# Simulación y control de un robot con ruedas con configuración Ackerman

---

Proyecto de curso de la asignatura Control y Programación de Robots

---


---

Realizado por Patricio De Mariano Aguilera
DNI: 29555267Z
4º GIERM

# Índice

# 1. Introducción

El primer apartado de esta memoria trata de aportar al lector contexto sobre la elección y desarrollo de este proyecto. Se explicarán brevemente los conocimientos bases requeridos para entender el trabajo y se definirán los objetivos impuestos a lo largo de su progreso.

Este trabajo consta de la simulación de un entorno virtual junto con un modelo de robot con ruedas con configuración Ackermann, el diseño e implementación de un generador de trayectorias y un controlador por Pure Pursuit, y por último la comparación del sistema completo ante diferentes escenarios de ruido y errores.

Para realizar el proyecto, uno se debe haber tenido que familiarizar con una serie de conceptos y teorías. El primero de esos términos que se debe conocer es la configuración Ackermann para ruedas en un vehículo.

La configuración Ackermann es un modelo de dirección utilizado comúnmente en vehículos terrestres que deben seguir trayectorias suaves sin deslizamiento lateral. A diferencia de otras configuraciones de ruedas, como el modelo diferencial (empleado por ejemplo en robots como TurtleBot) en el que las ruedas giran a diferentes velocidades para maniobrar y desplazarse, el modelo Ackermann simula el comportamiento de vehículos reales donde solo las ruedas delanteras giran para cambiar la dirección, mientras que las traseras se limitan a rodar.

