



## ESCUELA DE INGENIERÍAS INDUSTRIALES

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Área de conocimiento de Ingeniería de Sistemas y Automática

### TRABAJO FIN DE GRADO

---

#### **Planificación de caminos reactiva mediante persecución pura en coche a escala utilizando ROS y Gazebo**

---

Pure-pursuit reactive path planning with a scale car using ROS and Gazebo

Titulación: Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Autor: Alba Marqués

Tutor:



**Resumen:**

El objetivo de este Trabajo de Fin de Grado consiste en la implementación de un método reactivo de planificación de caminos que, mediante persecución pura, permita realizar el seguimiento de paredes, pasillos y personas con un coche a escala 1:10, equipado con un telémetro láser bidimensional (2D). Se utilizará el metasistema operativo ROS sobre una placa de desarrollo Jetson TX2.

**Palabras claves:**

ROS, Gazebo, navegación autónoma, planificación reactiva de caminos, persecución pura, telémetro láser bidimensional

---

**Abstract:**

The aim of this Degree's Final Project entails the implementation of a reactive path planning method, utilizing pure-pursuit, to track walls, corridors and individuals with a model car to a scale 1:10, equipped with a two-dimensional (2D) laser rangefinder. The operating metasystem ROS will be utilized in the Jetson TX2 development board.

**Keywords:**

ROS, Gazebo, autonomous navigation, reactive path-planning, pure-pursuit, two-dimensional laser rangefinder



---

# Índice de contenidos

---

<b>1. Introducción</b>	<b>1</b>
1.1. Antecedentes . . . . .	1
1.2. Objetivo . . . . .	1
1.3. Fases de trabajo . . . . .	2
1.4. Estructura de la memoria . . . . .	2
<b>2. Herramientas empleadas</b>	<b>3</b>
2.1. ROS . . . . .	3
2.2. Gazebo . . . . .	3
2.3. UMA RACECAR . . . . .	4
2.4. Láser 2D . . . . .	5
<b>3. Planificación reactiva de caminos</b>	<b>7</b>
3.1. Introducción . . . . .	7
3.2. Algoritmo de persecución pura . . . . .	8
3.3. Seguimiento de pared . . . . .	11
3.4. Seguimiento de pasillo . . . . .	13
3.5. Seguimiento de persona . . . . .	14
<b>4. Implementación en ROS y modelado en Gazebo</b>	<b>17</b>
4.1. Entorno de simulación en Gazebo . . . . .	17
4.1.1. Mapa de simulación . . . . .	17
4.1.2. Modelo de persona . . . . .	17
4.1.3. MIT RACECAR . . . . .	18
4.2. Modelo cinemático de un vehículo con sistema de dirección tipo Ackermann	19
4.3. Zona de seguridad . . . . .	21
4.4. Nodo de seguimiento en ROS . . . . .	22
4.5. Archivos de configuración . . . . .	23
4.6. Archivos de lanzamiento . . . . .	25
4.7. Ejecución mediante la terminal . . . . .	26
<b>5. Experimentación y resultados</b>	<b>27</b>
5.1. Ajuste de parámetros de los métodos de seguimiento . . . . .	27
5.1.1. Ajuste del parámetro $d_w$ en el seguimiento de pared . . . . .	27
5.1.2. Ajuste del parámetro <i>lookahead L</i> . . . . .	28
5.1.3. Ajuste del parámetro $d_f$ en el seguimiento de persona . . . . .	30
5.2. Análisis de resultados . . . . .	30

## ÍNDICE DE CONTENIDOS

---

<b>6. Conclusiones y futuras líneas de trabajo</b>	<b>35</b>
6.1. Conclusiones . . . . .	35
6.2. Futuras líneas de trabajo . . . . .	35
<b>Apéndice A. Códigos</b>	<b>39</b>
A.1. Nodo de seguimiento . . . . .	39
A.2. Plugin para mover el cilindro . . . . .	47
A.3. Configuración del control remoto en simulación . . . . .	49
A.4. Archivo de configuración .world de Gazebo . . . . .	51
<b>Apéndice B. Introducción a ROS</b>	<b>53</b>
B.1. Introducción . . . . .	53
B.2. Configuración . . . . .	53
B.3. Instalación . . . . .	54
B.4. Creación del entorno de trabajo . . . . .	54
B.5. Funcionamiento básico . . . . .	55
<b>Apéndice C. Puesta en marcha del UMA RACECAR</b>	<b>57</b>
C.1. Introducción . . . . .	57
C.2. Guía de arranque del UMA RACECAR . . . . .	57
C.3. Conexión entre el equipo de trabajo y el UMA RACECAR . . . . .	58
C.4. Teleoperación con mando . . . . .	59
<b>Apéndice D. Software adicional</b>	<b>61</b>
D.1. Visual Studio Code . . . . .	61
D.2. PlotJuggler . . . . .	61
D.3. RQt . . . . .	62
<b>Bibliografía</b>	<b>62</b>

---

## Índice de figuras

---

2.1. UMA RACECAR. . . . .	5
2.2. Telémetro bidimensional “Hokuyo UTM-30LX”. Fuente: [7] . . . . .	5
3.1. Representación gráfica del seguimiento mediante persecución pura. Fuente: [8] (modificado) . . . . .	9
3.2. Representación gráfica del seguimiento de pared. Fuente: [8] (modificado) . . . . .	11
3.3. Diagrama gráfico de la geometría del seguimiento de pared. . . . .	12
3.4. Representación gráfica del seguimiento de pasillo. Fuente: [8] (modificado) . . . . .	13
3.5. Representación gráfica del seguimiento de persona. Fuente: [8] (modificado) . . . . .	15
3.6. Variación de velocidad en el seguimiento de persona. Fuente: [8] (modificado) . . . . .	16
4.1. Mapa de simulación. . . . .	18
4.2. Modelo de simulación para la persona. . . . .	18
4.3. MIT RACECAR. . . . .	19
4.4. Cinemática de Ackermann. Fuente: [14] . . . . .	19
4.5. Estructura de un mensaje de tipo “AckermannDriveStamped” . . . . .	22
4.6. Estructura de un mensaje de tipo “LaserScan” . . . . .	23
5.1. Variación de $d_w$ en el seguimiento de pared. . . . .	28
5.2. Variación de $L$ en el seguimiento de pared. . . . .	29
5.3. Variación de $L$ en el seguimiento de pasillo. . . . .	29
5.4. Pruebas de simulación (a) y real (b) para el seguimiento de pasillo. . . . .	31
5.5. Mensajes terminal en el seguimiento de pasillo. . . . .	31
5.6. Pruebas de seguimiento de pared izquierda en simulación (a) y real (c), y de pared derecha en simulación (b) y real (d). . . . .	32
5.7. Mensajes terminal en el seguimiento de pared. . . . .	33
5.8. Pruebas de simulación (a) y real (b)(c) para el seguimiento de persona. . . . .	33
5.9. Mensajes terminal en el seguimiento de persona. . . . .	33
B.1. Esquema básico del funcionamiento de ROS. . . . .	56
C.1. Funcionalidades de los botones del joystick. . . . .	59



# CAPÍTULO 1

---

## Introducción

---

### 1.1. Antecedentes

El desarrollo de la tecnología de control autónomo de vehículos es un campo de trabajo joven de rápido progreso. Alcanza desde robots domésticos hasta escalas superiores como la conducción autónoma de automóviles y otros vehículos con pasajeros en entornos controlados.

Este proyecto se centrará en un sistema de navegación reactiva de un coche a escala robotizado desarrollado en varios Trabajos Fin de Grado realizados en el Departamento de Ingeniería de Sistemas y Automática [1][2]. Dicho vehículo dispone de un controlador electrónico de motores, un sensor inercial, una cámara y un telémetro láser bidimensional (2D). Este último sensor permite obtener medidas de distancia en un plano horizontal que se pueden utilizar para detectar elementos del entorno como paredes, pasillos, personas, etc.

El coche a escala dispone también de una placa de desarrollo Jetson TX2 con el metasisistema operativo ROS [3][4], con el que se puede interactuar con los sensores y actuadores a bordo, así como implementar estrategias de navegación. También se cuenta con un modelo del vehículo en Gazebo para el facilitar el desarrollo sin necesidad de usar el vehículo físico a cada paso.

### 1.2. Objetivo

El objetivo de este proyecto es implementar un método reactivo de planificación de caminos para el seguimiento de paredes, pasillos y personas en un coche a escala utilizando un telémetro láser 2D. Todo esto, implementado para poder comutar entre los distintos modos de funcionamiento.

### 1.3. Fases de trabajo

Para alcanzar el objetivo planteado, se seguirán las siguientes fases de actuación:

1. **Instalación de la plataforma y software de desarrollo.** Instalación del sistema operativo Ubuntu 18.04. y de la versión Melodic de ROS (compatible con el S.O.). También se instalará Gazebo para la simulación.
2. **Estudio bibliográfico**, tanto del Sistema Operativo Robótico (ROS) y del simulador Gazebo, como de los métodos de planificación reactiva de caminos.
3. **Desarrollo del método reactivo** de planificación y seguimiento de caminos para un coche a escala en el simulador.
4. **Implementación en el robot**, pruebas y análisis de resultados.
5. **Transferencia al vehículo real** del método de navegación reactiva, pruebas y análisis de resultados.
6. **Documentación de la memoria del proyecto.**

### 1.4. Estructura de la memoria

La memoria del presente proyecto se encuentra dividida en 6 capítulos, 4 apéndices y la bibliografía. Todos estos siguen al primer capítulo, que es esta introducción, y se presentan a continuación:

En el capítulo 2, “Herramientas empleadas”, se detallan aquellas herramientas empleadas para la realización del proyecto, tanto software como hardware.

En el capítulo 3, “Planificación reactiva de caminos”, se desarrollan los fundamentos teóricos necesarios para la resolución de los objetivos.

En el capítulo 4, “Implementación en ROS y modelado en Gazebo”, se describe el proceso de implementación del modelo teórico en simulación y en el UMA RACECAR.

En el capítulo 5, “Experimentación y resultados”, se exponen los experimentos realizados y los resultados obtenidos que conducen a la resolución del proyecto.

En el capítulo 6, “Conclusiones y futuras líneas de trabajo”, se exponen las conclusiones del proyecto así como futuros trabajos derivados del mismo.

En los apéndices se han descrito tareas auxiliares que no se relacionan directamente con los objetivos propuestos en este trabajo, pero han sido indispensables para su realización.

En el apéndice A, “Scripts”, se incluyen los scripts necesarios para este proyecto.

En el apéndice B, “Introducción a ROS”, se detalla el funcionamiento básico de ROS.

En el apéndice C, “Puesta en marcha del UMA RACECAR”, se describe el proceso de arranque y funcionamiento del UMA RACECAR.

En el apéndice D, “Software adicional”, se describen las herramientas de software adicional utilizadas.

# CAPÍTULO 2

---

## Herramientas empleadas

---

### 2.1. ROS

ROS (Robot Operating System) es un meta-sistema operativo de código abierto constituido por un conjunto de librerías y herramientas que facilitan el desarrollo de aplicaciones para robots. ROS proporciona servicios como abstracción hardware, control de dispositivos a bajo nivel, implementación de funcionalidades comúnmente usadas en la robótica, comunicación entre procesos y manejo de paquetes. Esta plataforma es habitualmente usada para compartir código e ideas. Además, ROS proporciona soporte mediante documentación, tutoriales, foros [5] y una wiki propia [6].

La segunda fase de este proyecto consiste en el estudio bibliográfico del software empleado. Para el estudio de ROS se ha llevado a cabo la lectura del libro *A Gentle Introduction to ROS* [3], en el cual se explica el funcionamiento básico del sistema y algunos conceptos más avanzados que serán de utilidad. Para el resto del desarrollo del proyecto, se han consultado los distintos foros de respuestas así como su wiki.

En el apéndice B se puede consultar el funcionamiento de ROS con más detalle.

### 2.2. Gazebo

Gazebo es un entorno de simulación tridimensional que proporciona una plataforma para modelar, simular y visualizar robots en un entorno virtual. Esto permite a los usuarios controlar y supervisar los robots en tiempo real, así como desarrollar algoritmos de control y planificación de movimiento.

Gazebo también da soporte para sensores y actuadores, incluyendo modelos de una amplia variedad de sensores (como cámaras, láseres y sensores iniciales) y actuadores (como motores y articulaciones) que pueden ser configurados y utilizados para simular el comportamiento de los sistemas robóticos en condiciones diversas y realistas.

Además de ser un simulador, Gazebo es una colección de librerías de código abierto, diseñada para simplificar el desarrollo software. Cada librería tiene unas dependencias mínimas, lo que las hace ideales para el trabajo de simulación.

Open Robotics, junto con una comunidad de desarrolladores, lideran un proceso de mejora y mantenimiento de las librerías, asegurando la fiabilidad y confianza de las mismas.

Cada librería dentro de Gazebo ha sido diseñada para cumplir un propósito específico, estableciendo consistencia entre las librerías y simplificando la búsqueda de soluciones.

El desarrollo de Gazebo sigue dos principios de especial interés para el desarrollo de este Trabajo Fin de Grado:

- Generalidad: Gazebo no se especializa en un tipo de robot, sino que sus herramientas están diseñadas para ser aplicadas a la mayor variedad de casos posibles en robótica.
- Modularidad y flexibilidad: El usuario tiene poder de elección sobre las características de las que hace uso. Por ejemplo, el usuario puede optar por cargar una simulación sin físicas o con físicas alternativas. Es más, el usuario no necesita instalar, compilar ni conocer las físicas de Gazebo.

### 2.3. UMA RACECAR

El coche utilizado para la implementación de este proyecto ha sido el UMA RACECAR. Se trata de un coche a escala 1:10 robotizado desarrollado en dos Trabajos Fin de Grado. Uno de ellos se centró en la construcción y el control de bajo nivel [1] y el otro, en la incorporación y calibración de sensores [2].

El sistema mecánico del vehículo cuenta con motor, servo de dirección, eje de transmisión, diferenciales y amortiguadores. En cuanto al sistema de control, dispone de: una computadora de abordo Jetson TX2, una placa controladora de la velocidad y del servo de dirección, una batería LiPo para la placa controladora de la velocidad y una *PowerBank* para alimentar a la computadora de abordo.

Las medidas del chasis necesarias para el control del coche son:

- Longitud: 550 mm
- Anchura: 310 mm
- Distancia entre ejes delantero y trasero: 345 mm

Además, el vehículo tiene incorporado una serie de sensores. Estos son un sensor inercial, un telémetro bidimensional y una cámara de profundidad. En el caso de este proyecto solo se precisa del telémetro bidimensional.

En la Fig. 2.1, se puede ver el coche robotizado, UMA RACECAR, utilizado para el desarrollo de este Trabajo Fin de Grado.

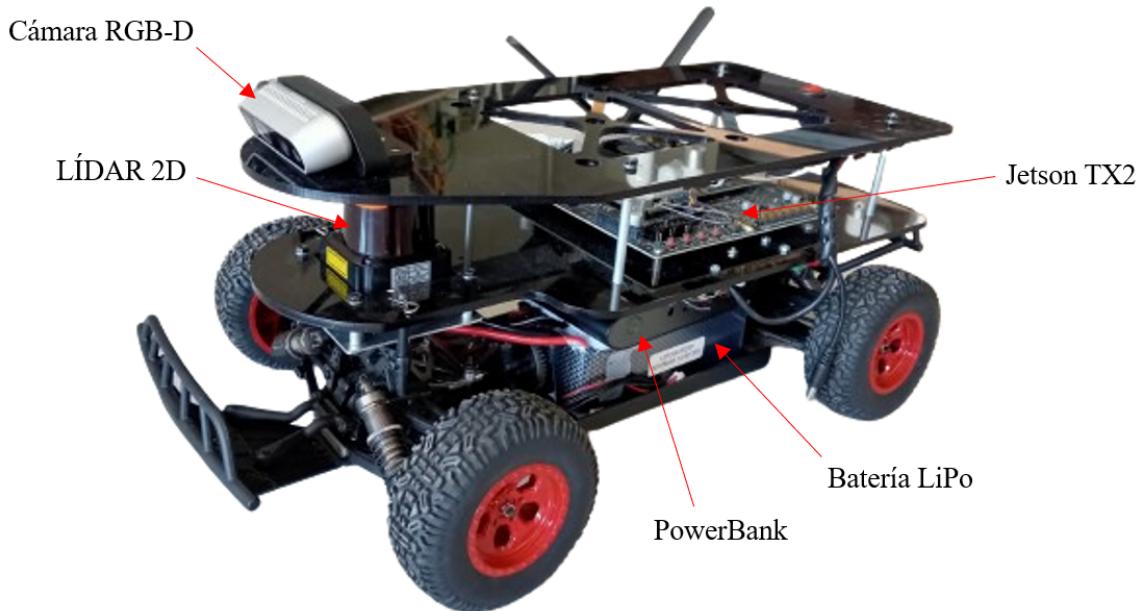


Figura 2.1: UMA RACECAR.

## 2.4. Láser 2D

Un telémetro bidimensional o láser 2D es un dispositivo capaz de medir distancias. Esta medida se obtiene mediante el envío de un pulso láser y la medición del tiempo que tarda en retornar al emisor. Estos pulsos los realiza en forma de barrido, obteniendo la distancia a todos los obstáculos del entorno que se encuentran dentro de su campo de visión.

El telémetro bidimensional incorporado es el “Hokuyo UTM-30LX”, que cuenta con las siguientes características:

- Campo de visión de  $270^\circ$   $[-135, 135]$ .
- Rango de medida de 0,1 a 30 m.
- Precisión de 30 mm, garantizada en el rango de 0,1 a 10 m.



