UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Implementación de un Péndulo Invertido con DSPIC

Trabajo Fin de Grado

Grado en Ingenieria Informatica

Autor: Antonio Dudarev Dudareva

Tutor: Pascual Perez Blasco

25 de noviembre de 2015

Resumen

En este documento se describe el proceso de diseño e implementación de un péndulo invertido de dos ruedas utilizando un acelerómetro, un giroscopio y un magnetómetro como periféricos de entrada y unos servomotores modificados como actuadores del sistema. Un algoritmo PID se encarga de controlar el sistema, dicho algoritmo esta integrado dentro de un microcontrolador de la familia dsPIC33F. Se utiliza bluetooth como capa de comunicaciones con el sistema, por el cual se puede obtener datos y ajustar parámetros del mismo.

Palabras clave: Péndulo invertido, PID, microcontrolador

Abstract

This document describes de design and implementation process of an two-wheeled inverted pendulum using an accelerometer, gyroscope and a magnetometer as inputs and two modified servos as actuators for the system. A PID algorithm is responsible for controlling the system, this algorithm is integrated into a dsPIC33F family microcontroller. Bluetooth is used as a communications layer for the system, thus we can obtain data or adjust parameters of the system

Keywords: Inverted pendulum, PID, microcontroller

Índice

1. Introducción

Un péndulo invertido es un problema clásico de control, donde se quiere obtener como salida del sistema un equilibrio contrario a la fuerza de la gravedad. En este caso vamos a limitar el sistema a un grado de libertad, es decir solo controlaremos un eje en el espacio. El sistema tiene como entrada un sensor que contiene un acelerómetro, un giroscopio y un magnetómetro, a través de los cuales usando un algoritmo de control que implementaremos sobre un microcontrolador, actuara sobre dos servo-motores para mantener en equilibrio el péndulo invertido. Uno de los objetivos indirectos de este proyecto es abaratar costes, para ellos vamos a utilizar los componentes mínimamente necesarios para el desarrollo del proyecto.

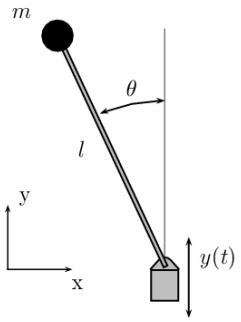
1.1. Motivación

La motivación para hacer este proyecto es practicar con los conocimientos adquiridos para poder aplicarlos a la vida real, ademas el desconocimiento que tengo en el campo de los microcontroladores y sistemas empotrados es algo que debo aprender para poder mejorar en el campo de la informática.

2. Descripción del problema

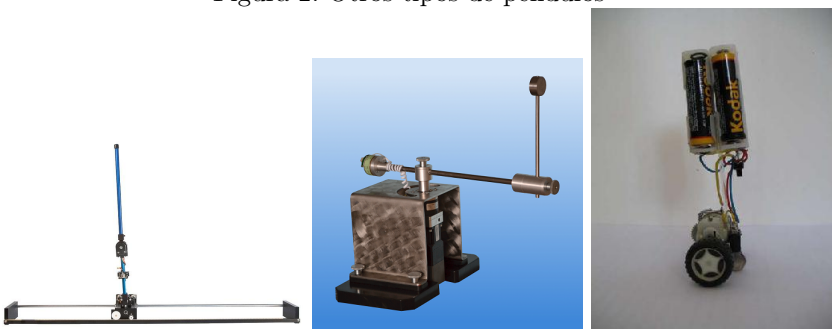
Como se ha mencionado antes el péndulo invertido es un problema clásico de control en el cual se intenta obtener el equilibrio de una barra rígida con una masa en el extremo superior. Este problema se puede limitar a un eje de libertad, es decir que solo puede variar el ángulo en uno de los planos del espacio tridimensional. La Figura 1 ilustra la configuración del polo en libertad.

Figura 1: Un péndulo invertido



θ es el ángulo con respecto al punto de equilibro del péndulo, l la longitud de la barra, m la masa en uno de los extremos del la barra. Lo que se intenta conseguir es que θ sea lo menor posible, esto sera el punto de equilibrio. Existen varios tipos de péndulos invertidos, con varias implementaciones, pero el objetivo es el mismo para todos. Existen dobles péndulos, e triples péndulos, estos utilizan dos y tres barras rígidas respectivamente. También existe una variante llamada péndulo de furuta que tiene menos actuadores que grados de libertad tiene el sistema, estos sistemas se denominan sistemas subactuados. El péndulo descrito en esta memoria es un robot de dos ruedas que mantendrá el equilibro, un sistema similar es el segway.

Figura 2: Otros tipos de péndulos



3. dsPIC33F

Para la realización de este trabajo se ha optado por un microcontrolador de marca Microchip, mas concretamente un dsPIC33FJ128MC804. Estos son microcontroladores capaces de trabajar con señales digitales y están diseñados para soportar grandes cargas computacionales en todo tipo de entornos. Estos pueden encontrarse empotrados en varios sistemas, desde industriales hasta productos de consumidor. (vehículos motorizados, lavadoras, cámaras, ...). La familia dsPIC33F fue introducida en 2001, pero solo 3 años mas tarde comenzó la producción en masa de dichos microcontroladores, se describen como microcontroladores de propósito general e incluyen la capacidad de procesar señales digitales. La programación del microcontrolador se puede escribir en C gracias al compilador XC16 que es una variante de GCC, lo que simplifica mucho el trabajo del programador ya que también se puede utilizar instrucciones especificas de ensamblador a través de macros.

3.1. Características MCU

La siguiente tabla nos muestra las características descritas en la descripción del producto de la familia dsPIC33F[4].

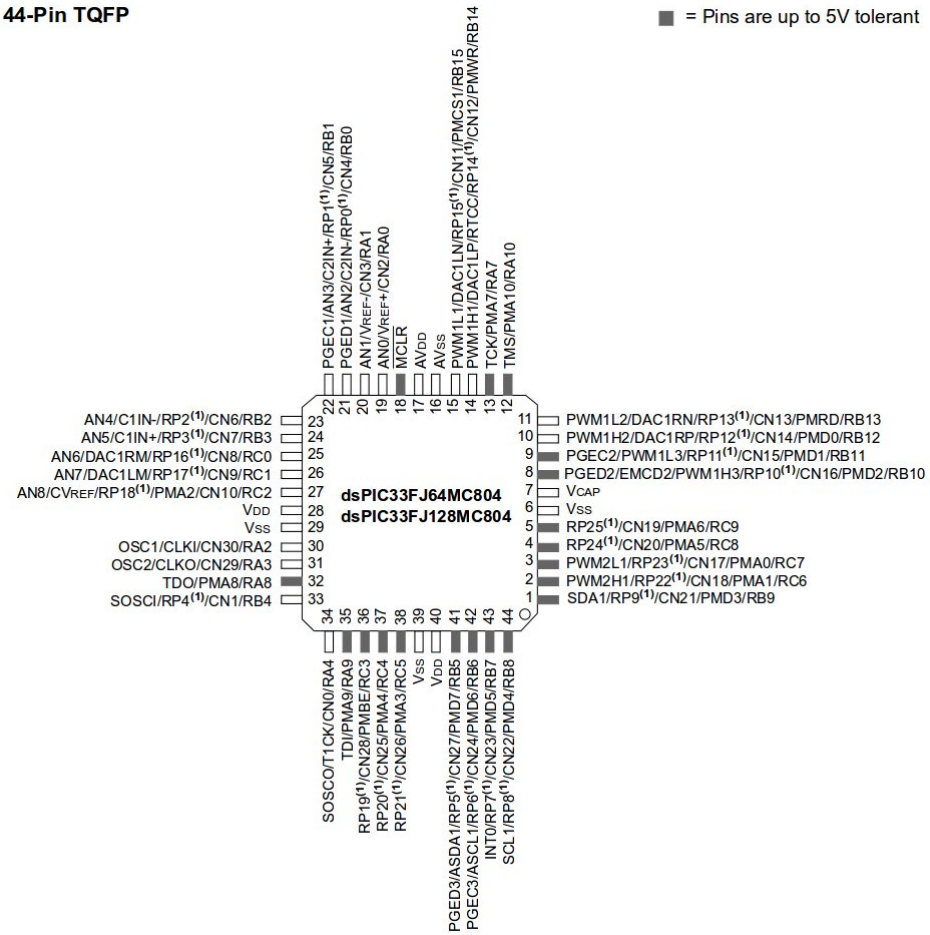
Tabla 1: Características de la familia dsPIC33F

Condiciones operacionales	3.0V hasta 3.6V CC, -40°C hasta +150°C, 20 MIPS 3.0V hasta 3.6V CC, -40°C hasta +125°C, 40 MIPS	Periféricos del sistema	Modulo CRC Convertidor Audio DAC 100ksps 16-bits Hasta 5 contadores de 16 bits o 2 de 32 bits Cuatro módulos IC (Input Capture) Cuatro módulos OC (Output Capture) Dos Encoders de cuadraturas (QEI) Reloj de tiempo real y calendario (RTCC)
Manejo del reloj del sistema	Oscilador interno del 2 % PLL programable y entradas de reloj Monitor de fallos de reloj Contador Watchdog Modos de bajo-consumo Despertar y Comienzo rápidos	Interfaces de comunicación	Puerto Paralelo maestro Dos módulos UART a 10Mbps Dos módulos SPI a cuatro cables (15Mbps) Un modulo ECAN 1Mbaudio con soporte 2.0B Un modulo I^2C de 100K, 400K o 1MBaudio
Rendimiento del Core	Hasta 40 MIPS Dos acumuladores de 40 bits de ancho MAC/MPY de un ciclo con búsqueda dual Multiplicaciones de un ciclo, división hardware	Direct Memory Access (DMA)	DMA de 8 canales sin stalls de CPU o sobrecarga UART,SPI,ADC,ECAN,IC,OC,INT0
Control de motores PWM	Hasta cuatro generadores de PWM con 8 salidas Tiempo muerto para subidas y bajadas de reloj Resolución PWM de 25ns Soporte para control de motores: BLDC, PMSM, ACIM y SRM Entradas de fallo programables Triggers para conectar con los ADC	Características Analógicas Avanzadas	ADC de 10/12 bits con un ratio de conversión de 1.1Msps/500ksps 9 canales de entrada de ADC y cuatro S&H Dos Comparadores de 150ns DAC de 4 bits con dos rangos de comparación
Entradas/Salidas	Pines remapeables por software Pines tolerantes a 5V Drenadores abiertos y resistencias de pull-up internas seleccionables Múltiples interrupciones externas	Soporte para depuración	Programación In-circuit Dos breakpoints programables Trazas y sondeo de variables en tiempo de ejecución

El microcontrolador se puede pedir en varios empaquetados, pero el que vamos a utilizar es TQFP, y el pinout del microcontrolador es el que se muestra

en la Figura 3.

Figura 3: Diagrama de pines del microcontrolador



3.2. Módulos

Como se puede apreciar en la Tabla 1, la MCU contiene varios módulos que se pueden utilizar para implementar el péndulo. El péndulo invertido se controlara por medio de dos servomotores SM-S4303R, estos son unos servomotores con un par motor de 6Kg/cm^2, una velocidad de 50RPM y de rotación continua, modificados para funcionar sin electrónica que se van a utilizar como actuadores del sistema por medio de una señal PWM que se describe mas adelante en el documento. Esta señal controlara la fuerza con la que giran los motores, por lo tanto afectara al momento de fuerza aplicado sobre el péndulo, aplicando el correcto momento de fuerza en dirección contraria podemos equilibrarlo. Además para corregir la posición del péndulo, debemos conocer nuestra posición en el espacio por lo que utilizaremos un sensor de Invensense, para comunicarnos con dicho sensor utilizamos el modulo I^2C a través del cual podremos consultar el estado del sistema en un instante dado. Para las comunicaciones utilizaremos el

protocolo Bluetooth, utilizando un modulo externo ZS404 para conectarnos a otros dispositivos y una conexión serie con el microcontrolador se obtiene la base de las comunicaciones. Las comunicaciones serie son gestionadas por el modulo UART del microcontrolador. Cada modulo tiene un conjunto de registros que se utilizan para la configuración del mismo, las varias opciones de cada modulo y como se configura se puede consultar en la documentación del microcontrolador, en esta sección solo se mostrara el código que hace uso de los módulos.