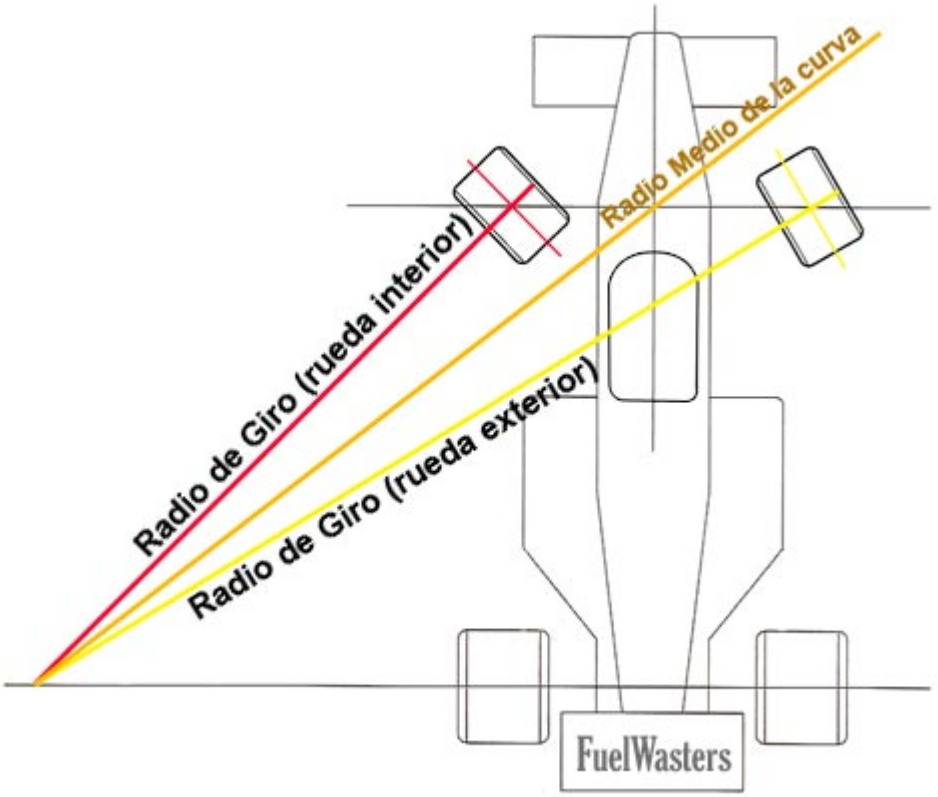


**Figura 1: Diagrama de configuración Ackermann**

Como se puede observar en la anterior figura, la particularidad de esta geometría radica en que, durante un giro, todas las ruedas apuntan hacia un único centro instantáneo de rotación. Esto se consigue ajustando el ángulo de las ruedas delanteras de modo que las líneas prolongadas de todas las ruedas se crucen en un mismo punto. Esta propiedad reduce el deslizamiento lateral de las ruedas y mejora la estabilidad al tomar curvas. Es especialmente útil en aplicaciones como simulación de coches autónomos o conducción en entornos realistas, donde se requiere un modelo cinemático más fiel al comportamiento de un vehículo real (en comparación con modelos como el diferencial).

El siguiente término que conviene aclarar es el del control por Pure Pursuit (Persecución/Búsqueda Pura en español). Este diseño de controladores es un algoritmo de seguimiento de trayectoria geométrico con gran relevancia en el sector de vehículos autónomos con dirección Ackermann. Su funcionamiento se basa en la idea de proyectar un punto objetivo (lookahead point) sobre la trayectoria deseada a una distancia fija por delante del vehículo, conocida como distancia de seguimiento (lookahead distance). El controlador calcula entonces el ángulo de giro necesario para dirigir el vehículo hacia ese punto como si se tratara de un arco circular, lo que resulta en trayectorias suaves y continuas.

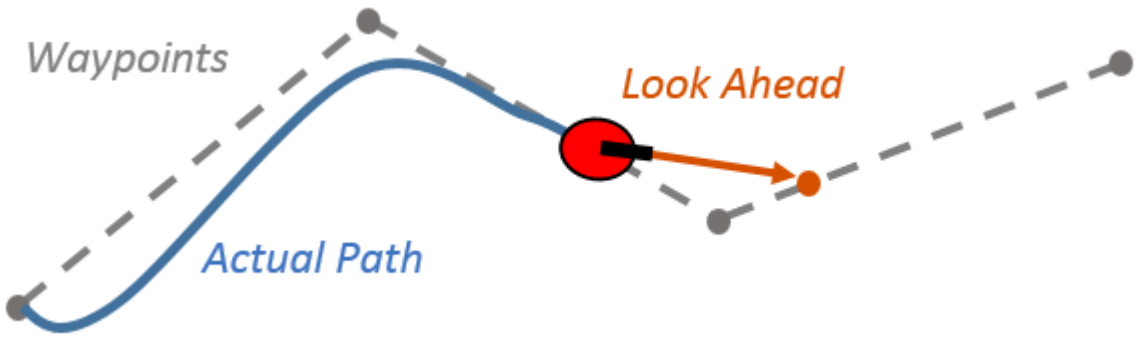


**Figura 2: Diagrama de Controlador Pure Pursuit**

Este enfoque es especialmente apropiado para vehículos con configuración Ackermann, ya que se conforma adecuadamente a sus restricciones cinemáticas: el Pure Pursuit genera comandos de dirección que no requieren giros instantáneos ni cambios bruscos, complementándose naturalmente con la geometría del sistema de dirección. Además, su simplicidad, estabilidad en curvas y facilidad de implementación lo hacen ideal para escenarios de navegación reactiva y simulaciones de conducción autónoma.

Por último, es esencial saber que el marco de desarrollo de este proyecto se ha llevado a cabo usando herramientas de uso común y libre en el sector de la robótica. Algunas de éstas son: ROS / ROS 2 (framework), Docker y rocker (aislamiento y portabilidad), RViz y OpenGL (visualización) o GitHub y Python (programación).

# 2. Modelo y entorno de simulación

En esta sección se pretende explicar el criterio de elección del entorno de simulación, así como del modelo de robot con ruedas con configuración Ackermann seleccionado para los experimentos. Debido a su amplio uso en la industria automovilística, no es de extrañar que haya multitud de opciones disponibles en el apartado de la robótica para simular. Por ésto, se siguieron una serie de requisitos para la elección del modelo y entorno de simulación de este proyecto.