Figura 2.2: Telémetro bidimensional “Hokuyo UTM-30LX”. Fuente: [7]



# CAPÍTULO 3

---

## Planificación reactiva de caminos

---

### 3.1. Introducción

La planificación reactiva se refiere al control y toma de decisiones donde las acciones del robot se determinan en respuesta directa al entorno y a los estímulos percibidos en tiempo real. Esto quiere decir que, en lugar de seguir un plan predefinido, el robot reacciona continuamente a los cambios y obstáculos que encuentra durante su camino.

Este tipo de planificación se caracteriza por tener una rápida velocidad de respuesta y por la capacidad de afrontar entornos desconocidos o cambiantes. Este enfoque flexible se basa en la percepción directa y la retroalimentación sensorial para la toma de decisiones.

El objetivo del seguimiento de una ruta mediante planificación reactiva es la conducción autónoma de un robot mediante la generación continua de comandos de velocidad y dirección.

Algunos métodos de seguimiento se basan en teoría de control no lineal, como el control predictivo o el control difuso. Alternativamente, los métodos geométricos son más sencillos. En ellos se consideran relaciones geométricas entre la posición actual del vehículo y la ruta a seguir. En el caso de este Trabajo Fin de Grado, la planificación reactiva de caminos se ha realizado mediante el método geométrico de persecución pura [8].

El método de persecución pura, más conocido como *pure-pursuit* en inglés, es un algoritmo de control de trayectoria utilizado en robótica móvil y en sistemas de navegación autónoma. Este método se utiliza para guiar un vehículo hacia un objetivo específico, manteniendo una trayectoria suave y precisa. Para realizarlo, calcula un arco de circunferencia que une la posición actual del vehículo y un punto objetivo en la ruta. Este punto objetivo se elige a una distancia de búsqueda, que es la longitud de cuerda de este arco. Por ello, este método facilita el ajuste de la distancia de búsqueda. Otras ventajas son su simplicidad computacional y la ausencia de términos derivados en su expresión matemática.

Por otro lado, las rutas para robots pueden clasificarse como explícitas o implícitas. Una ruta explícita se define como una secuencia de puntos que están unidos por segmentos de línea recta. El cálculo del error de seguimiento implica la estimación en tiempo real de la posición del vehículo respecto a la ruta. En cambio, una ruta implícita está descrita por características del entorno que son perceptibles por sensores. En este caso, el cálculo

del error de seguimiento no requiere estimación de la posición global. Este enfoque permite un procesamiento reactivo, permitiendo realizar la planificación de caminos antes mencionada.

En un seguimiento se necesita detectar un objetivo al que dirigirse. Para hacerlo se ha utilizado el láser 2D. Se detectará este punto objetivo, de forma reactiva, mediante la interpretación de los datos de distancia del láser.

La ventaja de este algoritmo es la posibilidad de realizar la persecución pura para el seguimiento tanto de rutas implícitas como explícitas. Además, facilita la aplicación de la planificación reactiva de caminos para realizar el seguimiento de paredes, pasillos y personas.

## 3.2. Algoritmo de persecución pura

En la realización del seguimiento de caminos implícitos se han seguido una serie de pasos. Lo primero será la selección del objetivo, esto se hará mediante la lectura de los datos del láser. La conmutación entre los distintos tipos de caminos implícitos a seguir se hará utilizando un *joystick*. Este proceso se explicará en el siguiente capítulo, donde se desarrollará el modelo.

El siguiente paso es calcular la trayectoria hacia el objetivo seleccionado. Una vez que se ha seleccionado el objetivo, el algoritmo de persecución pura calcula una trayectoria que guiará al vehículo hacia el objetivo de forma suave y eficiente. Posteriormente, en este capítulo, se desarrollará el algoritmo específico para cada tipo camino implícito.

Por último, se debe realizar el control de dirección. El algoritmo ajusta continuamente la dirección del vehículo para mantenerlo en la trayectoria adecuada. Esto se logra mediante el cálculo del ángulo de dirección correcto para el seguimiento de la trayectoria. Este cálculo de dirección se realizará de forma continua para seguir la trayectoria y mantenerse cerca del objetivo. Esta acción se realiza en tiempo real, a medida que el vehículo avanza, utilizando la retroalimentación del sensor láser del vehículo para corregir cualquier desviación de la trayectoria deseada.

Por tanto, las órdenes que se mandarán al robot son el ángulo de las ruedas de dirección y la velocidad lineal adecuada para el seguimiento del objetivo.

El algoritmo de persecución pura, como se ha indicado anteriormente, define la ruta hacia el objetivo mediante un arco de circunferencia entre la posición actual del vehículo y un punto objetivo en la ruta, cuya longitud de cuerda es la distancia de búsqueda  $L$  (término más usado en inglés como *lookahead*). En la Fig. 3.1 se puede observar gráficamente cómo se realiza.

En todos los cálculos realizados en este proyecto, se han utilizado los ejes como lo hace ROS. Estos ejes, como se observará en todas las representaciones gráficas, están rotados 90° en sentido antihorario respecto a los utilizados comúnmente.

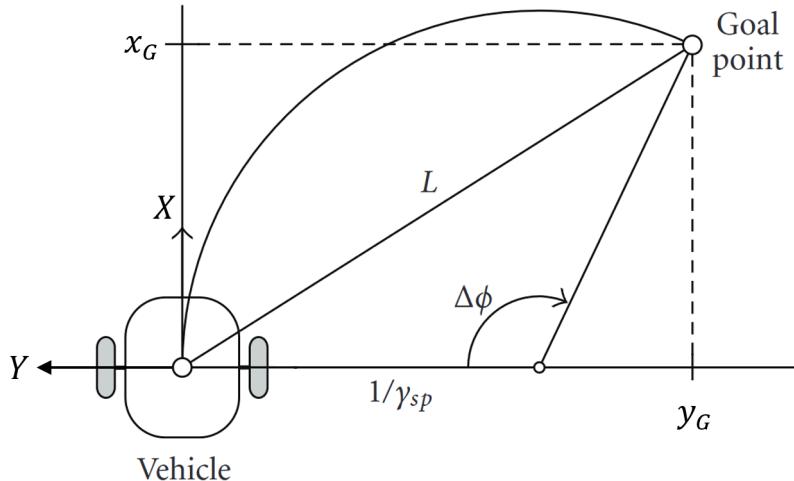


Figura 3.1: Representación gráfica del seguimiento mediante persecución pura. Fuente: [8] (modificado)

La curvatura de un vehículo se puede definir:

- como la inversa de la distancia  $r$  entre el origen del vehículo y su centro instantáneo de rotación (CIR), es decir, la inversa del radio de giro,
- como el cambio instantáneo de orientación del vehículo  $d\phi$  con respecto a la distancia recorrida  $ds$ , y
- como la variación de la velocidad angular respecto a la lineal.

$$\gamma = \frac{1}{r} = \frac{d\phi}{ds} = \frac{w}{v} \quad (3.1)$$

Las coordenadas locales del punto objetivo ( $x_G, y_G$ ) se calculan geométricamente mediante relaciones trigonométricas, a partir de la Fig. 3.1:

$$x_G = \frac{\sin \Delta\phi}{\gamma_{sp}} \quad y_G = \frac{1 - \cos \Delta\phi}{\gamma_{sp}} \quad (3.2)$$

La curvatura del punto de referencia ( $\gamma_{sp}$ ), que se tomará para obtener la velocidad angular deseada, se calcula a partir de la distancia de búsqueda como:

$$L^2 = x_G^2 + y_G^2 = \frac{2 - 2 \cdot \cos(\Delta\phi)}{\gamma_{sp}^2} = \frac{2 \cdot y_G}{\gamma_{sp}} \quad (3.3)$$

$$\gamma_{sp} = \frac{2 \cdot y_G}{L^2} \quad (3.4)$$

Esta ecuación conforma la ley de control de curvatura, donde  $y_G$  es el error de señal y  $2/L^2$  es la ganancia del controlador.

Demostración de la Ec. 3.3:

$$\begin{aligned}
 L^2 = x_G^2 + y_G^2 &= \frac{(\sin \Delta\phi)^2}{\gamma_{sp}^2} + \frac{(1 - \cos \Delta\phi)^2}{\gamma_{sp}^2} = \frac{(\sin \Delta\phi)^2 + (1 - \cos \Delta\phi)^2}{\gamma_{sp}^2} = \\
 &= \frac{(\sin \Delta\phi)^2 + 1 - 2 \cdot \cos \Delta\phi + (\cos \Delta\phi)^2}{\gamma_{sp}^2} = \{(\sin \Delta\phi)^2 + (\cos \Delta\phi)^2 = 1\} = \\
 &= \frac{1 - 2 \cdot \cos \Delta\phi + 1}{\gamma_{sp}^2} = \frac{2 - 2 \cdot \cos \Delta\phi}{\gamma_{sp}^2} = \frac{2 \cdot (1 - \cos \Delta\phi)}{\gamma_{sp}^2} = \frac{2 \cdot \frac{(1 - \cos \Delta\phi)}{\gamma_{sp}}}{\gamma_{sp}} = \\
 &= \frac{2 \cdot y_G}{\gamma_{sp}}
 \end{aligned} \tag{3.5}$$

Por tanto, si relacionamos las Ec. 3.1 y 3.4, obtenemos la velocidad angular que debe tener el vehículo para seguir el punto objetivo indicado.

$$w = \gamma \cdot v = \frac{2 \cdot y_G \cdot v}{L^2} \tag{3.6}$$

Finalmente, se calcula el ángulo de giro de las ruedas de dirección del vehículo necesario para el seguimiento del objetivo. Utilizando el modelo cinemático para vehículos con direccionamiento tipo Ackermann, las velocidades angular y lineal, y la distancia entre los ejes delantero y trasero ( $l$ ), se obtiene el ángulo de dirección:

$$\psi = \arctan\left(\frac{\omega \cdot l}{v}\right) \tag{3.7}$$

Sustituyendo el valor de la velocidad angular, quedaría:

$$\psi = \arctan\left(\frac{\frac{2 \cdot y_G \cdot v}{L^2} \cdot l}{v}\right) = \arctan\left(\frac{2 \cdot y_G \cdot v \cdot l}{v \cdot L^2}\right) = \arctan\left(\frac{2 \cdot y_G \cdot l}{L^2}\right) \tag{3.8}$$

El cálculo del ángulo de dirección se ha desarrollado primero ya que es común a todos los casos de seguimiento.

A continuación, se va a detallar el algoritmo para el cálculo del punto objetivo en cada opción de seguimiento. Además, se especificará la velocidad lineal necesaria en cada caso, que es el otro comando a enviar al robot.

Los tres casos de seguimiento que se van a desarrollar son:

- Seguimiento de pared (apartado 3.3)
- Seguimiento de pasillo (apartado 3.4)
- Seguimiento de persona (apartado 3.5)

En todos ellos, se ha tomado la ruta como implícita. Estas rutas irán describiéndose de forma dinámica, adaptándose al entorno gracias a los datos del láser 2D.

### 3.3. Seguimiento de pared

El camino implícito en el seguimiento de pared se define mediante una línea paralela a la pared. Esta línea estará a una distancia  $d_w$  de la pared, parámetro que se ajustará posteriormente. El punto objetivo al que se guiará el vehículo debe estar contenido en esta paralela, y a una distancia  $L$  (distancia de búsqueda) del coche.

En esta opción de seguimiento tenemos dos casos donde la pared puede ser la izquierda o la derecha. El algoritmo será el mismo en ambos casos, ya que la única diferencia entre ambos es la dirección de giro. Para solucionarlo, en las ecuaciones aparece el signo del giro, que cambiará dependiendo del ángulo, positivo o negativo, que capte el láser.

El primer paso, que no es común a ambos casos, se trata de obtener los datos del sensor. En el caso de la pared derecha se tomarán los datos del láser en la parte derecha, de  $-135$  a  $0^\circ$ . Para la pared izquierda, los ángulos varían de  $0$  a  $135^\circ$ . El resto de pasos es común a ambos casos.

Lo siguiente será buscar el valor mínimo dentro del vector de rangos, vector correspondiente al barrido del láser. Este valor será el que llamaremos  $\rho$ . El otro valor que obtenemos del láser es el ángulo en el que se encuentra el punto objetivo,  $\theta$ . Este ángulo lo sacamos del vector de ángulos, buscando el mismo índice que tiene el rango mínimo.

En la Fig. 3.2 se puede ver el planteamiento visual del algoritmo que se va a emplear.

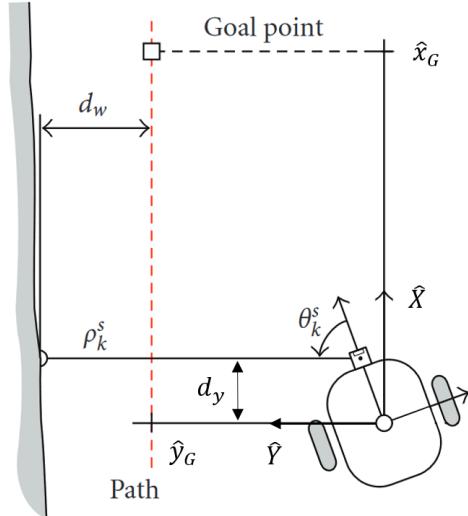


Figura 3.2: Representación gráfica del seguimiento de pared. Fuente: [8] (modificado)

La ruta implícita es paralela al eje de coordenadas  $\hat{X}$ , por lo que se puede definir la ruta mediante su coordenada en el eje  $\hat{Y}$ . El cálculo de esta coordenada ( $y_G$ ) en el sistema de referencia global se hará según la coordenada en el sistema de referencia local del vehículo ( $\hat{y}_p$ ).

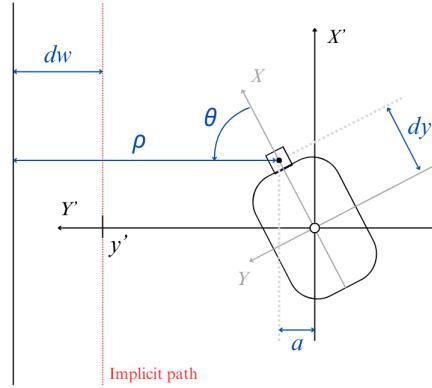


Figura 3.3: Diagrama gráfico de la geometría del seguimiento de pared.

En la Fig. 3.3, se puede ver un diagrama mediante el cual se resuelve geométricamente el valor de la coordenada local  $y'$  (también denominada  $\hat{y}_p$ ). En ella aparece una constante  $d_y$  que representa la distancia entre el centro del robot y el centro del láser. Además, aparece un valor  $a$  que se tomará como variable auxiliar para los cálculos.

Geométricamente, se puede seguir el siguiente razonamiento:

1.  $\cos \theta = \frac{a}{d_y} \Rightarrow a = d_y \cdot \cos \theta$
2. Por otro lado:  $d_w + y = \rho + a \Rightarrow y = \rho + a - d_w$
3. Sustituyendo el valor de  $a$ :  $y = \rho + d_y \cdot \cos \theta - d_w$

Con esto queda demostrado que el valor de la coordenada local, en el eje  $\hat{Y}$ , de la ruta es:

$$\hat{y}_p = \text{sign}(\theta_k^s) \cdot (\rho_k^s + d_y \cdot \cos(\theta_k^s) - d_w) \quad (3.9)$$

Si  $|\hat{y}_p| < L$ , las coordenadas locales del punto objetivo se calcularán según:

$$\begin{aligned} \hat{y}_G &= \hat{y}_p \\ \hat{x}_G &= \sqrt{L^2 - (\hat{y}_p)^2} \end{aligned} \quad (3.10)$$

En otro caso, significa que el robot está muy alejado, por tanto:

$$\begin{aligned} \hat{y}_G &= L \cdot \text{sign}(\hat{y}_p) \\ \hat{x}_G &= 0 \end{aligned} \quad (3.11)$$

Estas coordenadas locales deben rotarse para obtener sus correspondientes coordenadas globales.

$$y_G = \text{sign}(\theta_k^s) \cdot (\sin(\theta^c), \cos(\theta^c)) \cdot \begin{bmatrix} \hat{y}_G \\ -\hat{x}_G \end{bmatrix} = \text{sign}(\theta_k^s) \cdot (\hat{y}_G \cdot \sin(\theta_k^s) - \hat{x}_G \cdot \cos(\theta_k^s)) \quad (3.12)$$

Por último, sustituyendo este valor en la Ec. 3.8, obtenemos el ángulo de dirección que debe tomar el robot para llegar al punto objetivo. En este tipo de seguimiento, la velocidad lineal es un valor constante que se especifica como parámetro de diseño y que se elegirá dependiendo de  $d_w$ .

### 3.4. Seguimiento de pasillo

En el seguimiento de pasillo se sigue un camino implícito definido por la línea media de dicho pasillo. Este se irá modificando continuamente para poder adaptarse a los cambios que puedan aparecer en la anchura o la dirección del pasillo conforme se va circulando. El punto de destino al que se dirigirá el vehículo, debe pertenecer a esta línea media y a una distancia de búsqueda  $L$ .

La línea media se calcula como la media aritmética de las mínimas distancias a ambas paredes (Ec. 3.15). El coche obtiene estas medidas mediante el láser 2D. Se hacen dos barridos; uno de  $-135$  a  $0^\circ$  con el que se obtiene el índice con rango mínimo del lado derecho del coche; y otro de  $0$  a  $135^\circ$ , consiguiendo el mínimo izquierdo. Con estos índices, se pueden buscar tanto la mínima distancia ( $\rho_k$ ) dentro del vector de rangos, como su respectivo ángulo ( $\theta_k$ ) dentro del vector de ángulos.

