE LA LIBRERÍA WIRE Y SERVO:
#include <Wire.h>   //Librería que habilita la comunicación I2C en los pines A4 (SDA) y A5 (SCL) del Arduino.
#include <Servo.h>  //Librería de control de servomotores y motores brushless.


//DECLARACIÓN DE LAS VARIABLES DEL SENSOR: ACELERÓMETRO GIROSCOPIO MPU6050
//Si el PIN A0 está conectado a GND o a nada, el address del MPU6050 será 0X68, pero si está conectado a 5V, será 0X69.
const int MPU = 0X68;                                          //Dirección I2C del SLAVE MPU6050 = 0X68.
//Datos crudos recabados del sensor MPU6050:
float AccX, AccY, AccZ;                                        //3 grados de libertad acelerómetro.
float GyroX, GyroY, GyroZ;                                     //3 grados de libertad giroscopio.
//Datos previos a la orientación resultante después de ejecutar las primeras operaciones matemáticas sobre los datos crudos:
float accAngleX, accAngleY, gyroAngleX, gyroAngleY, gyroAngleZ;   //Orientación: Dirección XY y ángulo de inclinación 3D.
float accErrorX, accErrorY, gyroErrorX, gyroErrorY, gyroErrorZ;   //Errores de orientación XY y ángulo de inclinación.
int pruebasError = 0;                                          //Número de pruebas realizadas para promediar el error.
//Resultados de orientación: Roll (giro eje x), Pitch (giro eje y) y Yaw (giro eje z).
float roll, pitch, yaw;
float elapsedTime, currentTime, previousTime;                  //Tiempos recabados por medio del método millis().

//DECLARACIÓN DE LOS OBJETOS Y LAS VARIABLES DEL ACTUADOR: MOTOR BRUSHLESS KV = 2300
/*Objeto de la clase Servo, este no recibe ningún parámetro, solamente sirve para poder usar los
métodos de la librería.*/
Servo motorBLDC;               //Objeto que permite utilizar la librería Servo para crear la señal PWM.
int PINPWM_BLDC = 3;           //Pin 3 = Cable de señal PWM conectada al PIN3~ digital del Arduino.
/*Cuando el valor bajo del duty cycle en la señal PWM dura 1000 µs = 1 ms, la velocidad del BLDC
está en 0 [rpm]. Esta es la teoría, pero si el motor no gira, este valor se debe calibrar para
encontrar el óptimo, además al editar esto se puede cambiar el alcance de la velocidad de giro
del motor.*/
int DUTYCYCLEMIN = 0;          //Calibración BLDC: High Duty Cycle Mínimo ≈ 0 µs = 0 s.
/*Cuando el valor alto del duty cycle en la señal PWM dura 2000 µs = 2 ms, la velocidad del BLDC
está en KV * V_ESC [rpm]. Esta es la teoría, pero si el motor deja de girar, este valor se debe
calibrar para encontrar el óptimo, además al editar esto se puede cambiar el alcance de la
velocidad de giro del motor.*/
int DUTYCYCLEMAX = 3000;       //Calibración BLDC: High Duty Cycle Máximo ≈ 3000 µs = 3 ms.
//Velocidad mínima alcanzada con la señal PWM generada con la librería Servo = 20; Velocidad máxima = 180.
float PWM_PID = 20;            //Ajuste en la señal PWM por medio del control PID, empieza en el mínimo que es de 20.

//DECLARACIÓN DE LAS VARIABLES PID: Error, Ecuación PID Proporcional, Integral, Derivativa y Constantes Kp, Ki y Kd.
float error, previous_error;   //Error actual y anterior.
/*La ecuación diferencial del PID se compone de tres partes, la proporcional, la integral y la derivativa, cada una de ellas
está encargada de eliminar el error en el sistema en un punto del tiempo diferente:
Proporcional = Presente; Integral = Pasado; Derivativo = Futuro.
Dicho error e(t) se obtiene al restar el punto deseado donde se busca que se encuentre el sistema menos el punto en el que
realmente se encuentra el sistema, dado por el sensor; el punto puede ser una distancia, ángulo, temperatura, orientación, etc.
La ecuación diferencial del controlador PID es:
PID = kp*e(t) + (PID_I + ki*e(t)) + kd*(e(t)/t) = PID_P + PID_I + PID_D
Donde: PID_P = kp * e(t);      PID_I = PID_I + ki * e(t);      PID_D = kd * (e(t)/t)
Se deben elegir distintos valores para las constantes kp, ki y kd, de la ecuación, al hacerlo se calibrará el controlador PID.*/
float PID_P = 0;               //Reinicio de componente proporcional de la ecuación PID.
float PID_I = 0;               //Reinicio de componente integral de la ecuación PID.
float PID_D = 0;               //Reinicio de componente derivativa de la ecuación PID.
//Constantes de calibración kp, ki y kd.
float kp = 3.55;               //Constante proporcional kp = Presente; Esta se empieza a calibrar con valores entre 1 y 10.
float ki= 0.005;               //Constante integral ki = Pasado; Esta se empieza a calibrar con valores entre 0 y 1.
float kd = 2.05;               //Constante derivativa kd = Futuro; Esta se empieza a calibrar con valores entre 1 y 10.
//Ecuación PID final = PID = kp*e(t) + (PID_I + ki*e(t)) + kd*((e(t)-e(t-1))/t) = PID_P + PID_I + PID_D.
float PID;
//Constante que busca alcanzar o mantener por medio del control PID.
float setPoint = 0;            //Ángulo pitch (alrededor del eje y) = 0°.


//CONFIGURACIÓN DE LOS PINES Y LA COMUNICACIÓN SERIAL:
void setup() {
  //CONFIGURACIÓN DEL SENSOR ACELERÓMETRO Y GIRÓSCOPO MPU6050 QUE OBTIENE LA ORIENTACIÓN DEL SISTEMA:
  /*Serial.begin(baudRate): Este método inicializa la comunicación serial entre la placa Arduino y la computadora, además de
  que configura su velocidad de transmisión dada en unidad de baudios (bit trasmitido por segundo) que recibe como su único
  parámetro:
      - En general, 9600 baudios es una velocidad de transmisión comúnmente utilizada y es compatible con la mayoría de los
        dispositivos y programas.
      - Si se necesita una transferencia de datos más rápida y el hardware/software lo admiten, se puede optar por velocidades
        más altas como 115200 o 57600 baudios, pero en comunicación I2C habilitada por la librería Wire, se pueden usar
        velocidades de 19,200 hasta 115,200 baudios.
  Es importante asegurarse de que la velocidad de transmisión especificada coincida con la velocidad de comunicación del otro
  dispositivo al que se conecta el Arduino. Si la velocidad de transmisión no coincide, los datos pueden no transmitirse o
  recibirse correctamente.*/
  Serial.begin(19200);         //Velocidad de transmisión serial I2C: 19,200 baudios.
  /*Wire.begin(): Método que inicializa la comunicación I2C, después de este se deberá indicar con qué dirección de SLAVE se
  está estableciendo la conexión y la velocidad del reloj.*/
  Wire.begin();                //Método que inicia la comunicación I2C.
  /*Wire.beginTransmission(dirección): Método que permite a un dispositivo mandar o recibir datos de una dirección SLAVE en
  específico. MASTER es un dispositivo que puede mandar datos y SLAVE es uno que los puede recibirlos mediante el protocolo
```