Estos criterios fundamentales se establecieron con intención de garantizar lo máximo posible una integración fluida con ROS 2, facilitar la implementación de nodos personalizados y permitir una evaluación rigurosa del controlador. Los criterios fundamentales utilizados fueron los siguientes:

- **Modularidad y facilidad de implementación de nodos adicionales:** Era esencial que el entorno ofreciera una arquitectura limpia y modular que permitiera añadir y modificar nodos sin tener que alterar la base del simulador. Esto facilita el desarrollo iterativo y la experimentación con distintos enfoques de control y percepción.

- **Eficiencia computacional:** Al tratarse de un proyecto que requiere ejecutar múltiples pruebas con distintas condiciones y parámetros, se priorizó un entorno ligero, con tiempos de simulación rápidos y bajo consumo de recursos, evitando así cuellos de botella durante la evaluación.

- **Capacidad de visualización y personalización:** Se buscó un entorno que ofreciera herramientas visuales claras y configurables para representar la escena, el robot y su interacción con el entorno, además de los marcadores que se le irán añadiendo a lo largo del desarrollo. Esto permite identificar comportamientos no deseados de forma más intuitiva y analizar visualmente el rendimiento del sistema.

- **Acceso a parámetros estructurales y de diseño:** Se valoró el fácil acceso a información fundamental como el modelo del robot, el mapa del entorno y ficheros configurables, incluyendo su sistema de frames, sus tópicos de entrada/salida y los parámetros físicos del vehículo (dimensiones, masa, geometría de dirección, etc.). Estos datos son imprescindibles para implementar y evaluar correctamente algoritmos de control.

- **Capacidad sensorial adecuada:** Aunque el objetivo principal sea la navegación utilizando odometría, se valoró que el modelo del vehículo ofreciera también sensores realistas como LIDAR, cámaras o ultrasonidos. Esto abre la posibilidad de incorporar técnicas de percepción más avanzadas en fases posteriores, como evasión de obstáculos, SLAM o seguimiento visual de carriles.

Una vez establecidas las cualidades deseadas, se terminó seleccionando el entorno de simulación **f1tenth_gym_ros,** una herramienta de simulación basada en ROS 2 y Gazebo que permite simular vehículos con configuración Ackermann, utilizados habitualmente en competiciones y pruebas de algoritmos de conducción autónoma.

Este entorno combina la física del simulador **f1tenth_gym** con una interfaz completa en ROS 2, facilitando la integración con nodos personalizados de planificación, control y percepción. Para empezar, la facilidad de instalación destaca sobre la de muchas de las opciones valoradas debido a la disposición de una imagen de Docker montable tanto en una CPU común como en una GPU de NVIDIA integrada, permitiendo aumentar el rendimiento general del proyecto. Gracias al puente de comunicaciones que este repositorio establece con el entorno de simulación original, es posible integrar nodos adicionales, como generadores de trayectoria, controladores personalizados o simuladores de ruido y fallos, sin necesidad de modificar la lógica interna del simulador. Toda la estructura del entorno, incluyendo frames, tópicos, y parámetros del modelo, es transparente y está bien documentada, lo que facilita la depuración, el análisis de resultados y la conexión con otros módulos del sistema. Además, **f1tenth_gym_ros** permite la simulación de sensores relevantes como odometría, cámaras o LIDAR, e incluye compatibilidad con RViz 2, lo que permite visualizar tópicos clave relacionados como la trayectoria planificada, la odometría o los escaneos LIDAR. Por último, cabe destacar el realismo del modelo, que está basado en el coche radiocontrol comercial Traxxas Slash 4x4, ampliamente utilizado en investigación académica y competiciones de vehículos autónomos a escala 1/10 (al igual que el entorno de simulación).


**Figura 3: Demostración de un Traxxas Slash 4x4**

Una vez elegido el entorno de simulación y modelo de vehículo, cabe destacar la creación de un entorno virtual de Python para la instalación de las dependencias adecuadas con el fin de evitar conflictos innecesarios.

