

# ROBÓTICA MÓVIL

Laboratorio de Robótica 21/22

Francisco Javier Román Cortés  
UVUS: fraromcor3  
DNI: 54179754-B  
4º GIERM

Dirección de correo electrónico:  
javierromancortes1999@gmail.com

## 1.- INTRODUCCIÓN

1.1.- *Materiales disponibles.*

1.2.- *Objetivos a lograr con este proyecto.*

## 2.- PRIMEROS PASOS CON EL ROBOT MÓVIL

2.1.- *Montaje básico del robot móvil.*

2.2.- *Comprendión y análisis de elementos del sistema.*

2.2.1.- *Puente H L298N como controlador de motores DC.*

2.2.2.- *Conexionado del L298N con Arduino Mega 2560.*

## 3.- MODO 0 – MONTAR Y PROGRAMAR EL ROBOT PARA REALIZAR LOS MOVIMIENTOS BÁSICOS DE UNA CONFIGURACIÓN DIFERENCIAL

3.1.- *Programación y tests de los movimientos básicos de un robot con configuración diferencial.*

## 4.- MODOS 1\_2 – PROGRAMA QUE PERMITA QUE EL ROBOT SE PARE FRENTE A UNA PARED, A LAS DISTANCIAS QUE SE VAYAN INDICANDO COMO REFERENCIA Y ADEMÁS SE ORIENTE LO MÁS PERPENDICULAR POSIBLE A LA PARED

4.1.- *Montaje y configuración del robot para este modo.*

4.1.1.- *Conceptos teóricos a aplicar.*

4.1.2.- *Decisiones sobre el montaje y funcionalidad del sistema.*

4.2.- *Programación y resolución de estos modos.*

4.3.- *Gráficas y resultados.*

## 5.- MODO 3 – PROGRAMA QUE PERMITA QUE EL ROBOT SE DESPLACE PARALELO A UNA PARED

5.1.- *Montaje y configuración del robot para este modo.*

5.2.- *Programación y resolución de este modo.*

5.3.- *Gráficas y resultados.*

## 6.- MODO 4 – PROGRAMA QUE PERMITA QUE EL ROBOT SE DESPLACE PARALELO A UNA PARED A DISTANCIA INDICADA

6.1.- *Programación y resolución de este modo.*

6.2.- *Gráficas y resultados.*

## 7.- MODO 5 – MONTAR SENSORES DE VELOCIDAD Y REALIZAR TEST QUE MUESTRE LA RESPUESTA ANTE ESCALONES DE VELOCIDAD

7.1.- *Montaje y configuración del robot para este modo.*

7.1.1.- *Conceptos teóricos a aplicar.*

7.2.- *Programación y resolución de este modo.*

7.3.- *Gráficas y resultados.*

## 8.- MODO 6 – MEDIANTE EL USO DE SENSORES DE VELOCIDAD/ CONTROL EN VELOCIDAD, REALIZAR TEST QUE MUESTRE LA RESPUESTA MOVIÉNDOSE EN LÍNEA RECTA

8.1.- *Comentarios generales.*

8.2.- *Gráficas y resultados.*

## 9.- MODO 7 – ESTIMACIÓN DE POSICIÓN

9.1.- *Ecuaciones – Modelo.*

9.2.- *Implementación de la odometría en Arduino.*

9.3.- *Control de posición basado en odometría.*

9.4.- *Implementación del control de posición en Arduino.*

9.5.- *Test en el cual el robot realiza una trayectoria cuadrada de 1 metro de lado.*

9.5.1.- *Conceptos teóricos para la consecución de esta tarea.*

9.5.2.- *Código en Arduino para realizar esta tarea.*

## 10.- MODO 8 – PROYECTO LIBRE:

10.1.- *Planteamiento del proyecto.*

10.2.- *Conceptos teóricos necesarios.*

10.3.- *Montaje y añadidos necesarios sobre el robot.*

10.4.- *Programación y resolución del proyecto.*

10.5.- *Resultados en dos circuitos diferentes.*

## 11.- CONCLUSIONES

## 12.- ANEXO

## 1.- INTRODUCCIÓN

### 1.1.- Materiales disponibles

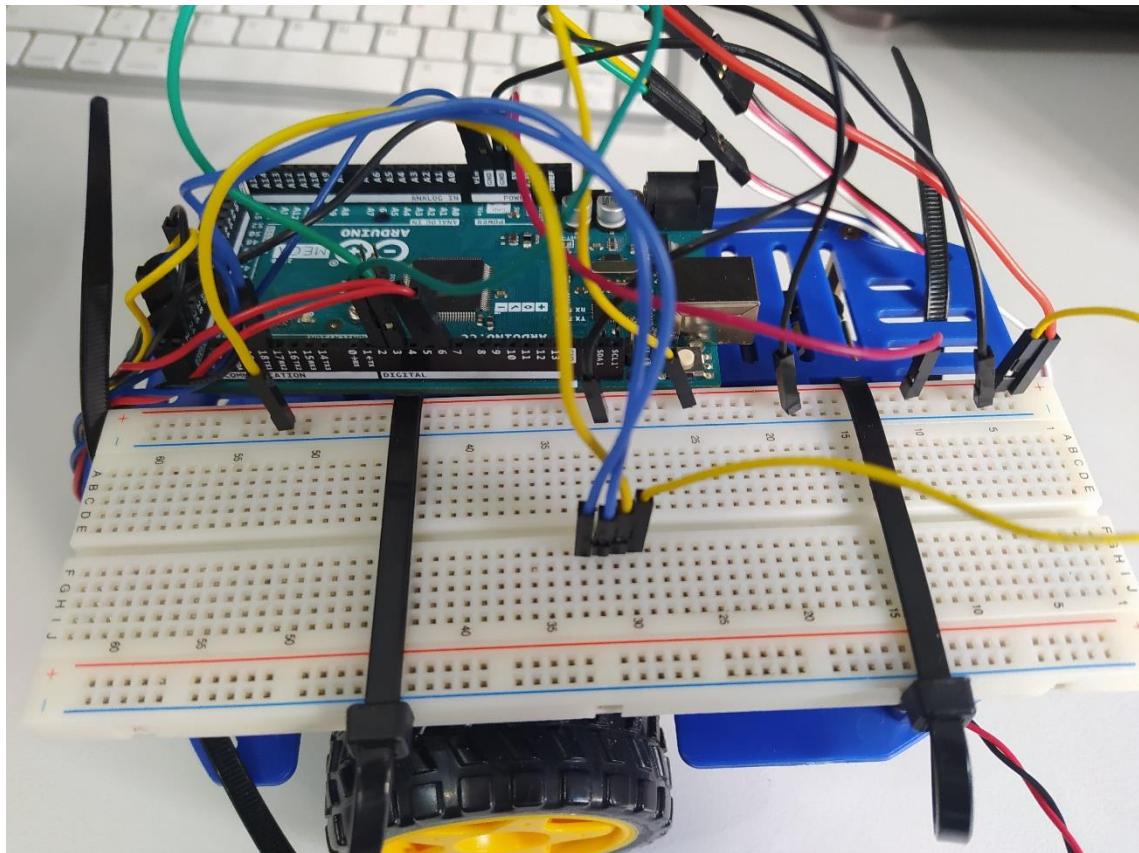
Para esta parte de la asignatura se ha proporcionado un kit de montaje para a través de las diferentes sesiones asignadas a robótica móvil, trabajar montando y programando un robot móvil con configuración diferencial para que realice una serie de tareas objetivo. Si revisamos detenidamente los materiales de los que disponemos:



*Figura 1.1: Conjunto de elementos disponibles.*

Si observamos de izquierda a derecha tenemos:

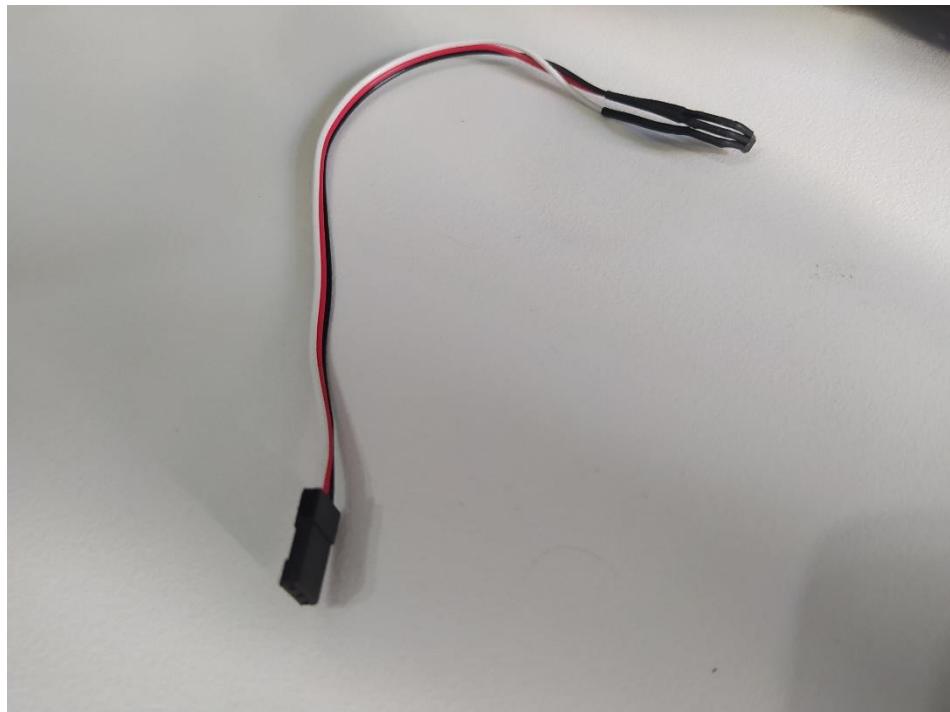
- 1 módulo bluetooth HC-05.
- Cable USB para conexión (puerto serie COM5) entre PC y Arduino Mega 2560.
- Cables para conexión entre pines de Arduino y Breadboard.
- 2 módulos ultrasonidos HC-SR04.
- 2 motores DAGU DG01D 48:1 (este aparece desconectado porque fue necesario cambiar uno de ellos ya que el funcionamiento era muy errático).



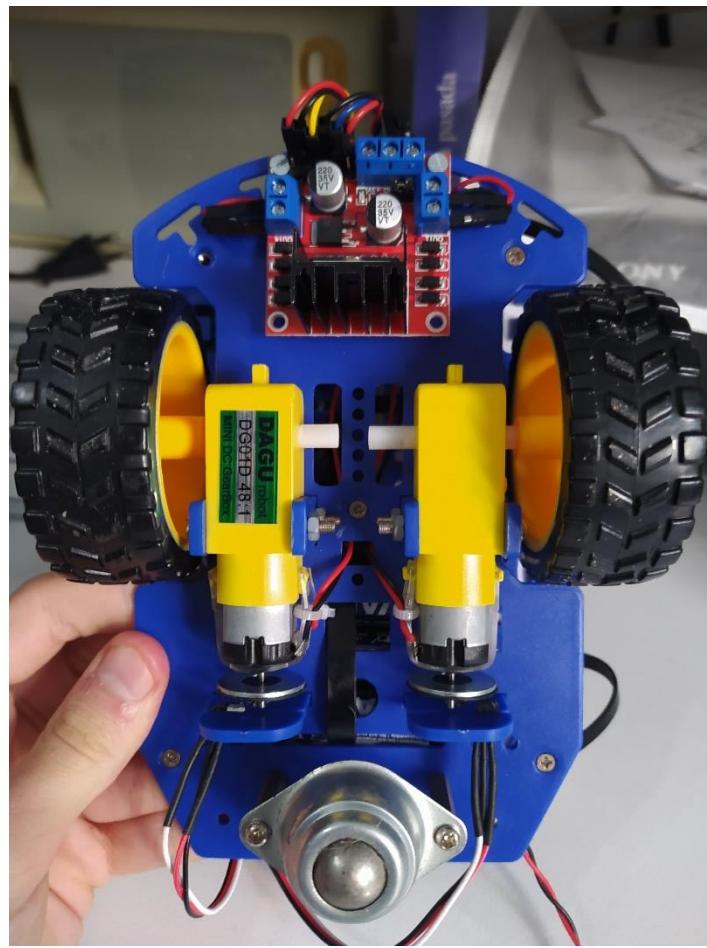
*Figura 1.2: Conjunto de elementos disponibles ya montados sobre el robot.*

En esta figura podemos observar otra serie de elementos cómo son:

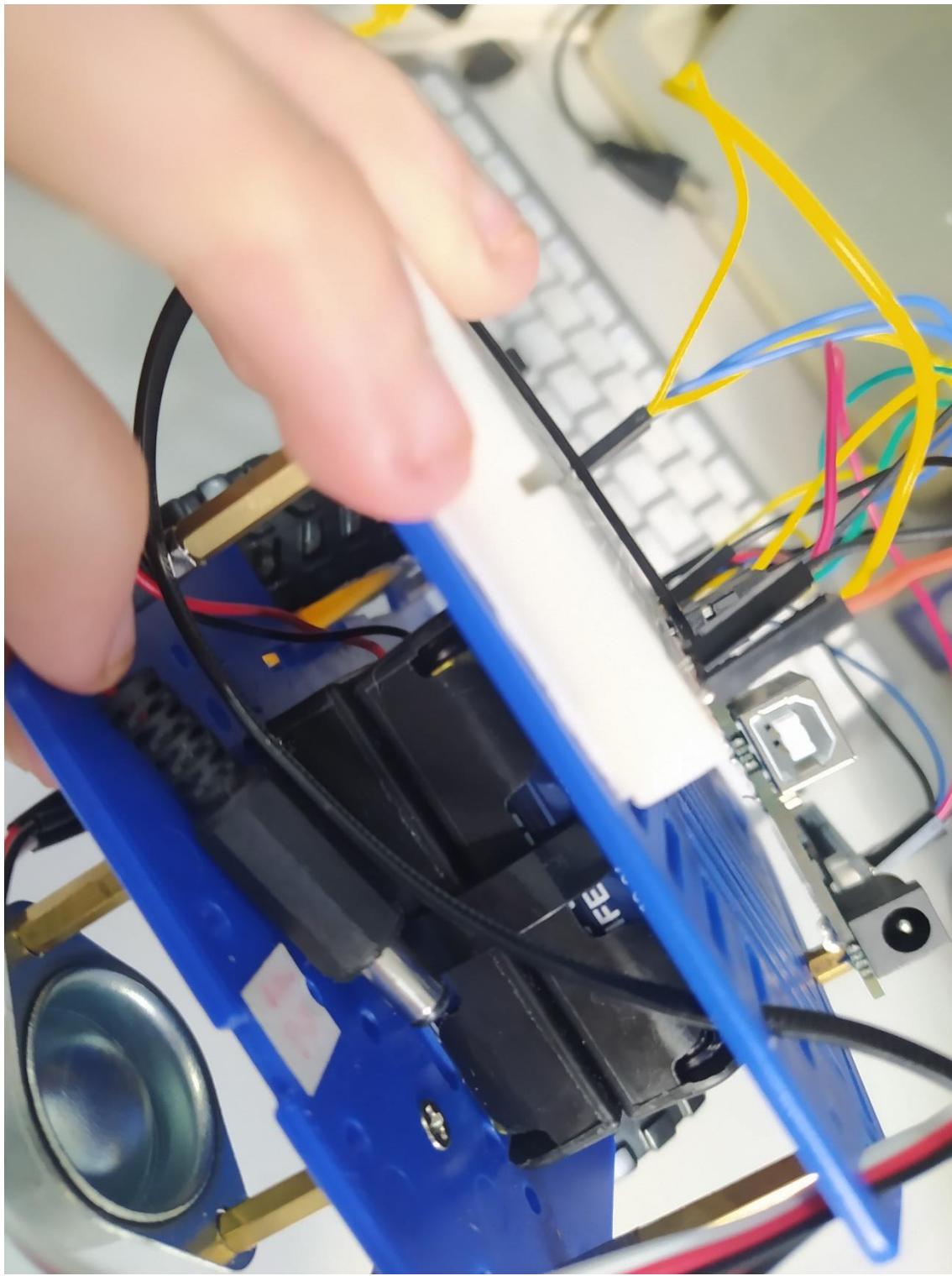
- Breadboard para realizar diferentes conexiones, principalmente con los sensores o el módulo bluetooth.
- Bridas para fijación de la breadboard a la estructura del robot móvil.
- Vemos como aparecen el resto de cables que no aparecen en la figura anterior, ya conectados.
- Placa Arduino Mega 2560 ya montada sobre la estructura del robot móvil.



*Figura 1.3: Encoder magnético parte del DAGU Simple Encoder Kit RS030.*



*Figura 1.4: Parte inferior del robot, donde se aprecian ambos motores y el puente H L298N.*



**Figura 1.5:** Parte intermedia del robot, donde se aprecia la fuente de alimentación para los motores, consistente en 6 pilas AA de 1,5V, que en conjunto proporcionan hasta 9V.

## 1.2.- Objetivos a lograr con este proyecto

Con estos materiales se irán desarrollando las siguientes tareas que se detallarán a lo largo de la memoria. Como es lógico, al trabajar con elementos físicos (hasta ahora en la carrera todo era más bien teórico), veremos que aparecerán detalles que no se esperan y con los que habrá que lidiar para que no impidan el correcto funcionamiento del robot respecto a las tareas.

Como se irá comentando a lo largo de esta memoria, y, gracias a los comentarios del profesor, los cuales se agradecen enormemente, ya que posiblemente por nuestra cuenta no habríamos notado ciertos potenciales problemas en nuestras implementaciones, iremos observando distintas peculiaridades a la hora de trabajar con un sistema real. Donde habrá que realizar ciertas modificaciones, las cuales, tras ciertas dudas de si merecía la pena realizar, se llegaba a la conclusión de que es buena práctica aprender a hacer las “cosas bien”, para ser exigentes con uno mismo a la hora de trabajar con sistemas más críticos en el futuro en cuanto a pérdidas económicas o malfuncionamiento en caso de no hacer esas “cosas bien”.

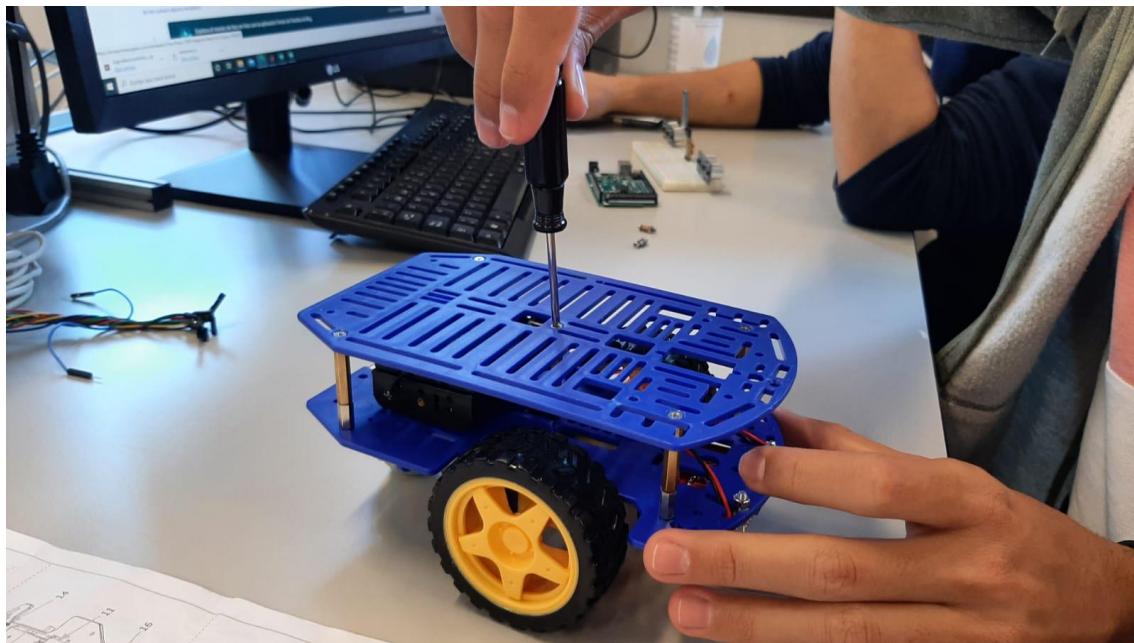
Podemos resumir así los siguientes objetivos a perseguir durante la realización de este proyecto:

- Aprender a trabajar con diferentes tipos de sensores y “exprimir” el máximo de ellos aún siendo sus posibilidades limitadas (respecto a calidad/precio).
- Establecer un cierto protocolo de comunicaciones entre el sistema y el usuario, el cuál no afecte negativamente al sistema de control u otras funcionalidades.
- Reafirmar conceptos relacionados con la robótica, como la estimación de velocidades mediante encoders, la estimación de posición a partir de velocidades (odometría), etc., y llegar a su implementación.
- Aprender a detectar las posibles fuentes de error entre diferentes posibilidades que impiden el funcionamiento esperado del sistema.
- Personalmente, considero como un objetivo también, hacer florecer nuestra pasión por la robótica a lo largo de estas tareas y finalmente, en el proyecto libre de este trabajo.

## 2.- PRIMEROS PASOS CON EL ROBOT MÓVIL

### 2.1- Montaje y funcionamiento de módulos

Se tendrán varias configuraciones de montaje a lo largo del proceso de trabajo, sin embargo, para comenzar, comentaré el montaje realizado durante la primera sesión. Este consistía en un montaje del robot sin sensores, para buscar comprobar la realización de los movimientos básicos pedidos.

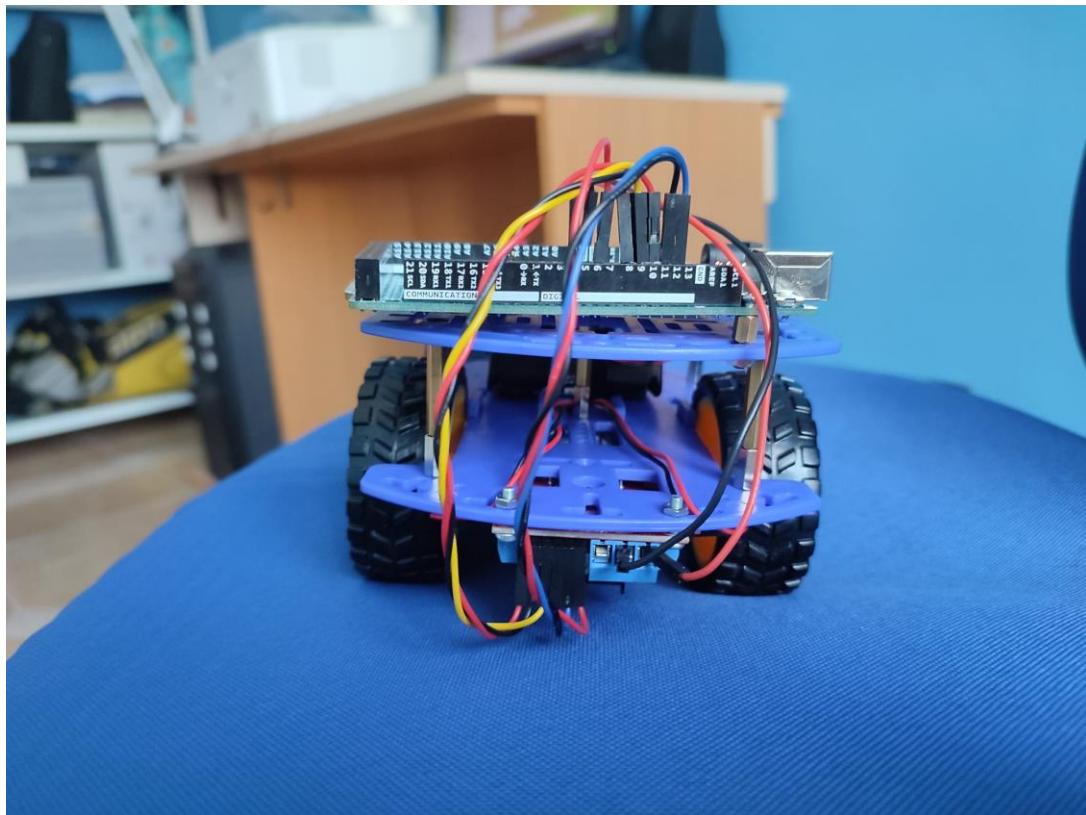


*Figura 2.1: Primeros pasos del montaje del robot móvil.*

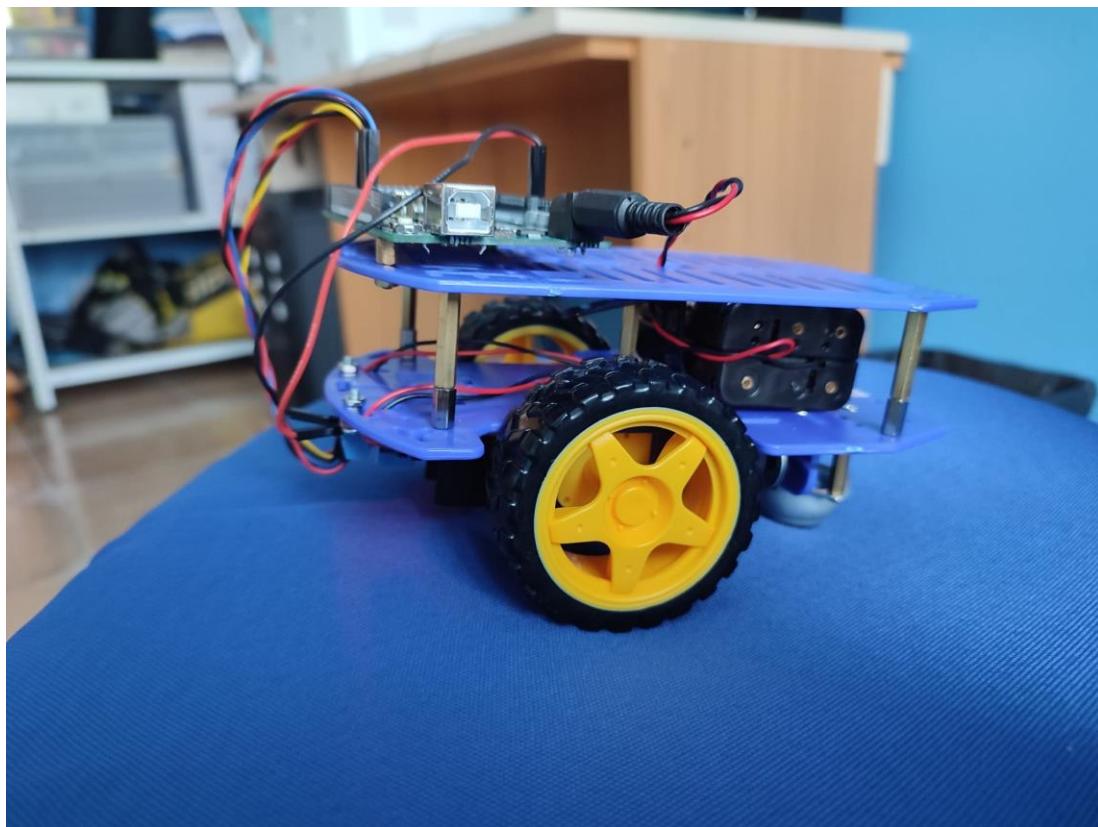
En primer lugar, se colocó el portapilas que proporciona la alimentación necesaria en el “hueco intermedio”, de la mejor manera posible para que el conector alcanzara la placa Arduino.

Tras esta fase, se realizó un montaje bastante desordenado en el que ya se conectaban el Arduino y el controlador de los motores (L298N). El problema era como dijo el profesor, que los colores de los cables se podían confundir, que por el propio montaje del Arduino se necesitaban empalmes de cables (que se soltaban fácilmente) y que el Arduino no se había fijado correctamente y por tanto se movía bastante. Ante estos problemas, lo mejor era rehacer el montaje trenzando cables y ordenándolos y recolocar el Arduino para que los cables llegasen a los pines correctamente.

Podemos ver en las siguientes imágenes como resultó este montaje, ya más “ordenado”:



*Figura 2.2: Vista trasera del montaje inicial del robot móvil.*

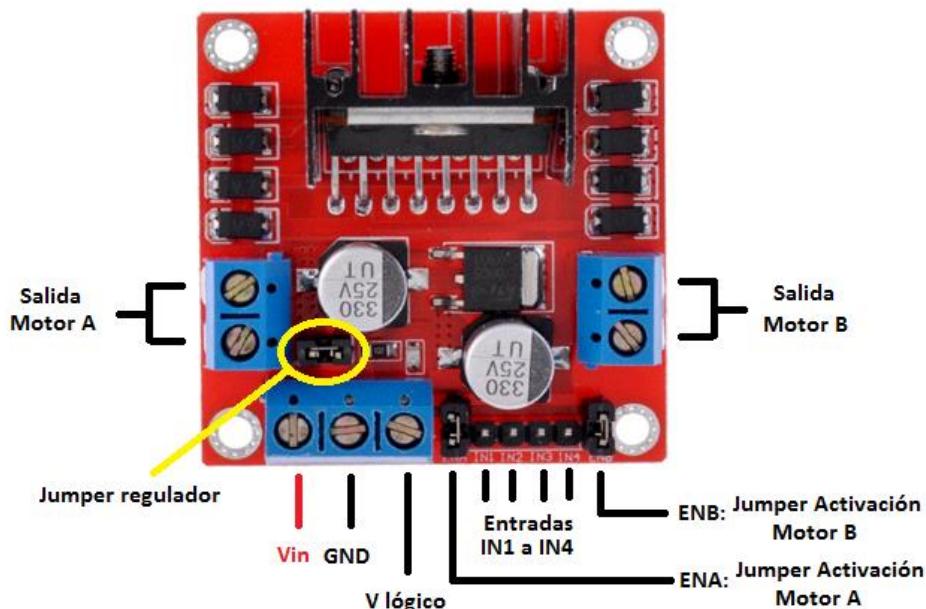


*Figura 2.3: Vista lateral del montaje inicial del robot móvil.*

## 2.2- Comprensión y análisis de elementos del sistema

### 2.2.1- Puente H L298N como controlador de motores DC

A raíz de estas imágenes se puede explicar cómo funciona el control de los motores gracias al controlador L298N H-bridge (basado en la información de este enlace: <https://www.prometec.net/l298n/>). Para guiarnos podemos basarnos en el siguiente esquema del controlador, situado en la parte baja del robot móvil, junto a ambos motores.



*Figura 2.4: Esquema explicativo del controlador Puente H L298N.*

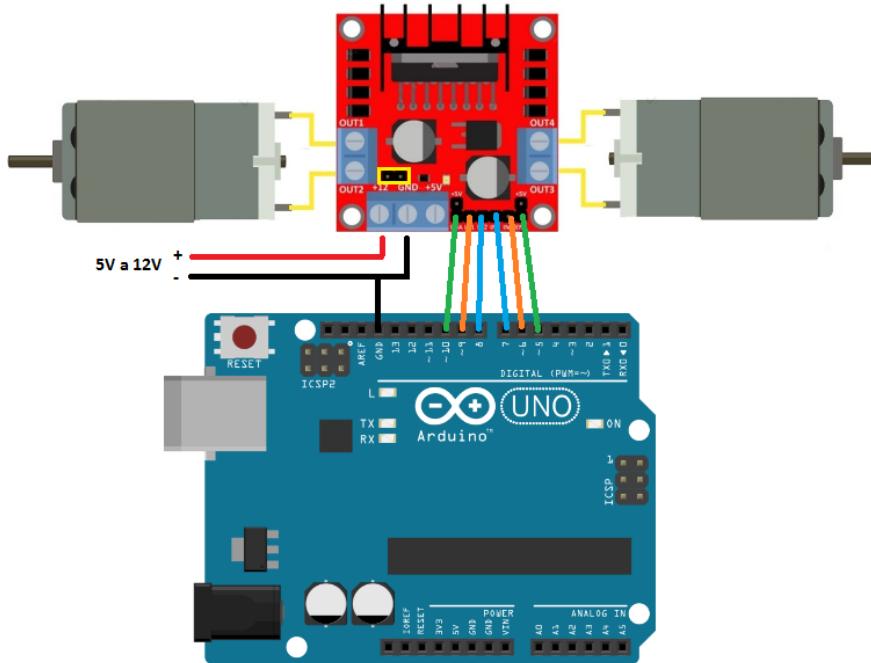
En nuestro caso no se utilizan los Jumper que ahí se ven en ENA y ENB, sino que los conectaremos a pines PWM del Arduino para controlar la velocidad de ambos motores.

Los pines IN1 e IN2 nos sirven para controlar el sentido de giro del motor A, y los pines IN3 e IN4 el del motor B. Funcionan de forma que, si IN1 está a HIGH e IN2 a LOW, el motor A gira en un sentido, y si está IN1 a LOW e IN2 a HIGH lo hace en el otro. Y análogo para los pines IN3 e IN4 y el motor B.

Para controlar la velocidad de giro de los motores tenemos que quitar los jumpers y usar los pines ENA y ENB. Los conectaremos a dos salidas PWM de Arduino de forma que le envíemos un valor entre 0 y 255 que controle la velocidad de giro. Si tenemos los jumpers colocados, los motores girarán a la misma velocidad, de ahí que no se usen, ya que sí interesa modificar las velocidades, sobre todo para el tema de giros y rotaciones, debido a que, por la naturaleza de la configuración diferencial, por ejemplo, la rotación (entendida como pivotado sin desplazamiento) se consigue girando ambas ruedas a la misma velocidad, pero en sentidos contrarios. Por su parte para el giro, ambas ruedas deben girar en el mismo sentido, pero siendo que una de ellas lo haga más rápido que la otra, provocando dicho giro del robot.