3.2.1. Modulo UART

El protocolo UART (Universal Asynchronous Reciever Transmiter) es una conexión serie en la cual se utiliza una sola linea para la transferencia de datos, lo que significa que no lleva señal de reloj y los dos dispositivos deben saber a priori la velocidad del bus de datos que en nuestro caso es de 9600 baudios. En el microcontrolador hay que configurar un registro para generar la frecuencia de funcionamiento deseada, podemos calcular el valor de dicho registro usando la Formula 1, así podemos configurar la frecuencia de funcionamiento de la linea de datos para que el modulo bluetooth externo pueda comunicarse con el microcontrolador.

$$\begin{aligned} \text{U1BRG} &= \frac{F_P}{4 \times \text{Baud rate}} - 1 \\ \text{U1BRG} &= \frac{(7{,}37MHz/2)}{4 \times 9600} - 1 \Rightarrow 94{,}9635416667 = 95 \end{aligned} \tag{1}$$

F_P es la frecuencia de oscilación dividido entre 2 y Baud_Rate la tasa de baudios deseada. Una vez tenemos el valor calculado debemos calcular el máximo error del modulo, este error no debe superar el 5 %. Para calcular el error utilizamos la Formula 2

$$\begin{aligned} \text{Calculated Baud Rate} &= \frac{F_P}{4 \times (\text{U1BRG} + 1)} \\ \text{Calculated Baud Rate} &= \frac{(7{,}37MHz/2)}{4 \times (95 + 1} \Rightarrow 9596{,}35416667 = 9596 \\ \text{Error} &= \frac{9596 - 9600}{9600} \Rightarrow -0{,}042\,\% \end{aligned} \tag{2}$$

El error se asocia con la desviación a la hora de muestrear de cada bit, si el error superase el 5 % podrían perderse bits en la comunicación. A través del modulo vamos a enviar y recibir caracteres por lo que podemos calcular cuantos caracteres se podrían enviar a través del modulo por segundo, esto se calcula con la Formula 3, cada carácter se codifica con 8 bits y también tenemos el start y stop bit por lo que cada carácter se envía con 10 bits, al tener un baud rate de 9600, esto significa que podemos mandar hasta 9600 símbolos por segundo, si un símbolo es un bit mas el start bit y el stop bit podemos obtener los caracteres por segundo que podemos mandar.

$$9600/10 = 960 \text{ Caracteres por segundo} \tag{3}$$

Hay que tener en cuenta que tenemos un bucle de control el cual nos consume tiempo de computo, por lo que debemos controlar cuanto tiempo nos consume el modulo UART a la hora de enviar. Esto se explica mas adelante en este

documento. El modulo UART nos permitirá comunicarnos con el dispositivo Bluetooth que a su vez enviara datos a un dispositivo conectado para la obtención de datos y ajuste de parámetros del algoritmo PID a través de un protocolo que se describe mas adelante en este documento. Para la implementación vamos a hacer uso de interrupciones que nos ofrece la MCU, esto se debe a que así podemos controlar cuando esta activo el modulo para que no consuma mas tiempo del deseado, se tiene una interrupción para la recepción y otra para el envió usando dos lineas separadas. Lo primero a tener en cuenta es que la entrada y salida al modulo no se encuentra asociado a ningún pin externo por lo que necesitaremos remapear la salida y la entrada utilizando el modulo PPS, el PPS se describe mas adelante en este documento. Una vez tenemos el modulo UART conectado a unos pines físicos podemos comenzar a programar la interrupciones, la inicialización de las interrupciones se hace a través de unos registros especiales que se encuentran descritos en la ficha técnica del dispositivo[4]. Las interrupciones se muestran en el Código 1

Codigo 1: Interrupciones dentro del modulo UART

```
char string[MAX_BUFF];
unsigned int CharsLeft=0;
int head, tail;
char rstring[READ_BUFF];
unsigned int read_from=0;
//Interrupt when one character finishes transmiting
void __attribute__((__interrupt__,no_auto_psv)) _U1TXInterrupt(void)
{
    int aux;
    if(CharsLeft==0){/*Do nothing*/}else{
        aux = head;
        head==MAX_BUFF-1 ? head=0 : head++ ;
        CharsLeft--;
        U1TXREG = string[aux];
    }
    IFS0bits.U1TXIF = 0;
}
//Interrupt when recieve buffer is ready to be readed
void __attribute__((__interrupt__,no_auto_psv)) _U1RXInterrupt(void)
{
        char b;
        if(U1STAbits.URXDA == 1) { //Get the data
            if(read_from > READ_BUFF) {  //OVERFLOW
                        read_from =0;
            }else{
                    b=U1RXREG;
                    if(b ==  '\r' || b == '\n'){
                        rstring[read_from]='\0';
                        parseProtocol(rstring);
                        SendLetter(rstring);
                        read_from=0;
                    }else{
                        rstring[read_from]=b;  // Appending to string
                        read_from+=1;
                    }
            }
        }
        IFS0bits.U1RXIF = 0;
}
```

Se ha optado por la implementación de una cola circular de 1024 caracteres, primero se implementa la interrupción de envió, y cuando tenemos esta, podemos implementar la de recepción. Ahora que tenemos las interrupciones podemos comenzar a enviar datos y recibir del dispositivo bluetooth. Para una primera prueba se ha utilizado un programa disponible en la Play Market de android que se llama Bluetooth terminal, así podemos comprobar que el envió de datos desde la MCU funciona. Después de comprobar que el envió de datos funciona, podemos proceder a recibir y enviar lo que se acaba de recibir para comprobar la interrupción de recepción. La velocidad de dsPIC esta muy por encima de la velocidad de comunicaciones por lo que para su correcto funcionamiento debemos ajustarlo usando un registro de configuración que baja la frecuencia de funcionamiento del modulo, en nuestro caso los baudios. Las interrupciones funcionan con dos buffers diferentes, para la recepción la interrupción ocurre cuando el ultimo carácter que se ha introducción en el registro de envió de bits se ha vaciado, entonces busca un nuevo carácter en el buffer circular y lo carga de nuevo en el mismo registro, cuando la cola esta vacía esta deja de ocurrir. En el caso de la recepción es al revés, cuando el registro de recepción se llena la interrupción ocurre, entonces la rutina recoge el resultado de dicho registro y lo guarda en un buffer de recepción, esta comprueba que el ultimo carácter es un retorno de carro o un salto de linea y si lo es, activa la capa de protocolo para el procesamiento del mensaje recibido. Mas adelante se entrara en detalle de la implementación del protocolo que recibe los mensajes, y lo que podemos obtener gracias a el. El Código 2 complementa a esta implementación para envió de cadenas de caracteres usando el mismo buffer circular.

Codigo 2: Funciones de envio y recepcion

```
int SendString(char *str){
    int i;
    int j = strlen(str); // How long is the string?
    if(CharsLeft + j > MAX_BUFF-1) { return 0; }  //OVERFLOW

    for(i=0;i<j;i++){
        // Appending to string
        // *string starts at 0 and is j long! 0 = i - CharsLeft!
        string[tail]=str[i];
        if(tail==MAX_BUFF-1){
            tail=0;
        }else{
            tail++;
        }
    }

    if(CharsLeft==0){
        CharsLeft=j;
        IFS0bits.U1TXIF = 1;
    }else{
        CharsLeft+=j;
    }
    return 1;
}
```

Las interrupciones del modulo UART pueden ocurrir en cualquier momento de la ejecución del programa, la única condicione es que no se pueden solapar, aunque se puede activar el solapamiento de interrupciones a través de registros

especiales. Las comunicaciones deben ocurrir solo cuando el procesador no tiene que calcular nada en el bucle de control por lo que se activan solo al final de dicho bucle. El Código 3 muestra las rutinas que se ocupan de desactivar y activar las interrupciones UART.

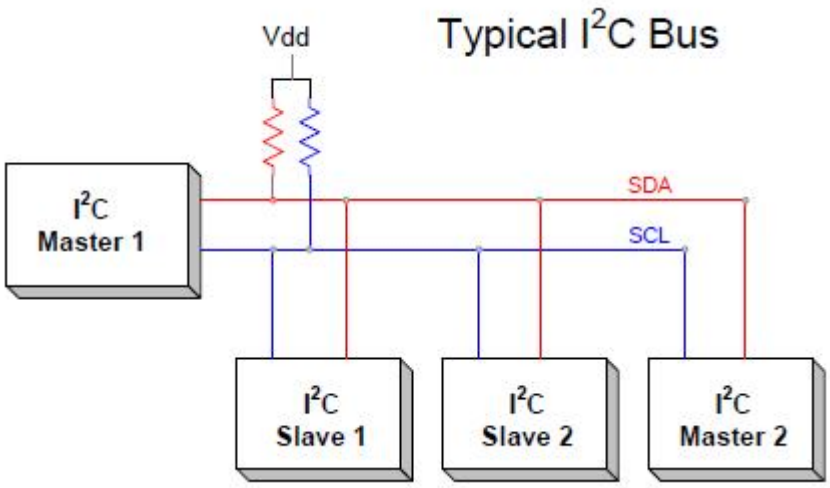
Codigo 3: Funciones de encendido y apagado de interrupciones

```
void enableUARTInt(){
        IEC0bits.U1TXIE = 1;
}
void disableUARTInt(){
        IEC0bits.U1TXIE = 0;
}
```

3.2.2. Modulo I^2C

El protocolo I^2C (Inter Integrated Circuit) es un protocolo maestro-esclavo, este se basa en dos lineas, una de reloj y otra de datos comúnmente nombradas SCL y SDA respectivamente. El diseño del bus es de drenador abierto, por lo que siempre se necesitaran resistencias de pull-up. Este protocolo puede comunicar a mas de dos dispositivos a la vez en el mismo bus, en el cual el Maestro vuelca una señal de reloj en la linea SCL y los demás dispositivos se sincronizan con el. A partir de ahí se utiliza la linea de datos para transferir la información.

Figura 4: Ejemplo de bus I^2C



Como la frecuencia de la señal de reloj del microcontrolador es mucho mayor que la de la bus, debemos ajustarla a través de un registro a una frecuencia de funcionamiento contenida entre 100KHz hasta 1MHz, estos son los valores en los que se contiene la velocidad de funcionamiento del bus I^2C. Este modulo sera el responsable de comunicarse con la unidad que contiene los sensores, esto nos condiciona a una frecuencia máxima de muestreo. Se ha optado por I^2C ya que la unidad de sensores solo es capaz de utilizar este protocolo de comunicación. Para el calculo del valor del registro generador de frecuencia se utiliza la Formula 4.

Se debe escoger un valor arbitrario de espera, en la ficha técnica se recomiendan valores entre 110ns y 130ns. Se opta por utilizar 400KHz de velocidad ya que en la ficha técnica de la MPU-9150 se menciona esta velocidad como la máxima permitida para la linea SCL. Asimismo se opta a un delay de 110ns para ganar la mayor velocidad posible.

$$\begin{aligned} &\text{I2C1BRG} = ((1/FSCL - \text{Delay}) \times FCY) - 2 \\ &\text{I2C1BRG} = ((1/400\text{KHz} - 110\text{ns}) \times (7{,}37/2)) - 2 \Rightarrow 6{,}80715 = 7 \end{aligned} \tag{4}$$

Una vez configurada la velocidad de funcionamiento del bus podemos comenzar a implementar las comunicaciones, para ello debemos consultar la ficha técnica del sensor, En nuestro caso una MPU-9150[3].

Figura 5: Secuencia I^{2}C de la MPU-9150

Single-Byte Write Sequence

Master	S	AD+W		RA		DATA		P
Slave			ACK		ACK		ACK	

Burst Write Sequence

Master	S	AD+W		RA		DATA		DATA		P
Slave			ACK		ACK		ACK		ACK	

Single-Byte Read Sequence

Master	S	AD+W		RA		S	AD+R			NACK	P
Slave			ACK		ACK			ACK	DATA		

Burst Read Sequence

Master	S	AD+W		RA		S	AD+R			ACK		NACK	P
Slave			ACK		ACK			ACK	DATA		DATA		

9.4 I^{2}C Terms

Signal	Description
S	Start Condition: SDA goes from high to low while SCL is high
AD	Slave I^{2}C address
W	Write bit (0)
R	Read bit (1)
ACK	Acknowledge: SDA line is low while the SCL line is high at the 9th clock cycle
NACK	Not-Acknowledge: SDA line stays high at the 9th clock cycle
RA	MPU-9150 internal register address
DATA	Transmit or received data
P	Stop condition: SDA going from low to high while SCL is high

En la Figura 5 se observa el funcionamiento del bus con el sensor. El microcontrolador implementa unos registros que controlan las señales en el bus por lo que la programación es sencilla, solo hay que poner a uno los campos correspondientes en los registros pertinentes para mandar señales como por ejemplo el start bit o el stop bit. El start bit y el stop bit se utilizan para comenzar/seguir o terminar las comunicaciones por por el bus respectivamente. Las direcciones I^{2}C son de 7 bits de longitud, además en nuestro sensor tenemos un pin externo que dependiendo del voltaje en dicho pin el bit de menor peso de la dirección

estará a 0 o a 1. Como las comunicaciones son del tamaño de un byte el octavo bit sirve para comunicar al esclavo si se esta haciendo una lectura o una escritura sobre sus registros. Cuando el esclavo con la dirección correspondiente manda confirmación, se procede a enviar la dirección del registro, en memorias EEPROM la dirección consta de dos bytes por lo que se mandan dos bytes de dirección de registro por separado, en este caso las direcciones son de un byte de largo por lo que después de mandar la dirección del dicho registro y recibir confirmación del esclavo podemos mandar los datos. Dependiendo de la operación que se ha pedido se procede a leer o mandar los datos del contenido del registro marcado en el paso anterior.

Codigo 4: Funciones de comunicaion I^2C

```
unsigned int StartCondition_i2c(unsigned int addr, char mode) {
    unsigned int data = addr;
    if (mode == 'w') {
        data = data & 0xFE;
    } else if (mode == 'r') {
        data = data | 0x01;
    } else {
        return 1;
    }
    I2C1CONbits.ACKDT = 0;
    I2C1CONbits.SEN = 1;
    while (I2C1CONbits.SEN != 0);
    while (I2CSTATbits.TBF != 0);

    I2C1TRN = data;
    if (I2CSTATbits.IWCOL) {
        return 1;
    } else {
        while ( I2CCONbits.SEN ||
                I2CCONbits.RSEN ||
                I2CCONbits.PEN ||
                I2CCONbits.RCEN ||
                I2CCONbits.ACKEN ||
                I2CSTATbits.TRSTAT);
        return I2C1STATbits.ACKSTAT;
    }
}

void StopCondition_i2c() {
    while ( I2CCONbits.SEN ||
            I2CCONbits.RSEN ||
            I2CCONbits.PEN ||
            I2CCONbits.RCEN ||
            I2CCONbits.ACKEN ||
            I2CSTATbits.TRSTAT);

    I2C1CONbits.PEN = 1;
    while (I2C1CONbits.PEN != 0);
}

unsigned int Read_i2c() {
    unsigned int buff;
    I2C1CONbits.RCEN = 1; //DATA*n
    while (I2C1CONbits.RCEN != 0);
    I2C1CONbits.ACKDT = 1;
    buff = I2C1RCV;
    return buff;
```

```
}

unsigned int Write_i2c(unsigned int val) {
    I2C1TRN = val;
    if (I2CSTATbits.IWCOL) {
        return 1;
    } else {
        while ( I2CCONbits.SEN ||
                I2CCONbits.PEN ||
                I2CCONbits.RCEN ||
                I2CCONbits.ACKEN ||
                I2CSTATbits.TRSTAT);
        return I2C1STATbits.ACKSTAT;
    }
}
```

El Código 4 muestra la implementación de este protocolo utilizando los registros proporcionados por el microcontrolador. Las comunicaciones comienzan con la función StartCondition_i2c, a este se le pasan dos parámetros, la dirección del dispositivo y el tipo de operación (escritura/lectura). La dirección tiene que estar previamente desplazada para ocupar su posición correcta en el byte, seguidamente se aplica una mascara al byte dependiendo del tipo de operación. Cuando se tienen los datos preparados se manda el start bit (I2C1CONbits.SEN = 1) y se espera a recibir confirmación, al recibirla se cargan los datos sobre el registro de salida y se comprueba si tenemos una colisión en el registro, si no se procede a esperar confirmación. La función StopCondition_i2c comprueba que no hay ninguna operación en curso y seguidamente manda el stop bit. Read_i2c y Write_i2c hacen uso de los registros de entrada y salida para leer o escribir los datos, estas funciones siempre se encuentran entre el StartCondition_i2c e StopCondition_i2c. Este código no es muy cómodo para trabajar con el, por ello se ha implementado por encima unas funciones que se encargan de leer o escribir registros, también se debe implementar ciertas funciones que se utilizan para reprogramar el sensor y que se explican mas adelante en el documento. Estas se muestran en el Código 5.