En la Fig. 3.4 está representado el planteamiento del problema.

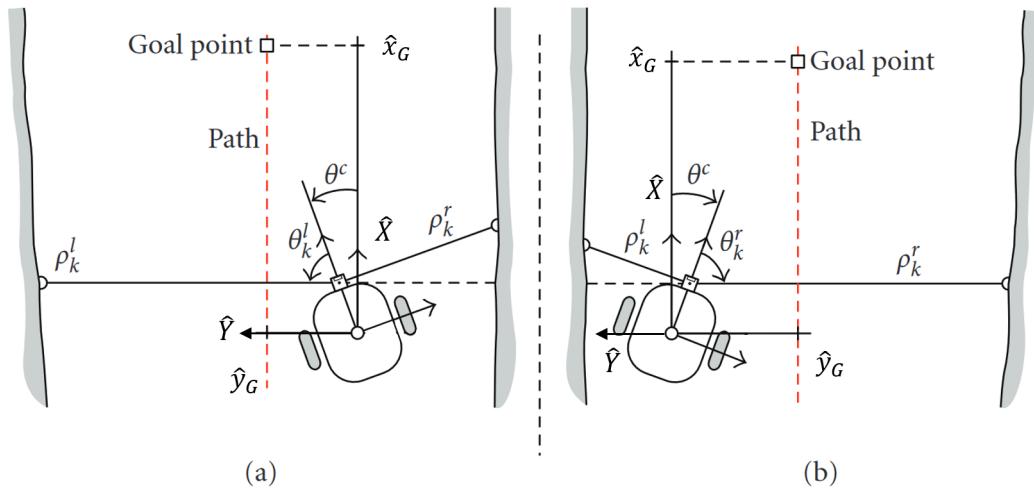


Figura 3.4: Representación gráfica del seguimiento de pasillo. Fuente: [8] (modificado)

La orientación global del vehículo se obtendrá como la suma de ambos ángulos de rango mínimo.

$$\theta^c = \theta_k^l + \theta_k^r \quad (3.13)$$

Una vez conocidos los rangos mínimos y sus respectivos ángulos, se deben transformar estos puntos a coordenadas cartesianas:

$$\begin{aligned} \hat{y}^l &= \rho_k^l + d_y \cdot \cos(\theta_k^l) \\ \hat{y}^r &= -\rho_k^r - d_y \cdot \cos(\theta_k^r) \end{aligned} \quad (3.14)$$

Con estos valores se obtiene la coordenada de la línea media:

$$\hat{y}_p = \frac{\hat{y}^l + \hat{y}^r}{2} \quad (3.15)$$

Las coordenadas locales del punto de destino se determinan dependiendo de la comparación entre la coordenada de la línea media y la distancia de búsqueda. Si  $|\hat{y}_p| < L$ , se

puede calcular el punto objetivo sobre la línea media según:

$$\begin{aligned}\hat{y}_G &= \hat{y}_p \\ \hat{x}_G &= \sqrt{L^2 - (\hat{y}_p)^2}\end{aligned}\tag{3.16}$$

En otro caso, el robot está muy alejado y se establece:

$$\begin{aligned}\hat{y}_G &= L \cdot \text{sign}(\hat{y}_p) \\ \hat{x}_G &= 0\end{aligned}\tag{3.17}$$

Con estas coordenadas locales y la orientación global del vehículo se consigue la referencia global del punto de destino.

$$y_G = (\sin(\theta^c), \cos(\theta^c)) \cdot \begin{bmatrix} \hat{x}_G \\ \hat{y}_G \end{bmatrix} = \hat{x}_G \cdot \sin(\theta^c) + \hat{y}_G \cdot \cos(\theta^c)\tag{3.18}$$

Finalmente, se obtiene el ángulo de dirección, a partir de esta coordenada global, como se mostraba previamente en la Ec. 3.8.

En el caso del seguimiento de pasillo, la velocidad lineal se fija en función del ancho del pasillo que se vaya a recorrer. Como se plantea el problema para cualquier pasillo, se fijará la velocidad lineal a 0,8 m/s para poder adaptarse a todas las posibilidades.

### 3.5. Seguimiento de persona

El seguimiento de persona tiene una peculiaridad respecto a los otros métodos. En este tipo de seguimiento el camino implícito es dinámico, pues variará según el movimiento de la persona a seguir. La ruta implícita será, por tanto, los puntos de posición de la persona. Todo esto hace que la distancia de búsqueda  $L$  no sea un parámetro constante.

Inicialmente, se debe buscar la persona objetivo a seguir. Esto se hará buscando un objeto en el eje longitudinal del láser ( $0^\circ$ ). Además, se limitará la búsqueda a una distancia de 0,8 m, pero con un margen de 0,54 m. Por tanto, la búsqueda estará en el rango de distancias:

$$\begin{aligned}(d_s - d_{th}) &\leq \rho_k \leq (d_s + d_{th}) \\ (0,8 - 0,54) &\leq \rho_k \leq (0,8 + 0,54) \\ 0,26 &\leq \rho_k \leq 1,34\end{aligned}\tag{3.19}$$

Una vez detectado un objeto en esa posición, obtenemos sus coordenadas, que serán las coordenadas iniciales del punto objetivo.

$$\begin{aligned}x_k &= d_y + \rho_k \cdot \cos(\theta_k) \\ y_k &= \rho_k \cdot \sin(\theta_k)\end{aligned}\tag{3.20}$$

Se irá escaneando alrededor de las posiciones de la persona para encontrar el siguiente punto objetivo. Si la distancia euclídea  $d_k$  (Ec. 3.21) entre ellos es menor o igual que el

umbral  $d_{th}$ , el último punto detectado pasará a ser el objetivo. En caso contrario, se habrá perdido el objetivo, se parará el coche y se tendrá que comenzar la búsqueda de nuevo.

$$d_k = \sqrt{(x_k - x_{k-1})^2 + (y_k - y_{k-1})^2} \quad (3.21)$$

En el caso de haber encontrado este nuevo punto objetivo, se aplicará el algoritmo de persecución pura.

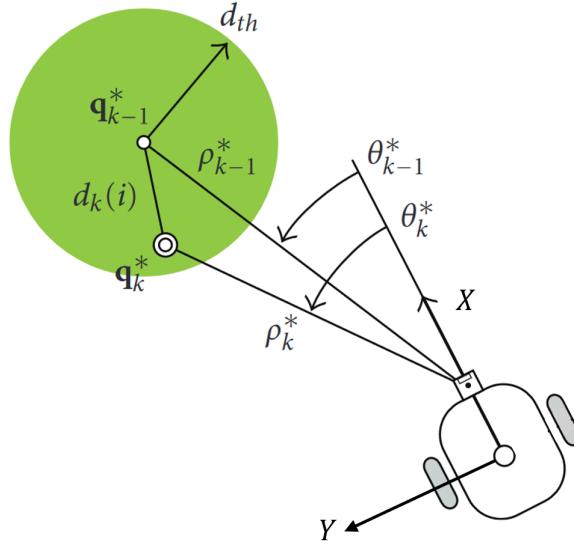


Figura 3.5: Representación gráfica del seguimiento de persona. Fuente: [8] (modificado)

Para calcular el ángulo de dirección haremos uso de la Ec. 3.8, donde:

$$y_G = \rho_k \cdot \sin(\theta_k) \quad (3.22)$$

$$L = \sqrt{\rho_k \cdot (2 \cdot d_y \cdot \cos(\theta_k) + \rho_k)} \quad (3.23)$$

El cálculo de la velocidad lineal  $v_{sp}$  dependerá de la distancia a la que se encuentre la persona, cumpliendo con la siguiente condición:

$$v_{sp} = \begin{cases} 0, & \text{si } \rho_k < d_s \\ \frac{\rho_k - d_s}{d_f - d_s} \cdot v_{max}, & \text{si } d_s \leq \rho_k \leq d_f \\ v_{max}, & \text{si } \rho_k > d_f \end{cases} \quad (3.24)$$

Donde la velocidad máxima será aproximadamente 1 m/s.

En la Fig. 3.6 se ve visualmente los rangos de velocidad.

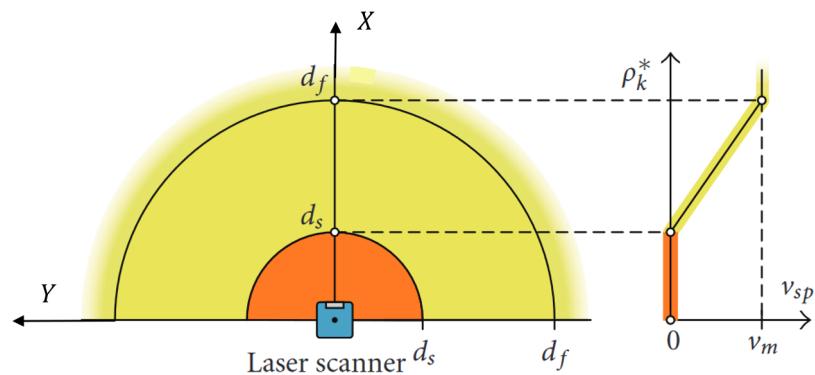


Figura 3.6: Variación de velocidad en el seguimiento de persona. Fuente: [8] (modificado)

# CAPÍTULO 4

---

## Implementación en ROS y modelado en Gazebo

---

En el capítulo anterior se ha planteado el modelo teórico para el seguimiento de caminos implícitos utilizando el método de persecución pura. En este, se presenta la implantación en ROS y la creación de un entorno de simulación en Gazebo. Para este entorno, será necesario crear un mapa con las paredes a seguir, un objeto que simule a una persona e integrar el coche.

### 4.1. Entorno de simulación en Gazebo

#### 4.1.1. Mapa de simulación

Se ha creado un mapa adaptado para poder simular los distintos modos de seguimiento. Tiene dos zonas con paredes irregulares con obstáculos para el seguimiento de pared, tanto derecha como izquierda. Además, incluye un pasillo con cambio de dirección para probar el seguimiento del mismo, y una zona amplia para el seguimiento de persona. En la Fig. 4.1 se muestra el mapa completo con las zonas diferenciadas.

El mapa se ha creado con la interfaz gráfica de Gazebo utilizando la funcionalidad *Building Editor*.

#### 4.1.2. Modelo de persona

Para la creación de un modelo que simule a una persona se ha optado por un cilindro al ser la figura geométrica que más se asemeja al cuerpo humano. Se ha tomado como referencia el diseño de *The Construct* [9]. Este diseño incluye el modelo físico (Fig. 4.2) y la configuración para su movimiento.

La descripción física del modelo se ha editado para asemejarse en el alto y ancho a una persona, unificando este archivo con el de configuración física del mapa. En cuanto a la configuración de la cinemática, se ha modificado el plugin que controla el cilindro (incluido en el apéndice A.2) para que se suscriba a un tópico (*"cmd\_vel\_object"*) de movimiento distinto al del coche.

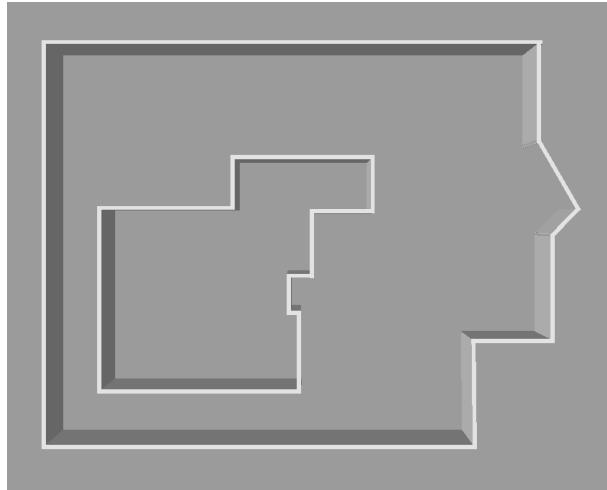


Figura 4.1: Mapa de simulación.

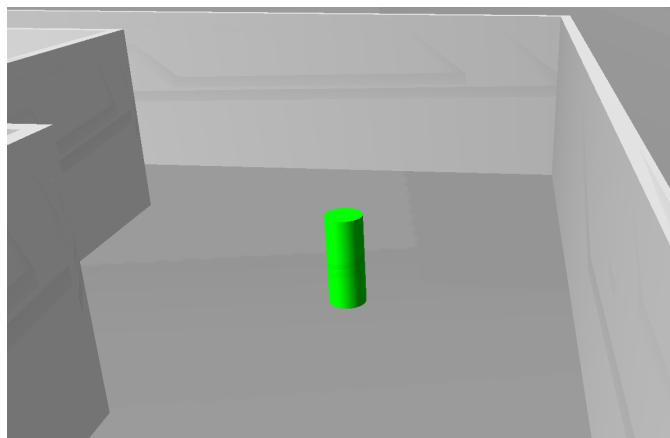


Figura 4.2: Modelo de simulación para la persona.

#### 4.1.3. MIT RACECAR

El coche UMA RACECAR, mencionado en el capítulo 2.3, se diseñó y construyó en base al coche fabricado por un equipo del Instituto de Tecnología de Massachusetts (MIT), el *MIT RACECAR* [10]. Por tanto, en este Trabajo Fin de Grado se va a hacer uso del proyecto que realizó el MIT para la simulación.

Se ha ido realizando el estudio de este proyecto a lo largo de todo el desarrollo, con el fin de poder implementarlo y hacer las modificaciones necesarias para el funcionamiento.

El proyecto del *MIT RACECAR* se puede encontrar como un repositorio en *Github* [11]. De los paquetes que conforman el proyecto se va a hacer uso principalmente de tres:

- racecar: incluye los scripts, nodos, ficheros de lanzamiento y de parámetros relacionados con el robot real.
- racecar\_gazebo: incluye los scripts, nodos, ficheros de lanzamiento y de parámetros relacionados con el robot en simulación.
- vesc: incluye la configuración y los ficheros de funcionamiento de la controladora del robot real.

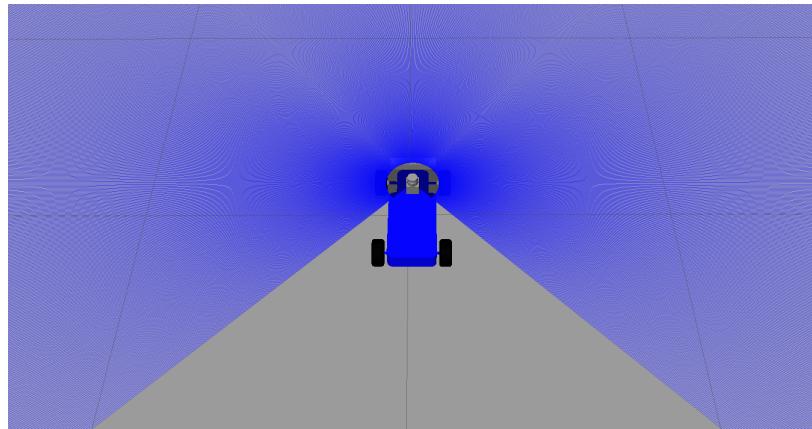


Figura 4.3: MIT RACECAR.

## 4.2. Modelo cinemático de un vehículo con sistema de dirección tipo Ackermann

El vehículo del proyecto *MIT RACECAR* está modelado en base a la cinemática de Ackermann [12][13].

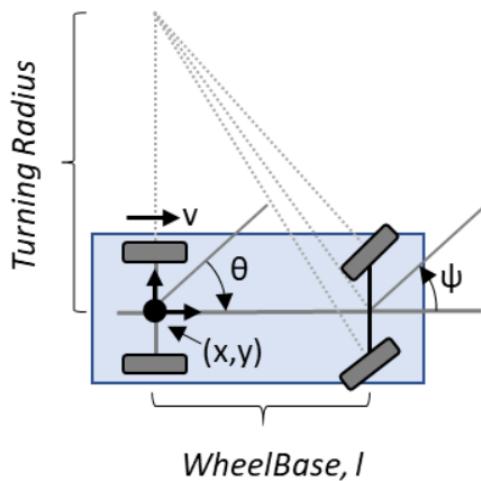


Figura 4.4: Cinemática de Ackermann. Fuente: [14]

El vehículo, como se observa en la Fig. 4.4, solo tiene las ruedas delanteras directrices. En el modelo cinemático de Ackermann el coche gira respecto a un centro instantáneo de rotación alineado con el eje trasero. Por tanto, el ángulo de giro de las ruedas ( $\psi$ ) no será el mismo que la orientación del vehículo ( $\theta$ ).

De acuerdo con este modelo, el coche es de tipo no holonómico. Esto quiere decir que tiene restricciones de movimiento, no pudiendo moverse libremente en todas las direcciones del espacio ni cambiar su orientación de manera independiente.

Estas restricciones de movimiento se han tenido en cuenta para calcular el giro máximo que permite la geometría del robot según el modelo cinemático de Ackermann.

La configuración de vehículo se define mediante el estado del modelo de tracción diferencial:

$$\dot{q} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ \tan(\psi)/l & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v \\ w \end{bmatrix}, \quad (4.1)$$

donde,  $(x, y)$  es la localización respecto de un sistema de referencia global del punto medio del eje trasero,  $\theta$  es la orientación del vehículo respecto de dicho sistema de referencia,  $\psi$  es el ángulo de dirección del vehículo (la orientación de las ruedas respecto del sistema de referencia local del vehículo),  $l$  es la distancia entre ejes y  $v$  la velocidad longitudinal del vehículo.