## 2.2.2- Conexión del puente H L298N con Arduino Mega 2560

El esquema de conexión con Arduino sería bastante similar al siguiente, salvo el modelo de Arduino, ya que en nuestro caso se Arduino Mega 2560.



*Figura 2.5: Ejemplo de conexionado del Puente H L298N con los motores y la placa Arduino (en el esquema aparece Arduino UNO, pero el conexionado es similar para nuestro Arduino Mega 2560).*

Con esto, conociendo el funcionamiento de este controlador de motores L298N, ya se puede realizar la primera tarea.

### 3.- MODO o – MONTAR Y PROGRAMAR EL ROBOT PARA REALIZAR LOS MOVIMIENTOS BÁSICOS DE UNA CONFIGURACIÓN DIFERENCIAL

#### 3.1- Programación y tests de los movimientos básicos de un robot con configuración diferencial

Tras el montaje necesario, explicado en el apartado 2 de esta memoria, se procede a la programación necesaria para realizar estos movimientos básicos.

Sin entrar a analizar el código línea por línea, comentaré generalmente lo que se realiza en el mismo.

Primeramente, se busca asignar los pines del controlador L298N a sus respectivos pines a los que se conecta en Arduino, tras ello, se declaran todos como salidas.

```
modo0_mov_basicos
//Programa Test de movimientos básicos

// Motor A (derecha) (asignación de pines de Arduino)
#define ENA 6
#define IN1 22
#define IN2 23

// Motor B (izquierda) (asignación de pines de Arduino)
#define ENB 5
#define IN3 24
#define IN4 32

unsigned long int tipo_movimiento = 0;      //Variable para asignar el tipo de movimiento deseado
unsigned long int aux_buffer = 0;
const int LongitudBuffer = 50;
```

*Figura 3.1: Asignación de pines a usar y variables para el buffer de comunicaciones.*

```
void setup()      //Declaración de pines y configuración de la comunicación serie
{
    pinMode (ENA, OUTPUT);
    pinMode (ENB, OUTPUT);
    pinMode (IN1, OUTPUT);
    pinMode (IN2, OUTPUT);
    pinMode (IN3, OUTPUT);
    pinMode (IN4, OUTPUT);
    Serial1.begin(38400);
}
```

*Figura 3.2: Declaración de los diferentes pines como salidas para controlar enviar señales al Puente H e inicialización del puerto serie (en este caso Serial1 es el que se usa para conexión Bluetooth).*

Posteriormente, se crean funciones para cada uno de los movimientos buscados:

*Adelante(), Atras(), Parar(), AdelanteIzq(), AdelanteDer(), AtrasIzq(), AtrasDer()*

Aunque originariamente se planteó el uso de las funciones *Adelante()* y *Atras()* con parámetros para la velocidad de cada motor, se hizo conveniente la creación de las funciones

*AdelanteIzq(), AdelanteDer(), AtrasIzq(), AtrasDer()*, que permitían de manera mucho más clara la entrega de PWM deseado a cada rueda con el sentido de giro del motor ya configurado en cada una de las funciones para que hiciese el movimiento que los propios nombres de las funciones indican.

En ellas básicamente se configura el sentido de giro de los motores (y por tanto de las ruedas), actuando sobre los pines IN1, IN2, IN3 e IN4, para tener el sentido deseado en cada una según las necesidades del movimiento en cuestión. Además, se configura también la velocidad de giro de ambos motores, ajustando el PWM que se entregará a cada motor, para que sea la necesaria para cada movimiento también.

```
void AdelanteDer (int Vforward2)
{
    //Configuración motor derecho
    digitalWrite (IN1, HIGH);
    digitalWrite (IN2, LOW);
    analogWrite (ENA, Vforward2); //Velocidad motor derecho
}

void AdelanteIzq (int Vforward1)
{
    //Configuración motor izquierdo
    digitalWrite (IN3, HIGH);
    digitalWrite (IN4, LOW);
    analogWrite (ENB, Vforward1); //Velocidad motor izquierdo
}

void AtrasDer (int Vbackwards2)
{
    //Configuración motor derecho
    digitalWrite (IN1, LOW);
    digitalWrite (IN2, HIGH);
    analogWrite (ENA, Vbackwards2); //Velocidad motor derecho
}

void AtrasIzq (int Vbackwards1)
{
    //Configuración motor izquierdo
    digitalWrite (IN3, LOW);
    digitalWrite (IN4, HIGH);
    analogWrite (ENB, Vbackwards1); //Velocidad motor izquierdo
}

void Parar ()
{
    //Configuración motor derecho
    digitalWrite (IN1, LOW);
    digitalWrite (IN2, LOW);
    analogWrite (ENA, 0); //Velocidad motor derecho
    //Configuración motor izquierdo
    digitalWrite (IN3, LOW);
    digitalWrite (IN4, LOW);
    analogWrite (ENB, 0); //Velocidad motor izquierdo
}
```

*Figura 3-3: Conjunto de funciones para actuar sobre cada una de las ruedas de la manera deseada.*

Ahora básicamente, tenemos un loop con un switch(case), en el cual se permite realizar los diferentes movimientos básicos mediante comandos enviados mediante Bluetooth desde el PC remoto, de manera que enviando una serie de números: 0,1,2,3,4,5,6 se ejecuta cada tipo de movimiento configurado:

```

void loop(){
    switch(tipo_movimiento){
        /*PARADO POR DEFECTO*/
        case 0:
            Buffer();
            tipo_movimiento = aux_buffer;
            Parar();
            break;

        /*AVANZAR*/
        case 1:
            Buffer();
            tipo_movimiento = aux_buffer;
            AdelanteIzq(120);
            AdelanteDer(130);
            break;

        /*RETROCEDER*/
        case 2:
            Buffer();
            tipo_movimiento = aux_buffer;
            AtrasIzq(115);
            AtrasDer(130);
            break;

        /*PIVOTAR HACIA LA DERECHA*/
        case 3:
            Buffer();
            tipo_movimiento = aux_buffer;
            AdelanteIzq(125);
            AtrasDer(145);
            break;

        /*PIVOTAR HACIA LA IZQUIERDA*/
        case 4:
            Buffer();
            tipo_movimiento = aux_buffer;
            AtrasIzq(125);
            AdelanteDer(145);
            break;

        /*GIRAR HACIA LA IZQUIERDA*/
        case 5:
            Buffer();
            tipo_movimiento = aux_buffer;
            AdelanteIzq(90);
            AdelanteDer(175);
            break;

        /*GIRAR HACIA LA DERECHA*/
        case 6:
            Buffer();
            tipo_movimiento = aux_buffer;
            AdelanteIzq(165);
            AdelanteDer(100);
            break;
    }
}

```

*Figura 3.4: Prueba de los diferentes movimientos básicos del robot móvil.*

Finalmente, aunque se desarrolló en las fases intermedias-finales, se incluye un sistema de buffer de comunicaciones para la recepción y tratamiento de los datos recibidos por puerto serie (ya sea por Bluetooth o por USB (COM5)), por tanto, podemos explicarlo ahora, ya que se ha incluido en este programa también.

```

void Buffer(){
    if(Serial1.available() > 0) //Para procesar los diferentes caracteres que hay en el buffer del serial, debemos quedarnos aquí hasta que llegue un \r
    {
        //Array/Buffer para almacenar el mensaje entrante
        static char Buffer[LongitudBuffer];
        static unsigned int IndiceBuffer = 0;

        //Leer el siguiente byte disponible en el buffer del Serial
        char byte_temporal = Serial1.read();

        //Comprobar si el siguiente mensaje es carácter terminador
        if(byte_temporal != '\r')
        {
            //Añadir el byte entrante al array:
            Buffer[IndiceBuffer] = byte_temporal;
            IndiceBuffer++;
        }

        //Mensaje completo recibido:
        else
        {
            //Añadir carácter nulo al array:
            Buffer[IndiceBuffer] = '\0';

            //Convertir el mensaje a entero:
            sscanf(Buffer, "%lu", &aux_buffer); //Se formatea como unsigned long para que funcione la separación de los números

            //Resetear índice del buffer:
            IndiceBuffer = 0;
        }
    }
}

```

*Figura 3.5: Función que implementa la versión final del buffer de comunicaciones.*

En este buffer implementado se realiza básicamente lo siguiente:

- Sólo si hay datos en el puerto serie, se procesan en el buffer.
- En cada ciclo del loop (muy rápido) se procesa un byte de los datos enviados por puerto serie.
- Se reserva un array de caracteres (buffer en sí mismo) con longitud suficiente.
- Se lee el siguiente byte disponible en el puerto serie (1 carácter).
- Se comprueba si se ha llegado el carácter '\r' que es la forma de detectar que se ha procesado toda la información (*string* completa).
- Si no lo es, se guarda el byte su posición correspondiente del array y se incrementa el índice que indexa dicho array para guardar los bytes en el orden correspondiente.
- Si llega '\r' y por tanto se recibe la cadena completa, se añade a la cadena un carácter terminador '\0', y se transforma mediante *sscanf* la cadena recibida a entero, ya que generalmente lo que se mandan son comandos exclusivamente numéricos, como referencias de velocidades o modos de funcionamiento.
- Finalmente se resetea el índice que indexa el array, para el caso en que lleguen futuros datos estar preparado para repetir el procesado de los nuevos datos.

Las ventajas que aporta esto, es un tiempo de procesado uniforme de los datos, al procesar en cada ciclo siempre un byte, lo cual no interfiere irregularizando los tiempos de muestreo de los controladores de los siguientes modos y pudiendo empeorar su funcionamiento. Esto es muy importante ya que el hecho de hacer compatible una comunicación eficiente con el resto del sistema sin decrementar excesivamente su rendimiento es una tarea crítica de los sistemas, la cual no estábamos siendo capaces de ver hasta ciertos comentarios del profesor.

También es necesario mostrar las primeras versiones del "protocolo" de comunicaciones, para entender por qué era necesario implementar este buffer algo más sofisticado.

```
//Es necesario buffer, comunicación problemática
if(Serial1.available() > 0)
{
    var_referencia = Serial1.parseInt();
}

if(var_referencia == 1) referencia = 30.0;
if(var_referencia == 2) referencia = 50.0;
Serial1.println(referencia);
```

*Figura 3.6: Función de comunicaciones primitiva, previa al buffer completo anterior.*

Como podemos ver, es muy simple y se hace uso de una función que busca enteros en la cadena, pero esto podía generar muchos problemas si no se detectaba el final de la cadena u otras irregularidades, ya que prácticamente bloqueaba todo el sistema de control al quedarse bloqueada esta función, o simplemente no recibir correctamente los datos enviados. De aquí que finalmente se considerase necesario seguir las indicaciones e implementar el buffer de la *figura 3.4*.

Se pueden ver los diferentes movimientos básicos probados en el programa en el siguiente video adjunto.

- Enlace a video del test:

[https://drive.google.com/file/d/1kZ327\\_QOuNhFS\\_VoTrO87d\\_UzXoFG5cW/view?usp=sharing](https://drive.google.com/file/d/1kZ327_QOuNhFS_VoTrO87d_UzXoFG5cW/view?usp=sharing)

## 4.- MODOS 1\_2 – PROGRAMA QUE PERMITA QUE EL ROBOT SE PARE FRENTE A UNA PARED, A LAS DISTANCIAS QUE SE VAYAN INDICANDO COMO REFERENCIA Y ADEMÁS SE ORIENTE LO MÁS PERPENDICULAR POSIBLE A LA PARED

### 4.1- Montaje y configuración del robot para este modo

#### 4.1.1- Conceptos teóricos a aplicar

En este modo, necesitamos conocer lógicamente las distancias a dicha pared, por tanto, aparece la necesidad de hacer uso de sensores, concretamente, los **sensores de ultrasonidos** ya introducidos en apartados anteriores de esta memoria. Mediante ellos, podemos conocer la distancia a obstáculos, en este caso una pared para realizar control sobre las ruedas y lograr comportamientos adecuados. Podemos empezar explicando brevemente el funcionamiento de estos sensores de ultrasonidos como introducción a la resolución de este modo, previo a mostrar el montaje de estos sensores en el propio robot.

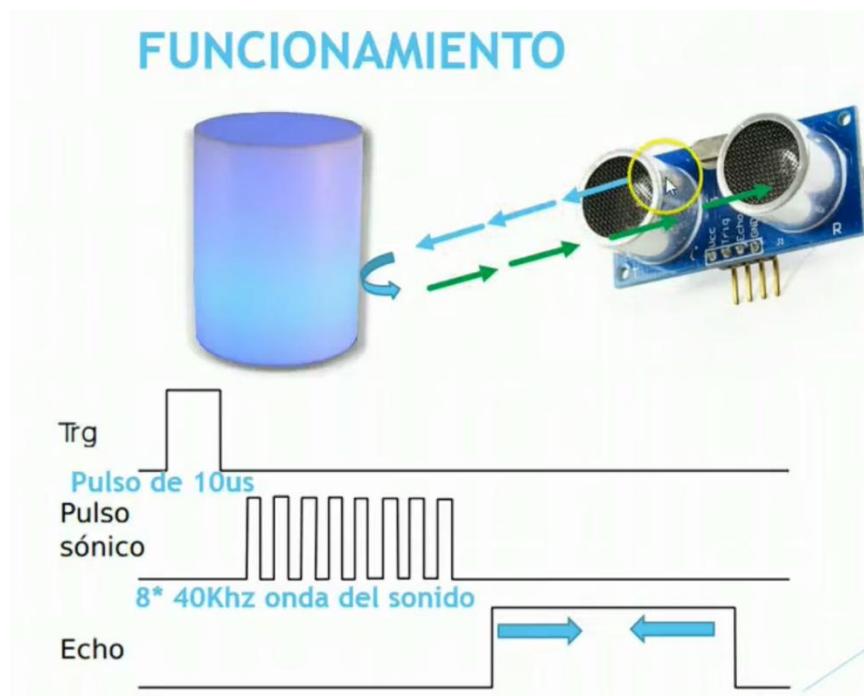


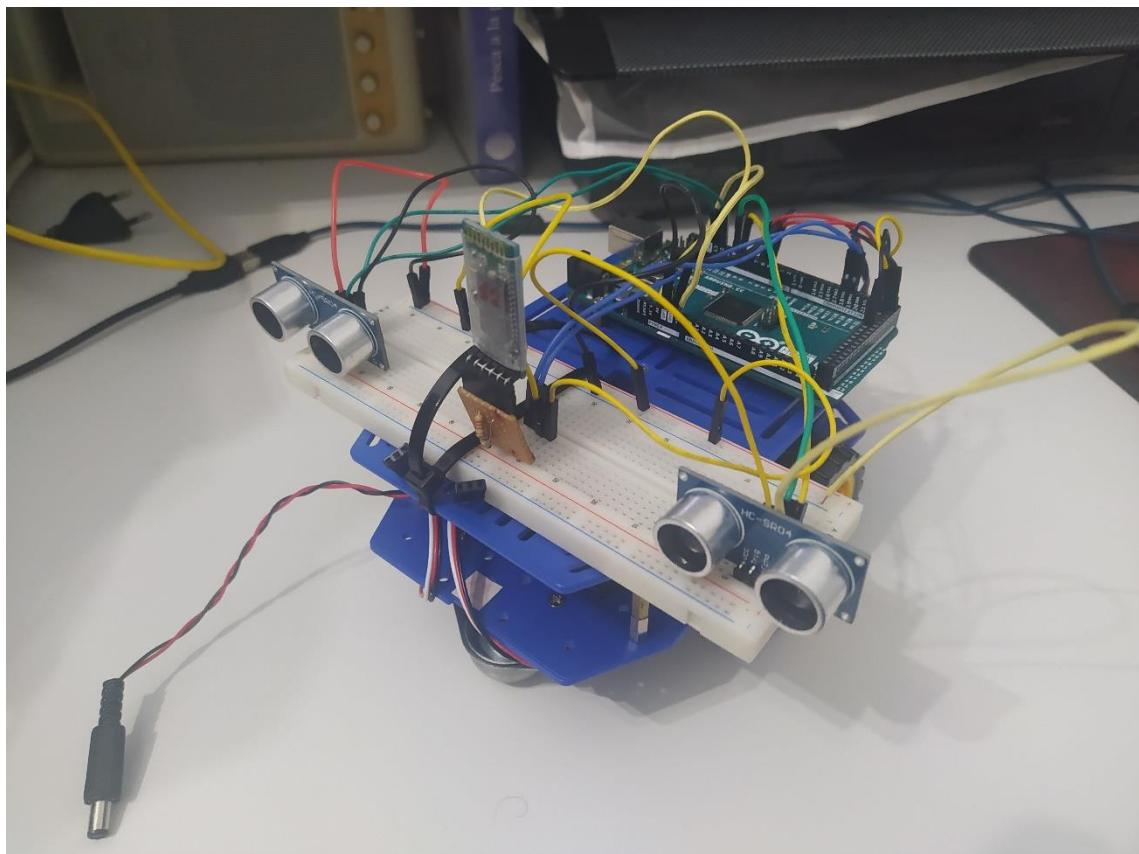
Figura 4.1: Imagen ilustrativa del funcionamiento del sensor HC-SR04.

El funcionamiento de estos sensores se resume en:

- El elemento transmisor emite un sonido con frecuencia 40kHz (no audible).
- Estas ondas ultrásonicas rebotan con los obstáculos y son recibidas por el elemento receptor.
- El tiempo que ha tardado el pulso en ir y volver y ser recibido por el receptor, permite calcular la distancia, basándose en la velocidad del sonido (343 m/s).
- Para realizar estos pasos anteriores, se manda un pulso de 10 microsegundos por el transmisor, se espera hasta que es recibido por el receptor y como se ha dicho, según el tiempo que se tarda en recibir y por tanto, el tiempo que tarda la señal del pin ECHO en pasar de LOW a HIGH, se puede obtener la distancia.

#### 4.1.2- Decisiones sobre el montaje y funcionalidad del sistema

Conocidas las nociones básicas de estos sensores, necesitamos montarlos adecuadamente para este modo, es decir, como queremos que se desplace perpendicularmente a una pared, los sensores se deben colocar en la parte frontal del robot, para ello, el montaje se realiza con la configuración que aparece en la siguiente figura.



*Figura 4.2: Imagen del montaje de los sensores HC-SR04 para desplazamiento perpendicular a una pared (MODOS 1 y 2).*

Como vemos, aunque para el Modo 1 en el que no se requiere reorientación para colocarse perpendicular a la pared, si no simplemente que avance o retroceda hasta las distancias de referencia, finalmente, se decidió usar para ambos modos este montaje de doble sensor ultrasónico frontal para aprovechar los recursos. Esto quiere decir, que hubo un cierto tiempo en que se planteó para el Modo 1 usar un único sensor situado en el centro de la Breadboard que indicase la distancia frontal, pero esto generaba muchos problemas ya que, al estar descompensadas las velocidades de las ruedas, el avance no era equitativo y por tanto se desviaba. Esto provocaba que midiese distancias más alejadas que la real a la pared, el control no fuese efectivo y se comportase erráticamente. Además, el hecho de tener un único sensor en el centro de la Breadboard, implica una única fuente de información y por lo tanto una única referencia para ambas ruedas, lo cual sólo permite un control o bien conjunto de ambas ruedas, o bien desacoplado pero con la misma referencia, lo cual tampoco es práctico.

Aquí aparece la ventaja fundamental de incluir los dos sensores con la disposición mostrada en la figura, de manera que tenemos dos referencias de distancia que permiten desacoplar y controlar cada rueda con su referencia de distancia. Es decir, si se desvía hacia la derecha, el sensor de la derecha detectará que está más alejado de la referencia, la actuación será mayor

sobre la rueda derecha y lograremos así que el robot se reoriente, con esto se logran tanto las funcionalidades requeridas para el Modo 1 (desplazamiento del robot a ciertas distancias de referencia) y Modo 2 (desplazamiento del robot a ciertas distancias de referencia manteniendo su orientación lo más perpendicular posible a la pared).

## 4.2- Programación y resolución de estos Modos (1 y 2)

La codificación necesaria para estos modos se hace algo más extensa, por tanto, intentaré que las capturas sean lo más concisas posibles y tampoco extenderme mucho en las explicaciones de las mismas.

En primer lugar, comenzamos declarando variables necesarias y asignando pines de nuevo.

```
Control_Modo_1_y_2

// Motor A (derecha) (asignación de pines de Arduino)
#define ENA 6
#define IN1 22
#define IN2 23

// Motor B (izquierda) (asignación de pines de Arduino)
#define ENB 5
#define IN3 24
#define IN4 32

//Dos pines para TRIG1 y ECO1
int TRIG1 = 10; //Trigger al Pin 10 (PWM)
int ECO1 = 9; //ECO al Pin 9 (PWM)

//Dos pines para TRIG2 y ECO2
int TRIG2 = 12; //Trigger al Pin 12 (PWM)
int ECO2 = 13; //ECO al Pin 13 (PWM)

//Variables de medida del primer ultrasonido:
int DURACION1; //Variable para medir la duración de transmisión/recepción
double DISTANCIA1; //Variable para medir la distancia a la pared u obstáculo.

//Variables de medida del segundo ultrasonido:
int DURACION2; //Variable para medir la duración de transmisión/recepción
double DISTANCIA2; //Variable para medir la distancia a la pared u obstáculo.

//Variables para el Buffer:
const int LongitudBuffer = 50; //Tamaño que se reserva para el buffer implementado, que almacena los datos que entran al puerto serie de Arduino (por Putty o COM5)
unsigned long int referencia = 30; //Variable para indicar la referencia de distancia en [cm]
unsigned long int movimiento = 0; //Variable para indicar al robot si se debe mover o no (0 parado; 1 puede moverse)
unsigned long int aux_buffer = 0; //Para separar movimiento de referencia en el buffer
```

*Figura 4.3: Variables y asignación de pines para estos modos.*

```

int modo=1;
//Variables para hacer gestión del incremento de tiempo en el loop, principalmente para representación de telemetría
unsigned long tiempo_actual = 0;
unsigned long tiempo_anterior = 0;
unsigned long delta_tiempo = 0;

// ***** Variables Globales PI 1 *****
unsigned long lastTime1 = 0,     SampleTime1 = 0; //Variables de tiempo discreto.
double      Input1   = 0.0;        //Entrada del control -> salida del sistema = distancia medida por el ultrasonido
static double Setpoint1 = 0.0;    //Entrada al control (referencia en centímetros)
double      cumError1 = 0.0;     //Integral del error
double      kpl     = 0.0;        //Parámetro de la parte proporcional
double      kil     = 0.0;        //Parámetro de la parte integral
double      Prop1   = 0.0;        //Parte proporcional
double      Integ1  = 0.0;        //Parte integral
double      outMin1 = 0.0, outMax1 = 0.0; //Limites para no sobrepasar la resolución del PWM (saturación)
double      error1  = 0.0;        //Diferencia entre la referencia solicitada y la medida del ultrasonido
double      Out1    = 0.0;        //Salida del controlador 1
//***** Variables Globales PI 2 *****
unsigned long lastTime2 = 0,     SampleTime2 = 0; //Variables de tiempo discreto.
double      Input2   = 0.0;        //Entrada del control -> salida del sistema = distancia medida por el ultrasonido
static double Setpoint2 = 0.0;    //Entrada al control (referencia en centímetros)
double      cumError2 = 0.0;     //Integral del error
double      kp2     = 0.0;        //Parámetro de la parte proporcional
double      ki2     = 0.0;        //Parámetro de la parte integral
double      Prop2   = 0.0;        //Parte proporcional
double      Integ2  = 0.0;        //Parte integral
double      outMin2 = 0.0, outMax2 = 0.0; //Limites para no sobrepasar la resolución del PWM (saturación)
double      error2  = 0.0;        //Diferencia entre la referencia solicitada y la medida del ultrasonido
double      Out2    = 0.0;        //Salida del controlador 1
//*****

```

*Figura 4.4: Más variables para estos modos (principalmente para los controladores tipo PI).*

Aprovechando el detalle de los comentarios del código en las capturas que se adjuntan se considera que no es necesario explicar la función de cada variable de nuevo.

Atravesando la declaración de variables, entramos en el Setup, donde se configuran ciertos pines, incluyendo ahora los pines de los sensores ultrasónicos, la configuración de la comunicación con Bluetooth y la inicialización de ciertos parámetros de los controladores:

```

void setup() //Declaración de pines y configuración de la comunicación serie
{
    Input1 = DISTANCIA1;
    Input2 = DISTANCIA2;
    pinMode(TRIG1, OUTPUT);
    pinMode(EC01, INPUT);
    pinMode(TRIG2, OUTPUT);
    pinMode(EC02, INPUT);
    pinMode(ENA, OUTPUT);
    pinMode(ENB, OUTPUT);
    pinMode(IN1, OUTPUT);
    pinMode(IN2, OUTPUT);
    pinMode(IN3, OUTPUT);
    pinMode(IN4, OUTPUT);
    Serial1.begin(38400);

    // Ajustación máxima y mínima; corresponde a Max.: 0=0V hasta 255=5V (PWMA), y Min.: 0=0V hasta -255=5V (PWMB). El PWM se convertirá a la salida en un valor absoluto, nunca negativo.
    outMax1 = 255.0;           // Límite máximo del controlador 1 PI.
    outMin1 = -outMax1;         // Límite mínimo del controlador 1 PI.

    outMax2 = outMax1;          // Límite máximo del controlador 2 PI.
    outMin2 = outMin1;          // Límite mínimo del controlador 2 PI.

    SampleTime1 = 200;          // Se asigna el tiempo de muestreo de los controladores en milisegundos.
    SampleTime2 = SampleTime1;

    //Valores iniciales del controlador 1
    kpl = 2.0;
    kil = 0.0;

    //Valores iniciales del controlador 2
    kp2 = 2.0;
    ki2 = 0.0;
}

```

*Figura 4.5: Setup de pines, configuración Bluetooth e inicialización de parámetros de control.*

Llegamos ahora al Loop, donde omitiremos en la captura toda la parte del Buffer ya explicado en el apartado anterior (Modo 0) y principalmente, comenzamos recogiendo el tiempo al inicio del loop mediante *millis()*:

```
void loop(){
    int tiempo_anterior = millis();
    if(Serial1.available() > 0) //Para procesar los diferentes caracteres que hay en el buffer del serial, debemos quedarnos aquí hasta que llegue un \r
    {
        //Array/buffer para almacenar el mensaje entrante
        static char Buffer[LongitudBuffer];
        static unsigned int IndiceBuffer = 0;
        //.... (sigue)
```

*Figura 4.6: Comienzo del loop, omitiendo el buffer ya explicado.*

A continuación, recogemos la referencia de distancia, que debe venir comandada con cierto formato para que el código la compute, es decir, viendo la siguiente figura:

```
//Se asigna la referencia de distancia deseada para ambos controladores

movimiento = aux_buffer/100;
referencia = aux_buffer%100;
Setpoint1 = double(referencia);
Setpoint2 = Setpoint1;

//Ultrasonidos 1:
digitalWrite(TRIG1, HIGH);           //Se envía pulso
delayMicroseconds(10);               //Esperamos 10 us
digitalWrite(TRIG1, LOW);            //Dejamos de enviar el pulso
DURACION1 = pulseIn(ECO1, HIGH);     //Se recibe y calcula la duración
DISTANCIA1 = DURACION1 / 58.2;       //Calculamos la distancia al obstáculo

//DELAY PEQUEÑO EN MEDIO DE LAS DOS LECTURAS
delayMicroseconds(1000);

//Ultrasonidos 2:
digitalWrite(TRIG2, HIGH);           //Se envía pulso
delayMicroseconds(10);               //Esperamos 10 us
digitalWrite(TRIG2, LOW);            //Dejamos de enviar el pulso
DURACION2 = pulseIn(ECO2, HIGH);     //Se recibe y calcula la duración
DISTANCIA2 = DURACION2 / 58.2;       //Calculamos la distancia al obstáculo
```

*Figura 4.7: Recolección de información del buffer y cálculo de distancias a través de los sensores.*

Como se comentaba, el comando debe ser un número de 3 cifras por puerto serie que se segmentará en dos partes, siendo uno asignado a la variable “movimiento” y otro a la variable “referencia”. Esto se realiza para que el control no empiece a actuar hasta que movimiento tenga valor 1, ya que, si no se hacía esto, el control actuaba simplemente al conectar el sistema, al tener ya una referencia inicializada a cualquier valor en la declaración de variables.

Para explicar la segmentación de los datos y cómo responderá a lo que se le mande por puerto serie utilizaré los siguientes ejemplos. Suponemos que llega lo siguiente por puerto serie:

**130** En este caso “movimiento” = 1 y “referencia” = 30 (el robot iría a distancia 30 cm de la pared).

**040** En este caso “movimiento” = 0 y “referencia” = 40 (Aunque sí que recoge los 40 cm como referencia, movimiento está inactivo y por tanto el control no actúa).

A continuación, se procesa la información que aportan los pulsos provocados por los sensores ultrasónicos en sus respectivos pines. Para esto, se realiza lo explicado en la parte de conceptos teóricos de este apartado, y finalmente mediante la duración del pulso del Pin ECO de LOW a HIGH y aplicando una fórmula de conversión para calcular la distancia a partir de esa duración, tenemos almacenada en cada ciclo la distancia en [cm] que está detectando cada uno de los sensores.

Aunque es recomendable filtrar la información de estos sensores, no se ha realizado para estos modos, ya que no entraba ruido apenas ni comportamientos erráticos, por las características del propio experimento. Sin embargo, sí que se hace necesario esto en modos posteriores donde si se pone de manifiesto los problemas que genera no filtrar estas medidas.

Continuamos ahora con la parte en la que se realiza control sobre ambas ruedas para alcanzar la distancia de referencia en el loop.

```

if(movimiento == 1){
//Parte PI
Input1 = DISTANCIA1;           //Input como se ha comentado, serán las lecturas de los ultrasonidos
Input2 = DISTANCIA2;
Out1 = Compute1();             //La función "Compute()" nos dará la salida del control, el PWM que debemos aplicar a los motores
Out2 = Compute2();

//Caso para cuando estamos en torno a la referencia (se deja un margen de 0.5 centímetros): paramos
if (abs(Setpoint1 - Input1) <= 0.5)
{
    PararIzq();
    Out1 = 0.0;          //Para correcta visualización de telemetría
}

if(abs(Setpoint2 - Input2) <= 0.5)           //Setpoint - Input = error en centímetros
{
    PararDer();
    Out2 = 0.0;          //Para correcta visualización de telemetría
}

//Caso para cuando estamos más cerca del obstáculo que lo que nos indica la referencia: retrocedemos
if((Setpoint1 - Input1) > 0.5)
{
    AtrasIzq(abs(Out1));
}

if((Setpoint2 - Input2) > 0.5)
{
    AtrasDer(abs(Out2));
}

//Caso para cuando estamos más lejos del obstáculo que lo que nos indica la referencia: avanzamos
if((Setpoint1 - Input1) < -0.5)
{
    AdelanteIzq(abs(Out1));
}
if((Setpoint2 - Input2) < -0.5)
{
    AdelanteDer(abs(Out2));
}

if(movimiento == 0){
    Parar();
}
}

```

*Figura 4.8: Cómputo de las señales de actuación sobre cada rueda y condiciones para indicar el sentido de movimiento de las ruedas.*

Como vemos, esta parte está condicionada con el valor de “movimiento” como ya se comentaba, para tener “control” de cuando empezar a controlar.

Se computan las señales de actuación, PWM que se aplica a cada rueda, mediante ciertas funciones que se verán más adelante, en las que se hace uso de controladores tipo PI para este cálculo. También se aprecia como se asigna como referencias (“Input 1” e “Input2”) las distancias medidas anteriormente en cada uno de los ultrasonidos.

Entonces se definen una serie de condiciones de manera que se realiza lo siguiente:

- Si el error respecto a la referencia en valor absoluto (por delante o por detrás de la referencia) es inferior o igual a 0.5 cm, se para el robot. Esta condición quizás es muy restrictiva y provoca que el robot quede aplicando ciertas correcciones. Probablemente conviene ser un poco más flexible con este error para evitar actuaciones innecesarias y relativamente forzadas en los actuadores.
- Si el error es positivo y por encima de 0.5 cm, equivale a que el robot se ha pasado de la referencia y debe retroceder, como vemos haciendo uso de las funciones *AtrasIzq()* y *AtrasDer()*, aplicando las señales de actuación procedentes de los controladores.
- Si el error es negativo y por debajo de -0.5 cm, equivale a que el robot está más alejado de la pared que lo que indica la referencia y debe avanzar, como vemos haciendo uso de las funciones *AdelanteIzq()* y *AdelanteDer()*, aplicando las señales de actuación procedentes de los controladores.

Básicamente con esto se logra la funcionalidad solicitada en estos Modos 1 y 2, como comprobaremos en el siguiente apartado de resultados.