```
  I2C, pero para que un SLAVE pueda recibir datos se debe indicar su dirección, ya que un MASTER puede mandar datos a varios
  SLAVE a la vez.*/
  //CONFIGURACIÓN DE LA SEÑAL DE RELOJ Y MODO DE BAJO CONSUMO DEL MPU6050:
  Wire.beginTransmission(MPU);  //Método que indica a qué dirección de SLAVE se enviarán datos con el protocolo I2C.
  /*Wire.write(): Método que se debe usar dos veces, la primera vez permite indicar a qué registro del SLAVE se quiere acceder
  y la segunda vez permite mandar un byte de información a dicho registro del SLAVE, cuya dirección fue previamente
  inicializada con el método Wire.beginTransmission().
  El registro 6B = 107 del MPU6050 permite encender un modo de bajo consumo e indicar cuál será la fuente de la señal de reloj.
  Mandando el valor de 0X00 se indica que el modo de bajo consumo se apague y que la señal de reloj sea la interna del módulo
  de 8MHz.*/
  Wire.write(0x6B);             //Método que accede al registro 6B del SLAVE con dirección 0X68.
  Wire.write(0x00);             //Método que escribe el valor 00 en el registro 6B del SLAVE con dirección 0X68.
  Wire.endTransmission(true);   //Método que finaliza la transmisión de datos mandados a cualquier dirección de SLAVE.
  //CONFIGURACIÓN DE LA SENSIBILIDAD DE MEDICIÓN DE LA ACELERACIÓN EN EL MPU6050:
  Wire.beginTransmission(MPU);  //Método que indica a qué dirección de SLAVE se enviarán datos con el protocolo I2C.
  /*El registro 1C = 28 del MPU6050 permite indicar el rango de medición del acelerómetro, yendo desde ±2g hasta ±16g, se elije
  un rango mayor cuando el dispositivo al que se quiere incorporar el MPU6050 estará sometido a aceleraciones muy grandes o
  expuesto a fuerzas extremas, como lo puede ser en vehículos y aeronaves, monitoreo de impactos, robótica y sistemas de
  movimiento rápido, etc. Mandando el valor de 0X00 se indica que el rango de medición del acelerómetro sea el mínimo de ±2g.*/
  Wire.write(0x1C);             //Método que accede al registro 1C del SLAVE con dirección 0X68.
  Wire.write(0x00);             //Método que escribe el valor 00 en el registro 1C del SLAVE con dirección 0X68, obteniendo un
rango de ±2g.
  Wire.endTransmission(true);   //Método que finaliza la transmisión de datos mandados a cualquier dirección de SLAVE.
  //CONFIGURACIÓN DE LA SENSIBILIDAD DE MEDICIÓN DE LA ACELERACIÓN EN EL MPU6050:
  Wire.beginTransmission(MPU);  //Método que indica a qué dirección de SLAVE se enviarán datos con el protocolo I2C.
  /*El registro 1B = 27 del MPU6050 permite indicar el rango de medición del giroscopio, yendo desde ±250°/s hasta ±2000°/s, se
  elije un rango mayor cuando el dispositivo al que se quiere incorporar el MPU6050 estará sometido a aceleraciones muy grandes
  o expuesto a fuerzas extremas, como lo puede ser en vehículos y aeronaves, monitoreo de impactos, robótica, sistemas de
  movimiento rápido, etc. Mandando el valor de 0X00 se indica que el rango de medición del acelerómetro sea el mínimo de
±250°/s.*/
  Wire.write(0x1B);             //Método que accede al registro 1B del SLAVE con dirección 0X68.
  Wire.write(0x00);             //Escribe el valor 00 en el registro 1B del SLAVE con dirección 0X68, obteniendo un rango de
±250°/s.
  Wire.endTransmission(true);   //Método que finaliza la transmisión de datos mandados a cualquier dirección de SLAVE.
  //FUNCIÓN PROPIA DE ESTE CÓDIGO PARA CALCULAR EL ERROR DE LOS DATOS RECABADOS DEL ACELERÓMETRO GIROSCOPIO MPU6050:
  calcularErrorSensor();

  //CONFIGURACIÓN DEL ACTUADOR DE TIPO MOTOR BRUSHLESS QUE SE ACCIONA PARA ELIMINAE EL ERROR DEL SISTEMA:
  /*servomotor.attach(): Método que sirve para inicializar el objeto del servomotor:
    - El primer parámetro indica el Pin del Arduino al que se conectó el cable de la señal PWM.
    - El segundo parámetro indica la duración mínima en µs de la señal PWM y el tercero la máxima.*/
  motorBLDC.attach(PINPWM_BLDC, DUTYCYCLEMIN, DUTYCYCLEMAX);

  //CONFIGURACIÓN PARA EL CONTROLADOR PID:
  /*millis(): Método que devuelve el tiempo transcurrido en milisegundos, empezando a contar desde que se enciende la placa
  Arduino y no deteniéndose hasta que esta se apague o llegue a su límite, que es el mismo dado por el tipo de dato unsigned
  long: De 0 a 4,294,967,295 milisegundos = 1,193.0464 horas = 49.7102 días = 49 días y 17 horas.
  Cada que se utilice el método millis, se estará actualizando el tiempo guardado en una variable cualquiera, en este caso se
  guarda en dos variables para crear un temporizador que mida un tiempo de duración, realizando una simple resta entre dos
  medidas de tiempo.*/
  currentTime = millis();       //Guarda el tiempo transcurrido desde que se prendió el Arduino.

  /*delay(ms): Método que detiene la ejecución del programa un cierto tiempo dado en milisegundos.*/
  delay(500);
}


//EJECUCIÓN DEL PROGRAMA EN UN BUCLE INFINITO: Operaciones matemáticas que calculan el sentido de giro y orientación del sensor.
void loop() {
  //OBTENCIÓN DE DATOS DE ORIENTACIÓN DEL MPU6050 A TRAVÉS DE SU ACELERÓMETRO Y GIRÓSCOPO:
  //LECTURA DE LOS DATOS DEL ACELERÓMETRO:
  /*En los 6 registros 3B, 3C, 3D, 3E, 3F y 40, cuyo correspondiente decimal son los 59, 60, ..., 64, viene contenida la
  información de los acelerómetros correspondientes a los 3 ejes X, Y, Z. Estos vienen agrupados en dos registros diferentes
  porque la información completa es de 16 bits, pero cada registro como máximo puede transportar 8.*/
  Wire.beginTransmission(MPU);  //Método que indica a qué dirección de SLAVE se enviarán datos con el protocolo I2C.
  Wire.write(0x3B);             //Método que accede al registro 3B del SLAVE con dirección 0X68.
  Wire.endTransmission(false);  //Método que deja abierta la transmisión de datos mandados a cualquier dirección de SLAVE.
  /*Wire.requestFrom(address, quantity, stop): Método que permite solicitar datos a un dispositivo SLAVE conectado al bus I2C,
  para ello previamente se tuvieron que haber ejecutado los métodos: Wire.beginTransmission(address), Wire.write(registro) y
  Wire.endTransmission(false), ya que con ellos se indica a partir de qué registro de un SLAVE en específico se extraerá el
  número de datos con longitud de 8 bits (byte) indicado en este método.*/
  Wire.requestFrom(MPU, 6, true); //Método que extrae datos de 6 registros (2 por acelerómetro), desde el 3B hasta el 40 del
SLAVE con dirección 0X68.
  /*Wire.read(): Permite leer un dato de un SLAVE, para ello previamente se tuvo que haber ejecutado el método
  Wire.requestFrom(address, quantity, stop). Como la aceleración X, Y, Z se comparte a través de 6 registros, donde cada 2
  corresponden a una medición de 16 bits, se realiza la operación shift << y luego se aplica una compuerta OR | para primero
  recorrer 8 bits a la derecha el contenido del primer registro dentro de un número binario y luego unir eso con el contenido
  del segundo registro, obteniendo al final la información completa de 16 bits en una misma variable. Dependiendo del rango de
  medición del acelerómetro, yendo desde ±2g hasta ±16g, se deberá dividir el dato obtenido de cada una de las 3 aceleraciones
  entre cierto número, indicado en la página 29 del datasheet, bajo el nombre de LSB sensitibity. Para ±2g, LSB = 16,384.*/
  AccX = (Wire.read() << 8 | Wire.read())/16384.0;  //Aceleración lineal medida en el eje X.
  AccY = (Wire.read() << 8 | Wire.read())/16384.0;  //Aceleración lineal medida en el eje Y.
  AccZ = (Wire.read() << 8 | Wire.read())/16384.0;  //Aceleración lineal medida en el eje Z.

  //CALCULO DE LOS ÁNGULOS DE ROTACIÓN ALREDEDOR DEL EJE "X" (ROLLL), "Y" (PITCH): Se deben convertir a grados, porque
  //inicialmente vienen en radianes.
  //CORRECCIÓN DE LAS SALIDAS DEL ACELERÓMETRO CON EL ERROR CALCULADO POR MEDIO DE LA FUNCIÓN PROPIA calcularErrorSensor():
  //accErrorX obtenido de la función calcularErrorSensor() = -1.93
  //Inclinación alrededor del eje X = Roll  = arctan(Ay/√(Ax^2+Az^2))
  accAngleX = (atan((AccY)/sqrt(pow(AccX, 2) + pow(AccZ, 2))) * (180/PI)) - (-1.93);
  //accErrorY obtenido de la función calcularErrorSensor() = -1.88
  //Inclinación alrededor del eje Y = Pitch = arctan(Ax/√(Ay^2+Az^2))
  accAngleY = (atan(-1*(AccX)/sqrt(pow(AccY, 2) + pow(AccZ, 2))) * (180/PI)) - (-1.88);
```