# 3. Desarrollo de módulos

En este tercer capítulo del documento se procederá a describir la creación, desarrollo y refinamiento de cada uno de los paquetes creados. Cada sub-apartado contendrá información sobre sus correspondientes nodos para ejecución, ficheros de configuración añadidos y modificados, y tópicos de suscripción y publicación que use cada uno para la comunicación entre ellos.

## 3.1. Paquete de simulación

El primer paquete presente en el proceso del trabajo es, obviamente, el paquete de simulación ya descrito en el apartado anterior. Para comenzar, el entorno virtual necesario para correr las simulaciones sin conflictos internos de versiones está creado en este paquete con el nombre de **venv**. Además de eso, también se dispone directamente del archivo **Dockerfile** necesario para construir la imagen del docker y del archivo **docker-compose.yml** requerido para simular si no se tiene el sistema operativo y la versión de ROS adecuada en la máquina local.

Una vez se tiene realizada la configuración inicial, cada contenedor lanzado con dicha imagen de docker compartirá el volumen del directorio de trabajo (workspace) con la máquina local, de forma que cualquier paquete, fichero o imagen que se cree por un medio será accesible por el otro para edición, representación y ejecución.

Dentro del paquete de simulación **f1tenth_gym_ros** hay multitud de archivos, por lo que con motivo de amenizar la memoria se tratarán únicamente los más relevantes; es decir, los archivos directamente utilizados y modificados para la realización de este proyecto. Fuera de éstos, cabe destacar la presencia de archivos de configuración e información sobre la comunicación establecida con el simulador original **f1tenth_gym**, explicado en forma de diagrama en el siguiente fichero (**f1tenth_gym_ros.png**):

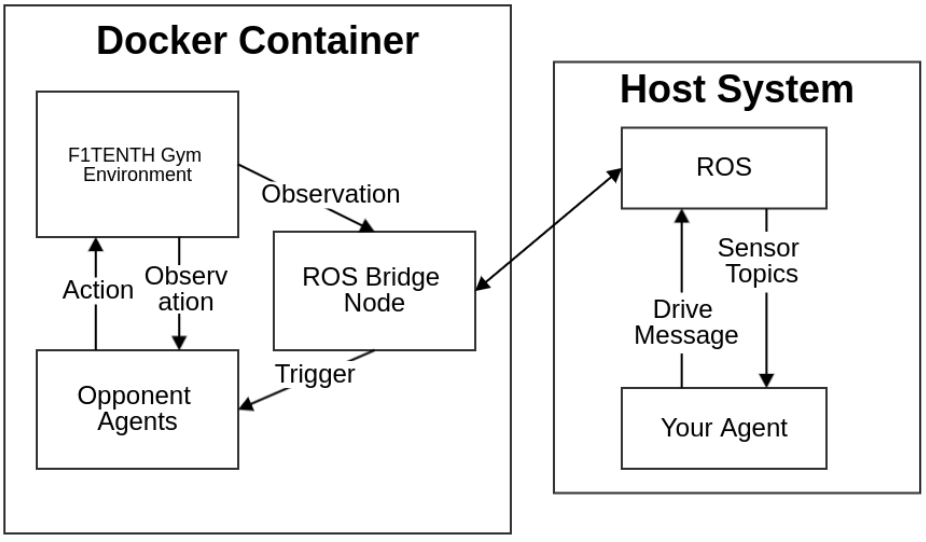


**Figura 4: Diagrama de comunicación del simulador**

Los archivos usados para el lanzamiento y la comunicación de la simulación con el marco de trabajo de ROS 2 son el nodo de simulación **gym_bridge.py** dentro de la carpeta **f1tenth_gym_ros** y el lanzador **gym_bridge_launch.py** dentro de la carpeta **launch**. Éstos se encargan de lo ya mencionado, utilizando el mapa y la configuración de RViz establecida en dichos archivos. Aquí entra la modificación de parámetros, ya que se querrán elegir los adecuados para la generación de trayectoria y visualización de tópicos en RViz más adelante.