A partir de la Ec. 4.1 se puede obtener el radio de giro mínimo según se muestra a continuación.

La derivada de la dirección de giro es la velocidad angular,  $\dot{\theta} = w$ . Por tanto, se obtiene el valor de la velocidad angular respecto del ángulo de giro y viceversa:

$$w = \dot{\theta} = v \cdot \frac{\tan(\psi)}{l} \quad \Rightarrow \quad \psi = \arctan\left(\frac{w \cdot l}{v}\right) \quad (4.2)$$

El radio de giro mínimo que puede realizar el coche se obtendrá cuando el ángulo de dirección sea máximo, por tanto:

$$w_{max} = v \cdot \frac{\tan(\psi_{max})}{l}, \quad r_{min} = \frac{v}{w_{max}} \quad \Rightarrow \quad r_{min} = \frac{l}{\tan(\psi_{max})} \quad (4.3)$$

Físicamente, el máximo ángulo que giran las ruedas, tanto en simulación como en el coche real, es aproximadamente de  $\pm 20^\circ$ . Además, como se ha visto anteriormente que  $l = 0,345$  m, en el UMA RACECAR, por tanto:

$$r_{min} = \frac{0,345 \text{ m}}{\tan(20^\circ)} = 0,947 \text{ m} \quad (4.4)$$

Este radio de giro permite calcular la mínima distancia a la que necesita estar el coche para poder realizar un giro sin chocar con una pared que tenga delante, teniendo en cuenta las dimensiones del mismo.

### 4.3. Zona de seguridad

Se han tomado una serie de restricciones para que el coche pueda realizar el seguimiento sin producirse ningún choque. La primera de ellas es una distancia mínima a cualquier objeto. Se ha configurado el modo autónomo para que el vehículo se detenga si se acerca a una distancia menor de 0,25 m de cualquier objeto/persona. Además, se han restringido individualmente los modos de seguimiento de pared y pasillo.

En el seguimiento de pared se ha tenido en cuenta la posibilidad de encontrar una pared en el frontal del coche. En este caso, se debe conseguir que el vehículo haga el cambio de dirección sin chocar con esa pared y, además, respetando el seguimiento de la ruta implícita. Para ello, se ha declarado una zona de seguridad definida como la suma del radio mínimo de giro y la distancia a la pared que mantiene la ruta.

$$sz = d_w + r_{min} \quad (4.5)$$

Si el vehículo entra en esta zona de seguridad se le indica que debe girar con la máxima velocidad angular para poder esquivar el obstáculo.

En el seguimiento de pasillo se procede de forma similar. Cuando se encuentre con una pared frontal debe girar a máxima velocidad. Al igual que en el seguimiento de pared, se debe respetar la ruta implícita. Por tanto, en este caso la zona de seguridad se tomará respecto a la línea media del pasillo.

$$sz = \left( \frac{\rho^r + \rho^l}{2} \right) + r_{min} \quad (4.6)$$

Cuando el vehículo entre en esta zona de seguridad debe comprobar hacia qué lado se dirige la curva y, en función de esto, girar para el lado correspondiente:

- Si  $\rho^l > \rho^r \rightarrow \omega = \omega_{max}$  (Giro hacia la izquierda)
- Si  $\rho^l < \rho^r \rightarrow \omega = -\omega_{max}$  (Giro hacia la derecha)

## 4.4. Nodo de seguimiento en ROS

Los modos de seguimiento descritos anteriormente se han implementado en un nodo de ROS. Este nodo está escrito en lenguaje de programación *Python*, y se ejecutará cuando se active el modo de funcionamiento autónomo con el *joystick*.

El programa realiza una serie de funciones:

- Se suscribe al tópico donde publica el láser (“/scan”), para leer continuamente los datos del sensor.
- Se suscribe al tópico donde publica el *joystick* o mando controlador (“/vesc/joy”), para comprobar qué botón se ha presionado y entrar en el modo de funcionamiento correspondiente.
- Realizar el algoritmo de persecución pura del modo seleccionado, integrándole la cinemática de *Ackermann* y las zonas de seguridad anteriormente mencionadas.
- Por último, publica la velocidad lineal y el ángulo de dirección, calculados según el modo correspondiente, en el tópico “/vesc/high\_level/ackermann\_cmd\_mux/input/nav\_0” al cual está suscrito el nodo del coche.

El código completo se puede consultar en el apéndice A.1.

El controlador de bajo nivel del UMA RACECAR interpreta instrucciones en formato “*AckermannDriveStamped*” [14]. Este es el motivo por el cual el nodo del coche se suscribe al tópico “/vesc/high\_level/ackermann\_cmd\_mux/input/nav\_0”. Este tipo de mensaje tiene la siguiente estructura:

```
alba@ubuntu:~/catkin_racecar$ rosmsg info ackermann_msgs/AckermannDriveStamped
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
ackermann_msgs/AckermannDrive drive
  float32 steering_angle
  float32 steering_angle_velocity
  float32 speed
  float32 acceleration
  float32 jerk
```

Figura 4.5: Estructura de un mensaje de tipo “*AckermannDriveStamped*”.

Como se observa en el recuadro de la Fig. 4.5, la conducción del coche se puede realizar mediante los campos: ángulo de dirección, velocidad del ángulo de dirección, velocidad lineal, aceleración lineal y la tasa de cambio de la aceleración lineal. En el caso de este proyecto, los comandos que reciba el robot serán, como ya se ha indicado, el ángulo de dirección y la velocidad lineal.

Por otro lado, se necesita la información del láser para poder reconocer el entorno. El láser publica mensajes de tipo “*LaserScan*” [15] en el tópico “*/scan*”. Este tipo de mensaje tiene la estructura que aparece en la Fig. 4.6.

```
alba@ubuntu:~/catkin_racecar$ rosmsg info sensor_msgs/LaserScan
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Figura 4.6: Estructura de un mensaje de tipo “*LaserScan*”.

Un mensaje de este tipo proporciona, además de una cabecera, la siguiente información:

- Ángulo mínimo: ángulo en el que empieza el barrido del láser.
- Ángulo máximo: ángulo en el que finaliza el barrido del láser.
- Incremento de ángulo: incremento de ángulo para cada medición del barrido.
- Incremento de tiempo: incremento de tiempo entre mediciones.
- Tiempo de escaneo.
- Rango mínimo: mínima distancia a la que puede medir el láser.
- Rango máximo: máxima distancia a la que puede medir el láser.
- Rangos: vector con todos los valores de distancias del barrido.
- Intensidades: vector con todos los valores de intensidades del barrido.

## 4.5. Archivos de configuración

Una vez desarrollada toda la teoría e implementación que engloba el modelo de este proyecto, la siguiente fase es crear y modificar los archivos necesarios para poder comunicarse con el coche.

El primer paso para poder controlar el coche simulado es modificar el archivo de configuración para el funcionamiento teleoperado (incluido en el apéndice A.3). Inicialmente, se disponía solo de la configuración para teleoperar manualmente mediante control remoto. A la cual se le ha añadido el modo de funcionamiento autónomo. En este caso, el mando se utiliza para indicar que el coche está en modo autónomo y para mover el cilindro en la simulación, mediante la publicación de mensajes en el tópico “*cmd\_vel\_object*”. Esta funcionalidad se ha añadido al archivo mediante el siguiente fragmento de código.

```
1 autonomous_control:
2   type: topic
3   message_type: geometry_msgs / Twist
4   topic_name: cmd_vel_object
5   deadman_buttons: [5]
6   axis_mappings:
7     -
8       axis: 1
9       target: linear.x
10      scale: 1.0
11      offset: 0.0
12     -
13       axis: 0
14       target: linear.y
15       scale: 1.0
16       offset: 0.0
17
```

El otro archivo que hace falta modificar es el que define el *mundo* cuando se abre la simulación en Gazebo. El código completo se puede consultar en el script, con extensión .world, en el apéndice A.4.

Un archivo .world es un tipo de archivo utilizado principalmente en el contexto de simuladores de robots. En este archivo se define el entorno de simulación, incluyendo la configuración del *mundo*, los modelos, las propiedades físicas y otros elementos necesarios para la simulación. Los archivos .world suelen estar escritos en un formato de texto estructurado, comúnmente XML, e incluyen:

- Entorno físico: define el terreno, el cielo, la iluminación, y otras características del entorno.
- Modelos: especifica los modelos de objetos y robots que estarán presentes en la simulación, incluyendo sus propiedades físicas y visuales.
- Propiedades físicas: define parámetros físicos como la gravedad, la fricción, etc.
- Plugins: incluye plugins que añaden funcionalidades específicas a los elementos del mundo o al propio simulador.
- Sensores: configura los sensores que estarán disponibles para los robots.

El archivo original incluye la configuración de un entorno específico para el *MIT RACECAR*. Para poder adaptarlo, se ha cambiado el mapa del entorno original por el creado en este proyecto.

```
53 <uri>model://racecar_description/models/Mapa_TFG</uri>
```

Además, se ha añadido el modelo del cilindro, incluyendo sus propiedades físicas y el plugin que controla su movimiento.

## 4.6. Archivos de lanzamiento

Por último, para poder comenzar tanto las simulaciones como las pruebas reales, será necesario crear los archivos de lanzamiento .launch. Este tipo de archivos, en formato XML, se utilizan para definir y configurar el lanzamiento de uno o más nodos y otros parámetros necesarios para ejecutar una aplicación o simulación en ROS. Tienen una serie de funcionalidades:

- Lanzamiento de nodos (<node>): especifica qué nodos deben ser ejecutados, incluyendo la ruta a los ejecutables (*type*), en qué paquete se encuentran (*pkg*) y los parámetros necesarios, como el nombre del nodo (*name*) o dónde se muestran los mensajes de salida (*output*).
- Configuración de parámetros (<param>): permite definir parámetros en el servidor de parámetros de ROS, para que luego puedan ser utilizados por los nodos.
- Remapeo de tópicos (<remap>): permite cambiar el nombre de los tópicos sin necesidad de modificar el nombre local en el código de los nodos.
- Inclusión de otros archivos de lanzamiento (<include>): permite incluir otros archivos de lanzamiento, facilitando la ejecución simultánea de varios archivos y la reutilización de configuraciones.

Para la ejecución de este Trabajo Fin de Grado se han realizado dos archivos de lanzamiento. El script 4.1 incluye el archivo modificado para la simulación. En él se ha incluido el fichero de lanzamiento del entorno de Gazebo y el fichero de configuración para teleoperar con control remoto. También se ha añadido el lanzamiento del nodo de seguimiento.

Por otro lado, se ha creado un script para las pruebas reales con el UMA RACECAR (script 4.2). Este código solo incluye el archivo de configuración para el control remoto y el lanzamiento del nodo de seguimiento.

Script 4.1: gazebo\_sim\_joy.launch

```

1 <!-- -*- mode: XML -*- -->
2
3 <launch>
4
5   <remap from="/ackermann_cmd_mux/input/teleop" to="/racecar/
6     ackermann_cmd_mux/input/teleop"/>
7
8   <include file="$(find racecar_gazebo)/launch/racecar_tunnel.launch">
9     <!-- MAP -->
10
11  <include file="$(find racecar_control)/launch/teleop.launch">
12    <!-- TELEOPERATION -->
13
14  <node pkg="path_tracker" name="follower" type="follower.py" output=
15    "screen"/>    <!-- PATH TRACKING ALGORITHM-->
16
17 </launch>
```

Script 4.2: mi\_launch.launch

```
1 <?xml version='1.0'?>
2
3 <launch>
4
5   <include file="$(find racecar)/launch/teleop.launch"/>
6     <!-- TELEOPERATION -->
7   <node pkg="tfg_alba" name="follower" type="follower.py" output="screen"
8     >   <!-- PATH TRACKING ALGORITHM-->
9     <param name="scan" value="scan"/>
10    </node>
11 </launch>
```

## 4.7. Ejecución mediante la terminal

La ejecución de los comandos para poner en marcha el robot se realizará mediante la terminal de Ubuntu. Lo primero es situarse en el directorio del proyecto. Después, se configura este como entorno de trabajo en la terminal con el comando *source*. Por último, se ejecuta el archivo de lanzamiento para simulación (script 4.1). Este archivo incluye el lanzamiento del mapa, la teleoperación y el nodo de seguimiento.

Estos pasos se realizan ejecutando en una terminal los siguientes comandos:

```
$ cd catkin_racecar
$ source devel/setup.bash
$ roslaunch racecar_control gazebo_sim_joy.launch
```

Cuando se realicen las pruebas reales, el procedimiento será similar. Se cambiará el último comando para la ejecución del archivo de lanzamiento 4.2:

```
$ roslaunch tfg_alba mi_launch.launch
```

Además, en las pruebas reales, se deberá realizar la conexión con el mando de control remoto, en otra terminal, mediante:

```
$ sudo sixad -s
```

En el apéndice C se ha detallado el procedimiento de arranque del UMA RACECAR, la conexión entre el equipo de trabajo y el UMA RACECAR, y la teleoperación del mismo con un mando de PlayStation.

# CAPÍTULO 5

---

## Experimentación y resultados

---

En este capítulo se van a presentar las pruebas realizadas para comprobar el correcto funcionamiento de los métodos de seguimiento de caminos implementados en el UMA RACECAR. Aunque, primero se mostrará el ajuste de los parámetros de seguimiento.

### 5.1. Ajuste de parámetros de los métodos de seguimiento

El ajuste de parámetros se ha realizado mediante simulación. No obstante, la elección de estos parámetros también es válida para la implementación en el coche real. Esto se debe a que ambos modelos de robot tienen medidas físicas equivalentes. Para apreciar variaciones en el comportamiento del coche, se han tomado valores extremos de los parámetros y se han realizado gráficas con los mismos.

#### 5.1.1. Ajuste del parámetro $d_w$ en el seguimiento de pared

Tras probar una serie de valores para el parámetro  $d_w$ , se ha llegado a la conclusión de que lo más cerca que se puede seguir la pared es a una distancia de 1 m.

En la Fig. 5.1 se puede ver que, si se disminuye esta distancia a 0,8 m, el coche choca contra la pared. Esto queda indicado ya que la línea naranja no continúa el mismo tiempo que las otras. Esto se debe a que no tiene distancia suficiente para realizar curvas cerradas. Esto se indicó anteriormente mediante el máximo ángulo de giro de las ruedas en el modelo de Ackermann y su correspondiente radio de giro.

Por el contrario, si ponemos un valor mayor o igual a 1 m, la simulación funciona sin interrupción. Esto se debe a que al alejarse de la pared no tiene problema ya que no se adapta a los cambios de dirección de la misma. En este caso, se ha registrado el recorrido del robot para un valor de  $d_w = 2$  m. Se puede comprobar como los giros que realiza el coche son más suaves pero el recorrido no se asemeja a la distribución de las paredes.

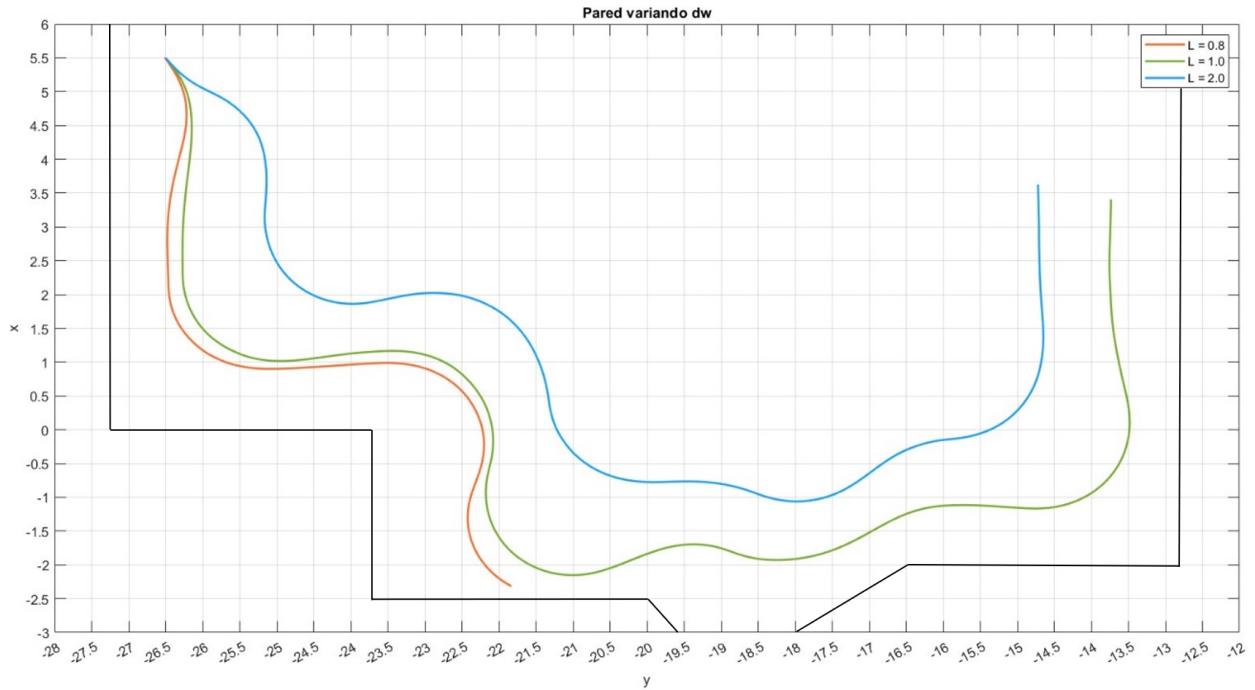


Figura 5.1: Variación de  $d_w$  en el seguimiento de pared.

### 5.1.2. Ajuste del parámetro *lookahead* L

El valor del parámetro L se ha ajustado tanto en el seguimiento de pared como en el seguimiento de pasillo. Se utiliza una distancia de *lookahead* (L) ya que los vehículos no holonómicos no pueden corregir errores respecto al punto más cercano de la ruta implícita.

Una mayor distancia de *lookahead* implica una ganancia menor, por lo que el control de la dirección es más suave a expensas de una menor precisión en el seguimiento. Por otro lado, una distancia de *lookahead* menor puede reducir los errores de seguimiento, pero los comandos de dirección aumentan de valor y el movimiento del vehículo se vuelve inestable.

En las Fig. 5.2 y 5.3 se muestran las pruebas realizadas para el seguimiento de pared y de pasillo respectivamente. Se observa que para valores grandes de L (por ejemplo,  $L = 3$  m) realiza el recorrido correctamente sin fluctuaciones. No obstante, no lo realiza con exactitud. En cambio, para valores pequeños, como  $L = 0,1$  m, se ve que el recorrido oscila mucho respecto a la ruta implícita. En el caso del seguimiento de pared, esta oscilación provoca el choque con la pared.

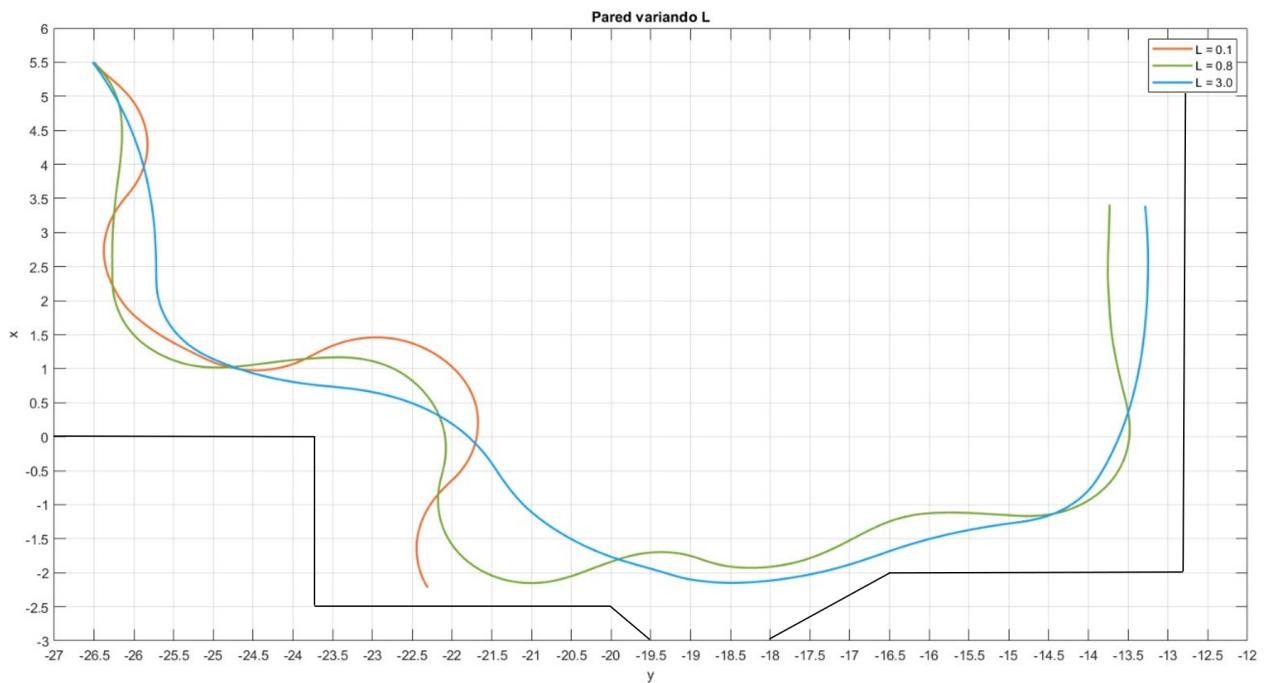


Figura 5.2: Variación de  $L$  en el seguimiento de pared.

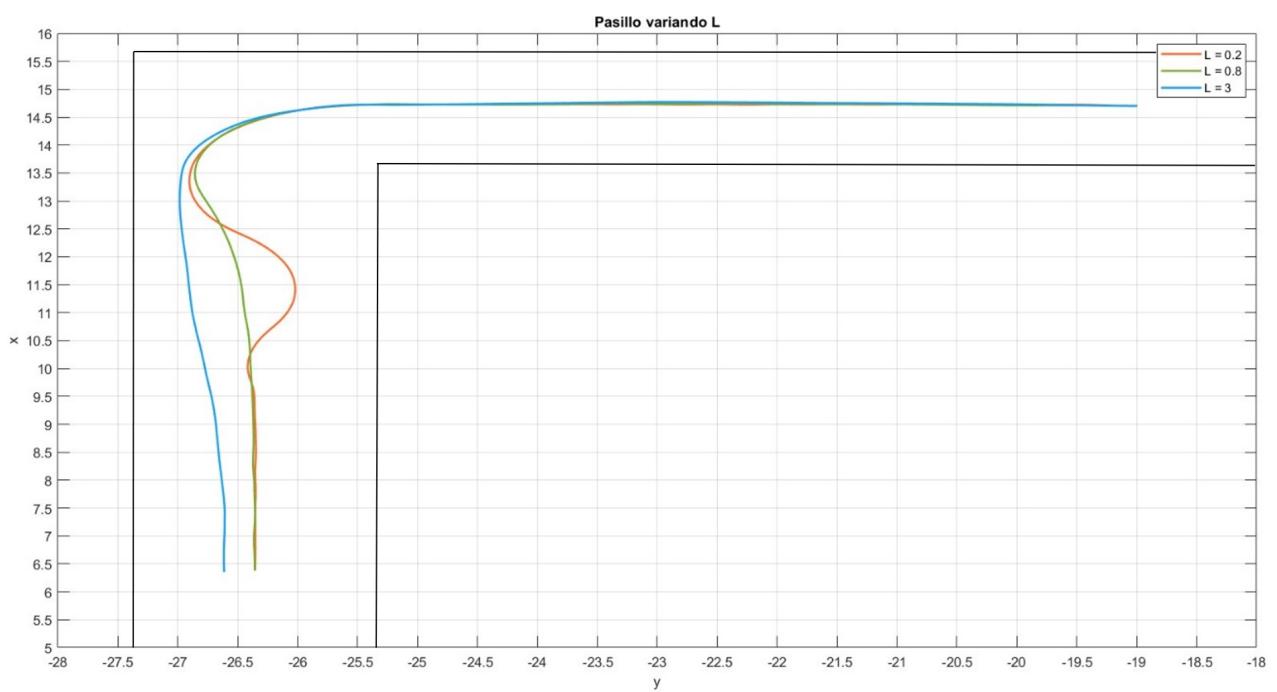


Figura 5.3: Variación de  $L$  en el seguimiento de pasillo.

### 5.1.3. Ajuste del parámetro $d_f$ en el seguimiento de persona

En el seguimiento de persona tenemos varios parámetros. Para la distancia mínima de seguimiento, coincidente con la distancia de búsqueda, se ha comprobado experimentalmente que el valor  $d_s = 0,8\text{ m}$  proporciona un buen funcionamiento.

El parámetro determinante que se debe ajustar es la distancia máxima a la que se sigue a la persona. Si la persona supera esta distancia, el robot dejará de seguirla y deberá comenzar de nuevo la búsqueda. Esta distancia también influye en la velocidad de avance del robot.

La variación de velocidad no se aprecia con claridad en simulación, por lo que se ha hecho el ajuste mediante pruebas reales. Se ha llegado a la conclusión de que el valor  $d_f = 3,0\text{ m}$  es un valor adecuado. Si el valor de la distancia máxima se acerca al valor de la mínima, la velocidad del robot conmuta entre el valor mínimo y el máximo, como indica la Ec. 3.5. Esto se percibe visualmente como cambios bruscos de velocidad, frenando y acelerando continuamente.

Una vez realizados todos los ajustes, el conjunto de parámetros se resume en:  $d_w = 1,0\text{ m}$ ,  $L_{wall} = 0,8\text{ m}$ ,  $L_{corridor} = 0,8\text{ m}$ ,  $d_s = 0,8\text{ m}$  y  $d_f = 3,0\text{ m}$ .

## 5.2. Análisis de resultados

El análisis de los resultados se realiza mediante la implementación de cada uno de los códigos definidos anteriormente en el vehículo autónomo. Se realizarán dos tipos de pruebas, unas de carácter simulado en el equipo de trabajo y otras reales con el UMA RACECAR.

En ambos tipos de prueba se ha seguido un orden específico para la realización de los tests. Primero se ha probado el funcionamiento del seguimiento de pasillos. Seguidamente se han realizado las pruebas con la pared, tanto izquierda como derecha. Y por último, se ha comprobado el correcto seguimiento de la persona.

Con el objetivo de seguir una línea de trabajo acorde al proyecto, el primer grupo de pruebas se realizará mediante simulación y posteriormente se realizarán las pruebas con el UMA RACECAR en un entorno real. El estudio del comportamiento de las pruebas reales se ha realizado en la Escuela de Ingenierías Industriales de la Universidad de Málaga.

Las pruebas simuladas y reales se han realizado de forma independiente, no obstante los resultados se pueden analizar de manera conjunta. Esto se hará para una mayor brevedad y facilidad de comprensión, y se desarrollarán según el tipo de seguimiento.

Los vídeos completos de las simulaciones y de las pruebas reales se pueden visualizar en los siguientes enlaces:

- Simulación: <https://youtu.be/KU1TqksImQI>
- Prueba real: [https://youtu.be/temHTyEgR\\_Q](https://youtu.be/temHTyEgR_Q)

La primera de las pruebas afrontadas será el seguimiento de pasillo. En la Fig. 5.4 se puede ver el coche realizando el seguimiento de pasillo tanto en simulación como en la prueba real.

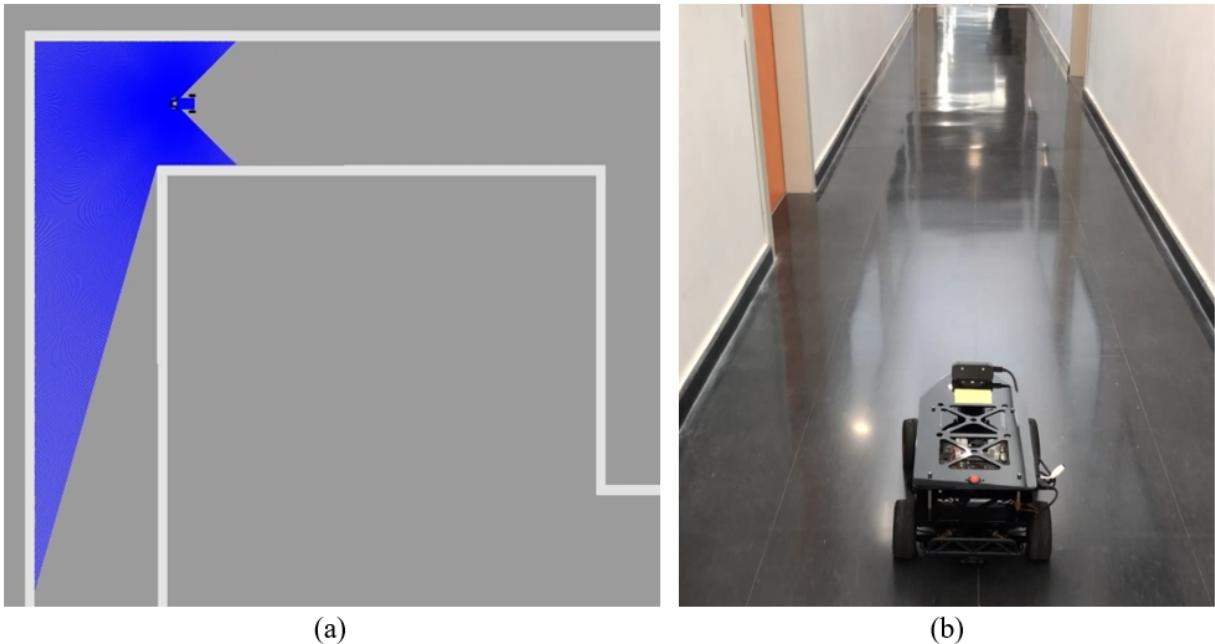


Figura 5.4: Pruebas de simulación (a) y real (b) para el seguimiento de pasillo.

El nodo de seguimiento está configurado para mostrar por pantalla una serie de mensajes. Estos se han creado con el fin de monitorizar el funcionamiento en tiempo real. En este ensayo, el mensaje (mostrado en la Fig. 5.5) indica la distancia entre el láser y las paredes laterales. Los datos recogidos permiten al robot situarse sobre la línea media del pasillo, como se describía en el apartado 3.4.

Los mensajes de distancias se enviarán de manera periódica, evidenciando la adaptación del robot a posibles cambios en la morfología del pasillo. Esto se puede corroborar observando que el robot irá corrigiendo su trayectoria con el fin de mantener equidistancias a ambas paredes.

En la Fig. 5.5 aparece otro tipo de mensaje, referente a “safe zone”. Este mensaje indica que el coche ha entrado en la zona de seguridad, debido a una aproximación a un obstáculo frontal. Esta interrupción hará que el robot aumente su velocidad angular al máximo, con el fin de esquivar el obstáculo percibido.

```
CORRIDOR - rho_right = 0.935053467751, min_right = -0.0277201319767
CORRIDOR - rho_left = 0.919757068157, min_left = 0.0286339824814
CORRIDOR - rho_right = 0.937774837017, min_right = -0.0267301272632
CORRIDOR - rho_left = 0.907861411572, min_left = 0.0268824356807
CORRIDOR - rho_right = 0.92941403389, min_right = -0.0299286040298
CORRIDOR - rho_left = 0.912624180317, min_left = 0.0268824356807
('CORRIDOR - safe zone, w = wmax = ', -0.5076870253639161)
CORRIDOR - rho_right = 0.936010122299, min_right = -0.0277962861854
CORRIDOR - rho_left = 0.910289108753, min_left = 0.0281009030203
('CORRIDOR - safe zone, w = wmax = ', -0.5076870253639161)
```

Figura 5.5: Mensajes terminal en el seguimiento de pasillo.

La siguiente prueba realizada ha sido el seguimiento de paredes. Se han tomado como objetivo tanto la pared izquierda como la derecha, pero en simulaciones separadas. En la Fig. 5.6 se pueden ver los cuatro casos de simulación.

La comprobación del funcionamiento de esta prueba es similar a la realizada en el seguimiento de pasillo. Esto es debido a que el mensaje mostrado (Fig. 5.7) será la monitorización en tiempo real de la distancia a la pared. Sin embargo, en este caso la ruta implícita de seguimiento es una línea paralela a la pared, como se observa en la Fig. 5.6.

El mensaje que indica la distancia a la pared es de utilidad para comprobar que se está siguiendo la trayectoria deseada. Este valor debe ser constante y aproximado a  $d_w = 1$  m. Cuando la pared no sea recta, la distancia mostrada irá estabilizándose alrededor de este valor. Esto es indicativo de que el robot se está adaptando a los cambios de dirección de la pared.

Al igual que en el seguimiento de pasillo, se mostrará un mensaje que indique la entrada en la zona de seguridad. Con el fin de evitar posibles colisiones frontales.

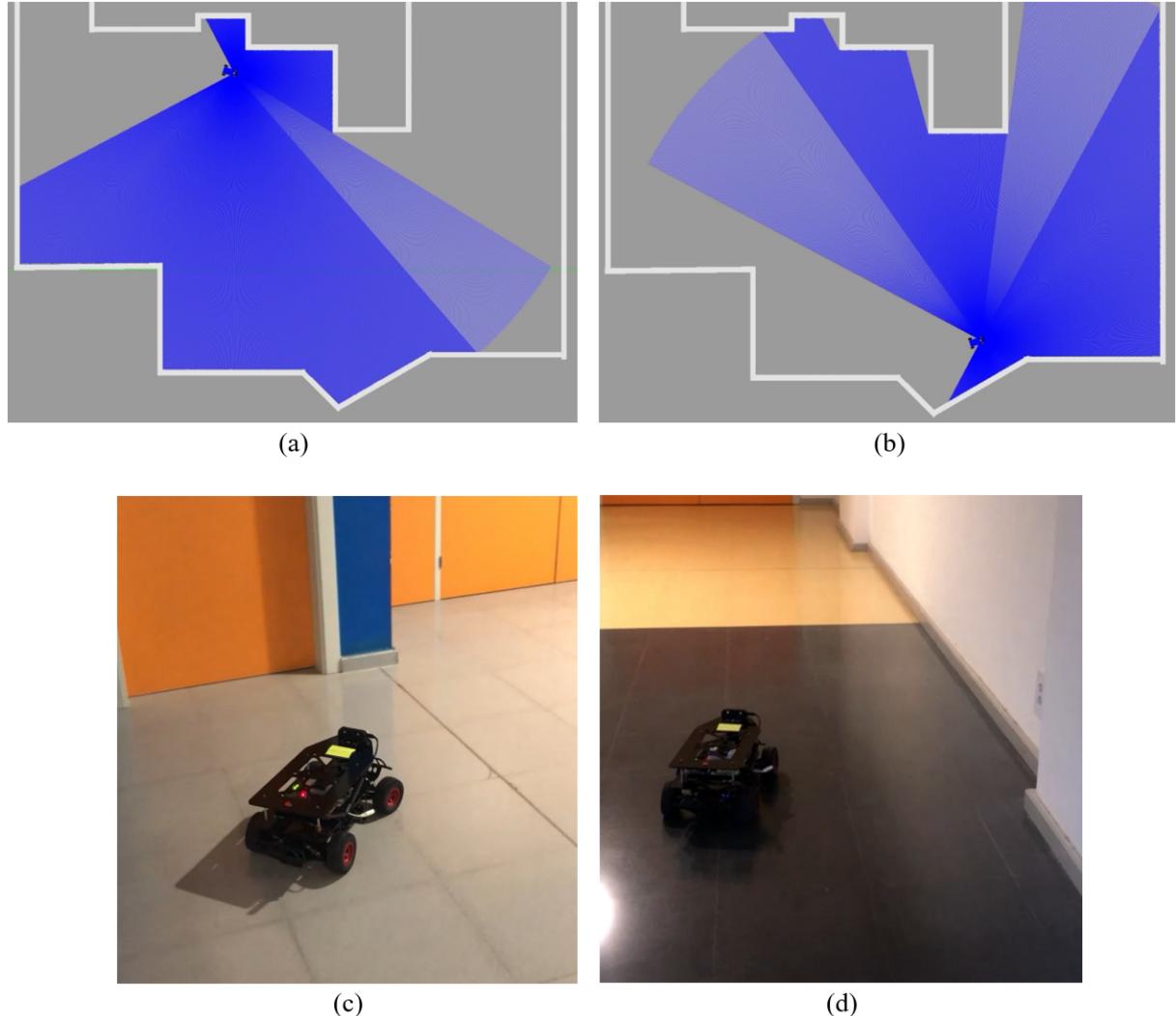


Figura 5.6: Pruebas de seguimiento de pared izquierda en simulación (a) y real (c), y de pared derecha en simulación (b) y real (d).