Codigo 5: Funciones de comunicaion I^2C simplificadas

```
unsigned int ReadRegister_i2c(unsigned int addr, unsigned int reg) {
    unsigned int data;
    unsigned int res = 1;
    unsigned int barr = 1;
    Timer2Initialize();
    set_flag2();
    while (res != 0 && barr == 1) {
        barr = barrier2();
        res = 0;
        res += StartCondition_i2c(addr, 'w');
        res += Write_i2c(reg);
        res += StartCondition_i2c(addr, 'r');
        data = Read_i2c();
        StopCondition_i2c();
        if (I2CSTATbits.IWCOL) I2CSTATbits.IWCOL = 0;
    }
    if(res == 0 && barr == 0){
        printStr("I2C Read Comm LastRound\n");
    }else if(res > 0 && barr == 0){
        printStr("I2C Read Comm Failure\n");
        printInt(I2C1STAT);
```

```
    }
    return data;
}

void WriteRegister_i2c(unsigned int addr, unsigned int reg, unsigned int val) {
    unsigned int res = 1;
    unsigned int barr = 1;
    Timer2Initialize();
    set_flag2();
    while (res != 0 && barr == 1) {
        barr = barrier2();
        res = 0;
        res += StartCondition_i2c(addr, 'w');
        res += Write_i2c(reg);
        res += Write_i2c(val);
        StopCondition_i2c();
    }

    if(res == 0 && barr == 0){
        printStr("I2C Write Comm LastRound\n");
    }else if(res > 0 && barr == 0){
        printStr("I2C Write Comm Failure\n");
        printInt(I2C1STAT);
    }
}

int WriteRegister_bit_i2c(unsigned int devAddr,unsigned int regAddr, int bitNum, unsigned int data) {
    unsigned int b;
    b = ReadRegister_i2c(devAddr, regAddr);
    b = (data != 0) ? (b | (1 << bitNum)) : (b & ~(1 << bitNum));
    WriteRegister_i2c(devAddr, regAddr, b);
    return b;
}
int WriteRegister_bits_i2c(unsigned int devAddr, unsigned int  regAddr, unsigned int  bitStart,
         unsigned int  length, unsigned int data) {
    unsigned int b;
    unsigned int mask;
    b = ReadRegister_i2c(devAddr, regAddr);
        mask = ((1 << length) - 1) << (bitStart - length + 1);
        data <<= (bitStart - length + 1); // shift data into correct position
        data &= mask; // zero all non-important bits in data
        b &= ~(mask); // zero all important bits in existing byte
        b |= data; // combine data with existing byte
        WriteRegister_i2c(devAddr, regAddr, b);
        return b;
}
int ReadRegister_bit_i2c(unsigned int devAddr,unsigned int regAddr, int bitNum, unsigned int *data,
        unsigned int timeout){
    unsigned int b;
    b = ReadRegister_i2c(devAddr, regAddr);
    *data = b & (1 << bitNum);
    return 1;
}

int ReadRegister_bits_i2c(unsigned int  devAddr, unsigned int  regAddr, unsigned int  bitStart,
        unsigned int  length, unsigned int  *data, unsigned int timeout){
    unsigned int b;
    unsigned int mask;
    b = ReadRegister_i2c(devAddr, regAddr);
    mask = ((1 << length) - 1) << (bitStart - length + 1);
    b &= mask;
    b >>= (bitStart - length + 1);
```

```
    *data = b;
    return (bitStart-length);

}
```

Las funciones de WriteRegister_i2c y ReadRegister_i2c comprueban si ha habido un fallo en la comunicación, además, contamos con un timer que se activa al principio de la comunicación, y expira dado un tiempo T, este tiempo es aproximadamente 3 intentos de del bucle de comunicaciones, es decir si tras tres intentos de comunicación fallidos, se sale del bucle con un error de comunicación recuperable. Las funciones de lectura y escritura de bits son una combinación de estas anteriores en las cuales los bits son modificados dentro del microcontrolador y no directamente sobre el sensor. Los registros que nos interesan son registros repartidos en high y low, estos registros contienen un numero de 16 bits que representa el valor del sensor en cuestión en uno de sus tres ejes en complemento a dos. Esto nos obliga a leer dos registros y formarlo en el destino utilizando desplazamiento de bits.

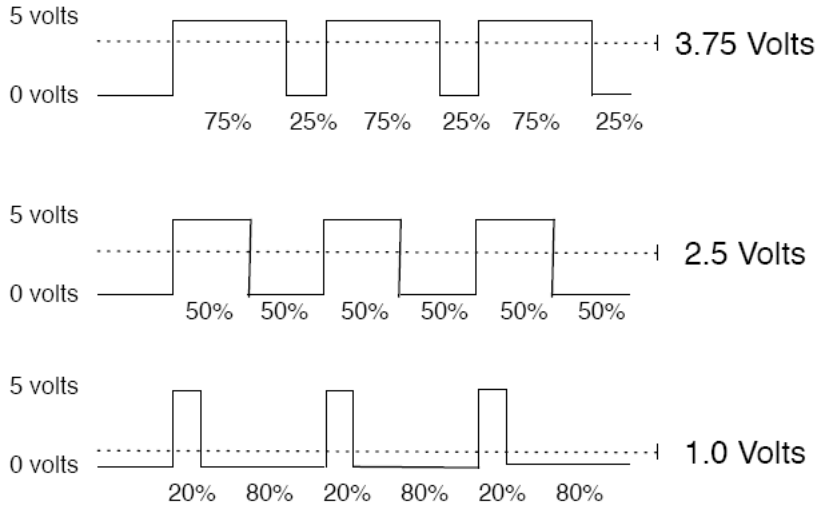
3.2.3. Modulo PWM

El modulo PWM (Pulse Width Modulation) se utiliza con frecuencia para el manejo de motores, aunque puede ser utilizado también para transmitir información. Usa solo una linea en la cual el voltaje y corriente varia según lo que especifiquemos. La Formula 5 describe el principio de funcionamiento del PWM. Si tomamos una función cuadrada $f(t)$ con un periodo T, un valor mínimo y_{min} de voltaje y un valor máximo y_{max} de voltaje, y un ciclo de trabajo D que es el tiempo en el cual la señal esta activa dentro del periodo T, podemos obtener un valor medio de voltaje por cada periodo.

$$\begin{aligned} \overline{y} &= \frac{1}{T}\int_0^T f(t)dt \\ \overline{y} &= \frac{1}{T}(\int_0^{DT} y_{max}dt + \int_{DT}^{T} y_{min}dt) \\ \overline{y} &= \frac{DTy_{max} + T(1-D)y_{min}}{T} \\ \overline{y} &= Dy_{max} + (1-D)y_{min} \end{aligned} \tag{5}$$

Si tenemos que $y_{min} = 0$, la media es un porcentaje del voltaje máximo. La Figura 6 muestra gráficamente la relación directa que existe entre el valor máximo y el ciclo de trabajo.

Figura 6: Ejemplo de una señal PWM con varios ciclos de trabajo



Para especificar el periodo se utiliza un contador que contara hasta un valor dado en el registro P1TPER, este valor sera el máximo del contador, el contador en si se encuentra en el registro P1TMR, este incrementara la cuenta cada ciclo de reloj. El microcontrolador tiene varios modos de funcionamiento, el que se utiliza ofrece una resolución del doble contenido en el registro P1TPER. Para especificar el ciclo de trabajo y sabiendo que tiene una resolución del doble se utiliza el registro P1DCH o P1DCL, donde H o L significa si el registro comienza en estado activo o estado bajo respectivamente. El péndulo utilizara una cuenta de 0 a 16384, podemos calcular el tiempo del periodo utilizando la Formula 6

$$\begin{aligned} \text{FCY} &= \text{FOSC}/2 \\ \text{FCY} &= 7{,}37\text{MHz}/2 \Rightarrow 3{,}67\text{MHz} \\ \text{TCY} &= \frac{1}{\text{FCY}} \\ \text{TCY} &= \frac{1}{3{,}67\text{MHz}} \Rightarrow 272{,}5\text{ns} \\ \text{Periodo} &= \text{TCY} \times \ \text{P1TPER} \\ \text{Periodo} &= 272{,}5\text{ns} \times 16384 \Rightarrow 4{,}46\text{ms} \end{aligned} \tag{6}$$

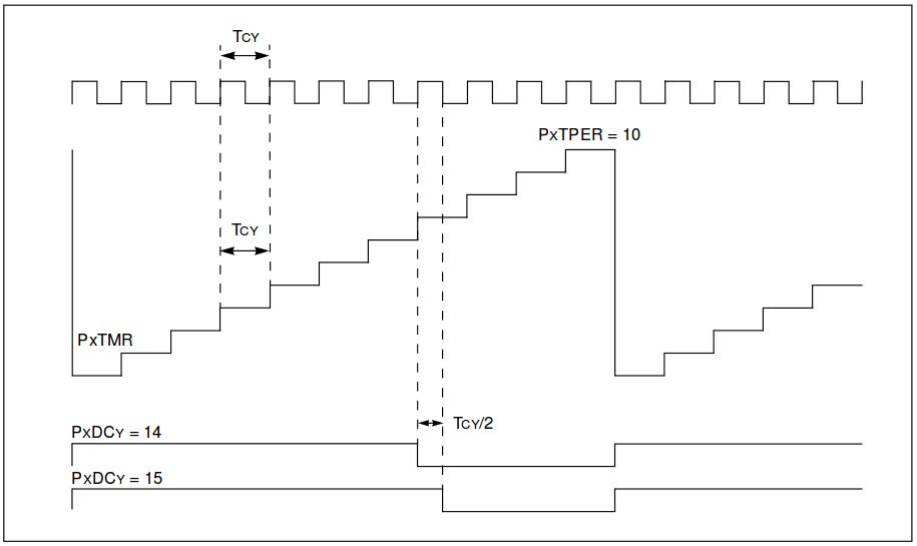
Sabemos que cada 4.46ms tenemos el periodo del PWM, podemos calcular la resolución con la Formula 7, también podemos calcular cual es salto en porcentaje de por bit.

$$\begin{aligned} \text{Resolution} &= \text{P1TPER} \times 2 \\ \text{Resolution} &= 16384 \times 2 \Rightarrow 32768 \\ \text{Step} &= \frac{100}{\text{Resolution}} \\ \text{Step} &= \frac{100}{32768} \Rightarrow 0{,}0031\,\%/\text{LSB} \end{aligned} \tag{7}$$

Lo que se intenta describir en la Formula 7 es que tenemos 32768 velocidades posibles para los motores, y que cada bit menos significativo aumenta el ciclo de

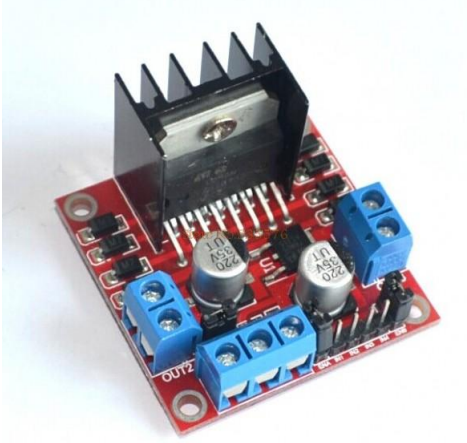
trabajo en 0.0031 %. La Figura 7 describe gráficamente todo lo anteriormente descrito, este modo de funcionamiento se llama Free-Running Mode.

Figura 7: Diagrama de tiempo del funcionamiento del modulo PWM



El microcontrolador tiene una corriente de entrada/salida en cualquier pin es de un máximo de 25mA por lo que no es suficiente para alimentar un motor. Para alimentar los motores vamos a hacer uso de un puente en H, este es un externo al microcontrolador, el modelo es L298N. Este recibirá la salida PWM a través de uno de sus puertos y suministrara la corriente desde una batería externa, a través de otro pin de entrada/salida simple podremos controlar también la dirección de giro de los motores. La Figura 8 muestra el modulo externo.

Figura 8: Puente en H L298N



Codigo 6: Funciones PWM para control de motores

```
void setTimerX(unsigned int count)
{
        if( count > PTPER_MAX || count <= PTPER_MIN  ){
                //Exception!?
        }else{
                P1TPERbits.PTPER=count;
        }
}
void setDutyCycleX(int duty){
        if(duty >= ACTUAL_PTPER*2 ){
                P1DC1=ACTUAL_PTPER*2-1;
        }else if(duty < 1){
                P1DC1=1;
        }else{
            P1DC1=duty;
        }

}
```

En el Código 6 están implementadas las funciones para el control de motores usando los registros anteriormente descritos. Al puente en H se pueden conectar dos lineas PWM, por lo que se utilizan lineas separadas para controlar cada motor, esto se denota como X en las funciones donde X puede ser 1 o 2 (setTimer1,setTimer2,...). Esto nos sirve también para poder corregir el error que puede haber en la velocidad de giro de los motores si son diferentes. Utilizamos dos funciones, una para aumentar reducir el tiempo del contador y otra para para controlar el pulso. Estas dos funciones solo controlan la linea PWM por lo que nos falta controlar la dirección de giro de los motores, para ello usamos dos pines que definen la dirección de giro siendo un pin complementario a otro lógicamente. Esto nos da un total de tres pines por motor, dos para la dirección y una para el PWM, una librería que hace uso de estas funciones simplifica el uso de los motores. El Código 7 muestra esta implementación.

