

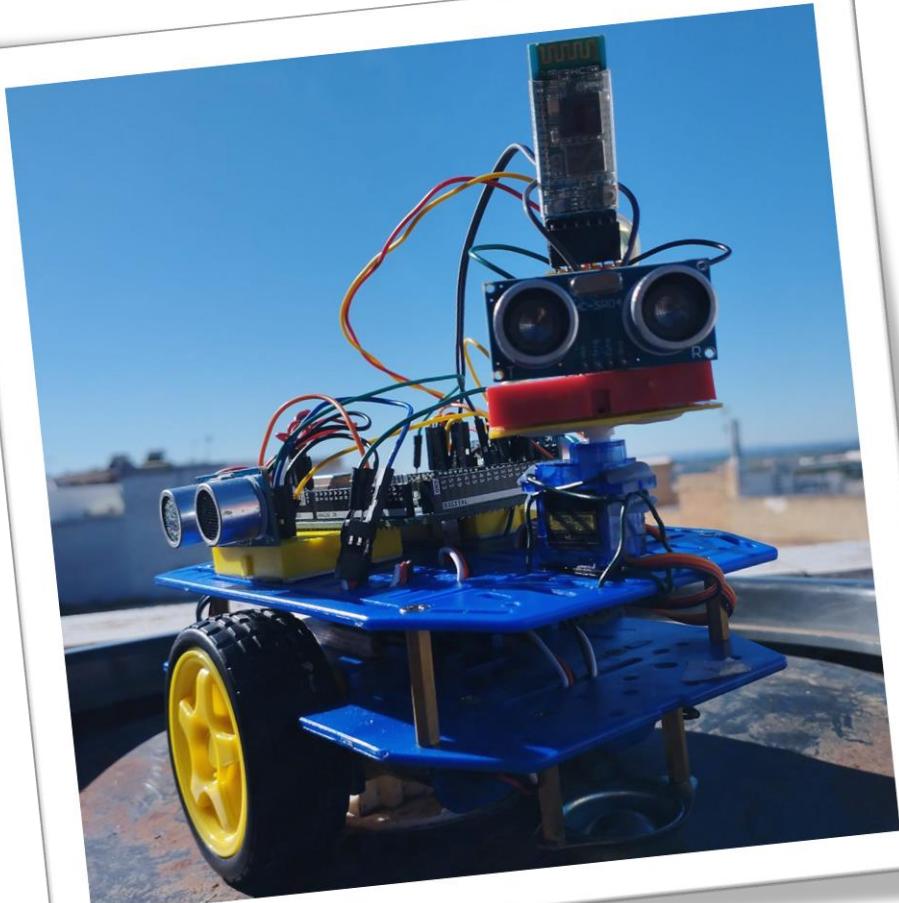
MEMORIA ROBOT MÓVIL

Memoria de trabajo desarrollado y resultados obtenidos durante el montaje y control de un robot móvil mediante realimentación sensorial en distintos modos de funcionamiento

Laboratorio de Robótica – 4ºGIERM

G01

Curso 2022/2023



Nombre: Álvaro García Lora

CONTENIDOS

0. INTRODUCCIÓN	3
1. MOVIMIENTOS BÁSICOS.....	4
2. CONTROL CON SENSORES FRONTALES	6
2.1. MODO 1	6
2.1.1. DESCRIPCIÓN	6
2.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO	6
2.1.3. ESQUEMA DE CONTROL	8
2.1.4. CÁLCULOS ANALÍTICOS.....	9
2.1.5. IMPLEMENTACIÓN SOFTWARE	9
2.1.6. RESULTADOS EXPERIMENTALES	12
2.2. MODO 2	13
2.2.1. DESCRIPCIÓN	13
2.2.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO	13
2.2.3. ESQUEMA DE CONTROL	14
2.2.4. CÁLCULOS ANALÍTICOS.....	14
2.2.5. IMPLEMENTACIÓN SOFTWARE	16
2.2.6. RESULTADOS EXPERIMENTALES	17
3. CONTROL CON SENSORES LATERALES.....	19
3.1. MODO 3	19
3.1.1. DESCRIPCIÓN	19
3.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO	19
3.1.3. ESQUEMA DE CONTROL	19
3.1.4. CÁLCULOS ANALÍTICOS.....	20
3.1.5. IMPLEMENTACIÓN SOFTWARE	20
3.1.6. RESULTADOS EXPERIMENTALES	21
3.2. MODO 4	22
3.2.1. DESCRIPCIÓN	22
3.2.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO	22
3.2.3. ESQUEMA DE CONTROL	22
3.2.4. CÁLCULOS ANALÍTICOS.....	23
3.2.5. IMPLEMENTACIÓN SOFTWARE	23
3.2.6. RESULTADOS EXPERIMENTALES	23

4. CONTROL SENSORES VELOCIDAD	25
4.1. MODO 5	25
4.1.1. DESCRIPCIÓN	25
4.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO	25
4.1.3. ESQUEMA DE CONTROL	26
4.1.4. CÁLCULOS ANALÍTICOS.....	26
4.1.5. IMPLEMENTACIÓN SOFTWARE	27
4.1.6. RESULTADOS EXPERIMENTALES	28
4.2. MODO 6	29
4.2.1. DESCRIPCIÓN	29
4.2.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO	29
4.2.3. ESQUEMA DE CONTROL	29
4.2.4. CÁLCULOS ANALÍTICOS.....	30
4.2.5. IMPLEMENTACIÓN SOFTWARE	30
4.2.6. RESULTADOS EXPERIMENTALES	31
5. ESTIMACIÓN DE POSICIÓN	33
5.1. MODO 7	33
5.1.1. DESCRIPCIÓN	33
5.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO	33
5.1.3. ESQUEMA DE CONTROL	33
5.1.4. CÁLCULOS ANALÍTICOS.....	34
5.1.5. IMPLEMENTACIÓN SOFTWARE	35
5.1.6. RESULTADOS EXPERIMENTALES	37
6. PROYECTO LIBRE.....	40
6.1. MODO 8	40
6.1.1. DESCRIPCIÓN	40
6.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO	40
6.1.3. ESQUEMA DE CONTROL	42
6.1.4. IMPLEMENTACIÓN SOFTWARE	44
6.1.5. RESULTADOS EXPERIMENTALES	51
6.1.6. AMPLIACIÓN CONTROLADOR PURE-PURSUIT	57

0. INTRODUCCIÓN

En la presente memoria se recogerán las explicaciones del trabajo realizado durante el curso sobre la construcción de un robot móvil con ruedas en configuración diferencial y el control realizado del mismo para conseguir distintos modos de funcionamiento mediante realimentación sensorial.

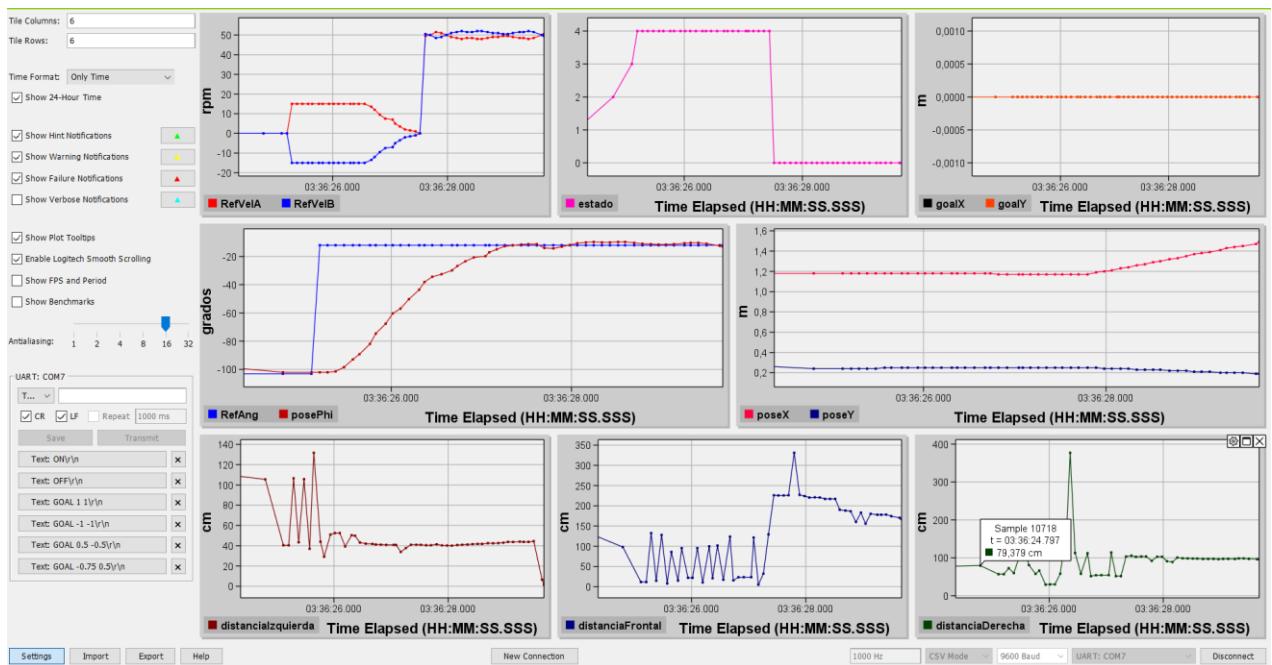
Nos centraremos en los componentes hardware usados, esquemas de conexionado realizados, metodologías de control implementadas, algoritmos y estrategias software desarrolladas en los códigos de nuestros programas, así como análisis de resultados obtenidos.

A continuación, se detallan algunas consideraciones fundamentales que hemos adoptado por iniciativa propia:

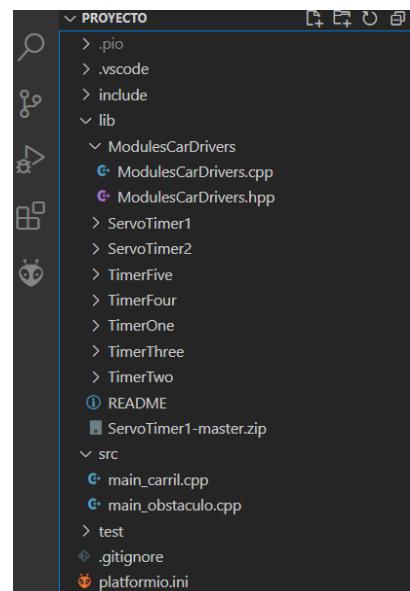
- Hemos creado nuestra propia librería software para tratar de reducir el volumen de código en los ficheros principales, dotar de claridad a los programas y hacerlos más generales. Para ello, se ha identificado aquellas partes a las que se necesitarían recurrir frecuentemente en los distintos modos como son, por ejemplo: la toma de medidas de distancia de los sensores de ultrasonidos, controlador PID, movimientos básicos del coche, envío y recepción de datos para telemetría y órdenes para control del vehículo...

De esta forma, haciendo uso de los conceptos de clases y métodos (funciones asociadas a cada clase) en C++, logramos una separación en módulos funcionales independientes. Simplemente debemos incluir la librería en la cabecera de nuestro programa principal. Durante el desarrollo de los distintos modos se ha ido añadiendo funcionalidades a la librería. Por tanto, el software desarrollado de cada uno de los modos cuenta con 3 ficheros distintos:

- main.cpp: archivo principal del modo en cuestión.
- ModulesCarDrivers.cpp: archivo con implementación detallada del código perteneciente a los métodos de cada clase usada en la librería.
- ModulesCarDrivers.hpp: archivo de cabeceras que contiene las declaraciones de las diferentes clases usadas en la librería.
- A la hora de realizar pruebas experimentales para la verificación y validación del funcionamiento de los distintos modos se ha hecho uso de un software externo desarrollado en Java, el cual hemos adaptado y configurado para poder comunicarnos con el robot. De esta forma conseguimos enviar órdenes como encendido/apagado y cambios de referencia en tiempo de ejecución, a la vez que recibimos la telemetría en vivo, la cual podemos representar y entender mejor lo que ocurre en cada momento. Se ha dedicado un tiempo a implementar esta filosofía de trabajo frente a la de recoger los datos para un procesamiento a posteriori. Hemos tomado esta iniciativa con vistas a lograr un mayor grado de entendimiento experimental de nuestro sistema y agilizar las tareas de depuración y mejora del robot. Una vez se alcanzan las especificaciones fijadas y se obtiene un buen comportamiento en cada modo se pasa a realizar experimentos con garantías de éxito, de los cuales extraemos los datos para generar la telemetría a través de Matlab.



- Se ha trabajado sobre PlatformIO, una extensión del IDE Visual Code Studio, la cual nos facilita el desarrollo de código y librerías propias y de terceros. Se recomienda la lectura del fichero README.txt contenido en la carpeta de ficheros fuente si el lector desconoce el uso y compatibilidad de esta extensión, así como la adaptación si se desea usar la IDE Arduino. La arquitectura de directorios es necesaria para el uso de la herramienta. El software desarrollado se encuentra en /lib y /src.

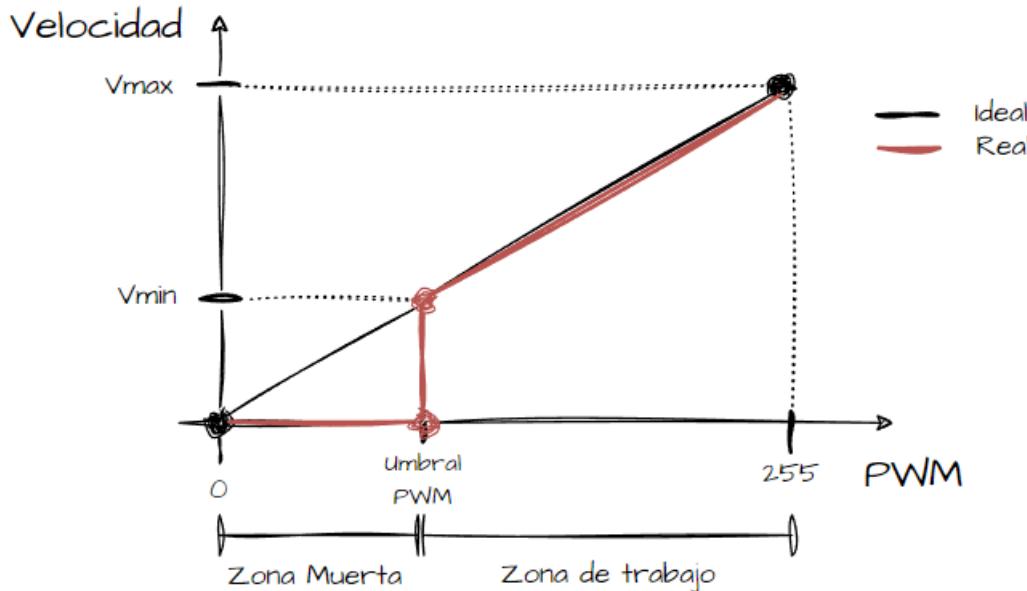


1. MOVIMIENTOS BÁSICOS

Empezamos por un control del robot en bucle abierto con el fin de verificar que el montaje básico del mismo se ha realizado satisfactoriamente. Realizamos los movimientos básicos del vehículo en los cuales podemos identificar las influencias físicas que introducen los actuadores reales y el medio sobre el movimiento ideal esperado.

Tras implementar las acciones de avanzar y retroceder en linea recta, rotar en ambos sentidos (respecto al eje centrado en el cuerpo y pivotar respecto a cada una de las ruedas) y giros de múltiples tipos, hemos identificado cuales son algunas de las consideraciones a tener en cuenta a la hora de trabajar en futuros modos con este sistema real:

- Comportamiento no simétrico de ambos motores
- Comportamiento no simétrico entre movimiento elementales (avance hacia delante y retroceso)
- Identificación de zonas muertas en aplicación de PWM a motores



El umbral PWM en vacío (coche en el aire) se localiza en un nivel de PWM igual a 35. Por otro lado, con el vehículo en el suelo dicho umbral asciende entorno al nivel de 65 aproximadamente. A partir de dicho valor de PWM es cuando realmente conseguimos tener una cierta velocidad mínima de movimiento, debido a los motores con los que contamos y rozamiento con el suelo. Este umbral no será siempre fijo, sino que se verá influido por el estado de carga de la batería que alimenta al sistema y por el tipo de suelo en el que se esté trabajando.

- Imperfecciones del suelo/terreno, deslizamiento y fricción

A la vista está la necesidad de usar sensores para conseguir realimentación e implementar técnicas de control en bucle cerrado que permitan corregir los errores que se producen, alcanzar las referencias especificadas y rechazar las perturbaciones del medio con éxito.

Estos experimentos previos han servido para realizar un preajuste de nuestro sistema, ayudando al control en bucle cerrado, con el fin de que posteriormente obtengamos mejores resultados de control.

2. CONTROL CON SENSORES FRONTALES

2.1. MODO 1

2.1.1. DESCRIPCIÓN

El objetivo será conseguir que el vehículo se pare frente a una pared a una cierta referencia de distancia fijada. Para ello únicamente contaremos con un sensor de ultrasonido con el que realizaremos las medidas de distancia con las que cerrar el bucle de control.

2.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO

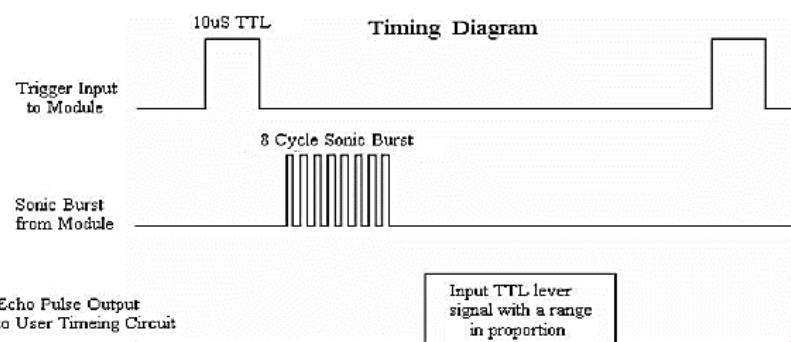
El hardware usado para el primero de los modos cuenta con los siguientes componentes:

- Placa Arduino Mega 2560:

Cuenta con el microcontrolador en el que cargaremos nuestros programas, así como con los pines de entrada y salida digital y PWM para comunicarnos con el mismo.

- Sensor Ultrasonido HC-SR04

El principio de funcionamiento se basa en la emisión y recepción cíclica de un impulso acústico de alta frecuencia. El impulso se propaga a la velocidad del sonido por el aire y al encontrar un obstáculo, rebota, siendo reflejado y volviendo como eco al sensor ultrasónico. La distancia hacia al obstáculo podemos calcularla basándonos en el tiempo que transcurre entre la emisión de la señal (por el pin de trigger) y la recepción de la señal (por el pin de echo). A continuación se muestra un diagrama de tiempos de operación del sensor, sacado del datasheet proporcionado por el fabricante:



- Módulo Bluetooth HC-05

Para comunicarnos con el microcontrolador y enviarle órdenes para controlar el sistema hacemos uso del módulo bluetooth HC-06. El programa que usamos comentado en la introducción de la memoria transmite dicha información al micro por la UART. Hemos configurado para ello las velocidades de ambos a 9600 baudios.

- Driver Puente-H L298N

Módulo para controlar la velocidad y sentido de giro de los motores de corriente continua. Alimentamos el módulo desde la placa de Arduino ya que la batería se encontrará enchufada a dicha placa.

Los motores irán conectados directamente a las salidas que tiene el módulo.

Los pines IN1 e IN2 nos sirven para controlar el sentido de giro del motor 1, y los pines IN3 e IN4 el del motor 2. Poniendo IN1 e IN2 a nivel alto y bajo y viceversa se consigue un sentido de giro o el contrario respectivamente. Igual se aplica para IN3 e IN4 para el otro motor.

Para controlar la velocidad de giro de los motores usamos los pines ENA y ENB. Los conectaremos a dos salidas PWM de Arduino de forma que le enviemos un valor entre 0 y 255 que controle la velocidad de giro.

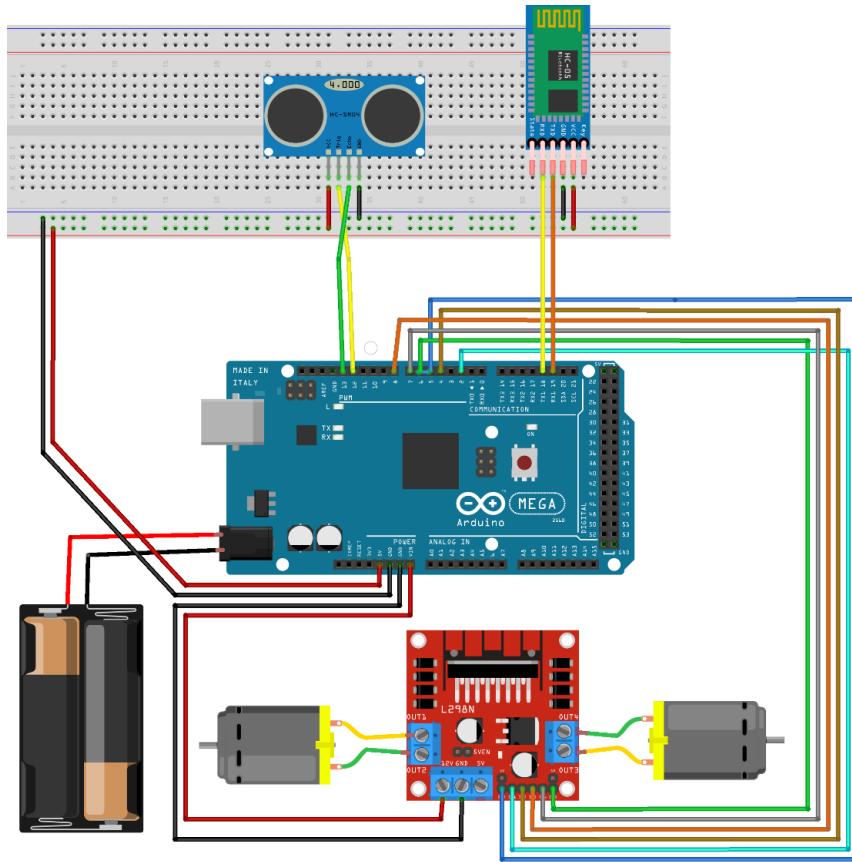
- Motores DC con caja reductora

Conectados al eje de las ruedas con una relación de transmisión de 48:1. Están alimentados por medio del puente H. Dan el movimiento al robot.

- Batería Li-Ion

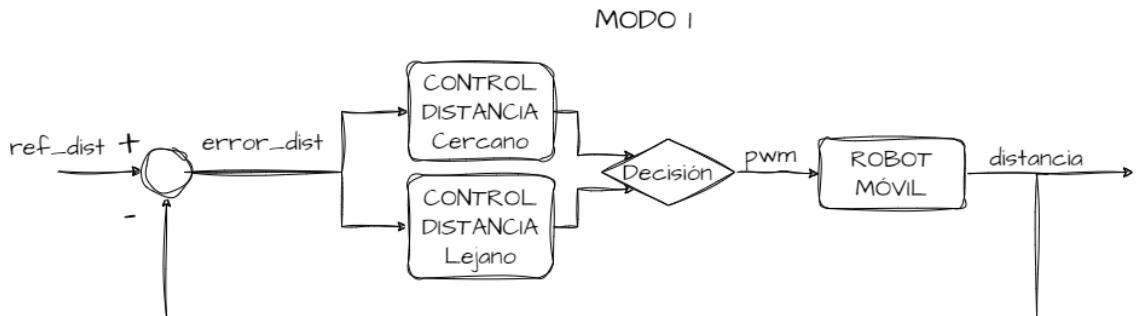
Contamos con una batería de ión de litio de 2 celdas (7.2 V) con una capacidad de 2.600 mAh la cual es la encargada de alimentar al sistema completo.

El esquema de conexionado implementado es el que se muestra:



2.1.3. ESQUEMA DE CONTROL

La arquitectura de control usada es relativamente básica. Consiste en la obtención de la distancia a la pared mediante la lectura del sensor ultrasónico mediante la cual realimentamos y hallamos el error cometido respecto a la referencia de distancia fija. Dicho error irá variando respecto al tiempo y será entrada del único controlador que disponemos en este primer modo. Se trata de un controlador encargado de minimizar el error en distancia. La salida del mismo serán las pwm a aplicar en cada motor para conseguir el movimiento de las ruedas.



Profundizamos en el bloque del controlador. Realmente se divide en dos controles PID ya que se ha implementado un control por tramos (cercano y lejano).