Para proseguir, tenemos ahora la parte de visualización de datos que permitirá hacer uso de la función proporcionada de Matlab: “telemetria.m” para generar gráficas con la información recogida a través del puerto serie a lo largo del experimento. Esto se realiza imprimiendo en cada ciclo la información por puerto serie formateada de la manera adecuada para que la función de Matlab la procese adecuadamente.

```

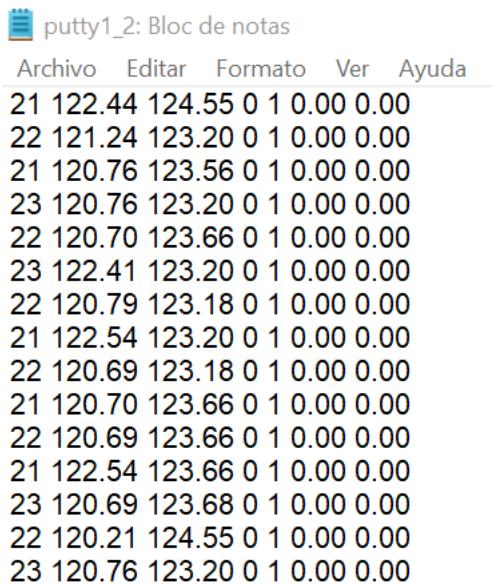
delay(5);
int tiempo_actual = millis();
int delta_tiempo = tiempo_actual-tiempo_anterior;

//Paso de valores para telemetría
Serial1.print(delta_tiempo);
Serial1.print(' ');
Serial1.print(DISTANCIA1);
Serial1.print(' ');
Serial1.print(DISTANCIA2);
Serial1.print(' ');
Serial1.print(referencia);
Serial1.print(' ');
Serial1.print(modo);
Serial1.print(' ');
Serial1.print(Out1);
Serial1.print(' ');
Serial1.println(Out2);

```

*Figura 4.9: Impresión de datos por puerto serie para obtención de gráficas.*

Básicamente se introduce un delay de 5 milisegundos para tener unos incrementos de tiempo razonablemente uniformes (la función “telemetría.m” está diseñada para recibir cadenas de información cada cierto incremento de tiempo, por tanto, el tiempo se tiene que mandar como incrementos). Con esto logramos obtener un “.log” con la información formateada por filas con espacios entre los diferentes datos, para que la telemetría pueda ser representada correctamente.



```

putty1_2: Bloc de notas
Archivo Editar Formato Ver Ayuda
21 122.44 124.55 0 1 0.00 0.00
22 121.24 123.20 0 1 0.00 0.00
21 120.76 123.56 0 1 0.00 0.00
23 120.76 123.20 0 1 0.00 0.00
22 120.70 123.66 0 1 0.00 0.00
23 122.41 123.20 0 1 0.00 0.00
22 120.79 123.18 0 1 0.00 0.00
21 122.54 123.20 0 1 0.00 0.00
22 120.69 123.18 0 1 0.00 0.00
21 120.70 123.66 0 1 0.00 0.00
22 120.69 123.66 0 1 0.00 0.00
21 122.54 123.66 0 1 0.00 0.00
23 120.69 123.68 0 1 0.00 0.00
22 120.21 124.55 0 1 0.00 0.00
23 120.76 123.20 0 1 0.00 0.00

```

**Figura 4.10:** Información formateada por filas con espacios entre los diferentes datos. Siendo: “incremento de tiempo en ms”, “distancia sensor\_1”, “distancia sensor\_2”, “referencia”, “modo”, “PWM rueda izquierda” y “PWM rueda derecha”.

Ahora, queda comentar una serie de aspectos más para culminar este código que engloba las funcionalidades de ambos modos 1 y 2.

```

/*
* Función de mapeo
double map(double valor, double entradaMin, double entradaMax, double salidaMin, double salidaMax)
{
    return (((valor-entradaMin)*(salidaMax-salidaMin))/(entradaMax-entradaMin))+salidaMin;
}

/*

```

**Figura 4.11:** Función que permite mapear un valor de un cierto rango de valores a otro.

Esta función de mapeado se utiliza fundamentalmente en los controladores para corregir el problema conocido del desfase entre ambas ruedas (con el mismo PWM la rueda izquierda va más rápido que la rueda derecha).

Finalmente, queda comentar la función "Compute()" usada para obtener el PWM necesario a aplicar a cada rueda para seguir cierta referencia (en este caso la distancia indicada por su sensor ultrasónico correspondiente). Como ambos son exactamente iguales, sólo explicaré por ejemplo el Controlador 1, que gestiona el PWM a aplicar a la rueda izquierda, y el Controlador 2 es totalmente análogo.

```
//Controlador PI 1
double Compute1(void)
{
    unsigned long nowl = millis(); // Toma el número total de milisegundos que hay en ese instante desde el comienzo de la ejecución del programa
    unsigned long timeChang1 = (nowl - lastTimel); // Resta el tiempo actual con el último tiempo que se guardó (esto último se hace al final de esta función)
    // Es la diferencia de tiempo que ha transcurrido desde que se entró por última vez a la función y la vez actual
    if(timeChang1 >= SampleTime1) // Si se cumple el tiempo de muestreo entonces calcula la salida
    {
        Input1 = DISTANCIAL; // Se toma como entrada la distancia medida por el sensor ultrasónico

        if(abs(Setpoint1 - Input1) <= 20){ // Si estamos cerca de la referencia introducimos parte integral, si no es así, actúa solo la parte proporcional
            kpl = 1.0;
            kil = 0.05;
        }
        else{
            kpl = 2.0;
            kil = 0.0;
        }

        error1 = Setpoint1 - Input1; // Calcula la parte proporcional como "Parte proporcional = Error * Kp"
        Prop1 = error1 * kpl;

        cumError1 += error1 * timeChang1; // Integral del error
        Integ1 = cumError1 * kil; // Calcula la parte integral como "Parte integral = Parte proporcional * Ki"

        // Acota el error integral para eliminar el "efecto windup".
        if (Integ1 > 10) Integ1 = 10;
        else if (Integ1 < -10) Integ1 = -10;

        double Output1 = Prop1 + Integ1; // Suma todos los errores, es la salida del control PI.

        //Se satura la señal de control (PWM) al rango [-255, 255]
        if (Output1 > outMax1) Output1 = outMax1;
        else if (Output1 < outMin1) Output1 = outMin1;

        lastTimel = nowl; // Se guarda el tiempo para usarlo como tiempo pasado en la siguiente iteración

        //Mapear la salida de control (PWM) al rango funcional
        if(Output1 >= 0.0) Output1 = map(Output1,0.0,255.0,110.0,255.0);
        if(Output1 < 0.0) Output1 = map(Output1,0.0,-255.0,-110.0,-255.0);

        return Output1; // Devuelve el valor de salida PID.
    }
}

//Mapear la salida de control (PWM) al rango funcional
if(Output2 >= 0.0) Output2 = map(Output2,0.0,255.0,140.0,255.0);
if(Output2 < 0.0) Output2 = map(Output2,0.0,-255.0,-140.0,-255.0);
```

*Figura 4.12: Función que implementa control tipo PI para la rueda izquierda en este caso.*

De nuevo, el código de la captura está comentado con bastante detalle, aunque sí es necesario hacer hincapié en ciertos detalles, como es la estrategia de control que se está llevando a cabo. Esta consiste en que, si estamos a más de 20 cm de distancia, se aplica un control exclusivamente de tipo P, con actuaciones mayores. Hasta que si entramos en ese rango del error inferior a 20 cm, además de rebajar a la mitad la ganancia proporcional, se incluye también la parte integral del controlador, para tratar de evitar errores no nulos en régimen permanente.

Como último detalle, si vemos la siguiente figura:

```
//Mapear la salida de control (PWM) al rango funcional
if(Output2 >= 0.0) Output2 = map(Output2,0.0,255.0,140.0,255.0);
if(Output2 < 0.0) Output2 = map(Output2,0.0,-255.0,-140.0,-255.0);
```

*Figura 4.13: Mapeado del PWM de la rueda derecha.*

Como se había mencionado, en este mapeado se compensa la "lentitud relativa" de la rueda derecha respecto a la izquierda para que el control sea coherente con el comportamiento de las ruedas y por tanto, se consiga los desplazamientos deseados por parte del robot. Estos rangos de mapeado se pueden ver afectados por la alimentación de las baterías.

### 4.3- Gráficas y resultados

Tras todo el conjunto de aspectos relacionados con las decisiones de diseño y montaje, así como la implementación del control y demás, es hora en este apartado de mostrar los resultados obtenidos con la implementación comentada anteriormente.

En primer lugar, se adjunta un video mostrando el test que se ha realizado para ambos modos en conjunto:

<https://drive.google.com/file/d/1OZ3Xxhlt1yRUcQTfWbvoBITmaxatGk1g/view?usp=sharing>

Partiendo de este video, se adjunta también una gráfica representativa de dicho test:

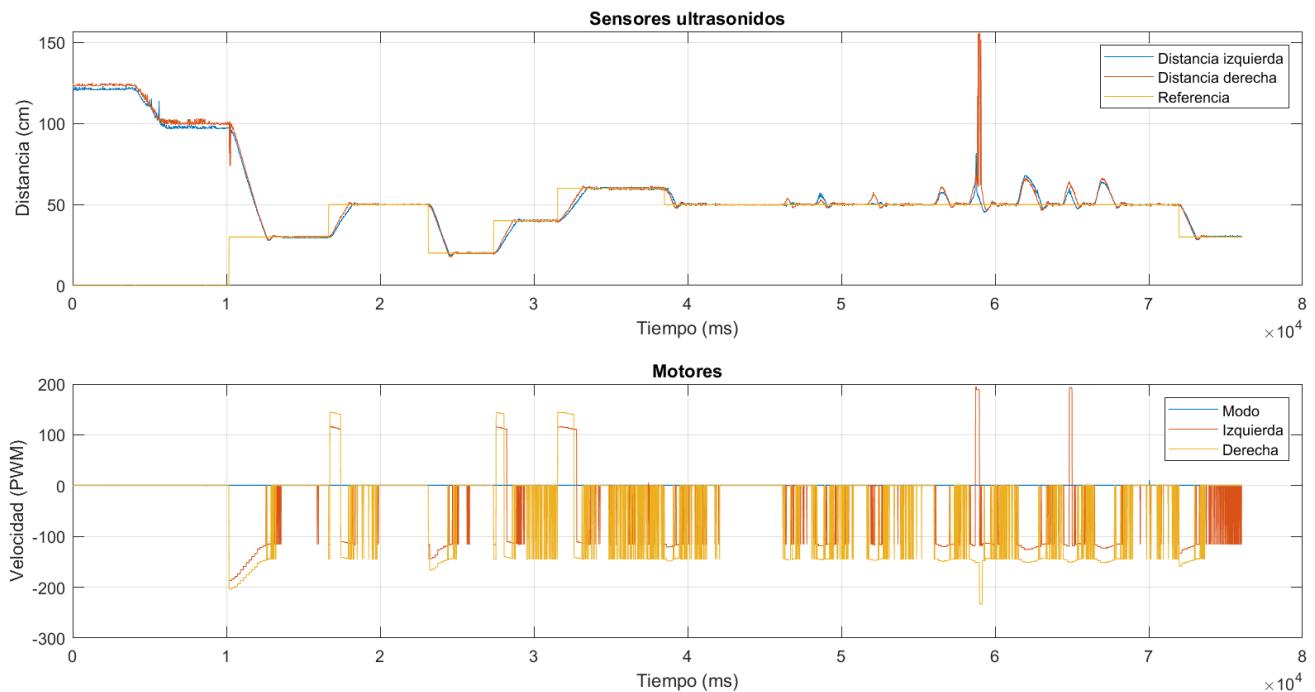


Figura 4.14: Gráfica correspondiente al test realizado para estos modos.

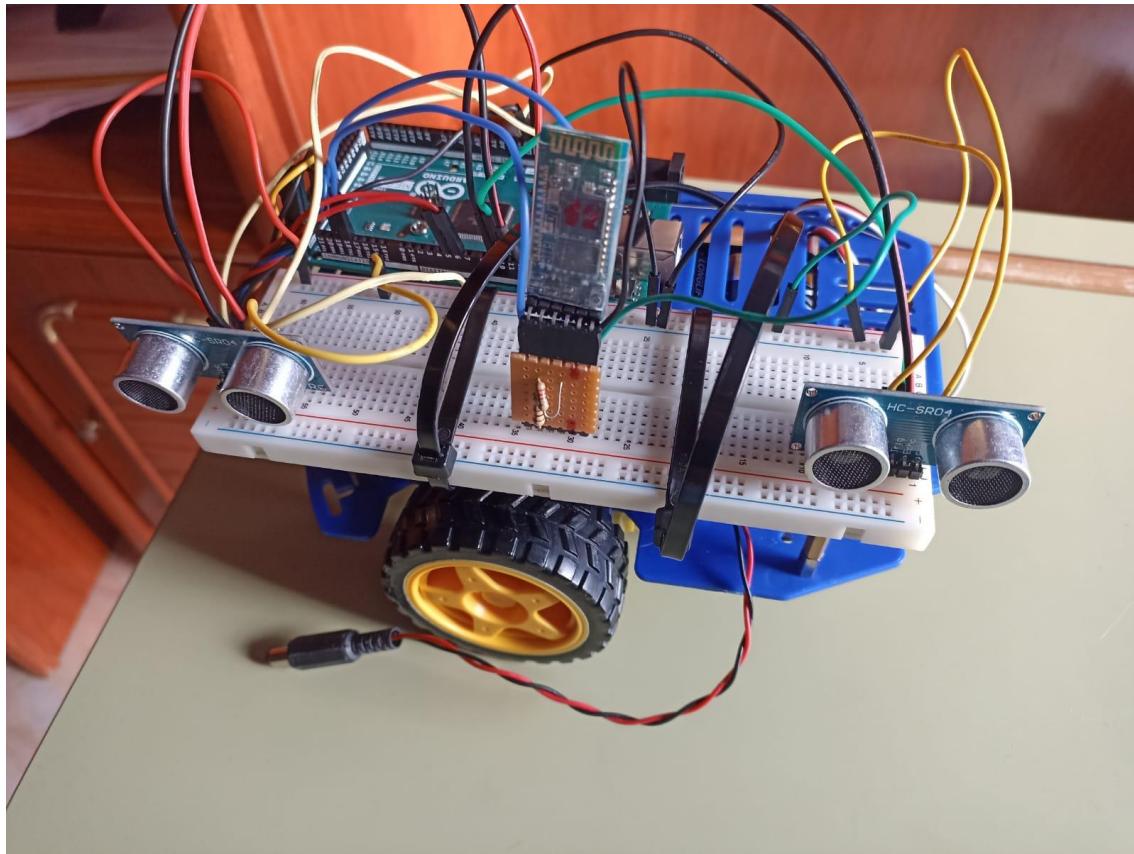
A partir de esta gráfica en conjunción con el video, podemos sacar varias conclusiones y reafirmar varios aspectos ya comentados anteriormente:

- Se comprueba como la condición de parar el robot al ser el error en valor absoluto respecto a la referencia inferior a 0,5 cm es demasiado estricto, en el sentido de que como vemos, los actuadores (motores), se paran y vuelven a actuar, realizando correcciones sin llegar a “pararse limpiamente”. Como se dijo, quizás convendría perder precisión siendo más flexibles con este error a cambio de no forzar tanto los motores.
- Como se observa en el experimento se parte del robot situado a 120 cm de la pared, manualmente se acerca a 100 cm (con el control desactivado aún).
- Una vez a 100cm, se activa el control con referencia de distancia de 30 cm, y vemos como el robot se acerca hasta quedarse a esa distancia de la pared.
- A continuación, vemos como se suceden una serie de cambios en la referencia, donde todos se alcanzan correctamente.
- Finalmente, tenemos la referencia fija a 50 cm y vemos ciertos cambios en las mediciones de distancia, estas se corresponden a las perturbaciones manuales

ejercidas sobre el robot, donde se puede ver las correcciones (en PWM) que realiza el mismo para llevar ambas distancias a 50 cm de nuevo y por tanto reorientarse o volver a la distancia requerida.

## 5.- MODO 3 – PROGRAMA QUE PERMITA QUE EL ROBOT SE DESPLACE PARALELO A UNA PARED

### 5.1- Montaje y configuración del robot para este modo



*Figura 5.1: Imagen del montaje de los sensores HC-SR04 para desplazamiento paralelo a una pared (MODOS 3 y 4).*

Como vemos, ahora tanto la protoboard como ambos ultrasonidos están colocados en la parte lateral del robot. Gracias a este montaje, podemos realizar el control de manera que el robot se desplace de manera paralela a una pared, como se solicita en este modo. Sin necesidad de volver a explicar el funcionamiento de los ultrasonidos ya comentados en el modo anterior, pasamos a la parte de programación de este modo.

## 5.2- Programación y resolución de este modo

En primer lugar, comenzamos declarando variables necesarias y asignando pines de nuevo.

### Modo\_3

```
#include <stdio.h>

// Motor A (derecha)
#define ENA 6
#define IN1 22
#define IN2 23

// Motor B (izquierda)
#define ENB 5
#define IN3 24
#define IN4 32

//Dos pines para TRIG1 y ECO1
int TRIG1 = 10; //Trigger al Pin 10 (PWM)
int ECO1 = 9; //ECO al Pin 9 (PWM)

//Dos pines para TRIG2 y ECO2
int TRIG2 = 12; //Trigger al Pin 12 (PWM)
int ECO2 = 13; //ECO al Pin 13 (PWM)

//Variables para comando de referencia y activación del control
int movimiento = 0;
int tipomov_y_referencia = 0;

//Variables de medida del primer ultrasonido:
int DURACION1; //Variable para medir la duración de transmisión/recepción
float DISTANCIA1; //Variable para medir la distancia a la pared u obstáculo.

//Variables de medida del segundo ultrasonido:
int DURACION2; //Variable para medir la duración de transmisión/recepción
float DISTANCIA2; //Variable para medir la distancia a la pared u obstáculo.

//Referencia constante para el modo 3 de 50 cm
double referencia=50.0;

//Variables de interés para la representación de la telemetría
int modo=4;
unsigned long tiempo;
unsigned long deltaT;
unsigned long tiempo_ant;

//Variables para el Buffer:
const int LongitudBuffer = 50;

//Variable para el error integral del controlador PI de la rueda derecha:
float eintegral1 = 0.0;
```

*Figura 5.2: Variables y asignación de pines para este modo.*

Aprovechando el detalle de los comentarios del código en las capturas que se adjuntan se considera que no es necesario explicar la función de cada variable de nuevo.

Atravesando la declaración de variables, entramos en el Setup, donde se configuran ciertos pines, incluyendo ahora los pines de los sensores ultrasónicos y la configuración de la comunicación con Bluetooth.

```
void setup()
{
    pinMode(TRIG1, OUTPUT);
    pinMode(ECO1, INPUT);
    pinMode(TRIG2, OUTPUT);
    pinMode(ECO2, INPUT);
    pinMode(ENA, OUTPUT);
    pinMode(ENB, OUTPUT);
    pinMode(IN1, OUTPUT);
    pinMode(IN2, OUTPUT);
    pinMode(IN3, OUTPUT);
    pinMode(IN4, OUTPUT);
    Serial1.begin(38400);
    Serial.begin(38400);

}
```

*Figura 5.3: Configuración de pines y bluetooth.*

Se omite tanto la explicación de las funciones para los distintos movimientos del robot, así como la del buffer, ya que sería muy repetitivo.

Pasamos por tanto al loop del programa:

```
void loop(){
    tiempo=millis();
    deltaT = tiempo-tiempo_ant;
    tiempo_ant = tiempo;

    if(Serial.available() > 0) //Para procesar los diferentes caracteres que hay en el buffer del serial, debemos quedarnos aquí hasta que llegue un \r
    {
        //Array/buffer para almacenar el mensaje entrante
        static char Buffer[LongitudBuffer];
        static unsigned int IndiceBuffer = 0;
        // Sigue ...
    }
}
```

*Figura 5.4: Comienzo del loop, omitiendo la parte del buffer.*

Como vemos, se captura el tiempo en el instante inicial y se calcula el incremento de tiempo entre ciclos de loop, principalmente para representación de telemetría y aplicación en el control. Prosiguiendo con el loop:

```
//Segmentación de datos entregados por puerto serie en "movimiento" y "referencia"
movimiento = tipomov_y_referencia/100;

if(movimiento == 1){
    referencia = tipomov_y_referencia%100; //La referencia ya se inicializa a 50 cm, pero habría que meter por Putty un 150 para mantener dicho valor
    referencia = (float)referencia;
}

//Ultrasonidos 1:
digitalWrite(TRIG1, HIGH);
delayMicroseconds(10); // Hay que enviar un pulso a nivel alto 1ms y luego a nivel bajo
digitalWrite(TRIG1, LOW);
DURACION1 = pulseIn(ECO1, HIGH);
DISTANCIA1 = DURACION1 / 58.2;

//Delay de 1 ms entre las lecturas de ambos sensores
delay(1);

//Ultrasonidos 2:
digitalWrite(TRIG2, HIGH);
delayMicroseconds(10); // Hay que enviar un pulso a nivel alto 1ms y luego a nivel bajo
digitalWrite(TRIG2, LOW);
DURACION2 = pulseIn(ECO2, HIGH);
DISTANCIA2 = DURACION2 / 58.2;
```

*Figura 5.5: Continuación del loop, cálculo de distancias por parte de los sensores.*

En este fragmento, básicamente, se vuelve a recibir la referencia por parte del buffer (aunque ya estuviese inicializada hay que volver a darle los 50 cm), y además también necesita un 1 delante de dicha referencia para que comience a funcionar, es decir, para este modo, habría que introducir por puerto serie (Putty) un "150".

Proseguimos con la parte que ejecuta el control:

```
//Controlar fijando velocidad rueda izquierda y controlando la de la rueda derecha con un control tipo PI
float kpi = 3.0;
float kii = 0.05; //Constantes del controlador PI para la rueda derecha

//Cálculo del error como la diferencia entre la referencia y la media de las distancias medidas por ambos sensores
float e1 = referencia - (DISTANCIA1+DISTANCIA2)/2.0;

//Saturación del error integral para evitar efecto Wind Up
if(eintegral1 >= 10.0) eintegral1 = 10.0;
if(eintegral1 < -10.0) eintegral1 = -10.0;

float u2 = kpi*e1 + kii*eintegral1; //Cálculo de las señales de control mediante control PI

int pwmizq = 80; //se fija la velocidad de la rueda izquierda
AdelanteIzq(pwmizq);

//La velocidad de la rueda derecha será tal que se impone un cierto valor constante más un aporte del controlador PI calculado según los sensores laterales
int pwmder = 98+u2;
AdelanteDer(pwmder);

//Se satura el pwm a su valor máximo
if(pwmizq > 255) pwmizq = 255;
if(pwmder > 255) pwmder = 255;

//AntiWindup, se satura el error integral en caso de superar los límites máximos de los actuadores (pwm máximo)
if (pwmder < 255.0) eintegral1 = eintegral1 + e1*deltaT;

//Se satura directamente el error integral sobre cierto valor, ya que el valor del antiwindup implica que acumula mucho error y empeora las actuaciones
if(eintegral1 >= 10.0) eintegral1 = 10.0;
if(eintegral1 < -10.0) eintegral1 = -10.0;

//No acumular error integral hasta que se active el control
if(movimiento == 0) {
    eintegral1 = 0.0;
    pwmizq = 0.0;
    pwmder = 0.0;
    Parar(); //que esté parado hasta que se active el control
}
```

**Figura 5.6:** Continuación del loop, implementación del control (basado en fijar las velocidades de una de las ruedas y la otra imponer un valor constante más un cierto aporte controlado).

Como se puede apreciar, el control se realiza fijando la rueda izquierda a una cierta velocidad constante, mientras que la rueda derecha tiene otro valor constante junto a un cierto aporte el cual si es controlado según el error (que es la diferencia entre la referencia y la media de las distancias medidas por ambos sensores).

Por último, se imprimen por puerto serie (Putty) ciertos valores de interés para poder representar gráficamente la telemetría del experimento:

```
//Paso de valores para telemetría
Serial1.print(deltaT);
Serial1.print(' ');
Serial1.print(DISTANCIA1);
Serial1.print(' ');
Serial1.print(DISTANCIA2);
Serial1.print(' ');
Serial1.print(pwmizq);
Serial1.print(' ');
Serial1.println(pwmder);

}
```

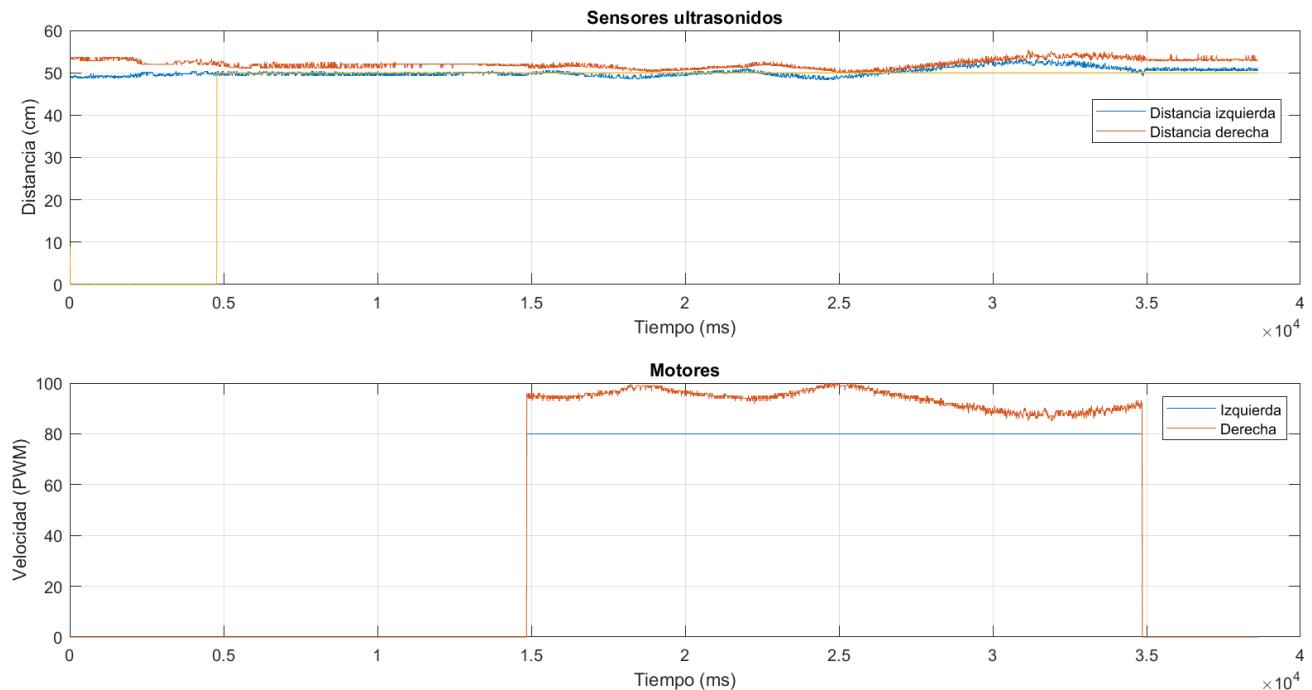
**Figura 5.7:** Continuación del loop, print de información relevante para telemetría.

### 5.3- Gráficas y resultados

En primer lugar, se adjunta un video mostrando el test que se ha realizado para este modo:

<https://drive.google.com/file/d/13LaqcAgyUA3997cIQSmUGAv4Og2Ji5AG/view?usp=sharing>

Partiendo de este video, se adjunta también una gráfica representativa de dicho test:



*Figura 5.8: Experimento realizado para el Modo 3, en el que se busca un avance paralelo a la pared a una distancia constante de 50 cm.*

A partir de esta gráfica en conjunción con el video, podemos sacar varias conclusiones:

- Vemos como aproximadamente, el robot ya está inicialmente colocado paralelo a la pared y a esa distancia de 50 cm.
- Entre los 0 y 5 primeros segundos se realiza la colocación del robot a esa distancia.
- Mientras se introducía el comando “150” para que el robot comience a moverse y además conserve dicha referencia de 50 cm, pasan unos 10 s.
- Por tanto, el robot pasa unos 20 s (de 15 a 35 segundos) avanzando con el control, manteniéndose aproximadamente paralelo y a la distancia impuesta.
- Finalmente, los últimos 5s de la simulación ya se ha parado el robot y es el tramo donde se desconecta Putty y termina el experimento.

## 6.- MODO 4 – PROGRAMA QUE PERMITA QUE EL ROBOT SE DESPLACE PARALELO A UNA PARED A LA DISTANCIA QUE SE LE VAYA INDICANDO

En primer lugar, podemos mencionar que el montaje es exactamente el mismo que en el apartado anterior. Esto es debido a que este apartado busca también un desplazamiento paralelo, con la diferencia de que ahora el robot deberá desplazarse en paralelo a una cierta distancia que se le solicite.

### 6.1- Programación y resolución de este modo

Aunque se entregan dos códigos, se va a explicar aquel con el que se obtienen los resultados que se van a mostrar, que es más específico y funcionaba para el test concreto solicitado.

Comenzaremos mostrando las variables utilizadas en este código:

```
Control_PI_Paralelo_Modo4

// Motor A (derecha)
#define ENA 6
#define IN1 22
#define IN2 23

// Motor B (izquierdo)
#define ENB 5
#define IN3 24
#define IN4 32

//Dos pines para TRIG1 y ECO1

int TRIG1 = 10; //Trigger al Pin 10 (PWM)
int ECO1 = 9; //ECO al Pin 9 (PWM)

//Dos pines para TRIG2 y ECO2

int TRIG2 = 12; //Trigger al Pin 12 (PWM)
int ECO2 = 13; //ECO al Pin 13 (PWM)

//Variable para recibir la referencia
unsigned long int movimiento_buffer = 0;
const int LongitudBuffer = 50;

int movimiento = 0;
unsigned long int tipomov_y_referencia = 0;
int flag_control_1 = 0;
int flag_control_2 = 0;
```

```

//Variables de medida del primer ultrasonido:
int DURACION1; //Variable para medir la duración de transmisión/recepción
double DISTANCIA1; //Variable para medir la distancia a la pared u obstáculo.

//Variables de medida del segundo ultrasonido:
int DURACION2; //Variable para medir la duración de transmisión/recepción
double DISTANCIA2; //Variable para medir la distancia a la pared u obstáculo.
double distancia2_ant;
double distancial_ant;

double referencia=50.0;
String StrUno = "Distancia: ";

int modo=4;
unsigned long tiempo;
unsigned long deltaT;
unsigned long tiempo_ant;

// **** Variables Globales PID1 ****
unsigned long lastTime1 = 0, SampleTime1 = 0; // Variables de tiempo discreto.
double Input1 = 0.0;
static double Setpoint1 = 0.0; // de posición del motor y posición a la que queremos llevar el motor (posición designada).
double ITerm1 = 0.0, dInput1 = 0.0, lastInput1 = 0.0; // de error integral, error derivativo y posición anterior del motor
double kp1 = 0.0, ki1 = 0.0, kd1 = 0.0; // Constantes: proporcional, integral y derivativa.
double outMin1 = 0.0, outMax1 = 0.0; // Límites para no sobrepasar la resolución del PWM.
double error1 = 0.0; // Desviación o error entre la posición real del motor y la posición designada.
double Out1 = 0.0;
// **** Variables Globales PID2 ****
unsigned long lastTime2 = 0, SampleTime2 = 0; // Variables de tiempo discreto.
double Input2 = 0.0;
static double Setpoint2 = 0.0; // de posición del motor y posición a la que queremos llevar el motor (posición designada).
double ITerm2 = 0.0, dInput2 = 0.0, lastInput2 = 0.0; // de error integral, error derivativo y posición anterior del motor
double kp2 = 0.0, ki2 = 0.0, kd2 = 0.0; // Constantes: proporcional, integral y derivativa.
double outMin2 = 0.0, outMax2 = 0.0; // Límites para no sobrepasar la resolución del PWM.
double error2 = 0.0; // Desviación o error entre la posición real del motor y la posición designada.
double Out2 = 0.0;
// ****

```

*Figura 6.1: Conjunto de todas las variables utilizadas para este modo.*

Se ha decidido agrupar todas las variables en una “figura grande” ya que carece de interés explicar de nuevo las variables.