Codigo 7: Funciones simplificadas del uso de los motores

```
void motor(char direction, int strength1, int strength2){
    if(direction=='b'){
        LATBbits.LATB13=0;
        LATBbits.LATB12=1;
    }else if(direction=='f'){
        LATBbits.LATB13=1;
        LATBbits.LATB12=0;
    }
    setDutyCycle1(strength1);
    setDutyCycle2(strength2);
}
```

3.2.4. PPS

El modulo PPS (Peripheral Pin Select) se utiliza para remapear pines del microcontrolador, esto es debido a que hay muchas mas entradas y salidas que pines tiene el microcontrolador. Esto se hace a través de unos registros especiales en los cuales puedes seleccionar los pines marcados como RPX en la Figura 3

y remapearlos a entradas y salidas de casi cualquier modulo. Para el péndulo invertido se deben remapear las dos entradas y salidas del modulo UART.

Codigo 8: Remapeado de dentro del modulo UART

```
__builtin_write_OSCCONL(OSCCON & ~(1<<6));
RPINR18bits.U1RXR=20; //RP20 is now RX
RPOR0bits.RP1R=3; //RP1 is now TX
__builtin_write_OSCCONL(OSCCON | (1<<6));
```

El código que se muestra ene 8 muestra el procedimiento para remapear la entrada y salida uart a los pines RP20 y RP1. La primera y ultima linea del código son rutinas especiales de desbloqueo para poder remapear los pines implementadas por el compilador, mientras que las lineas centrales hacen uso de los registros que remapean salidas/entradas de módulos a pines físicos. Se puede utilizar cualquier pin pero se ha optado por estos por su distribución en la placa.

3.2.5. Interrupciones

El dsPIC también implementa un sistema a través del cual podemos tener interrupciones externas provenientes de otros módulos, tanto externas como internas. Esto también nos sirve a la hora de comunicarnos con los sensores ya que la MPU9150 puede lanzar una interrupción y entonces el microcontrolador lanza una rutina para atender la petición por I^2C. Esto servirá para dividir la fuerza computacional del microcontrolador con el procesador integrado (Digital Motion Processor) que tiene el sensor.

Codigo 9: Interrupcion externa

```
void __attribute__((__interrupt__,no_auto_psv)) _INT0Interrupt(void)
{
  extern int interrupcionMPU;
  IFS0bits.INT0IF = 0;
  interupcionMPU = 1;

}
```

En la Figura 23 se puede observar de donde viene la interrupción externa que provoca la ejecución de esta rutina. Esta interrupción nos sirve para controlar el bucle principal que sera el responsable de consultar esta variable y atender la petición I^2C.

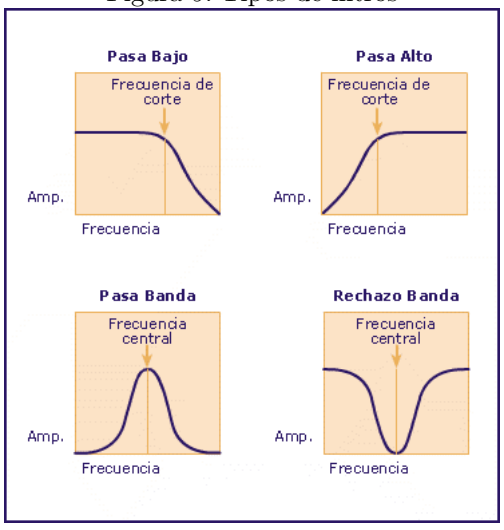
4. Sensores

La unidad de sensores es de la marca Invensense, de todos los productos el que vamos a utilizar es la MPU-9150. Los sensores disponibles en esta unidad son: un acelerómetro, un giroscopio y un magnetómetro. El acelerómetro mide la fuerza de la aceleración de nuestro péndulo mientras que el giroscopio mide la velocidad angular del péndulo. Uno de los problemas que nos plantea utilizar el acelerómetro es que es muy ruidoso por lo que hay que aplicarle un filtro de paso bajo. Por el otro lado tenemos el giroscopio, si integramos su valor obtenemos la distancia recorrida pero al integrar también se integra un error muy pequeño, esto a la larga se desvía mucho del valor real, esto es conocido como deriva.

4.1. Filtros

Los filtros son utilizados para eliminar un rango de frecuencias en una señal analógica. Existen cuatro tipos de filtros, todos estos se diferencian dependiendo del rango de frecuencias que se quiere filtrar. Los filtros paso bajo eliminan las frecuencias mas altas, esto nos puede servir para eliminar ruidos en el acelerómetro o en comunicaciones para obtener la señal de una onda portadora. El filtro paso alto hace lo contrario, deja elimina las frecuencias menores dejando pasar solo valores superiores a cierta frecuencia, podríamos utilizar esto para obtener la onda portadora. Existen tipos mas de filtros, estos son una combinación de las dos anteriores dejando pasar un rango de frecuencias o rechazando un rango de frecuencias, estas son un filtro pasa banda y un filtro rechazo banda respectivamente. La Figura 9 muestra gráficamente el funcionamiento de dichos filtros.

Figura 9: Tipos de filtros



Los filtros paso bajo y paso alto se pueden replicar por software, las implementaciones para cada filtro son diferentes. Para eliminar el ruido del acelerómetro se ha optado por utilizar una media móvil que es una variante del filtro paso bajo para un numero finito de valores, el código para implementar

la media móvil se muestra en el Código 10. Este lee directamente los datos del acelerómetro y calcula la media. La implementación es igual para todos los ejes, en este caso solo se muestra el código del eje X.

Codigo 10: Implementacion de la media movil

```
float readAX(){
    float val;
    int i;
    float mean=0;
    unsigned int dataH, dataL, data;
    dataH=ReadRegister_i2c(MCU_ADDR, ACCEL_XH);
    dataL=ReadRegister_i2c(MCU_ADDR, ACCEL_XL);
    data=dataH<<8;
    data=data+dataL;
    val = (signed int) data;
    val = val*0.000061;
    for(i=1;i<SOFT_LOWPASSFILTER;i++){
        accellowpassx[i-1]=accellowpassx[i];
        mean+=accellowpassx[i-1];
    }
    accellowpassx[SOFT_LOWPASSFILTER - 1]=val;
    mean+=val;
    mean=mean/SOFT_LOWPASSFILTER;
    return mean;
}
```

4.2. Obteniendo el ángulo

Con las comunicaciones ya funcionando, podemos sondear cualquier registro del sensor. El péndulo solo necesita 12 registros, 6 por cada sensor, de los 6, 2 por cada eje del sensor. Una vez obtenidos la información de dichos registros hay que dividir su contenido por una constante para que nos devuelva el valor real. Para cada sensor se debe dividir por su sensividad que es ajustable a través de los registros de la MPU9150.

4.2.1. Acelerómetro

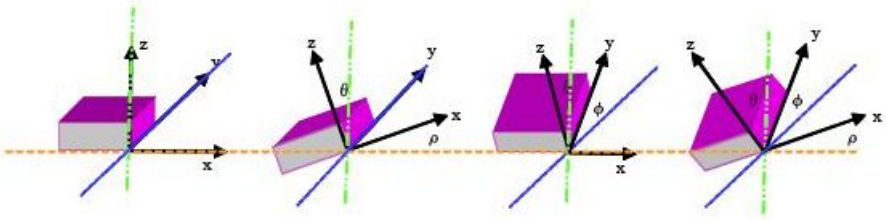
Con el filtro paso bajo implementado obtenemos valores mas estables, ahora que podemos leer la aceleración en cada eje a partir de los tres se puede obtener el ángulo de euler para cualquier de ellos. El acelerómetro tiene varios modos de funcionamiento, estos se diferencian entre si en la sensibilidad y por tanto la precisión de las medidas. La Figura **??** obtenida de la ficha técnica ilustra estos valores.

Figura 10: Sensibilidad de cada eje del acelerómetro

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
ACCELEROMETER SENSITIVITY						
Full-Scale Range	AFS_SEL=0		±2		*g*	
	AFS_SEL=1		±4		*g*	
	AFS_SEL=2		±8		*g*	
	AFS_SEL=3		±16		*g*	
ADC Word Length	Output in two's complement format		16		bits	
Sensitivity Scale Factor	AFS_SEL=0		16,384		LSB/*g*	
	AFS_SEL=1		8,192		LSB/*g*	
	AFS_SEL=2		4,096		LSB/*g*	
	AFS_SEL=3		2,048		LSB/*g*	
Initial Calibration Tolerance			±3		%	
Sensitivity Change vs. Temperature	AFS_SEL=0, -40°C to +85°C		±0.02		%/°C	
Nonlinearity	Best Fit Straight Line		0.5		%	

Si suponemos que solo tenemos una aceleración constante, en este caso la gravedad, a partir de esta fuerza se puede obtener el ángulo de cualquiera de los ejes.

Figura 11: Midiendo el ángulo en tres ejes



En la Figura 12 describe gráficamente como se obtendrían los ángulos para cada eje. La Formula 8 se observa como se relacionan todos los valores de los ejes para obtener cualquier ángulo.

$$\begin{aligned} \rho &= arctan(\frac{A_X}{\sqrt{A_Y^2 + A_Z^2}}) \\ \phi &= arctan(\frac{A_Y}{\sqrt{A_X^2 + A_Z^2}}) \\ \theta &= arctan(\frac{\sqrt{A_Y^2 + A_X^2} A_Z}{}) \end{aligned} \tag{8}$$

4.2.2. Giroscopio

El giroscopio es un dispositivo que mide la velocidad angular, si integramos esta velocidad angular podemos obtener la posición del ángulo a partir de una posición conocida. El problema que nos plantea el giroscopio es que al no ser exacto cada vez que integramos se suma un error, este se conoce como deriva. Es algo que no se puede corregir sin la ayuda de otro sensor, por lo que solo con la integración no sera posible obtener la distancia recorrida por el ángulo en grados. La Figura **??** muestra la tabla de sensibilidad del giroscopio.

Figura 12: Sensibilidad de cada eje del acelerómetro

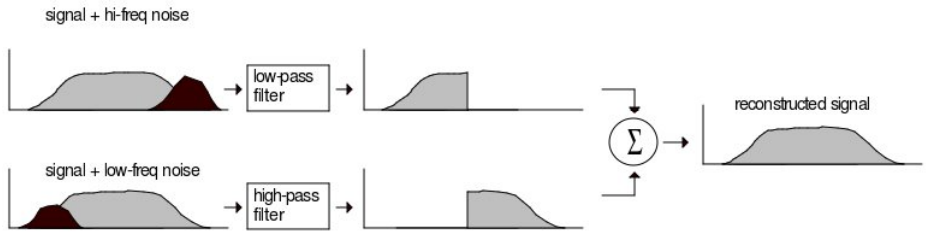
PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
GYROSCOPE SENSITIVITY						
Full-Scale Range	FS_SEL=0		±250		º/s	
	FS_SEL=1		±500		º/s	
	FS_SEL=2		±1000		º/s	
	FS_SEL=3		±2000		º/s	
Gyroscope ADC Word Length			16		bits	
Sensitivity Scale Factor	FS_SEL=0		131		LSB/(º/s)	
	FS_SEL=1		65.5		LSB/(º/s)	
	FS_SEL=2		32.8		LSB/(º/s)	
	FS_SEL=3		16.4		LSB/(º/s)	
Sensitivity Scale Factor Tolerance	25°C	-3		+3	%	
Sensitivity Scale Factor Variation Over Temperature	-40°C to +85°C		±0.04		%/°C	
Nonlinearity	Best fit straight line; 25°C		0.2		%	
Cross-Axis Sensitivity			±2		%	

Como pasa con el acelerómetro aquí también debemos calcular cual es la sensibilidad, se opta a utilizar la mayor sensibilidad de giro posible y por ello si calculamos $1/131 \Rightarrow 0{,}0076$ °/s por LSB. Para resolver el problema de la deriva se plantea utilizar el filtro complementario.

4.3. Filtro complementario

Usando solo el acelerómetro y el giroscopio podemos obtener el ángulo con bastante precisión pero el acelerómetro sigue siendo ruidoso y el giroscopio sigue afectado por la deriva. El filtro complementario es una técnica que utilizando dos filtros complementarios, la suma de de dichos filtros te devuelve una señal reconstruida que tiene el ruido reducido o puede no tener ruido. La Figura 13 muestra esto gráficamente.

Figura 13: Filtro complementario descrito gráficamente



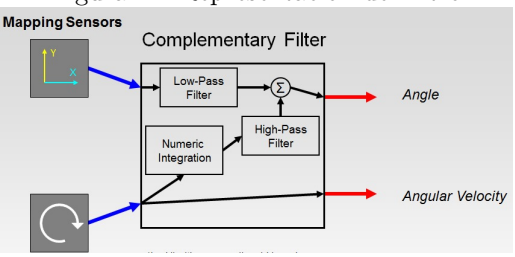
Las señales pueden tener mas o menos ruido, por lo que se escoge un valor de confianza, este valor de confianza determina cuanta información útil se suma a la señal resultante y debería estar contenida entre cero y uno. En el caso del péndulo se busca un valor de confianza tal que no dejemos pasar mucho ruido del acelerómetro pero sea suficiente como para corregir la deriva del giroscopio. La Formula 9 muestra como se implementaría este filtro, este corrige gradualmente la deriva del giroscopio de tal manera que si hay un pico de ruido en el acelerómetro este se vera reducido por el valor de confianza.

$$v = c \times g + (1 - c) \times a : c <= 1 \tag{9}$$

Se ha optado por un valor de confianza de 0,955, se ha llegado a este valor empíricamente dejando el péndulo en reposo y observando la variación del ángulo, esta variación sera el error cometido ya que físicamente no se tiene ningún cambio,

se observa que el error en reposo es de 0,1º con dicho valor de confianza. Otra manera de representar el filtro se puede observar en la Figura 14, el filtro complementario es una de las múltiples opciones disponibles para obtener el ángulo, entre las cuales destaca el filtro de Kalman, el cual toma parámetros del sistema como la inductancia, masa, fuerza de los motores, etc... obteniendo así el mejor filtro posible, el único impedimento es la matemática compleja de dicho filtro.

Figura 14: Representación del filtro



Esta opción aun descrita en esta memoria, tras su implementación, se observa cambios bruscos en el ángulo del péndulo cuando este esta en contacto con cualquier superficie. Esto se debe a que aunque sean los motores eléctricos estos crean una vibración considerable en la estructura, que hacen que el acelerómetro obtenga picos de aceleración. Otro factor decisivo para descartar este método es que la velocidad lineal en cualquier dirección afecta al resultado, impidiendo así el equilibrio del péndulo cuando este intenta equilibrarse moviéndose en cualquier dirección. La solución a este problema es escoger alguna técnica para representar la orientación de un objeto en el espacio al que no le afecten las velocidades lineales, este tipo de representación viene dada por los cuaterniones.