En el tramo lejano (error superior a una cierta cota) aplicamos solamente parte proporcional. En el tramo cercano (error cometido inferior a la cota en cuestión) incluimos término integral a la vez que se reduce la constante del proporcional. De esta forma conseguimos una aproximación relativamente rápida a la referencia para después alcanzarla suavemente con error nulo en el permanente. Cabe destacar que se está haciendo un control incremental entorno al punto de equilibrio, el cual equivale a la PWM mínima para lograr el movimiento del robot. Por tanto, la salida del controlador será una realmente una cierta PWM que se sumará al punto de equilibrio. Dicha PWM total es la que se les aplican a los motores a través del driver puente-H.

2.1.4. CÁLCULOS ANALÍTICOS

Destacamos en este apartado como se realiza el cálculo de la distancia a partir de la medida de la diferencia de tiempos entre la emisión del pulso y recepción del mismo que realiza el sensor HC-SR04.



La velocidad del sonido es conocida (tomamos condiciones de temperatura ambiente, 50% de humedad y presión atmosférica a nivel del mar). Debemos tener en cuenta que la distancia recorrida es la mitad para tener en cuenta el tiempo de ida y vuelta.

2.1.5. IMPLEMENTACIÓN SOFTWARE

Como hemos descrito en la introducción de la memoria, el código principal se encuentra bastante compactado gracias al uso de la librería desarrollada.

Tanto en este como en los sucesivos modos se mostrarán únicamente las partes principales del código, con el motivo a no sobrecargar demasiado el presente documento.

La declaración de variables, definición de parámetros, configuración de pines asociados a los distintos sensores ultrasonidos, encoders, driver de los motores, configuración de UART para comunicaciones a través de puerto USB y bluetooth, ... puede verse con detalle en los códigos disponibles.

En este sentido, vemos las partes más notorias:

```
// Tomar medida con ultrasonidos
distancia = sonar.tomaMedidaUltrasonidos();
// Calcular señal de control
int PWM = eligeControl(distancia, ControlMotoresNear, ControlMotoresFar, Ref_Distancia);

// Actuar sobre los motores
aplicaSignal(motorA, PWM);
PWM = aplicaSignal(motorB, PWM);

// Enviar datos de telemetría por Bluetooth
tiempoTranscurrido = millis() - tiempoAnterior;
tiempoAnterior = millis();

moduloBluetooth.DebugValuesBluetooth(tiempoTranscurrido, distancia, distancia, Ref_Distancia, MODO, PWM, PWM);
delay(60); // tiempo entre medidas (60 max según datasheet)
```

En resumen, tomamos la medición de la distancia actual con la cual decidir tras hallar el error cometido respecto a la referencia el controlador a usar.

Como resultado obtenemos la PWM incremental la cual es sumada al punto de operación en la aplicación de la señal a los motores. Por último, enviamos los datos de telemetría por bluetooth.

Para asegurar la correcta medida de los sensores la frecuencia máxima que recomienda el fabricante es la equivalente a tomar medidas cada 60 ms. Por debajo de esos tiempos podríamos obtener mediciones corruptas, por lo que nos quedamos del lado de la seguridad.

En la siguiente función puede verse con más detalle como seleccionamos un controlador u otro:

```
int eligeControl(float Medida, Controlador &ControlMotoresNear, Controlador &ControlMotoresFar, unsigned int referencia)
{
    int PWM_calculado;
    if (abs(Medida - (float)referencia) > SwtichDistance)
    {
        PWM_calculado = ControlMotoresFar.calcularSalida((float)referencia - Medida);
        ControlMotoresNear.resetPID();
    }
    else
    {
        PWM_calculado = ControlMotoresNear.calcularSalida((float)referencia - Medida);
        ControlMotoresFar.resetPID();
    }
    return PWM_calculado;
}
```

En esta otra vemos como se tiene en cuenta el punto de equilibrio a la hora de dar la actuación a los motores:

```
int aplicaSignal(movimientoCoche &motor, int signalControl)
{
    int PWM;
    if (signalControl > 0)
    {
        PWM = signalControl + puntoEquilibrio;
        motor.forward(PWM);
    }
    else if (signalControl < 0)
    {
        PWM = abs(signalControl) + puntoEquilibrio;
        motor.backward(PWM);
    }
    else
    {
        PWM = 0;
        motor.stop();
    }
    return PWM;
}
```

Nota importante: Recordamos que gracias al uso de la librería desarrollada podemos usar la clase Controlador la cual tiene implementada diferentes métodos (funciones) asociadas a la misma. En cada modo podemos crear un objeto de dicha clase, modificar sus atributos de acuerdo con las necesidades del controlador concreto (ajuste de constantes Kp, Ki y Kd, límites de saturación...). Dicho controlador será un PID completo con saturación, anti-windup, reinicio de errores acumulados, etc. En adelante usaremos bastantes controladores PID para los distintos modos de funcionamiento y nos basaremos en esta filosofía de trabajo. Mostramos la parte de la librería dedicada a esto ya que puede resultar de interés tenerlo presente:

```
// Clase asociada al control del coche
class Controlador
{
private: // Atributos
    float Kp, Ki, Kd;
    int min, max;
    double tiempoAnterior;
    float ErrorAnterior, ErrorAcumulativo;
    bool FlagFirstValue;

public: // Métodos
    Controlador(float _Kp, float _Ki, float _Kd, int _min, int _max); // Constructor
    void setGain(float _Kp, float _Ki, float _Kd); // Set
    void resetPID(void);
    int calcularSalida(float MeasuredError);
    double tiempoTranscurrido;
};


```

```
Controlador::Controlador(float _Kp, float _Ki, float _Kd, int _min, int _max)
{
    Kp = _Kp;
    Ki = _Ki;
    Kd = _Kd;

    min = _min;
    max = _max;

    resetPID();
}

void Controlador::setGain(float Kp, float Ki, float Kd)
{
    Kp = Kp;
    Ki = Ki;
    Kd = Kd;
}

void Controlador::resetPID(void)
{
    tiempoAnterior = 0;
    ErrorAnterior = 0;
    ErrorAcumulativo = 0;
    FlagFirstValue = true;
}
```

```
int Controlador::calcularSalida(float MeasuredError)
{
    float SalidaControlador;
    double tiempoActual;
    float ErrorDerivativo;

    tiempoActual = millis();
    tiempoTranscurrido = tiempoActual - tiempoAnterior;

    if (FlagFirstValue)
    {
        ErrorAcumulativo = 0;
        ErrorDerivativo = 0;
        FlagFirstValue = false;
    }
    else
    {
        ErrorDerivativo = (MeasuredError - ErrorAnterior) / tiempoTranscurrido;
        ErrorAcumulativo += MeasuredError * tiempoTranscurrido;
    }

    SalidaControlador = round(Kp * MeasuredError + Ki * ErrorAcumulativo + Kd * ErrorDerivativo); // calcular la salida del PID
    if (SalidaControlador > max) // saturar señal de control
    {
        ErrorAcumulativo -= MeasuredError * tiempoTranscurrido; // efecto antiwindup (dejo de acumular error)
        SalidaControlador = max ;
    }
    else if (SalidaControlador < min)
    {
        ErrorAcumulativo -= MeasuredError * tiempoTranscurrido; // efecto antiwindup (dejo de acumular error)
        SalidaControlador = min ;
    }
    ErrorAnterior = MeasuredError;
    tiempoAnterior = tiempoActual;
    return SalidaControlador;
}
```

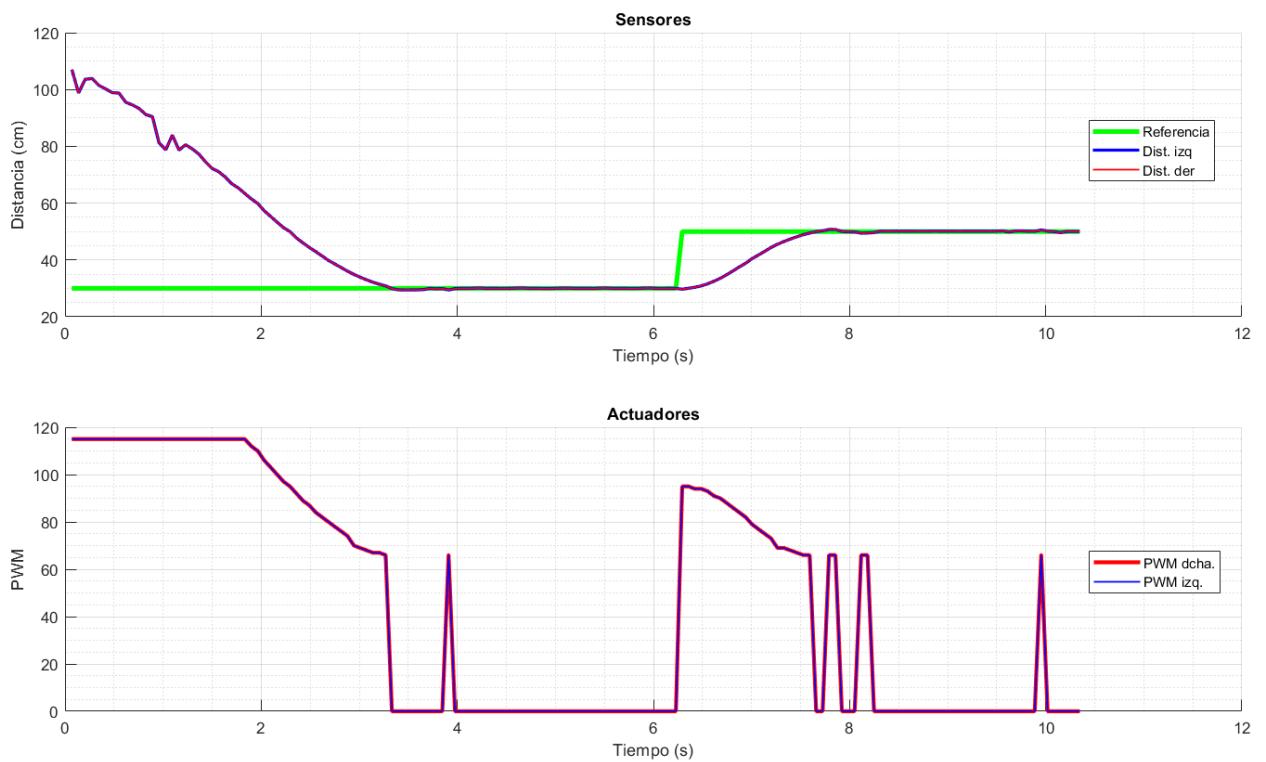
2.1.6. RESULTADOS EXPERIMENTALES

El experimento cuyos datos se muestran corresponde a la colocación del robot a una cierta distancia inicial de la pared (de un metro aproximadamente como vemos). Le indicamos que se coloque a 30 cm y tras alcanzar dicha cota cambiamos la referencia a 50 cm.

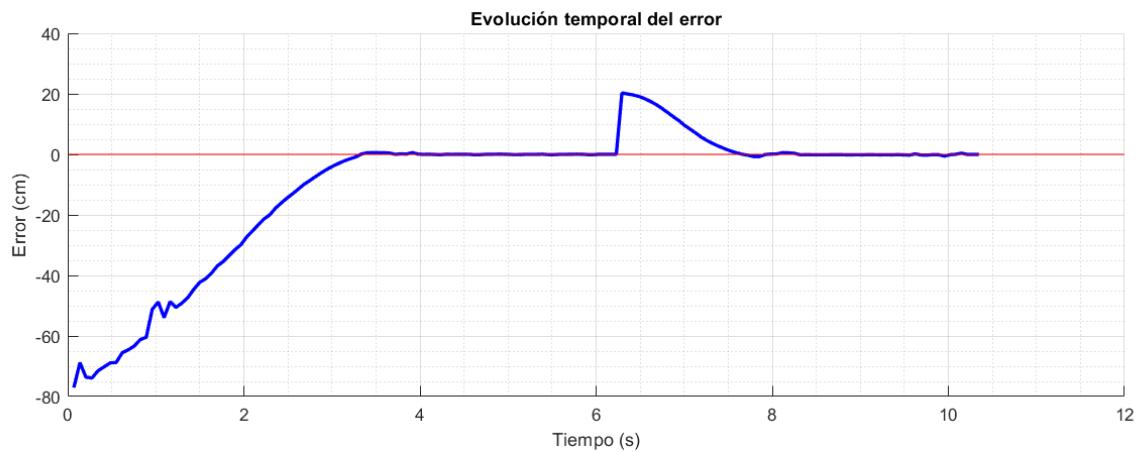
En primer lugar, analizamos la medida de la distancia y la señal de control sobre los motores.

Vemos como distancia izquierda y derecha coinciden ya que en este modo únicamente tenemos un único sensor. A la vista de los resultados obtenidos, podemos decir que se alcanza la referencia de distancia fijada sin error en régimen permanente y con un transitorio suave sin oscilaciones ni sobreoscilación.

En cuanto a la señal de control, vemos como es incremental sobre el punto de equilibrio colocado en el nivel de PWM igual a 65. Cuando la señal de control incremental que proporciona el controlador es nula no se aplica ninguna, provocando la parada del robot. Destacamos que, en este caso, la señal de control aplicada a ambos motores es la misma puesto que nuestro sistema está bastante equilibrado.



Vemos en la siguiente gráfica la evolución durante el experimento:



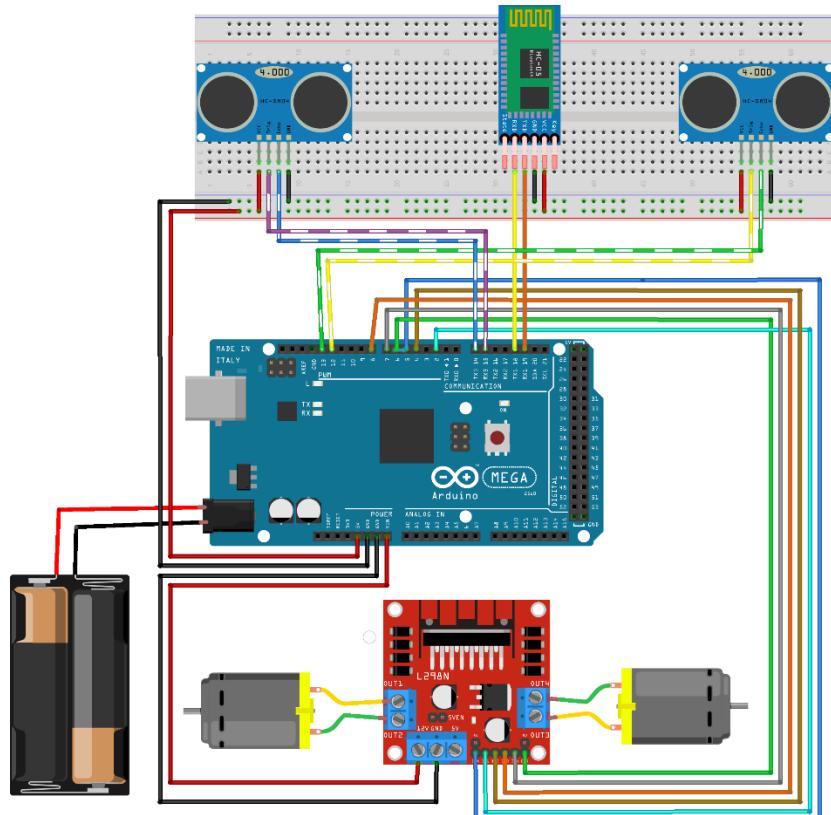
2.2. MODO 2

2.2.1. DESCRIPCIÓN

Se pretende mejorar el modo anterior incluyendo un segundo sensor de ultrasonido, tomándose una segunda medida de distancia con la que poder conseguir una orientación lo más perpendicular posible a la pared frente a la cual se mantiene la distancia fijada en la referencia que se le proporcione.

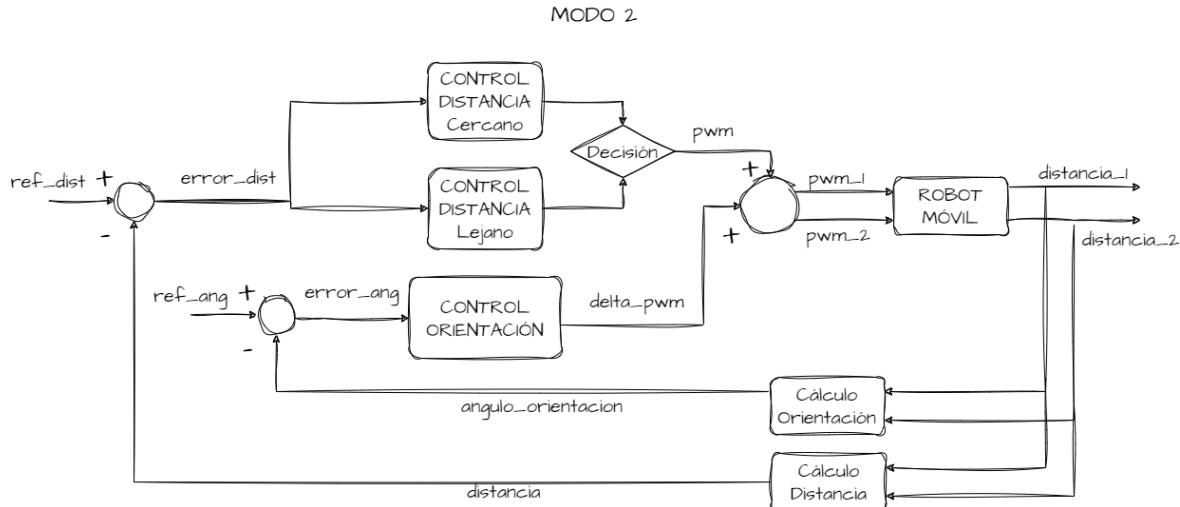
2.2.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO

Añadimos un nuevo sensor HC-SR04 manteniendo el resto del montaje idéntico respecto al modo anterior. En el esquema de conexionado que resulta es el que se muestra:



2.2.3. ESQUEMA DE CONTROL

Contar con una segunda medida de la distancia nos permite determinar la orientación que tenemos respecto a la pared para corregirla y posicionarnos de forma perpendicular a ésta. Tomaremos como distancia a la pared la distancia a la que se encuentra el punto medio de ambos sensores.



Tendremos un esquema parecido al anterior al cual se le suma un nuevo lazo de control referente a la orientación. A la salida del control de distancia (cercano o lejano) se le suma un término añadido. Quien aporta esta nueva parte de la pwm incremental es el controlador de orientación. La referencia será de 0° para conseguir nuestro objetivo.

En función del signo del ángulo que forme el coche con la pared y el tipo de movimiento (adelante o retroceso), se aportará el término extra de actuación a una rueda o a otra. De esa forma, al tener en esta ocasión señales de actuación diferentes para ambas ruedas es posible la corrección del ángulo.

Recordamos que las señales de actuación que sacan los controladores son incrementales en torno al punto de equilibrio que es el que el vehículo necesita para comenzar a moverse. Por simplicidad en los esquemas de control no estamos representando ese offset extra de PWM que se le aplica antes de entrar al motor.

2.2.4. CÁLCULOS ANALÍTICOS

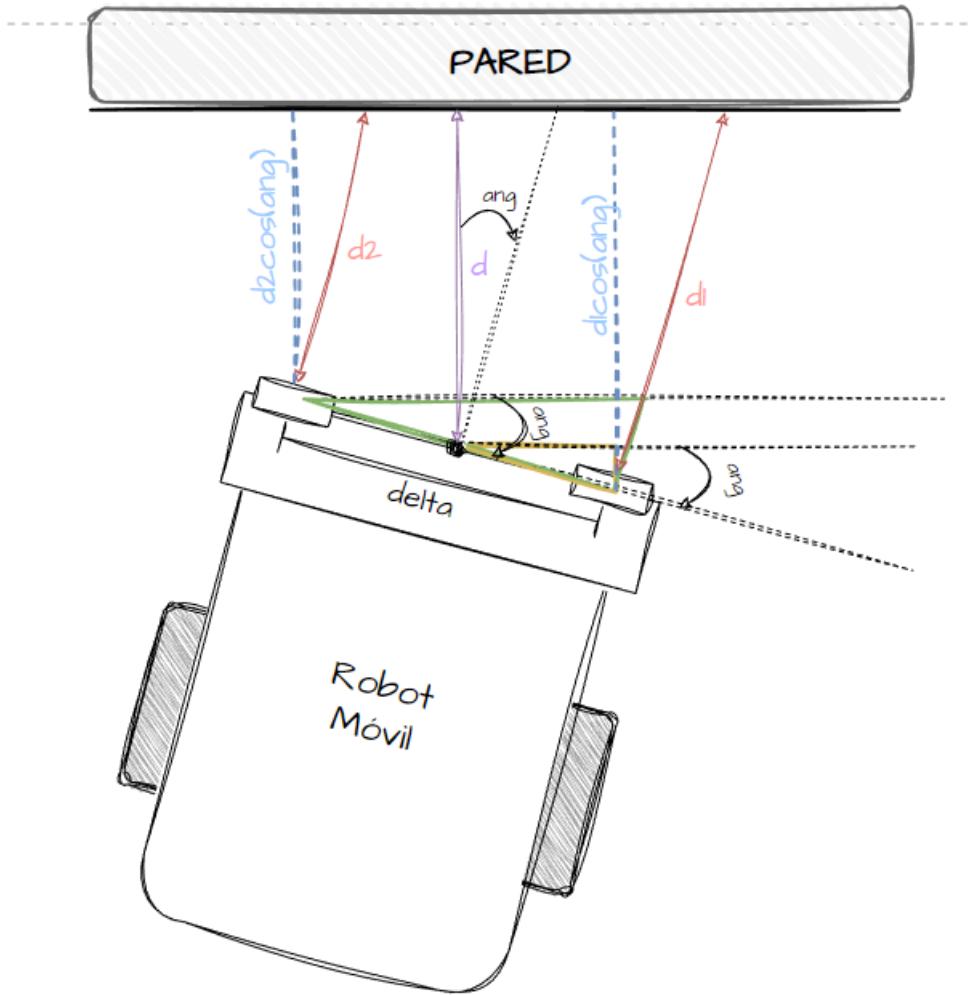
Mediante desarrollo trigonométrico podemos hallar la distancia y ángulo de orientación del coche con la pared, conocidos:

δ = Distancia entre sensores

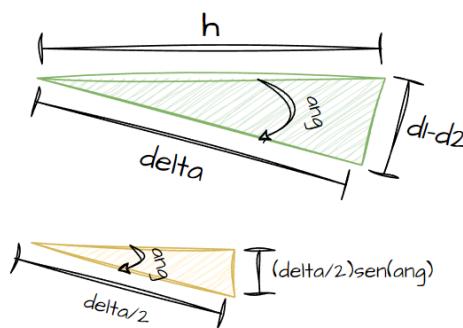
d_1 = Distancia medida por sensor 1

d_2 = Distancia medida por sensor 2

El esquema básico a partir del cual nos basamos para sacar las ecuaciones es el siguiente:



Aplicando trigonometría en los triángulos rectángulos señalados:



Visto esto y de igual forma, teniendo en cuenta el caso análogo en el que la distancia del sensor 2 fuese superior a la del sensor 1, llegamos a las expresiones que se muestran:

$$\left. \begin{array}{l} d_2 > d_1 \\ \theta = \frac{\pi}{2} - \arcsen\left(\frac{\delta}{\sqrt{\delta^2 + (d_2 - d_1)^2}}\right) \\ d = d_2 \cdot \cos(\theta) - \frac{\delta}{2} \cdot \sin(\theta) \end{array} \right| \left. \begin{array}{l} d_1 > d_2 \\ \theta = -\left(\frac{\pi}{2} - \arcsen\left(\frac{\delta}{\sqrt{\delta^2 + (d_2 - d_1)^2}}\right)\right) \\ d = d_1 \cdot \cos(\theta) - \frac{\delta}{2} \cdot \sin(\theta) \end{array} \right)$$

En el desarrollo realizado no se tiene en cuenta el cono que emite el sensor de ultrasonido, sino que idealmente este emite en una linea perpendicular al plano del sensor. Por tanto, las expresiones obtenidas sirven como aproximación a los valores deseados de ángulo y distancia. A pesar de esto, el comportamiento del robot en la práctica es bueno aplicando estas ideas ideales por lo que se ha decidido adoptarlas.