```

LEFT_WALL - rho = 0.983175456524, angle = 110.805961256
LEFT_WALL - rho = 0.975525975227, angle = 116.808993017
LEFT_WALL - rho = 0.987990617752, angle = 110.555834932
LEFT_WALL - rho = 0.982929587364, angle = 111.556340226
LEFT_WALL - safe zone
LEFT_WALL - rho = 0.979876160622, angle = 107.554319052
LEFT_WALL - safe zone

RIGTH_WALL - rho = 0.981777131557, angle = -82.2915603907
RIGTH_WALL - rho = 0.987204730511, angle = -93.2971186192
RIGTH_WALL - rho = 0.983614504337, angle = -85.0429499478
RIGTH_WALL - rho = 0.988005757332, angle = -93.2971186192
RIGTH_WALL - safe zone
RIGTH_WALL - rho = 0.987210452557, angle = -92.2966133257
RIGTH_WALL - safe zone
    
```

Figura 5.7: Mensajes terminal en el seguimiento de pared.

La última prueba realizada será el seguimiento de persona. El objetivo de seguimiento es dinámico, por lo que el coche se irá direccionando a la posición del mismo. En la Fig. 5.8 se puede observar como el robot se va orientando a medida que el objetivo cambia de posición.

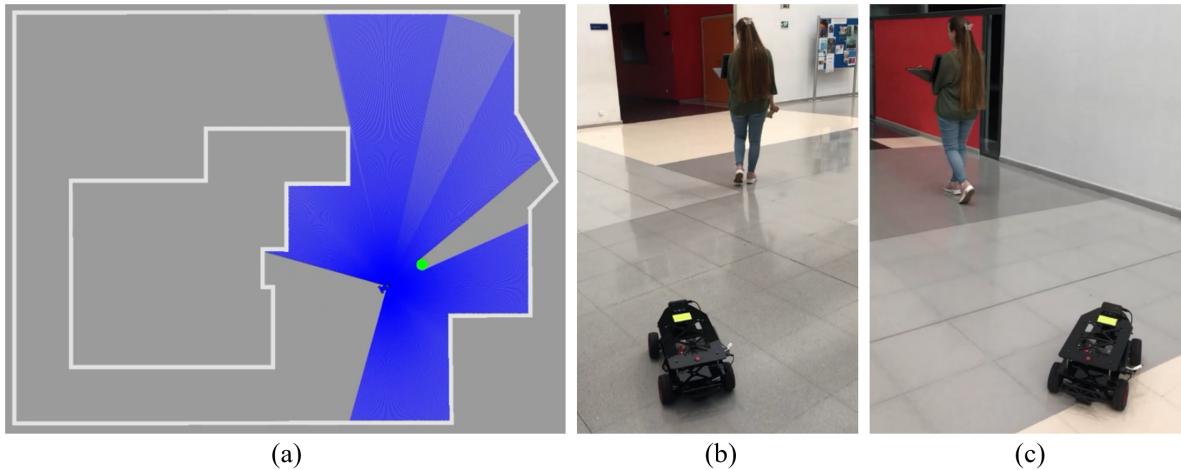


Figura 5.8: Pruebas de simulación (a) y real (b)(c) para el seguimiento de persona.

En este tipo de seguimiento se mostrarán dos tipos de mensajes (Fig. 5.9). El primero indica que el coche está parado buscando un objetivo. Cuando el robot detecte un objeto a una distancia menor o igual a  $d_s = 0,8\text{ m}$  se comenzará el seguimiento de persona. Una vez iniciado el seguimiento, el mensaje cambiará al segundo tipo. Este proporciona información sobre la ubicación del objetivo respecto del robot. Estos datos se usarán para ajustar la velocidad en función de la distancia lineal a la que se encuentra la persona.

```

PERSON - x_k = 0, y_k = 0, rho_k = 1.79769313486e+308, vsp = 0
('PERSON - first_rho', 1.7976931348623157e+308)
PERSON - x_k = 0, y_k = 0, rho_k = 1.79769313486e+308, vsp = 0
('PERSON - first_rho', 1.0943058729171753)

PERSON - x_k = 1.08689832687, y_k = 0.0, rho_k = 1.08689832687, vsp = 0.204927376338
PERSON - x_k = 0.958516710039, y_k = -0.0796464430384, rho_k = 0.961820065975, vsp = 0.115585761411
PERSON - x_k = 0.935201328965, y_k = -0.139766502741, rho_k = 0.94558775425, vsp = 0.103991253035
PERSON - x_k = 0.930324391524, y_k = -0.168229724976, rho_k = 0.945412456989, vsp = 0.103866040707
    
```

Figura 5.9: Mensajes terminal en el seguimiento de persona.



# CAPÍTULO 6

---

## Conclusiones y futuras líneas de trabajo

---

### 6.1. Conclusiones

En este Trabajo de Fin de Grado se ha llevado a cabo el desarrollo de un método de navegación reactiva para un vehículo robotizado a escala con direccionamiento de tipo Ackermann. El objetivo de este proyecto ha sido dotar al vehículo con la capacidad de realizar seguimientos de pared, pasillo y persona de forma autónoma. Para ello, se ha realizado una exhaustiva investigación acerca de la planificación reactiva de caminos. Además, se profundizó en la comprensión de la cinemática del coche a escala UMA RACECAR, para su correcta implementación.

Todo este desarrollo ha ido acompañado del continuo estudio de ROS y Gazebo. Estos sistemas, que están orientados al control y simulación en robótica, han permitido desarrollar la tecnología necesaria para cada una de las exigencias del proyecto. Prueba de ello son los resultados experimentales obtenidos tanto en simulación como mediante pruebas reales.

En conclusión, se ha logrado implementar en el UMA RACECAR un método reactivo de planificación y seguimiento autónomo de caminos implícitos.

### 6.2. Futuras líneas de trabajo

La realización de este Trabajo Fin de Grado ofrece posibilidades de mejora del mismo. La línea de trabajo principal sería la realización de un análisis de la nube de puntos del láser con el objetivo de mejorar la detección de paredes, pasillos, obstáculos y personas. Esto permitiría, por ejemplo, anticiparse a cambios de dirección en paredes, pasillos o evitar obstáculos, que permitiría reducir el tamaño de la zona de seguridad.

En cuanto a la estructura mecánica del UMA RACECAR, se podría modificar su cinemática y hacer las cuatro ruedas directrices. Esto permitiría realizar giros más cerrados y una mejor adaptación a la morfología del entorno.



## Apéndices



# APÉNDICE A

## Códigos

### A.1. Nodo de seguimiento

Script A.1: follower.py

```
1 #!/usr/bin/env python2
2
3 import rospy
4 from sensor_msgs.msg import LaserScan
5 from sensor_msgs.msg import Joy
6 from ackermann_msgs.msg import AckermannDriveStamped
7 import math
8 import numpy as np
9
10
11 class Drive:
12
13     def __init__(self):
14
15         ## PARAMETERS ##
16
17         self.mode = 4      # Initial operation mode (do nothing)
18
19         self.L_wall = 0.8    # Look-ahead (m) for Wall Following
20         self.L_cor = 0.8    # Look-ahead (m) for Corridor Following
21         self.dw = 1.0       # Distance to the wall (m)
22         self.v = 0.8        # Linear velocity (m/s)
23
24         # SIMULATED CAR
25         self.dy = 0.265          # Distance between the center of the
26         # robot and the center of the laser (m)
27         self.l = 0.32            # Wheelbase between the front and
28         # rear wheels (m)
29         self.psi = 18*(np.pi/180)   # Maximum steering angle of the
30         # wheels
31
32         # REAL CAR
33         # self.dy = 0.315
34         # self.l = 0.315
35         # self.psi = 18*(np.pi/180)
```

## APÉNDICE A. CÓDIGOS

---

```
34     # SAFE ZONE
35     self.rmin = self.l / (math.tan(self.psi))                      # Minimum
36     steering radius
37     # print("rmin", self.rmin)
38     self.sz = self.dw + self.rmin                                     # Safe
39     distance for wall following
40     self.wmax = self.v * (math.tan(self.psi) / self.l)    # Maximum
41     angular velocity for minimum steering angle
42
43     # PERSON FOLLOWER
44     self.ds = 0.8                                         # Initial search distance for the
45     person
46     self.dth = 0.54                                       # Threshold
47     self.df = 3.0                                         # Maximum distance
48     self.detected = False                                    # Check if the person has been
49     detected
50     self.init_angle = False                                 # Initial angle of the detected
51     person
52     self.x_k = 0                                         # Initial x-coordinate of the
53     detected person
54     self.y_k = 0                                         # Initial y-coordinate of the
55     detected person
56     self.rho_k = 0
57     self.theta_k = 0
58
59     ## TOPICS ##
60
61     laser_topic = '/scan'
62     joy_topic = '/vesc/joy'
63     drive_topic = '/vesc/high_level/ackermann_cmd_mux/input/nav_0'
64
65     self.sent_message = AckermannDriveStamped()
66
67     ## SUBSCRIPTIONS ##
68
69     rospy.Subscriber(joy_topic, Joy, self.joy_callback)
70     rospy.Subscriber(laser_topic, LaserScan, self.scan_callback)
71
72     ## PUBLICATIONS ##
73
74     self.pub = rospy.Publisher(drive_topic, AckermannDriveStamped,
75     queue_size=1)
76
77     rospy.spin()
78
79 def joy_callback(self, data):
80
81     ## SIMULATION ##
82
83     if data.buttons[0]:
84         self.mode = 0
85         print("X button pressed, running: person")
86
87     elif data.buttons[1]:
88         self.detected = False
89         self.mode = 1
90         print("O button pressed, running: right wall")
```

```

83
84     elif data.buttons[2]:
85         self.detected = False
86         self.mode = 2
87         print("D button pressed, running: corridor")
88
89     elif data.buttons[3]:
90         self.detected = False
91         self.mode = 3
92         print("[] button pressed, running: left wall")
93
94     ## TESTING ##
95
96     # if data.buttons[14]:
97     #     self.mode = 0
98     #     print("X button pressed, running: person")
99
100    # elif data.buttons[13]:
101    #     self.detected = False
102    #     self.mode = 1
103    #     print("O button pressed, running: rigth wall")
104
105    # elif data.buttons[12]:
106    #     self.detected = False
107    #     self.mode = 2
108    #     print("D button pressed, running: corridor")
109
110    # elif data.buttons[15]:
111    #     self.detected = False
112    #     self.mode = 3
113    #     print("[] button pressed, running: left wall")
114
115     return self.mode
116
117
118 def left_wall(self, msg):
119
120     ## SCANNER DATA ##
121
122     n_ranges = self.ranges.size
123     index_min = np.argmin(self.ranges[int(n_ranges/2):n_ranges]) + int(n_ranges/2)
124     rho = self.ranges[index_min]
125     angle = self.angles[index_min]
126     sign_angle = math.copysign(1, angle)
127
128     print("LEFT_WALL - distance = {}, angle = {}".format(rho, angle *180/3.14))
129
130     ## SAFE ZONE ##
131
132     safe = self.ranges[int(n_ranges/2)]
133
134     if safe < self.sz:
135         w = - self.wmax
136         print("LEFT_WALL - safe zone")
137
138     else:

```

## APÉNDICE A. CÓDIGOS

---

```
139
140     ## PURE-PURSUIT ALGORITHM ##
141
142     yp = sign_angle * (rho + self.dy * math.cos(angle) - self.dw
143 )
144
145     if abs(yp) < self.L_wall:
146         ygg = yp
147         xgg = math.sqrt(self.L_wall**2 - yp**2)
148
149     else:
150         ygg = self.L_wall * math.copysign(1, yp)
151         xgg = 0
152
153     yg = sign_angle * (ygg * math.sin(angle) - xgg * math.cos(
154 angle))
155     gamma = (2 * yg) / (self.L_wall**2)
156     w = gamma * self.v
157
158     ## ACKERMANN CONVERSION ##
159
160     steering_angle = math.atan((w * self.l) / self.v)
161
162
163 def righth_wall(self, msg):
164
165     ## SCANNER DATA ##
166
167     n_ranges = self.ranges.size
168     index_min = np.argmin(self.ranges[0:int(n_ranges/2)])
169     rho = self.ranges[index_min]
170     angle = self.angles[index_min]
171     sign_angle = math.copysign(1, angle)
172
173     print("RIGTH_WALL - distance = {}, angle = {}".format(rho, angle
174 *180/3.14))
175
176     ## SAFE ZONE ##
177
178     safe = self.ranges[int(n_ranges/2)]
179
180     if safe < self.sz:
181         w = self.wmax
182         print("RIGTH_WALL - safe zone")
183
184     else:
185
186         ## PURE-PURSUIT ALGORITHM ##
187
188         yp = sign_angle * (rho + self.dy * math.cos(angle) - self.dw
189 )
190
191         if abs(yp) < self.L_wall:
192             ygg = yp
193             xgg = math.sqrt(self.L_wall**2 - yp**2)
```

```

193         else:
194             ygg = self.L_wall * math.copysign(1, yp)
195             xgg = 0
196
197             yg = sign_angle * (ygg * math.sin(angle) - xgg * math.cos(
198                 angle))
199             gamma = (2 * yg) / (self.L_wall**2)
200             w = gamma * self.v
201
202             ## ACKERMANN CONVERSION ##
203
204             steering_angle = math.atan((w * self.l) / self.v)
205
206             return steering_angle
207
208     def corridor(self, msg):
209
210         ## SCANNER DATA ##
211
212         n_ranges = self.ranges.size
213
214         min_right = np.argmin(self.ranges[0:int(n_ranges/2)])
215             # Index min from -135 to 0 grades
216         min_left = np.argmin(self.ranges[int(n_ranges/2+1):n_ranges]) +
217             int(n_ranges/2+1)    # Index min from 0 to +135 grades
218
219         rho_right = self.ranges[min_right]
220         rho_left = self.ranges[min_left]
221
222         theta_right = self.angles[min_right]
223         theta_left = self.angles[min_left]
224         theta_c = theta_right + theta_left
225
226         print("CORRIDOR - dist_right = {}, angle_right = {}".format(
227             rho_right, theta_right*np.pi/180))
228         print("CORRIDOR - dist_left = {}, angle_left = {}".format(
229             rho_left, theta_left*np.pi/180))
230
231         ylg = (rho_left) + (self.dy * math.cos(theta_left))
232         yrg = -(rho_right) - (self.dy * math.cos(theta_right))
233         ypg = (ylg + yrg) / 2
234
235         ## SAFE ZONE ##
236
237         safe = self.ranges[int(n_ranges/2)]
238         sz = ((rho_right + rho_left) / 2) + self.rmin
239
240         if safe < sz:
241
242             if rho_left > rho_right:          # Like right wall following
243                 w = self.wmax
244
245             elif rho_left < rho_right:        # Like left wall following
246                 w = - self.wmax
247
248             print("CORRIDOR - safe zone")
249
250

```

## APÉNDICE A. CÓDIGOS

---