4.4. MotionApps 4.1

Jeff Rowberg es el creador de una librería ̈I2C Device Library"que se usa extensivamente en plataformas arduino, consiguió hacer ingeniería inversa sobre la MPU9150 pudiendo así modificar el código que ejecuta el procesador interno de la MPU. Su código esta abierto en Github[6] para que cualquiera lo pueda utilizar, este código permite obtener cuaterniones por I2C de una manera similar a leer los registros del acelerómetro o el giroscopio.

4.4.1. Cuaterniones

Los cuaterniones son una extensión de los números complejos, estos fueron descritos por el matemático William Rowan Hamilton en 1843, y se aplicaron a la mecánica en tres dimensiones. Esta técnica se utiliza mucho en el diseño gráfico por ordenador que resuelve problemas como el cierre de gimbal que afecta a los ángulos de euler. Las identidades de pueden observar en la Formula 10.

$$i^2 = j^2 = k^2 = ijk = -1 \tag{10}$$

Un cuaternion se puede representa como un pareja ordenada tal como se muestra en la Formula 11 donde v es un vector de tres elementos y s el la parte escalar.

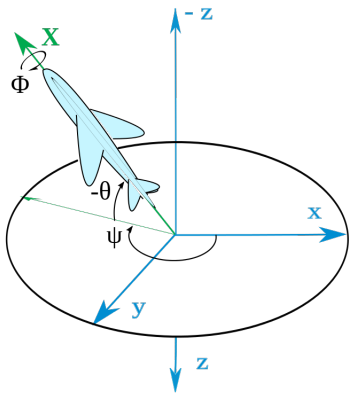
$$\begin{aligned} q &= q_0 + q_1 + q_2 + q_3 \\ q &= s + xi + yj + zk \ s, x, y, z \in \Re \\ q &= [s, xi + yj + zk] \ s, x, y, z \in \Re \\ q &= [s, v] \ s \in \Re, v \in \Re^3 \end{aligned} \tag{11}$$

A partir de los cuaterniones se puede obtener el ángulo de los tres ejes utilizando las formulas 12. Se utiliza atan2 ya el resultado de arctan solo te devuelve un resultado entre $[\frac{-\pi}{2}, \frac{\pi}{2}]$, lo que no proporciona todas las orientaciones posibles, mientras que atan2 devuelve un resultado entre $[-\pi, \pi]$, así obtendremos todas las orientaciones.

$$\begin{aligned} \phi &= atan2(2(q_0q_1 + q_2q_3), 1 - 2(q_1^2 + q_2^2) \\ \theta &= arcsin(2(q_0q_2 - q_3q_1)) \\ \psi &= atan2(2(q_0q_3 + q_1q_2), 1 - 2(q_2^2 + q_3^2) \end{aligned} \tag{12}$$

Estas orientaciones se pueden representan gráficamente en la Figura 15 aunque no tiene que coincidir con los ejes del péndulo ya que depende de la orientación del sensor.

Figura 15: Representación gráfica de los ángulos de giro



4.4.2. Obteniendo cuaterniones

Dependiendo de la posición del sensor nos interesaría obtener un eje o otro pero en nuestro caso queremos obtener el ángulo alrededor del eje x que seria ϕ debido a la orientación del sensor. En el programa la función atan2 nos devuelve el valor en radianes por lo que convertimos a grados por simplicidad. El sensor tiene una memoria volátil, esto nos obliga a tener que cargar el firmware que nos proporciona los cuaterniones cada ve que se enciende el sensor. Después de cargar el firmware por I^2C se utilizan interrupciones que proporciona el sensor para informar al microcontrolador de que se tienen datos preparados. Los datos se estructuran en un paquete formado en el sensor de 48 bytes de longitud que contiene varia información referente a los sensores y se le añaden los cuaterniones. El paquete es de 48 bytes de largo y se recoge de manera secuencial, leyendo

48 veces un registro específicamente marcado como registro FIFO que cada vez que se lee se rellena con el siguiente byte a leer. Cuando obtenemos el paquete se hacen las operaciones de desplazamiento de bits correspondientes para obtener la parte del paquete que nos interesa, en nuestro caso solo utilizaremos los cuaterniones para obtener los ángulos usando la Formula 12. La estructura del paquete se puede observar en la Figura 16.

Figura 16: Estructura del paquete FIFO

```
/* ============================================================================================== *
 | Default MotionApps v4.1 48-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][      ][MAG X ][MAG Y ][MAG 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  42  43  44  45  46  47  |
 * ============================================================================================== */
```

5. Protocolo de comunicación

Las comunicaciones del péndulo invertido se basan sobre el protocolo Bluetooth, esta comunicación nos servirá para obtener datos en vivo del péndulo. Este protocolo tiene un set de instrucciones muy reducido, este nos permite ajustar parámetros del péndulo invertido mientras se ejecuta el algoritmo teniendo el mínimo impacto en los cálculos usando prioridades.

5.1. Implementación del protocolo

El protocolo tiene un conjunto de comandos reducido para poder ajustar o obtener la información dentro del algoritmo de control. El Código 11 muestra la implementación del protocolo.

Codigo 11: Implementacion del protocolo

```
void parseProtocol(char * command){
    trim(command);
    char * arg;
    extern float kp;
    extern float ki;
    extern float kd;
    extern int state;
    extern float incl;
    extern float roll;
    extern float m1;
    extern float m2;
    extern float sat;
    extern float pid_in;
    extern int var;

    char show[10];
    if(!strcmp(command, "on")){
        state = ON;

        return;
    }
    if(!strcmp(command, "off")){
        state = OFF;

        return;
    }
    arg = strstr(command," ");
    arg[0]='\0';
    arg++;
    if(arg!=NULL){
        if(!strcmp(command,"print")){
            if(!strcmp(arg, "p")){
                var = 1;
                //printStr(show);
            }else if(!strcmp(arg, "i")){
                var = 2;
            }else if(!strcmp(arg, "d")){
                                var = 3;
                        }else if(!strcmp(arg, "a")){
                                var = 4;
                        }else if(!strcmp(arg, "e")){
                                var = 5;
                        }else if(!strcmp(arg, "es")){
```

```
                        var = 6;
                }else if(!strcmp(arg, "ea")){
                        var = 7;
                }
        }
        if(!strcmp(command,"in")){
             if(!strcmp(arg, "show")){
                  sprintf(show, "%f\n",pid_in);
                  printStr(show);
             }else{
                  pid_in=atof(arg);
             }
        }
        if(!strcmp(command,"a")){
             if(!strcmp(arg, "show")){
                  sprintf(show, "%f\n", roll);
                  printStr(show);
             }else{
                  incl=atof(arg);
             }
        }
        if(!strcmp(command,"sat")){
             if(!strcmp(arg, "show")){
                  sprintf(show, "%f\n", sat);
                  printStr(show);
             }else{
                  sat=atof(arg);
             }
        }

        if(!strcmp(command,"m1")){
             if(!strcmp(arg, "show")){
                  sprintf(show, "%f\n", m1);
                  printStr(show);
             }else{
                  m1=atof(arg);
             }
        }

        if(!strcmp(command,"m2")){
             if(!strcmp(arg, "show")){
                  sprintf(show, "%f\n", m2);
                  printStr(show);
             }else{
                  m2=atof(arg);
             }
        }

        if(!strcmp(command,"p")){
             if(!strcmp(arg, "show")){
                  sprintf(show, "%f\n", kp);
                  printStr(show);
             }else{
                  kp=atof(arg);
             }
        }
        if(!strcmp(command,"i")){
             if(!strcmp(arg, "show")){
                  sprintf(show, "%f\n", ki);
                  printStr(show);
             }else{
                  ki=atof(arg);
```

```
            }
        }
        if(!strcmp(command,"d")){
            if(!strcmp(arg, "show")){
                sprintf(show, "%f\n", kd);
                printStr(show);
            }else{
                kd=atof(arg);
            }
        }

        if(!strcmp(command,"echo")){
            printStr(arg);
        }
    }

}
void trim(char *input){
    int len;
    int i;
    len = strlen(input);

    for(i=0;i<len && input[i] == ' ';i++){
        input++;
    }

    for(i=len-1;i>0 && input[i] == ' ';i--){
        input[i]='\0';
    }


}
```

En el código antes mencionado, tenemos la función trim, esta función sirve para borrar los espacios antes y después del string que se le pasa como argumento. Tenemos dos buffers, uno para guardar la información que se va a enviar, y otro para que contiene el argumento, todas las demás variables son externas y han de estar declaradas fuera del protocolo y ser declaradas como extern en el protocolo para poder ser modificadas desde el protocolo. Los dos primeros comandos que se pueden utilizar son 'on' y 'off' para controlar el bucle de control, estos atrapan la ejecución dentro de un bucle de espera activa a través de la variable state. Si no es ninguno de estos comandos tenemos que comprobar hay algún argumento, lo obtenemos cambiando el espacio que separa el comando del argumento por una terminación de linea. Para obtener datos en vivo podemos usar el comando "print x"donde x es la variable que quieres consultar, podemos consultar las variables en la Tabla 2

Tabla 2: Comando print para obtención de datos

Control Proporcional	p
Control Integral	i
Control Derivativo	d
Control PID	a
Error actual	e
Error actual con signo	es
Error acumulado	ea

Esto cambia una variable externa que se comprueba en el bucle de control principal y proporciona un flujo estable de datos numéricos que podemos procesar en el otro lado, en este caso un ordenador conectado por bluetooth. Se han usado números enteros para simplificar la programación, estos se pueden consultar en el código antes mencionado. También podemos cambiar parámetros a través de nombres de variables, estos nombres y su significado en el bucle de control que se explica en la Tabla 3.

Tabla 3: Comandos para modificar o consultar parámetros de del bucle de control

in	Ángulo de funcionamiento de la parte integral de algoritmo PID
a	Inclinación del péndulo
sat	Saturación del error acumulado
m1	Constante proporcional del motor 1
m2	Constante proporcional del motor 2
p	Constante proporcional del algoritmo PID
i	Constante integrativa del algoritmo PID
d	Constante derivativa del algoritmo PID

Cualquiera de las variable se puede ajustar tanto como consultar de esta manera "{nombre_variable} {[float valor — string show]}", a excepción de la variable .ª"que nos mostrara el error en el instante actual. Esto nos permite recordar cuales fueron los últimos parámetros introducidos en el algoritmo. También se dispone de la función echo que te devuelve lo que le mandas.

5.2. Representación de los datos

En esta sección vamos a hacer uso de Processing, una API basada en Java para poder representar los datos que recibimos a través de Bluetooth. El código esta pensado para ser utilizado sobre Ubuntu 14.04. Lo primero es encender el péndulo, para poder escribir sobre los dispositivos necesitamos permisos de root y modificar unos ficheros para poder conectarnos por bluetooth. Para poder utilizar dichos comandos tenemos que conocer la MAC del dispositivo bluetooth desde el cual queremos conectarnos. Esto se puede hacer con "hciconfig". En mi caso hci0 tiene la MAC 00:15:83:CA:1C:B3, Ahora que tenemos nuestra MAC necesitamos la del dispositivo al cual queremos conectarnos, en el caso del péndulo 00:14:02:27:58:44, para conectarnos necesitaremos saber el pin de

conexión, se ha configurado en este caso para ser 1989 a través del modo AT del dispositivo Bluetooth. El código 12 muestra todos los comandos para configurar el bluetooth. Primero navegamos a la carpeta correspondiente al nuestro dispositivo bluetooth, allí crearemos el fichero pincodes si no esta creado y introduciremos la MAC del dispositivo destino y el pin que e utilizara para la conexión.

Codigo 12: Configuracion del Bluetooth en Ubuntu 14.04

```
> cd /var/lib/bluetooth/00\:15\:83\:CA\:1C\:B3
> echo "00:14:02:27:58:44 1989" >> pincodes
```

Cuando tenemos todo preparado podemos conectarnos usando el comando de Linux que se muestran en el Código 13.

Codigo 13: Comandos de conexion

```
> rfcomm connect 0
```

En este instante se ha creado un dispositivo en la ruta /dev/rfcomm0 al cual root puede escribir y leer usando cat y tee como se muestra en el código 14. Cada uno de los comandos es preferible utilizarlo en diferentes terminales, uno para rfcomm otro para tee y otro para cat.

Codigo 14: Comandos de escritura y lectura sobre el dispositivo

```
> cat /dev/rfcomm0
> tee /dev/rfcomm0
```

En nuestro caso no vamos a utilizar el comando cat, vamos a utilizar Processing, el código 15 muestra el programa escrito usando Processing para obtener la acción proporcional. Para obtener los demás datos el programa es igual, solo hay que cambiar ciertos parámetros y tendremos los datos que nos interesan.

Codigo 15: Codigo de Processing para accion proporcional

```
import processing.serial.*;
import grafica.*;
Serial myPort;
int npoints=1000;
int i=0;
int lf = 10;
int save=0;
float lastFloat = 0;
float time = 0;
float time_inc = 0.02;
GPlot plot;
GPointsArray points;
void setup(){
   size (783,484);
   plot = new GPlot(this);
   plot.setPos(0,0);
   plot.setDim(683,384);
   points = new GPointsArray(npoints);
   plot.getXAxis().setAxisLabelText("Tiempo (s)");
   plot.getYAxis().setAxisLabelText("Error (°)");
   plot.setTitleText("Accion proporcional");
```

```
  String portName = "/dev/rfcomm0";
  myPort = new Serial(this,portName,9600);
  println(myPort);
    while(i<npoints){
       if(0<myPort.available()){
         String instr = myPort.readStringUntil(lf);
         if(instr == null){continue;}
         if(instr.length()==0){continue;}
         float inFloat = float(instr);
         if(Float.isNaN(inFloat)){continue;}
         points.add(time,inFloat*10);
         i=i+1;
         time=time+time_inc;
       }

    }
    plot.setPointSize(0.5);
    plot.setPoints(points);
    plot.defaultDraw();
    save("p.png");
    save++;
    i=0;
}
```

Processing se basa en eventos, la función setup es llamada automáticamente cuando se inicia el programa. En setup definimos el nombre del puerto serie que se va a utilizar, y creamos un objeto Serial usando este nombre y el baud rate deseado en el puerto. También se ha utilizando una librería externa de Processing denominada ”grafica”, esta se utiliza para simplificar el proceso de graficado con Processing. Cuando tenemos declarados los objetos necesarios como la grafica (GPlot) y el array de puntos (GPointsArray) debemos recoger los datos por el puerto serie. Para recoger los datos por el puerto serie se deben encolar los datos hasta que se encuentre una nueva linea, es entonces cuando se recogen los datos y se añaden al array. El array esta limitado a 1000 puntos, esto nos proporciona suficiente información para procesar 20 segundos de ejecución. Este después de grafica y se guarda bajo el nombre deseado. Usando este simple programa hemos obtenido las Figuras 17, 18, 19, que son la acción proporcional, integrativa y derivativa respectivamente.