2.2.5. IMPLEMENTACIÓN SOFTWARE

Como principal novedad respecto al código del modo anterior está el cálculo del ángulo de orientación que forma el robot con la pared y la distancia del punto central de ambos sensores. Una vez que disponemos de las medidas tomadas por ambos sensores realizamos dichos cálculos como vemos:

```
// Tomar medida con ultrasonidos
d1 = sonarA.tomaMedidaUltrasonidos();
d2 = sonarB.tomaMedidaUltrasonidos();

//Cálculo ángulo de orientación y distancia punto central
if (d2 > d1)
{
    theta = PI / 2 - asin(delta / sqrt(pow(delta, 2) + pow(d2 - d1, 2)));
    distancia = d2 * cos(theta) - delta / 2 * sin(theta);
}
else if (d1 > d2)
{
    theta = -(PI / 2 - asin(delta / sqrt(pow(delta, 2) + pow(d1 - d2, 2))));
    distancia = d1 * cos(theta) - delta / 2 * sin(theta);
}
else
{
    theta = 0;
}
theta = theta * 180 / PI; // cambio de rad a grados
```

El siguiente paso sería la elección del controlador de lejanía o cercanía, la obtención de la pwm incremental para ambas ruedas y la delta_pwm que debe tener una rueda sobre la otra en función del ángulo theta hallado:

```
// Calcular señal de control
int PWM = eligeControl(distancia, ControlDistanciaNear, ControlDistanciaFar, Ref_Distancia);
int delta_PWM = ControlOrientacion.calcularSalida(0 - theta);
```

Seleccionamos a que rueda debe aplicarse esa delta_pwm para corregir de forma adecuada la orientación.

```
// Actuar sobre los motores
if ((delta_PWM > 0 && PWM >= 0) || (delta_PWM < 0 && PWM < 0))
{
    PWM_A = aplicaSignal(motorA, PWM);
    PWM_B = aplicaSignal(motorB, PWM + delta_PWM);
}
else
{
    PWM_A = aplicaSignal(motorA, PWM - delta_PWM);
    PWM_B = aplicaSignal(motorB, PWM);
}

tiempoTranscurrido = millis() - tiempoAnterior;
tiempoAnterior = millis();
```

Por último, enviamos los datos de telemetría por bluetooth.

```
// Enviar datos de telemetría por Bluetooth
moduloBluetooth.DebugValuesBluetooth(tiempoTranscurrido, d2, d1, Ref_Distancia, MODO, PWM_A, PWM_B);
delay(60); // tiempo entre medidas (60 max según datasheet)
```

2.2.6. RESULTADOS EXPERIMENTALES

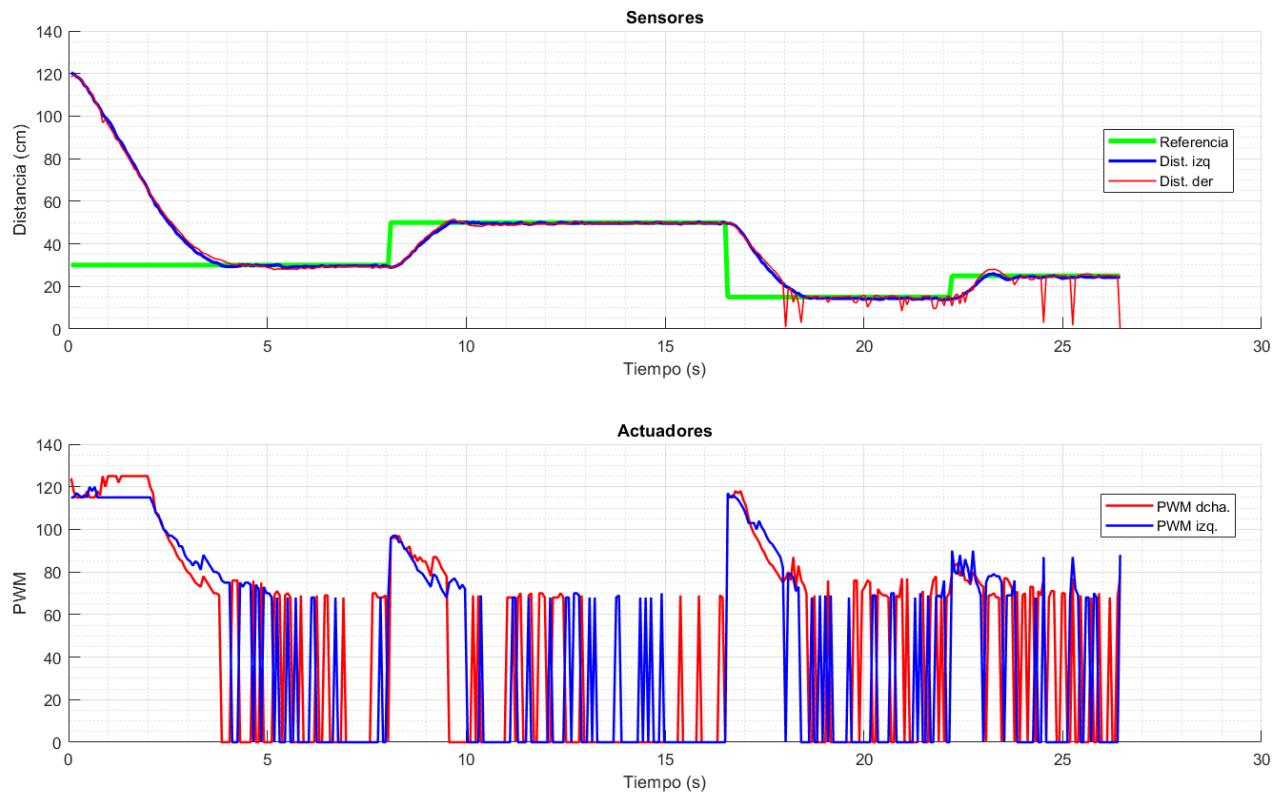
El experimento realizado es similar al del modo 1 con diferencia de que en este caso damos más saltos de referencia: Partimos con la referencia en 30 cm, cambiamos a 50 cm, pasamos a 15 cm y finalmente a 25 cm.

Centrándonos en las medidas tomadas por los sensores podemos ver como alcanzan la referencia fijada en cada momento con transiciones suaves. Consideramos que, en cuanto a tiempos, nuestro sistema cumple bastante bien las especificaciones que podríamos pedirle.

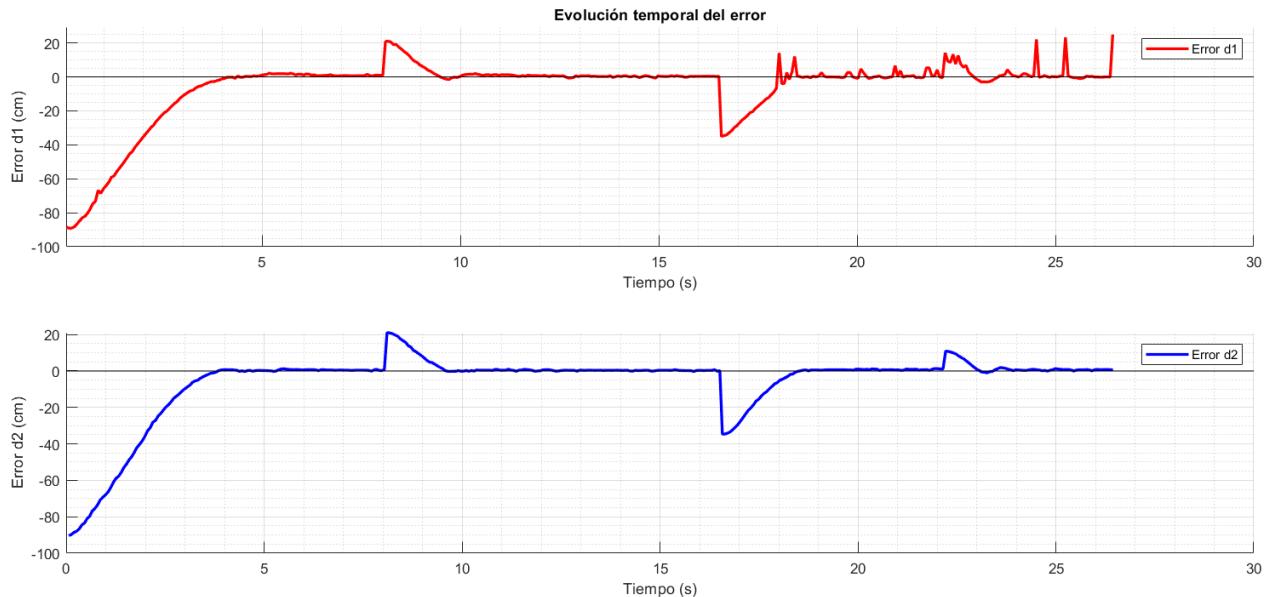
Si nos fijamos en las medidas tomadas por el sensor derecho, podemos ver como en distancias cortas existen picos inesperados. Esto es uno de los problemas que tienen este tipo de sensores de bajo coste. Sin embargo, son casos puntuales donde el sensor falla, pero la mayoría de las medidas se toman correctamente no afectando de forma perceptible a nuestro sistema.

En cuanto a la señal de control podemos ver con existen tramos temporales en los que la PWM asciende de forma notoria bajando paulatinamente. Estas zonas se corresponden con los cambios de referencia. Una vez que se alcanza aproximadamente la distancia deseada el robot corrige su orientación moviendo un poco la rueda necesaria. Estas zonas pueden apreciarse también en los resultados obtenidos.

A diferencia del modo anterior vemos como las pwm1 y pwm2 aplicadas a cada motor son distintas en todo momento gracias a la introducción del controlador de orientación explicado.



Por otro lado, analizamos la diferencia entre la referencia y las distancias medidas por ambos sensores. En caso en el que el coche se encuentre perpendicular a la pared dicha diferencia será nula en ambos casos. Es una forma de visualizar el error.



Tanto en este modo como en el resto, además de estos resultados gráficos en la telemetría en tiempo real desarrollada mandamos información extra como el ángulo que formaba en todo momento. Esta información extra de telemetría se proporciona en los videos de los distintos modos.

3. CONTROL CON SENSORES LATERALES

3.1. MODO 3

3.1.1. DESCRIPCIÓN

El objetivo de este modo será lograr un desplazamiento paralelo a la pared manteniendo la distancia inicial que tenga a la misma en todo momento.

3.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO

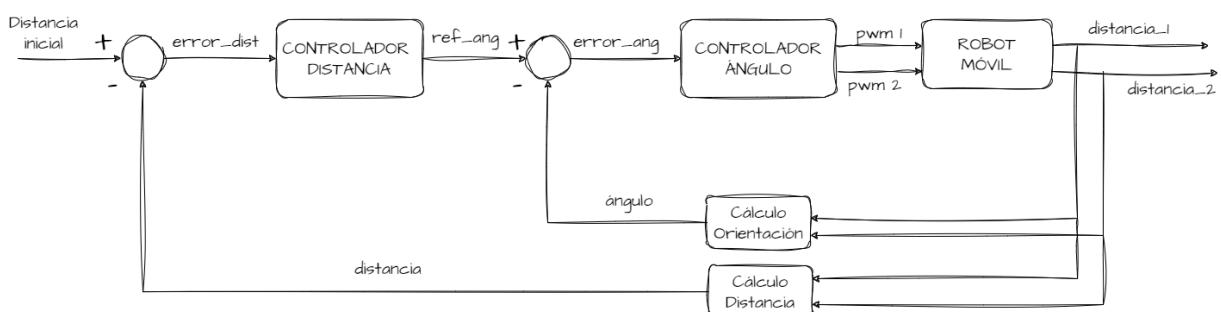
Los componentes usados son los mismos que en el control con los sensores dispuestos frontalmente. La diferencia por tanto se encuentra en la disposición física de los mismos en el coche, pasando de estar delante del mismo a situarse en uno de los laterales. El conexionado realizado no se ha visto modificado.

3.1.3. ESQUEMA DE CONTROL

La estrategia de control usada cuenta con un doble lazo de control. El lazo interno es el encargado de controlar la orientación del vehículo respecto a la pared para rechazar las perturbaciones del medio y conseguir mantenerse a la distancia inicial del experimento que será la referencia del sistema y que supone el lazo externo del control.

Es una estrategia basada en control master-slave donde el controlador de distancia es el maestro y el controlador de ángulo el esclavo. El controlador de distancia es un PI que a partir del error cometido entre la distancia inicial y la distancia en cada instante concreto es capaz de dar el ángulo de referencia con el que debe moverse el robot para corregir dicho error. La entrada del controlador de ángulo (también un PI) es precisamente el error entre la referencia fijada por el controlador de distancia y el ángulo actual. Tanto ángulo como distancia pueden ser hallados a partir de las mediciones de ambos sensores al igual que se hacía en el modo 2.

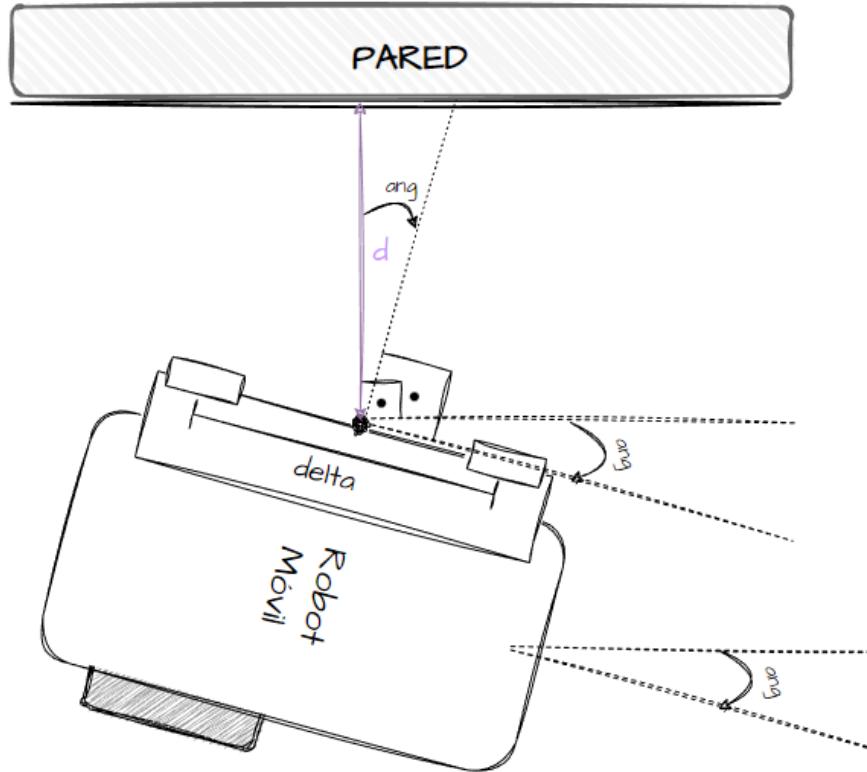
MODO 3



En este caso, la pwm que irá a cada motor se calcula como la suma del punto de equilibrio, una cierta cantidad que será la velocidad fija que llevará el coche en el movimiento y una delta_pwm que será la que el controlador de ángulo proporcione a una rueda o a otra para conseguir la orientación deseada. De nuevo, por simplicidad en el esquema no incluimos el punto de operación ni el offset de velocidad fija descrito.

3.1.4. CÁLCULOS ANALÍTICOS

Los cálculos para conseguir el ángulo y distancia son los mismos a los desarrollados en el modo 2, ya que el ángulo que se pretende controlar es el mismo:



3.1.5. IMPLEMENTACIÓN SOFTWARE

El cálculo de theta y distancia, como hemos dicho, se hace de forma igual que en el modo 2, para no repetir las mismas partes de código mostramos las novedades. Vemos como se implementa el control en cascada donde en primer lugar obtenemos la referencia de ángulo a partir del error en distancia y posteriormente la señal de control incremental en pwm a partir del error en ángulo.

```
// Calcular señal de control
static const int PWM = 25;
referencia_ang = ControlDistancia.calcularSalida(Ref_Distancia-distancia);
int delta_PWM = ControlOrientacion.calcularSalida(referencia_ang - theta);

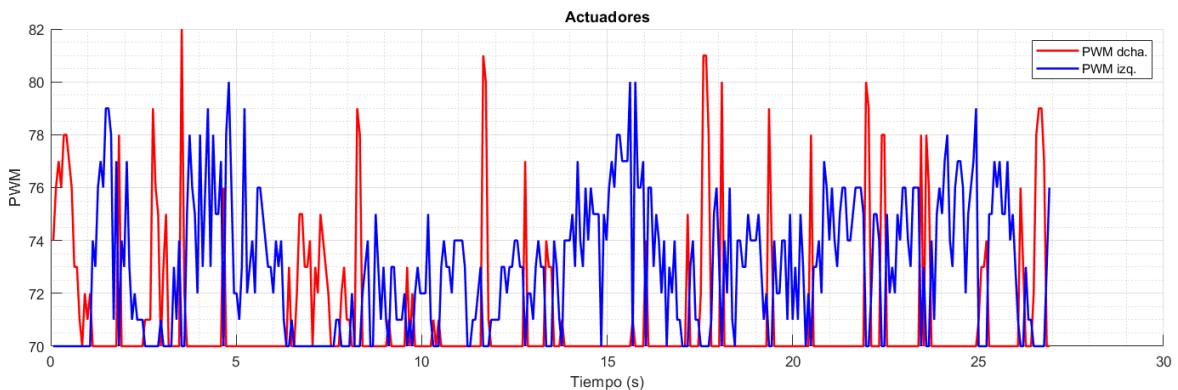
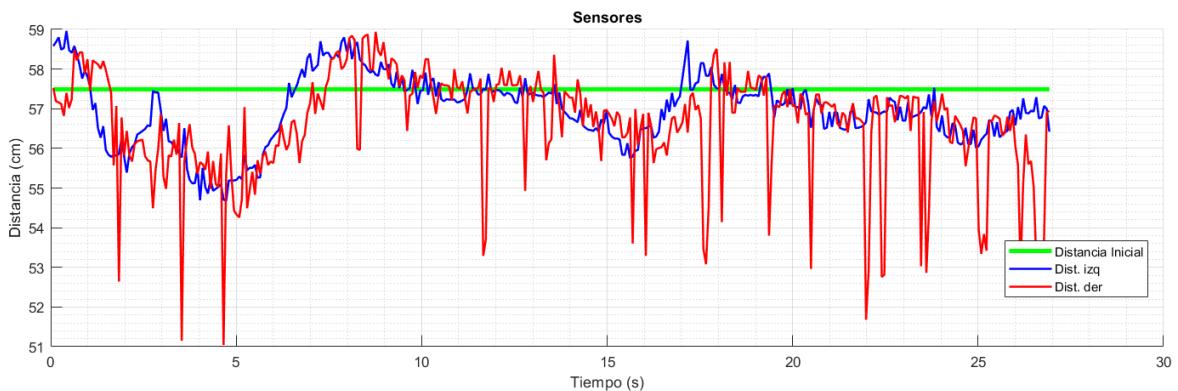
// Actuar sobre los motores
if ((delta_PWM > 0 && PWM > 0) || (delta_PWM < 0 && PWM < 0))
{
    PWM_A = aplicaSignal(motorA, PWM);
    PWM_B = aplicaSignal(motorB, PWM + delta_PWM);
}
else
{
    PWM_A = aplicaSignal(motorA, PWM - delta_PWM);
    PWM_B = aplicaSignal(motorB, PWM);
}
```

La referencia de distancia, como hemos comentado, será la distancia inicial a la cual se coloca el robot. Para ello, en el inicio del programa tomamos varias medidas y nos quedamos con la mediana de ellas. De esta forma eliminamos posibles medidas erróneas tomadas inicialmente por fallo en los sensores.

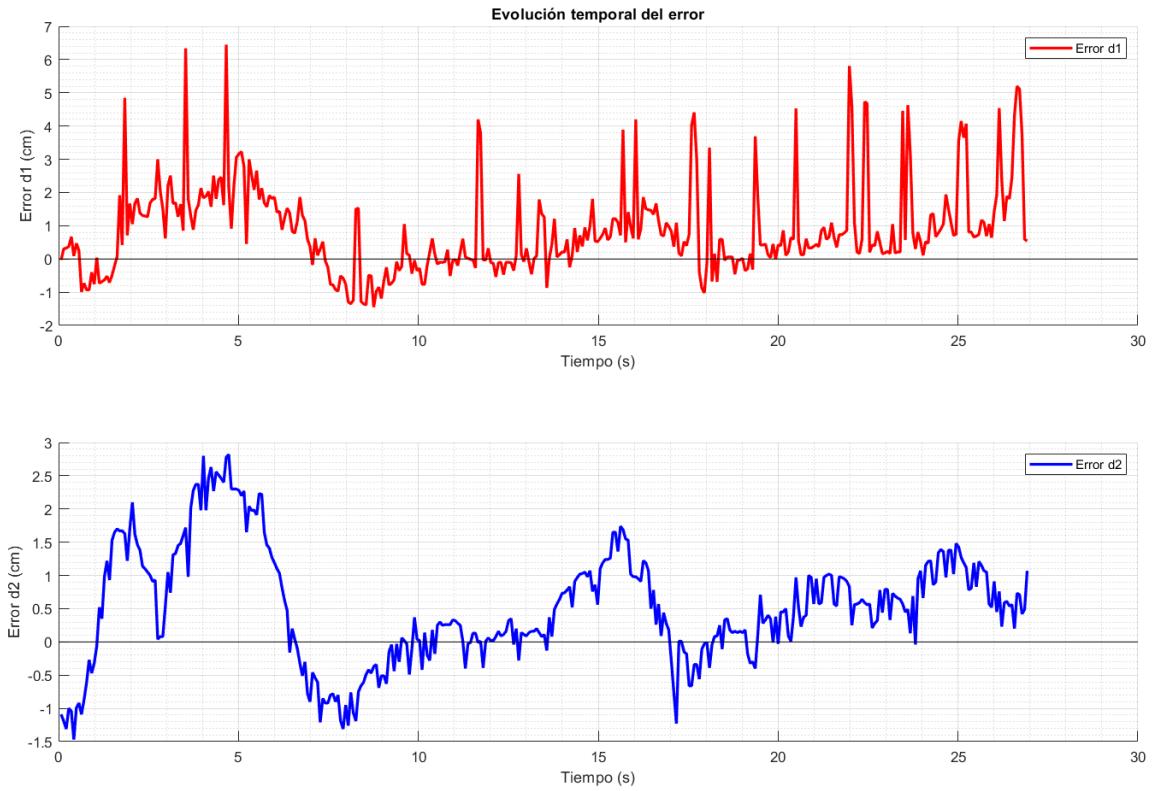
3.1.6. RESULTADOS EXPERIMENTALES

El experimento realizado consiste en la colocación del coche a una determinada distancia de la pared, dejando que se mueva de forma paralela a la misma.

Vemos en primer lugar como las distancias se mantienen en torno a la medida inicial tomada. Respecto a la señal de control en PWM vemos como mínimamente es siempre de 70 manteniendo una cierta velocidad constante sobre la cual se incrementa una rueda u otra en función del signo del error en ángulo.



En cuanto a la evolución temporal del error, puede verse como en este caso, el error no se hace nulo puesto que el sistema está sometido a constantes perturbaciones que debe rechazar como son las imperfecciones del suelo o el ángulo de caída de las ruedas lo cual produce desviaciones en su movimiento continuo. Sin embargo, el error se mantiene dentro de un umbral controlado, de varios centímetros. Teniendo en cuenta las características del robot, podemos decir que dicho margen de error es admisible para la aplicación que se pretendía conseguir.



3.2. MODO 4

3.2.1. DESCRIPCIÓN

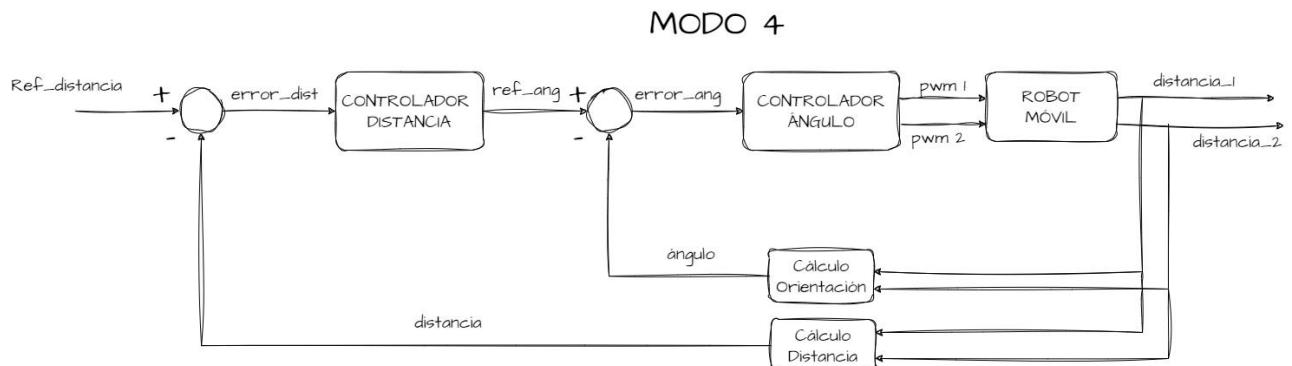
En esta ocasión la referencia de distancia será variable y el objetivo será que el robot se posicione a la distancia indicada de la pared a la vez que se desplaza paralelamente a la misma al igual que hacía en el modo anterior. En los permanentes, por tanto, el comportamiento que se pretende obtener será idéntico al modo 3, la diferencia radica en la habilidad para realizar las transiciones de acercamiento y alejamiento respecto al plano de la pared.