```
  //LECTURA DE LOS DATOS DEL GIROSCOPIO:
  //Variable que guarda el tiempo de ejecución del loop anterior para medir intervalos de tiempo entre ejecuciones.
  previousTime = currentTime;
  //Actualización del tiempo transcurrido cada que se ejecute la función loop, para realizar la medición del intervalo de tiempo.
  currentTime = millis();
  elapsedTime = (currentTime - previousTime)/1000;  //Intervalo de tiempo medido en segundos.
  /*En los 6 registros 43, 44, 45, 46, 47 y 48, cuyo correspondiente decimal son los 67, 68, ..., 72, viene contenida la
  información de los giróscopos correspondientes a los 3 ejes X, Y, Z. Estos vienen agrupados en dos registros diferentes
  porque la información completa es de 16 bits, pero cada registro como máximo puede transportar 8.*/
  Wire.beginTransmission(MPU);  //Método que indica a qué dirección de SLAVE se enviarán datos con el protocolo I2C.
  Wire.write(0x43);             //Método que accede al registro 43 del SLAVE con dirección 0X68.
  Wire.endTransmission(false);  //Método que deja abierta la transmisión de datos mandados a cualquier dirección de SLAVE.
  /*Wire.requestFrom(address, quantity, stop): Método que permite solicitar datos a un dispositivo SLAVE conectado al bus I2C,
  para ello previamente se tuvieron que haber ejecutado los métodos: Wire.beginTransmission(address), Wire.write(registro) y
  Wire.endTransmission(false), ya que con ellos se indica a partir de qué registro de un SLAVE en específico se extraerá el
  número de datos con longitud de 8 bits (byte) indicado en este método.*/
  Wire.requestFrom(MPU, 6, true); //Método que extrae datos de 6 registros (2 por giróscopo), desde el 43 hasta el 48 del SLAVE
con dirección 0X68.
  /*Wire.read(): Permite leer un dato de un SLAVE, para ello previamente se tuvo que haber ejecutado el método
  Wire.requestFrom(address, quantity, stop). Como la aceleración angular X, Y, Z se comparte a través de 6 registros, donde
  cada 2 corresponden a una medición de 16 bits, se realiza la operación shift << y luego se aplica una compuerta OR | para
  primero recorrer 8 bits a la derecha el contenido del primer registro dentro de un número binario y luego unir eso con el
  contenido del segundo registro, obteniendo al final la información completa de 16 bits en una misma variable. Dependiendo del
  rango de medición del acelerómetro, yendo desde ±2g hasta ±16g, se deberá dividir el dato obtenido de cada una de las 3
  aceleraciones entre cierto número, indicado en la página 31 del datasheet, bajo el nombre de LSB sensitibity. Para ±250°/s,
  LSB sensitibity = 131.*/
  GyroX = (Wire.read() << 8 | Wire.read())/131.0;  //Aceleración angular medida en el eje X.
  GyroY = (Wire.read() << 8 | Wire.read())/131.0;  //Aceleración angular medida en el eje Y.
  GyroZ = (Wire.read() << 8 | Wire.read())/131.0;  //Aceleración angular medida en el eje Z.

  //CORRECCIÓN DE LAS SALIDAS DEL GIROSCOPIO CON EL ERROR CALCULADO POR MEDIO DE LA FUNCIÓN PROPIA calcularErrorSensor():
  GyroX = GyroX - (-2.09); //gyroErrorX = -2.09
  GyroY = GyroY - (-0.87); //gyroErrorY = -0.87
  GyroZ = GyroZ - (-1.29); //gyroErrorZ = -1.29

  //CÁLCULO DEL ÁNGULO DE ROTACIÓN XY AL MULTIPLICAR EL DATO DEL GIROSCOPIO °/S POR EL TIEMPO DE DURACIÓN DEL TEMPORIZADOR,
  //ESTO ES EL EQUIVALENTE A INTEGRAR EL VALOR DE LA VELOCIDAD ANGULAR PARA OBTENER EL ÁNGULO Y ADEMÁS SE OBTIENE EL VALOR DE
  //YAW:
  gyroAngleX = gyroAngleX + GyroX * elapsedTime;    //gyroAngleX = °/s * s = grados.
  gyroAngleY = gyroAngleY + GyroY * elapsedTime;    //gyroAngleX = °/s * s = grados.
  yaw = yaw + GyroZ * elapsedTime;                  //gyroAngleZ = yaw = °/s * s = grados.

  //EL ÁNGULO EN XY MEDIDO CON EL ACELERÓMETRO SE COMBINA CON EL ÁNGULO XY MEDIDO CON EL GIRÓSCOPO PARA OBTENER EL RESULTADO DE
  //ROLL Y PITCH, ESTE SE CONSIDERA COMO UN FILTRO DE LA SEÑAL:
  roll  = 0.96*gyroAngleX + 0.04*accAngleX;
  pitch = 0.96*gyroAngleY + 0.04*accAngleY;
  /*Serial.println(): Método que escribe un mensaje en la consola serial de Arduino, a la cual se puede acceder en la esquina
  superior derecha donde se encuentra una lupa, pero si en vez de usar el Monitor Serie, nos introducimos en la opción de
  Herramientas -> Serial Plotter, podremos ver tres gráficas que indican el cambio de cada magnitud.*/
  Serial.print("Roll:" + String(roll));
  Serial.print(String(", "));
  Serial.print("Pitch:" + String(pitch));
  Serial.print(String(", "));
  Serial.print("Yaw:" + String(yaw));
  Serial.print(String("\n"));


  //CONTROL PID: En este caso se busca que el ángulo de inclinación alrededor del eje y (osea el pitch) sea igual a cero, si no
  //es así, se activa el actuador (motor BLDC), el cual hará lo posible para corregir el error y alcanzar el valor del setpoint.
  /*El error e(t) del sistema se obtiene al restar el punto deseado donde se busca que se encuentre el sistema, ya sea un ángulo,
  distancia, temperatura, orientación, etc. menos el punto en el que realmente se encuentra el sistema, dado por el sensor.*/
  //error = ÁnguloAlrededorDelEje_y - ValorDeseado; La fórmula es al revés, pero en este caso se hace así porque setPoint = 0.
  error = pitch - setPoint;                         //Error actual.

  /*La ecuación diferencial del PID se compone de tres partes, la proporcional, la integral y la derivativa, cada una de ellas
  está encargada de eliminar el error en el sistema en un punto del tiempo diferente:
  Proporcional = Presente;      Donde: PID_P = kp * e(t).*/
  PID_P = kp*error;                                 //Control P: Proporcional.
  /*La ecuación diferencial del PID se compone de tres partes, la proporcional, la integral y la derivativa, cada una de ellas
  está encargada de eliminar el error en el sistema en un punto del tiempo diferente:
  Integral = Pasado;            Donde: PID_I = PID_I + ki * e(t).
  En este caso la parte integral se utiliza para detectar errores muy pequeños, cuando esto sea asi, la componente del control
  integral irá aumentando gradualmente su valor para contrarrestar el error con una fuerza acumulativa, en este caso el rango
  de acción del control integral es de ±3°.*/
  if(-3 < error < 3){
    PID_I = PID_I + (ki*error);                     //Control I: Integral.
  }
  /*La ecuación diferencial del PID se compone de tres partes, la proporcional, la integral y la derivativa, cada una de ellas
  está encargada de eliminar el error en el sistema en un punto del tiempo diferente:
  Derivativo = Futuro;          Donde: PID_D = kd * (e(t)/t)
  Como la parte derivativa depende del tiempo, al ser una derivada, lo que se hace es dividirla entre el lapso de tiempo
  Calculada para la medición del sensor MPU6050 y además se considera un error anterior para realizar de igual manera un
  intervalo del error.*/
  PID_D = kd*((error - previous_error) / elapsedTime);  //Control D: Derivativo.
  /*Por lo tanto, la ecuación que conforma a todo el controlador PID es la siguiente:
  PID = kp*e(t) + (PID_I + ki*e(t)) + kd*(e(t)/t) = PID_P + PID_I + PID_D
  De esta ecuación se deben elegir distintos valores para las constantes kp, ki y kd, al hacerlo se calibrará el controlador
  PID.*/
  PID = PID_P + PID_I + PID_D;                      //Ecuación PID completa.

  /*Ya habiendo declarado las ecuaciones PID se debe sumar o restar el resultado de su ecuación final al actuador para así
  generar la automatización del sistema, ya que dependiendo de a donde se mueva el drone, el ángulo del pitch será positivo o
  negativo, pero para ello se deben establecer límites, porque el BLDC solo acepta velocidades de 20 a 180 utilizando el método
  Servo.write() y empieza con el valor de 20. Por lo tanto, el controlador PID puede adoptar valores de -20 a 160.*/
  if(PID <= -20){               //Condición que limita el valor del controlador PID.
```

```
    PID = -20;
  }else if(PID >= 160){
    PID = 160;
  }
  /*Después de haber considerado el límite del valor en el control PID, este se suma al valor de la variable PWM_PID que moverá
  al motor, o también se podría restar dependiendo del sentido de giro y el lado del drone en el que se encuentre el motor BLDC.
  Recordemos que cuando esto se aplique a un drone, el valor de PID se sumaría a la señal PWM de un motor y se le restaría al del
  otro lado para que el motor se pueda equilibrar, ya que el ángulo de orientación pitch es positivo hacia un lado y negativo
  hacia el otro. Posteriormente se creará otra condición que limite el valor entregado hacia la señal PWM de 20 a 180, que son
  sus valores mínimos y máximos de velocidad.*/
  //Control PID aplicado a la señal PWM que activa un motor sin escobillas.
  PWM_PID = PWM_PID + PID;        //Control PID del actuador.
  //Límite de velocidad mínima y máxima entregada al motor BLDC.
  if(PWM_PID <= 20){              //Condición que limita el duty cycle de la señal PWM entregada al actuador.
    PWM_PID = 20;                 //Velocidad mínima = 20.
  }else if(PWM_PID >= 180){
    PWM_PID = 180;                //Velocidad máxima = 180.
  }
  /*Servo.write(): Método usado para controlar la velocidad de giro del eje perteneciente a
  un motor sin escobillas. Recibe como argumento un número que representa la velocidad deseada.*/
  motorBLDC.write(PWM_PID);       //Velocidad mínima = 20; Velocidad máxima = 180.