Pasamos ahora a la parte del Setup, donde se configuran los pines y se inicializan las variables de los controladores.

```

void setup()
{
    Input1 = DISTANCIA1;
    Input2 = DISTANCIA2;
    pinMode(TRIG1, OUTPUT);
    pinMode(ECO1, INPUT);
    pinMode(TRIG2, OUTPUT);
    pinMode(ECO2, INPUT);
    pinMode(ENA, OUTPUT);
    pinMode(ENB, OUTPUT);
    pinMode(IN1, OUTPUT);
    pinMode(IN2, OUTPUT);
    pinMode(IN3, OUTPUT);
    pinMode(IN4, OUTPUT);
    Serial1.begin(38400);

    // Acotación máxima y mínima
    outMax1 = 255.0;
    outMin1 = -outMax1;

    outMax2 = outMax1;
    outMin2 = outMin1;

    SampleTime1 = 100;
    SampleTime2 = SampleTime1;

    kp1 = 8.0; //8 val
    ki1 = 0.0; //0.05 v
    kd1 = 0.0; //15 va

    kp2 = 8.0; //8 val
    ki2 = 0.0; //0.05 v
    kd2 = 0.0; //15 va
}

```

*Figura 6.2: Setup de los pines e inicialización de ciertas variables para los controladores.*

Si omitimos la parte donde se definen las variables para los diferentes posibles movimientos del robot, llegamos al loop, donde como de costumbre, empezamos con el cálculo de incrementos de tiempo entre ciclos del loop y el buffer de comunicaciones (el cual se omite para no ser repetitivo).

```
void loop() {
    tiempo=millis();
    deltaT = tiempo-tiempo_ant;
    tiempo_ant = tiempo;

    if(Serial1.available() > 0)
    {
        //Array/buffer para almacenar el mensaje entrante
        static char Buffer[LongitudBuffer];
        static unsigned int IndiceBuffer = 0;
        //Sigue...
    }
}
```

*Figura 6.3: Cálculo del incremento de tiempo en ms entre ciclos del loop y comienzo del buffer de comunicaciones.*

Este código que se comenta es bastante específico y está preparado para el test solicitado, de nuevo mencionar que se intentó hacer genérico y se adjunta en la entrega también pero no se pudo realizar experimentos con el mismo.

```
tipomov_y_referencia=movimiento_buffer;

//Referencia = 30cm
if(tipomov_y_referencia == 1){
    movimiento = 1;
}

if(tipomov_y_referencia == 2){
    movimiento = 2;
    flag_control_1 = 0;
    referencia = 30.0;
}

//Referencia = 40cm
if(tipomov_y_referencia == 3){
    movimiento = 2;
    flag_control_2 = 0;
    referencia = 40.0;
}

Setpoint1 = double(referencia);
Setpoint2 = Setpoint1;
```

*Figura 6.3: Asignación de la referencia de distancia para los controladores y habilitación del movimiento del robot.*

A continuación, se calculan las distancias respecto de la pared lateral, con un simple filtro para descartar cambios bruscos en la medida o incluso medidas negativas.

```
//Ultrasonidos 1:
digitalWrite(TRIG1, HIGH);
delayMicroseconds(10); // hay que enviar un pulso a nivel alto 1ms y luego a nivel bajo
digitalWrite(TRIG1, LOW);
DURACION1 = pulseIn(ECO1, HIGH); //ESTO TAMBIÉN ENTRA EN EL CICLO 1
DISTANCIA1 = DURACION1 / 58.2; //ESTO TAMBIÉN ENTRA EN EL CICLO 1

if(DISTANCIA1 < 0 || abs(DISTANCIA1-distancial_ant) >= 100){ //Si sale la DISTANCIA1 < 0 : se queda con el valor anterior
    DISTANCIA1 = distancial_ant;
}
else{ //Solo coge el valor si DISTANCIA1 > 0
    distancial_ant = DISTANCIA1;
}

//DELAY PEQUEÑITO EN MEDIO DE LAS DOS LECTURAS Y SEPARAR TODAS LAS INSTRUCCIONES
delay(1);

//Ultrasonidos 2:
digitalWrite(TRIG2, HIGH);
delayMicroseconds(10); // hay que enviar un pulso a nivel alto 1ms y luego a nivel bajo
digitalWrite(TRIG2, LOW);
DURACION2 = pulseIn(ECO2, HIGH); //ESTO TAMBIÉN ENTRA EN EL CICLO 1
DISTANCIA2 = DURACION2 / 58.2; //ESTO TAMBIÉN ENTRA EN EL CICLO 1

if(DISTANCIA2 < 0 || abs(DISTANCIA2-distancia2_ant) >= 100){ //Si sale la DISTANCIA2 < 0 : se queda con el valor anterior
    DISTANCIA2 = distancia2_ant;
}
else{ //Solo coge el valor si DISTANCIA2 > 0
    distancia2_ant = DISTANCIA2;
}
```

*Figura 6.4:* Cálculo de las distancias laterales de ambos sensores con un pequeño filtro de medidas irregulares.

A continuación, se comienza con un movimiento asignado al robot para que según sea la referencia, realice un movimiento de desvío hacia la pared o hacia fuera de la pared, jugando principalmente con el desfase de velocidades entre ambas ruedas, de manera que como la izquierda tiene mayor velocidad, en el caso de 50 a 30 el robot iría hacia la pared, mientras que en el caso de 30 a 40, es necesario darle un apoyo a la velocidad de la rueda derecha para que el robot se desvíe hacia afuera de la pared:

```
if(movimiento == 1) Parar(); //que esté parado hasta que se le diga que avance o retroceda.

if(movimiento==2 && referencia==30.0 && DISTANCIA1>=35 && DISTANCIA2>=35){ //Caso para pasar de 50 a 30
//(movimiento==1 && referencia==30 && abs(Setpoint1 - Input1) >= 15.0)

    AdelanteIzq(abs(Out1));
    AdelanteDer(abs(Out2));
}

if(movimiento == 2 && referencia==40.0 && DISTANCIA1<=35 && DISTANCIA2<=35){ //Caso para pasar de 30 a 40
//(movimiento==1 && referencia==40 && abs(Setpoint1 - Input1) >= 5.0)
    AdelanteIzq(abs(Out1));
    AdelanteDer(abs(Out2)*1.10);
}
```

*Figura 6.5:* Movimiento asignado a cada referencia para "introducirse" hacia la pared o "salirse" hacia afuera.

Entramos ahora en el caso en que la referencia sea 30 y el robot se está desviando (acermando) hacia la pared.

```

if(movimiento == 2 && referencia == 30.0 && DISTANCIA1 < 35){ //Para empezar a controlar
    flag_control_1 = 1;
}

if(movimiento == 2 && referencia == 30.0 && flag_control_1 == 1){
    //***** CASO COCHE PARALELO A LA PARED *****
    if (abs(Setpoint1 - Input1) <= 0.5)          //Si ambos errores laterales son prácticamente nulos,
    {
        AdelanteIzq(abs(Out1));
    }
    if (abs(Setpoint2 - Input2) <= 0.5)          //Si ambos errores laterales son prácticamente nulos,
    {
        AdelanteDer(abs(Out2));
    }

    //***** CASO MORRO DEL COCHE APUNTANDO A LA PARED *****
    if((Setpoint1 - Input1) > 0.5)
    {
        AdelanteIzq(abs(Out1));
    }
    if((Setpoint2 - Input2) < -0.5)           // De no ser igual, significa que el motor ha de girar en
    {
        AdelanteDer(abs(Out2)*1.30);
    }

    //***** CASO MORRO DEL COCHE APUNTANDO "AFUERA" DE LA PARED *****
    if((Setpoint1 - Input1) < -0.5)           // De no ser igual, significa
    {
        AdelanteIzq(abs(Out1)*1.40);
    }

    if((Setpoint2 - Input2) > 0.5)           // De no ser igual, significa
    {
        AdelanteDer(abs(Out2)*1.20);
    }
}

```

*Figura 6.6: Control para tratar de mantener el robot paralelo a la pared cuando la referencia es 30 cm y consigue llegar a dicha distancia.*

Pasamos ahora a la referencia de 40 cm, donde se realiza algo similar a lo anterior para que una vez el robot estuviese ya paralelo a 30 cm y se haga este cambio de referencia el robot se aleje de la pared buscando esos 40 cm y al llegar se estabilice.

```
/*
if(movimiento == 2 && referencia == 40.0 && DISTANCIA1 > 32){ //Para empezar a controlar
    flag_control_2 = 1;
}

if(movimiento == 2 && referencia == 40.0 && flag_control_2 == 1){
    /* CASO COCHE PARALELO A LA PARED ****
    if (abs(Setpoint1 - Input1) <= 0.5)      //Si ambos errores laterales son prácticamente nulos, av
    {
        AdelanteIzq(abs(Out1));
    }
    if (abs(Setpoint2 - Input2) <= 0.5)      //Si ambos errores laterales son prácticamente nulos, av
    {
        AdelanteDer(abs(Out2));
    }

    /* CASO MORRO DEL COCHE APUNTANDO A LA PARED ****
    if((Setpoint1 - Input1) > 0.5)
    {
        AdelanteIzq(abs(Out1));
    }
    if((Setpoint2 - Input2) < -0.5)           // De no ser igual, significa que el motor ha de girar en u
    {
        AdelanteDer(abs(Out2)*1.40);
    }

    /* CASO MORRO DEL COCHE APUNTANDO "AFUERA" DE LA PARED ****
    if((Setpoint1 - Input1) < -0.5)           // De no ser igual, significa
    {
        AdelanteIzq(abs(Out1)*1.30);
    }

    if((Setpoint2 - Input2) > 0.5)             // De no ser igual, significa que
    {
        AdelanteDer(abs(Out2)*1.10);
    }
}
*/

```

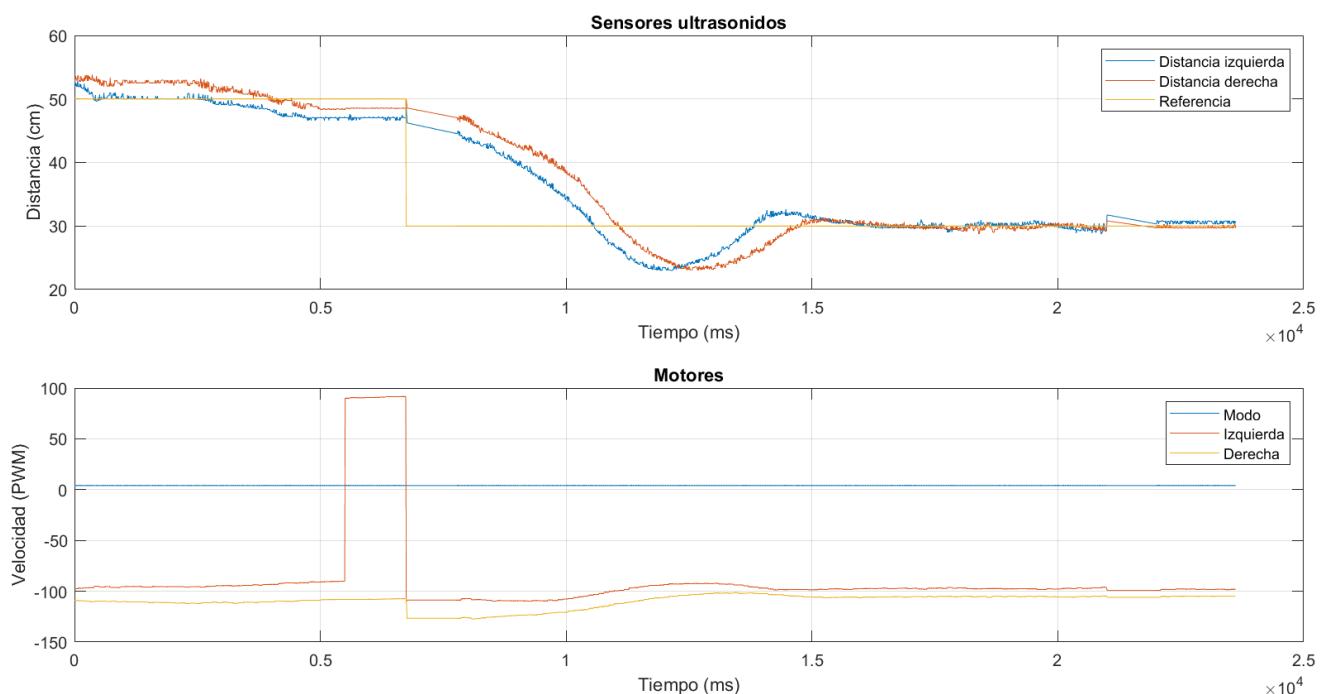
*Figura 6.7: Control para tratar de mantener el robot paralelo a la pared cuando la referencia es 40 cm y consigue llegar a dicha distancia.*

Pasando esta parte del control, sólo quedan una serie de prints, para obtener la telemetría, así como la definición de varias funciones más utilizadas a lo largo del código, como son: map(), Compute1() y Compute2(). Las capturas de estos elementos se van a omitir ya que no aportan tanto como las anteriores y se alargaría sin necesidad.

## 6.2- Gráficas y resultados

En este caso, perdimos el vídeo del experimento que teníamos, y por falta de tiempo no hemos podido replicarlo, pero si que se conservaron tanto el .log como las gráficas que obtuvimos. Otro aspecto a mencionar, es que por falta de espacio en el avance del robot, el experimento se realizó en dos fases, es decir, se partía de 50 cm de distancia lateral, se le indicaba referencia 30 cm y llegaba a dicha distancia, el caso es que una vez llegaba (realicé yo el experimento en mi casa) ya no tenía más espacio para seguir avanzando. Entonces lo que se hizo fue emular que el robot ya había llegado a esos 30 cm y se ha estabilizado, haciéndolo partir de 30 cm desde el principio del espacio disponible y entonces se le indicaba el cambio de referencia a 40 cm, para emular el experimento completo pero en dos fases, por dicha falta de espacio.

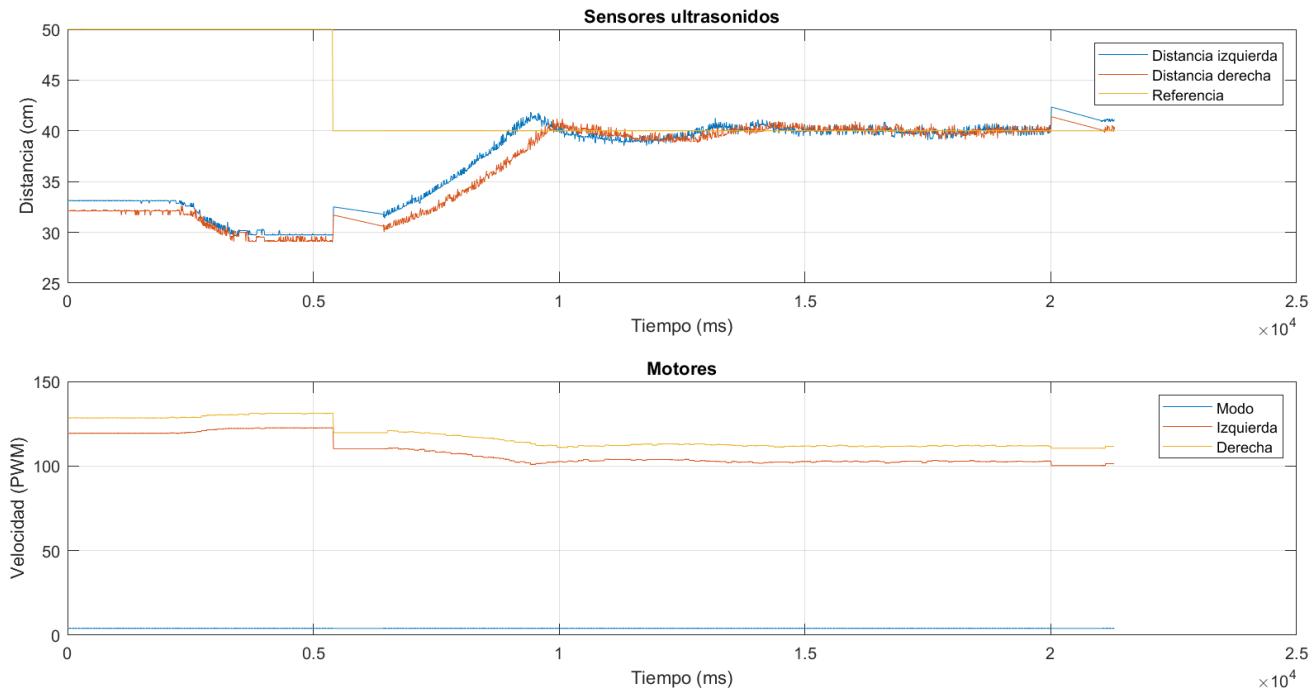
Con esto pasamos a mostrar los resultados obtenidos en ambos tests:



*Figura 6.8: Test para hacer el cambio de referencia de 50 cm a 30 cm lateralmente a la pared.*

De esta primera gráfica, obtenemos varias conclusiones:

- Por la dificultad de lograr estabilidad lateral, el control debe ser suave y además es necesario que sobreoscile para que pueda corregir y estabilizarse en la nueva referencia.
- Se necesita un cierto tiempo para lograr la estabilidad en la nueva referencia, lo que implica que, como se ha comentado, el robot se queda sin espacio de avance una vez se estabiliza.
- Las velocidades aparecen gráficamente como valores negativos. Esto se debe a que se debería haber representado el valor absoluto de PWM de cada rueda, que es realmente lo que se aplica a ambos motores, sin embargo, en su momento se graficó este valor obtenido de los controladores con su signo. Como se ha dicho, faltaba tiempo para rehacer y arreglar esto y sacar el vídeo extraviado.



*Figura 6.8: Test para hacer el cambio de referencia de 30 cm a 40 cm lateralmente a la pared.*

Como hemos mencionado, por la falta de espacio, se ha tenido que emular la continuidad del test colocando el robot inicialmente a 30 cm y comenzando parado indicarle el cambio de referencia a 40 cm. Por ello la primera fase de la gráfica representa la colocación manual del robot a eso 30 cm. De este segundo test, sacamos también algunas conclusiones:

- El control depende mucho de la orientación relativa que lleve el robot respecto de la pared, es decir, es notorio que de 50 a 30 cm, el robot llega más desviado (menos paralelo), luego el control vemos que tardaba más en estabilizarse y sobreoscilaba en mayor medida que ahora
- Como en este caso el cambio de referencia es menor (10 cm respecto de la posición del robot), el robot no pierde tanto el paralelismo respecto de la pared y por tanto, como vemos, el control necesita mucho menos esfuerzo y tiempo para estabilizar al robot en la nueva referencia.
- De nuevo ocurre lo mismo con las velocidades sólo que en este caso sus valores son positivos porque la distancia es menor que la referencia, sin embargo, se debería haber representado el valor absoluto de las mismas como se mencionaba previamente.

## 7.- MODO 5 – MONTAR SENSORES DE VELOCIDAD Y REALIZAR TEST QUE MUESTRE LA RESPUESTA ANTE ESCALONES DE VELOCIDAD

### 7.1- Montaje y configuración del robot para este modo

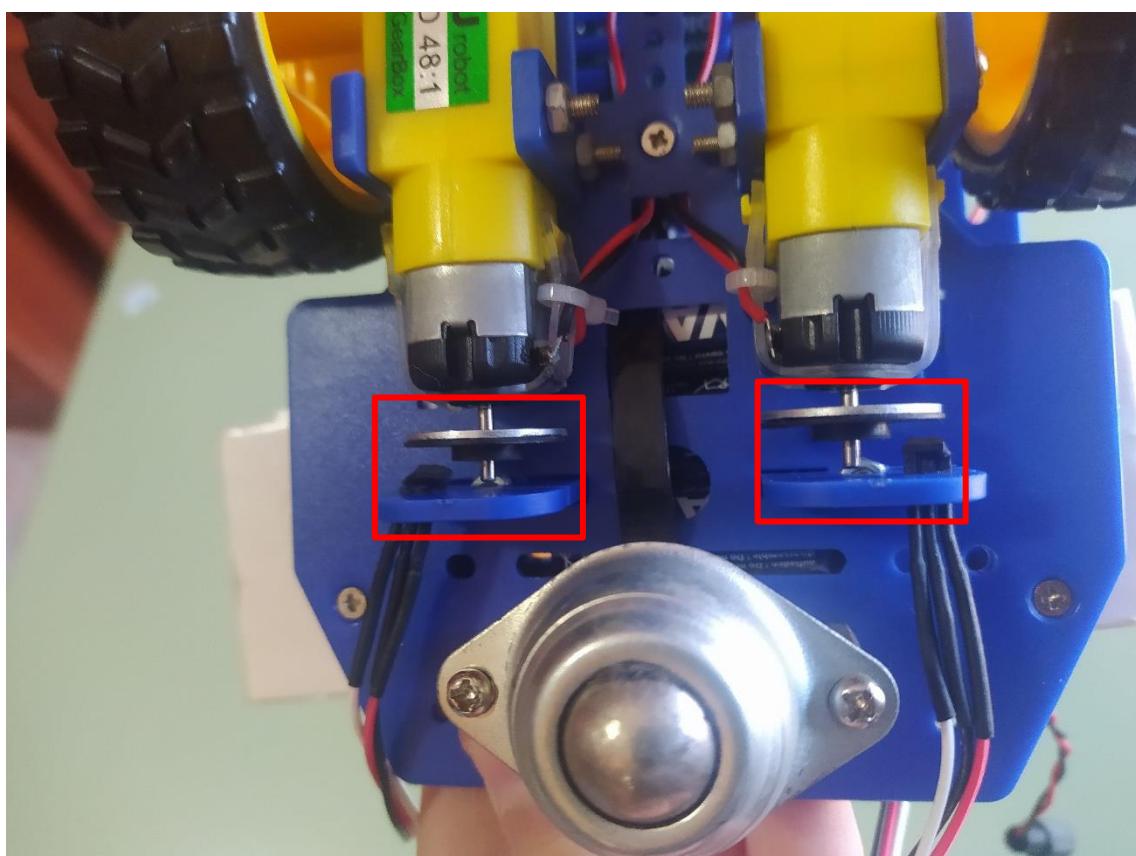
#### 7.1.1- Conceptos teóricos a aplicar

En este modo, necesitamos tener ahora una realimentación sensorial de las velocidades de giro de cada motor, por tanto, aparece la necesidad de hacer uso de sensores, concretamente, los sensores **encoders incrementales magnéticos** que forman parte del DAGU Simple Encoder Kit RS030. Mediante ellos, podemos conocer la velocidad de giro de cada motor, que aparte de permitirnos realizar un control en velocidad de cada rueda, permitirá en apartados posteriores realizar la estimación de posición a partir de las velocidades de cada rueda de un robot diferencial (Odometría). Podemos empezar explicando brevemente el funcionamiento de estos encoders como introducción a la resolución de este modo, previo a mostrar el montaje de estos sensores en el propio robot.

Enlace a gif descriptivo del funcionamiento de sensores tipo Hall para detección de variaciones de campo magnético (principio de funcionamiento de estos encoders):

[https://drive.google.com/file/d/1OXRFFkA2mcH1M5\\_YpUmrr7KNUVMUqmz1/view?usp=sharing](https://drive.google.com/file/d/1OXRFFkA2mcH1M5_YpUmrr7KNUVMUqmz1/view?usp=sharing)

De esta manera recordando cómo eran los encoders y sus conexiones:



*Figura 7.1: Colocación de los encoders para obtención de velocidades de cada motor.*

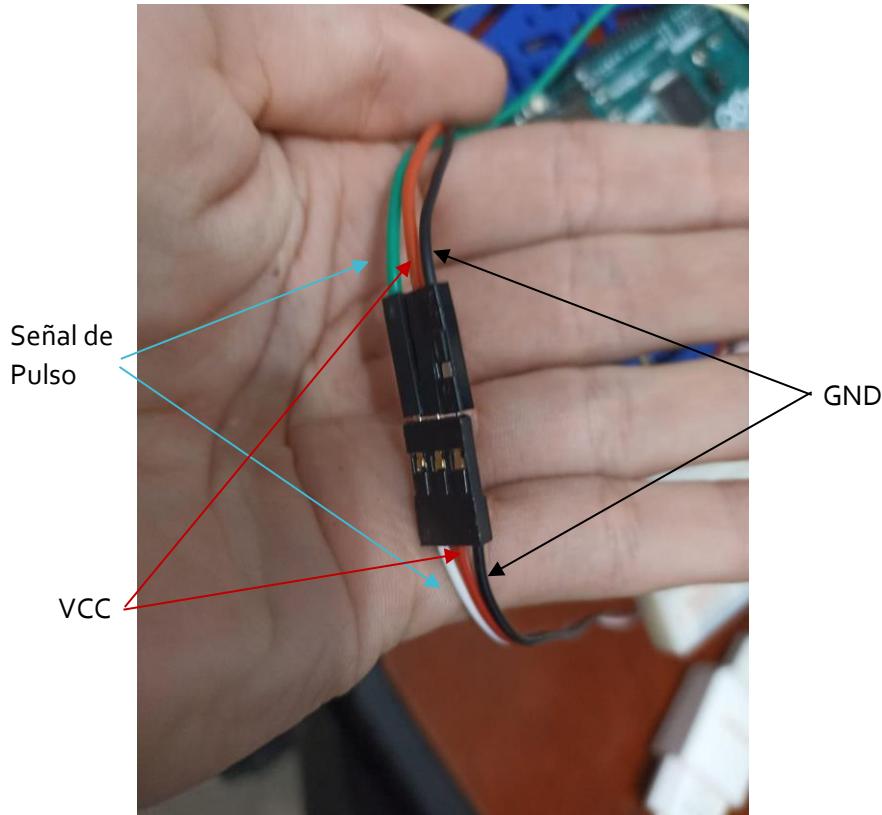


Figura 7.2: Conexionado de encoder magnético.

Visto el conexionado, la instalación de los encoders en el robot móvil y el principio de funcionamiento basado en el efecto Hall, podemos acudir al siguiente dibujo ilustrativo que indica la información que aportan estos sensores.

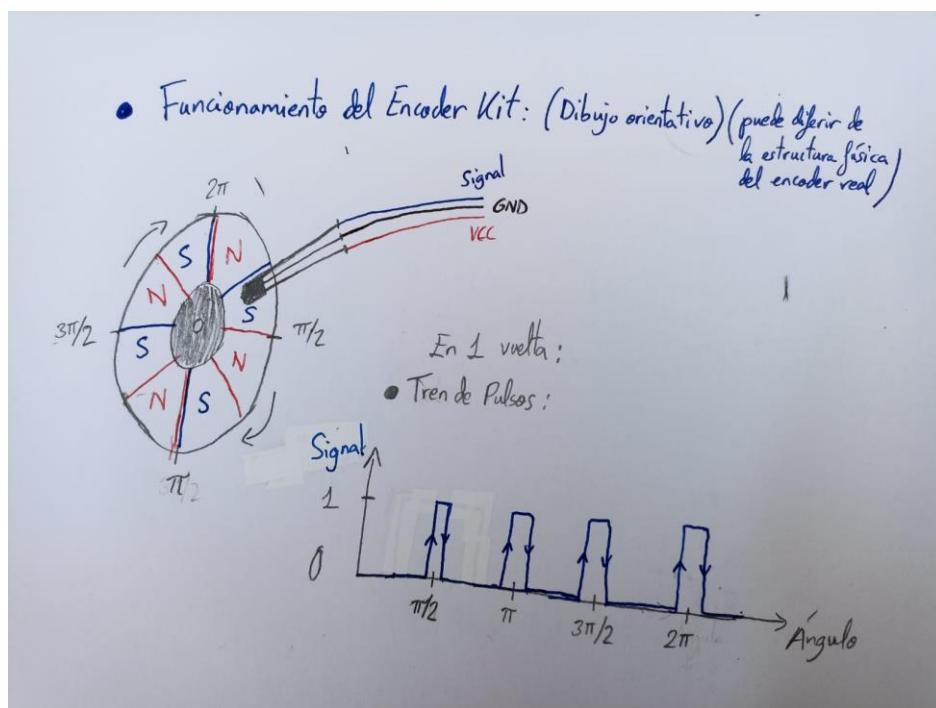


Figura 7.3: Dibujo ilustrativo del funcionamiento del encoder magnético.

Con este dibujo y sabiendo del folleto de instrucciones proporcionado que tiene 8 polos magnéticos, confirmando con las señales en el pin de Arduino, tenemos un tren de pulsos similar al de este dibujo.

De esta manera, podemos contar los pulsos de diferentes maneras desde Arduino:

- **Por flanco de subida:**

Tendremos 4 pulsos por vuelta del "disco" del encoder kit.

- **Por flanco de bajada:**

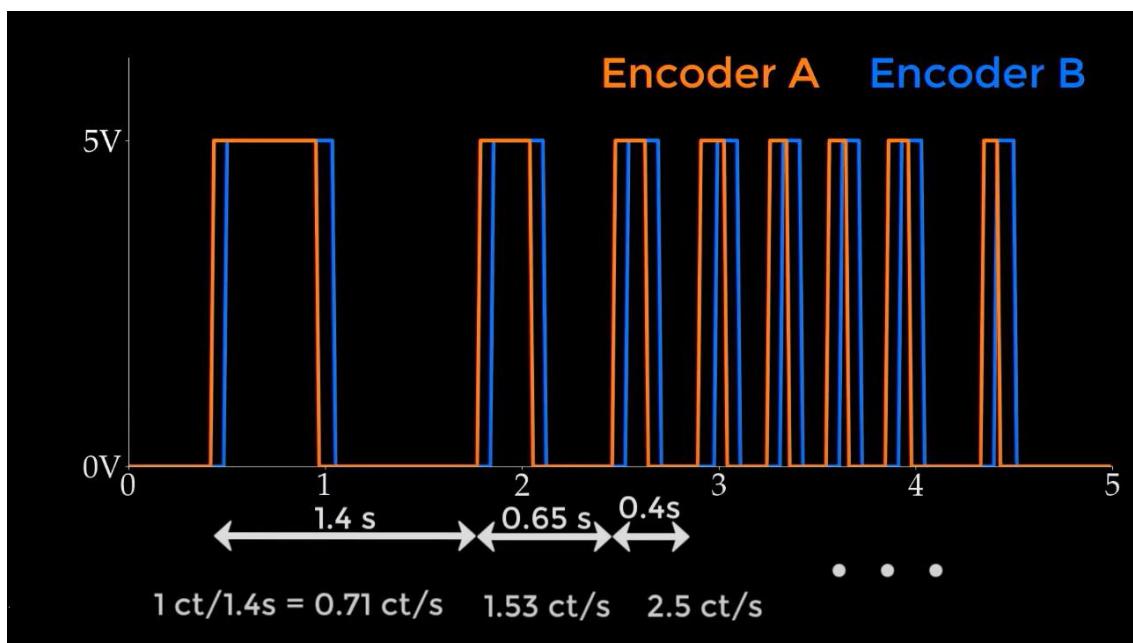
Tendremos 4 pulsos por vuelta del "disco" del encoder kit.

- **Por flanco de subida y de bajada:**

Tendremos 8 pulsos por vuelta del "disco" del encoder kit.

Partiendo de esto, podemos realizar ahora el cálculo de la velocidad de cada rueda. Para ello, se hace mediante el siguiente método, que se puede denominar como:

**"Medida del tiempo entre interrupciones"** (frecuencia con la que se entra en la interrupción)



*Figura 7.4: Esquema ilustrativo de cálculo de velocidad a partir de tics de encoder y frecuencia de entrada a interrupciones.*

Como vemos y se deduce del nombre del método, el cálculo de la velocidad se realiza a partir de la frecuencia con la que saltan las interrupciones (triggers), siendo que se entrará en estas interrupciones cada vez que se detecte un pulso o tic de los anteriormente descritos. De esta manera como se observa en la gráfica, si el tiempo entre una interrupción y otra es mayor, la velocidad será menor, mientras que, si el tiempo entre interrupciones se reduce, implica que la velocidad está aumentando.

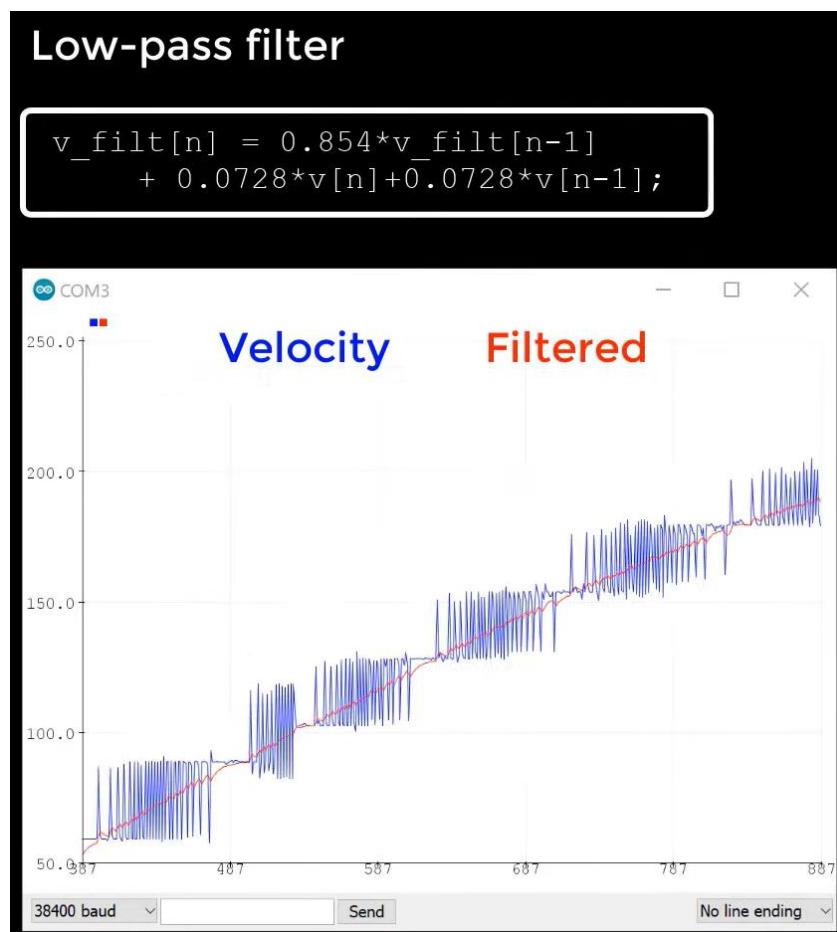
Entendido el mecanismo por el cual se va a calcular la velocidad, vamos a ahondar un poco más en cómo se obtiene numéricamente la velocidad a partir de este conteo de tics y frecuencia con la que entramos en las interrupciones.