Figura 17: Acción proporcional

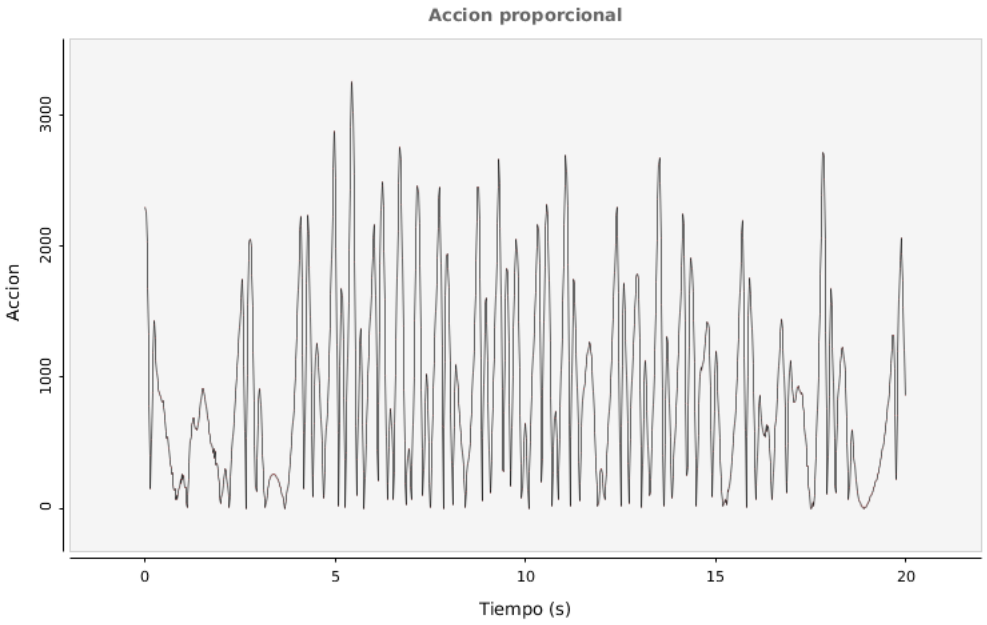


Figura 18: Acción integrativa

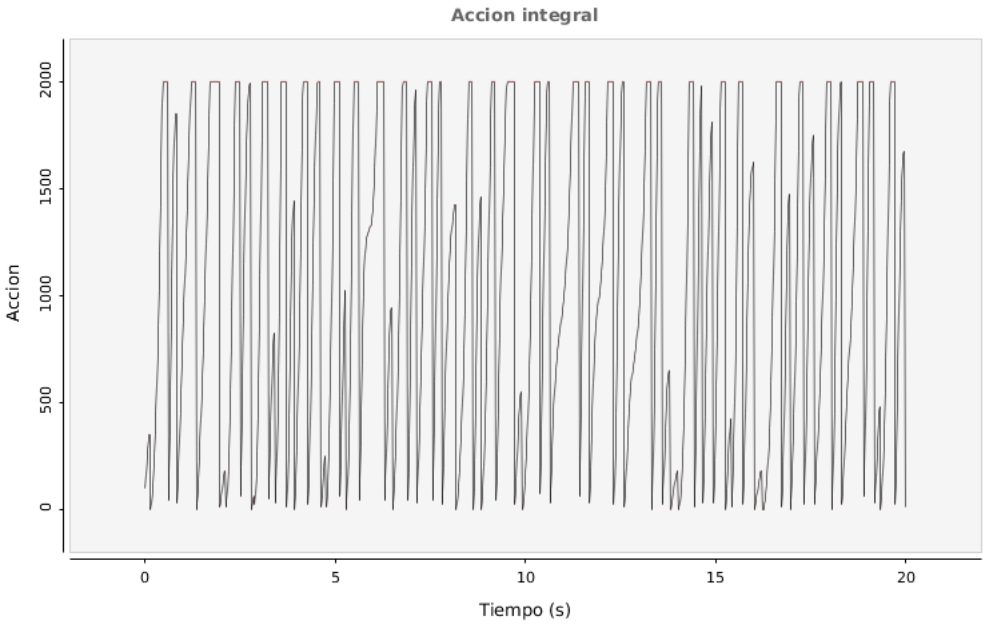
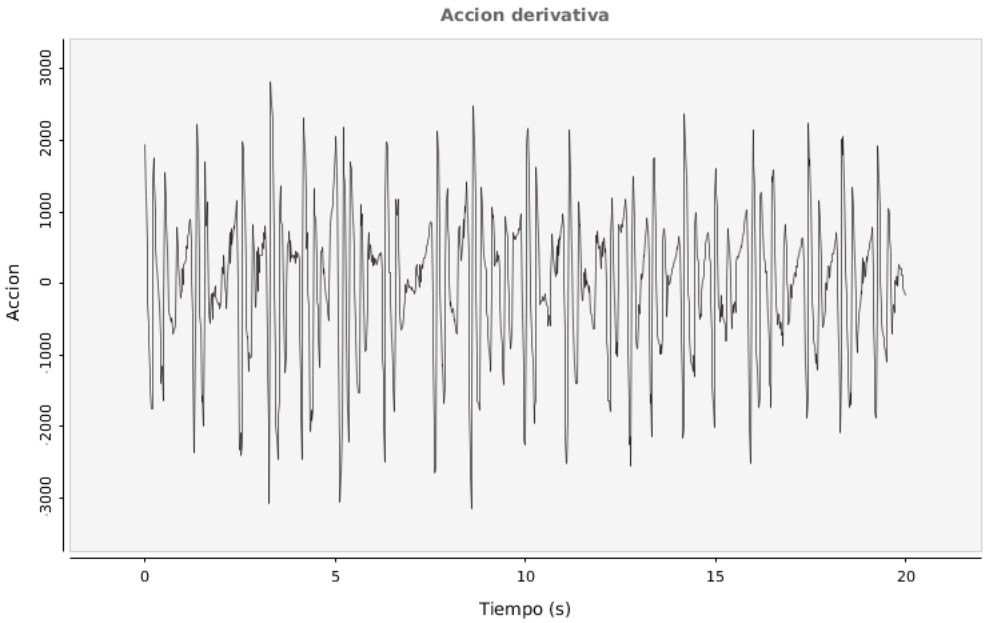


Figura 19: Acción derivativa



Se observa la saturación de la acción integrativa, también podemos ver que existe un pequeño ruido en la acción derivativa, este ruido no tiene un impacto significativo en el sistema. En las Figuras 20, 21 se observa la acción combinada que se envía a los motores y el error con signo respectivamente.

Figura 20: Acción combinada

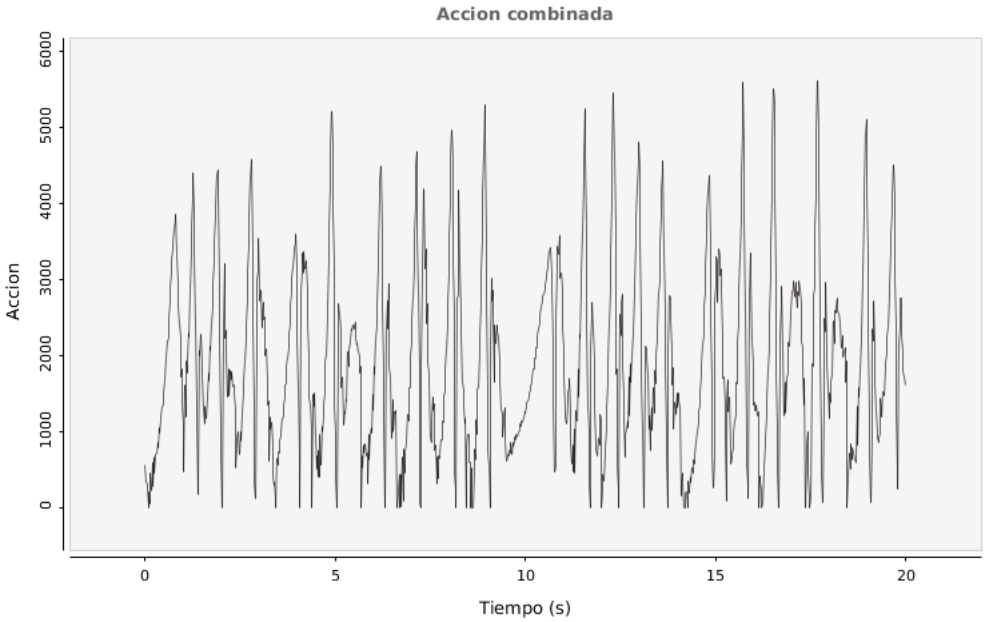
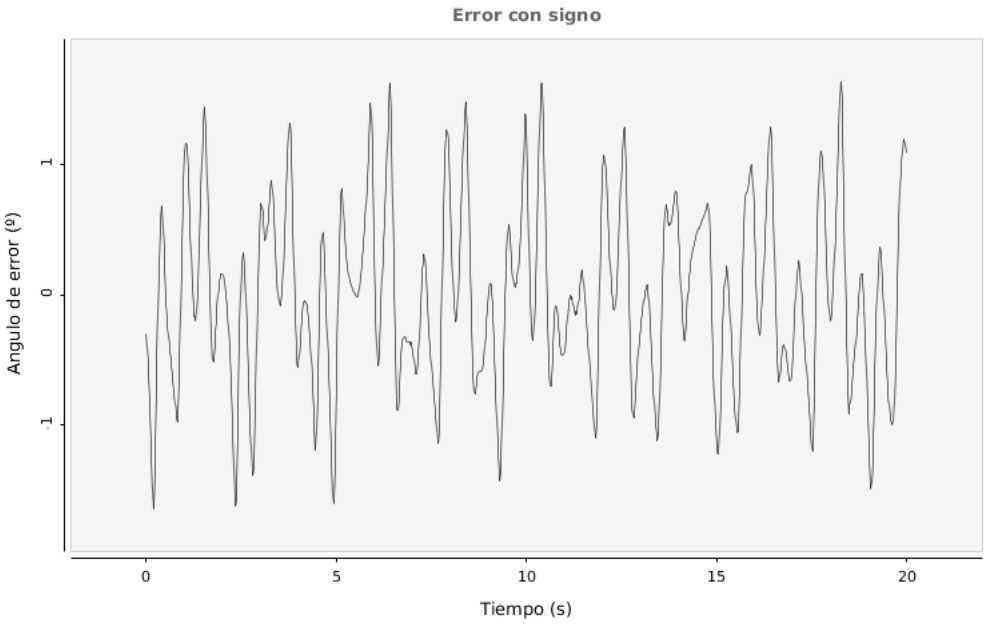


Figura 21: Error con signo



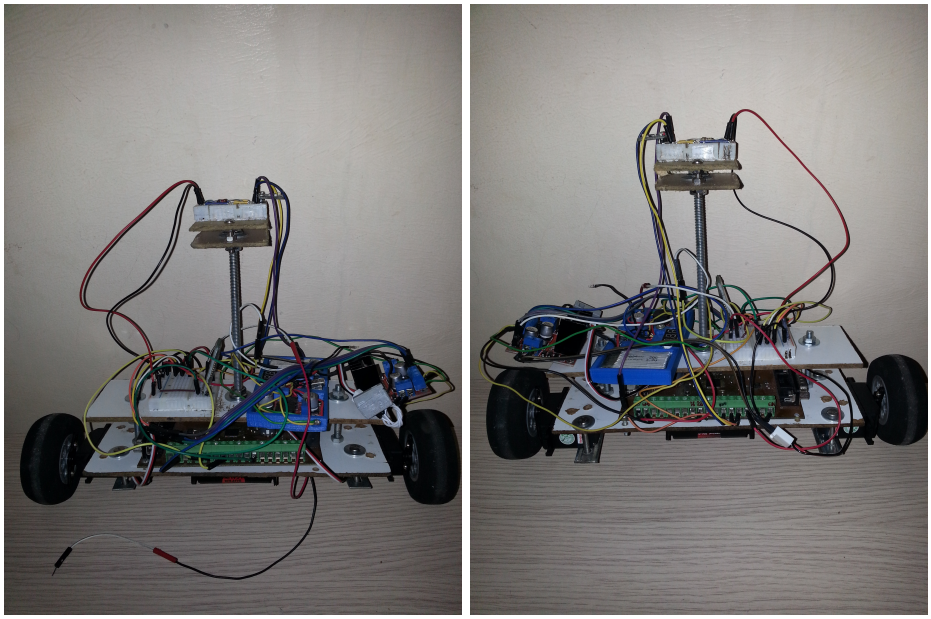
6. Péndulo invertido

El péndulo invertido es un problema clásico de control, se basa en mantener un polo en equilibrio opuesto al de un péndulo normal es decir en contra de la gravedad. El sistema tendrá como entrada un acelerómetro y un giroscopio, ellos marcaran el ángulo en el que se encuentra el péndulo, con la información del ángulo que nos proporciona el sensor el algoritmo PID puede obtener el error de ángulo y tomar una acción de control. Como se menciono en secciones anteriores la primera implementación del péndulo, es decir con el filtro complementario tenia problemas, se obtenían bruscas vibraciones al intentar estar en equilibrio.