3.2.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO

No se han añadido componentes extras. El montaje sigue siendo el mismo.

3.2.3. ESQUEMA DE CONTROL

La única diferencia respecto al explicado en el modo 3 está en que la referencia de distancia en este caso es variable en lugar de ser constante:



3.2.4. CÁLCULOS ANALÍTICOS

Los mismos que en modo 2 y modo 3.

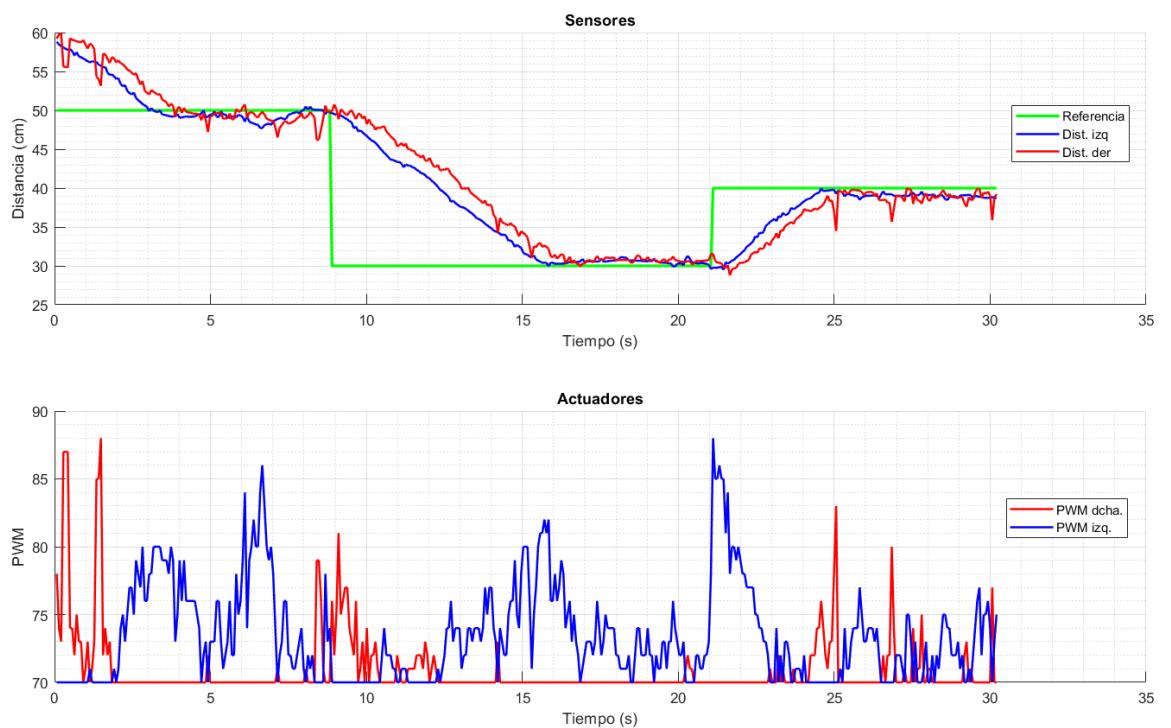
3.2.5. IMPLEMENTACIÓN SOFTWARE

La única novedad es que ya no se calcula la referencia como la distancia inicial, sino que el valor inicial está fijado, pero será cambiado durante la ejecución del programa.

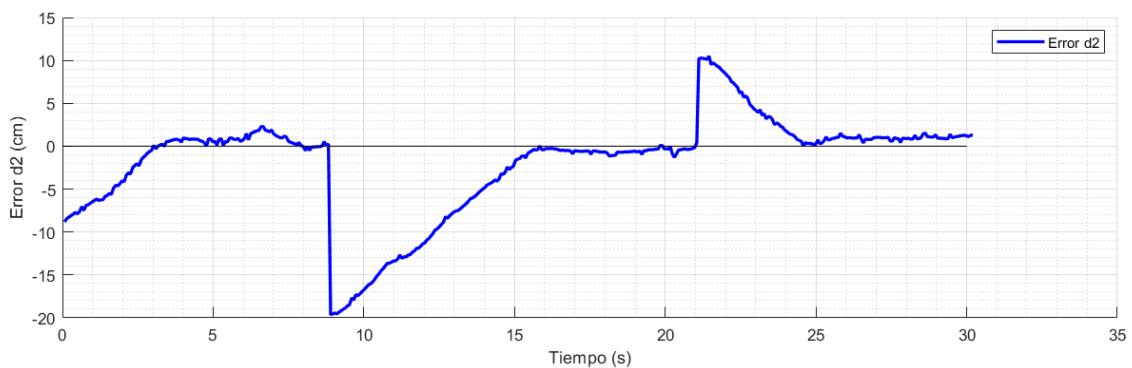
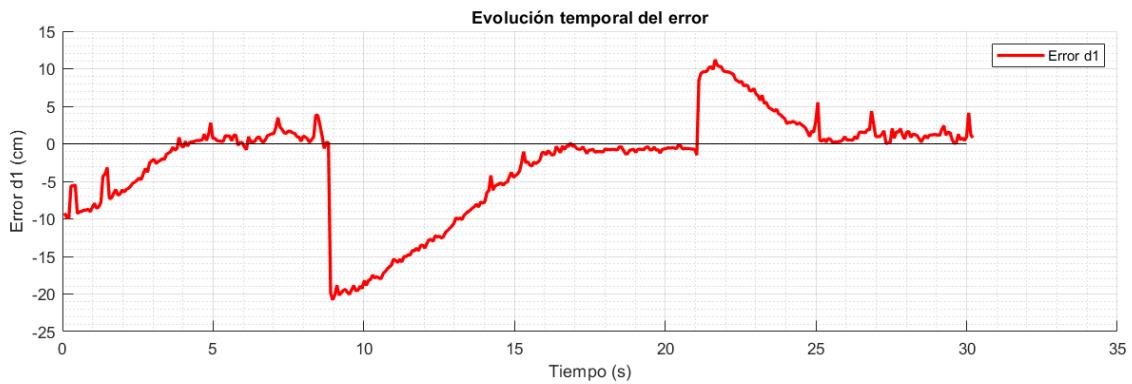
3.2.6. RESULTADOS EXPERIMENTALES

La comprobación experimental consiste en situar inicialmente el robot a 50 cm de la pared, indicarle que se coloque a 30 cm y, una vez alcanzado, que se posicione a 40 cm.

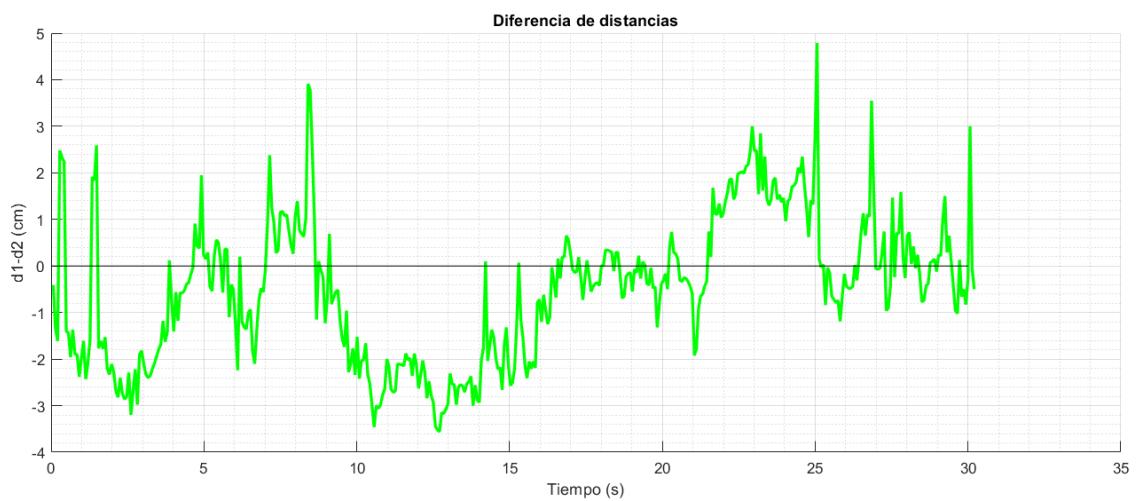
Vemos que en régimen permanente se comporta como preveíamos, igual que en modo 3, y las transiciones tanto de alejamiento como de acercamiento a la pared se hacen de forma progresiva en tiempos razonables.



Los errores obtenidos. Recordamos que estos errores están calculados como la diferencia entre la referencia y las medidas de los sensores. Esto nos da una idea de la orientación respecto a la pared, cuando ambos errores son cero el coche estaría situado perfectamente paralelo a la misma.



También puede representarse la diferencia entre las distancias medidas como magnitud del error que se comete.



4. CONTROL SENSORES VELOCIDAD

4.1. MODO 5

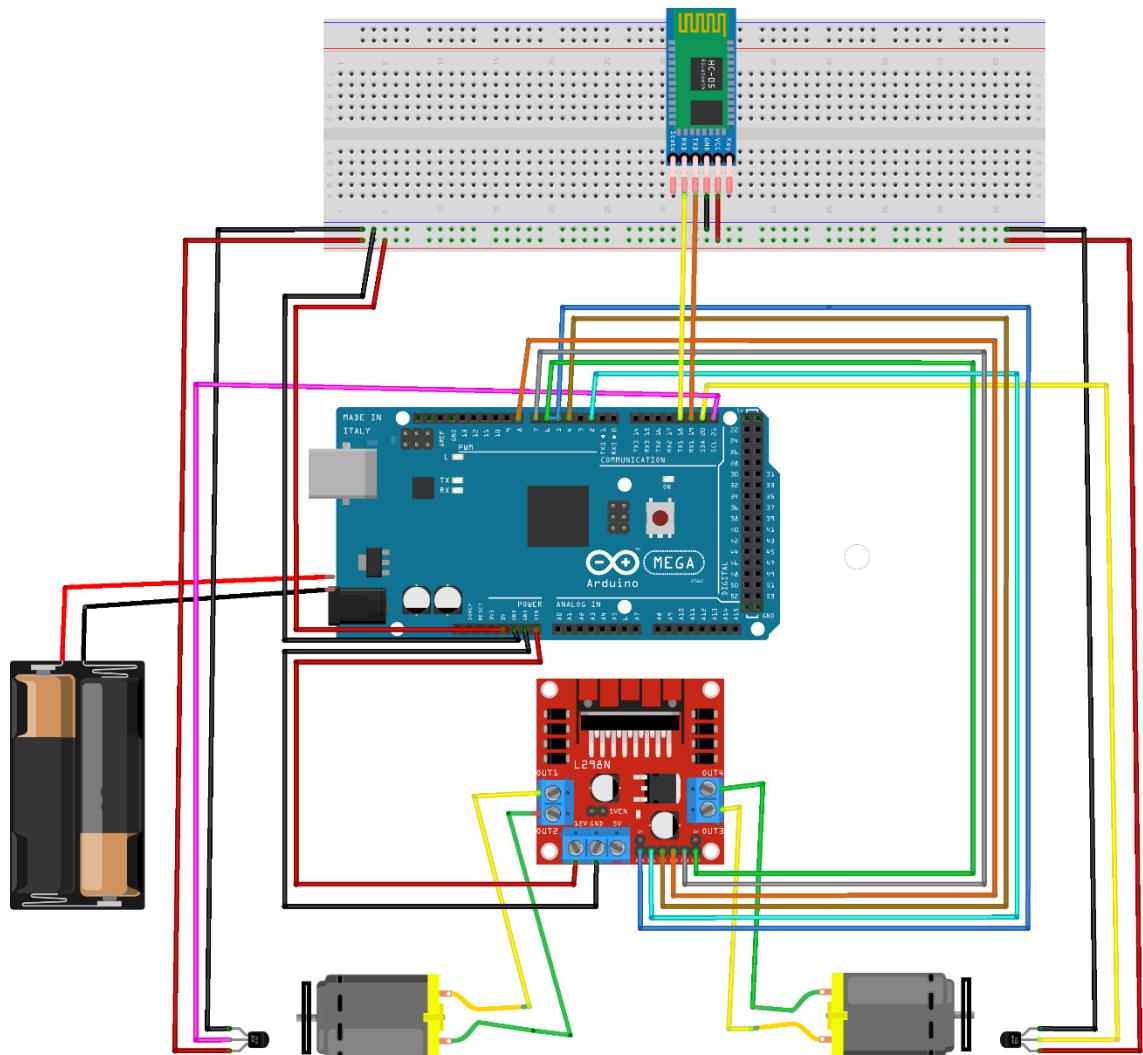
4.1.1. DESCRIPCIÓN

El fin de este modo será el de controlar la velocidad angular de cada rueda. Lo haremos en revoluciones por minuto. Para ello se instalarán sensores que permitan medir el giro de las ruedas y a partir de esto hacer una estimación de la velocidad.

4.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO

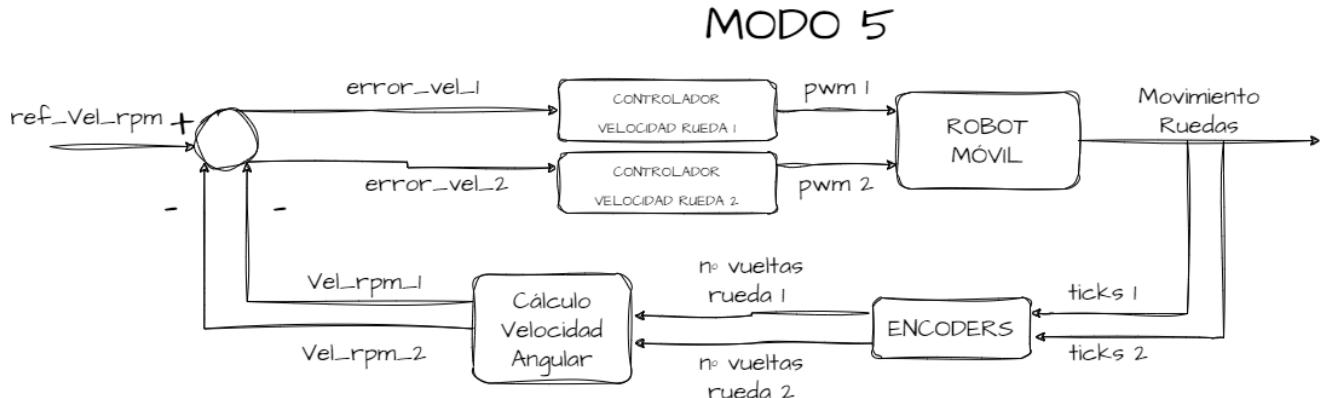
Tenemos la necesidad de incluir sensores para realizar una medida de la velocidad. Usaremos para ello encoders de tipo magnético.

Mediante sensores magnéticos (de efecto Hall) se detectan el movimiento de un conjunto de polos magnéticos dispuestos uno a continuación del otro en un disco (de 8 polos) que se acopla al eje de giro de los motores. Mediante este tipo de encoders somos capaces de transformar las variaciones en el campo magnético en señales digitales que contienen información del número de vueltas que experimenta cada motor y por tanto las ruedas.



4.1.3. ESQUEMA DE CONTROL

Tenemos dos controladores encargados de calcular la señal de control PWM que recibirá cada motor en función del error cometido entre la referencia de velocidad (igual para ambas ruedas) y las velocidades de cada rueda. Conseguimos estimar dichas velocidades angulares de las ruedas a partir de la medida de los ticks por parte de los encoders, conociendo el tiempo de muestreo, la relación de transformación y el número de polos usados.



Los controladores de bajo nivel implementados son PIDs sin términos derivativos (constante derivativa nula), que cuentan con niveles máximos y mínimos de señal de control (saturaciones) y no acumulan el error integral cuando se produce dicha saturación (incluyen algoritmo anti-windup).

4.1.4. CÁLCULOS ANALÍTICOS

Los encoders se encuentran conectados a pines de entrada de interrupción de la placa. Configurando dichos pines de una forma u otra (flanco de subida/bajada o ambos) somos capaces de tener 4 o 8 pulsos por vuelta (cada vez que se dé la interrupción entramos en una rutina asociada, las cuales llamamos como ISR).

En nuestro caso, para tratar de maximizar la resolución instalamos los encoders en el eje del motor y configuramos los pines para tener 8 pulsos por vuelta. Como la relación de transformación es de 48 vueltas de motor equivalen a 1 vuelta de la rueda y cada vuelta del motor son 8 pulsos o ticks contabilizados en la rutina de interrupción asociada a cada encoder, el número de vueltas que da cada rueda.

Conocido el tiempo transcurrido entre cálculo (tiempo en el cual se acumulan los ticks) podemos obtener las velocidades angulares de las ruedas en rpm:

$$\text{nº vueltas rueda} = \frac{\text{nº ticks}}{\text{reductora} \cdot \text{nº polos}}$$

$$\text{Velocidad angular (rpm)} = \frac{\text{Nº vueltas rueda}}{\text{tiempo muestreo (ms)}} \cdot \frac{1000 \text{ (ms)}}{1 \text{ s}} \cdot \frac{60 \text{ s}}{1 \text{ min}}$$

4.1.5. IMPLEMENTACIÓN SOFTWARE

El código principal tiene en cuenta si se cambia el signo de la referencia lo cual significa que tendría que pasar de moverse con cierta velocidad hacia delante a pasar a un movimiento marcha atrás. Si se da el caso la referencia pasa a valer 0 previamente ya que primero debemos reducir la velocidad a valor nulo para después alternar a referencia negativa para conseguir un correcto funcionamiento.

```
// Comprobamos si el signo de la referencia ha cambiado
comprueba_cambio_sentido(RefVelA, &RefVelA_aplicada, motorA);
comprueba_cambio_sentido(RefVelB, &RefVelB_aplicada, motorB);

void comprueba_cambio_sentido(float RefVel, float *RefVel_aplicada, movimientoCoche &motor)
{
    if (sign(RefVel) != sign(*RefVel_aplicada))
    {
        if (motor.sentidoGiro == 0)
        {
            *RefVel_aplicada = RefVel;
        }
        else
        {
            *RefVel_aplicada = 0;
        }
    }
    else
    {
        *RefVel_aplicada = RefVel;
    }
}
```

Calculamos la PWM a aplicar a ambas ruedas teniendo en cuenta un punto de equilibrio adecuado cercano al necesario para comenzar a moverse. En función del signo de la señal de control indicaremos un sentido de giro u otro al motor.

```
PWM_A = ControlVelocidadA.calcularSalida(RefVelA_aplicada - wA);
if (RefVelA_aplicada > 0)
{
    PWM_A += puntoEquilibrio;
    motorA.sentidoGiro = 1;
}
else if (RefVelA_aplicada < 0)
{
    PWM_A -= puntoEquilibrio;
    motorA.sentidoGiro = -1;
}

PWM_B = ControlVelocidadB.calcularSalida(RefVelB_aplicada - wB);
if (RefVelB_aplicada > 0)
{
    PWM_B += puntoEquilibrio;
    motorB.sentidoGiro = 1;
}
else if (RefVelB_aplicada < 0)
{
    PWM_B -= puntoEquilibrio;
    motorB.sentidoGiro = -1;
}
```

Aplicamos las señales de control y mandamos los datos por bluetooth para la representación de la telemetría:

```
aplicaSignal(motorA, PWM_A, RefVelA_aplicada);
aplicaSignal(motorB, PWM_B, RefVelB_aplicada);
moduloBluetooth.DebugValuesBluetooth(ControlVelocidadA.tiempoTranscurrido, wA, wB, RefVelA_aplicada, MODO, PWM_A, PWM_B);
```

Vemos como el incremento de ticks está asociado a una interrupción hardware:

```

void ISR_incrTicksA()
{
    ticksA++;
}

void ISR_incrTicksB()
{
    ticksB++;
}

```

El cálculo de las velocidades angulares se hace mediante interrupciones de un timer configurado a 50 ms. Por tanto, en función de los ticks acumulados podemos hallar fácilmente la velocidad de cada rueda en rpm, teniendo en cuenta la reductora y el número de polos como hemos explicado anteriormente:

```

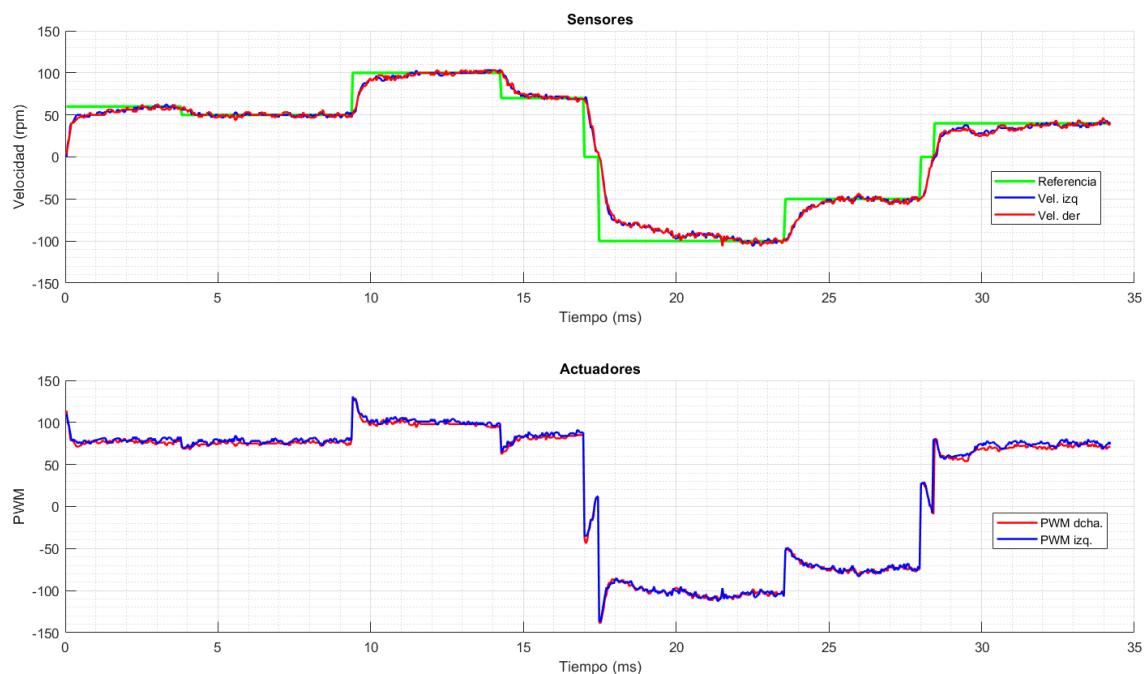
void ISR_calculateAngSpeed()
{
    if (ticksA == 0) // si la rueda A está parada
    {
        motorA.sentidoGiro = 0;
        wA = 0;
    }
    else
    {
        if (motorA.sentidoGiro == 1)
        {
            wA = ((float)ticksA * 1000.0 * 60.0) / ((float)npolos * (float)reductora * refreshRateEncoders);
        }
        else
        {
            wA = -((float)ticksA * 1000.0 * 60.0) / ((float)npolos * (float)reductora * refreshRateEncoders);
        }
        ticksA = 0;
    }
}

```

De igual manera se hace para la rueda B.