  //Actualización del error anterior, para que este se considere al calcular el control Derivativo.
  previous_error = error;
}



//FUNCIÓN PROPIA calcularErrorSensor(), PARA OBTENER UNA BUENA LECTURA DEL ERROR, EL SENSOR SE DEBE ENCONTRAR INICIALMENTE EN
//UNA ESTRUCTURA PLANA:
void calcularErrorSensor(){
  /*Para calcular el error se realizan 200 lecturas durante el setup del programa, se suma el resultado de todas ellas y luego
  se divide entre 200 para obtener así un promedio del error, este se imprimirá en pantalla y se deberá restar en el cálculo de
  la función loop() para así hacer la corrección de dicho error.*/
  //CÁLCULO DEL ERROR EN EL ACELERÓMETRO:
  while(pruebasError < 200){
    Wire.beginTransmission(MPU);  //Método que indica a qué dirección de SLAVE se enviarán datos con el protocolo I2C.
    Wire.write(0x3B);             //Método que accede al registro 3B del SLAVE con dirección 0X68.
    Wire.endTransmission(false);  //Método que deja abierta la transmisión de datos mandados a cualquier dirección de SLAVE.
    //Método que extrae datos de 6 registros (2 por acelerómetro), desde el 3B hasta el 40 del SLAVE con dirección 0X68.
    Wire.requestFrom(MPU, 6, true);
    AccX = (Wire.read() << 8 | Wire.read())/16384.0;  //Aceleración lineal medida en el eje X.
    AccY = (Wire.read() << 8 | Wire.read())/16384.0;  //Aceleración lineal medida en el eje Y.
    AccZ = (Wire.read() << 8 | Wire.read())/16384.0;  //Aceleración lineal medida en el eje Z.
    accErrorX = accErrorX + (atan((AccY)/sqrt(pow(AccX, 2) + pow(AccZ, 2))) * (180/PI));   //Actualización del error en el eje X.
    accErrorY = accErrorY + (atan(-1*(AccX)/sqrt(pow(AccY, 2) + pow(AccZ, 2))) * (180/PI));//Actualización del error en el eje Y.
    pruebasError++;               //Aumento de la variable pruebasError, para ejecutar el loop de 200 iteraciones.
  }
  //CÁLCULO DEL ERROR PROMEDIO EN EL ACELERÓMETRO:
  accErrorX = accErrorX / 200;
  accErrorY = accErrorY / 200;
  pruebasError = 0;               //Reinicio de la variable pruebasError
  //CÁLCULO DEL ERROR EN EL GIROSCOPIO:
  while(pruebasError < 200){
    Wire.beginTransmission(MPU);  //Método que indica a qué dirección de SLAVE se enviarán datos con el protocolo I2C.
    Wire.write(0x43);             //Método que accede al registro 43 del SLAVE con dirección 0X68.
    Wire.endTransmission(false);  //Método que deja abierta la transmisión de datos mandados a cualquier dirección de SLAVE.
    //Método que extrae datos de 6 registros (2 por acelerómetro), desde el 43 hasta el 48 del SLAVE con dirección 0X68.
    Wire.requestFrom(MPU, 6, true);
    GyroX = (Wire.read() << 8 | Wire.read());     //Aceleración angular medida en el eje X.
    GyroY = (Wire.read() << 8 | Wire.read());     //Aceleración angular medida en el eje Y.
    GyroZ = (Wire.read() << 8 | Wire.read());     //Aceleración angular medida en el eje Z.
    gyroErrorX = gyroErrorX + (GyroX / 131.0);    //Actualización del error en el eje X.
    gyroErrorY = gyroErrorY + (GyroY / 131.0);    //Actualización del error en el eje Y.
    gyroErrorZ = gyroErrorZ + (GyroZ / 131.0);    //Actualización del error en el eje Z.
    pruebasError++;               //Aumento de la variable pruebasError, para ejecutar el loop de 200 iteraciones.
  }
  //CÁLCULO DEL ERROR PROMEDIO EN EL GIRÓSCOPO:
  gyroErrorX = gyroErrorX / 200;
  gyroErrorY = gyroErrorY / 200;
  gyroErrorZ = gyroErrorZ / 200;
  pruebasError = 0;               //Reinicio de la variable pruebasError

  //IMPRESIÓN EN CONSOLA DEL PROMEDIO DE LOS ERRORES: Estps siempre se deberán restar, por lo que el signo obtenido aquí, será
  //invertido en el código.
  Serial.print("\naccErrorX: ");
  Serial.println(accErrorX);      //accErrorX calculado   = -1.93
  Serial.print("accErrorY: ");
  Serial.println(accErrorY);      //accErrorY calculado   = -1.88
  Serial.print("gyroErrorX: ");
  Serial.println(gyroErrorX);     //gyroErrorX calculado  = -2.09
  Serial.print("gyroErrorY: ");
  Serial.println(gyroErrorY);     //gyroErrorY calculado  = -0.87
  Serial.print("gyroErrorZ: ");
  Serial.println(gyroErrorZ);     //gyroErrorZ calculado  = -1.29
  /*delay(ms): Método que detiene la ejecución del programa un cierto tiempo dado en milisegundos. El valor de este específico
  delay se cambia para observar los resultados arrojados por la función y así poder restarlo en sus ecuaciones
  correspondientes.*/
  delay(100);
}
```





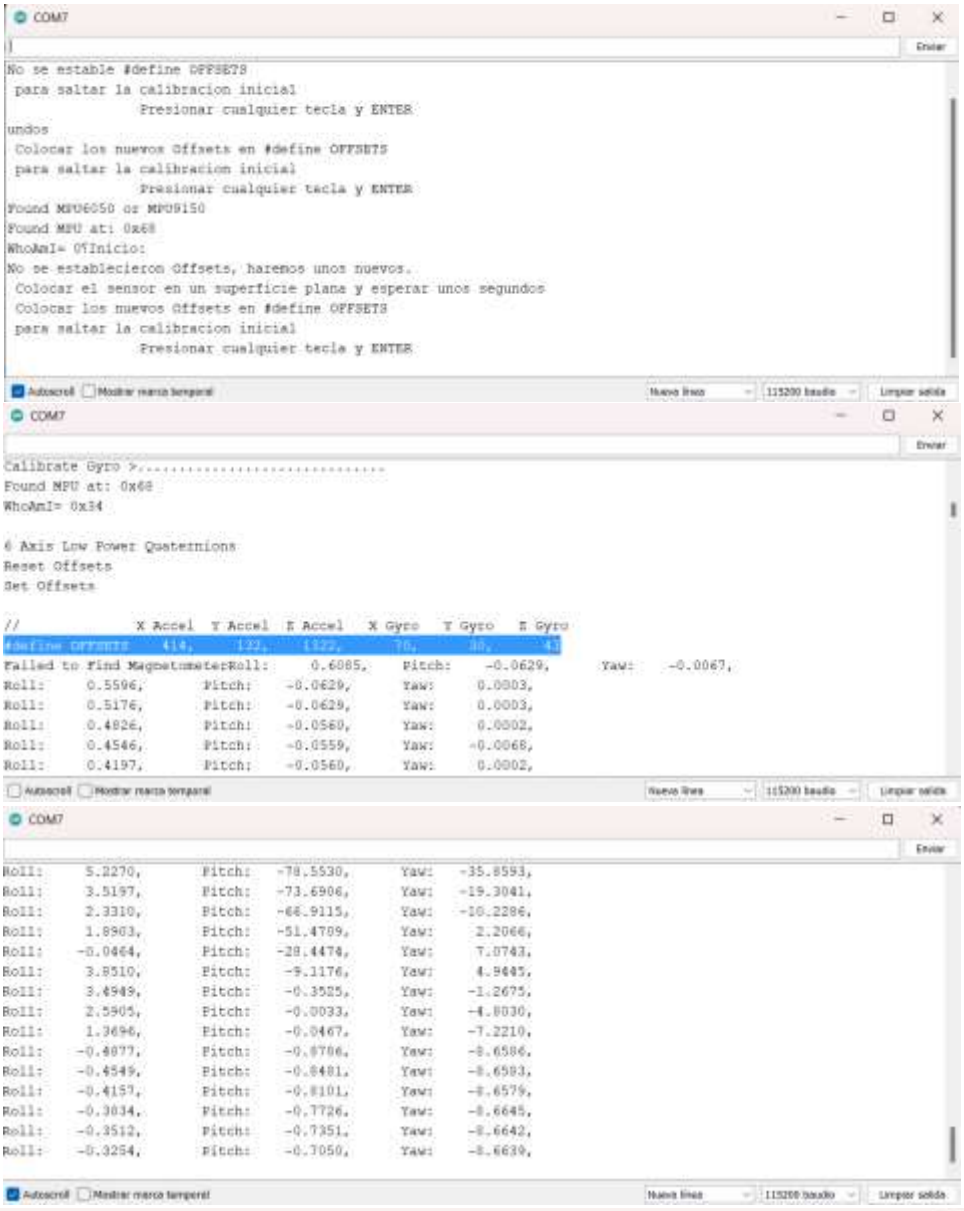
Control PID Drone: MPU6050 (**Librerías i2cdevlib** y **Simple_MPU6050**)/Actuador BLDC:

```
/*66.1.-El control PID es un algoritmo utilizado en el código de sistemas embebidos, su objetivo es lograr que alguna medición
en un robot o sistema se alcance y mantenga en un valor deseado o punto de referencia (setpoint) con una mínima desviación o
error; como lo puede ser una posición, distancia, temperatura, flujo, orientación, entre otros.
Para que un controlador PID pueda trabajar necesita contar con al menos un sensor, un actuador y un sistema embebido que realice
el análisis del sistema (microcontrolador, FPGA, Raspberry Pi, etc.). Con este código se utilizan las librerías i2cdevlib para
establecer la comunicación I2C y la librería Simple_MPU6050 para obtener los datos de orientación del sensor.*/
//IMPORTACIÓN DE LA LIBRERÍA Simple_MPU6050, QUE UTILIZA LAS LIBRERÍAS i2cdevlib Y Simple_Wire:
#include "Simple_MPU6050.h" //Librería que habilita la comunicación I2C en los pines A4 (SDA), A5 (SCL) y Pin2 (INT) del Arduino.
#include <Servo.h>          //Librería de control de servomotores y motores brushless.


//DECLARACIÓN DE LAS VARIABLES, OBJETOS Y CONSTANTES DEL SENSOR: ACELERÓMETRO GIROSCOPIO MPU6050
//Si el PIN A0 está conectado a GND o a nada, el address del MPU6050 será 0X68, pero si está conectado a 5V, será 0X69.
#define MPU6050_ADDRESS_A0_LOW   0X68                    //Adress del MPU6050 si el Pin A0 está conectado a GND.
#define MPU6050_ADDRESS_A0_HIGH  0X69                    //Adress del MPU6050 si el Pin A0 está conectado a 5V.
#define MPU6050_DEFAULT_ADDRESS   MPU6050_ADDRESS_A0_LOW  //Dirección I2C del SLAVE MPU6050 = 0X68.
/*La calibración se realiza de manera automática por el DMP incluido en el módulo MPU6050, pero si por alguna razón esos valores
calculados son erróneos, también se pueden incluir de forma manual. Cabe mencionar que los datos calculados automáticamente por
el sensor, siempre se muestran hasta el principio de la lectura de valores en consola y si estos no se incluyen en el código,
al ejecutar el programa aparecerá un mensaje en pantalla que no lo dejará correr hasta que introduzcamos cualquier letra y demos
```

```
clic en enter, al agregar la constante OFFSETS esto se deja de mostrar.*/
#define OFFSETS    388,     112,    1320,      68,      28,      40 //Valores personalizados de calibración para el sensor.
/*Objeto de la clase Simple_MPU6050 que permite establecer la comunicación I2C con el sensor MPU6050, obteniendo los
datos de aceleración lineal y rotativa para realizar cálculos con ellos y obtener la orientación del dispositivo.*/
Simple_MPU6050 mpu;
float elapsedTime, currentTime, previousTime;             //Tiempos recabados por medio del método millis().
//Número de muestras saltadas para evitar detectar los errores del proceso de calibración del DMP (Digital Motion Processor).
int muestrasConError = 300;                               //Muestras con error causadas por la calibracion del DMP.

//DECLARACIÓN DE LOS OBJETOS Y LAS VARIABLES DEL ACTUADOR: MOTOR BRUSHLESS KV = 2300
/*Objeto de la clase Servo, este no recibe ningún parámetro, solamente sirve para poder usar los
métodos de la librería.*/
Servo motorBLDC;               //Objeto que permite utilizar la librería Servo para crear la señal PWM.
int PINPWM_BLDC = 3;           //Pin 3 = Cable de señal PWM conectada al PIN3~ digital del Arduino.
/*Cuando el valor bajo del duty cycle en la señal PWM dura 1000 µs = 1 ms, la velocidad del BLDC
está en 0 [rpm]. Esta es la teoría, pero si el motor no gira, este valor se debe calibrar para
encontrar el óptimo, además al editar esto se puede cambiar el alcance de la velocidad de giro
del motor.*/
int DUTYCYCLEMIN = 0;          //Calibración BLDC: High Duty Cycle Mínimo ≈ 0 µs = 0 s.
/*Cuando el valor alto del duty cycle en la señal PWM dura 2000 µs = 2 ms, la velocidad del BLDC
está en KV * V_ESC [rpm]. Esta es la teoría, pero si el motor deja de girar, este valor se debe
calibrar para encontrar el óptimo, además al editar esto se puede cambiar el alcance de la
velocidad de giro del motor.*/
int DUTYCYCLEMAX = 3000;       //Calibración BLDC: High Duty Cycle Máximo ≈ 3000 µs = 3 ms.
//Velocidad mínima alcanzada con la señal PWM generada con la librería Servo = 20; Velocidad máxima = 180.
float PWM_PID = 20;            //Ajuste en la señal PWM por medio del control PID, empieza en el mínimo que es de 20.

//DECLARACIÓN DE LAS VARIABLES PID: Error, Ecuación PID Proporcional, Integral, Derivativa y Constantes Kp, Ki y Kd.
float error, previous_error;   //Error actual y anterior.
/*La ecuación diferencial del PID se compone de tres partes, la proporcional, la integral y la derivativa, cada una de ellas
está encargada de eliminar el error en el sistema en un punto del tiempo diferente:
Proporcional = Presente; Integral = Pasado; Derivativo = Futuro.
Dicho error e(t) se obtiene al restar el punto deseado donde se busca que se encuentre el sistema menos el punto en el que
realmente se encuentra el sistema, dado por el sensor; el punto puede ser una distancia, ángulo, temperatura, orientación, etc.
La ecuación diferencial del controlador PID es:
PID = kp*e(t) + (PID_I + ki*e(t)) + kd*(e(t)/t) = PID_P + PID_I + PID_D
Donde: PID_P = kp * e(t);      PID_I = PID_I + ki * e(t);      PID_D = kd * (e(t)/t)
Se deben elegir distintos valores para las constantes kp, ki y kd, de la ecuación, al hacerlo se calibrará el controlador PID.*/
float PID_P = 0;               //Reinicio de componente proporcional de la ecuación PID.
float PID_I = 0;               //Reinicio de componente integral de la ecuación PID.
float PID_D = 0;               //Reinicio de componente derivativa de la ecuación PID.
//Constantes de calibración kp, ki y kd.
float kp = 3.55;               //Constante proporcional kp = Presente; Esta se empieza a calibrar con valores entre 1 y 10.
float ki= 0.005;               //Constante integral ki = Pasado; Esta se empieza a calibrar con valores entre 0 y 1.
float kd = 2.05;               //Constante derivativa kd = Futuro; Esta se empieza a calibrar con valores entre 1 y 10.
//Ecuación PID final = PID = kp*e(t) + (PID_I + ki*e(t)) + kd*((e(t)-e(t-1))/t) = PID_P + PID_I + PID_D.
float PID;
//Constante que busca alcanzar o mantener por medio del control PID.
float setPoint = 0;            //Ángulo pitch (alrededor del eje y) = 0°.