Simplemente necesitamos aplicar la siguiente fórmula para transformar la frecuencia de entrada en la interrupción (tics/s) a velocidad angular (RPM):

$$V[\text{rpm}] = \frac{\text{Frecuencia} \left[ \frac{\text{tics}}{\text{s}} \right]}{\text{ResoluciónRueda} \left[ \frac{\text{tics}}{\text{VueltaRueda}} \right]} * 60.0 \left[ \frac{\text{s}}{\text{min}} \right] = \left[ \frac{\text{vueltasRueda}}{\text{min}} \right] = [\text{RPM}]$$

A continuación, una vez se obtienen las velocidades de las ruedas en RPM, aunque no es del todo necesario, ya que las señales son relativamente limpias, se realiza un filtrado en frecuencia tipo Paso Baja (Low Pass), para eliminar las componentes de ruido de alta frecuencia, en caso de que por el método de cálculo aplicado este ruido no exista, simplemente el filtrado no tendrá efecto útil.

La idea de este filtro es útil en el caso de que se compute la velocidad mediante el conteo de pulsos o tics de encoder en ciertos intervalos de tiempo fijos, siendo que este método si induce ruido a las medidas como se observa en la siguiente figura:



**Figura 7.5:** Mejora del ruido de la señal de velocidad con filtro LowPass en caso de que se haga el cálculo con el método de conteo de tics por intervalo de tiempo fijo.

**Recalcá** que en este caso la velocidad se ha calculado mediante el método de la frecuencia de entrada a las interrupciones, siendo que este método no induce tanto ruido, aunque aún así se ha implementado este filtro comentado de igual manera para garantizar tener medidas de velocidad lo más limpias posible.

Respecto del control en velocidad, mencionar simplemente que se va a seguir haciendo uso de controladores tipo PI, pero que se ha hecho un cambio en la estructura de dichos controladores, en lugar de usar las funciones Compute() como hasta ahora, se implementa ahora para este caso y en adelante, dentro del loop, como veremos al analizar el código.

Básicamente la función de los controladores en velocidad será:

- Dada una referencia en RPM para su rueda correspondiente, aplicará los valores de PWM necesarios para que mediante la información de velocidad ya procesada en RPM que le llega de los encoders, la rueda gire a la velocidad deseada e indicada mediante dicha referencia dada.

Habiendo comentado con esto los principales aspectos a destacar acerca de los sensores de velocidad y el control en velocidad, podemos pasar al apartado de programación para este modo.

## 7.2- Programación y resolución de este modo

Sin muchos preámbulos podemos empezar a analizar el código de este modo, que será además el que se use para el MODO 6, ya ese modo simplemente consiste en un test en el que el robot avance en línea recta gracias al control de velocidad, implementado en el código que analizaremos ahora, por tanto, se omitirá su explicación en el apartado del MODO 6.

Empezamos con la inclusión de librerías y variables necesarias:

```
Control_Modo_5_y_6
#include <stdio.h>
#include <util/atomic.h>

// Motor A (derecha)
#define ENA 6
#define IN1 22
#define IN2 23

// Motor B (izquierda)
#define ENB 5
#define IN3 24
#define IN4 32

//Deben ser el 2 y el 3 ya que en MEGA 2560, estos son pines habilitados para uso de interrupciones
//Pin para Encoder Derecho
int SignalEncoder_Derecha = 2;

//Pin para Encoder Izquierdo
int SignalEncoder_Izquierda = 3;

//Variables para recibir la referencia de velocidad a través del buffer que guarda los datos del puerto serie
unsigned long int velocidad_buffer = 0;
unsigned long int velocidad_izq_ref = 0;
unsigned long int velocidad_der_ref = 0;
```

*Figura 7.6: Inclusión de librerías, asignación de pines y declaración de variables.*

Seguimos con más variables:

```
unsigned long tiempo = 0;
unsigned long tiempo_ant = 0;

//Variables para el Buffer:
const int LongitudBuffer = 50;

//Creación de variables volátiles para aquellas que se utilizan en una interrupción
volatile int ContTicsIzq = 0;
volatile int ContTicsDer = 0;
volatile float frecuencia_izq = 0;
volatile float frecuencia_der = 0;
volatile long prevTIzq = 0;
volatile long prevTDer = 0;

//Variables para realizar el filtrado de la velocidad calculada (filtro LowPass)
float VIZqFilt = 0;
float VizqPrev = 0;
float VDerFilt = 0;
float VDerPrev = 0;

//Variables para el error integral de los controladores PI
float eintegral1 = 0;
float eintegral2 = 0;
```

*Figura 7.7: Resto de variables.*

Tras las explicaciones anteriores y los nombres explícitos de las variables no tiene mucho sentido explicar la funcionalidad concreta de cada variable, que además se irá viendo a lo largo del resto del código.

Vamos con el Setup:

```
void setup()
{
    pinMode(SignalEncoder_Derecha, INPUT_PULLUP); //Pin para el encoder derecho como entrada con resistencia de pull_up
    attachInterrupt(digitalPinToInterrupt(SignalEncoder_Derecha), EncoderDerecha, RISING);
    //Se va a contar los pasos POR POLOS del encoder por interrupción, para ello se indica el pin que
    //dispara la interrupción (solo por flanco de subida, ya que el pin se pondrá a 1 cada vez que pase por un polo)
    // y la rutina de interrupción que en este caso será EncoderDerecha().

    pinMode(SignalEncoder_Izquierda, INPUT_PULLUP); //Pin para el encoder izquierdo como entrada con resistencia de pull_up
    attachInterrupt(digitalPinToInterrupt(SignalEncoder_Izquierda), EncoderIzquierda, RISING);
    pinMode (ENA, OUTPUT);
    pinMode (ENB, OUTPUT);
    pinMode (IN1, OUTPUT); //Pines para el control de motores mediante Puente-H
    pinMode (IN2, OUTPUT);
    pinMode (IN3, OUTPUT);
    pinMode (IN4, OUTPUT);
    Serial1.begin(38400); //Setup de los puertos serie tanto para usar bluetooth, como para usar "monitor serial" y "serial plotter" del IDE de Arduino
    Serial.begin(38400);
}
```

*Figura 7.8: Configuración de pines e interrupciones en la función Setup.*

Como vemos, se declaran los pines asignados a los encoders como entradas con resistencia de pull up y se les asocia una interrupción a cada uno. Estas interrupciones se configuran tipo RISING, es decir, que se activen por flanco de subida. Como ya se ha comentado, son clave para el cálculo de la velocidad (mediante la frecuencia con la que entramos en ellas).

Dando el resto por explicado de ejemplos anteriores, proseguimos con el loop.

Omitiré la captura con la implementación del buffer ya que es exactamente igual que en los casos anteriores y no merece la pena.

```
//Buffer aparece anteriormente...
//Se registra el tiempo al inicio del loop
int tiempo_ant = millis();

//Separamos las referencias de velocidad, que vienen dadas como un numero de 4 cifras por puerto serie siendo p.ej. 2040, con 20 la referencia en [rpm]
//para el controlador de la rueda izquierda y 40 la referencia en [rpm] para el controlador de la rueda derecha
velocidad_izq_ref = velocidad_buffer/100;
velocidad_der_ref = velocidad_buffer%100;
int parar = velocidad_buffer;

//Se lee el número de tics en el intervalo de lectura dentro de un bloque ATOMIC para evitar potenciales errores en las lecturas
int tics_izquierda = 0;      //Contador tics_izquierda
int tics_derecha = 0;        //Contador tics_derecha
float frecuenciaIzq = 0.0;   //Frecuencia con la que se entra en la interrupción del encoder izquierdo
float frecuenciaDer = 0.0;   //Frecuencia con la que se entra en la interrupción del encoder derecho
ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
    tics_izquierda = ContTicsIzq;
    frecuenciaIzq = frecuencia_izq; //Se recogen los valores procedentes de las interrupciones
    tics_derecha = ContTicsDer;
    frecuenciaDer = frecuencia_der;
}

//Cálculo del lapso de tiempo entre el instante anterior y el actual para el control del motor izquierdo
long currTIZq = micros();
float deltaTimeIzq = ((float) (currTIZq-prevTIZq))/1.0e6; //Obtención de incrementos de tiempo para el cálculo del error integral izquierdo

//Cálculo del lapso de tiempo entre el instante anterior y el actual para el control del motor derecho
long currTDer = micros();
float deltaTimeDer = ((float) (currTDer-prevTDer))/1.0e6; //Obtención de incrementos de tiempo para el cálculo del error integral derecho
```

*Figura 7.9: Primera parte del Loop para este código.*

Aunque los comentarios están bastante detallados en la captura, es interesante hacer hincapié en el uso del bloque ATOMIC, una funcionalidad de Arduino que permite interaccionar de manera segura con las variables que se ven alteradas dentro de interrupciones, aún así, debe usarse con cuidado y tratar de permanecer dentro de él el menor tiempo posible ya que es bloqueante. Por ello, su única tarea útil, es recoger principalmente las frecuencias calculadas en las interrupciones, que como ya sabemos serán usadas a continuación para el cálculo de la velocidad.

```
// Convertir tics/s a RPM
float Vizq = frecuenciaIzq/182.5*60.0;
float Vder = frecuenciaDer/198.0*60.0;

// Filtro LP(Low-pass filter) (25 Hz cutoff) para limpiar la señal ruidosa de la velocidad que se entrega a los controladores de ambas ruedas
VIZqFilt = 0.854*VIZqFilt + 0.0728*Vizq + 0.0728*VizqPrev;
VizqPrev = Vizq;
VDerFilt = 0.854*VDerFilt + 0.0728*Vder + 0.0728*VDerPrev;
VDerPrev = Vder;

// Se establecen los valores de las referencias en velocidad en [RPM]
float VRefIzq = (float)velocidad_izq_ref;
float VRefDer = (float)velocidad_der_ref;

// Computar las señales de control u1 y u2 (mediante controladores PI)
float kp1 = 10;
float ki1 = 50; //Constantes de los controladores PI para cada rueda
float kp2 = 10;
float ki2 = 50;
float e1 = VRefIzq-VIZqFilt; //Cálculo de los errores para cada rueda
float e2 = VRefDer-VDerFilt;

float u1 = kp1*e1 + ki1*eintegral1; //Cálculo de las señales de control mediante control PI
float u2 = kp2*e2 + ki2*eintegral2;
```

*Figura 7.10: Cálculo de velocidades, filtrado y cómputo de las señales de control en velocidad.*

Como podemos ver, en esta parte se obtienen las velocidades ya en RPM de la manera descrita ya varias veces. A continuación, se le realiza el filtrado LP también explicado en el

apartado anterior. Para el control en velocidad, se comienza asignando las velocidades de referencia recibidas por puerto serie, se asignan las constantes K e I para cada controlador tipo PI y se calcula el error para cada controlador como la diferencia de la velocidad filtrada medida de cada rueda (feedback de los sensores) respecto de la referencia asignada.

Tras esto, se calculan las señales de control a aplicar a los motores para lograr la velocidad deseada en RPM.

```
//Según signo de la señal de actuación se define el sentido de giro de la rueda
int dirizq = 1;
if (ul<0) dirizq = -1;

// Se calcula el pwm a aplicar a la rueda como el valor absoluto de ul
int pwmizq = (int) fabs(ul); //fabs(): floating-point absolute value function, para pasar a valor absoluto valores tipo float

//Se satura el pwm a su valor máximo
if(pwmizq > 255) pwmizq = 255;

//Se hace girar la rueda en un sentido u otro según signo de la señal de control
if(dirizq == 1) AdelanteIzq(pwmizq);
if(dirizq != 1) AtrasIzq(pwmizq);

//Según signo de la señal de actuación se define el sentido de giro de la rueda
int dirder = 1;
if (u2<0) dirder = -1;

// Se calcula el pwm a aplicar a la rueda como el valor absoluto de u2
int pwmder = (int) fabs(u2);

//Se satura el pwm a su valor máximo
if(pwmder > 255) pwmder = 255;

//Se hace girar la rueda en un sentido u otro según signo de la señal de control
if(dirder == 1) AdelanteDer(pwmder);
if(dirder != 1) AtrasDer(pwmder);

//AntiWindup, se satura el error integral en caso de superar los límites máximos de los actuadores (pwm máximo)
if (pwmizq < 255.0) eintegral1 = eintegral1 + e1*deltaTIzq;
if (pwmder < 255.0) eintegral2 = eintegral2 + e2*deltaTDer;

//Se satura directamente el error integral sobre cierto valor con el fin de evitar el efecto windup
if(eintegral1 >= 10.0) eintegral1 = 10.0;
if(eintegral2 >= 10.0) eintegral2 = 10.0;
```

*Figura 7.11: Asignación de PWM a aplicar a cada rueda y sentido de giro, así como saturaciones sobre el error integral (AntiWindup) y sobre los valores máximos de PWM.*

De nuevo el código está bastante bien detallado, pero podemos destacar la decisión de saturar ambos errores integrales a cierto valor, lo cual se consideró y se comprobó necesario para lograr una respuesta adecuada en cuanto a tiempo de subida y comportamiento en general del control, ya que, sin este mecanismo, se llegaba a seguir la referencia, pero de forma bastante más lenta, debido a excesiva acumulación del error, lo que se conoce como efecto Wind Up.

Por último, volvemos a imprimir valores por puerto serie de Putty para poder graficar la telemetría como de costumbre:

```
//Delay de 5ms para poder tener intervalos de tiempo para las representaciones de telemetría
delay(5);
int tiempo = millis();
int deltaTiempo = tiempo-tiempo_ant;

Serial1.print(deltaTiempo);
Serial1.print(" ");
Serial1.print(VRefIzq);
Serial1.print(" ");
Serial1.print(VIzqFilt);
Serial1.print(" ");
Serial1.print(VRefDer);
Serial1.print(" ");
Serial1.println(VDerFilt);
}
```

*Figura 7.12: Impresión de valores por puerto serie para graficar la telemetría.*

Por último, en cuanto al código, omitiremos las capturas de las funciones de tipos de movimiento (*AdelanteIzq()*, *AtrasDer()*,...) y mostraremos las funciones de interrupción que se asignaron para cada encoder en la parte de *Setup()*:

```
/*********************************************
//Función de interrupción del encoder izquierdo
void EncoderIzquierda(){
    ContTicsIzq = ContTicsIzq + 1;      //Se incrementa el contador de tics cada vez que se detecta un pulso del encoder

    long currTIzq = micros();
    float deltaTIzq = ((float) (currTIzq - prevTIzq))/1.0e6;
    frecuencia_izq = 1/deltaTIzq;        //Se calcula la frecuencia con la que entramos a la interrupción (frecuencia con la que se detectan los tics)
    prevTIzq = currTIzq;
}

//Función de interrupción del encoder derecho
void EncoderDerecha(){
    ContTicsDer = ContTicsDer + 1;      //Se incrementa el contador de tics cada vez que se detecta un pulso del encoder

    long currTDer = micros();
    float deltaTDer = ((float) (currTDer - prevTDer))/1.0e6;
    frecuencia_der = 1/deltaTDer;       //Se calcula la frecuencia con la que entramos a la interrupción (frecuencia con la que se detectan los tics)
    prevTDer = currTDer;
}

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

*Figura 7.13: Funciones de interrupción para cada encoder.*

Ambas interrupciones son simétricas y como vemos, lo que se realiza en ellas es bastante simple, lo cual fue modificado ya que previamente se plantearon una serie de cálculos de velocidades y demás dentro de estas interrupciones, sin embargo, siguiendo los consejos del profesor, y cómo es lógico debido a la alta frecuencia con la que saltan estas interrupciones, se deben realizar el menor número de operaciones dentro de las interrupciones para evitar fallos en el conteo de tics u otros problemas.

Dicho esto, principalmente, lo que hacemos es incrementar el contador de tics del encoder, que no se utiliza para estos modos, pero será útil para la estimación de posición más adelante.

Además de esto, se calcula la frecuencia con la que entramos a la interrupción como la inversa del incremento de tiempo entre una entrada a la interrupción y la entrada posterior a dicha

interrupción. No se necesita nada más de estas interrupciones, ya que el procesado y cálculo de velocidades se realiza como ya vimos en el *loop()*.

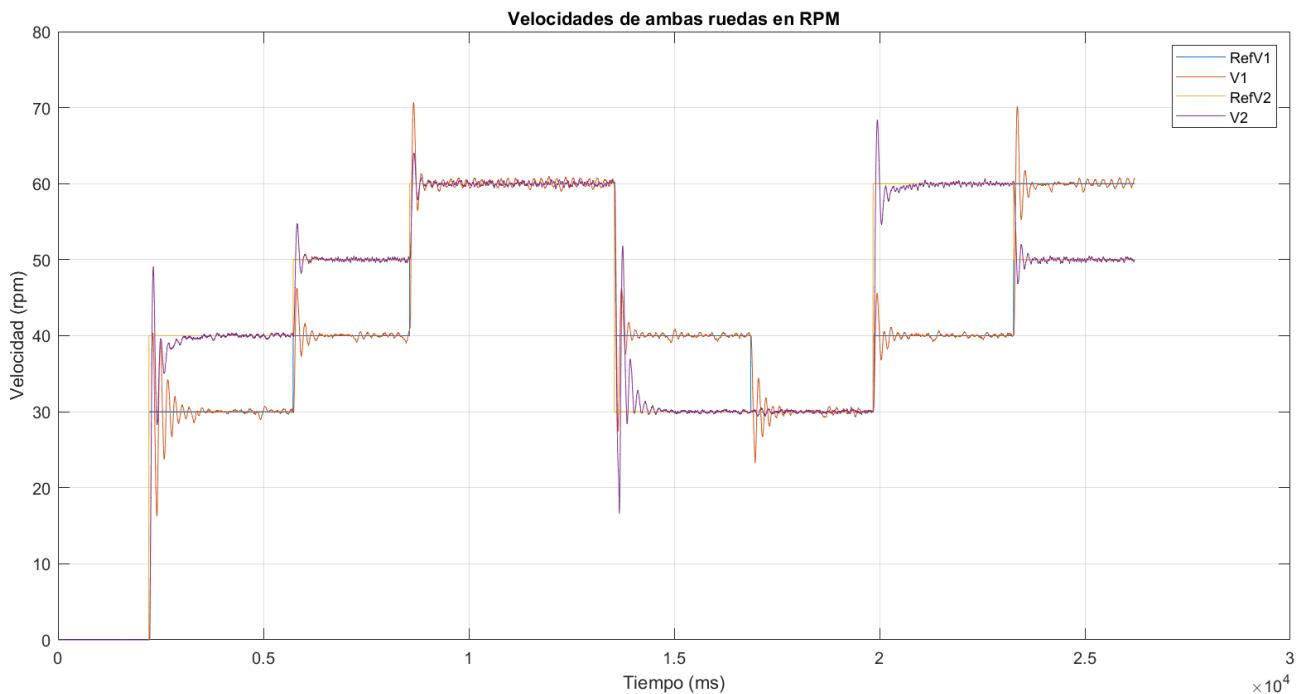
Visto todo esto, creo que ha quedado claro la forma en que se han obtenido las velocidades angulares en RPM a partir de la información que proporcionan los encoders de los que disponemos.

Podemos pasar ahora a ver los resultados que ofrece el control en velocidad planteado e implementado anteriormente.

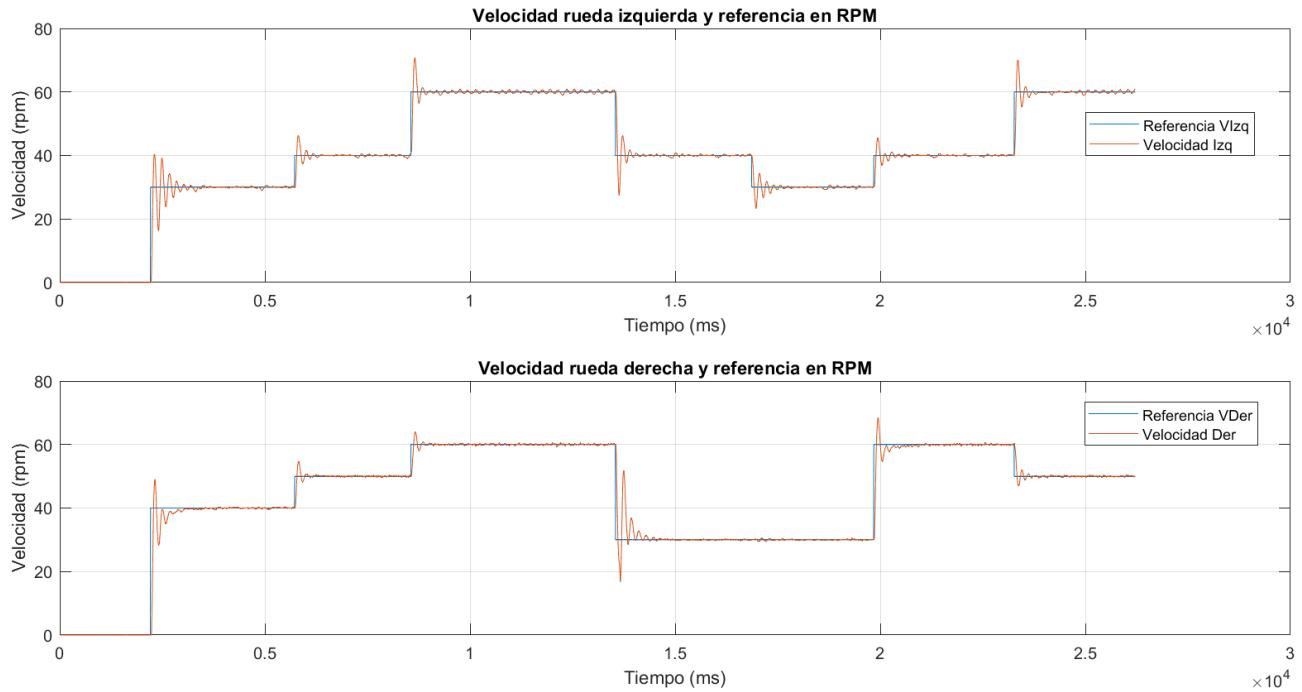
### 7.3- Gráficas y resultados

Como se requiere en este modo, vamos a someter al control en velocidad a una serie de escalones en las referencias de velocidad en RPM, diferenciando además la referencia entregada a cada rueda, para ver cómo se comporta y sacar ciertas conclusiones.

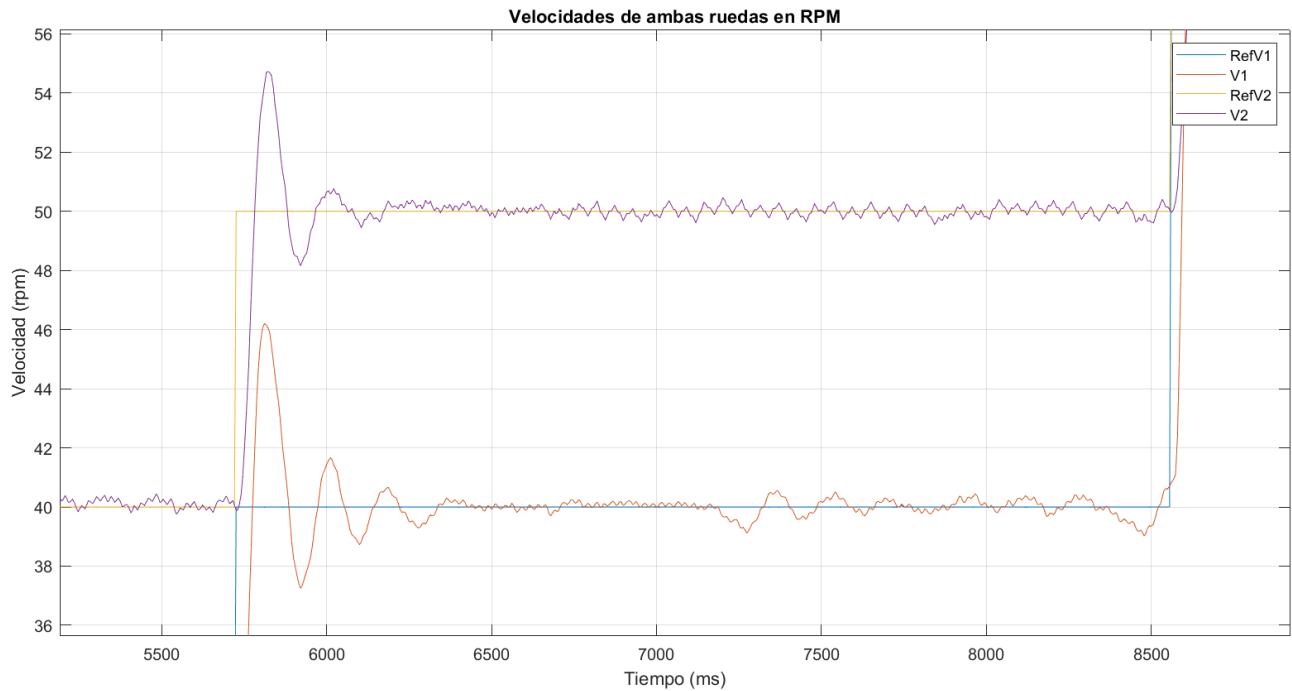
Como sólo se solicitan gráficas, mostraremos varias de ellas para tener información más detallada.



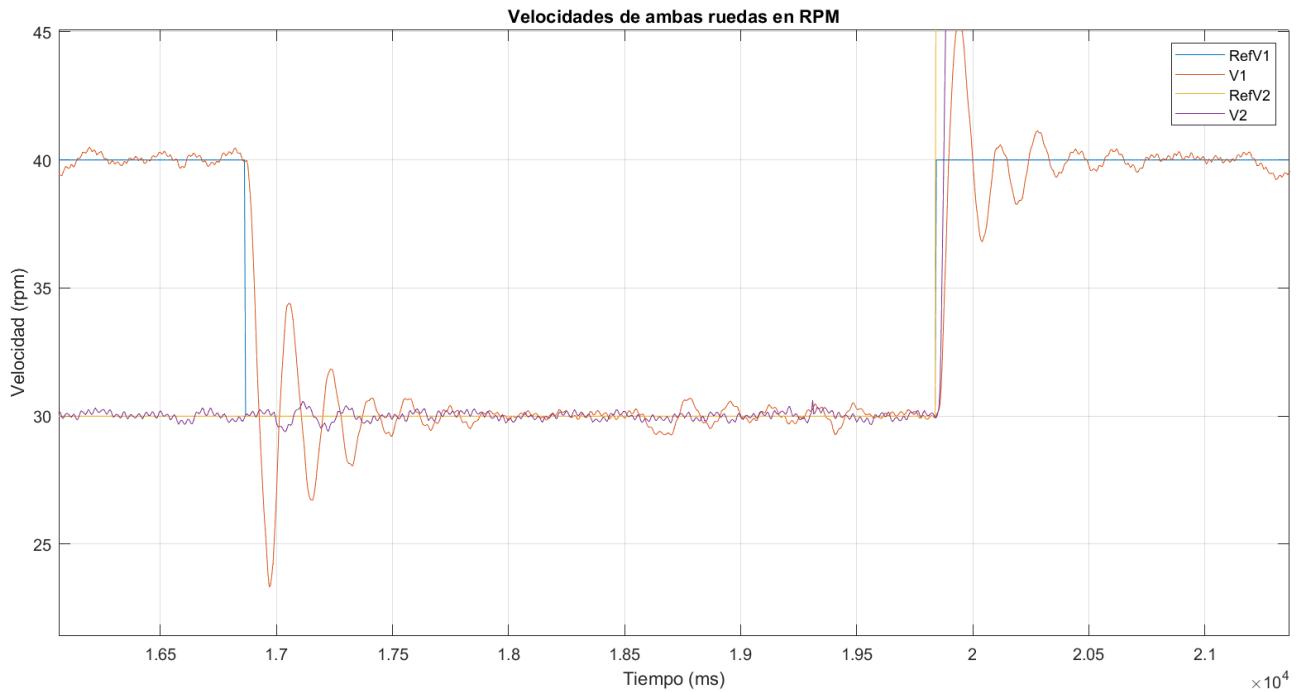
**Figura 7.14:** Gráfica conjunta que muestra el seguimiento por parte del control de cada rueda a su respectiva referencia.



*Figura 7.15:* Gráficas desacopladas que muestran el seguimiento por parte del control de cada rueda a su respectiva referencia.



*Figura 7.16:* Detalle de velocidades que muestran el seguimiento por parte del control en velocidad de cada rueda a su respectiva referencia, caso para referencias distintas.



**Figura 7.17:** Detalle de velocidades que muestran el seguimiento por parte del control en velocidad de cada rueda a su respectiva referencia, caso para referencias iguales.

Como vemos en este conjunto de gráficas, el comportamiento del control es el correspondiente a un sistema de 2º orden subamortiguado con su correspondiente sobreoscilación. Aunque esto podría generar problemas, se considera que es preferible un control rápido con el sobreprecio de estas sobreoscilaciones, las cuales, por sus dimensiones, se pueden considerar aceptables y como veremos, no generan funcionamientos incorrectos.

Habiendo comprobado esto, se considera válido el control en velocidad diseñado y se utilizará para el test del siguiente Modo 6, así como para posteriores modos, como por ejemplo en la odometría.

## 8.- MODO 6 – MEDIANTE EL USO DE SENSORES DE VELOCIDAD/ CONTROL EN VELOCIDAD, REALIZAR TEST QUE MUESTRE LA RESPUESTA MOVIÉNDOSE EN LÍNEA RECTA

### 8.1- Comentarios generales

A diferencia del resto de modos, los cuales han sido explicados con relativamente extenso detalle, este modo 6 se corresponde con una extensión natural del modo 5 explicado anteriormente, y es que tras haber puesto a prueba el control en velocidad propuesto anteriormente y ver que su funcionamiento es suficientemente válido, se propone realizar un test donde el robot avance en línea recta a través del control de velocidad para cada rueda.

Para lograr esto, lo único que necesitamos es aplicar la misma referencia de velocidad a cada rueda, siendo que, si el control funciona bien, aún estando presente el desfase en cuanto a PWM necesario para que ambas ruedas avancen a la misma velocidad, será el control en velocidad el que a través del *feedback* de los encoders, ajuste dichos PWM por separado para que incluso a pesar de este desfase, ambas consigan tener la misma velocidad en RPM.

Básicamente comentar que, considerando como ya se ha dicho que este modo es simplemente un test sobre el control velocidad sobre todo lo explicado en el apartado anterior, no es necesario repetirlo de nuevo, sino que haremos uso tanto de dicho código implementado con esos conceptos ya explicados, para llevar a cabo este test.

### 8.2- Gráficas y resultados

Como se ha mencionado anteriormente, se va a realizar un test donde se pretende que el robot avance en línea recta haciendo uso del control en velocidad para cada rueda. Para ello, simplemente aplicamos como referencia una velocidad de 30 RPM para ambas ruedas y veremos si efectivamente se desplaza en línea recta.

En primer lugar, se adjunta un enlace al video donde se realiza este test:

[https://drive.google.com/file/d/1oFfQl1uDUZgjuNgtvv-Co\\_jyi2C2NAW/view?usp=sharing](https://drive.google.com/file/d/1oFfQl1uDUZgjuNgtvv-Co_jyi2C2NAW/view?usp=sharing)

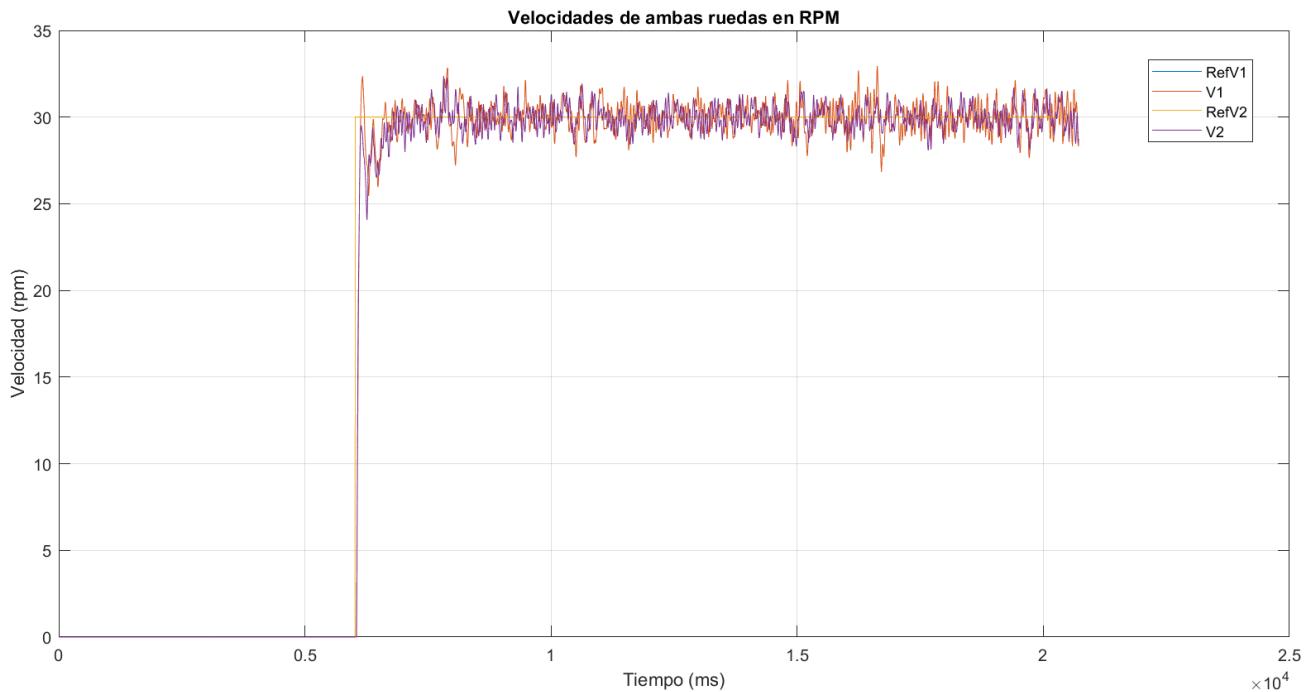
Tras observar el vídeo podemos concluir que como esperábamos, el control en velocidad es suficientemente bueno para lograr un avance en línea recta sin una desviación notable.

Aunque no tenemos forma de cuantificar si se ha desviado a simple vista en este modo, se ha realizado en el siguiente modo, en la *figura 9.13* una comparativa del robot avanzando 1 metro en línea recta, prácticamente el mismo test que el que comentamos ahora, de dos formas:

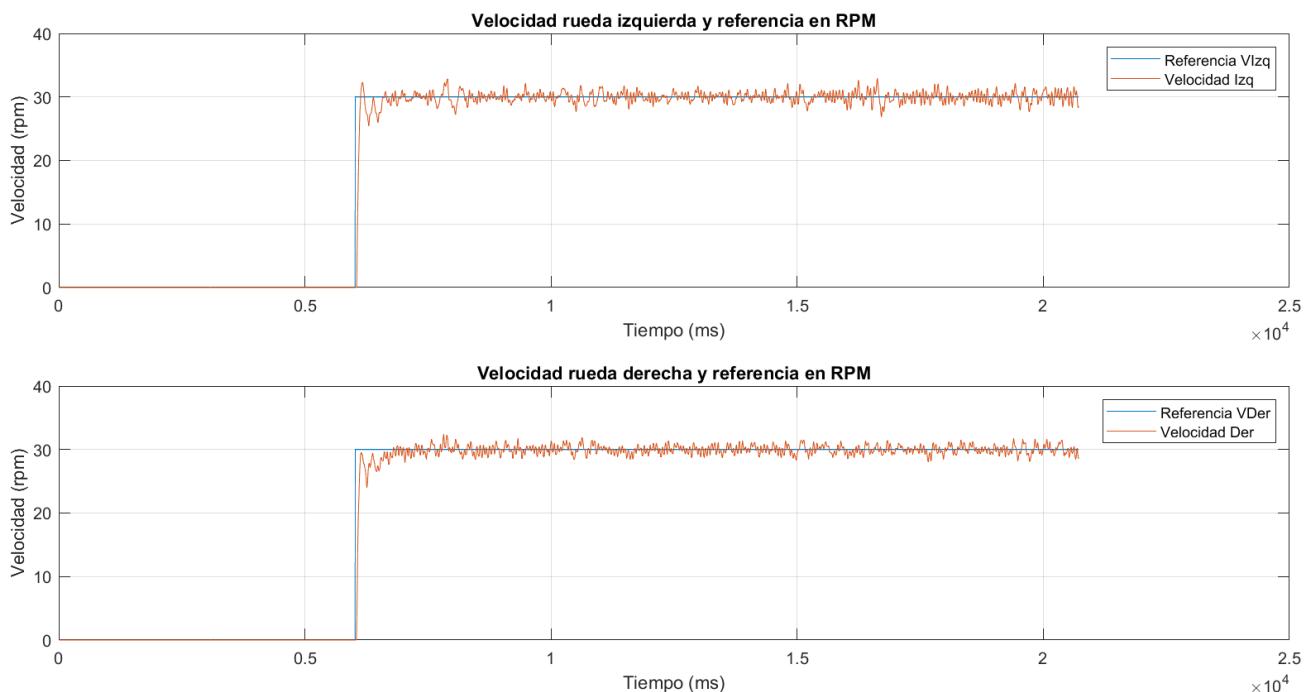
- Sólo con el control de velocidad (como ahora) con referencia de 30 RPMs para ambas ruedas.
- Añadiendo un control de posición a través de odometría.

Como los aspectos del siguiente apartado lógicamente se explican en su apartado correspondiente, sólo destacar que en esa *figura 9.13* logramos cuantificar la desviación del robot a lo largo de una línea recta de 1 metro en algo del orden de 3-4 cm hacia la izquierda. Lo cual se puede considerar aceptable o no según la aplicación, pero desde luego no es un valor desorbitado como para llegar a decir que el desplazamiento no ha sido en línea recta.

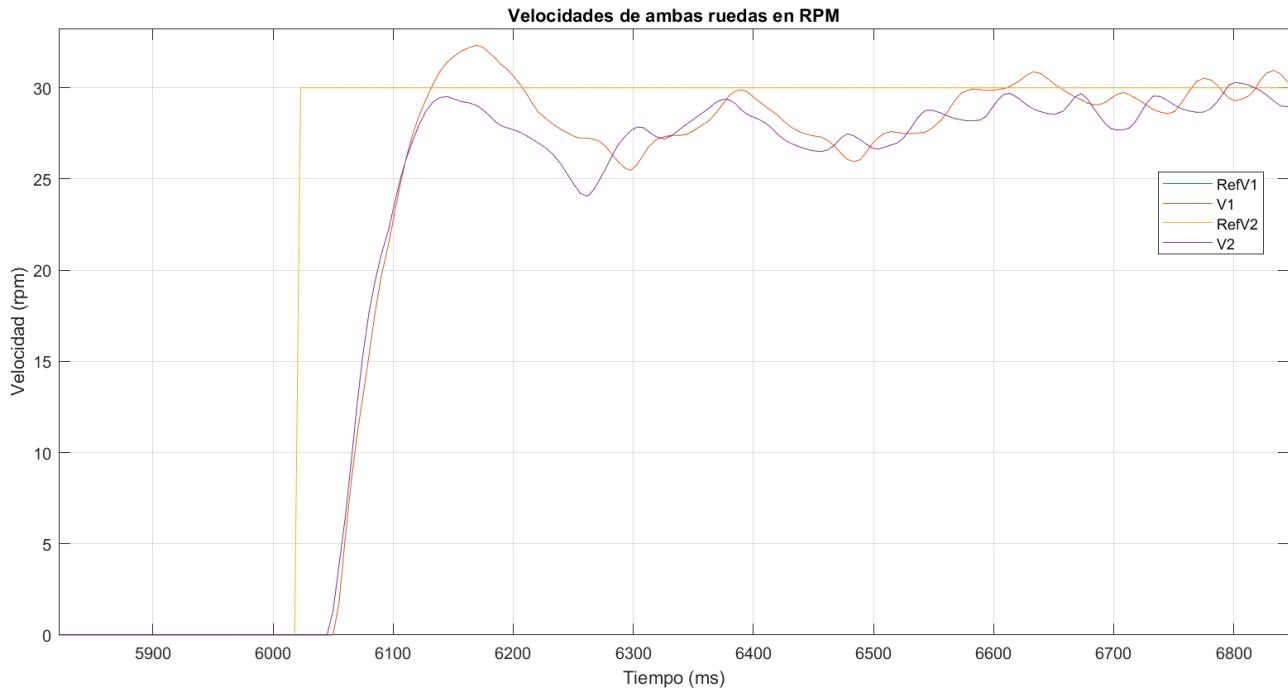
Tras este inciso, podemos proseguir mostrando las gráficas obtenidas del experimento mostrado en el video anterior:



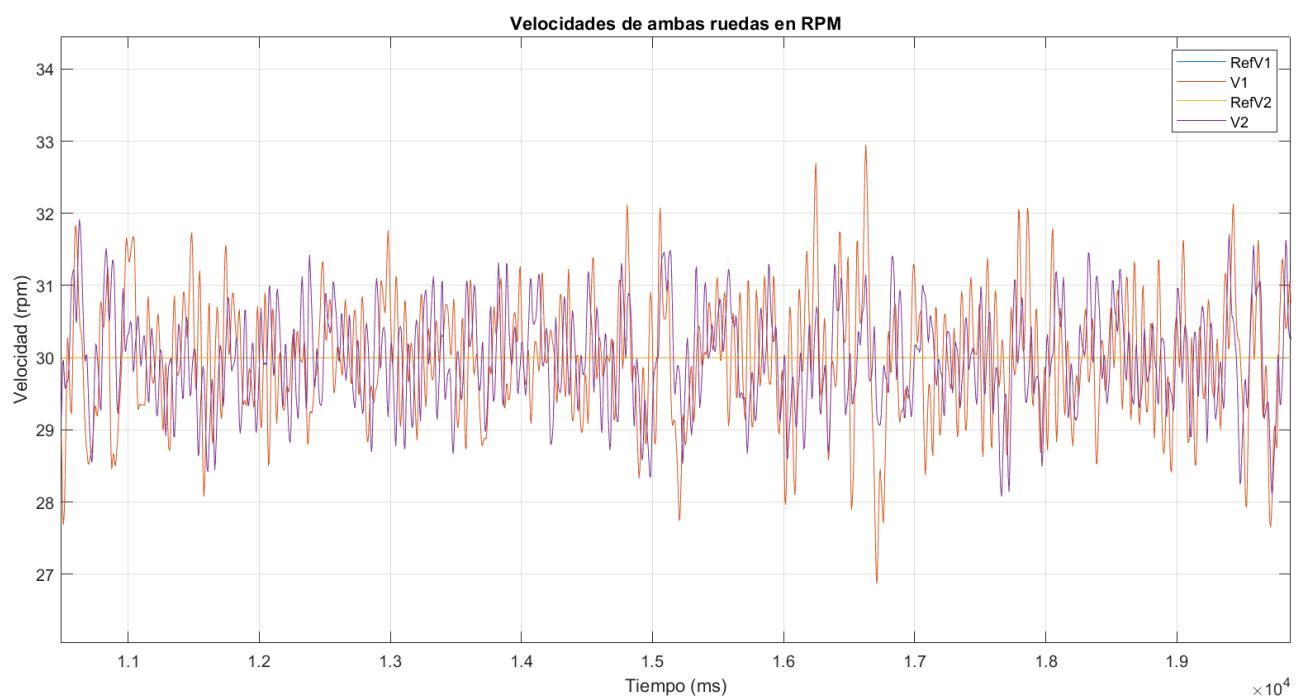
**Figura 8.1:** Gráfica conjunta de velocidades en RPM para ambas ruedas durante la realización del test en línea recta.



**Figura 8.2:** Gráfica desacoplada de velocidades en RPM para ambas ruedas durante la realización del test en línea recta.



**Figura 8.3:** Gráfica de velocidades en RPM para ambas ruedas durante la realización del test en línea recta. Detalle centrado en el escalón (régimen transitorio) de la referencia a 30 RPM.



**Figura 8.4:** Gráfica de velocidades en RPM para ambas ruedas durante la realización del test en línea recta. Detalle centrado en el régimen permanente.

Como vemos en esta serie de gráficas, ambas velocidades siguen correctamente a la referencia de 30 RPM.

Sin embargo, se puede deducir de las diferentes magnitudes en las oscilaciones en torno a referencia de cada una de las velocidades que, aunque no difieren excesivamente, sí que dicha diferencia entre ambas puede conllevar a la ligera desviación comentada previamente al análisis de estas gráficas de en torno a 3-4 cm hacia la izquierda.

Sin embargo, considero personalmente, que, para las condiciones del sistema y el control desarrollado, el resultado es aceptable y suficiente y que si quisieramos mejorarlos, se pueden combinar control en velocidad y posición (tras desarrollar e implementar la odometría) para mejorar este test, como hemos dicho antes y como además se demostrará su eficacia en el siguiente apartado.

## 9.- MODO 7 – ESTIMACIÓN DE POSICIÓN

### 9.1.- Ecuaciones - Modelo

Una vez logrado tanto la medición correcta de las velocidades angulares de las ruedas a través de los encoders como el propio control en velocidad, se puede avanzar y hacer uso de esa información y posibilidades que ofrece el control en velocidad para implementar una estimación de posición a través de la información de velocidad angular de ambas ruedas y las dimensiones físicas del robot móvil, así como su modelo cinemático, esto se conoce como odometría.

De esta manera, planteando una serie de ecuaciones, podemos llegar al cálculo de las expresiones que nos calculan el desplazamiento en coordenadas ( $x, y$ ) del robot móvil, así como su ángulo de orientación (yaw)( $\theta$ ), obteniéndose los desplazamientos en cm y los ángulos en radianes. Si planteamos estas ecuaciones:

- En un tiempo  $t$ :
  - El encoder izquierdo cuenta  $N_i$  tics.
  - El encoder derecho cuenta  $N_d$  tics.
- Se definen unos factores de conversión  $f_i$  y  $f_d$  entre tics y desplazamiento:

$$f_i = \frac{\pi \cdot D}{R_i} \quad f_d = \frac{\pi \cdot D}{R_d}$$

Siendo:

- $D$  -> Diámetro de la rueda en [cm], idéntico para ambas ruedas (6.9 cm).
- $R_i$  -> Resolución del encoder izquierdo en [Nº de tics/vuelta]. (182.5 tics/vuelta).
- $R_d$  -> Resolución del encoder derecho en [Nº de tics/vuelta]. (198.0 tics/vuelta).

- Se puede obtener ahora la distancia recorrida  $L$  por cada rueda:

$$L_i = f_i \cdot N_i \quad L_d = f_d \cdot N_d$$

- Tras obtener la distancia recorrida por cada rueda, podemos calcular la distancia recorrida por el centro del robot:

$$L_c = \frac{L_i + L_d}{2}$$

- A continuación, se puede obtener el giro  $\Delta\theta$  del robot sobre su eje (veremos después un dibujo aclaratorio):

$$\Delta\theta = \frac{L_d - L_i}{B}$$

Siendo:

- $B$  -> Distancia base del robot (distancia entre ambas ruedas) en [cm], (8.0 cm).
- $L_d$  -> Distancia recorrida por la rueda derecha en [cm].
- $L_i$  -> Distancia recorrida por la rueda izquierda en [cm].

- Finalmente, si tenemos  $x(0)$ ,  $y(0)$  y  $\theta(0)$ , que son las coordenadas iniciales del robot, podemos obtener las nuevas coordenadas del robot tras cierto movimiento tal que:

$$\begin{aligned}\theta(t) &= \theta(0) + \Delta\theta \\ x(t) &= x(0) + L_c \cdot \sin(\theta(t)) \\ y(t) &= y(0) + L_c \cdot \cos(\theta(t))\end{aligned}$$

**Nota:** el ángulo  $\theta$  deberá ser transformado a radianes para realizar estos cálculos y que resulten correctos.

Con estas ecuaciones, simplemente implementadas en Arduino, podemos obtener fácilmente una estimación de la posición del robot a lo largo de sus movimientos.

Es interesante ver un dibujo ilustrativo del que parten las ecuaciones planteadas anteriormente:

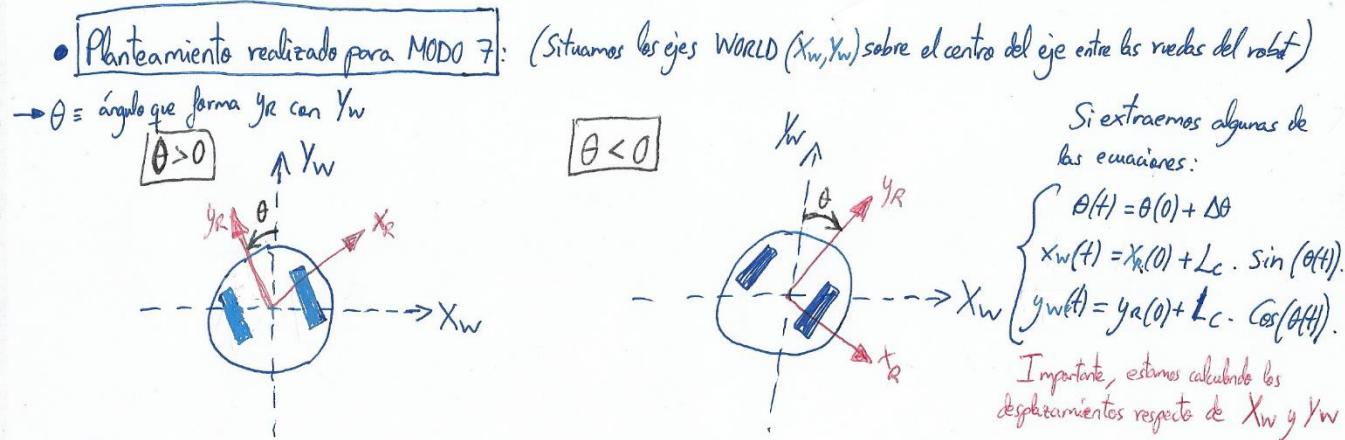
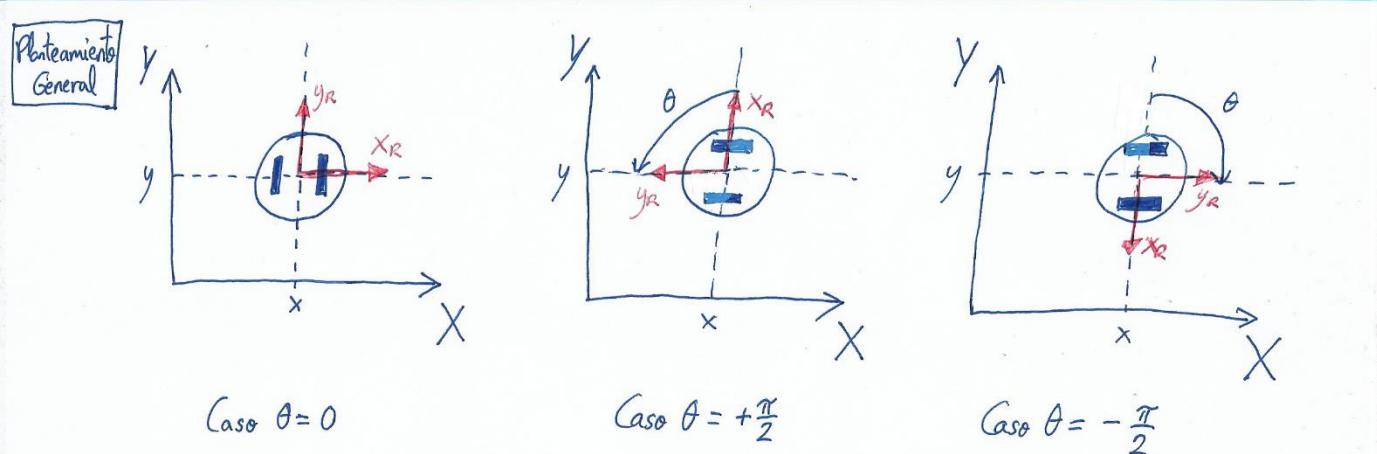


Figura 9.1: Planteamiento de las ecuaciones que resuelven la odometría a partir de la información de los encoders y las ecuaciones explicadas anteriormente.

## 9.2.- Implementación de la odometría en Arduino

Podemos ver cómo la implementación de esta odometría que parece relativamente densa en cuanto a cálculos se puede resolver de manera bastante simple mediante una simple función que implemente este conjunto de ecuaciones comentadas:

```
void Odometria(int TicsIzq, int TicsDer){
    resolucion_izq = 182.5; //Tics/vuelta rueda izquierda
    resolucion_der = 198.0; //Tics/vuelta rueda izquierda
    diametro_rueda = 6.9; //Diámetro de la rueda en [cm]
    longitud_base = 8.0; //Longitud entre ambas ruedas en [cm]
    fi = pi * diametro_rueda/resolucion_izq; //Factor de conversion entre tics y desplazamiento
    fd = pi * diametro_rueda/resolucion_der; //Factor de conversion entre tics y desplazamiento
    Li = fi * TicsIzq; //Longitud recorrida por la rueda derecha
    Ld = fd * TicsDer; //Longitud recorrida por la rueda izquierda
    Lc = (Li + Ld)/2; //Longitud recorrida por el centro del robot
    phi = (Ld - Li)/longitud_base; //Ángulo de orientación del robot
    phi_rad = phi*pi/180;
    y = Lc * cos(phi_rad); //Coordenada x del robot
    x = Lc * sin(phi_rad); //Coordenada y del robot

    //Muestreo por puerto serie de la odometria:
    //Serial1.print("x = ");
    Serial1.print(" ");
    Serial1.print(x);
    Serial1.print(" ");
    //Serial1.print("y = ");
    Serial1.print(y);
    Serial1.print(" ");
    Serial1.print(phi);
    Serial1.print(" ");
}
```

*Figura 9.2: Implementación en Arduino de la odometría planteada anteriormente.*

Vemos como con esta simple función, cuyas variables son globales, para que puedan interaccionar con el resto del programa, básicamente implementamos las ecuaciones de que resuelven la estimación de posición.

Esta función recibe el conteo de tics de ambos encoders, esto es necesario básicamente para resolver el problema de “pivotado” en las esquinas del cuadrado a la hora de que el robot realice el test propuesto en el modo 7, un desplazamiento realizando un cuadrado de 1 metro de longitud. De esta manera, cuando se pare en las esquinas, cambiaremos el signo del conteo de tics de la rueda derecha, para que se obtenga correctamente el ángulo  $\theta$  al pivotar y por tanto el robot pueda conocer cuando ha terminado de pivotar los 90º en esa esquina para disponerse a realizar los siguientes pasos.

El resto de la función básicamente consiste en ir implementando paso por paso las ecuaciones propuestas teóricamente en el apartado anterior.

Para que el robot calcule la odometría, simplemente se llama a la función **Odometria()** dentro del loop, de manera que iremos realizando los cálculos en todo momento.

```
//Calculamos la odometria en cada iteración:
Odometria();
while(y >= 100.0) Parar();
```

*Figura 9.3: Llamada a la función de Odometria() en el loop junto a orden de parada al llegar a 100 cm (no es necesario corrección de inercia debido a que las velocidades son bajas).*

De esta manera, para realizar un test comprobando la corrección de la odometría se han colocado dos marcas a 1 metro de longitud y en el código se impone que el robot se pare una vez la Y valga 100, es decir el robot se haya desplazado 100 cm hacia adelante. Además, podremos observar el desplazamiento en x a lo largo de ese 1 metro que se desplaza para ver cuánto se desvía respecto de una trayectoria totalmente recta.

Enlace a video del test:

[https://drive.google.com/file/d/1AVehfFH547qsy8VxPjM4XK\\_peUoTMK2s/view?usp=sharing](https://drive.google.com/file/d/1AVehfFH547qsy8VxPjM4XK_peUoTMK2s/view?usp=sharing)



*Figura 9.4: Situación inicial del test de desplazamiento de 1m.*



Figura 9.5: Zoom sobre la zona final de la trayectoria para ver los 100 cm.

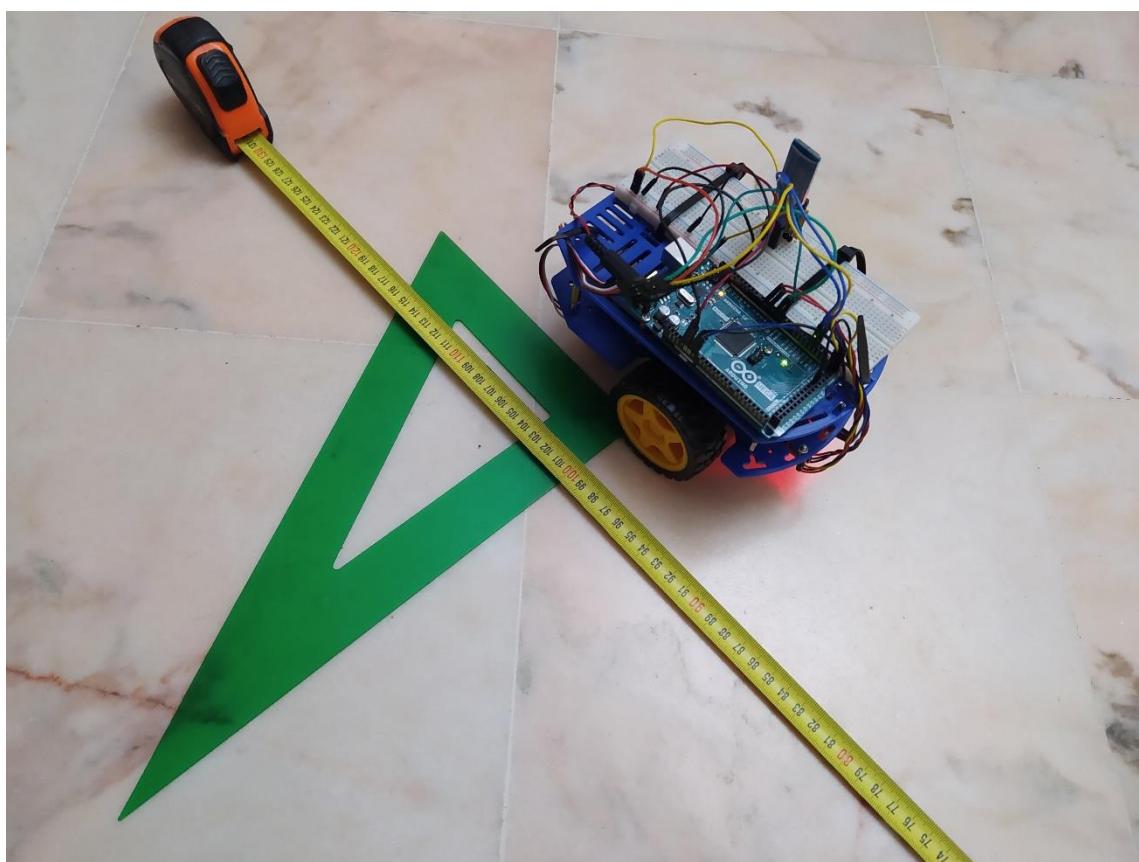
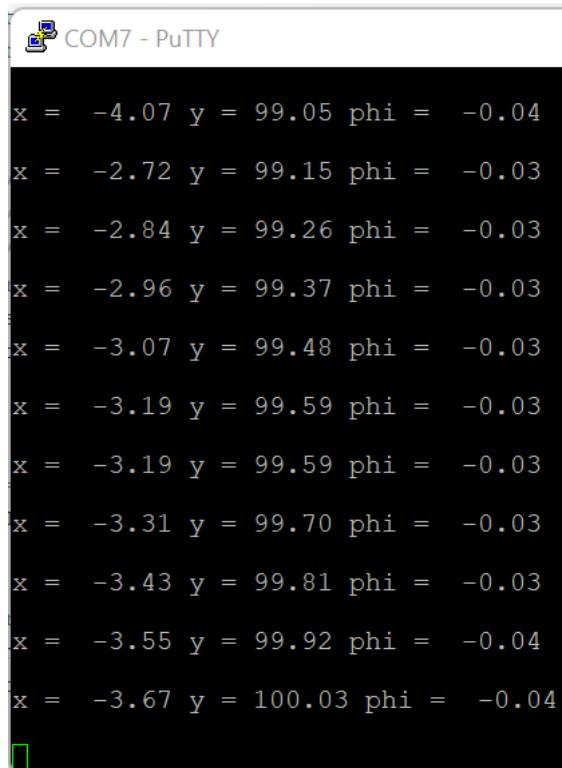


Figura 9.6: Lugar de parada del robot al obtener la odometría  $y = 100$  cm



```

COM7 - PuTTY

x = -4.07 y = 99.05 phi = -0.04
x = -2.72 y = 99.15 phi = -0.03
x = -2.84 y = 99.26 phi = -0.03
x = -2.96 y = 99.37 phi = -0.03
x = -3.07 y = 99.48 phi = -0.03
x = -3.19 y = 99.59 phi = -0.03
x = -3.19 y = 99.59 phi = -0.03
x = -3.31 y = 99.70 phi = -0.03
x = -3.43 y = 99.81 phi = -0.03
x = -3.55 y = 99.92 phi = -0.04
x = -3.67 y = 100.03 phi = -0.04

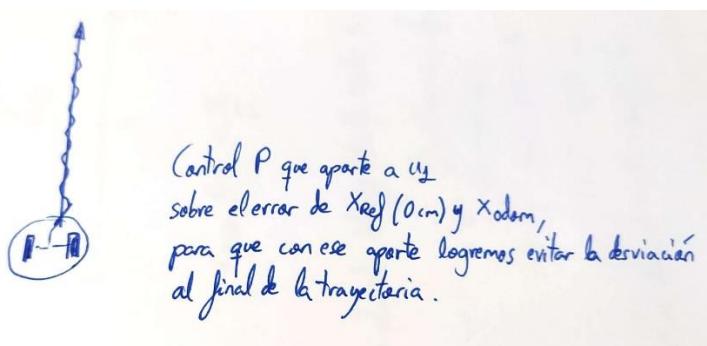
```

*Figura 9.7: Registros de la posición del robot, de manera que x e y están referidos a Xw e Yw.*

Como vemos en esta figura, el robot se ha parado cuando la y ha cumplido ser mayor o igual que 100 cm y como también veíamos en las imágenes anteriores, la estimación de posición tiene una precisión relativamente buena. Además, observamos que, al terminar la trayectoria, se ha desviado algo más de 3.5 cm en el eje Xw, es decir, se ha desviado un poco hacia la izquierda respecto a su posición inicial. No es una desviación excesivamente grande, pero es suficiente para tenerla en cuenta a la hora de valorar su precisión.

### 9.3.- Control de posición basado en odometría

Para mejorar esa desviación en el eje Xw, podemos tomar ventaja de la información que nos ofrece la odometría e incorporar un control que haga seguimiento de la referencia  $Xw = 0$ , y realice, por ejemplo, un pequeño aporte en la señal de control de la rueda izquierda que vaya corrigiendo esa desviación a lo largo de la trayectoria recta. La idea es simple, pero se puede ver ilustrada en la siguiente imagen:



*Figura 9.8: Idea simple del control P de posición para evitar la desviación previa.*

## 9.4.- Implementación del control de posición en Arduino

Para la implementación de esta funcionalidad, también se ha diseñado una función la cual es la siguiente:

```
float ControlPosicion(float Ref_x, float Ref_y, int ModoControl){
    float u1_aux = 0.0;
    if(ModoControl == 1){      //ModoControl = 1, control de la coordenada x seguimiento a valor 0
        float kp1 = 20;
        float e1 = Ref_x - x;
        float u1 = kp1*e1;
        u1_aux = u1;
    }
    if(ModoControl == 2){      //ModoControl = 2, control de la coordenada y seguimiento a valor 100
        float kp1 = 20;
        float e1 = Ref_y - y;
        float u1 = kp1*e1;
        u1_aux = u1;
    }
    float u1_posicion = u1_aux;
    return u1_posicion;
}
```

*Figura 9.9: Función que implementa el cómputo del aporte de la señal de control a aplicar a la rueda izquierda (elección arbitraria) para evitar la desviación.*

Como vemos, esta función permite controlar un seguimiento de referencia en posición tanto para la x como para la y, aunque para la trayectoria del cuadrado, como veremos, sólo lo aplicamos para la x. Simplemente realizamos un control tipo P, con una kp 20, que, aunque pueda parecer elevada, los errores van a ser muy pequeños del orden de 0.0... con lo cual tendremos una aportación pequeña sobre la u1 de la rueda izquierda, pero suficientemente dimensionada para que evite la desviación en el eje x vista en el test anterior de la figura 9.7.

Para aplicarla a nuestro sistema global, la llamamos y aplicamos su aporte sobre u1, como ya hemos dicho, actuando únicamente sobre la rueda izquierda:

```
//Se computa el aporte al control de velocidad por parte del control de posicion para mantener el robot recto con el feedback de la odometría
//este aporte es bastante pequeño y se aplica simplemente a una de las ruedas para garantizar que el robot permanece recto en su movimiento:
float aporte_u1_pos = ControlPosicion(0.0, 100.0, 1);
//Serial1.print(aporte_u1_pos);

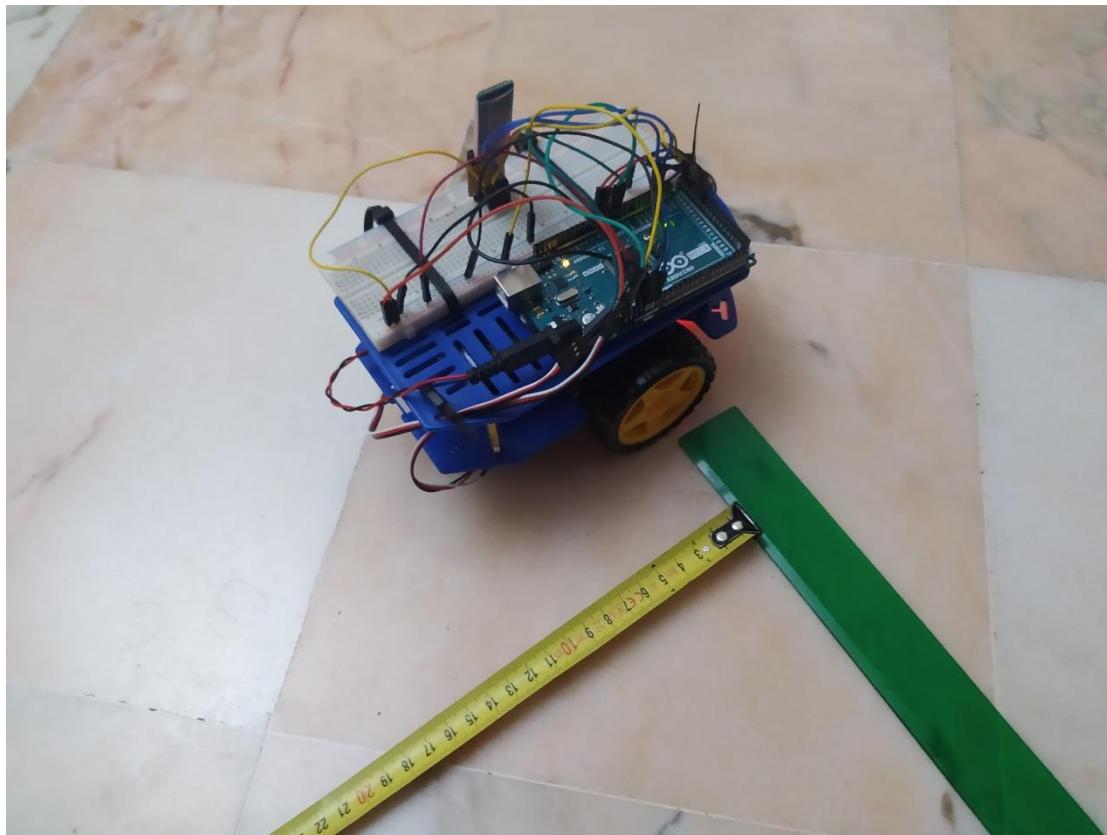
float u1 = aporte_u1_pos + kp1*e1 + ki1*eintegral1; //Cálculo de las señales de control mediante control PI
float u2 = kp2*e2 + ki2*eintegral2;
```

*Figura 9.10: Obtención del aporte a aplicar sobre u1 por parte del control de posición.*

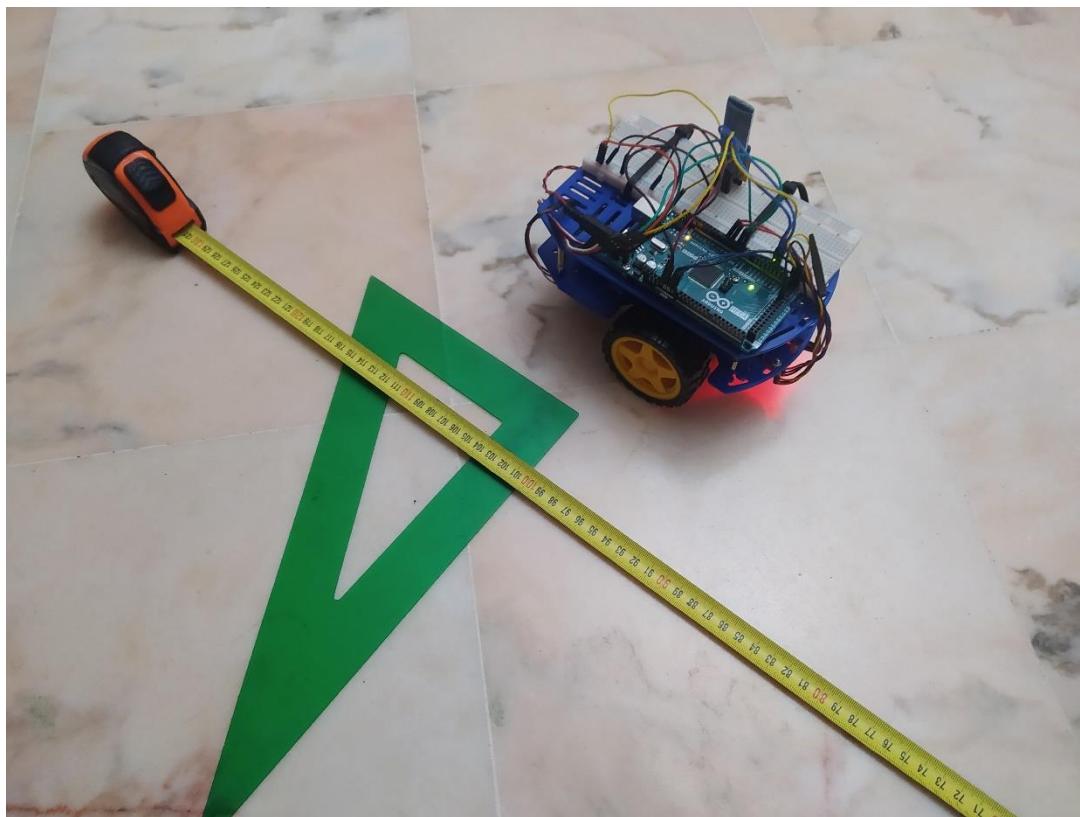
Es interesante ahora repetir el mismo test del apartado anterior, incluyendo ahora este control de posición, para ver si mejora esa desviación.

Al igual que antes se deja un enlace al vídeo del test realizado:

<https://drive.google.com/file/d/1AkPXrlFzZJ2ZyCd1cZlFJMCT8U1mwxb/view?usp=sharing>



**Figura 9.11:** Posición inicial del robot en el test de desplazamiento de 1 metro en línea recta con control de posición para evitar desviación.



**Figura 9.12:** Posición final del robot en el test de desplazamiento de 1 metro en línea recta con control de posición para evitar desviación.