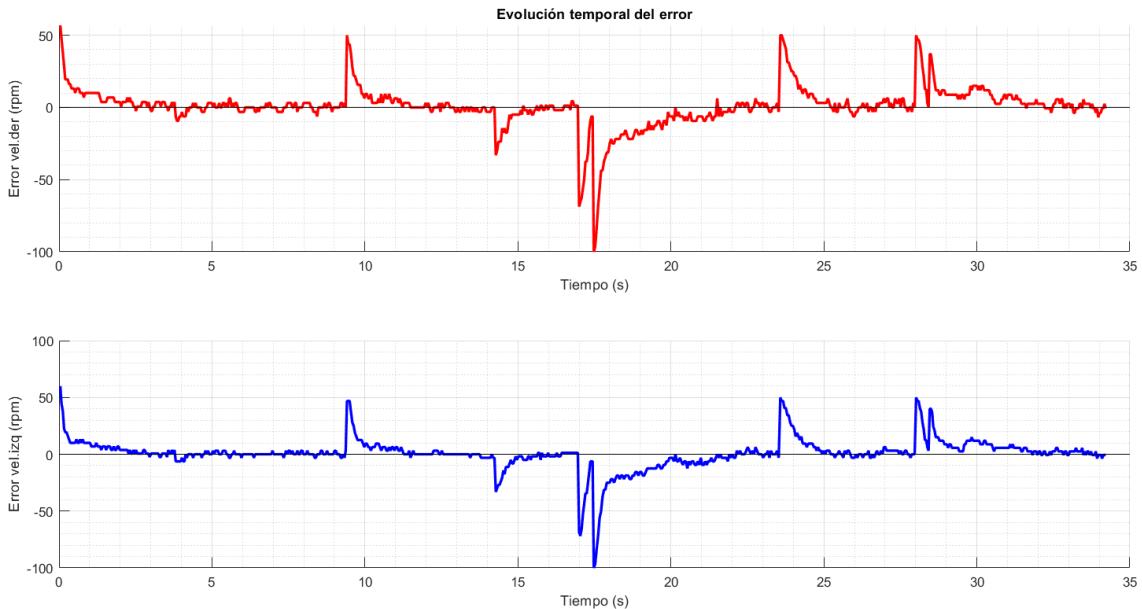
4.1.6. RESULTADOS EXPERIMENTALES

El experimento realizado consiste en cambiar la referencia de velocidad angular de ambas ruedas tanto escalones positivos como negativos y ver como estas cambian de velocidad siguiendo la referencia fijada, lo cual se traduce en cambios de velocidad de desplazamiento total del vehículo.



En la gráfica anterior se podía ver como para lograr cada velocidad de referencia la PWM aplicada en las ruedas experimenta un transitorio adecuado para en el permanente adoptar un valor idealmente constante correspondiente al de la característica estática del sistema.

El error cometido en velocidad en cada rueda queda recogido a continuación:



4.2. MODO 6

4.2.1. DESCRIPCIÓN

Partiendo del modo anterior, se desea hacer que el robot se mueva en línea recta a partir del control en velocidad de cada rueda.

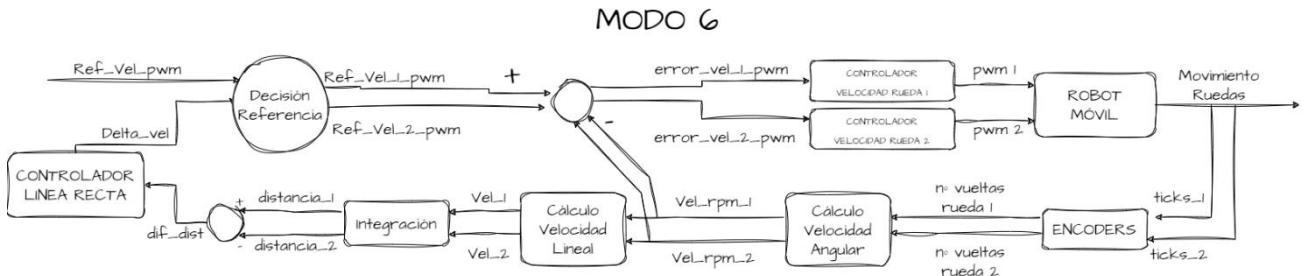
4.2.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO

Los componentes usados son los mismos que en el modo 5, no cambiando ni el conexionado ni el montaje en este caso.

4.2.3. ESQUEMA DE CONTROL

La diferencia con respecto al modo 5 a nivel de estructura de control está en la inclusión de un nuevo módulo encargado de corregir las variaciones existentes en las distancias diferenciales recorridas por cada rueda para corregir la velocidad de referencia en cada rueda de modo que se pueda describir una linea recta.

Para ello transformamos la velocidad angular de cada rueda en una velocidad lineal la cual integrando podemos hallar el incremento de recorrida por cada rueda en cada tiempo de muestreo. A partir de la diferencia de distancias recorridas, el controlador de linea recta me dará una delta_{vel} (velocidad extra) que sumar a la referencia de velocidad de una rueda u otra para conseguir corregir es desviación que se tiene.



Como vemos, la parte novedosa por tanto es la de la izquierda del esquema. El controlador de linea recta será un PID con saturaciones y anti-windup como el resto de los controladores que hemos usado. El controlador de bajo nivel sigue siendo el mismo.

4.2.4. CÁLCULOS ANALÍTICOS

Es sencillo hallar la distancia incremental recorrida en cada periodo de muestreo a partir de la velocidad angular de las ruedas y el tiempo transcurrido (el marcado por el timer asociado al cálculo de dicha velocidad angular):

$$\text{Velocidad lineal } \left(\frac{m}{s} \right) = \text{Velocidad angular} \cdot \text{Radio ruedas}$$

$$\text{Distancia recorrida (m)} = \text{Velocidad lineal} \cdot \text{tiempo transcurrido}$$

4.2.5. IMPLEMENTACIÓN SOFTWARE

La estructura del código implementado es muy similar a la del modo 5. Lo más destacable como novedad sería el calculo de la velocidad extra que sumar a una de las referencias para conseguir corregir la diferencia de distancias recorridas.

```

deltaVel = LíneaRecta.calcularSalida(0-(distanciaA-distanciaB));

if(RefVelA > 0){
if(deltaVel<0)
{
    RefVelB_aplicada -= deltaVel;
}
else
{
    RefVelA_aplicada += deltaVel;
}
} else{
    if(deltaVel<0)
    {
        RefVelB_aplicada += deltaVel;
    }
    else
    {
        RefVelA_aplicada -= deltaVel;
    }
}

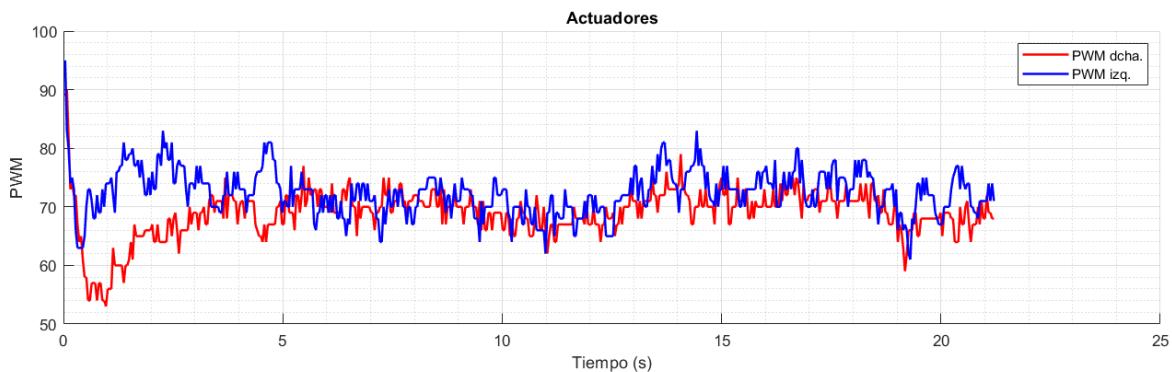
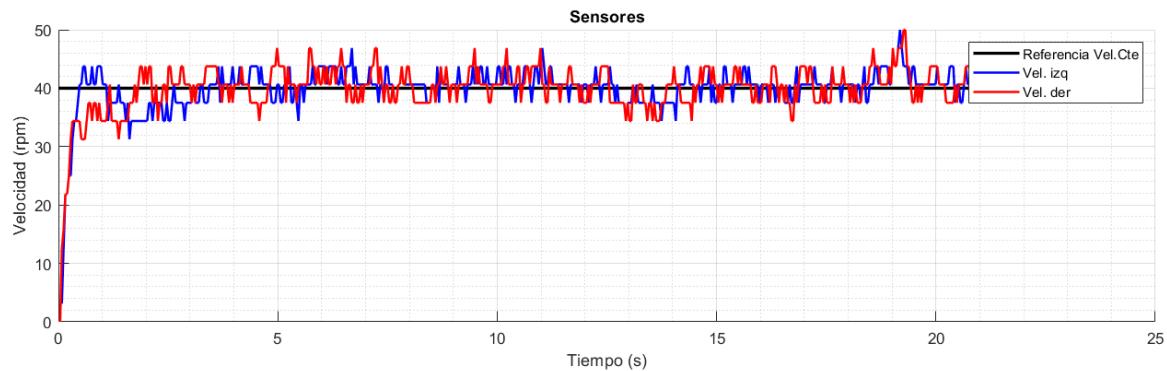
```

En función del signo de la referencia de velocidad y de deltaVel determinamos las nuevas referencias como se muestra.

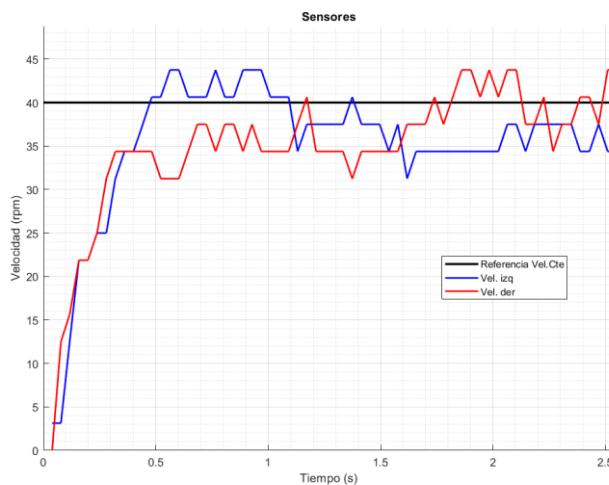
4.2.6. RESULTADOS EXPERIMENTALES

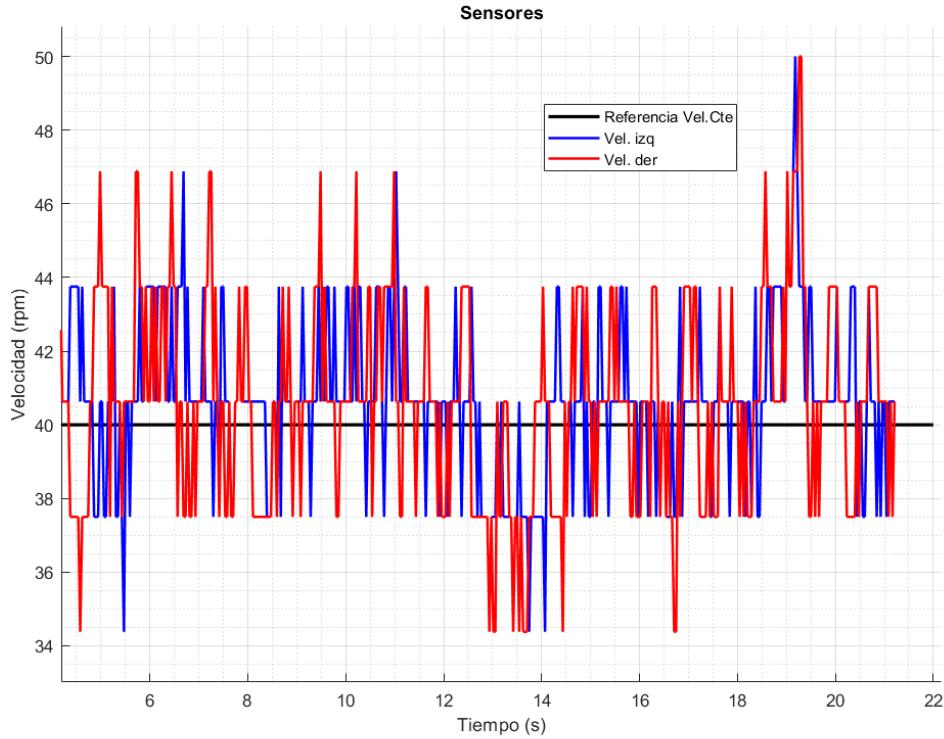
El experimento cuyos datos se representan consiste en la colocación del robot en el suelo en reposo, pedirle una referencia de velocidad de 40 rpm y ver como este consigue alcanzar dicha referencia siguiendo una linea recta gracias a la referencia variable en la que interviene el pequeño incremento de velocidad entorno a la referencia constante.

Podemos ver como ambas velocidades de las ruedas durante el experimento se encuentran en torno a la referencia de velocidad angular fija.

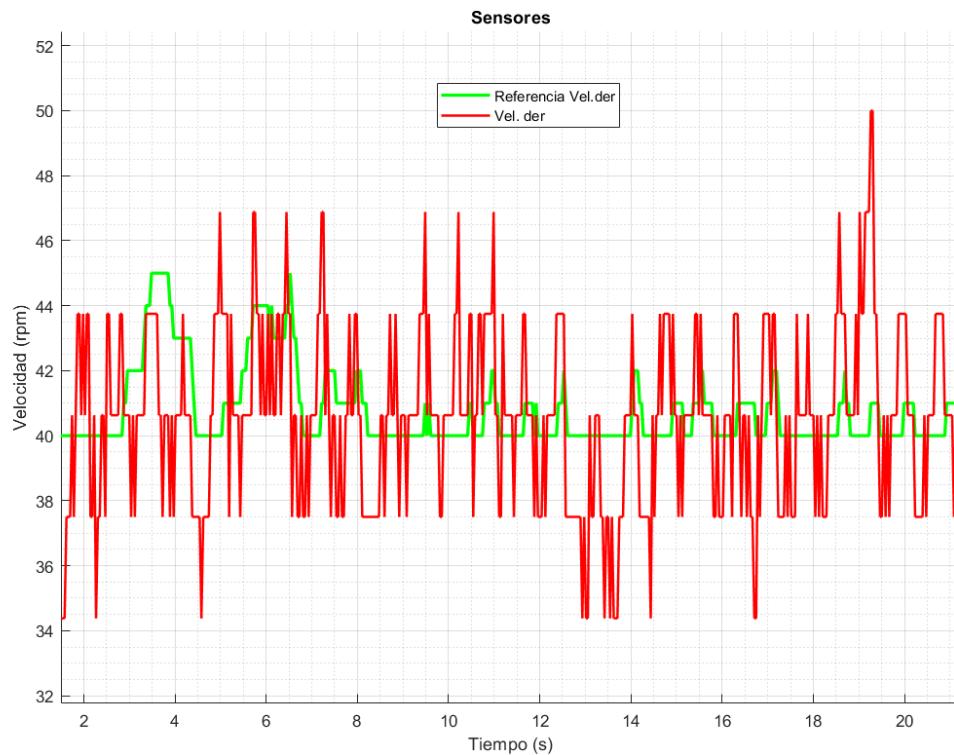


Hemos dividido las gráficas de velocidades anteriores en transitorio inicial y permanente para ver mejor cada parte respecto a la representada anteriormente.





También representamos la referencia variable de una de las ruedas para demostrar que gracias a dicha referencia que tiene en cuenta el incremento de velocidad se consigue navegar en linea recta. Las velocidades realmente tienden a ser dichas referencias cambiantes y no la referencia constante representada anteriormente:



5. ESTIMACIÓN DE POSICIÓN

5.1. MODO 7

5.1.1. DESCRIPCIÓN

Introducimos el modelo cinemático conocido de la configuración diferencial, que nos permitirá la obtención de una estimación de la posición basada en odometría.

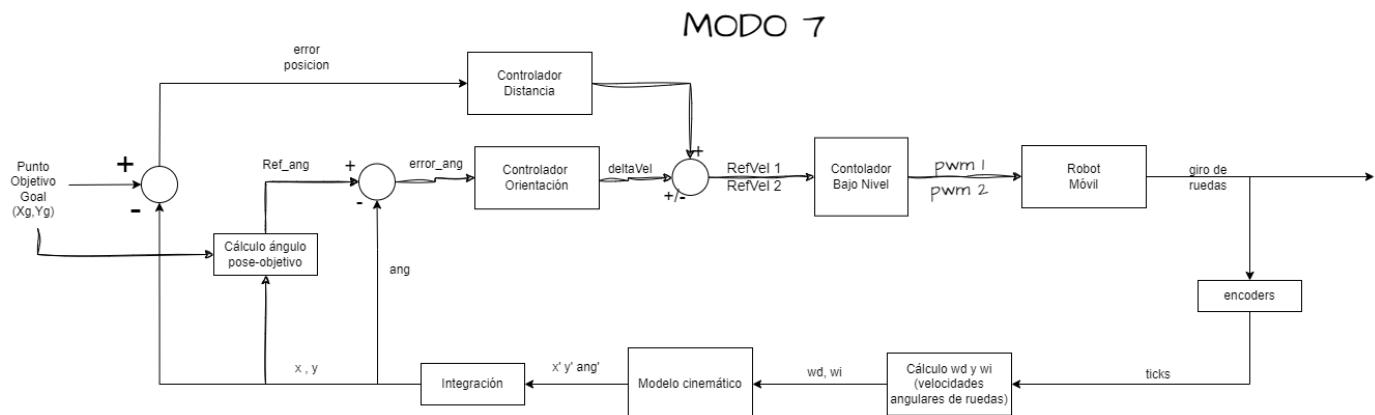
El objetivo será el movimiento del robot a posiciones deseadas respecto a un sistema de referencia global el cual tomaremos como el punto inicial donde posicionamos el robot al ejecutar el programa. El sistema de ejes locales estará posicionado en el punto central sitiado entre ambas ruedas. Al hablar de la pose del robot nos estaremos refiriendo a dicho punto.

5.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO

Para la realización de este modo no se hace uso de nuevos componentes ni se altera el conexionado que teníamos.

5.1.3. ESQUEMA DE CONTROL

La metodología de control implementada se recoge en el esquema.



En resumen, controlaremos la velocidad de ambas ruedas para lograr describir las trayectorias rectas y el giro del coche para reorientarnos hacia los sucesivos puntos objetivos, los cuales serán, en esta aplicación particular, los vértices de un cuadrado. Será el controlador de bajo nivel el encargado de traducir dicha velocidad deseada para lograr los movimientos en PWM. La velocidad requerida es suma de dos términos:

El primero de ellos aportado por un controlador de distancia que en función del error entre la pose actual y el punto objetivo en cada caso es capaz de determinar una cierta velocidad de base a la que debemos movernos.

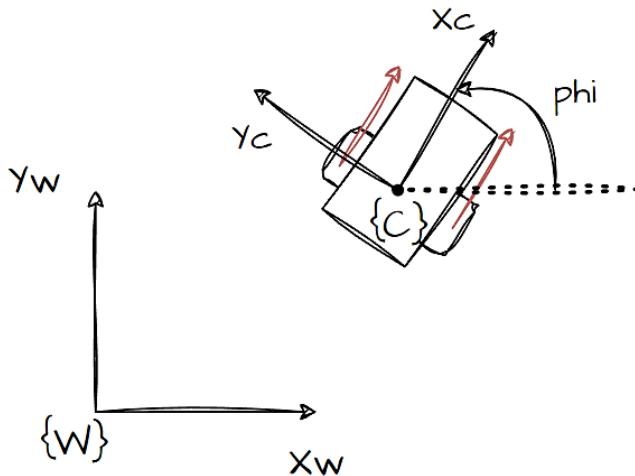
El segundo término proviene de la salida del controlador de orientación, que a partir del error cometido en el ángulo (diferencia entre la referencia angular a la cual debe dirigirse y ángulo actual de la pose) calcula un cierto incremento de velocidad el cual deberá tenerse en cuenta respecto al primer término de base para corregir la orientación.

Es importante fijarse en el detalle de que cuando deba darse el giro, el controlador de velocidad no aportará nada a la velocidad deseada debido a que el error en distancia cometido será nulo.

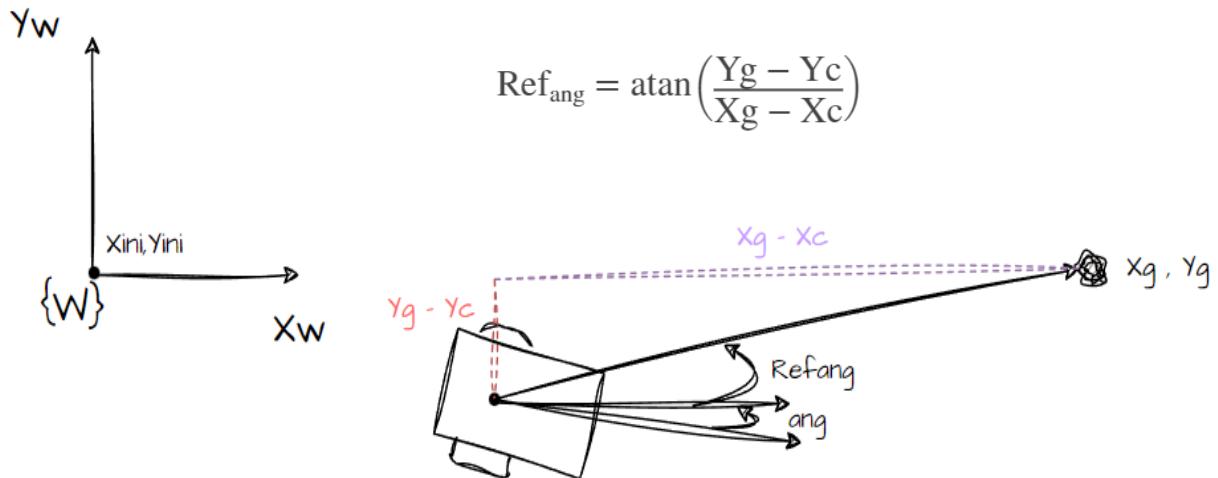
Los controladores implementados son de tipo PID, nuevos objetos de la librería que venimos usando parametrizados y ajustados correctamente para su fin particular en cada caso (orientación, giro y avance en linea recta).

5.1.4. CÁLCULOS ANALÍTICOS

Esquema sistemas de referencia local y global:



La referencia angular será el ángulo que se forma entre la posición actual (X, Y) y la objetivo. Podemos hallarla de la siguiente forma:



5.1.5. IMPLEMENTACIÓN SOFTWARE

Se exponen las principales partes del programa desarrollado.

Los vértices del cuadrado serán considerados como puntos objetivos que debemos alcanzar.

Implementamos una maquina de estados con tres modos: inicialización, giro y avance en línea recta.