//FUNCIONES PROPIAS DECLARADAS EN UNA LÍNEA:
/*La sintaxis utilizada en C++, que es el lenguaje que se usa para programar los microcontroladores de Arduino, es la sig:
#define   nombreFunción(parámetros)   Acción a ejecutar*/
/*spamtimer(): Función propia de 1 sola línea que crea un delay para evitar problemas al mostrar los datos de orientación
en consola.*/
#define spamtimer(t) for (static uint32_t SpamTimer; (uint32_t) (millis() - SpamTimer) >= (t); SpamTimer = millis())
/*printfloatx(): Función propia de 1 sola línea que proporciona una forma personalizada de imprimir datos en consola, para
ello recibe 5 parámetros:
  Texto que muestra el nombre de la variable,
  variable,
  Número de caracteres que muestran el valor de la variable,
  número de decimales mostrados,
  texto puesto al final.
Esto es muy útil al mostrar los resultados de orientación Roll, Pitch y Yaw obtenidos del sensor MPU6050.*/
#define printfloatx(Name,Variable,Spaces,Precision,EndTxt) print(Name); {char S[(Spaces + Precision + 3)];Serial.print(F(" "));
Serial.print(dtostrf((float)Variable,Spaces,Precision ,S));}Serial.print(EndTxt);

//CONFIGURACIÓN DE LOS PINES Y LA COMUNICACIÓN SERIAL:
void setup() {
  //CONFIGURACIÓN DEL SENSOR ACELERÓMETRO Y GIRÓSCOPO MPU6050 QUE OBTIENE LA ORIENTACIÓN DEL SISTEMA:
  /*uint8_t: Número binario sin signo que puede adoptar valores de 0 a 255. La presencia de la letra "t" del final significa
  "tipo" y se utiliza para hacer que la variable sea portátil y consistente en diferentes plataformas.*/
  uint8_t val;
  /*Condicional que evalúa a través de un condicional if cuál herramienta de la librería i2cdevlib se utilizará para implementar
  la comunicación del protocolo I2C, ya sea la proporcionada por la biblioteca Wire de Arduino (I2CDEV_ARDUINO_WIRE) o una
  implementación optimizada llamada Fastwire (I2CDEV_BUILTIN_FASTWIRE). En ambas opciones de inicialización se elige una señal
  de reloj de 400,000 Hz = 400 kHz. Se utiliza la sintaxis de directiva #if para el condicional porque en esta parte se está
  analizando una condición de la configuración del programa, por lo que se ejecuta en el momento de la compilación.*/
  #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    /*Wire.begin(): Método que inicializa la comunicación I2C, después de este se deberá indicar con qué dirección de SLAVE se
    está estableciendo la conexión y la velocidad del reloj.*/
    Wire.begin();              //Método que inicia la comunicación I2C con la librería Wire.
    /*Wire.setClock(): Método que configura la velocidad de transmisión (frecuencia del reloj) del bus I2C en Arduino.*/
    Wire.setClock(400000);     //Reloj de 400,000 Hz = 400 kHz = 2.5 µs.
  #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
    Fastwire::setup(400, true); //Método que inicia la comunicación I2C con la librería Fastwire.
  #endif
  /*Serial.begin(baudRate): Este método inicializa la comunicación serial entre la placa Arduino y la computadora, además de
  que configura su velocidad de transmisión dada en unidad de baudios (bit trasmitido por segundo) que recibe como su único
  parámetro:
      - En general, 9600 baudios es una velocidad de transmisión comúnmente utilizada y es compatible con la mayoría de los
        dispositivos y programas.
      - Si se necesita una transferencia de datos más rápida y el hardware/software lo admiten, se puede optar por velocidades
```

```
      más altas como 115200 o 57600 baudios, pero en comunicación I2C habilitada por la librería Wire, se pueden usar
      velocidades de 19,200 hasta 115,200 baudios.
  Es importante asegurarse de que la velocidad de transmisión especificada coincida con la velocidad de comunicación del otro
  dispositivo al que se conecta el Arduino. Si la velocidad de transmisión no coincide, los datos pueden no transmitirse o
  recibirse correctamente.*/
  Serial.begin(115200);  //Baudios = 115,200
  while(!Serial);
  /*Serial.println(): Método que escribe un mensaje en la consola serial de Arduino, a la cual se puede acceder en la esquina
  superior derecha donde se encuentra una lupa, pero si en vez de usar el Monitor Serie, nos introducimos en la opción de:
  Herramientas -> Serial Plotter, podremos ver tres gráficas que indican el cambio de cada magnitud. La función F(), permite que
  el texto se almacene en la memoria de programa, lo que ayuda a ahorrar espacio en la memoria RAM.*/
  Serial.println(F("Inicio: "));
  /*Condicional que evalúa si se han declarado valores de calibración para el sensor en la directiva OFFSETS o no, luego pasa a
  indicar el address SLAVE del MPU6050, calibrar su rango de medición y cargar dichos datos al DMP (Digital Motion Processor)
  para que se encargue de realizar los cálculos de orientación.*/
  #ifdef OFFSETS
    Serial.println(F("Usando offsets predefinidos"));
    /*Simple_MPU6050.SetAddress(): Este método de la librería Simple_MPU6050 lo que hace es establecer la dirección SLAVE del
    sensor MPU6050 para poder efectuar una comunicación I2C entre el Arduino y él.
    Simple_MPU6050.CalibrateMPU(): Este método de la librería Simple_MPU6050 lo que hace es indicar el rango de medición del
    acelerómetro y giroscopio del MPU6050 para ajustar la precisión de sus lecturas al uso que se le vaya a dar.
    Simple_MPU6050.load_DMP_Image(): Este método carga una imagen (firmware) en el sensor MPU6050, utilizando el DMP (Digital
    Motion Processor), el cual es un procesador dentro del sensor que puede realizar cálculos y filtrar datos de forma autónoma,
    lo que simplifica el procesamiento en el microcontrolador principal y corrige el error acumulativo en la lectura del Yaw que
    se tenía al utilizar la librería Wire. Cargar una imagen DMP en el sensor permite aprovechar estas capacidades.
    OFFSETS es una directiva declarada en este programa que permite calibrar el sensor, */
    mpu.SetAddress(MPU6050_ADDRESS_AD0_LOW).load_DMP_Image(OFFSETS);
  #else
    Serial.println(F("No se establecieron Offsets, haremos unos nuevos.\n"  // muestra texto estatico
                   " Colocar el sensor en un superficie plana y esperar unos segundos\n"
                   " Colocar los nuevos Offsets en #define OFFSETS\n"
                   " para saltar la calibracion inicial \n"
                   " \t\tPresionar cualquier tecla y ENTER"));
    while(Serial.available() && Serial.read());
    while(!Serial.available());
    while(Serial.available() && Serial.read());
    /*Simple_MPU6050.SetAddress().CalibrateMPU().load_DMP_Image(): Este método combinado permite indicar la dirección SLAVE del
    MPU, calibrar su rango de medición dependiendo de su aplicación y cargar una imagen al DMP (Digital Motion Processor) que
    corrija el error acumulativo en la lectura del Yaw que se tenía al usar la librería Wire.*/
    mpu.SetAddress(MPU6050_ADDRESS_AD0_LOW).CalibrateMPU().load_DMP_Image();
  #endif
  /*Simple_MPU6050.on_FIFO(): Este método se utiliza para habilitar la lectura continua de datos desde el FIFO del sensor
  MPU6050. FIFO es una pequeña memoria introducida dentro del sensor, cuyas siglas significan First Input First Output y cuando
  está llena de información es porque el DMP (Digital Motion Processor) ha terminado de realizar el cálculo de los datos de
  orientación: Roll, Pitch y Yaw, por lo que están listos para mostrarse en consola, por eso recibe como parámetro la función
  mostrar_valores().*/
  mpu.on_FIFO(mostrar_valores);

  //CONFIGURACIÓN DEL ACTUADOR DE TIPO MOTOR BRUSHLESS QUE SE ACCIONA PARA ELIMINAE EL ERROR DEL SISTEMA:
  /*servomotor.attach(): Método que sirve para inicializar el objeto del servomotor:
    - El primer parámetro indica el Pin del Arduino al que se conectó el cable de la señal PWM.
    - El segundo parámetro indica la duración mínima en µs de la señal PWM y el tercero la máxima.*/
  motorBLDC.attach(PINPWM_BLDC, DUTYCYCLEMIN, DUTYCYCLEMAX);

  //CONFIGURACIÓN PARA EL CONTROLADOR PID:
  /*millis(): Método que devuelve el tiempo transcurrido en milisegundos, empezando a contar desde que se enciende la placa
  Arduino y no deteniéndose hasta que esta se apague o llegue a su límite, que es el mismo dado por el tipo de dato unsigned
  long: De 0 a 4,294,967,295 milisegundos = 1,193.0464 horas = 49.7102 días = 49 días y 17 horas.
  Cada que se utilice el método millis, se estará actualizando el tiempo guardado en una variable cualquiera, en este caso se
  guarda en dos variables para crear un temporizador que mida un tiempo de duración, realizando una simple resta entre dos
  medidas de tiempo.*/
  currentTime = millis();       //Guarda el tiempo transcurrido desde que se prendió el Arduino.
  /*delay(ms): Método que detiene la ejecución del programa un cierto tiempo dado en milisegundos.*/
  delay(20);
}