```
246     else:
247
248         ## PURE-PURSUIT ALGORITHM ##
249
250         if abs(ypg) < self.L_cor:
251             ygg = ypg
252             xgg = math.sqrt(self.L_cor**2 - ypg**2)
253
254         else:
255             ygg = self.L_cor * math.copysign(1, ypg)
256             xgg = 0
257
258             yg = xgg * math.sin(theta_c) + ygg * math.cos(theta_c)
259             gamma = (2 * yg) / (self.L_cor**2)
260             w = gamma * self.v
261
262         ## ACKERMANN CONVERSION ##
263
264         steering_angle = math.atan((w * self.l) / self.v)
265
266         return steering_angle
267
268
269     def person(self, msg):
270
271         ## SCANNER DATA ##
272
273         n_ranges = self.ranges.size
274
275         if self.detected == False:
276
277             self.rho_k = self.ranges[int(n_ranges/2)]
278             print("PERSON - first distance detected = ", self.rho_k)
279             self.theta_k = (msg.angle_min + msg.angle_max) / 2
280
281             if (self.rho_k >= (self.ds - self.dth)) and (self.rho_k <= (self.ds + self.dth)):
282                 self.detected = True
283                 self.x_k = self.dy + self.rho_k * math.cos(self.theta_k)
284                 self.y_k = self.rho_k * math.sin(self.theta_k)
285
286             else:
287                 steering_angle, vsp = (0, 0)
288
289         elif self.detected == True:
290
291             x_scan = self.ranges * np.cos(self.angles)
292             y_scan = self.ranges * np.sin(self.angles)
293             d_k = np.sqrt(((x_scan + self.dy) - self.x_k)**2 + (y_scan - self.y_k)**2)
294             d_k_min = np.min(d_k)
295
296         ## PURE-PURSUIT ALGORITHM ##
297
298             yg = self.rho_k * math.sin(self.theta_k)
299             L = math.sqrt(self.rho_k * (2 * self.dy * math.cos(self.theta_k) + self.rho_k))
300             gamma = (2 * yg) / (L**2)
```

```

301         if self.rho_k < self.ds:
302             vsp = 0
303
304         elif self.rho_k >= self.ds and self.rho_k <= self.df:
305             vsp = ((self.rho_k - self.ds) / (self.df - self.ds)) *
306             self.v
307
308         if d_k_min <= self.dth:
309             k_min = np.argmin(d_k)
310             self.x_k = x_scan[k_min]
311             self.y_k = y_scan[k_min]
312             self.rho_k = self.ranges[k_min]
313             self.theta_k = self.angles[k_min]
314
315         else:
316             self.detected = False
317             steering_angle, vsp = (0, 0)
318
319         if self.detected == True:
320
321             yg = self.rho_k * math.sin(self.theta_k)
322             L = math.sqrt(self.rho_k * (2 * self.dy * math.cos(self.
323             theta_k) + self.rho_k))
324             gamma = (2 * yg) / (L**2)
325
326             ## LINEAR VELOCITY ##
327
328             if self.rho_k < self.ds:
329                 vsp = 0
330
331             elif self.rho_k >= self.ds and self.rho_k <= self.df:
332                 vsp = ((self.rho_k - self.ds) / (self.df - self.ds)) *
333                 self.v
334
335             elif self.rho_k > self.df:
336                 vsp = self.v * 1.3
337
338             w = gamma * vsp
339
340             ## ACKERMANN CONVERSION ##
341
342             steering_angle = math.atan2(w * self.l, vsp)
343
344             print("PERSON - x = {}, y = {}, dist = {}, vel = {}".format(
345                 self.x_k, self.y_k, self.rho_k, vsp))
346
347             return steering_angle, vsp
348
349     def scan_callback(self, msg):
350
351         if self.init_angle == False:
352             self.angles = np.linspace(msg.angle_min, msg.angle_max, len(
353             msg.ranges))
354             self.init_angle = True
355
356             self.ranges = np.nan_to_num(np.array(msg.ranges))

```

## APÉNDICE A. CÓDIGOS

---

```
354         self.ranges[self.ranges < 0.0011] = 1e6
355
356     ## BREAK ZONE ##
357
358     if np.min(self.ranges) < 0.25:
359         steering_angle = 0
360         v = 0
361
362     else:
363
364         ## Steering angle and linear velocity depending on path
365         tracking mode ##
366
367         if self.mode == 0:
368             steering_angle, v = Drive.person(self, msg)
369
370         elif self.mode == 1:
371             steering_angle = Drive.righth_wall(self, msg)
372             v = self.v
373
374         elif self.mode == 2:
375             steering_angle = Drive.corridor(self, msg)
376             v = self.v
377
378         elif self.mode == 3:
379             steering_angle = Drive.left_wall(self, msg)
380             v = self.v
381
382         else:
383             steering_angle = 0
384             v = 0
385
386     ## PUBLISH SPEED & STEERING ANGLE ##
387
388     if steering_angle > self.psi:
389         steering_angle = self.psi
390
391     elif steering_angle < -self.psi:
392         steering_angle = -self.psi
393
394     self.sent_message.drive.speed = v
395     self.sent_message.drive.steering_angle = steering_angle
396 # Simulation
397     # self.sent_message.drive.steering_angle = - steering_angle
398 # Testing
399     self.pub.publish(self.sent_message)
400
401
402 if __name__ == '__main__':
403     rospy.init_node('follower')
404     Drive()
405     rospy.spin()
```

## A.2. Plugin para mover el cilindro

Script A.2: plannar\_mover.cpp

```

1 #include <functional>
2 #include <gazebo/gazebo.hh>
3 #include <gazebo/physics/physics.hh>
4 #include <gazebo/common/common.hh>
5 #include <ignition/math/Vector3.hh>
6 #include <thread>
7 #include "ros/ros.h"
8 #include "ros/callback_queue.h"
9 #include "ros/subscribe_options.h"
10 #include "std_msgs/Float32.h"
11 #include "geometry_msgs/Twist.h"
12 #include <gazebo/transport/transport.hh>
13 #include <gazebomsgs/msg.h>
14
15 // MADE by TheConstruct , contact duckfrots@theconstructsim.com for any
16 // doubts or questions
17 // Or leave a request in the public git that contains this code.
18
19 namespace gazebo
20 {
21     class PlannarMover : public ModelPlugin
22     {
23         public: void Load(physics::ModelPtr _parent, sdf::ElementPtr /*_sdf
24         */)
25         {
26             // Store the pointer to the model
27             this->model = _parent;
28
29             // Listen to the update event. This event is broadcast every
30             // simulation iteration.
31             this->updateConnection = event::Events::ConnectWorldUpdateBegin(
32                 std::bind(&PlannarMover::OnUpdate, this));
33
34             this->old_secs = ros::Time::now().toSec();
35
36             // Create a topic name
37             std::string plannar_pos_topicName = "/vesc/cmd_vel_object"; //Topic name modified from /vesc/cmd_vel because it actually exists
38
39             // Initialize ros, if it has not already bee initialized.
40             if (!ros::isInitialized())
41             {
42                 int argc = 0;
43                 char **argv = NULL;
44                 ros::init(argc, argv, "plannar_rosnode",
45                           ros::init_options::NoSigintHandler);
46             }
47
48             // Create our ROS node. This acts in a similar manner to
49             // the Gazebo node
50             this->rosNode.reset(new ros::NodeHandle("plannar_rosnode"));
51
52             // Plannar Pose
53             ros::SubscribeOptions so =

```

## APÉNDICE A. CÓDIGOS

---

```
52     ros::SubscribeOptions::create<geometry_msgs::Twist>(
53         plannar_pos_topicName ,
54         1,
55         boost::bind(&PlannarMover::OnRosMsg_Pos , this, _1),
56         ros::VoidPtr(), &this->rosQueue);
57     this->rosSub = this->rosNode->subscribe(so);
58
59     // Spin up the queue helper thread.
60     this->rosQueueThread =
61         std::thread(std::bind(&PlannarMover::QueueThread , this));
62
63     ROS_WARN("Loaded PlannarMover Plugin with parent...%s, only X Axis
64 Freq Supported in this V-1.0", this->model->GetName().c_str());
65 }
66
67 // Called by the world update start event
68 public: void OnUpdate()
69 {
70 }
71
72
73
74     void MoveModelsPlane(float linear_x_vel, float linear_y_vel, float
linear_z_vel, float angular_x_vel, float angular_y_vel, float
angular_z_vel)
75 {
76
77     std::string model_name = this->model->GetName();
78
79     ROS_DEBUG("Moving model=%s",model_name.c_str());
80
81     this->model->SetLinearVel(ignition::math::Vector3d(linear_x_vel,
linear_y_vel, linear_z_vel));
82     this->model->SetAngularVel(ignition::math::Vector3d(
angular_x_vel, angular_y_vel, angular_z_vel));
83
84     ROS_DEBUG("Moving model=%s....END",model_name.c_str());
85 }
86
87
88
89     public: void OnRosMsg_Pos(const geometry_msgs::TwistConstPtr &_msg)
90 {
91     this->MoveModelsPlane(_msg->linear.x, _msg->linear.y,_msg->
linear.z, _msg->angular.x, _msg->angular.y, _msg->angular.z);
92 }
93
94     /// \brief ROS helper function that processes messages
95     private: void QueueThread()
96 {
97     static const double timeout = 0.01;
98     while (this->rosNode->ok())
99     {
100         this->rosQueue.callAvailable(ros::WallDuration(timeout));
101     }
102 }
```

```

104
105     // Pointer to the model
106     private: physics::ModelPtr model;
107
108     // Pointer to the update event connection
109     private: event::ConnectionPtr updateConnection;
110
111     // Time Memory
112     double old_secs;
113
114     // Direction Value
115     int direction = 1;
116     // Frequency of earthquake
117     double x_axis_pos = 1.0;
118     // Magnitude of the Oscillations
119     double y_axis_pos = 1.0;
120
121     /// \brief A node use for ROS transport
122     private: std::unique_ptr<ros::NodeHandle> rosNode;
123
124     /// \brief A ROS subscriber
125     private: ros::Subscriber rosSub;
126     /// \brief A ROS callbackqueue that helps process messages
127     private: ros::CallbackQueue rosQueue;
128     /// \brief A thread the keeps running the rosQueue
129     private: std::thread rosQueueThread;
130
131
132     /// \brief A ROS subscriber
133     private: ros::Subscriber rosSub2;
134     /// \brief A ROS callbackqueue that helps process messages
135     private: ros::CallbackQueue rosQueue2;
136     /// \brief A thread the keeps running the rosQueue
137     private: std::thread rosQueueThread2;
138
139 };
140
141 // Register this plugin with the simulator
142 GZ_REGISTER_MODEL_PLUGIN(PlannarMover)
143 }
```

### A.3. Configuración del control remoto en simulación

Script A.3: joy\_teleop\_sim.yaml

```

1 joy_node:
2   deadzone: 0.01
3   autorepeat_rate: 20
4   coalesce_interval: 0.01
5
6 teleop:
7   # Default mode - Stop for safety
8   default:
9     type: topic
10    is_default: true
```

## APÉNDICE A. CÓDIGOS

---

```
11     message_type: ackermann_msgs/AckermannDriveStamped
12     topic_name: low_level/ackermann_cmd_mux/input/teleop
13     message_value:
14     -
15         target: drive.speed
16         value: 0.0
17     -
18         target: drive.steering_angle
19         value: 0.0
20
21 # Enable Human control by holding Left Bumper
22 human_control:
23     type: topic
24     message_type: ackermann_msgs/AckermannDriveStamped
25     topic_name: low_level/ackermann_cmd_mux/input/teleop
26     deadman_buttons: [4]
27     axis_mappings:
28     -
29         axis: 1
30         target: drive.speed
31         scale: 1.0                         # joystick will command plus or
32         minus 2 meters / second
33         offset: 0.0
34     -
35         axis: 3
36         target: drive.steering_angle
37         scale: 0.34                         # joystick will command plus or
38         minus ~20 degrees steering angle
39         offset: 0.0
40
41 # Enable autonomous control by pressing right bumper
42 # This switch causes the joy_teleop to stop sending messages to input/
43 # teleop
44 autonomous_control:
45     type: topic
46     message_type: geometry_msgs/Twist
47     topic_name: cmd_vel_object
48     deadman_buttons: [5]
49     axis_mappings:
50     -
51         axis: 1
52         target: linear.x
53         scale: 1.0                         # joystick will command
54         offset: 0.0
55     -
56         axis: 0
57         target: linear.y
58         scale: 1.0                         # joystick will command
59         offset: 0.0
```

## A.4. Archivo de configuración .world de Gazebo

Script A.4: racecar\_tunnel.world

```

1 <?xml version="1.0"?>
2 <sdf version="1.4">
3 <world name="racecar_tunnel">
4
5   <include>
6     <uri>model://sun</uri>
7   </include>
8
9   <gui>
10    <camera name='main'>
11      <pose>0 0 10 0 1.571 0</pose>
12    </camera>
13  </gui>
14
15 <model name="ground_plane">
16   <static>true</static>
17   <link name="link">
18     <collision name="collision">
19       <geometry>
20         <plane>
21           <normal>0 0 1</normal>
22           <size>150 150</size>
23         </plane>
24       </geometry>
25       <surface>
26         <friction>
27           <ode>
28             <mu>100</mu>
29             <mu2>50</mu2>
30           </ode>
31         </friction>
32       </surface>
33     </collision>
34     <visual name="visual">
35       <cast_shadows>false</cast_shadows>
36       <geometry>
37         <plane>
38           <normal>0 0 1</normal>
39           <size>150 150</size>
40         </plane>
41       </geometry>
42       <material>
43         <script>
44           <uri>file:///media/materials/scripts/gazebo.material</uri>
45           <name>Gazebo/Grey</name>
46         </script>
47       </material>
48     </visual>
49   </link>
50 </model>
51
52 <include>
53   <uri>model://racecar_description/models/Mapa_TFG</uri>
54   <pose>6 20 0 0 0 0</pose>

```

```

55 </include>
56
57 <!-- CUSTOM Cylinder -->
58 <model name="custom_ground_plane_box">
59   <pose>5.5 14.5 0.5 0 0 0</pose>
60   <link name="link_box">
61     <inertial>
62       <pose>0 0 0 0 0 0</pose>
63       <mass>100</mass>
64       <inertia>
65         <ixx>3.33341666667e+3</ixx>
66         <ixy>0.0</ixy>
67         <ixz>0.0</ixz>
68         <iyy>3.33341666667e+3</iyy>
69         <iyz>0.0</iyz>
70         <izz>6.66666666667e+3</izz>
71       </inertia>
72     </inertial>
73     <collision name="collision_box">
74       <geometry>
75         <cylinder>
76           <radius>0.2</radius>
77           <length>1.0</length>
78         </cylinder>
79       </geometry>
80       <surface>
81         <friction>
82           <ode>
83             <mu>0</mu>
84             <mu2>0</mu2>
85           </ode>
86         </friction>
87       </surface>
88     </collision>
89
90     <visual name="visual_box">
91       <geometry>
92         <cylinder>
93           <radius>0.2</radius>
94           <length>1.0</length>
95         </cylinder>
96       </geometry>
97       <material>
98         <script>
99           <uri>file://media/materials/scripts/gazebo.material</uri>
100          <name>Gazebo/Green</name>
101        </script>
102      </material>
103    </visual>
104  </link>
105  <plugin name="plannar_mover_plugin" filename="
106    libplannar_mover_plugin.so"/>
107 </model>
108 </world>
109 </sdf>

```

# APÉNDICE B

---

## Introducción a ROS

---

### B.1. Introducción

En este apéndice se va a hacer una introducción al sistema de ROS. Además, se explicará cómo se debe instalar y configurar.

Como se ha visto anteriormente, ROS proporciona soporte mediante varias vías. En los siguientes enlaces, se puede acceder a cada una de ellas:

- Página oficial: <https://www.ros.org/>
- Documentación: <http://docs.ros.org/>
- Tutoriales: <https://wiki.ros.org/ROS/Tutorials>
- Foros: <https://discourse.ros.org/>
- Wiki: <https://wiki.ros.org/>
- APIs: <https://wiki.ros.org/APIs>

### B.2. Configuración

Lo primero que se debe hacer es configurar el ordenador para poder aceptar software de ROS.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ubuntu trusty main" > /etc/apt/sources.list.d/ros-latest.list'
```

También será necesario instalar la clave de autenticación:

```
$ wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
$ sudo apt-key add ros.key
```

## B.3. Instalación

Antes de la instalación, se debe asegurar que el índice de paquetes está actualizado.

```
$ sudo apt-get update
```

Para la instalación de ROS, solo será necesario saber que distribución queremos instalar y escribir el siguiente comando en la terminal.

```
$ sudo apt install ros-<version>-desktop-full
```

En el caso de este proyecto, la versión instalada ha sido *Melodic*. No se ha utilizado la última versión por problemas de incompatibilidad con el sistema operativo.

Por último, se debe inicializar el sistema *rosdep*, que es una herramienta utilizada para instalar dependencias requeridas por los paquetes de ROS.

```
$ sudo rosdep init  
$ rosdep update
```

Cuando ya esté todo configurado correctamente, se podrán instalar paquetes concretos con el comando:

```
$ sudo apt install ros-<version>-<paquete>
```

## B.4. Creación del entorno de trabajo

Se crea la carpeta de trabajo:

```
$ mkdir catkin_ws
```

Se crea un subdirectorio (llamado *src*) que contendrá los códigos fuente de los paquetes.

```
$ cd catkin_ws  
$ mkdir src
```

El siguiente paso es compilar el espacio de trabajo (*workspace*). Para crearlo, ejecutamos el siguiente comando.

```
$ catkin_make
```

Este comando es una herramienta para trabajar con espacios de trabajo catkin. Al ejecutarlo por primera vez se crean dos carpetas de configuración *build* y *devel*, y dos archivos *CMakeLists.txt* y *package.xml* en la carpeta *src*. Estos archivos incluyen la configuración de los paquetes y los nombres de todos los ejecutables que haya en los paquetes.