La inicialización consiste en actualizar el punto objetivo y hacer los cálculos de la referencia de ángulo inicial para dar el giro.

```
switch (estadoM)
{
    case 0: // calculoRefs (estado inestable)
        indicePunto++;
        if (indicePunto > 3)
        {
            indicePunto = 0;
        }
        if(indicePunto!=0)
        {
            deltaX = posicionObjetivo[indicePunto].x-posicionObjetivo[indicePunto-1].x;
            deltaY = posicionObjetivo[indicePunto].y-posicionObjetivo[indicePunto-1].y;
        }
        else
        {
            deltaX = posicionObjetivo[indicePunto].x-posicionObjetivo[3].x;
            deltaY = posicionObjetivo[indicePunto].y-posicionObjetivo[3].y;
        }

        RefAng = (float)atan2(deltaY, deltaX);
        if (RefAng == -PI) // quitar -PI del intervalo para evitar ambiguedades
        {
            RefAng += 2 * PI;
        }
        RefAng *= 180.0 / PI;
        estadoM = 1;
        break;
}
```

En el estado de giro se usa el controlador de orientación para hallar la deltaVel que aplicar a cada rueda. Para conseguir el giro respecto al eje centrado en el punto medio de las ruedas se suma la mitad de la deltaVel calculada a una rueda con sentido positivo y la otra mitad a la hora con sentido negativo.

```
case 1: // Giro
    errorAng = calculaErrorAng(RefAng,poseRobot.phi);
    deltaVel = Orientacion.calcularSalida(errorAng);
    RefVelA = deltaVel / 2;
    RefVelB = -deltaVel / 2;
    if (abs(errorAng) < 5 && motorA.sentidoGiro==0 && motorB.sentidoGiro==0)
    {
        estadoM = 2;

        Orientacion.resetPID();
        Distancia.resetPID();
    }
    break;
```

En el último estado correspondiente al avance en línea recta se usan tanto controlador de orientación como de distancia. La referencia de ángulo se actualiza constantemente según lo expuesto anteriormente consiguiendo de esta forma seguir una trayectoria recta.

```

case 2: // lineaRecta
    deltaX = posicionObjetivo[indicePunto].x - poseRobot.x;
    deltaY = posicionObjetivo[indicePunto].y - poseRobot.y;
    moduloDistancia = sqrt(pow(poseRobot.x - posicionObjetivo[indicePunto].x, 2)
                           + pow(poseRobot.y - posicionObjetivo[indicePunto].y, 2));
    if (moduloDistancia > errorAdmitidoDistance)
    {
        RefAng = (float)atan2(deltaY, deltaX);
        if (RefAng == -PI) // quitar -PI del intervalo para evitar ambiguedades
            RefAng += 2 * PI;
        RefAng *= 180.0 / PI;
    }

    RefVelA = Distancia.calcularSalida(0-moduloDistancia);
    RefVelB = RefVelA;
    errorAng = calculaErrorAng(RefAng,poseRobot.phi);
    deltaVel = Orientacion.calcularSalida(errorAng);
    RefVelA += deltaVel / 2;
    RefVelB += -deltaVel / 2;

    if (moduloDistancia < errorAdmitidoDistance )
    {
        estadoM = 0;
        Orientacion.resetPID();
    }
    break;
}

```

Común a todos los estados se encuentra el controlador de bajo nivel que a partir de la referencia completa de velocidad en ambas ruedas (tras tener en cuenta una serie de consideraciones para el correcto movimiento tales como sentidos de giro, signos de referencias, offset aplicados, ...) genera la PWM necesaria para los motores. Tras esto aplicamos dichas señales de actuación a los mismos.

```

// Comprobamos si el signo de la referencia ha cambiado
comprueba_cambio_sentido(RefVelA, &RefVelA_aplicada, motorA);
comprueba_cambio_sentido(RefVelB, &RefVelB_aplicada, motorB);

if(RefVelA_aplicada==0 || RefVelB_aplicada==0)
{
    RefVelA_aplicada=0;
    RefVelB_aplicada=0;
}

PWM_A = ControlVelocidadA.calcularSalida(RefVelA_aplicada - wA);
if (RefVelA_aplicada > 0)
{
    PWM_A += puntoEquilibrio;
    motorA.sentidoGiro = 1;
}
else if (RefVelA_aplicada < 0)
{
    PWM_A -= puntoEquilibrio;
    motorA.sentidoGiro = -1;
}

PWM_B = ControlVelocidadB.calcularSalida(RefVelB_aplicada - wB);
if (RefVelB_aplicada > 0)
{
    PWM_B += puntoEquilibrio;
    motorB.sentidoGiro = 1;
}
else if (RefVelB_aplicada < 0)
{
    PWM_B -= puntoEquilibrio;
    motorB.sentidoGiro = -1;
}

aplicaSignal(motorA, PWM_A, RefVelA_aplicada);
aplicaSignal(motorB, PWM_B, RefVelB_aplicada);

tiempoTranscurrido = millis() - tiempo_ant;
tiempo_ant = millis();
moduloBluetooth.DebugValuesBluetooth(tiempoTranscurrido, wA, wB, RefVelA_aplicada,
                                      RefVelB_aplicada, poseRobot.x, poseRobot.y, poseRobot.phi, PWM_A, PWM_B);
}

```

El modelo cinemático se ha implementado en código de la siguiente manera:

```
void odometria_ModCin()
{
    float phi = poseRobot.phi * PI / 180; // paso a radianes
    float wi = wB * 2 * PI / 60;
    float wd = wA * 2 * PI / 60;

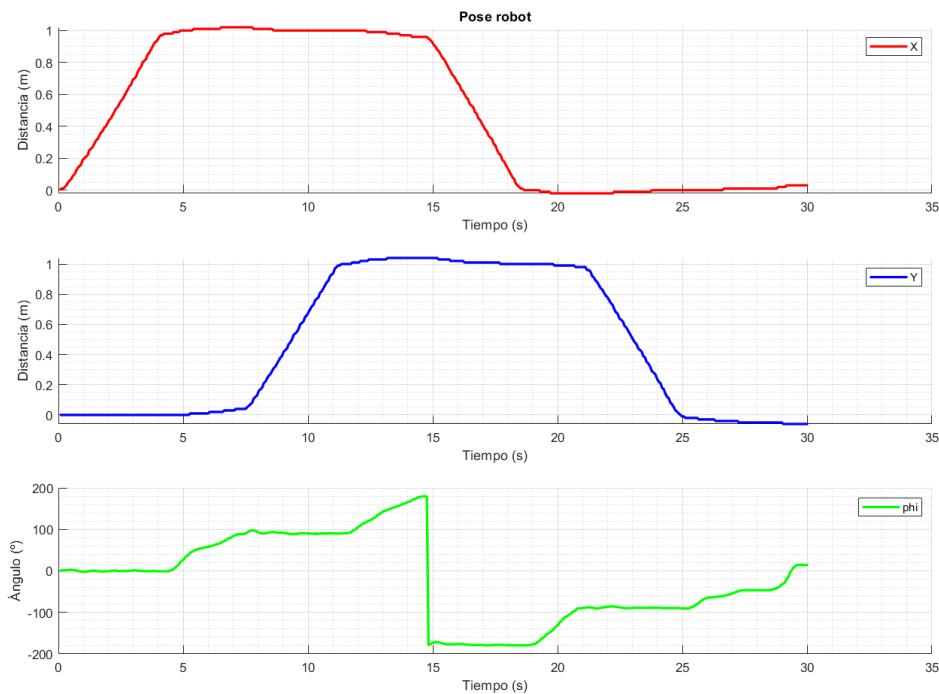
    pose_dRobot.x_d = R / 2 * (wi * cos(phi) + wd * cos(phi));
    pose_dRobot.y_d = R / 2 * (wi * sin(phi) + wd * sin(phi));
    pose_dRobot.phi_d = -(R / b) * wi + (R / b) * wd;
}
```

El resto de las funciones usadas son complementarias para reducir la cantidad de código en el bucle principal del programa. Si resulta de interés pueden consultarse directamente en los códigos entregados.

5.1.6. RESULTADOS EXPERIMENTALES

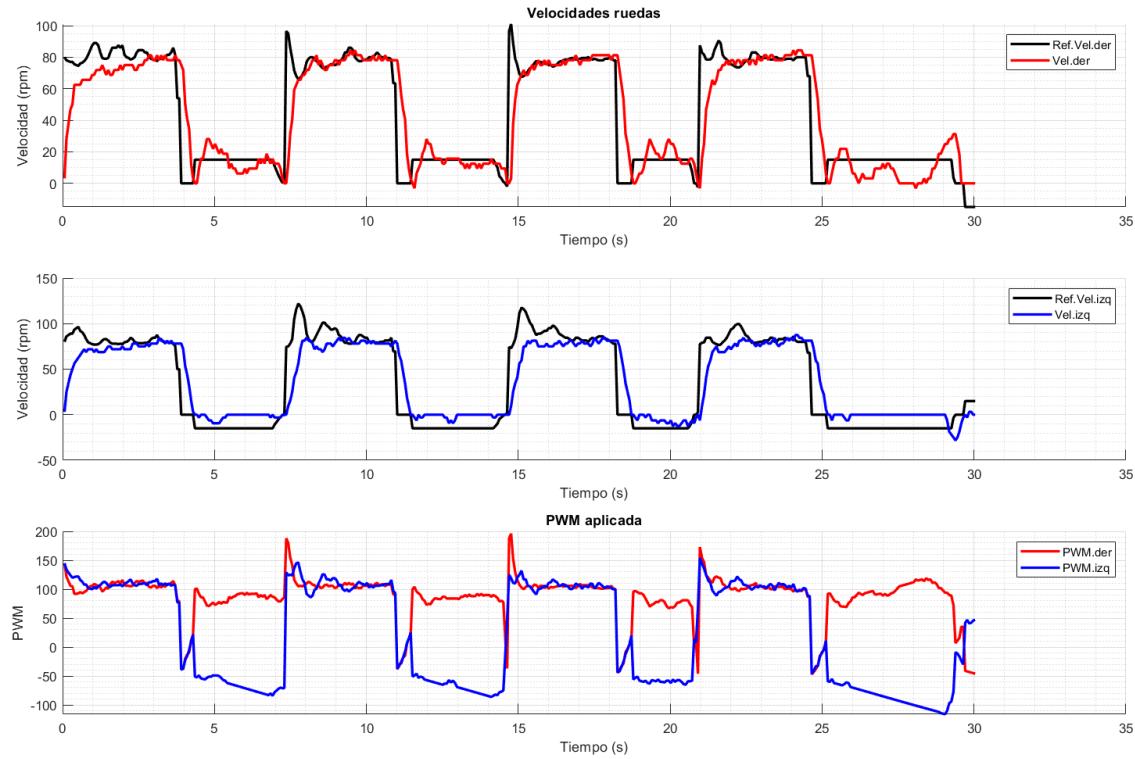
El experimento propuesto para la verificación del comportamiento deseado consiste en la descripción de un cuadrado de lado 1 m por parte del vehículo.

En este modo de funcionamiento hemos decidido representar información extra como la posición del robot en todo momento. Observamos como las coordenadas X e Y varían entorno al rango [0-1] ideal mientras que el ángulo de orientación cambia entre los valores de -180° y 180° a lo largo del experimento.

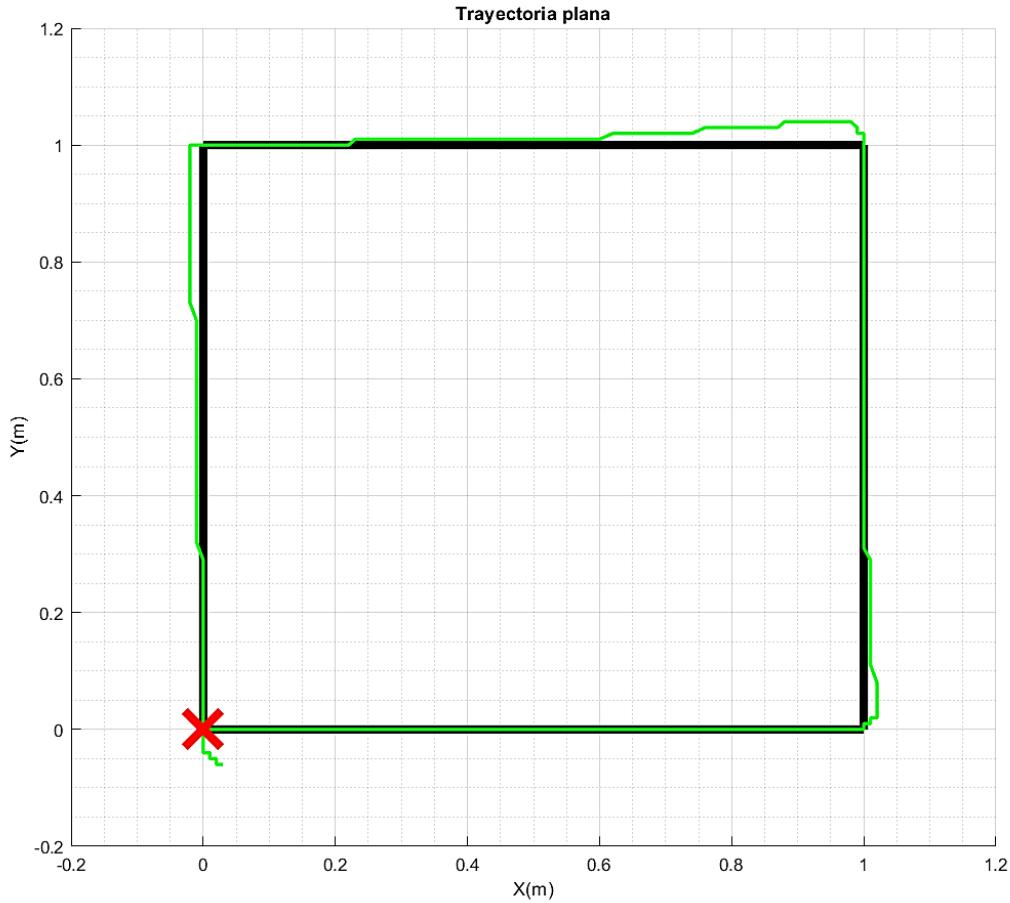


Por otro lado, representamos las referencias de velocidad solicitadas a cada rueda, la velocidad realmente obtenida respecto a la requerida en cada caso y las PWM que aplica el controlador de bajo nivel para conseguirlas. Analizando los resultados podemos identificar las diferentes líneas rectas y giros que se trazan. Centrándonos en las velocidades, los tramos rectos corresponden a aquellos en los que son mayores y de igual signo.

Las distintas velocidades en cada rueda permiten seguir la trayectoria recta deseada entre puntos. Por otro lado, en los giros vemos como las referencias son de signos contrarios para conseguir que el vehículo pivote respecto a su punto central.



Además, mostramos una comparativa entre la trayectoria cuadrada ideal propuesta con la obtenida de la odometría. Desde el punto de vista del vehículo, se consigue llegar a la posición de partida de nuevo tras describir el cuadrado. Los lados del mismo son líneas casi rectas con pequeñas desviaciones debido a perturbaciones que consigue rechazar. En las esquinas tras hacer los giros por pequeños errores se desvía un poco de la trayectoria ideal, pero consigue remediarlos, reinserándose de nuevo en la recta que debiera realizar. Se ratifica el buen comportamiento del control implementado.



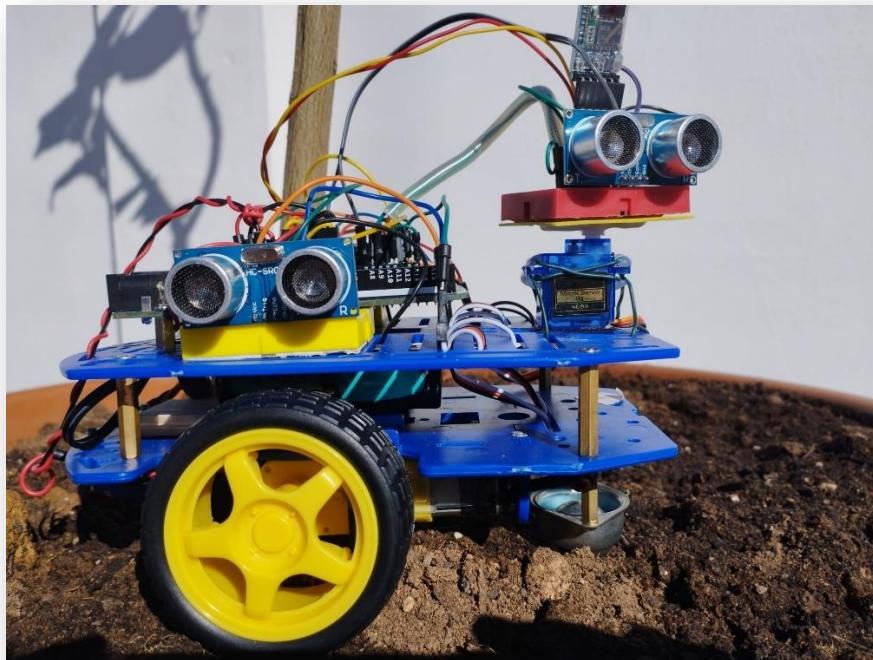
Un punto importante en el que incidimos es que la odometría no es perfecta, y dicha estimación de la posición que se realiza se desvía de la realidad. Esto se debe a que la odometría no puede medir la distancia de forma precisa, sino que depende de los cálculos basados en el número de vueltas de las ruedas, lo cual se traduce en que los datos generados pueden ser no muy exactos respecto a los reales. Los errores en la odometría pueden ser causados por una serie de factores, incluyendo la desalineación, desgaste y rotación irregular de las ruedas, el terreno desigual imperfecto... Estos errores pueden acumularse con el tiempo, lo que resulta en una desviación significativa de la posición real. Por este motivo, la trayectoria descrita en la práctica no es tan del todo buena como la mostrada, sin embargo, podemos estar satisfechos con los resultados, ya que, a pesar de las limitaciones en cuanto a prestaciones, conseguimos alcanzar el punto inicial de partida tras realizar el cuadrado completo con un error aceptable en la práctica.

6. PROYECTO LIBRE

6.1. MODO 8

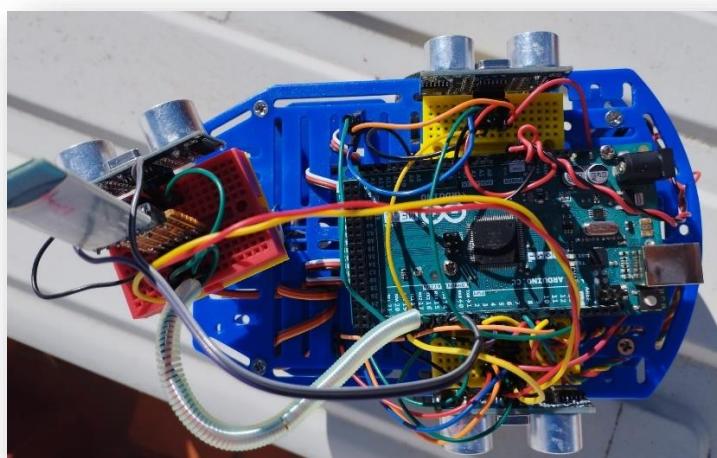
6.1.1. DESCRIPCIÓN

Se pretende desarrollar un nuevo modo de alcance de puntos objetivos dados con detección y evitación de obstáculos. También se propone una aplicación alternativa de seguimiento de carriles con paredes.



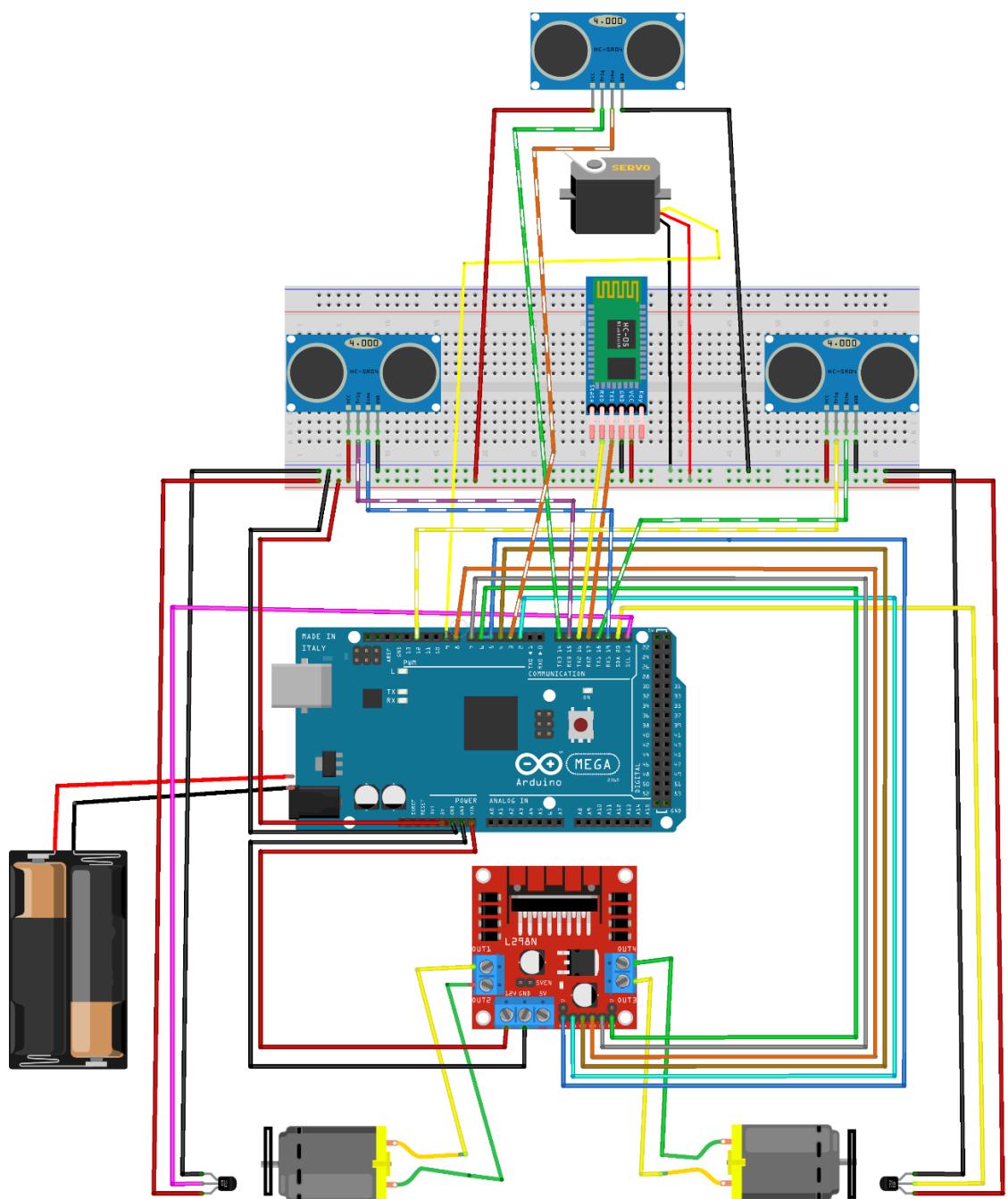
6.1.2. COMPONENTES, MONTAJE Y CONEXIONADO FÍSICO

Incluimos un servomotor que permite realizar un barrido de ángulos para la medición de distancias con un nuevo sensor ultrasonido frontal acoplado al mismo. De esta forma conseguiremos tomar decisiones tras la identificación de un obstáculo con vistas a esquivarlo.



El esquema de conexionado final por tanto incluye estos dos componentes.

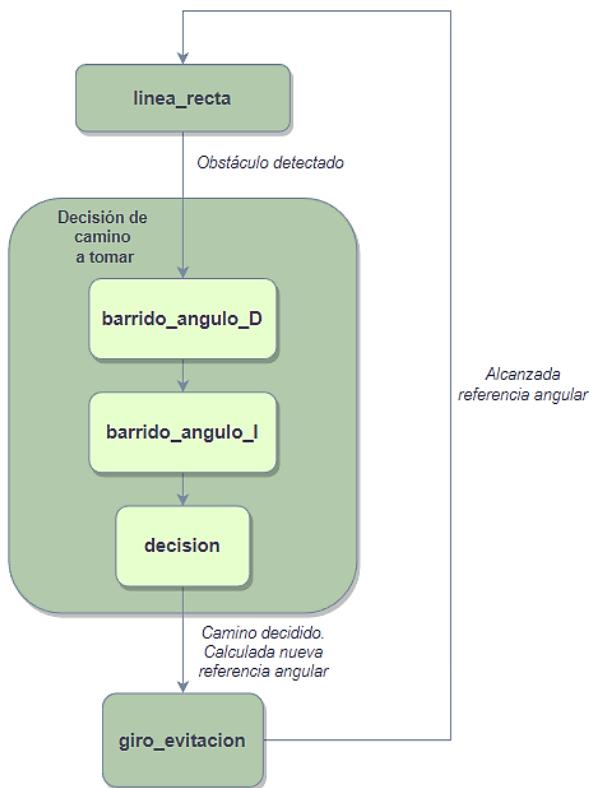
Destacamos los cambios realizados en los pines Echo de los sensores ultrasonidos puesto que los hemos traspasado a pines asociados a interrupciones hardware para medir las distancias teniendo en cuenta los flancos de subida y bajada de estos pines de los sensores. Esto lo hacemos a partir de la duración del pulso en alto. En los primeros modos se hacía con una función de alto nivel la cual introducía un delay despreciable en esos modos básicos pero incompatible en este proyecto avanzado por el uso del servomotor y estimación de la posición con odometría.