//EJECUCIÓN DEL PROGRAMA EN UN BUCLE INFINITO: Interrupciones que identifican el sentido de giro y orientación del sensor
MPU6050.
void loop() {
  /*Simple_MPU6050.dmp_read_fifo():  Este método se utiliza para realizar la lectura de datos que se encuentran dentro de la
  memoria FIFO en el sensor MPU6050.*/
  mpu.dmp_read_fifo();

  /*Servo.write(): Método usado para controlar la velocidad de giro del eje perteneciente a
  un motor sin escobillas. Recibe como argumento un número que representa la velocidad deseada.*/
  motorBLDC.write(PWM_PID);       //Velocidad mínima = 20; Velocidad máxima = 180.
}

//FUNCIÓN PROPIA DECLARADA EN VARIAS LINEAS:
/*mostrar_valores(): Esta función propia es llamada cada que se ejecute la interrupción conectada al Pin digital 2.*/
void mostrar_valores(int16_t*gyro, int16_t*accel, int32_t*quat, int32_t*timestamp){
  uint8_t SpamDelay = 100;              //Delay aplicado en ms cada vez que se muestran los valores de orientación en consola.
  /*Quaternion: Es un tipo de dato que representa una rotación en el espacio tridimensional, el cual almacena la orientación
  calculada por el sensor.*/
  Quaternion q;
  /*VectorFloat: Es un tipo de dato que representa un vector de tres dimensiones con valores decimales de punto flotante que se
  utilizan para almacenar la información de la gravedad calculada por el sensor.*/
  VectorFloat gravity;
  float ypr[3] = {0, 0, 0};             //Vector que almacena los resultados de la orientación: Yaw, Pitch y Roll (ypr).
  float xyz[3] = {0, 0, 0};             //Vector que considera los 3 ejes coordenados tridimensionales XYZ.
  /*spamtimer(SpamDelay): Función propia que se ejecuta cada 100 milisegundos (indicado por la variable SpamDelay), la cual
  realiza los cálculos del ángulo de Euler 3D para obtener los resultados de orientación: Yaw, Pitch y Roll en unidad de grados.
  Además dentro se incluye el control PID.*/
  spamtimer(SpamDelay){
    mpu.GetQuaternion(&q, quat);        //Representa la orientación del sensor en términos de una rotación tridimensional.
```

```
    mpu.GetGravity(&gravity, &q);          //Cálculo de la gravedad en términos de un vector tridimensional.
    mpu.GetYawPitchRoll(ypr, &q, &gravity);//Cálculo para la obtención de los resultados de orientación: Yaw, Pitch y Roll (ypr).
    mpu.ConvertToDegrees(ypr, xyz);     //Conversión de radianes a grados, ya que el resultado del cálculo anterior se da en rad.
    /*Serial.printfloatx(): Método que permite utilizar el formato de la función propia printfloatx() para imprimir en consola
    los datos recabados del sensor MPU6050 de forma personalizada, recibiendo como parámetros: El texto que muestra antes del
    valor de la variable, el valor de la variable, número de caracteres que muestran el valor de la variable, número de decimales
    mostrados y el texto que aparece al final.*/
    Serial.printfloatx(F("Roll: "),   xyz[2], 9, 4, F(",\t"));  //Giro alrededor del eje X = Roll.
    Serial.printfloatx(F("Pitch: "),  xyz[1], 9, 4, F(",\t"));  //Giro alrededor del eje Y = Pitch.
    Serial.printfloatx(F("Yaw: "),    xyz[0], 9, 4, F(",\t"));  //Giro alrededor del eje Z = Yaw.
    Serial.println();
    //OPERACIONES CONDICIONALES UTILIZANDO LOS VALORES DE ORIENTACIÓN OBTENIDOS POR EL SENSOR:
    if(xyz[2] >= 90){
      Serial.println("Roll = 90°;   Rotación alrededor del eje X.");
    }else if(xyz[1] >= 90){
      Serial.println("Pitch = 90°;  Rotación alrededor del eje Y.");
    }else if(xyz[0] >= 90){
      Serial.println("Yaw   = 90°;  Rotación alrededor del eje Z.");
    }


    /*CONTROL PID: En este caso se busca que el ángulo de inclinación alrededor del eje y (osea el pitch) sea igual a cero, si no
    es así, se activa el actuador (motor BLDC), el cual hará lo posible para corregir el error y alcanzar el valor del
    setpoint.*/
    //Variable que guarda el tiempo de ejecución del loop anterior para medir intervalos de tiempo entre ejecuciones.
    previousTime = currentTime;
    //Actualización del tiempo transcurrido cada que se ejecute la función loop, para realizar la medición del intervalo de
    //tiempo.
    currentTime = millis();
    elapsedTime = (currentTime - previousTime)/1000;      //Intervalo de tiempo medido en segundos.
    /*El error e(t) del sistema se obtiene al restar el punto deseado donde se busca que se encuentre el sistema, ya sea un
    ángulo, distancia, temperatura, orientación, etc. menos el punto en el que realmente se encuentra el sistema, dado por el
    sensor.*/
    //error = ÁnguloAlrededorDelEje_y - ValorDeseado; La fórmula es al revés, pero en este caso se hace así porque setPoint = 0.
    error = xyz[1] - setPoint;                            //Error actual = Pitch != 0°;  Rotación alrededor del eje Y.
    /*La ecuación diferencial del PID se compone de tres partes, la proporcional, la integral y la derivativa, cada una de ellas
    está encargada de eliminar el error en el sistema en un punto del tiempo diferente:
    Proporcional = Presente;      Donde: PID_P = kp * e(t).*/
    PID_P = kp*error;                                     //Control P: Proporcional.
    /*La ecuación diferencial del PID se compone de tres partes, la proporcional, la integral y la derivativa, cada una de ellas
    está encargada de eliminar el error en el sistema en un punto del tiempo diferente:
    Integral = Pasado;            Donde: PID_I = PID_I + ki * e(t).
    En este caso la parte integral se utiliza para detectar errores muy pequeños, cuando esto sea asi, la componente del control
    integral irá aumentando gradualmente su valor para contrarrestar el error con una fuerza acumulativa, en este caso el rango
    de acción del control integral es de ±3°.*/
    if(-3 < error < 3){
      PID_I = PID_I + (ki*error);                         //Control I: Integral.
    }
    /*La ecuación diferencial del PID se compone de tres partes, la proporcional, la integral y la derivativa, cada una de ellas
    está encargada de eliminar el error en el sistema en un punto del tiempo diferente:
    Derivativo = Futuro;          Donde: PID_D = kd * (e(t)/t)
    Como la parte derivativa depende del tiempo, al ser una derivada, lo que se hace es dividirla entre el lapso de tiempo
    calculada para la medición del sensor MPU6050 y además se considera un error anterior para realizar de igual manera un
    intervalo del error.*/
    PID_D = kd*((error - previous_error) / elapsedTime);  //Control D: Derivativo.
    /*Por lo tanto, la ecuación que conforma a todo el controlador PID es la siguiente:
    PID = kp*e(t) + (PID_I + ki*e(t)) + kd*(e(t)/t) = PID_P + PID_I + PID_D
    De esta ecuación se deben elegir distintos valores para las constantes kp, ki y kd, al hacerlo se calibrará el controlador
    PID.*/
    PID = PID_P + PID_I + PID_D;                          //Ecuación PID completa.

    /*Ya habiendo declarado las ecuaciones PID se debe sumar o restar el resultado de su ecuación final al actuador para así
    generar la automatización del sistema, ya que dependiendo de a donde se mueva el drone, el ángulo del pitch será positivo o
    negativo, pero para ello se deben establecer límites, porque el BLDC solo acepta velocidades de 20 a 180 utilizando el método
    Servo.write() y empieza con el valor de 20. Por lo tanto, el controlador PID puede adoptar valores de -20 a 160.*/
    if(PID <= -20){                 //Condición que limita el valor del controlador PID.
      PID = -20;
    }else if(PID >= 160){
      PID = 160;
    }

    /*Después de haber considerado el límite del valor en el control PID, este se suma al valor de la variable PWM_PID que moverá
    al motor, o también se podría restar dependiendo del sentido de giro y el lado del drone en el que se encuentre el motor
    BLDC. Recordemos que cuando esto se aplique a un drone, el valor de PID se sumaría a la señal PWM de un motor y se le
    restaría al del otro lado para que el motor se pueda equilibrar, ya que el ángulo de orientación pitch es positivo hacia un
    lado y negativo hacia el otro. Posteriormente se creará otra condición que limite el valor entregado hacia la señal PWM de 20
    a 180, que son sus valores mínimos y máximos de velocidad.*/
    //Control PID aplicado a la señal PWM que activa un motor sin escobillas.
    PWM_PID = PWM_PID + PID;        //Control PID del actuador.
    //Límite de velocidad mínima y máxima entregada al motor BLDC.
    if(PWM_PID <= 20){              //Condición que limita el duty cycle de la señal PWM entregada al actuador.
      PWM_PID = 20;                 //Velocidad mínima = 20.
    }else if(PWM_PID >= 180){
      PWM_PID = 180;                //Velocidad máxima = 180.
    }