```
x = -4.07 y = 99.05 phi = -0.04
x = -2.72 y = 99.15 phi = -0.03
x = -2.84 y = 99.26 phi = -0.03
x = -2.96 y = 99.37 phi = -0.03
x = -3.07 y = 99.48 phi = -0.03
x = -3.19 y = 99.59 phi = -0.03
x = -3.19 y = 99.59 phi = -0.03
x = -3.31 y = 99.70 phi = -0.03
x = -3.43 y = 99.81 phi = -0.03
x = -3.55 y = 99.92 phi = -0.04
x = -3.67 y = 100.03 phi = -0.04
[...]
x = -0.05 y = 98.22 phi = -0.03
x = -0.06 y = 98.34 phi = -0.03
x = -0.03 y = 98.39 phi = -0.02
x = -0.04 y = 98.50 phi = -0.02
x = -0.04 y = 98.62 phi = -0.02
x = -0.04 y = 98.73 phi = -0.02
x = -0.04 y = 98.85 phi = -0.02
x = -0.02 y = 98.90 phi = -0.01
x = -0.04 y = 98.96 phi = -0.03
x = -0.05 y = 99.07 phi = -0.03
x = -0.05 y = 99.19 phi = -0.03
x = -0.05 y = 99.30 phi = -0.03
x = -0.05 y = 99.42 phi = -0.03
x = -0.03 y = 99.47 phi = -0.02
x = -0.03 y = 99.59 phi = -0.02
x = -0.03 y = 99.70 phi = -0.02
x = -0.06 y = 99.76 phi = -0.03
x = -0.06 y = 99.87 phi = -0.03
x = -0.06 y = 99.99 phi = -0.04
x = -0.04 y = 100.04 phi = -0.02
```

*Figura 9.13: Comparativa de los registros de la posición del robot al incluir control de posición (derecha) y sin aplicarlo (izquierda) (figura 9.7), de manera que x e y están referidos a Xw e Yw.*

Como vemos en esta comparativa, la ligera desviación hacia -Xw que sufría el robot al recorrer 1 metro de espacio, se ha corregido de manera bastante efectiva con el simple control P propuesto sobre la rueda izquierda, viendo que en el caso de tener este control de posición sobre la coordenada x, para que siga a referencia 0, la desviación se mantiene en valores prácticamente despreciables (décimas o centésimas de cm).

Por lo tanto, logramos así una implementación bastante efectiva y simple a la vez de lograr trayectorias que son casi perfectamente rectas (también depende la sensación de rectitud de la trayectoria) de cómo se oriente inicialmente al robot.

Esto será muy útil para reducir los errores a la hora de realizar el test que realice un desplazamiento en forma de cuadrado en el siguiente apartado.

## 9.5.- Test en el cual el robot realiza una trayectoria cuadrada de 1 metro de lado

### 9.5.1.- Conceptos teóricos para la consecución de esta tarea

Para realizar la trayectoria cuadrada de 1 metro de lado, básicamente partimos de los conceptos desarrollados en el resto de apartados anteriores relacionados tanto con la odometría como con el control de posición.

Partiendo de lo que ya tenemos, que consiste en obtener la posición de nuestro robot respecto a Xw y Yw (coordenadas absolutas) y un sistema para evitar desviaciones al trasladarse en línea recta, el planteamiento que se ha seguido para conseguir esa trayectoria ha sido el siguiente:

Realizaremos un proceso cíclico en el que el robot realiza los siguientes pasos secuencialmente y los repite indefinidamente:

- Se realiza un desplazamiento en línea recta de 1 metro lo más preciso posible apoyándose en el control de posición y en el control de velocidad de las ruedas (proporcionando la misma referencia de velocidad a ambas, 30 RPM en este caso).
- Una vez se llega a 1 metro desplazado ( $y \geq 100$ ), el robot pivota sobre si mismo 90º. Para poder realizar esto, es necesario que una de las velocidades sea negativa, en este caso, la de la rueda derecha, que será la que invierte su sentido de giro para permitir el pivotado. Para "contar" velocidad negativa y por tanto que esto se tenga en cuenta en el cálculo del ángulo de orientación, es necesario invertir el signo del contador de tics del encoder derecho. De esta manera, al llegar a  $-\pi/2$  teóricamente, aunque luego se ha tenido que poner un valor más elevado para que fuese más próximo a un giro de 90º, el pivotado habría terminado.
- Se resetean los contadores de tics de ambos encoders, lo que equivale a que el robot vuelve a estar en (0,0) en coordenadas  $X_w$  y  $Y_w$ , con lo que podemos volver al primer punto y repetir el proceso, para que realice los siguientes lados del cuadrado.

Esta explicación se puede ver gráficamente en la siguiente máquina de estados, la cual es similar a lo que se ha implementado en el código:

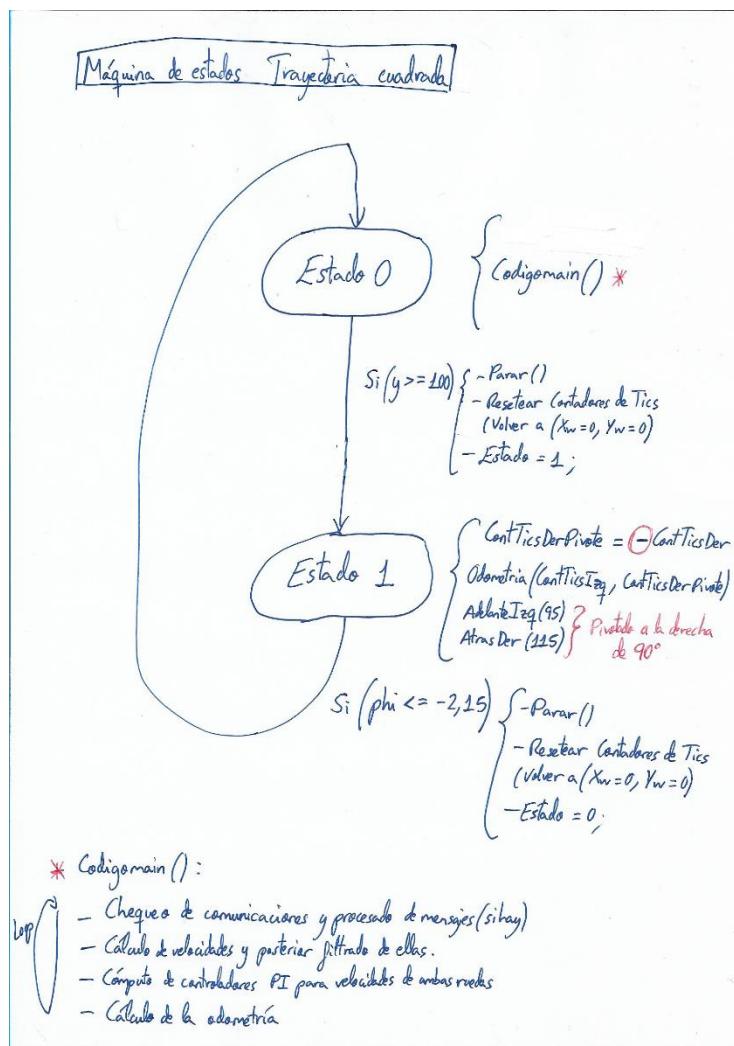


Figura 9.14: Máquina de estados planteada para la resolución de la trayectoria cuadrada.

### 9.5.2.- Código en Arduino para realizar esta tarea

Para implementar este test en Arduino, basta con incluir en el código el esquema de la máquina de estados anteriormente planteado, de esta manera quedaría el código de la siguiente manera (sólo muestro la parte de la FSM, el resto del código no comentado es bastante repetitivo respecto de apartados anteriores, si se quiere ver completo, acudir a los enlaces del Anexo o los códigos entregados en EV):

```

switch(estado) {
    //En el estado 0:
    case 0:
        tiempo_anterior = millis();
        codigo_main();           //Se realiza el código main, que incluye control de velocidad de las 2 ruedas + control de posición
        delay(5);
        tiempo_actual = millis();
        delta_tiempo = tiempo_actual-tiempo_anterior;
        Prints();
        if(y >= 100){           //Una vez se llega a 1m (100cm):
            Parar();             //Se para el robot
            ContTicsIzq = 0;      //Volvemos a estar en "(0,0)", lo cual se realiza reseteando ambos contadores de tics
            ContTicsDer = 0;
            estado = 1;           //Se pasa al estado 1
        }
        break;

    case 1:
        tiempo_anterior = millis();
        ContTicsDerPivote = -ContTicsDer;          //Se invierte el signo del contador de tics del encoder derecho (emula una velocidad negativa, n
        Odometria(ContTicsIzq, ContTicsDerPivote); //Se calcula la odometría teniendo en cuenta la "velocidad negativa de la rueda derecha"
        AdelanteIzq(95);                          //Se hace avanzar la rueda izquierda y retroceder la rueda izquierda a la vez.
        AtrasDer(115);                           //Con estos valores de PWM el pivotado se produce de manera correcta.

        float frecuenciaIzq = 0.0; //Frecuencia con la que se entra en la interrupción del encoder izquierdo
        float frecuenciaDer = 0.0; //Frecuencia con la que se entra en la interrupción del encoder derecho
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE){
            frecuenciaIzq = frecuencia_izq; //Se recogen los valores procedentes de las interrupciones dentro de un bloque ATOMIC, para evitar errores
            frecuenciaDer = frecuencia_der;
        }

        // Convertir tics/s a RPM (cálculo de velocidad de cada rueda en [rpm])
        float Vizq = frecuenciaIzq/182.5*60.0;
        float Vder = -frecuenciaDer/198.0*60.0;

        // Filtro LP(Low-pass filter) (25 Hz cutoff) para limpiar la señal ruidosa de la velocidad que se entrega a los controladores de ambas ruedas
        VizqFilt = 0.854*VizqFilt + 0.0728*Vizq + 0.0728*VizqPrev;
        VizqPrev = Vizq;
        VDerFilt = 0.854*VDerFilt + 0.0728*Vder + 0.0728*VDerPrev;
        VDerPrev = Vder;
        delay(5);
        tiempo_actual = millis();
        delta_tiempo = tiempo_actual-tiempo_anterior;
        Prints();

        if(yaw <= -2.05){           //Sabiendo el yaw (orientación del robot), una vez se ha girado 90° (debería ser -pi/2, pero no giraba 90° con ese valor):
            Parar();                 //Se para el robot
            ContTicsIzq = 0;          //Volvemos a estar en "(0,0)", lo cual se realiza reseteando ambos contadores de tics
            ContTicsDer = 0;
            estado = 0;               //Se pasa al estado 0. De esta manera, el robot estará realizando ciclicamente ambos procesos, realizando el cuadrado de 1m de lado
        }
        break;
}

```

*Figura 9.15: Proceso cíclico basado en dos estados (avance recto + pivotado) para realizar la trayectoria cuadrada de 1 metro de lado deseada, con la inclusión del cálculo de velocidades y Prints() para poder obtener la telemetría a lo largo de todo el experimento.*

Para observar el robot en acción realizando este test, la mejor forma es un video que muestre tanto el robot en sí realizando la trayectoria, como los registros de posición y demás que salen por puerto serie al mismo tiempo que el robot se desplaza a lo largo de la trayectoria:

<https://drive.google.com/file/d/125OZ907ngKZ2CoBwlqku32Q5v2ZsChxt/view?usp=sharing>

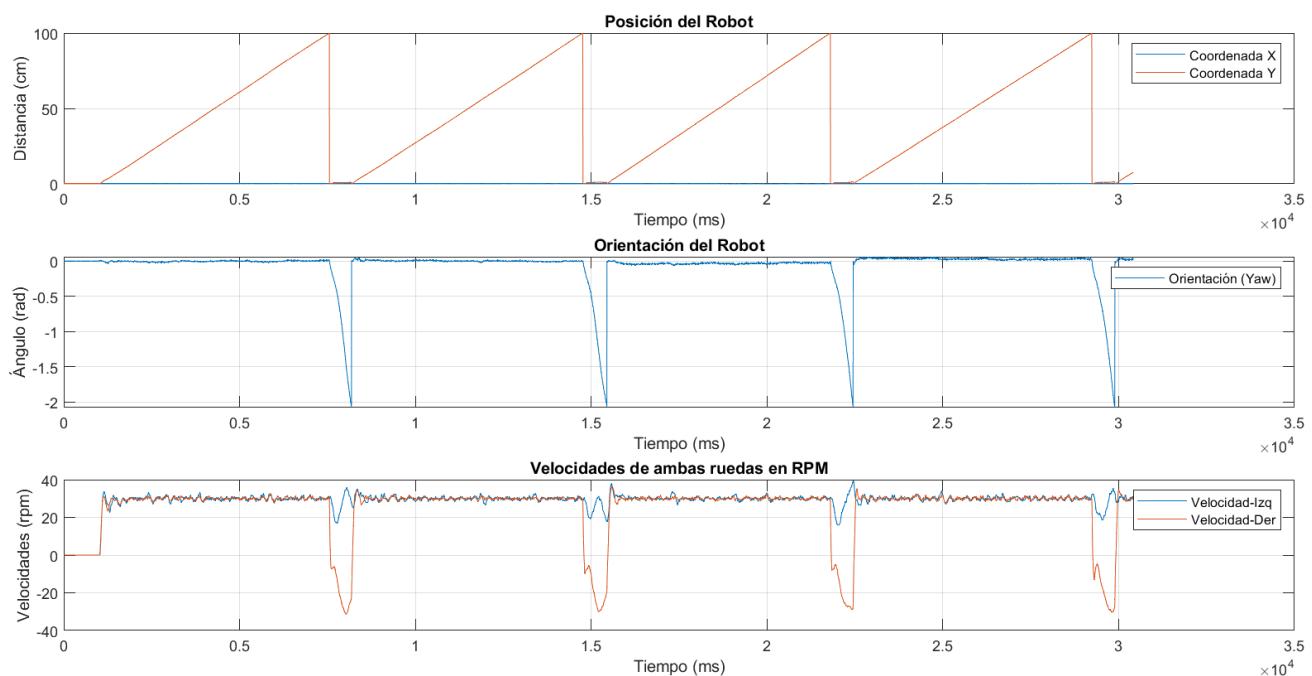
Se adjunta además un vídeo sólo con el robot realizando el cuadrado para que se pueda visualizar mejor dicho experimento:

<https://drive.google.com/file/d/1zJHsSLDdmnV4s3T5tqouOvvNnUBKxGAg/view?usp=sharing>

**Nota:** aunque pueda parecer que gira antes de llegar a  $y \geq 100$  cm, lo hace bien, lo que está pasando es que, al intentar unir las 2 grabaciones en un solo vídeo, están desfasados en 1-2 segundos y por tanto parece que gira antes de llegar a 100, pero no es así. Lo menciono por si da esa sensación en el vídeo, ya que era complicado sincronizar los tiempos de ambos vídeos sin conocer mucho acerca de edición de vídeo. Aún así creo que sirve como una buena demostración de la trayectoria cuadrada de 1 metro de lado.

En el vídeo podemos ir viendo como al realizar los pivotados resetean las coordenadas, así como la estabilidad en cuanto a la trayectoria recta gracias al control de posición. Destacar el hecho de que la primera vez que realiza el cuadrado lo hace con gran precisión, pero si se deja seguir y que lo haga una 2da o 3ra vez, ya se observa como la precisión se va degradando por la acumulación de errores, conocido esto como deriva odométrica.

Es también interesante mostrar la siguiente gráfica representativa del experimento:



**Figura 9.16:** Gráfica obtenida de la telemetría durante la realización de la trayectoria en forma de cuadrado de 1 metro de lado.

Como podemos apreciar, comprobamos aquí ciertos detalles y aspectos ya comentados en cuanto al diseño y consecución de la trayectoria cuadrada como son:

- Gracias al control de posición sobre la coordenada X, el robot mantiene su coordenada X prácticamente a 0 durante toda la trayectoria.

- Para cada lado del cuadrado, vemos como la coordenada Y del robot va aumentando hasta llegar a 100 cm aplicando velocidades a ambas ruedas de 30 rpm mediante el control de velocidad, al llegar a 1 m, el robot aplica velocidades prácticamente equivalentes en valor absoluto, pero con signos invertidos. Es decir, la rueda derecha gira hacia atrás y la rueda izquierda hacia delante, produciéndose el pivotado sobre sí mismo deseado.
- Este proceso de pivotado se prolonga hasta que la orientación llega idealmente a -90º (-pi/2 rad), pero por inercias y demás factores como rozamiento del suelo, era necesario aplicar una orientación de hasta -2 rad para que de verdad girase -90º.
- Tras realizar este conjunto de procesos, el robot resetea sus coordenadas (odometría) a (X = 0 cm, Y = 0 cm), así como su orientación (Yaw = 0 rad), con lo que está listo para repetir el mismo proceso de avance de 1 m y pivotado de -90º.
- Tras realizar 4 veces este proceso, habrá llevado a cabo una trayectoria cuadrada de 1 m de lado con bastante precisión.

## 10.- MODO 8 – PROYECTO LIBRE:

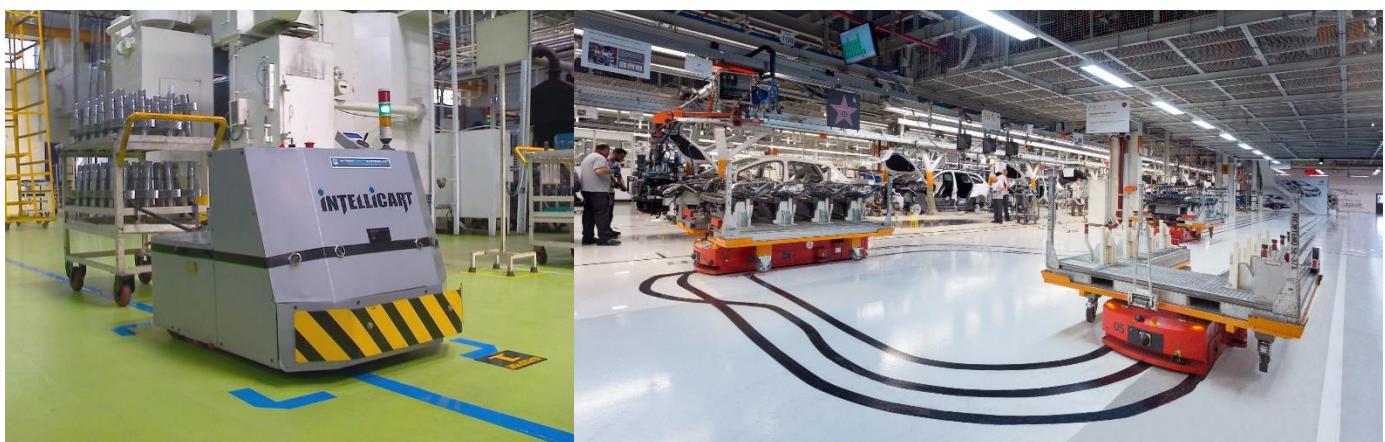
### 10.1.- *Planteamiento del proyecto*

Una vez llegados a este punto, al tratarse de un proyecto libre estuvimos dudando entre varias opciones que desarrollar, como por ejemplo la idea del aparcamiento autónomo, la idea de comprar una cámara (precio bastante elevado) y fusionar sensores con ultrasonidos para evitación de obstáculos, etc.

Sin embargo, uno de los integrantes tenía un kit de Arduino, que incluía varios sensores infrarrojos, y entonces se decidió enfocar el proyecto libre a implementar un sistema sigue líneas en nuestro robot móvil.

Como se conoce, el sistema de seguimiento de líneas por parte de robots es algo muy consolidado en robótica y muy usado sobre todo en la industria en AGVs u otros robots que se usan en fábricas o plantas automatizadas, aunque como veremos, también se puede plantear esto en un robot móvil con ruedas como el nuestro.

Algunos ejemplos de uso industrial de este sistema de guíado de robots son:



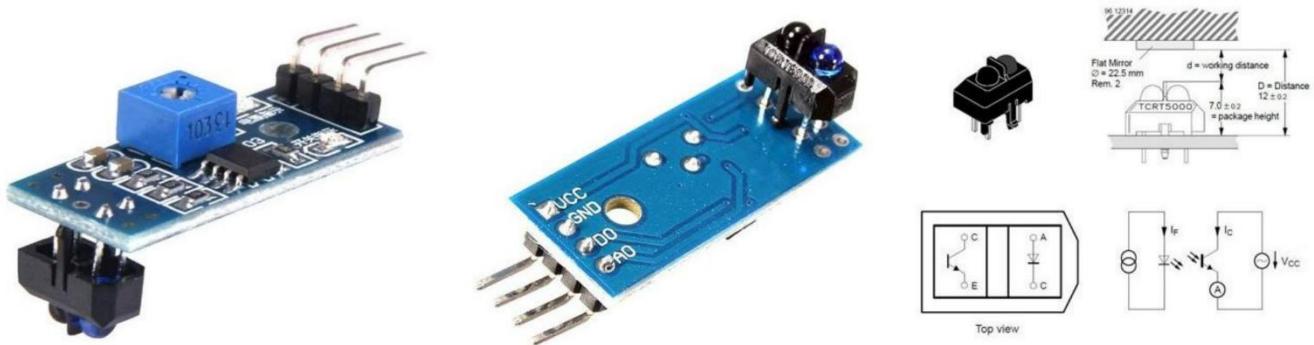
*Figura 10.1: Ejemplos de uso del sistema sigue líneas para robots en la industria, principalmente como vemos, para transporte de elementos a lo largo de la planta.*

Mencionar que nos hubiese gustado probar el robot en el circuito para sigue líneas que había montado en una de las piscinas de la ETSI, pero justo cuando nos pusimos a hacerlo lo retiraron, por lo que como se verá, nuestras pruebas son dentro de una casa con un circuito limitado por el espacio, pero donde se ven diferentes planteamientos de resolución de elementos complejos para un circuito de un robot sigue líneas.

### 10.2.- Conceptos teóricos necesarios

Aunque algorítmicamente estos sistemas no son muy complejos, si que considero necesario, al igual que en los modos anteriores, explicar las características de estos nuevos sensores.

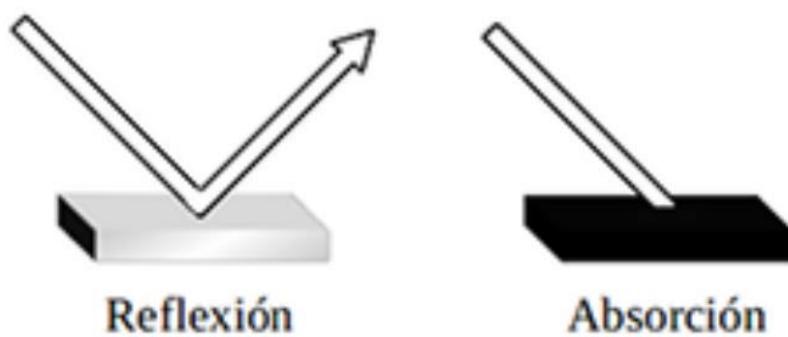
Para el desarrollo, se han usado varios sensores **TCRT5000**, los cuales son bastante compactos en tamaño y presentan una gran precisión para un precio bastante bajo (unos 2 euros cada uno).



*Figura 10.2: Sensor TCRT5000 utilizado para el sistema sigue líneas desarrollado junto con el esquemático del sistema emisor/receptor de luz infrarroja para detección de línea en este caso.*

Aunque los sensores infrarrojos se suelen usar también para detección de obstáculos, su uso es muy común también en los sistemas robóticos de seguimiento de líneas. Esto se debe a las propiedades reflectivas del color de un material, en lo cual se basan estos sistemas.

Es decir, según donde el emisor LED infrarrojo envíe el haz infrarrojo, este será absorbido casi por completo si cae en la línea negra del circuito, o bien será reflejado si cae fuera de la cinta negra del circuito. Podemos ver esto mejor con la siguiente imagen:



*Figura 10.3: Principio físico de detección de líneas que utiliza el sensor infrarrojo TCRT5000.*

De esta manera, tenemos una serie de sensores infrarrojos **activos**, es decir, con un par emisor (LED infrarrojo) y receptor (fotodiodo). De esta manera, si se emite luz infrarroja sobre el suelo (de color claro o al menos relativamente reflexivo), el fotodiodo recibe la luz reflejada y el sensor proporciona un “0” lógico en su pin digital. Si por el contrario emite la luz infrarroja sobre la cinta negra, esta será absorbida y al contrario que antes, el fotodiodo no recibe luz

reflejada, por tanto, tenemos “detección” y el sensor proporciona en este caso un “1” lógico en su pin digital.

Conociendo ahora el funcionamiento físico del sensor utilizado, pasamos a la fase de montaje y planteamiento del sistema con varios sensores combinados.

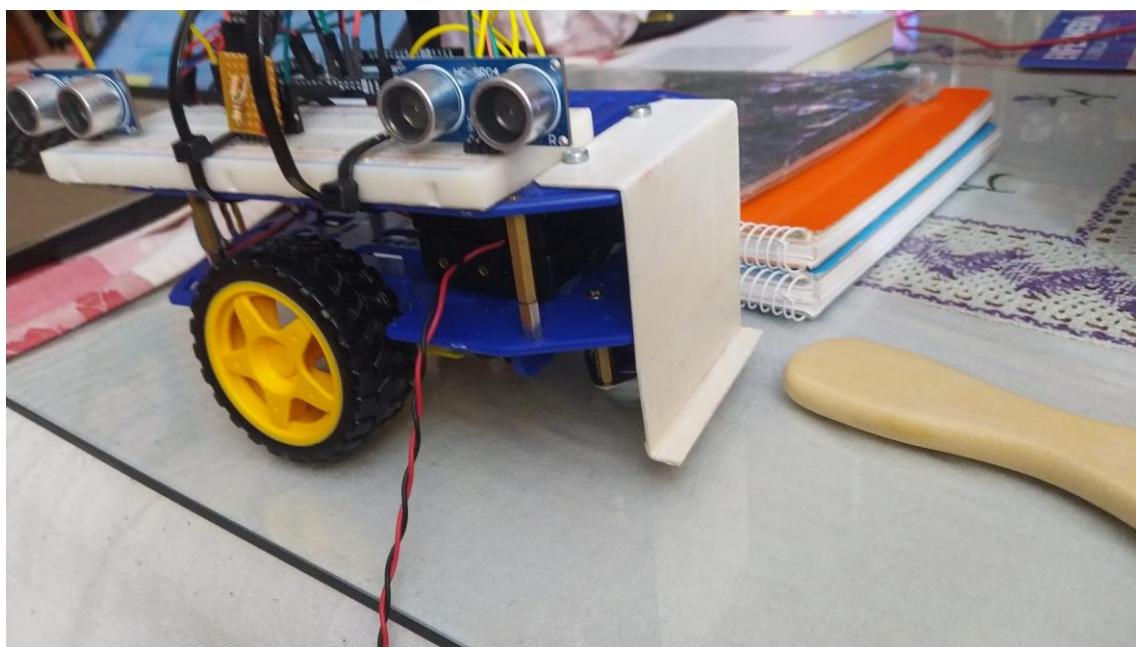
### **10.3.- Montaje y añadidos necesarios sobre el robot**

Como hemos dicho, una vez entendemos el funcionamiento básico de los sensores a utilizar, llega el turno de montar una estructura que permita a los sensores funcionar adecuadamente siguiendo la línea del circuito montado.

Para esto, es necesario tener en cuenta el rango funcional de detección de los sensores, por lo que es necesario plantear y montar la estructura a una distancia del suelo adecuada para que los sensores lleguen a detectar la línea.

Conocemos que el rango funcional de los sensores es [2,35] mm, que equivale a [0,2,3,5] cm. Por ello, el conjunto de sensores debe estar colocado a una distancia no más lejana de 3,5 cm para que puedan detectar de manera correcta.

Debido a los requerimientos mencionados de los sensores, se plantea el siguiente “complemento” a agregar al robot móvil, consistente en una chapa de aluminio plegada en forma de “Z” y envuelta en cinta aislante (para evitar posibles problemas de cortocircuitos con los cables) que se acopla en la parte delantera del robot móvil (especial agradecimiento a mi padre por dejarnos dicha chapa y ayudarme a plegarla y colocarla en el robot móvil):

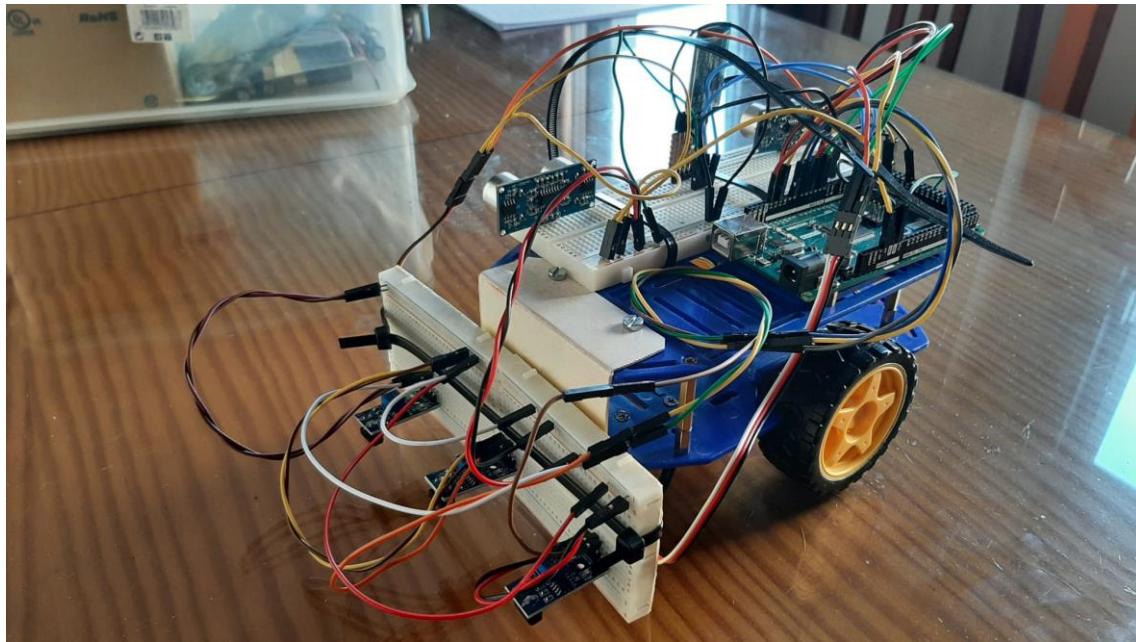


*Figura 10.4: Complemento añadido al robot móvil para integrar el sistema de sigue líneas.*

Una vez disponemos de esta base, lo que hacemos es colocar dos protoboard de tamaño medio “acopladas”, en las cuales conectaremos 3 sensores TCRT5000 adecuadamente colocados para cumplir con la correcta detección de la línea.

Esta colocación depende en gran medida del ancho de la línea negra, ya que por los planteamientos desarrollados, conviene que sólo el sensor situado en el centro sea el que detecte la cinta negra, mientras que los otros dos laterales caigan fuera de la cinta. Con esta idea, según cuando los laterales transicionen de no detectar a detectar cinta negra, tendremos conocimiento de la forma en que el robot se está desviando.

Con esto, el montaje con la protoboard y los 3 sensores ya montados sobre la chapa anterior queda tal que así:



**Figura 10.5:** Sistema final montado para lograr seguimiento de líneas con 3 sensores TCRT5000 adecuadamente espaciados.

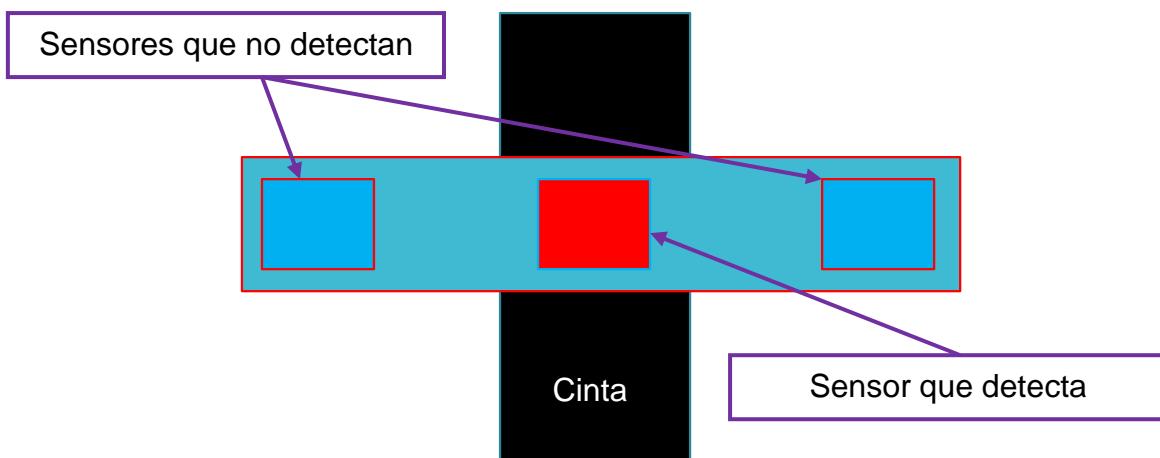
Tras terminar con el montaje, se puede pasar a la parte algorítmica y resolución del problema de seguimiento de líneas con la conjunción de estos 3 sensores tal cual los tenemos colocados.

#### **10.4.- Programación y resolución del proyecto**

Como hemos dicho, se ha buscado que cuando el robot avanza recto y siguiendo la línea, sólo sea el sensor central el que detecte la línea negra, para que cuando alguno de los laterales pase de no detectar cinta a detectarla, conoceremos en qué sentido ha ocurrido la desviación del robot móvil respecto de la cinta, para aplicar la corrección necesaria para que vuelva a la cinta lo más rápido posible, logrando un seguimiento relativamente continuo y suave del circuito.

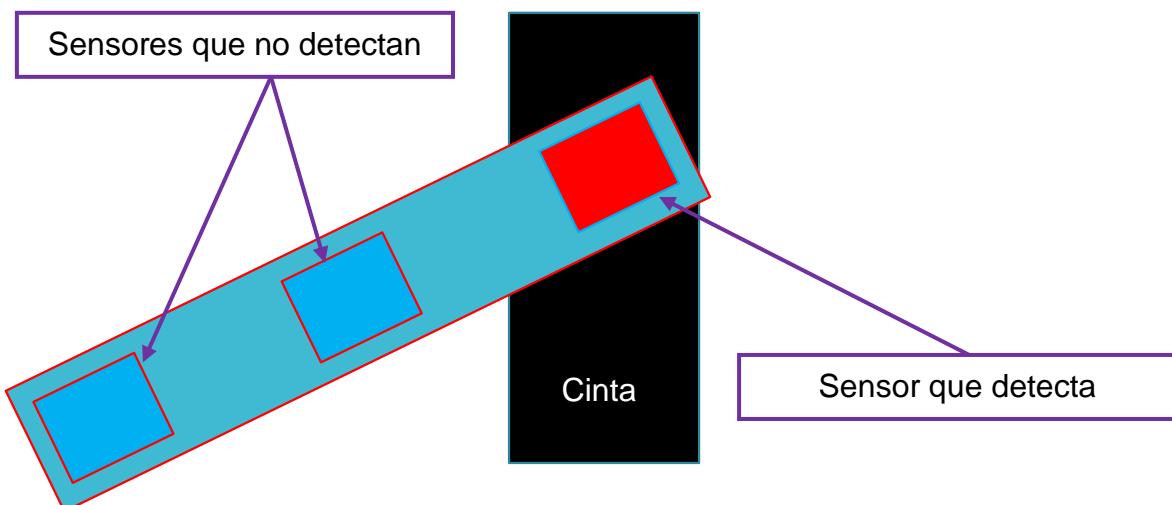
Podemos ver esta idea con los siguientes esquemas.

- Caso en que el robot avanza recto respecto de la cinta:



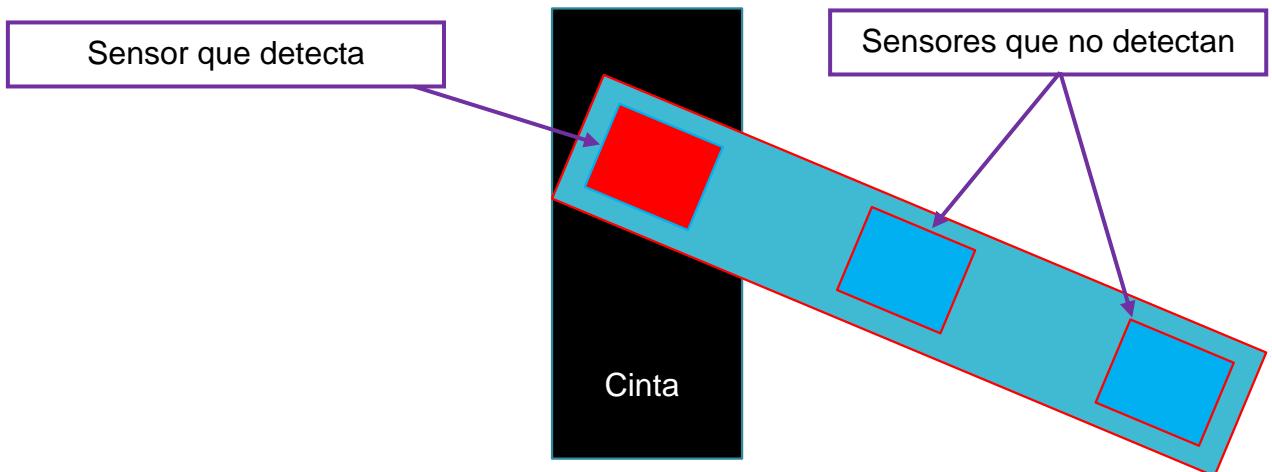
*Figura 10.6: Detección de los sensores cuando el robot avanza recto relativo a la cinta.*

- Caso en que el robot se desvía hacia la izquierda respecto de la cinta:



*Figura 10.7: Detección de los sensores cuando el robot se desvía hacia la izquierda respecto de la cinta.*

- Caso en que el robot se desvía hacia la derecha respecto de la cinta:



*Figura 10.7: Detección de los sensores cuando el robot se desvía hacia la derecha respecto de la cinta.*

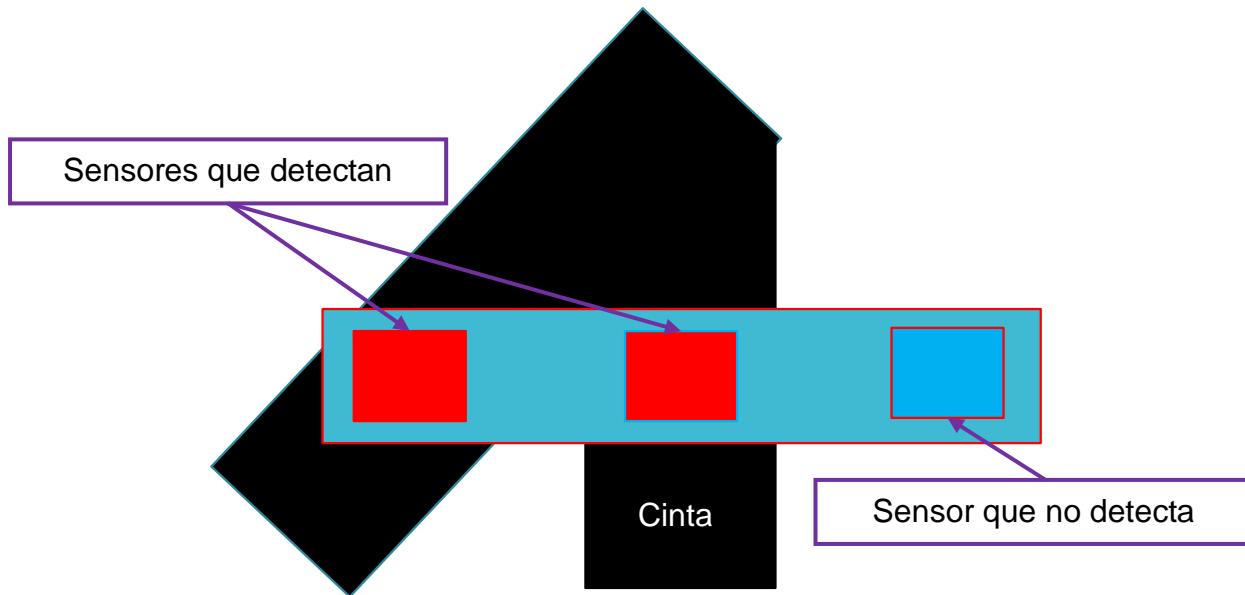
Con estos 3 casos, se puede conseguir un seguimiento de un circuito suave (realizando correcciones hacia el sentido contrario según el desvío que detectemos), es decir, sin ángulos rectos o agudos, que plantean dificultades añadidas, que también se han tratado de resolver.

Para poder tratar ángulos rectos o agudos, se ha planteado una doble detección, tanto de uno de los sensores laterales y del central, de manera que según cual de los laterales se activa junto al central, conocemos que tenemos un giro brusco en el circuito hacia izquierda o derecha, y para conseguir llevarlo a cabo y retomar el circuito, deberemos aplicar una diferencia de velocidades importante entre ambas ruedas para girar hacia donde corresponda.

Si hacemos un inciso, es importante mencionar que se ha hecho uso del **Control de Velocidad** desarrollado en modos anteriores, para poder realizar correcciones y ajustes con mucho más control que si se hiciese mediante PWM, que es mucho más arbitrario y dependiente además del voltaje que ofrece la batería. Por tanto, hacer uso de dicho control de velocidad ofrece una mayor robustez respecto de realizar las correcciones con PWM directamente.

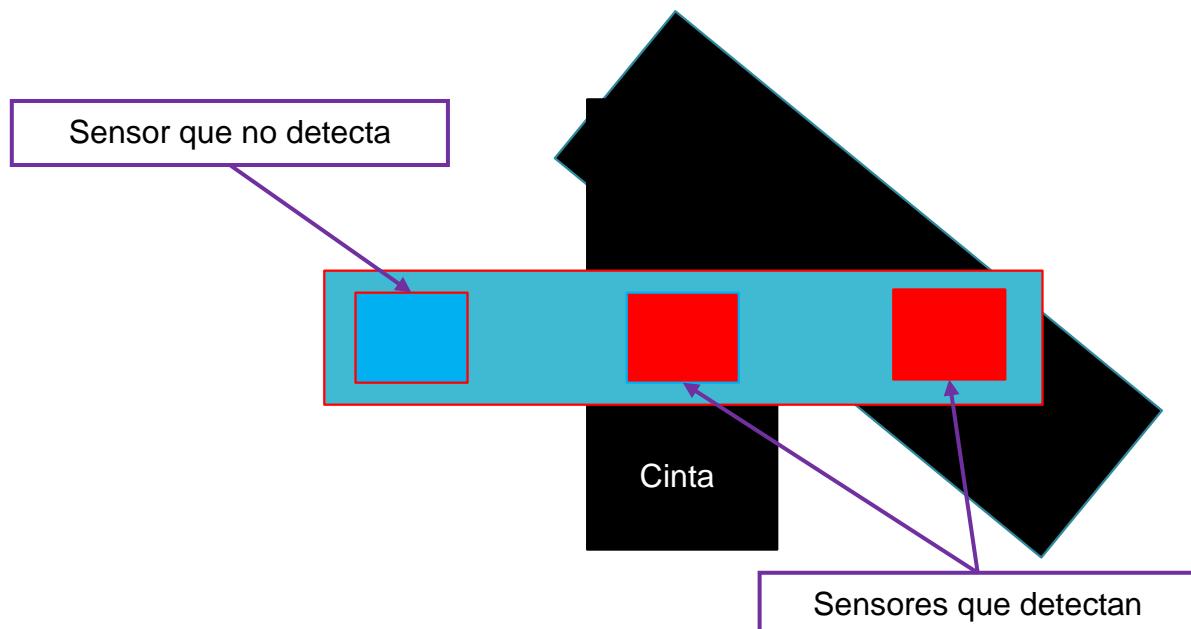
Vamos a representar gráficamente la idea planteada para poder atacar ángulos agresivos en el circuito:

- Caso en que el robot se encuentra con un giro brusco hacia la izquierda:



*Figura 10.8: Detección de los sensores cuando el robot se enfrenta a un giro brusco hacia la izquierda a lo largo del circuito.*

- Caso en que el robot se encuentra con un giro brusco hacia la derecha:



*Figura 10.8: Detección de los sensores cuando el robot se enfrenta a un giro brusco hacia la derecha a lo largo del circuito.*

Con esto, conocemos ya los diferentes casos a los que se puede enfrentar el robot a lo largo de su avance siguiendo la línea que define el circuito, por tanto podemos pasar ya a la parte de programación donde básicamente vamos a considerar estos casos descritos y definir el comportamiento del robot en cada uno de ellos para garantizar un adecuado seguimiento del circuito sin importar la dificultad del circuito.

Comenzamos con las variables utilizadas para este código:

```
SigueLineas

#include <stdio.h>
#include <util/atomic.h>

// Motor A (derecha)
#define ENA 6
#define IN1 22
#define IN2 23

// Motor B (izquierda)
#define ENB 5
#define IN3 24
#define IN4 32

// Sensores infrarrojos
#define Sensor_Ifr_Dcha 7
#define Sensor_Ifr_Izq 8
#define Sensor_Ifr_Centro 11

//Pin para Encoder Derecho
int SignalEncoder_Derecha = 2;

//Pin para Encoder Izquierdo
int SignalEncoder_Izquierda = 3;

//Variables para el control del siguelíneas
boolean flag_salido; //Variable bandera para detectar que el robot se ha salido de la linea
unsigned long tiempo_salido;
boolean flag_girobrusco; //Variable bandera para detectar si el robot se apromia a un giro recto o agudo

//Variables para recibir la referencia de velocidad a través del buffer que guarda los datos del puerto serie
unsigned long int velocidad_buffer = 0;
unsigned long int velocidad_izq_ref = 0;
unsigned long int velocidad_der_ref = 0;

unsigned long tiempo = 0;
unsigned long tiempo_ant;

//Variables para el Buffer:
const int LongitudBuffer = 50;

//Creación de variables volátiles para aquellas que se utilizan en una interrupción
volatile int ContTicsIzq = 0;
volatile int ContTicsDer = 0;
volatile float frecuencia_izq = 0;
volatile float frecuencia_der = 0;
volatile long prevTIzq = 0;
volatile long prevTDer = 0;

//Variables para realizar el filtrado de la velocidad calculada (filtro LowPass)
float VIZqFilt = 0;
float VIZqPrev = 0;
float VDerFilt = 0;
float VDerPrev = 0;

//Variables para el error integral de los controladores PI
float eintegral1 = 0;
float eintegral2 = 0;
```

**Figura 10.9:** Conjunto de todas las variables utilizadas para el código de seguimiento de línea desarrollado para este proyecto libre.

Pasamos ahora a la parte del Setup, donde deberemos añadir la configuración de los pines que gestionan la información de los nuevos sensores infrarrojos, recordar además que estamos utilizando el control de velocidad en conjunto con el nuevo sistema sigue líneas, por tanto volvemos a hacer uso de los encoders como se aprecia en los pines 2 y 3.

```
void setup()
{
    pinMode(SignalEncoder_Derecha, INPUT_PULLUP); //Pin para el encoder derecho como entrada con resistencia de pull_up
    attachInterrupt(digitalPinToInterrupt(SignalEncoder_Derecha), EncoderDerecha, RISING); //Se va a contar los pasos POR POLOS del encoder por interrupción, para ello se indica el pin que
                                                //dispara la interrupción (solo por flanko de subida, ya que el pin se pondrá a 1 cada vez que pase por un polo)
                                                // y la rutina de interrupción que en este caso será encoder().

    pinMode(SignalEncoder_Izquierda, INPUT_PULLUP); //Pin para el encoder izquierdo como entrada con resistencia de pull_up
    attachInterrupt(digitalPinToInterrupt(SignalEncoder_Izquierda), EncoderIzquierda, RISING);
    pinMode (ENA, OUTPUT);
    pinMode (ENB, OUTPUT);
    pinMode (IN1, OUTPUT); //Pines para el control de motores mediante Puente-H
    pinMode (IN2, OUTPUT);
    pinMode (IN3, OUTPUT);
    pinMode (IN4, OUTPUT);
    //Sensores infrarrojos
    pinMode (Sensor_Ifr_Dcha, INPUT);
    pinMode (Sensor_Ifr_Centro, INPUT);
    pinMode (Sensor_Ifr_Izq, INPUT);
    Serial1.begin(38400); //Setup de los puertos serie tanto para usar bluetooth, como para usar "monitor serial" y "serial plotter" del IDE de Arduino
    Serial.begin(38400);
}
```

**Figura 10.10:** Configuración de pines tanto para encoders, control de motores y nuevos sensores infrarrojos, así como para uso de bluetooth para comunicaciones.

A continuación, tendríamos el buffer para indicar las velocidades de referencia que queremos en RPM para ambas ruedas del robot móvil. Se va a omitir la captura del código de este buffer, ya que ha sido bastante recurrente a lo largo de la memoria y aquí no se modifica.

Además, después del buffer, tenemos gran parte del código con la implementación que se hizo en los MODOS 5 y 6 para el control de velocidad prácticamente reciclada tal cual, por tanto también se van a omitir sus capturas ya que se comentó en su apartado correspondiente.

Pasamos directamente entonces a la lógica implementada con la información de los sensores infrarrojos, explicada anteriormente para los diferentes casos.

```
//Lectura de sensores infrarrojos
boolean Lectura_Sensor_Dcha,Lectura_Sensor_Centro, Lectura_Sensor_Izq;
Lectura_Sensor_Dcha=digitalRead(Sensor_Ifr_Dcha);
Lectura_Sensor_Centro=digitalRead(Sensor_Ifr_Centro);
Lectura_Sensor_Izq=digitalRead(Sensor_Ifr_Izq);
if(flag_girobrusco==0){

    if(Lectura_Sensor_Izq==0 && Lectura_Sensor_Centro==1 && Lectura_Sensor_Dcha==0){ //Si los dos sensores a 1, ir hacia adelante
        velocidad_izq_ref=35;
        velocidad_der_ref=35;
        flag_salido=0;
    }
    else if(Lectura_Sensor_Izq==0 && Lectura_Sensor_Centro==0 && Lectura_Sensor_Dcha==1){ //Si me voy por la izquierda, ir a la derecha
        velocidad_izq_ref=40;
        velocidad_der_ref=25;
        flag_salido=0;
    }
    else if(Lectura_Sensor_Izq==1 && Lectura_Sensor_Centro==0 && Lectura_Sensor_Dcha==0){ //Si me voy por la derecha, ir a la izquierda
        velocidad_izq_ref=25;
        velocidad_der_ref=40;
        flag_salido=0;
    }
}
```

**Figura 10.11:** Lógica implementada según la información recibida por parte de los infrarrojos para desviaciones del robot durante el seguimiento de una línea suave (sin giros bruscos).

Básicamente en cada ciclo del loop iremos leyendo la información de los sensores, de manera que en la figura se aprecian los casos normales (sin giros bruscos), de manera que:

- **Si sólo detecta el sensor central:**

El robot puede avanzar recto, se aplican las mismas velocidades en RPM a ambas ruedas.

- **Si sólo detecta el sensor derecho:**

El robot se ha desviado hacia la izquierda, debe corregir girando hacia la derecha, por tanto necesita mayor velocidad en la rueda izquierda que en la derecha para volver a la línea negra.

- **Si sólo detecta el sensor izquierdo:**

El robot se ha desviado hacia la derecha, debe corregir girando hacia la izquierda, por tanto necesita mayor velocidad en la rueda derecha que en la izquierda para volver a la línea negra.

Lo que se hace es, según el caso, modificar las referencias de velocidad que se entregan a los controles PI de velocidad de cada rueda, para lograr las correcciones necesarias en cada momento.

Se pasa ahora a comentar las actuaciones llevadas a cabo si nos encontramos con los giros bruscos descritos anteriormente:

```

else if(Lectura_Sensor_Izq==1 && Lectura_Sensor_Centro==1 && Lectura_Sensor_Dcha==0){ //Ángulo recto o agudo a la izquierda
    //Tengo que girar con más fuerza a la izquierda
    velocidad_izq_ref=25;
    velocidad_der_ref=60;
    flag_salido=0;
    flag_girobrusco=1;
}
else if(Lectura_Sensor_Izq==0 && Lectura_Sensor_Centro==1 && Lectura_Sensor_Dcha==1){ //Ángulo recto o agudo a la derecha
    //Tengo que girar con más fuerza a la derecha
    velocidad_izq_ref=60;
    velocidad_der_ref=25;
    flag_salido=0;
    flag_girobrusco=1;
}

else{      //Si no en la linea, parar por seguridad si pasa un cierto tiempo
    if (flag_salido==0)          //Si es el primer instante en el que se sale de la linea, anoto el tiempo
    { tiempo_salido=millis();   //Indico que el robot se ha salido de la linea
    }
    if (tiempo_anterior-tiempo_salido>3000){ //Si pasa más de tres segundos fuera de la linea
        Parar();                  //Se para por seguridad
    }
}
}

else //Si flag_girobrusco=1 -) Debo mantener girando hasta que llegue a una posición "normal"(solo detecte centro){
    if(Lectura_Sensor_Izq==0 && Lectura_Sensor_Centro==1 && Lectura_Sensor_Dcha==0){ //Si vuelvo a una posición "normalita"
        flag_girobrusco=0;
    }
    else{}
}

```

**Figura 10.12:** Lógica implementada según la información recibida por parte de los infrarrojos para giros bruscos encontrados durante el seguimiento de la línea.

Vemos aquí la parte de los giros bruscos también descrita anteriormente:

- **Si detectan el sensor central y el sensor izquierdo:**

El robot se encuentra con un giro brusco hacia la izquierda, el cual será atacado imponiendo una velocidad mucho más rápida en la rueda derecha, para que gire agresivamente hacia la izquierda. Mientras está realizando este giro, entra en el último *else* de la figura anterior, donde se indica que debe mantener girando de dicha forma hasta que el robot retorne a una posición en que sólo sea el sensor central el que detecte, lo que implicaría que el robot ha vuelto al circuito tras superar el giro brusco.

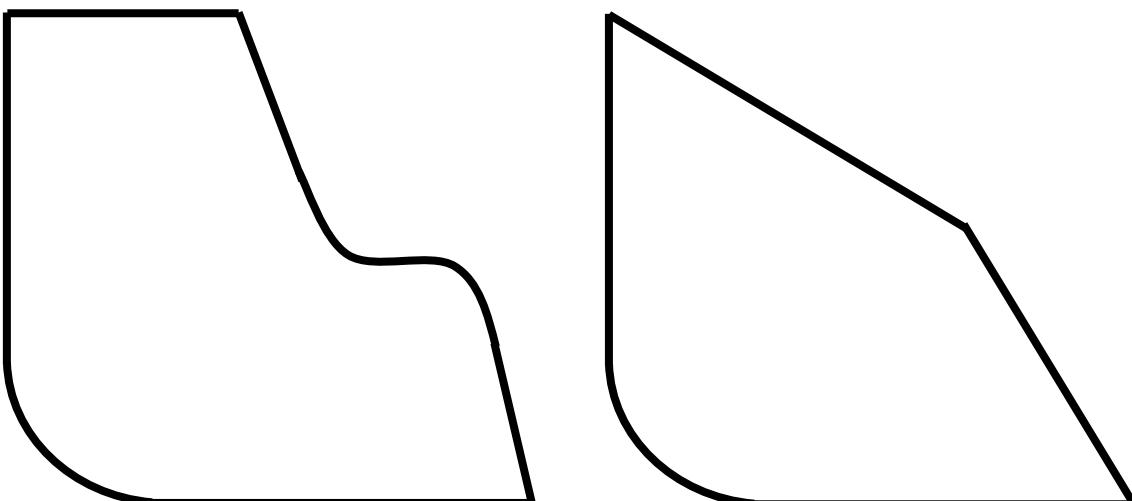
- **Si detectan el sensor central y el sensor derecho:**

El robot se encuentra con un giro brusco hacia la derecha, el cual será atacado imponiendo una velocidad mucho más rápida en la rueda izquierda, para que gire agresivamente hacia la derecha. Mientras está realizando este giro, entra en el último *else* de la figura anterior, donde se indica que debe mantener girando de dicha forma hasta que el robot retorne a una posición en que sólo sea el sensor central el que detecte, lo que implicaría que el robot ha vuelto al circuito tras superar el giro brusco.

Con esto ya se ha incidido en la lógica del sistema sigue líneas, siendo el resto del código la parte de los Prints para telemetría (en este caso no hemos visto necesaria obtener telemetría de este experimento ya que no hay mucha información relevante, más allá del control de velocidad) y las funciones de interrupción de cada encoder, así como las funciones para los diferentes movimientos característicos del robot móvil. Se omite esto, ya que también se explicó en sus apartados correspondientes y sería repetitivo.

### **10.5.- Resultados en dos circuitos diferentes.**

Llegamos al momento de poner a prueba el sistema sigue líneas desarrollado en nuestro robot móvil, para ello se plantean dos circuitos con la forma que se representa a continuación, elaborados con cinta negra con cierta anchura colocada sobre el suelo conformando ambos circuitos:



*Figura 10.12: Circuitos elaborados para probar el sistema sigue líneas implementado.*

Se adjuntan ahora los vídeos del robot recorriendo cada uno de los circuitos:

- **Círculo de la izquierda de la figura:**

[https://drive.google.com/file/d/1EJoQb\\_R5nEMLCsw4QPRI2kEoLsEBNMpB/view?usp=sharing](https://drive.google.com/file/d/1EJoQb_R5nEMLCsw4QPRI2kEoLsEBNMpB/view?usp=sharing)

- **Círculo de la derecha de la figura:**

[https://drive.google.com/file/d/1kClkotLHVhIBfTXD07fY-EuSkw\\_bKmuS/view?usp=sharing](https://drive.google.com/file/d/1kClkotLHVhIBfTXD07fY-EuSkw_bKmuS/view?usp=sharing)

Finaliza con esto tanto el proyecto libre, como la memoria explicativa del trabajo desarrollado con el robot móvil.

## 11.- CONCLUSIONES

Finalmente, llegamos al apartado de conclusiones, donde voy a aportar ciertas opiniones personales acerca del trabajo con robot móvil.

Por una parte, es muy satisfactorio conseguir que lo que desarollas e implementas en código funciona como esperas y cumple con lo que se pide.

Sin embargo, también es frustrante el hecho de trabajar con elementos físicos reales (sensores, fricciones, rozamientos, etc.) y sus notorios efectos sobre el sistema de robot móvil. Como por ejemplo el voltaje de la batería, o el desfase entre los motores de ambas ruedas en cuanto a cómo responde al PWM que se le aplica. Aunque sea algo molesto, está bien tener esta asignatura para tener un contacto real y digamos “montar” un robot casi desde cero, ya que es de las pocas veces (si no la única) en la carrera que se profundiza tanto en el aspecto de montaje físico de un robot (móvil en este caso), y su programación. Esto permite aprender ciertos aspectos muy importantes sobre la casuística de montar y programar robots desde sus inicios, que probablemente nos será útil en el futuro si nos dedicamos a algo relacionado con la robótica.

Por último, quiero agradecer personalmente a la atención proporcionada por el profesor Federico Cuesta, ya que nos ha guiado a la hora de ir realizando los diferentes planteamientos a desarrollar, con gran paciencia y explicando los aspectos importantes a tener en cuenta con gran detalle, para no pasarlo por alto. Así como el gran detalle que ofrecía a la hora de responder dudas por correo electrónico, lo cual también se agradece enormemente.

Mencionar que, aunque no me encanta la parte de montaje electrónico de un sistema, sí que he sentido como se han integrado ciertos conocimientos teóricos que traímos de otras asignaturas previas en la carrera, como electrónica y control, principalmente.

## 12.- ANEXO

Se adjuntan enlaces a Github con los códigos (si se quiere acceder y da algún problema, póngase en contacto conmigo):

### MODO o:

- Código Movimientos Básicos:

[https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modoo/modoo\\_mov\\_basicos.ino](https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modoo/modoo_mov_basicos.ino)

### MODOS 1 y 2:

- Código Modos 1-2:

[https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modos1\\_2/Control\\_Modo\\_1\\_y\\_2.ino](https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modos1_2/Control_Modo_1_y_2.ino)

### MODO 3:

- Código Modo 3:

[https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo3/Modo\\_3.ino](https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo3/Modo_3.ino)

### MODO 4:

- Código Modo 4 funcional para el test solicitado:

[https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo4/Control\\_PI\\_Paralelo\\_Modo4\\_completo.ino](https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo4/Control_PI_Paralelo_Modo4_completo.ino)

- Código Modo 4 planteamiento genérico (no validado):

[https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo4/Modo4\\_generico.ino](https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo4/Modo4_generico.ino)

### MODOS 5 y 6:

- Código Modos 5-6:

[https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modos5\\_6/Control\\_Modo\\_5\\_y\\_6.ino](https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modos5_6/Control_Modo_5_y_6.ino)

### MODO 7:

- Código Odometría (sin control de posición):

<https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo7/Odometria.ino>

- Código Test Pivote (ya con el control de posición):

[https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo7/Intento\\_Pivote.ino](https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo7/Intento_Pivote.ino)

- Código Test Cuadrado:

[https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo7/Cuadrado\\_Odometria.ino](https://github.com/Javirc99/LaboratorioRobotica/blob/main/Modo7/Cuadrado_Odometria.ino)

## **PROYECTO LIBRE:**

- Código que implementa el sistema sigue líneas combinado con el control de velocidad de los Modos 5 y 6:

[https://github.com/Javirc99/LaboratorioRobotica/blob/main/ProyectoLibre/Proyecto\\_libre.ino](https://github.com/Javirc99/LaboratorioRobotica/blob/main/ProyectoLibre/Proyecto_libre.ino)