6.1.3. ESQUEMA DE CONTROL

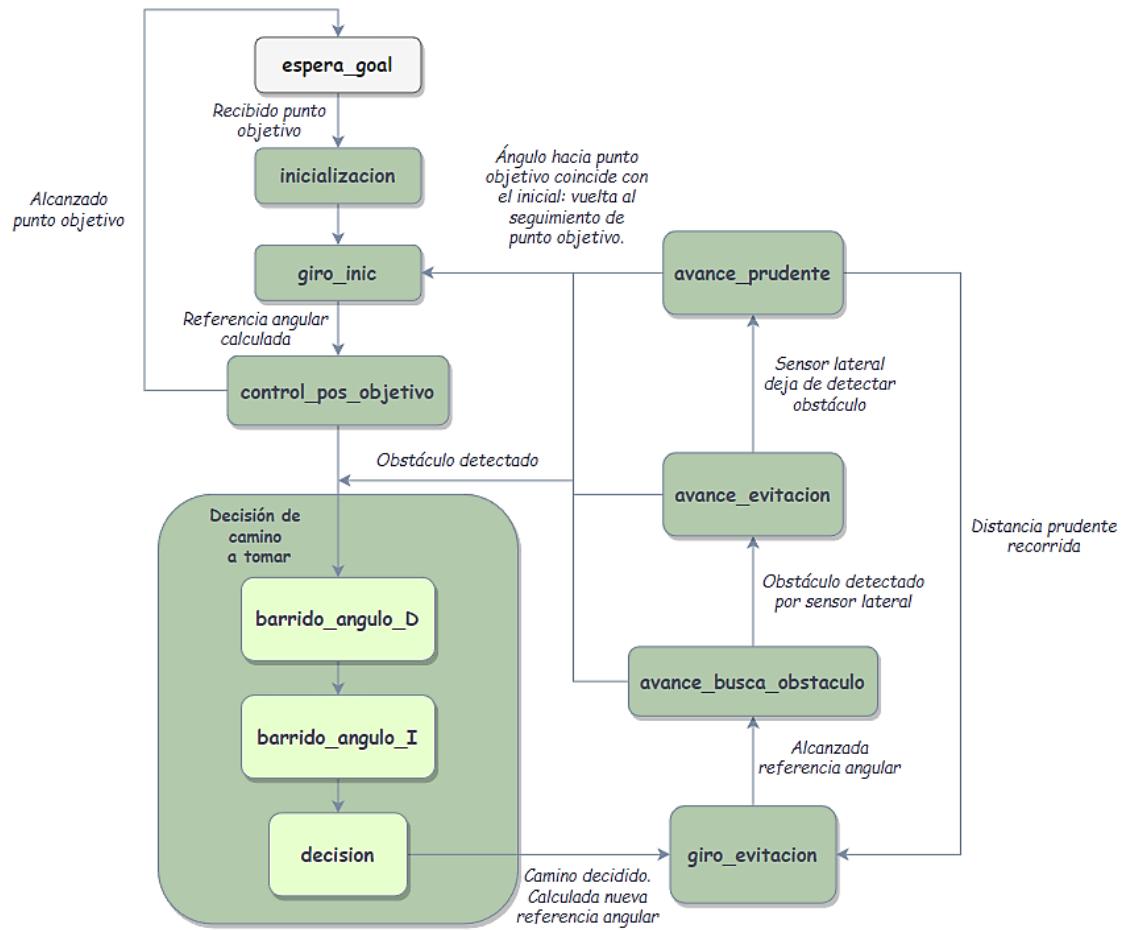
Hemos hecho uso de las distintas técnicas de control usadas hasta ahora explicadas en modos anteriores, combinándolas en máquinas de estados. Las trayectorias descritas se realizan en línea recta basándonos en odometría y esquema de control del modo 7.

Por un lado, para la aplicación de seguimiento de carriles/caminos con esquinas se plantea la siguiente máquina de estados:



Avanzamos constantemente en línea recta. Si detectamos un obstáculo debemos tomar una decisión. Esto se hace mediante la comparación del ángulo barrido por el servomotor a derecha e izquierda del obstáculo. El barrido cesa cuando se detecta una esquina del objeto (cambio abrupto en la medida de distancia por parte de sensor frontal dispuesto sobre el servomotor). Se elegirá realizar un giro de evitación en el sentido correcto eligiendo para ello el correspondiente al menor ángulo barrido. Una vez alcanzado el giro necesario continuamos nuevamente con el avance en línea recta, entrando en bucle.

Por otro lado, como hemos comentado, se pretende dar al robot una coordenada objetivo y que éste la alcance a pesar de los posibles obstáculos que encuentre en el camino, siendo capaz de sortearlos y de reincorporarse a la trayectoria recta inicial con la que se partió. Para conseguir esta funcionalidad se plantea una máquina de estados más avanzada respecto a la anteriormente mostrada. Una vez que se alcance las coordenadas objetivo fijadas nuestro programa se bloqueará esperando un nuevo punto deseado. En este estado se reinician los ejes de coordenadas globales, que pasan a situarse en la posición final alcanzada. Esto se hace para que sea más intuitivo para el usuario el hecho de indicar una nueva posición, sin que tenga que tener en cuenta los ejes globales iniciales al comienzo de los experimentos.



En primer lugar, una vez recibido las coordenadas que se quieren alcanzar nos orientamos hacia ellas con un giro inicial. Seguidamente pasamos al control de alcance de la posición dada. En caso de detección de un obstáculo el comportamiento es idéntico al explicado anteriormente.

Una vez decidido el sentido de giro para la evitación se avanza hasta la detección de una esquina del obstáculo. Hacemos uso para ello de los sensores laterales ultrasónicos. Un salto notorio en la medición de la distancia supondrá haber rebasado la esquina. Se procede a bordearla con un nuevo giro de evitación de sentido contrario al dado inicialmente en la decisión del lado a seguir.

Tenemos un estado de avance en el cual se busca de nuevo el obstáculo con un incremento grande en la medida de los ultrasonidos laterales, misma idea a la inversa que para detección del fin de un objeto. Avanzamos evitándolo en linea recta hasta que nuevamente se identifique una nueva esquina.

Durante los estados de avance podemos detectar nuevos obstáculos existentes. En dicho caso se actuará en consecuencia regresando a los estados de decisión y barridos angulares del servomotor.

También en estos estados de avance es posible salir de la esquivación del obstáculo por identificación de la trayectoria recta inicial de partida.

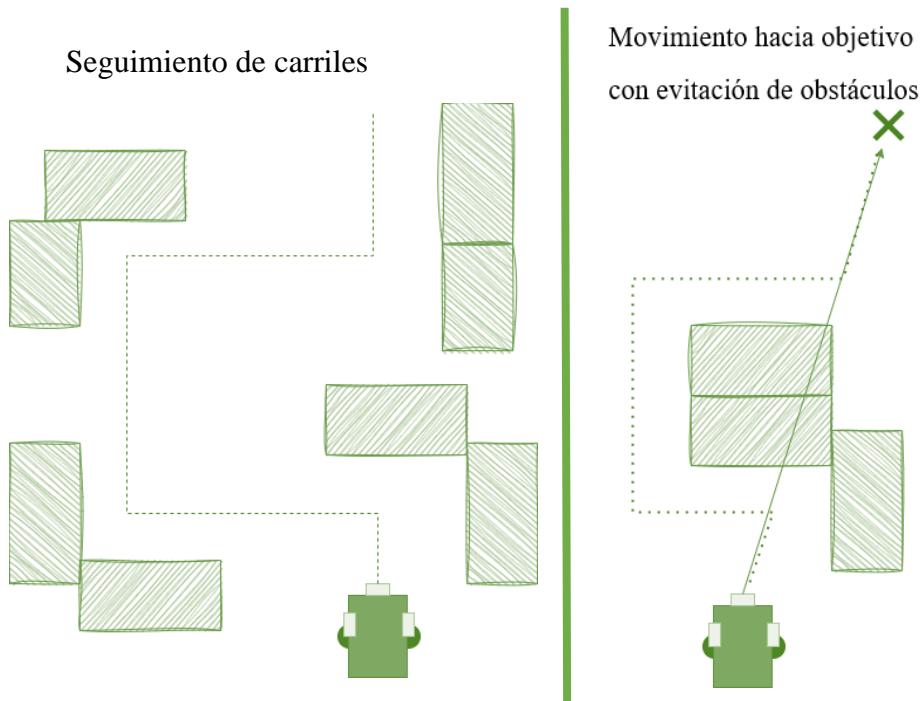
Teóricamente esto se daría en la intersección del movimiento de avance con la recta que va del punto inicial al objetivo. En la práctica comparamos, el ángulo existente entre la pose actual y la objetivo, con el ángulo inicial del movimiento de partida en el giro del comienzo hacia la orientación deseada.

Una vez identificada la posibilidad de continuar con la línea recta inicial se cambia el rumbo, alcanzado la posición objetivo solicitada.

Todos los casos de identificación de esquinas, saltos de distancias, reinserción a trayectoria tienen unos umbrales ajustados para el correcto funcionamiento.

Es importante destacar que los obstáculos con los que trabajamos se suponen que tendrán esquinas y tendrán una forma similar a la de prismas rectangulares.

Las ideas de funcionamiento descritas se recogen en los siguientes dibujos:



6.1.4. IMPLEMENTACIÓN SOFTWARE

El programa desarrollado es bastante extenso, recogemos fragmentos importantes del mismo para hacer apuntes. Si se desea puede revisarse completamente los ficheros adjuntos.

Tras recibir el punto objetivo en primer lugar hacemos la inicialización y giro inicial para orientarnos mirando al mismo.

```

case inicializacion:
    OjoFrontal.write(90);
    RefAngIni = calcula_RefAngAct();
    estadoM = giro_inicial;
    break;

case giro_inicial:
    errorAng = calculaErrorAng(RefAngIni, poseRobot.phi);
    deltaVel = Orientacion.calcularSalida(errorAng);
    RefVelA = deltaVel / 2;
    RefVelB = -deltaVel / 2;
    if (abs(errorAng) < 5 && motorA.sentidoGiro == 0 && motorB.sentidoGiro == 0)
    {
        estadoM = control_pos_objetivo;
        flagCompruebaObstaculo = true;
        Orientacion.resetPID();
        Distancia.resetPID();
    }
    break;

```

El estado principal de avance en linea recta hacia la posición indicada tiene en cuenta la medida del sensor ultrasónico frontal para entrar en los estados de barridos de ángulos y toma de decisión.

```

case control_pos_objetivo:
    RefAng = calcula_RefAngAct();
    moduloDistancia = sqrt(pow(poseRobot.x - posicionObjetivo.x, 2) + pow(poseRobot.y - posicionObjetivo.y, 2));

    RefVelA = Distancia.calcularSalida(0 - moduloDistancia);
    RefVelB = RefVelA;
    errorAng = calculaErrorAng(RefAng, poseRobot.phi);
    deltaVel = Orientacion.calcularSalida(errorAng);
    RefVelA += deltaVel / 2;
    RefVelB += -deltaVel / 2;

    if (moduloDistancia < errorAdmitidoDistance) // cuando el error sea pequeño y el robot ya no tenga inercia (esté parado)
    {
        estadoM = espera_goal;
        Orientacion.resetPID();
        RefVelA = 0;
        RefVelB = 0;
    }

    break;

```

Dichos estados asociados a la decisión del lado por el que continuar son los que se ven a continuación:

```

case barrido_servo_D:
    if ((abs(sonicFrontal.valorMedida - medida_ant) > umbral_inc_dist) || (ang_barrido_D >= 90))
    {
        // Inicialización para barrer angulo I
        OjoFrontal.write(90);
        medida_ant = distancia_prudente;
        ang_barrido_I = 0;
        delay(300);
        estadoM = barrido_servo_I;
    }
    else
    {
        ang_barrido_D += paso_angular;
        OjoFrontal.write(90 - ang_barrido_D);
        medida_ant = sonicFrontal.valorMedida;
    }
    delay(300);
    break;

```

```

case barrido_servo_I:
    if ((abs(sonicFrontal.valorMedida - medida_ant) > umbral_inc_dist) || (ang_barrido_I >= 90))
    {
        OjoFrontal.write(90);
        estadoM = decision;
    }
    else
    {
        ang_barrido_I += pasoAngular;
        OjoFrontal.write(90 + ang_barrido_I);
        medida_ant = sonicFrontal.valorMedida;
    }
    delay(300);
    break;
}

```

Decidimos y tomamos una referencia de ángulo respecto a la actual adecuada para el posicionamiento correcto respecto al obstáculo de cara a la evitación de las esquinas.

```

case decision:
    if (ang_barrido_D <= ang_barrido_I)
    {
        RefAng = poseRobot.phi - 90; // RefAng=PosePhi - 90 (respecto a pose actual giro 90 grados a la derecha)
        if (RefAng <= -180)
        | RefAng = 360 + RefAng;
        flagGiro = true;
    }
    else
    {
        RefAng = poseRobot.phi + 90; // RefAng=PosePhi + 90 (respecto a pose actual giro 90 grados a la izquierda)
        if (RefAng > 180)
        | RefAng = RefAng - 360;
        flagGiro = false;
    }
    estadoM = giro_evitacion;
    break;

case giro_evitacion:
    errorAng = calculaErrorAng(RefAng, poseRobot.phi);
    deltaVel = Orientacion.calcularSalida(errorAng);
    RefVelA = deltaVel / 2;
    RefVelB = -deltaVel / 2;
    if (abs(errorAng) < 5 && motorA.sentidoGiro == 0 && motorB.sentidoGiro == 0)
    {
        estadoM = avance_busca_obstaculo;
        RefVelA = 0;
        RefVelB = 0;
        Orientacion.resetPID();
        Distancia.resetPID();
        flagCompruebaObstaculo = true;
    }
    break;
}

```

Hablamos ahora de los distintos estados de avance.

El estado de buscar obstáculos sirve para una vez que se ha detectado la esquina y se ha girado respecto a la misma para bordearla, vuelva a avanzar en busca del obstáculo usando el sensor lateral que corresponda.

```

case avance_busca_obstaculo: // La primera vez es inestable (sirve para cuando bordeamos esquinas)
if (((sonicLatD.valorMedida < distancia_prudente + umbral_inc_dist) && (!flagGiro)) ||
    ((sonicLatI.valorMedida < distancia_prudente + umbral_inc_dist) && (flagGiro)))
{
    estadoM = avance_evitacion;
    // No reseteamos controladores usados porque se sigue avanzando en el mismo sentido
}
else
{
    RefVelA = 50;
    RefVelB = RefVelA;
    errorAng = calculaErrorAng(RefAng, poseRobot.phi);
    deltaVel = Orientacion.calcularSalida(errorAng);
    RefVelA += deltaVel / 2;
    RefVelB += -deltaVel / 2;
}
// Comprobación salida hacia punto objetivo
RefAngAct = calcula_RefAngAct();
if (abs(RefAngAct - RefAngIni) <= umbral_ang_recta)
{
    if (flag_ang_recta)
    {
        estadoM = giro_inicial;
        Orientacion.resetPID();
    }
}
break;

```

Este estado de avance es el principal, supone el avance paralelo respecto al plano del obstáculo teniendo en cuenta cuando se dará una nueva esquina para dada dicha condición pasar al estado de avance prudente.

```

case avance_evitacion:
RefVelA = 50;
RefVelB = RefVelA;
errorAng = calculaErrorAng(RefAng, poseRobot.phi);
deltaVel = Orientacion.calcularSalida(errorAng);
RefVelA += deltaVel / 2;
RefVelB += -deltaVel / 2;

// Comprobación salida hacia punto objetivo
RefAngAct = calcula_RefAngAct();
if (abs(RefAngAct - RefAngIni) <= umbral_ang_recta)
{
    if (flag_ang_recta)
    {
        estadoM = giro_inicial;
        Orientacion.resetPID();
    }
}

else
{
    flag_ang_recta = true;
    if (((sonicLatD.valorMedida > distancia_prudente + umbral_inc_dist) && (!flagGiro)) ||
        ((sonicLatI.valorMedida > distancia_prudente + umbral_inc_dist) && (flagGiro)))
    {
        pose_corner = poseRobot;
        estadoM = avance_prudente;
        // No reseteamos controladores usados porque se sigue avanzando en el mismo sentido
    }
}
break;

```

En avance prudente se avanza una cierta distancia para alejarse algo de la esquina a la hora de hacer el giro de evitación de la misma.

```

case avance_prudente:
    distanciaRecorrida = sqrt(pow(poseRobot.x - pose_corner.x, 2) + pow(poseRobot.y - pose_corner.y, 2));
    RefVelA = Distancia.calcularSalida(distanciaRecorrida - ((0.5 * distancia_prudente) * 0.01));
    RefVelB = RefVelA;
    errorAng = calculaErrorAng(RefAng, poseRobot.phi);
    deltaVel = Orientacion.calcularSalida(errorAng);
    RefVelA += deltaVel / 2;
    RefVelB += -deltaVel / 2;

    // Comprobación salida hacia punto objetivo
    RefAngAct = calcula_RefAngAct();
    if (abs(RefAngAct - RefAngIni) <= umbral_ang_recta)
    {
        if (flag_ang_recta)
        {
            estadoM = giro_inicial;
            Orientacion.resetPID();
        }
    }
    else
    {
        if (distanciaRecorrida >= ((0.5 * distancia_prudente) * 0.01))
        {
            estadoM = giro_evitacion;
            Orientacion.resetPID();
            Distancia.resetPID();
            RefVelA = 0;
            RefVelB = 0;
            if [flagGiro]
            {
                RefAng = poseRobot.phi + 90;
                if (RefAng > 180)
                    RefAng = RefAng - 360;
            }
            else
            {
                RefAng = poseRobot.phi - 90;
                if (RefAng <= -180)
                    RefAng = 360 + RefAng;
            }
        }
    }
}

break;

```

Vemos como en los estados de avance comentados se encuentran los condicionantes para la reinserción a la trayectoria recta original.

De forma común a todos los estados se encuentra el controlador de bajo nivel y la aplicación de la señal de control PWM a los motores para conseguir el movimiento del vehículo.

También está el envío vía bluetooth de todos los datos para la telemetría.

```

// Comprobamos si el signo de la referencia ha cambiado
comprueba_cambio_sentido(RefVelA, &RefVelA_aplicada, motorA);
comprueba_cambio_sentido(RefVelB, &RefVelB_aplicada, motorB);

if (RefVelA_aplicada == 0 || RefVelB_aplicada == 0) // para conseguir que ambas ruedas empiecen a moverse a la vez
{
    RefVelA_aplicada = 0;
    RefVelB_aplicada = 0;
}

PWM_A = ControlVelocidadA.calcularSalida(RefVelA_aplicada - wA);
if (RefVelA_aplicada > 0)
{
    PWM_A += puntoEquilibrio;
    motorA.sentidoGiro = 1;
}
else if (RefVelA_aplicada < 0)
{
    PWM_A -= puntoEquilibrio;
    motorA.sentidoGiro = -1;
}

PWM_B = ControlVelocidadB.calcularSalida(RefVelB_aplicada - wB);
if (RefVelB_aplicada > 0)
{
    PWM_B += puntoEquilibrio;
    motorB.sentidoGiro = 1;
}
else if (RefVelB_aplicada < 0)
{
    PWM_B -= puntoEquilibrio;
    motorB.sentidoGiro = -1;
}

aplicaSignal(motorA, PWM_A, RefVelA_aplicada);
aplicaSignal(motorB, PWM_B, RefVelB_aplicada);

tiempo = millis() - tiempoAnt;
tiempoAnt = millis();

moduloBluetooth.DebugValuesBluetooth(tiempo, RefVelA_aplicada, RefVelB_aplicada, RefAng, posicionObjetivo.x, posicionObjetivo.y,
| poseRobot.x, poseRobot.y, poseRobot.phi, estadoM, sonicLatD.valorMedida, sonicLatI.valorMedida, sonicFrontal.valorMedida);
}

```

La medición de las distancias se realiza en la rutina de interrupción del timer que tenemos, cada 100 ms (una de cada dos veces que se da, recordamos que la odometría la calculamos cada 50 ms).

```

void refresh_interrupt()
{
    static bool flagMedidasUltrasonidos = true;
    calculaVelAng();
    odometria_ModCin();
    poseRobot.x += pose_dRobot.x_d * 0.05;
    poseRobot.y += pose_dRobot.y_d * 0.05;
    poseRobot.phi += pose_dRobot.phi_d * 0.05 * 180 / PI;
    // cuadrar en rango (-180,180]
    if (poseRobot.phi <= -180)
    {
        poseRobot.phi += 360;
    }
    else if (poseRobot.phi > 180)
    {
        poseRobot.phi -= 360;
    }

    if (flagMedidasUltrasonidos)
    {
        medicionUltrasonidos();
        flagMedidasUltrasonidos = false;
    }
    else
        flagMedidasUltrasonidos = true;
}

```

Las rutinas de interrupción para la medida de las distancias son como la que se muestra de ejemplo para la lateral derecha:

```
void ISR_recogeEchoD()
{
    static bool flagRising = true;
    static unsigned long tiempoRising = 0;

    if (flagRising)
    {
        tiempoRising = micros();
        flagRising = false;
    }
    else
    {
        sonicLatD.valorMedida = (float)((micros() - tiempoRising) / 58.0); // convertimos a distancia, en cm
        flagRising = true;
    }
}
```

Particularmente en la asociada al sensor frontal tenemos en cuenta la detección continua de obstáculos:

```
void ISR_recogeEchoF()
{
    static bool flagRising = true;
    static unsigned long tiempoRising = 0;

    if (flagRising)
    {
        tiempoRising = micros();
        flagRising = false;
    }
    else
    {
        sonicFrontal.valorMedida = (float)((micros() - tiempoRising) / 58.0); // convertimos a distancia, en cm
        flagRising = true;
    }

    if ((sonicFrontal.valorMedida <= distancia_prudente) && flagCompruebaObstaculo)
    {
        Orientacion.resetPID();
        RefVelA = 0;
        RefVelB = 0;
        // Inicialización para barrer angulo D
        OjoFrontal.write(90);
        medida_ant = distancia_prudente;
        ang_barriodo_D = 0;
        estadoM = barriodo_servo_D;
        flagCompruebaObstaculo = false;
    }
}
```

NOTA: El seguimiento de carriles usa parte de los estados mostrados, por lo que es una simplificación respecto a lo mostrado.

6.1.5. RESULTADOS EXPERIMENTALES

Se han llevado a cabo numerosos experimentos para testear el funcionamiento.

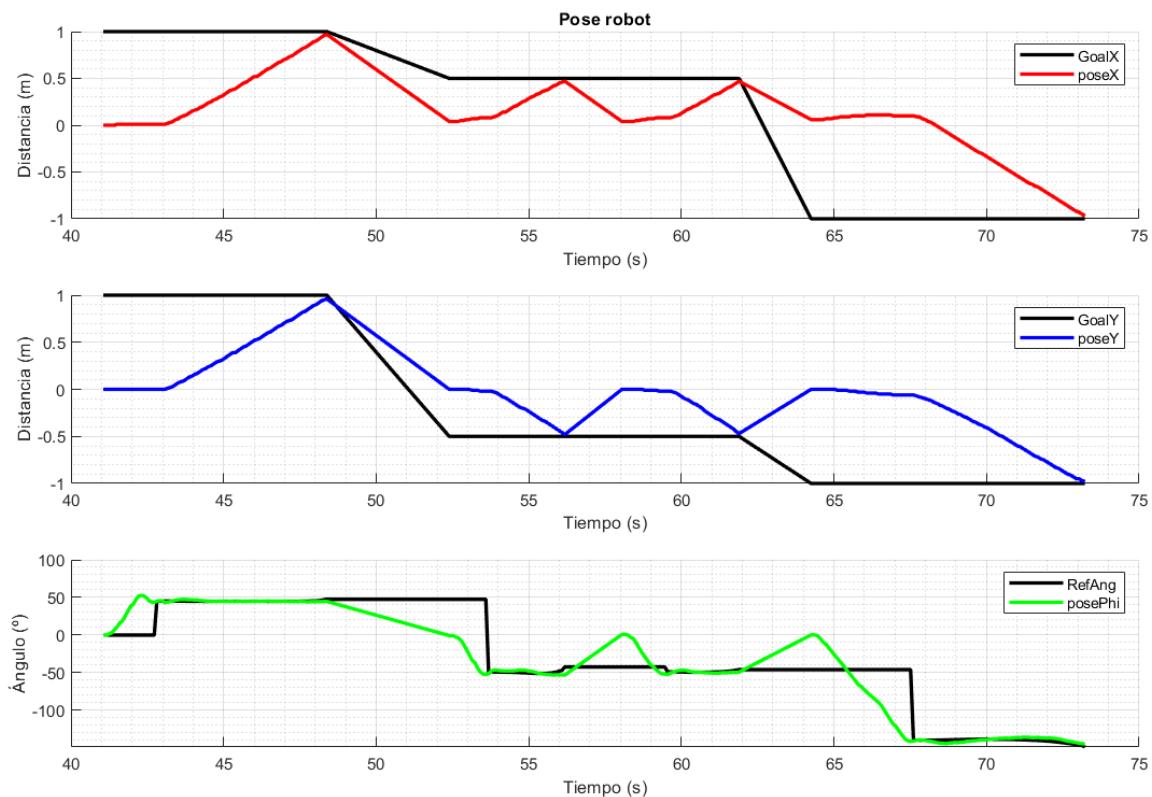
Recogemos algunos ejemplos destacados que corroboran el cumplimiento de las especificaciones previas fijadas.

- Cambio de puntos deseados:

Mandamos distintos puntos objetivos y comprobamos como el robot los alcanza.