Después debemos ejecutar un script llamado *setup.bash*, ubicado dentro de la carpeta *devel*. Este genera variables que permiten a ROS encontrar nuestro paquete y regenerar los ejecutables, específicamente en el workspace.

```
$ source devel/setup.bash
```

Una vez construido y configurado el workspace, se podrán crear los paquetes necesarios para trabajar con:

```
$ catkin_create_pkg <paquete>
```

## B.5. Funcionamiento básico

ROS implementa una red de comunicación *peer-to-peer*. Lo que facilita el intercambio de datos estructurados entre los diferentes componentes del sistema.

La estructura de ROS consta de:

- **Nodos:** Los nodos son programas ejecutables. Cada nodo es un proceso independiente que puede realizar tareas específicas. Los nodos pueden comunicarse entre ellos mediante la publicación de mensajes. Para ejecutar un nodo, se ejecuta el comando:

```
$ rosrun <paquete> <nodo>
```

- **Mensajes:** Los mensajes son un tipo de estructura de datos mediante la cual se comunican los nodos. Cada mensaje tiene una serie de campos definidos. Cada tipo de mensaje está incluido en un paquete específico. Para ver la información de uno, ejecutamos:

```
$ rosmsg info <paquete>/<tipo>
```

- **Tópicos:** La comunicación entre nodos se realiza a través de tópicos. Un tópico es un canal de comunicación unidireccional a través del cual un nodo publica mensajes y otro nodo puede suscribirse para recibir esos mensajes. En un único tópico puede haber muchos nodos publicando y otros muchos suscribiéndose. Además de que un solo nodo puede publicar y suscribirse a varios tópicos. Por ello, los tópicos deben tener un nombre único. Para ver la lista completa:

```
$ rostopic list
```

- **Servicios:** Los servicios son un mecanismo de comunicación entre nodos. Los servicios permiten que un nodo solicite a otro nodo que realice una tarea específica y quede a la espera de respuesta. Esto proporciona una comunicación bidireccional, con un modelo cliente/servidor.

- **Bags:** Los *bags* son un formato de archivo que proporciona ROS para grabar, almacenar y reproducir datos de mensajes. Esto es útil para el desarrollo, la depuración y la reproducción de experimentos.

- **Master:** El ROS Master se encarga de mantener el registro de los nodos que se están ejecutando y de los tópicos y servicios que están disponibles. Esto permitirá la comunicación entre los nodos. Para que funcione es necesario ejecutarlo desde la terminal.

```
$ roscore
```

Por último, ROS incluye archivos de lanzamiento *.launch*. Estos son de utilidad para poder ejecutar tanto el *master* como los nodos de forma conjunta. Para ejecutar uno de estos archivos, se hará del siguiente modo:

```
$ rosrun <paquete> <archivo>
```

En la Fig. B.1, se puede observar una explicación gráfica del funcionamiento de ROS.

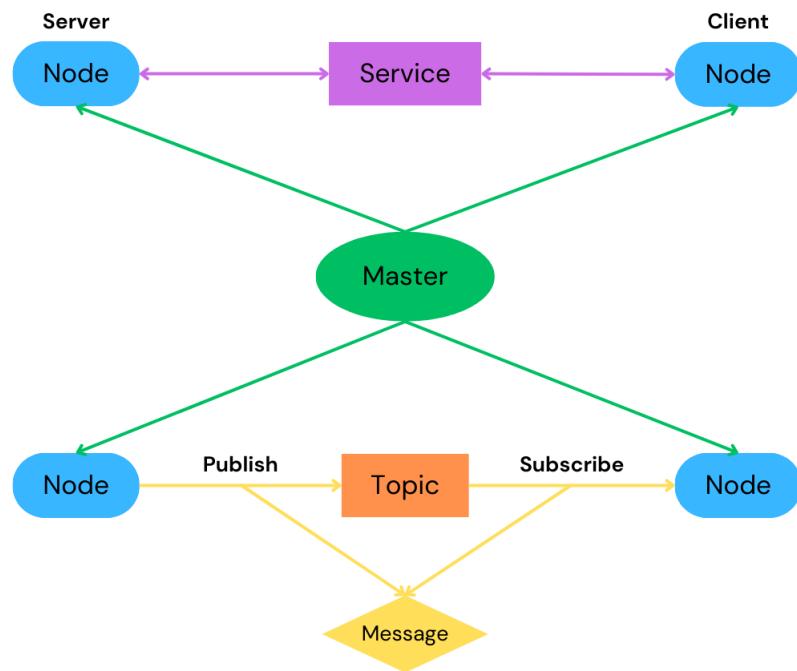


Figura B.1: Esquema básico del funcionamiento de ROS.

# **APÉNDICE C**

---

## **Puesta en marcha del UMA RACECAR**

---

### **C.1. Introducción**

El objetivo de este apéndice es facilitar la puesta en marcha del UMA RACECAR para poder realizar pruebas físicas. Esto incluye las conexiones físicas para el arranque de los componentes del coche y las conexión virtuales entre el coche y el ordenador de trabajo personal.

### **C.2. Guía de arranque del UMA RACECAR**

Para poner en funcionamiento el UMA RACECAR, debemos seguir los siguientes pasos, en el orden indicado, para la interconexión del hardware.

1. Conectar la Jetson TX2 al cable de corriente de la batería externa.
2. Conectar la batería LiPo.
3. Pulsar el botón de encendido de la batería externa durante unos segundos hasta que se encienda su pantalla.
4. Pulsar este botón de encendido (en intervalos de dos pulsaciones) hasta ver en la pantalla un voltaje de 16V.
5. Accionar el interruptor de encendido general del coche (situado en la parte superior).
6. Pulsar el botón de encendido (botón derecho de los cuatro que hay) de la Jetson TX2.

### C.3. Conexión entre el equipo de trabajo y el UMA RACECAR

La conexión entre el equipo de trabajo y el coche se ha realizado a través de una conexión WiFi [16]. La tarjeta WiFi del equipo de a bordo cuenta con una red a la que conectarse de forma inalámbrica.

Para realizar esta conexión correctamente, se debe modificar el archivo de configuración `.bashrc` en el ordenador de trabajo. Esto se hará con el fin de utilizar el UMA RACECAR como *master* de ROS, con el que realizar las comunicaciones. Para ello, se han añadido las siguientes líneas al final de dicho archivo:

```
export ROS_MASTER_URI=http://10.42.0.1:11311
export ROS_IP=127.0.0.1
export ROS_HOSTNAME=127.0.0.1
```

Se deben añadir las IP, para que ambos dispositivos se reconozcan y realicen correctamente las comunicaciones y conexiones entre el *master* y los nodos.

En el caso del desarrollo de este proyecto, se ha realizado en una máquina virtual y por ello la dirección IP es 127.0.0.1.

Por último, se va conectar el equipo de trabajo al ordenador de a bordo mediante clave ssh. La principal ventaja de este método es la mejora del rendimiento en el equipo de trabajo frente al de la placa de desarrollo Jetson TX2. Para realizarlo, se ejecuta el siguiente comando en una terminal del equipo y se introduce la contraseña de usuario pertinente:

```
$ ssh tx2@10.42.0.1
```

Con esta conexión, se evita la necesidad de conectar el UMA RACECAR a un monitor. Esto nos facilita los lanzamientos de programas al no tener que utilizar dos ordenadores simultáneamente. Además, se evita el problema de no poder conectar el coche a un monitor mientras este está en movimiento.

Se ha utilizado la aplicación de Visual Studio Code para la modificación de archivos en el ordenador de a bordo. Esto se hace de forma sencilla mediante la funcionalidad de conexión a *host* e indicando la dirección IP del mismo.

## C.4. Teleoperación con mando

La teleoperación del coche con un *joystick* sirve de utilidad tanto en la simulación como en las pruebas físicas para colocar el robot en la posición deseada. Además, es necesario para la comutación entre el modo manual y el automático, y entre los distintos modos de seguimiento automático.

Los pasos para la conexión son los siguientes (en el orden indicado):

1. Abrir un terminal y ejecutar el comando:

```
$ sudo sixad -s
```

2. Introducir la contraseña de usuario.

3. Pulsar el botón de encendido del mando (botón PS) cuando el programa indique, en la terminal, que está esperando conexión.

4. La conexión se habrá realizado con éxito cuando el mando vibre durante unos segundos y tras ello se encienda uno de los leds del mismo.

5. Para desconectar el *joystick*, se ejecuta en la terminal:

```
$ sudo sixad stop
```

Una vez se haya conectado el *joystick* con el coche, podremos utilizarlo de forma inalámbrica. En la Fig. C.1 vemos la funcionalidad de cada uno de los botones del *joystick* de PlayStation.

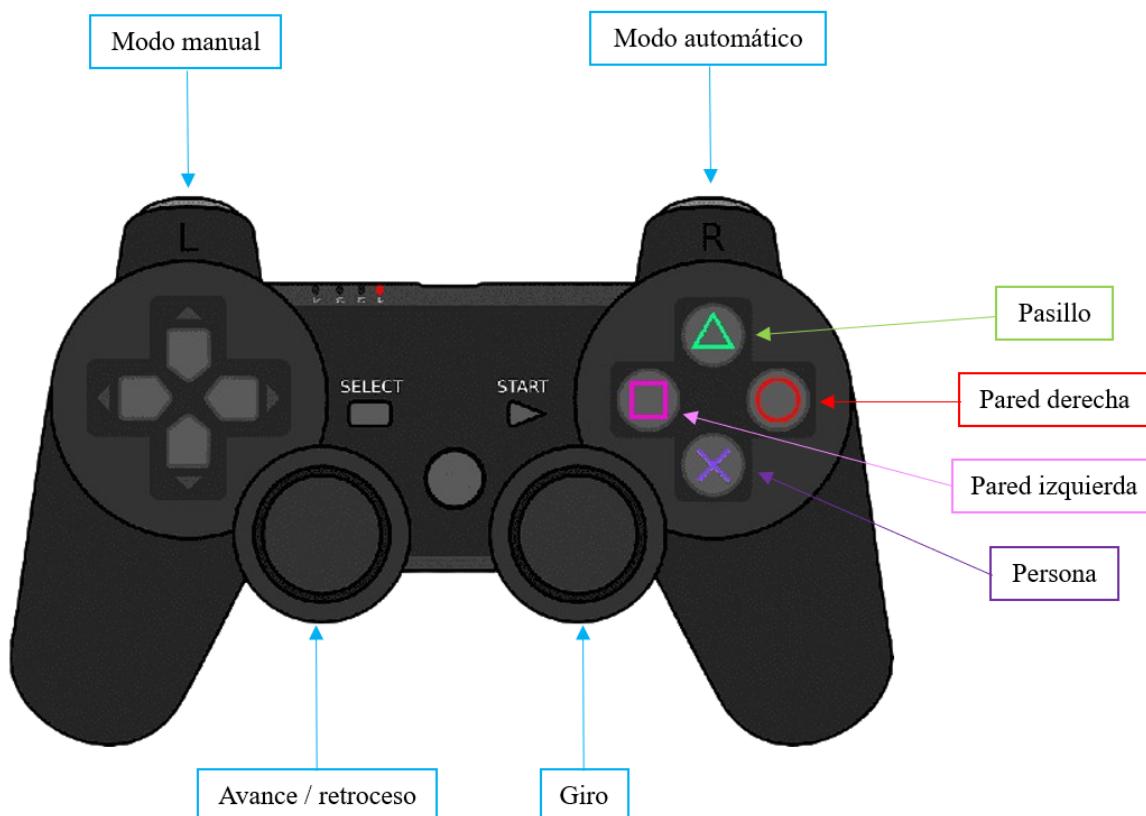


Figura C.1: Funcionalidades de los botones del *joystick*.



# **APÉNDICE D**

---

## **Software adicional**

---

### **D.1. Visual Studio Code**

Visual Studio Code (VS Code) es un entorno de desarrollo integrado (IDE) de código abierto y multiplataforma creado por Microsoft. Es una herramienta para escribir, editar y depurar código en numerosos lenguajes de programación.

Su principal característica es un editor de texto avanzado con soporte de autocompletado, resaltado de sintaxis y formato de código. Además, es personalizable y cuenta con un ecosistema de extensiones. Estas son útiles para agregar nuevas funcionalidades, integrar herramientas de desarrollo y para el soporte de lenguajes de programación.

Otras funcionalidades son: la integración con sistemas de control de versiones como Git, la depuración de código desde el editor y la incorporación de una terminal integrada.

### **D.2. PlotJuggler**

PlotJuggler es una herramienta rápida, intuitiva y extensible para la visualización de series temporales. Esta herramienta es de código abierto y cuenta con tutoriales propios y una serie de plugins.

Se va a utilizar este software para realizar las gráficas por varios motivos:

- Visualiza los datos de forma intuitiva y rápida, ya que su interfaz consiste en arrastrar y soltar los datos. Además, se pueden visualizar registros y datos fuera de línea y en tiempo real, lo cual es utilizado en el campo de la robótica.
- Transmite en tiempo real conectándose a una aplicación externa.
- Puede agregar múltiples fuentes de datos y funcionalidades. Incluyendo la adición de capa de transporte y/o protocolo propios.
- Se pueden aplicar funciones y transformaciones a series temporales para una mejor compresión de los datos. Esto es posible mediante transformaciones integradas y un editor de funciones para diseñar ecuaciones complejas.

La funcionalidad más importante es que incluye un plugin para ROS. PlotJugger está integrado tanto con ROS como con ROS2. Esto nos permite cargar *rosbags*, suscribirnos a tópicos, republicar mensajes y visualizarlos en RViz.

El programa se ejecuta escribiendo en la terminal:

```
$ rosrun plotjuggler plotjuggler
```

### D.3. RQt

RQt es un software framework de ROS que implementa diversas herramientas de interfaz gráfica de usuario en forma de plugins. RQt permite ejecutar en una misma ventana todas las herramientas, o lanzarlas como ventanas independientes.

Para utilizarlo, se debe ejecutar el siguiente comando en la terminal:

```
$ rqt
```

Esto nos abre la interfaz, donde se puede elegir los complementos disponibles en el sistema.

El sistema incluye algunas herramientas útiles como son el gráfico de nodos, el publicador de mensajes y la visualización de imagen, posición y entorno.

RQt se ha utilizado para publicar mensajes de movimiento al robot, con el fin de comprobar su funcionamiento. Además, el gráfico de nodos ha permitido ver las conexiones internas de ROS y comprobar que estas se han hecho correctamente de una forma visual.

También, cuenta con el plugin RViz, que es un framework de visualización 3D para ROS. Este se puede ejecutar independientemente en la terminal con:

```
$ rosrun rviz rviz
```

Esta herramienta es útil durante el desarrollo de este Trabajo Fin de Grado para la visualización de los datos del láser bidimensional.

---

## Bibliografía

---

1. F. Domínguez, *Construcción y control de bajo nivel de un coche a escala 1/10* (Trabajo Fin de Grado, Universidad de Málaga, 2018).
2. I. M. González, *Incorporación y calibración de sensores a un coche escala 1/10 en ROS* (Trabajo Fin de Grado, Universidad de Málaga, 2019).
3. J. M. O’Kane, *A gentle introduction to ROS* (University of South Carolina, 2014).
4. M. Quigley, B. Gerkey y W. D. Smart, *Programming Robots with ROS: a practical introduction to the Robot Operating System* (O'Reilly Media, Inc., 2015).
5. *Foro ROS* (<https://answers.ros.org/questions/>), Consultada en el año 2024.
6. *ROS Wiki* (<http://wiki.ros.org/es>), Consultada en el año 2024.
7. *Hokuyo UTM-30LX* (<https://www.hokuyo-aut.jp/search/single.php?serial=169>), Consultada en el año 2024.
8. J. Morales, J. L. Martínez y M. A. Martínez, Pure-Pursuit Reactive Path Tracking for Nonholonomic Mobile Robots with a 2D Laser Scanner. *EURASIP J. Adv. Signal Process* **2009**, (<https://doi.org/10.1155/2009/935237>) (2009).
9. *The Construct, Plannar Mover* ([https://bitbucket.org/theconstructcore/plannar\\_mover/src/master/](https://bitbucket.org/theconstructcore/plannar_mover/src/master/)), Consultada en el año 2024.
10. M. Boulet, O. Guldner, M. Lin y S. Karaman, *MIT Racecar* (<https://racecar.mit.edu/>), Consultada en el año 2024.
11. *Repositorio MIT RACECAR* (<https://github.com/mit-racecar>), Consultada en el año 2024.
12. MathWorks, *Ecuaciones de cinemática para robots móviles* (<https://es.mathworks.com/help/robotics/ug/mobile-robot-kinematics-equations.html>), Consultada en el año 2024.
13. K. M. Lynch y F. C. Park, *Modern robotics* (Cambridge University Press, 2017), cap. 13.3.
14. ROS, *Mensaje Ackermann en ROS* ([https://docs.ros.org/en/jade/api/ackermann\\_msgs/html/msg/AckermannDriveStamped.html](https://docs.ros.org/en/jade/api/ackermann_msgs/html/msg/AckermannDriveStamped.html)), Consultada en el año 2024.
15. ROS, *Mensaje escáner láser en ROS* ([https://docs.ros.org/en/groovy/api/sensor\\_msgs/html/msg/LaserScan.html](https://docs.ros.org/en/groovy/api/sensor_msgs/html/msg/LaserScan.html)), Consultada en el año 2024.
16. J. L. Cambil, *Navegación autónoma en circuito de un vehículo a escala con ladar utilizando ROS y GAZEBO* (Trabajo Fin de Grado, Universidad de Málaga, 2022).