6.1. Estructura

La estructura se compone por varios componentes, la base son dos laminas de madera, la inferior de 8 de ancho y 20 de largo y la superior 6,5 de ancho y 20 de largo, que se unen con dos espárragos de metal para que se puedan enroscar las tuercas por ambos lados y sujetar las bases una encima de otra. Encima de la base superior se ha hecho lo mismo con otro esparrago que en el lado superior del mismo se encuentra el sensor sobre una placa de prototipado. En la base inferior, por debajo tenemos los dos motores, uno a cada lado, en la parte superior de la misma base tenemos la placa del microcontrolador. En la placa superior se encuentra el puente en H donde se conectan los motores y a su lado la batería que los alimenta, en el lado opuesto de la misma base esta el bluetooth. La Figura 22 muestran fotos de la configuración del péndulo.

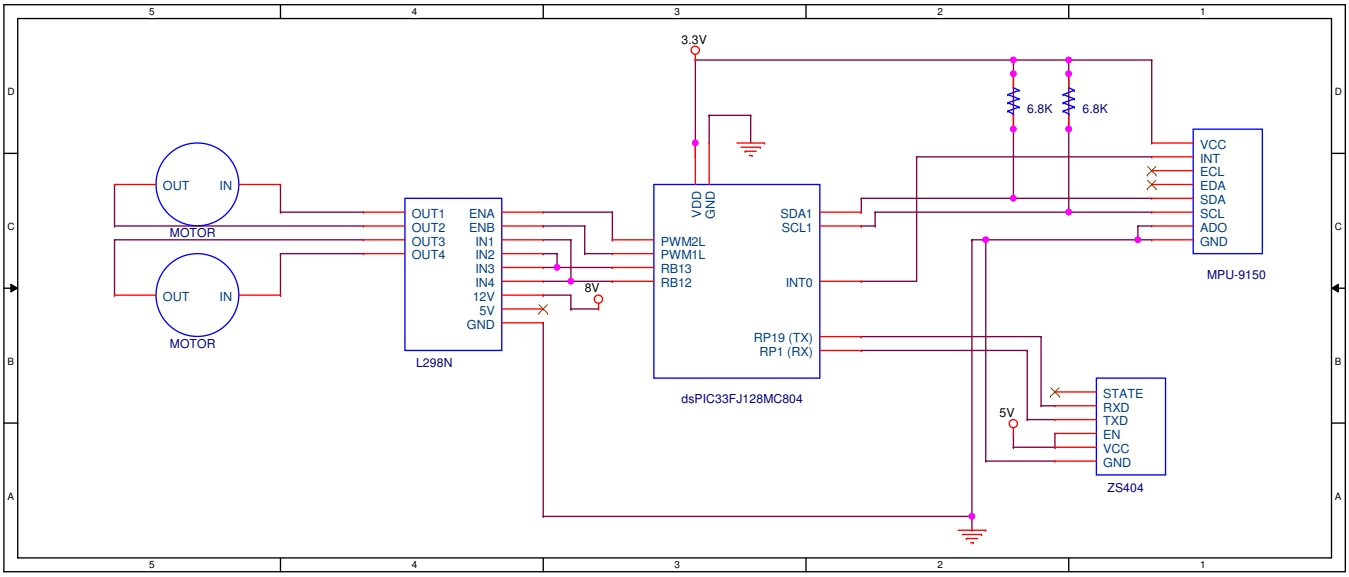
Figura 22: Configuración de la estructura del péndulo



6.2. Conexiones

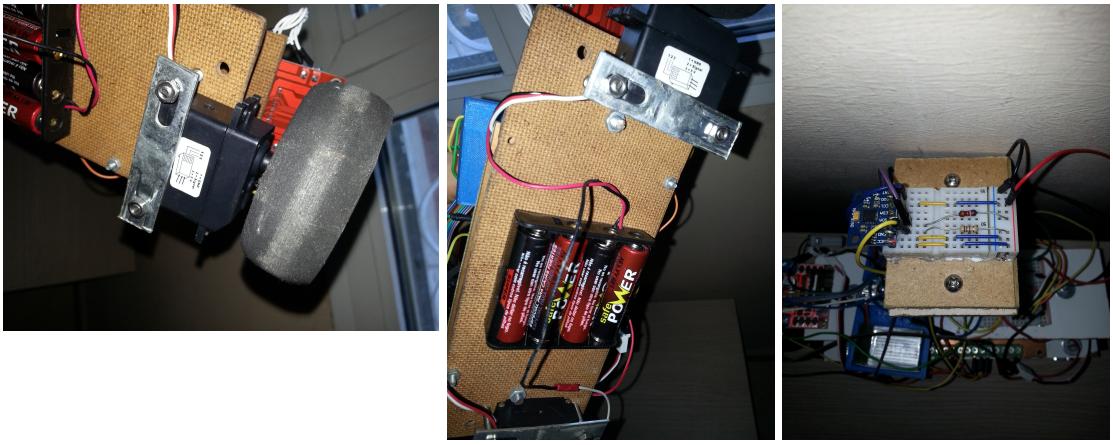
Esta sección describe las interconexiones del dispositivo con sus periféricos. En la Figura 23 se muestra el esquema de conexiones del péndulo.

Figura 23: Interconexiones del péndulo



En el centro del esquema se observa el dsPIC33F (con los pines no usados omitidos) y como están hechas las conexiones. A la izquierda se puede ver el chip L298N que es el puente en H, este esta conectado a dos pines de salida RB12 y RB13 que controlan la dirección de rotación del motor, mientras que la lineas conectada a ENB y ENA son las que emiten la señal PWM por separado. Estos están conectados internamente dentro del L298N a las salidas OUT que alimentan los motores desde la batería. En la parte superior derecha del esquema se puede ver como las lineas SDA y SCL se conectan a través de resistencias de pull-up de 6.8KΩ. La linea marcada como ADO es la que determina el menor de los bits de la dirección I^2C y al estar conectado a tierra la dirección acaba en un cero lógico, 0x67. Los pines RP19 y RP1 están remapeados a la transmisión y recepción de datos por serie con el modulo UART que se conectan al dispositivo Bluetooth ZS404. Cada chip se alimenta con diferentes voltajes, la alimentación de la placa, el modulo bluetooth y el sensor vienen de 4 pilas de 1,5V en serie lo que nos proporciona 6V teóricos, aunque la entrada llega a estabilizarse casi a 5V. El modulo bluetooth esta alimentado directamente desde las pilas en serie, mientras que la placa y los sensores pasan por un amplificador operacional para reducir el voltaje a 3.3V que es lo que admite la placa y el sensor. Los motores como ya se ha comentado en secciones anteriores viene alimentada por una batería de alta descarga, esta esta conectada al puerto que admite hasta 12V en el puente en H. El voltaje de salida de la batería se puede regular, esta emite 8V se ha optado por este voltaje para no sobrepasar el voltaje que pueden soportar los motores. En la Figura 24 se muestra el péndulo desde varios ángulos.

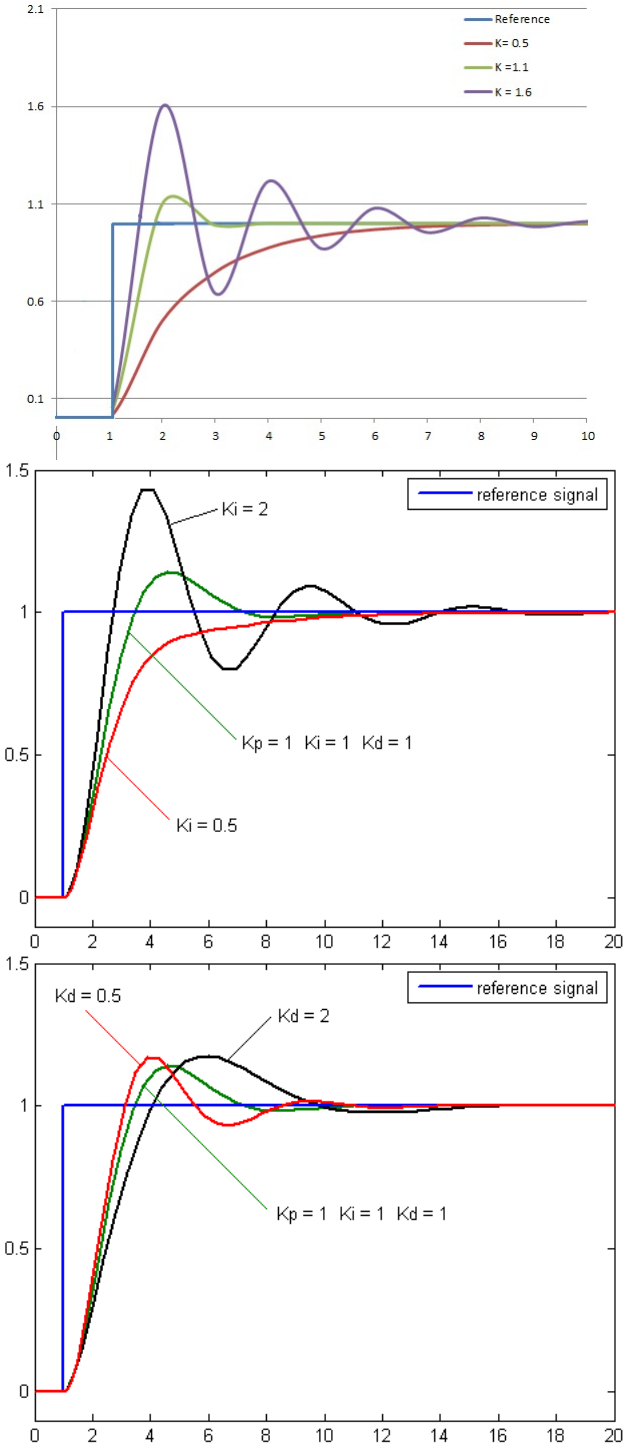
Figura 24: Componentes adicionales del péndulo



6.3. Algoritmo PID

En la teoría de control un controlador es un dispositivo que monitora y altera la condiciones operacionales de un sistema dinámico. Ejemplos clásicos del control pueden ser controla la temperatura dentro de un espacio como un refrigerador, controlar la velocidad de giro de una turbina hidráulica, etc. El nombre viene dado por sus variables en ingles "Proportional-Integrative-Derivative Controller", cada una asiste al controlador de manera diferente. La parte proporcional como su nombre indica es directamente proporcional al error en un instante dado, cuanto mas aumente este parámetro los cambios en la salida serán mas grandes y llega a desestabilizar el sistema. En sistemas como control de velocidad solo modificando la parte proporcional no es suficiente para llegar al valor deseado, aquí es donde entra la parte integrativa, esta puede ser definida en un rango temporal, saturada o total. La parte integrativa suma el error cometido durante un tiempo dado del algoritmo y al multiplicada por la constante K_i es la responsable de asegurar que la señal de referencia y la señal de salida llegan a coincidir, esto resolverá el problema de la parte proporcional, pero a su vez introduce otro problema, denominado en ingles Ïntegrative Windup"que se resuelve saturando el resultado de dicha operación. La parte integrativa también añade un tiempo de respuesta mas reducido a la parte proporcional por lo que se llega antes al resultado deseado pero con una constante integral muy grande produce un efecto conocido como .ºvershoot". La parte derivativa se añade para suavizar la respuesta de la parte integrativa, esta intenta anticiparse al resultado siguiente tomando datos de instantes anteriores, así obteniendo la tendencia del error en el instante actual. A diferencia de las demás esta puede devolver un resultado negativo y este resultado negativo es lo que evita que exista un .ºvershoot.[en] la respuesta. La Formula 13 describe matemáticamente la suma de las tres partes que es conocido como controlador PID. La Figura 25 muestra gráficamente el impacto de cada variable dependiendo del valor de sus constantes.

Figura 25: Impacto de las constantes PID



$$u(t) = K_p e(t) + K_i \int_0^t *e(\tau)d\tau + K_d \frac{d}{dt} e(t) \tag{13}$$

Teniendo las bases del algoritmo PID definimos cada constante y su significado en este caso. $u(t)$ es la acción a tomar por los motores en el tiempo t, K_p es la constante proporcional del algoritmo que se multiplica por el error actual del algoritmo, el error es la resta entre el ángulo deseado y el ángulo actual. K_i es la constante que se multiplica por el error acumulado durante toda la ejecución del algoritmo, en este caso se acumulan grado de error, cuanto tiempo ha estado el péndulo desequilibrado, este ayudara a llegar al ángulo deseado con exactitud pero su respuesta tiene que ser rápida y se debe utilizar saturación, si no al estar mucho tiempo desequilibrado en un lado al volver al llegar al ángulo deseado la parte integrativa ha acumulado tanto error y que desequilibra el péndulo hacia el otro lado. K_d es la constante derivativa del algoritmo y se multiplica por la tendencia del error entre dos instantes de tiempo, para nosotros no es mas si el péndulo esta moviéndose en la dirección opuesta o no del ángulo deseado. Para obtener el error se utiliza el ángulo calculado a partir del cuaternion que nos devuelve el sensor. El punto de equilibrio del péndulo esta desviado y no es exactamente opuesto a la gravedad por tanto el error es la resta del ángulo real menos el de equilibrio.

6.3.1. Saturación integrativa

La parte integrativa acumula el error durante la ejecución del algoritmo, el error acumulado se multiplica por la constante integrativa para actuar sobre los motores. Esto nos plantea varios problemas, el error acumulado en teoría solo es creciente, en nuestro caso el error acumulado puede ser tanto en una dirección como en otra del ángulo deseado. Para resolver este problema utilizaremos el error con signo, el signo nos muestra en que lado esta el péndulo sobre el punto de equilibrio (e.g. +5º o -5º con respecto al punto de equilibrio), esto nos permitirá tener un error acumulado que se acumula en ambas direcciones y al sumarse con signo puede crecer o decrecer. Otro problema que se nos plantea es el crecimiento incontrolado del error acumulado (Ïntegrative Windup"), esto tiene un efecto negativo sobre el punto de equilibrio llegando a desestabilizarlo cuando el error acumulado es tan grande que no llega a reducirse a tiempo, esto se resuelve poniendo a cero el error acumulado si se da la situación de que se ha cambiado de dirección, esto nos proporciona una reacción mas rápida del termino integrativo. Por otra parte también se opta por saturar el error acumulado, así podemos tener valores muy pequeños de acción que cambian con gran rapidez proporcionado una respuesta casi instantánea.