Las coordenadas a alcanzar y las posiciones instantáneas, así como la referencia de ángulo y ángulo de la pose experimentan las siguientes transiciones. Recordamos que tras la llegada al punto se resetean los ejes globales, siendo dicha posición el origen del sistema global. Observamos como en el cambio de posición deseada la pose actual pasa a ser la (0,0) y alcanza el punto marcado. La secuencia dada ha sido:

$$(1,1), (0.5,-0.5), (0.5, -0.5), (-1,-1)$$

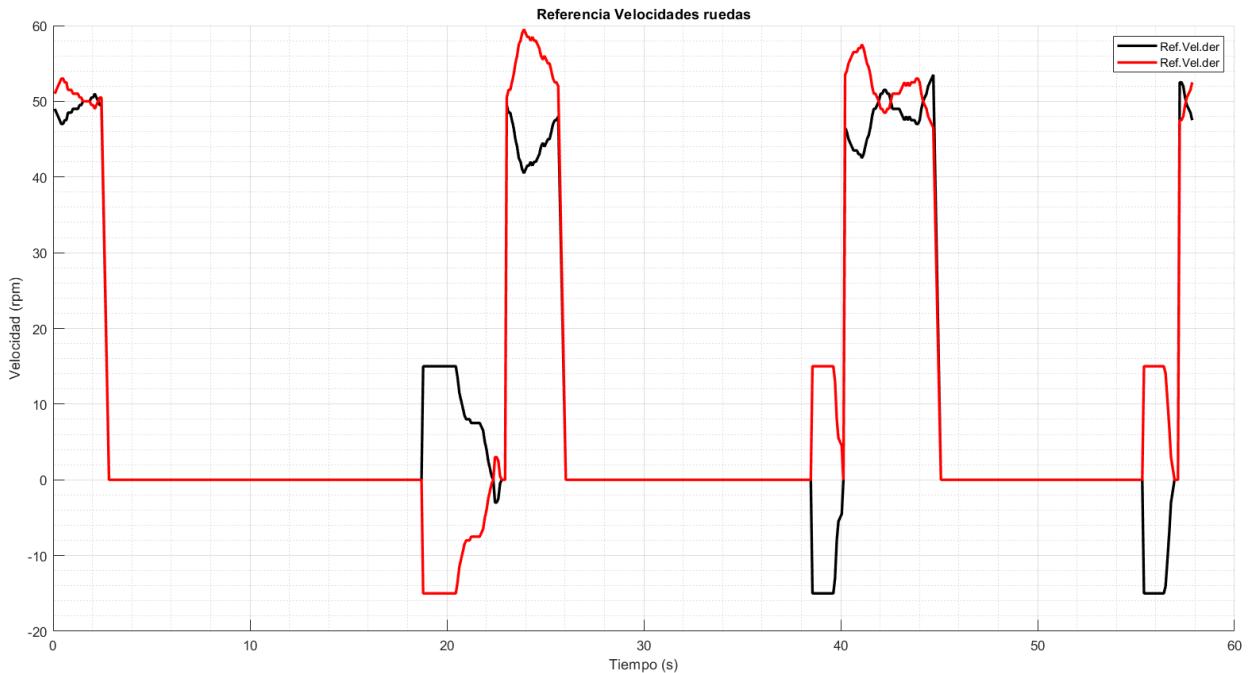


- Seguimiento de carriles con paredes:

Planteamos una disposición de cajas similar a la mostrada en el dibujo explicativo mostrado anteriormente.



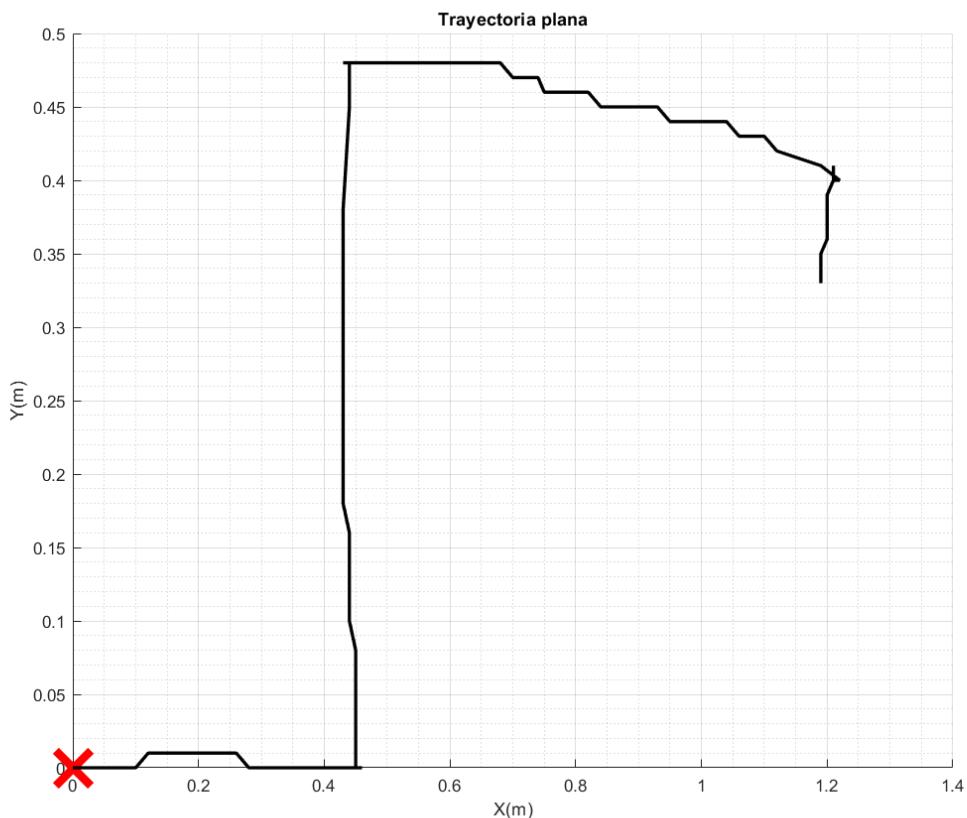
Las referencias de velocidades para las ruedas pueden verse en la siguiente gráfica. Distinguimos los giros y avances de forma clara. En este experimento se realizan 3 giros y 4 avances tras la identificación de las esquinas.



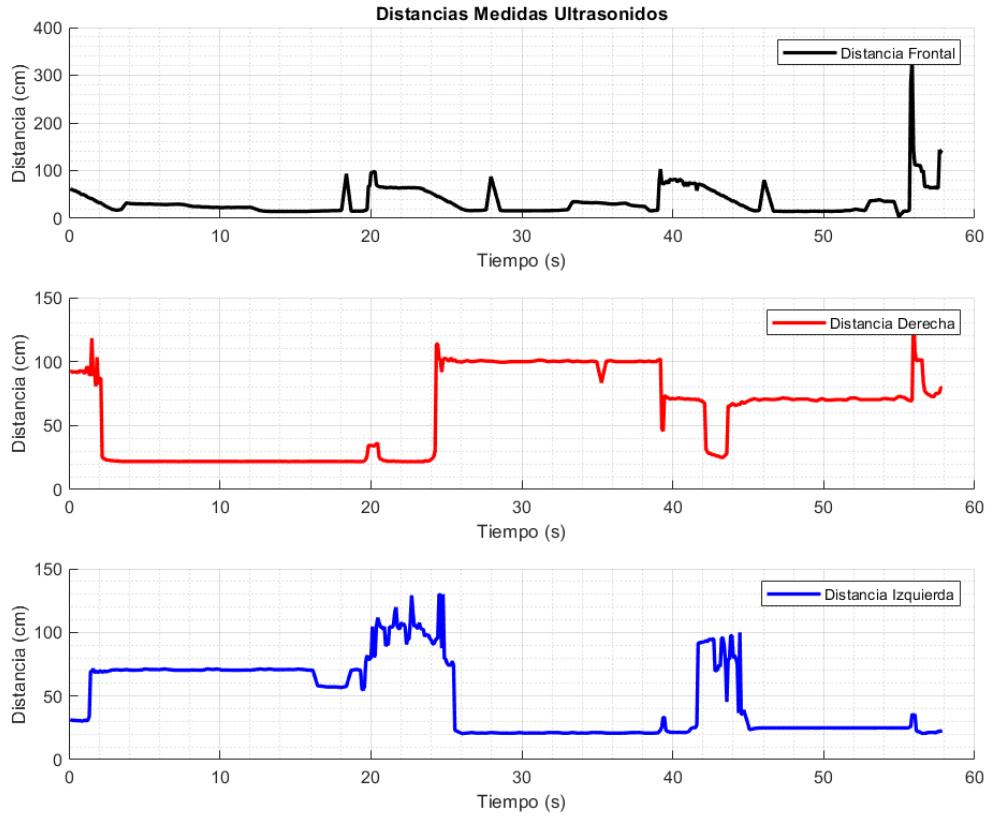
La trayectoria de movimiento seguido se recoge a la izquierda.

Podemos realizar una comparativa con la imagen real mostrada anteriormente y comprobar como se ha descrito el carril.

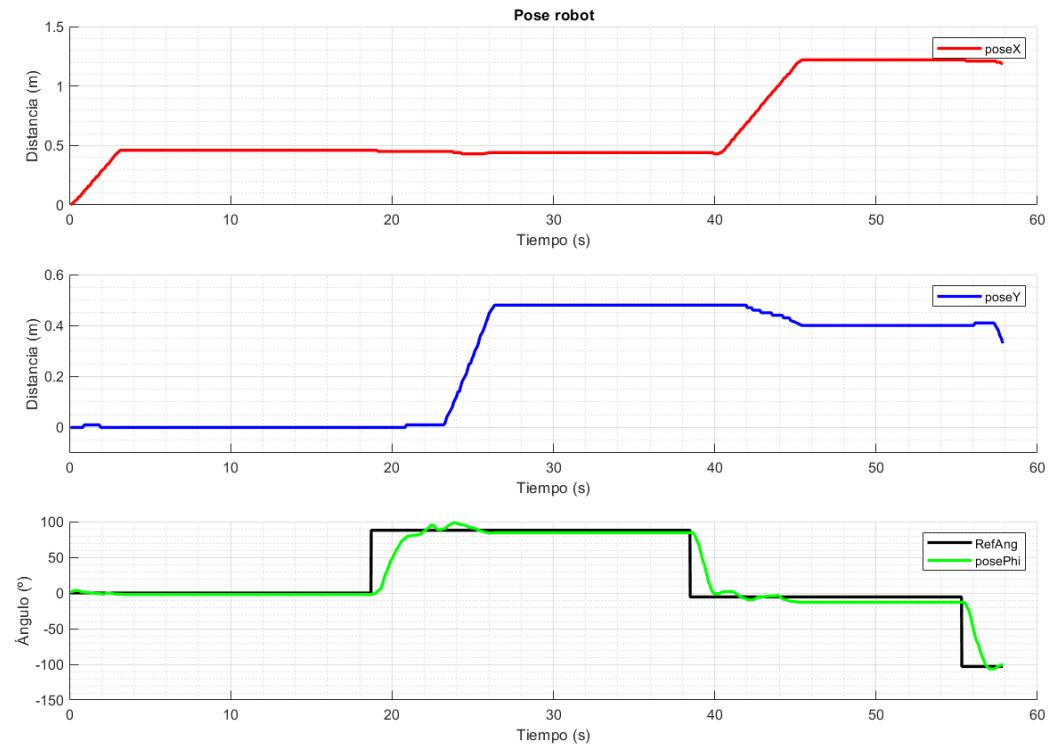
En líneas rectas se realiza un control del ángulo tras el giro dado, de tal manera que se mantenga el movimiento en linea recta controlando la orientación a dicho ángulo. Debido a deslizamientos del suelo que causan perturbaciones se da una pequeña desviación respecto a la recta ideal de partida de varios centímetros en el último tramo. No se trata de fallo en el control, ya que como vemos, corrige la orientación a la misma que tras el giro en los pequeños saltos en distancia que se introducen. En la práctica no supone ningún inconveniente en la consecución del seguimiento del carril, como se demuestra en videos.



Las distancias medidas en todo momento por los tres sensores ultrasonidos dispuestos en torno al vehículo (laterales y frontal) se recogen a continuación. Podemos ver en el frontal los picos que se dan, los cuales sirven para la identificación de esquinas usados para fin de barridos del servo. En este modo de funcionamiento los laterales no son usados para tomar decisiones. Sin embargo, muestran información acerca de las distancias existentes a los obstáculos en todo momento.



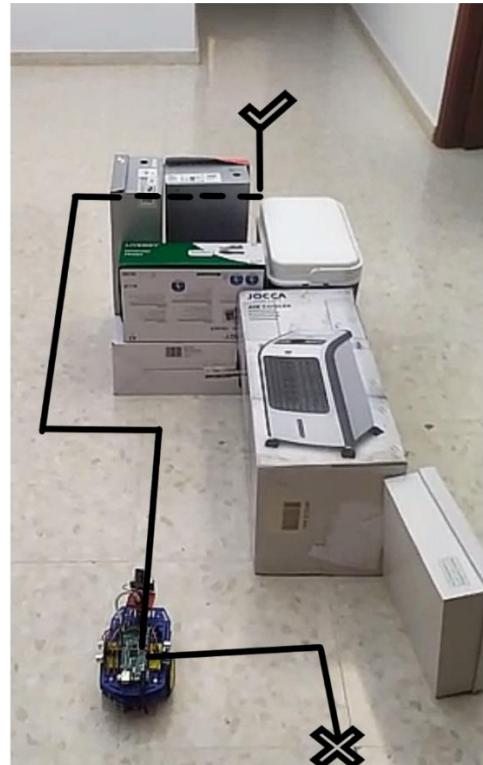
Por último, también mostramos la evolución de la pose y la referencia de ángulo para la orientación de nuestro robot diferencial. Esta vez no tenemos posiciones objetivo ya que se trata de un movimiento continuo.



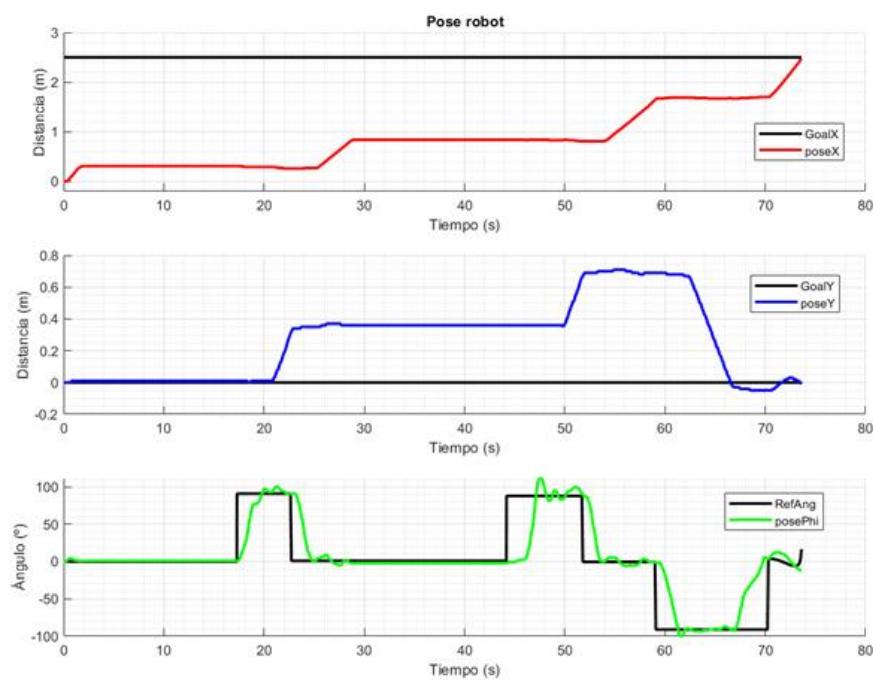
- Evitación de obstáculos en avance a punto deseado:

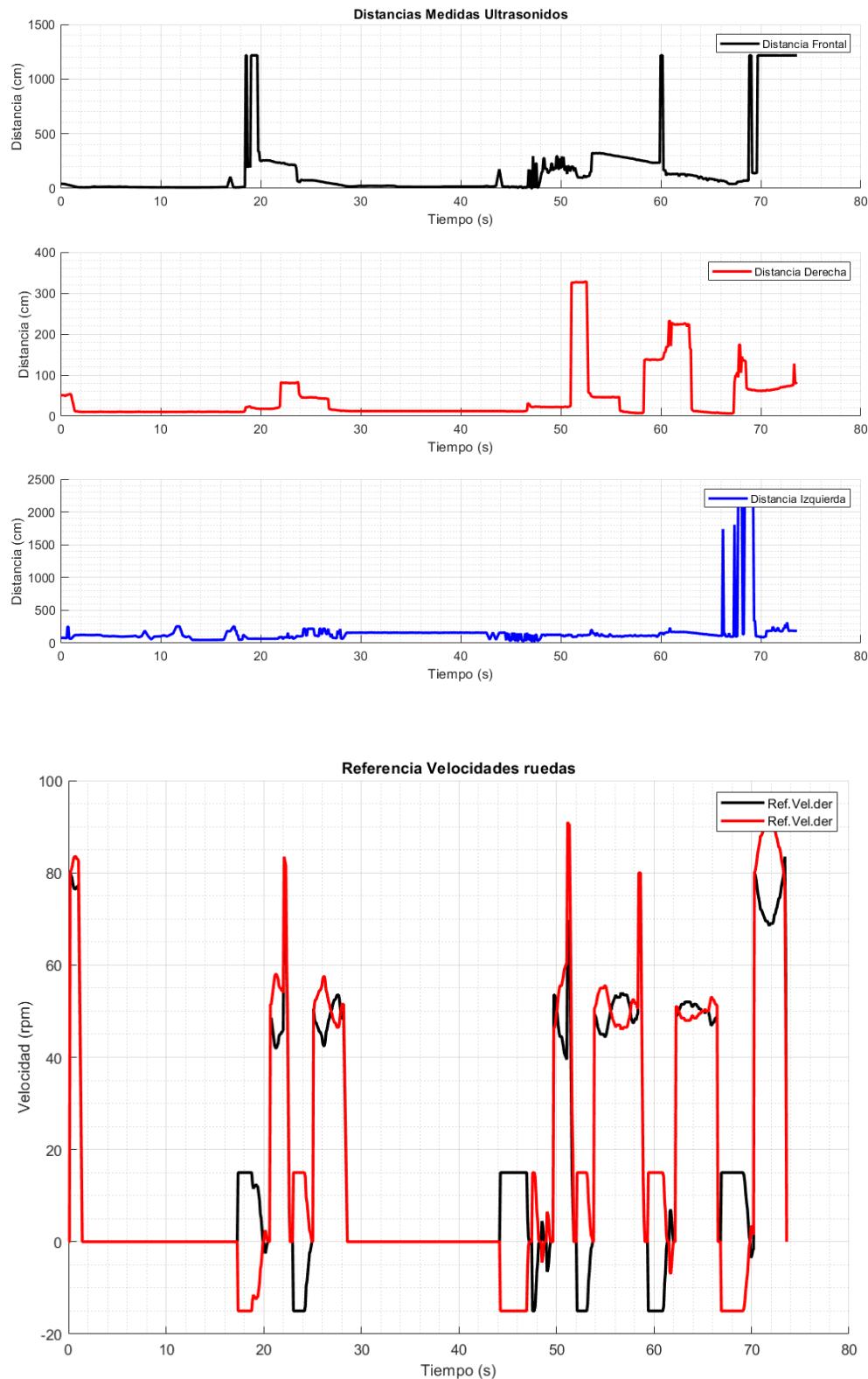
En el experimento de ejemplo planteado se pretenderá demostrar la evitación efectiva de objetos con esquinas.

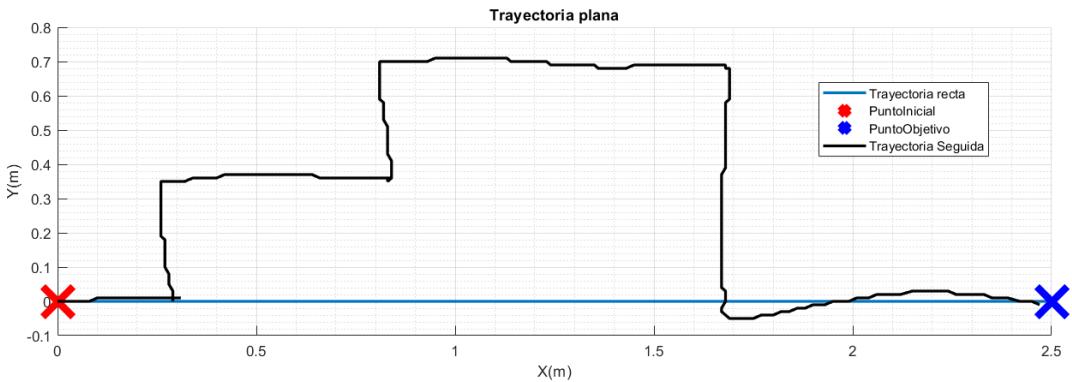
Esta ha sido la disposición del entorno y el camino seguido.



Mostramos las gráficas pertinentes con los resultados obtenidos. Se demuestra el alcance del punto deseado con la previa identificación y esquive de las cajas.







Hacemos especial hincapié en la trayectoria de movimiento obtenida, la cual puede contrastarse con la fotografía de la disposición de cajas mostrada.

6.1.6. AMPLIACIÓN CONTROLADOR PURE-PURSUIT

Planteamos una posible mejora al sistema actual consistente en la implementación de un controlador de alto nivel con técnica Pure-Pursuit para el seguimiento de caminos. Se ha desarrollado una versión de prueba simplificada con vistas a la posibilidad de tener que recurrir a ella en un momento dado si el comportamiento del sistema no cumplía especificaciones.

A continuación, comentaremos un poco acerca de lo desarrollado aunque finalmente no se ha implementado en el proyecto final debido al buen comportamiento conseguido sin necesidad de implementar esta técnica algo más avanzada.

En resumen, consiste en la división de la trayectoria en linea recta a seguir en puntos intermedios hasta alcanzar el objetivo final. De esta forma conseguiríamos un mejor comportamiento en teoría en el seguimiento del camino.

A dichos puntos intermedios los llamamos “waypoints” y al punto concreto al que nos dirigiremos será el punto de “lookahead”.

Para la obtención de los waypoints dada una posición objetivo concreta se toman puntos pertenecientes al segmento entre la posición inicial y la dada. Los waypoints se almacenan de forma ordenada respecto al sentido deseado de movimiento.

```
// Cálculo de waypoints de la trayectoria recta que se pretende seguir
inc_x_waypoint = (posicionObjetivo.x - poseRobot.x) / n_waypoints;
inc_y_waypoint = (posicionObjetivo.y - poseRobot.y) / n_waypoints;
for (int i = 0; i < n_waypoints - 1; i++)
{
    waypoints[i].x = poseRobot.x + inc_x_waypoint * (i + 1);
    waypoints[i].y = poseRobot.y + inc_y_waypoint * (i + 1);
}
waypoints[n_waypoints - 1].x = posicionObjetivo.x;
waypoints[n_waypoints - 1].y = posicionObjetivo.y;
```

En todo momento, el punto objetivo dejaría de ser el dado y pasaría a ser variable según la posición actual del robot. La referencia angular se hallaría respecto al punto de lookahead en cada instante, que como hemos dicho, será variable.

```
case control_pos_objetivo: // control pure pursuit para la trayectoria recta //control: necesito RefVelA y RefVelB

    // Punto de lookAhead
    lookAheadIndex = lookAheadWaypoint(poseRobot, waypoints, n_waypoints);
    lookAhead = waypoints[lookAheadIndex];
    deltaX = lookAhead.x - poseRobot.x;
    deltaY = lookAhead.y - poseRobot.y;
    RefAng = (float)atan2(deltaY, deltaX);
    if (RefAng == -PI)
        | RefAng += 2 * PI;
    RefAng *= 180.0 / PI;

    moduloDistancia = sqrt(pow(poseRobot.x - lookAhead.x, 2) + pow(poseRobot.y - lookAhead.y, 2));
```

Identificamos el punto de lookahead, en una versión propuesta simple, identificando el waypoint más cercano a la posición actual instantánea del vehículo. Escogemos como punto lookahead el siguiente waypoint al de mínima distancia.

```
int lookAheadWaypoint(pose state, goal waypoints[], int numWaypoints)
{ // función para calcular el indice del waypoint que sera el punto de lookAhead
    int closestIndex = 0;
    int lookAheadIndex = closestIndex;
    double closestDistance = 1000000;
    for (int i = 0; i < numWaypoints; i++)
    {
        double distance = sqrt(pow(state.x - waypoints[i].x, 2) + pow(state.y - waypoints[i].y, 2));
        if (distance < closestDistance)
        {
            closestIndex = i;
            closestDistance = distance;
        }
    }
    lookAheadIndex = closestIndex;
    if (closestIndex != numWaypoints - 1)
    {
        lookAheadIndex = closestIndex + 1;
    }
    return lookAheadIndex;
}
```

Ajustando el número de waypoints requeridos para un buen funcionamiento en relación con la distancia entre los mismos y ajustando las ganancias de los controladores PID se podría conseguir implementar este desarrollo planteado del cual se esperarían a priori buenos resultados.