    //Actualización del error anterior, para que este se considere al calcular el control Derivativo.
    previous_error = error;
  }
}
```

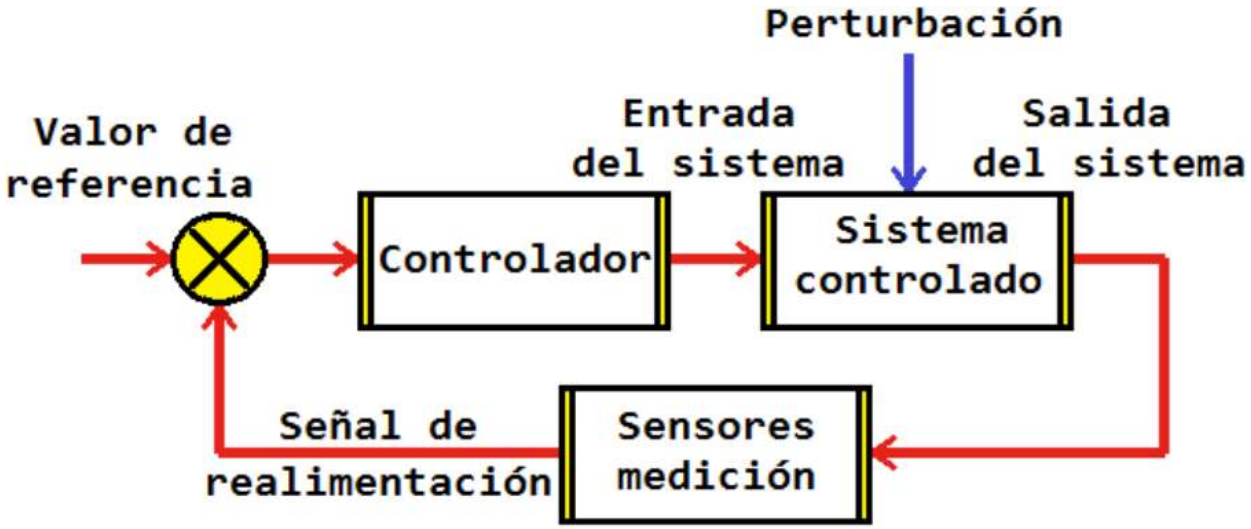

COM7
No se estable #define OFFSETS
para saltar la calibracion inicial
Presionar cualquier tecla y ENTER
undos
Colocar los nuevos Offsets en #define OFFSETS
para saltar la calibracion inicial
Presionar cualquier tecla y ENTER
Found MPU6050 or MPU9150
Found MPU at: 0x68
WhoAmI= 0fInicio:
No se establecieron Offsets, haremos unos nuevos.
Colocar el sensor en un superficie plana y esperar unos segundos
Colocar los nuevos Offsets en #define OFFSETS
para saltar la calibracion inicial
Presionar cualquier tecla y ENTER
Autoscroll
Mostrar marca temporal
Nueva línea
115200 baudio
Limpiar salida
COM7
Enviar
Calibrate Gyro >.....
Found MPU at: 0x68
WhoAmI= 0x34
6 Axis Low Power Quaternions
Reset Offsets
Set Offsets
// X Accel Y Accel Z Accel X Gyro Y Gyro Z Gyro
#define OFFSETS 414, 122, 1122, 76, 30, 43
Failed to find MagnetometerRoll: 0.6085, Pitch: -0.0629, Yaw: -0.0067,
Roll: 0.5596, Pitch: -0.0629, Yaw: 0.0003,
Roll: 0.5176, Pitch: -0.0629, Yaw: 0.0003,
Roll: 0.4826, Pitch: -0.0560, Yaw: 0.0002,
Roll: 0.4546, Pitch: -0.0559, Yaw: -0.0068,
Roll: 0.4197, Pitch: -0.0560, Yaw: 0.0002,
COM7
Enviar
Roll: 5.2270, Pitch: -78.5530, Yaw: -35.8593,
Roll: 3.5197, Pitch: -73.6906, Yaw: -19.3041,
Roll: 2.3310, Pitch: -68.9115, Yaw: -10.2286,
Roll: 1.8903, Pitch: -51.4709, Yaw: 2.2066,
Roll: -0.0464, Pitch: -28.4474, Yaw: 7.0743,
Roll: 3.8510, Pitch: -9.1176, Yaw: 4.9445,
Roll: 3.4949, Pitch: -0.3525, Yaw: -1.2675,
Roll: 2.5905, Pitch: -0.0033, Yaw: -4.8030,
Roll: 1.3696, Pitch: -0.0467, Yaw: -7.2210,
Roll: -0.4077, Pitch: -0.0706, Yaw: -8.6586,
Roll: -0.4549, Pitch: -0.0481, Yaw: -8.6593,
Roll: -0.4157, Pitch: -0.0101, Yaw: -8.6579,
Roll: -0.3034, Pitch: -0.7726, Yaw: -8.6645,
Roll: -0.3512, Pitch: -0.7351, Yaw: -8.6642,
Roll: -0.3254, Pitch: -0.7050, Yaw: -8.6639,
Autoscroll
Mostrar marca temporal
Nueva línea
115200 baudio
Limpiar salida


COM7
Roll
Pitch
Rotación
Yaw
del
eje
X. Gyro
Y Gyro
Z Gyro
1500.0
1000.0
500.0
0.0
-500.0
40
140
240
340
440
540
115200 baudio
Enviar
Nueva línea

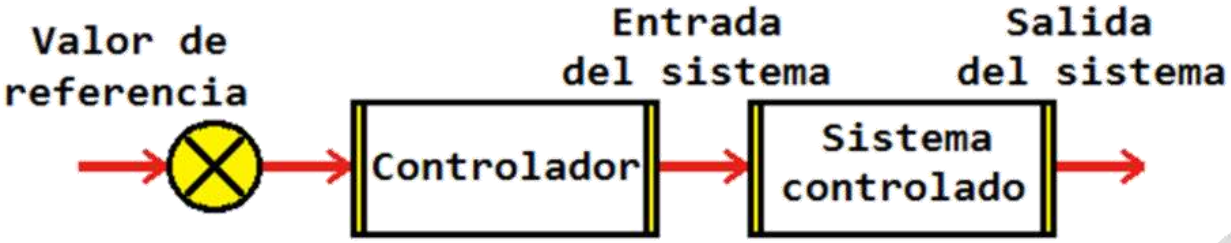
Introducción Teórica: Control de Lazo Abierto

Lo que define a un sistema de control es que este manipula las salidas de un sistema sin que el operador intervenga directamente sobre sus elementos. El operador manipula **el valor deseado que se busca alcanzar o punto de referencia (setpoint)** y el sistema de control se encarga de mantenerlo a través de los accionamientos de sus salidas. Pero dentro de esta categoría existen dos tipos globales de control:

- **Control de lazo cerrado:** Este tipo de control cuenta con una entrada de retroalimentación para constantemente medir el error del **setpoint** y corregirlo por medio de sus **actuadores**, dando como resultado una **mínima desviación, osea mucha precisión**.



- **Control de lazo cerrado:** Este tipo de control NO cuenta con una entrada de retroalimentación por lo que su salida no tiene efecto sobre la acción del **actuador**, por lo tanto, para cada **valor de referencia** corresponde una condición de operación fija, por lo que **la precisión del sistema depende totalmente del algoritmo de control y su calibración**.



Referencias

ELECTRONOOBS en Español, “Pr#83 - PID Balanza y Bola | Teoría y Calibración”, 2020 [Online], Available: https://www.youtube.com/watch?v=WsoaQNw04zs

El profe García, “Entendí el Control PID, si lo aprendes vas a poder...”, 2019 [Online], Available: https://www.youtube.com/watch?v=gtsZ2hswKJk&t=105s

How To Mechatronics, “How MEMS Accelerometer Gyroscope Magnetometer Work & Arduino Tutorial”, 2015 [Online], Available: https://www.youtube.com/watch?v=eqZgxR6eRjo

VirtualBrain, “Cómo Funciona un Giroscopio ⚡ Qué es un Giroscopio”, 2021 [Online], Available: https://www.youtube.com/watch?v=0cH2JCT8xbU&t=510s

ELECTRONOOBS en Español, “Pr#011 PID balanza con motores brushless”, 2018 [Online], Available: https://www.youtube.com/watch?v=jh9dXNzTzz8

ELECTRONOOBS en Español, “Arduino PID controller for one axis brushless drone motors”, 2017 [Online], https://www.youtube.com/watch?v=w2hZoZQyRw8

ELECTRONOOBS en Español, “Pr#36 Dron Arduino - Serie 2 | Parte 2 IMU y PID”, 2018 [Online], Available: https://www.youtube.com/watch?v=Rbl5n31bCE0&t=122s