6.4. Bucle de control

El bucle de control es el responsable de obtener el estado del sistema y actuar en consecuencia, este también implementa parte de las comunicaciones cuando no se están haciendo cálculos. El Código 16 muestra la implementación.

Codigo 16: Bucle de control

```
void delay_ms(unsigned long x){
```

```
        if(x>1000){
                __delay32((x/1000)*FCY);
        }else{
                __delay32((x*FCY)/1000);
        }

}

int state = ON;
int x=0;
int interrupcionMPU = 0;
int intr=0;
float pid_p = 0;
float pid_d = 0;
float pid_i = 0;
float pid_pid = 0;
int fifoCount=0;
int fifoBuffer[128];
float kp = 0;
float ki = 0;
float kd = 0;
float incl = 4.98; // 4.92
int aux;
int var;
float pid_in;
int strength1=0;
int strength2=0;
float err=0;
float sat = 0.04; //0.38;
float err_s=0;
float err_ant=0;
float err_acc=0;
float m1 = 1;
float m2 = 1;
int printed=true;
int quat_data[4];
float q[4];
float roll=0;
char direction;

float control_p(){
    return err*kp;
}

float control_d(){
    float de = err-err_ant;
    return kd*de/SAMPLING_RATE;
}

float control_i(){
        return ki*fabs(err_acc);

}

int main (void)
 {
    printStr("Initializing...\n");
    UARTinitialize();
    Enable_i2c();
    delay_ms(100);
    SetupMPU_i2c();
    PWM1initialize();
```

```
PWM2initialize();
initMotor();
setTimer1(ACTUAL_PTPER);
setDutyCycle1(ACTUAL_PTPER);
setTimer2(ACTUAL_PTPER);
setDutyCycle2(ACTUAL_PTPER);

pid_in = 0.05;
kp = 1750; //1750;//4100;//1100;//1600; //825
kd = 35; //78; //800;//2; //40000
ki = 50000; //6350; //1100;//1200; //5000
resetFIFO();
setDMPEnabled(true);
int dmpPacketSize = 48;

while(1){
    while (!interrupcionMPU && fifoCount < 48){
        err_ant = err;
        err_s = incl - roll;
        err = fabs(incl - roll);
        if(roll>=incl){
            if(direction=='f') err_acc=0;
            direction='b';
        }else if(roll<incl){
            if(direction=='b') err_acc=0;
            direction='f';
        }
        err_acc += err_s*SAMPLING_RATE;
        if(err_acc>sat) err_acc = sat; else
        if(err_acc<-sat) err_acc = -sat;
        pid_p = control_p();
        pid_d = control_d();
        pid_i = control_i();
        pid_pid = pid_p + pid_d + pid_i;
        pid_pid = max(0,pid_pid);
        pid_pid = min(ACTUAL_PTPER*2,pid_pid);
        x = (int)pid_pid;
        strength1 = (int) (ACTUAL_PTPER*2)-x*m1;
        strength2 = (int) (ACTUAL_PTPER*2)-x*m2;
        motor(direction,strength1,strength2);
        enableUARTInt();
        while(!interrupcionMPU){
            if(printed==false){
                if(var==1){ //p
                    aux = (int) pid_p/10;
                }else if(var==2){ //i
                    aux = (int) pid_i/10;
                }else if(var==3){ //d
                        aux = (int) pid_d/10;
                }else if(var==4){ //a
                        aux = (int) pid_pid/10;
                }else if(var==5){ //e
                        aux = ceil(err*100);
                }else if(var==6){ //es
                        aux = ceil(err_s*100);
                }else if(var==7){ //ea
                        aux = ceil(err_acc*1000);
                }
                if(var>0){
                        printInt(aux);
                        printStr("\n");
                        printed=true;
```

```
                    }
                }
            }
            printed=false;
            while(state == OFF){
                if (state == OFF){
                    motor('b',(ACTUAL_PTPER*2),(ACTUAL_PTPER*2));
                }

            }
            disableUARTInt();

        }
        interrupcionMPU = 0;
        intr = getIntStatus();
        fifoCount = getFIFOCount();
        if((intr & 0x10 ) == 16 || fifoCount >= 1024){
                printStr("OVERFLOW!\n");
                resetFIFO();
        }else{
            if((intr & 0x01) == 1){
                while(fifoCount<dmpPacketSize){
                    fifoCount = getFIFOCount();
                }
                getFIFOBytes(fifoBuffer, dmpPacketSize);
                fifoCount -= dmpPacketSize;
                dmpGetQuaternion(quat_data, fifoBuffer);
                q[0] = quat_data[0] / 16384.0;
                q[1] = quat_data[1] / 16384.0;
                q[2] = quat_data[2] / 16384.0;
                q[3] = quat_data[3] / 16384.0;
                roll = atan2(   (2*(q[0]*q[1] + q[2]*q[3])),
                                (1-2*(pow(q[1],2)+pow(q[2],2))));
                roll = roll * (180/3.1415);
            }
        }
    }
    return 0;
}
```

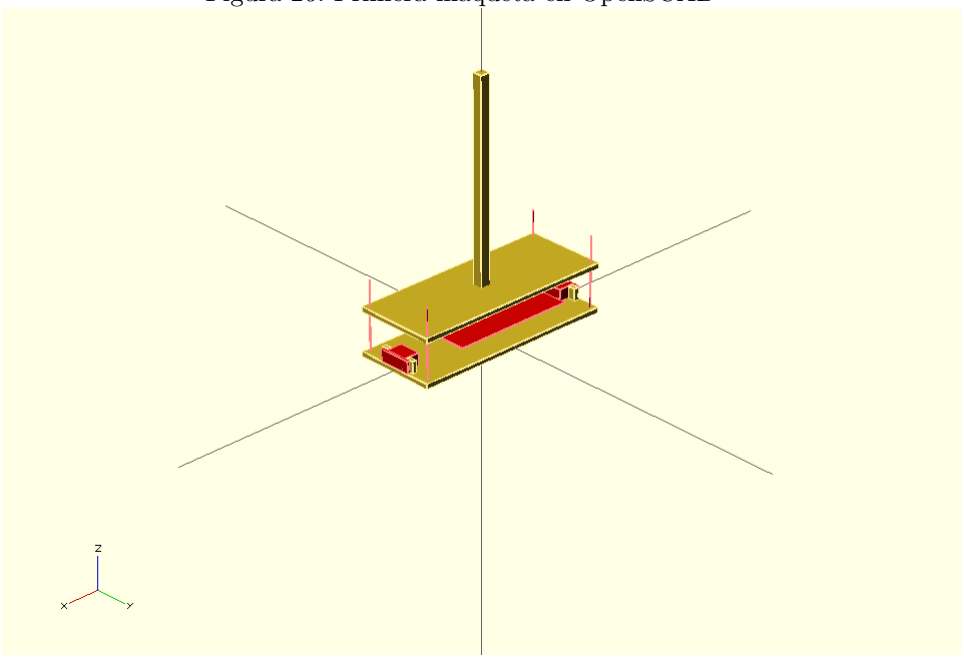
Lo primero que se implementa es una función de retardo, esta se basa en otra función de retardo que se basa en ciclos de reloj, si sabemos que una instrucción son dos ciclos de reloj, FCY representa esta frecuencia de ejecución basándose en la frecuencia del oscilador, en otras palabras n instrucciones por segundo. Por que los registros son de 16 bits y para no perder precisión las operaciones hasta y desde mil milisegundos son diferentes. En el bucle de control se implementan las funciones de control PID por separado, se utilizan variables globales para poder ser utilizadas desde cualquier punto del programa y poder ser modificadas por el protocolo de comunicaciones que se ha implementado anteriormente. Para el control proporcional tenemos el error multiplicado por la constante proporcional, para el control derivativo debemos guardar el error en el instante anterior por ello tenemos la variable err_ant para ayudarnos, cuando tenemos la diferencia del error del instante anterior dividimos por el tiempo entre las dos medidas que nos devolverá la derivada del error y esta se multiplica por la constante derivativa, la parte integrativa es el error acumulado multiplicado por la constante integrativa. El bucle de control comienza con unas rutinas de inicialización, estas nos sirven para poner todos los módulos en marcha, reescribir el firmware del sensor y ajustar los parámetros PID. Por prueba y error hemos obtenido los valores de

las constantes, estos valores se guardan en las variables kp, kd y ki. Después de inicializar todos los módulos se comienza a iterar sobre el bucle de control, este esta acotado a una frecuencia de muestreo, esta esta definida por la velocidad a la que llegan los datos del sensor y es de 50Hz lo que nos proporciona un sampling rate de 0.02 segundos. El bucle de control se divide en dos partes, la primera se compone por un bucle que hace los cálculos necesarios cuando no hay datos disponibles, si hay datos sale de dicho bucle de calculo y recoge los datos, esto se controla con la variable interrupcionMPU. Dentro del bucle de calculo se obtiene el error con y sin signo, el error con signo sirve para saber a que dirección estamos cayendo, si la dirección ha cambiado el error acumulado pasa a ser cero, y se vuelve a acumular. Luego se comprueba que el error acumulado no supera el nivel de saturación, la saturación es de 0.04 grados, esto hace que la ki sea muy grande para que la acción se note en los motores. cuanto mas pequeña la menos acción se le da a los motores, pero al aumentar la i mas rápida sera la acción. Cuando tenemos calculado el error acumulado se pasa a calcular la acción PID a través de las funciones antes descritas, saturamos también la acción de los motores dependiendo de lo que admita el modulo PWM, también es posible modificar a través de m1 y m2 la fuerza con la que cada motor actúa en comparación con el otro, esto se implemento en un principio por que los motores iniciales (Towerpro SG90) eran diferentes en los engranajes, y eso hacia que la respuesta de ambos no fuese la misma, esto puede usarse también para implementar el giro del péndulo en la dirección deseada. Cuando hemos calculado y mientras no tengamos ningún dato disponible podemos hacer uso de las comunicaciones por lo que se activa las interrupciones UART, solo se imprime una vez por vuelta y esperamos en un bucle de espera activa a que ocurra otra interrupción externa, cuando ocurre se vuelven a desactivar las interrupciones UART. La segunda parte del bucle de control es la de recoger los datos por I^2C cuando ocurre una interrupción externa. Cuando hay una interrupción debemos comprobar de que tipo es leyendo cierto registro del sensor. Esto nos puede decir si se ha dado un overflow o si los datos están preparados, el overflow ocurre cuando el buffer de 1024 del sensor se llena. En caso de que sean que los datos están preparados se lee el registro FIFO 48 veces para recoger el paquete FIFO, cuando ya hemos recogido el paquete podemos utilizar desplazamiento de bits para recoger los datos de los cuaterniones, para convertirlos a un valor han de ser divididos por 16364, así obtenemos un valor real, a partir de ahí usando atan2 convertimos uno de los ejes a ángulo de euler y después a grados con signo. Con estos datos podemos volver a calcular la acción de los motores volviendo al bucle de calculo hasta que se vuelva a tener nuevos datos.

7. Conclusión

No todos los objetivos del proyecto has sido cumplidos, la dificultad de hacer un proyecto sin conocimiento previo de la programación de microcontroladores ha sido un gran reto de auto aprendizaje. Los conocimientos adquiridos durante la realización del proyecto han sido extensos, desde la programación de microcontroladores, nuevos conceptos de los controladores PID, comunicaciones entre módulos, control de motores y muchos mas. Los problemas a los que me he tenido que enfrentar vienen por mi desconocimiento del tema, aun se puede mejorar mas el péndulo haciendo una estructura impresa en un 3D, esto obligaría a volver a buscar nuevos parámetros PID ya que el peso de la estructura cambiaría. También otro objetivo no cumplido es la realización de una nueva placa integrada para que integrase todos los componentes en la misma y no hacer uso de breadboards. La estructura se habría hecho usando el programa OpenSCAD para dibujar todos los componentes por separado y luego unirlos usando tornillos o otro tipo de adhesivo para plásticos. Una primera maqueta de la posible realización del péndulo fue creada al principio del proyecto y es la que se muestra en la figura 26

Figura 26: Primera maqueta en OpenSCAD



Se habría utilizado el programa Orkad o Eagle para crear la placa que integrase todos los componentes, por no tener experiencia en diseño de placas usando programas de modelado no ha sido posible la realización de dicha parte del proyecto. Otro programa que no se menciona en el proyecto es Proteus, este programa sirve para realizar simulaciones bastante exactas del microcontrolador, esto ha servido a modo de aprendizaje para ver como funcionan los módulos sin tenerlos físicamente, la única desventaja de este programa es que no contiene todos los modelos de microcontroladores y se debe escoger uno lo

mas parecido posible para poder hacer pruebas con el. Este proyecto a sido un gran lección de aprendizaje para introducirme a mi mismo en el mundo de los microcontroladores y sistemas empotrados, algo que no se enseña en tanto detalle en la universidad.

Referencias

[1] Javier Graciá Carpio. *Repositorio de Grafica*. URL: https://github.com/jagracar/grafica.

[2] Debra. *Gyroscopes and Accelerometers on a Chip*. URL: http://www.geekmomprojects.com/gyroscopes-and-accelerometers-on-a-chip/.

[3] Invensense. *Ficha tecnica del sensor MPU9150*. URL: http://store.invensense.com/datasheets/invensense/MPU-9150_DataSheet_V4%203.pdf.

[4] Microchip. *Ficha tecnica de la familia dsPIC33F*. URL: http://www.microchip.com/pagehandler/en-us/family/16bit/architecture/dspic33f.html?f=4&utm_source=&utm_medium=MicroSolutions&utm_term=&utm_content=&utm_campaign=dsPIC+DSCs.

[5] Jeremiah van Oosten. *Understanding Quaternions*. URL: http://www.3dgep.com/understanding-quaternions/.

[6] Jeff Rowberg. *Firmware de Jeff Rowberg*. URL: https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU9150.

[7] Colton Shane. *The Balance Filter*. URL: https://b94be14129454da9cf7f056f5f8b89a9b17da0be.googledrive.com/host/0B0ZbiLZrqVa6Y2d3UjFVWDhNZms/filter.pdf.

[8] Wikipedia. *Hisotria de la familia dsPIC33F*. URL: https://en.wikipedia.org/wiki/PIC_microcontroller.

[9] Wikipedia. *PID controller*. URL: https://en.wikipedia.org/wiki/PID_controller.