A pesar de que en el simulador original hubieran disponibles multitud de circuitos y espacios mapeados para elegir, sólo se han importado dos de ellos en este paquete por motivos de ahorro de recursos. Los mapas elegidos han sido **levine** (que representa una distribución común de laboratorio) y **Spielberg** (que forma un circuito de carreras de Fórmula 1, como sugiere el nombre del repositorio). Para la elaboración de las pruebas y recogida de resultados, se ha utilizado únicamente **levine** debido al volumen de pruebas realizado, que se verá más adelante.

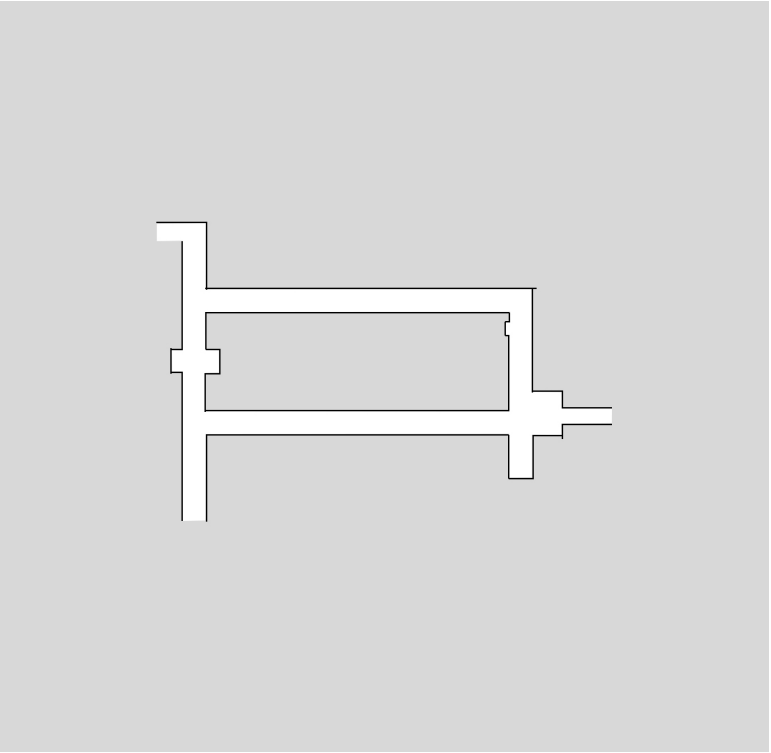

**Figura 5: Mapa “levine”**

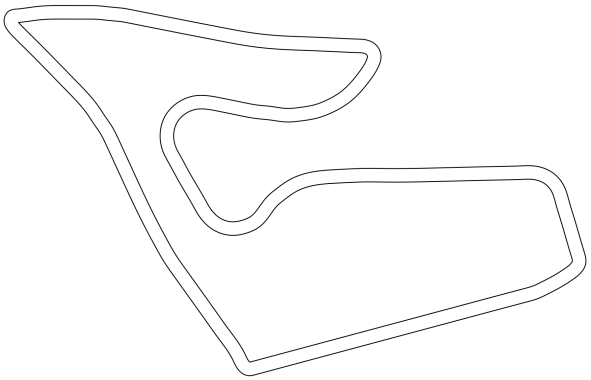

**Figura 6: Mapa “Spielberg”**

En cuanto a RViz, se pueden cargar dos configuraciones ubicadas en el mismo directorio que el lanzador: una con la composición base (**gym_bridge.rviz**) y otro con los tópicos relativos al generador de trayectoria y al controlador Pure Pursuit, como la trayectoria planeada, el punto objetivo o los marcadores de obstáculos (**gym_bridge_pp.rviz**). Se puede alternar entre configuraciones modificando el nodo de RViz dentro del launcher **gym_bridge_launch.py**.

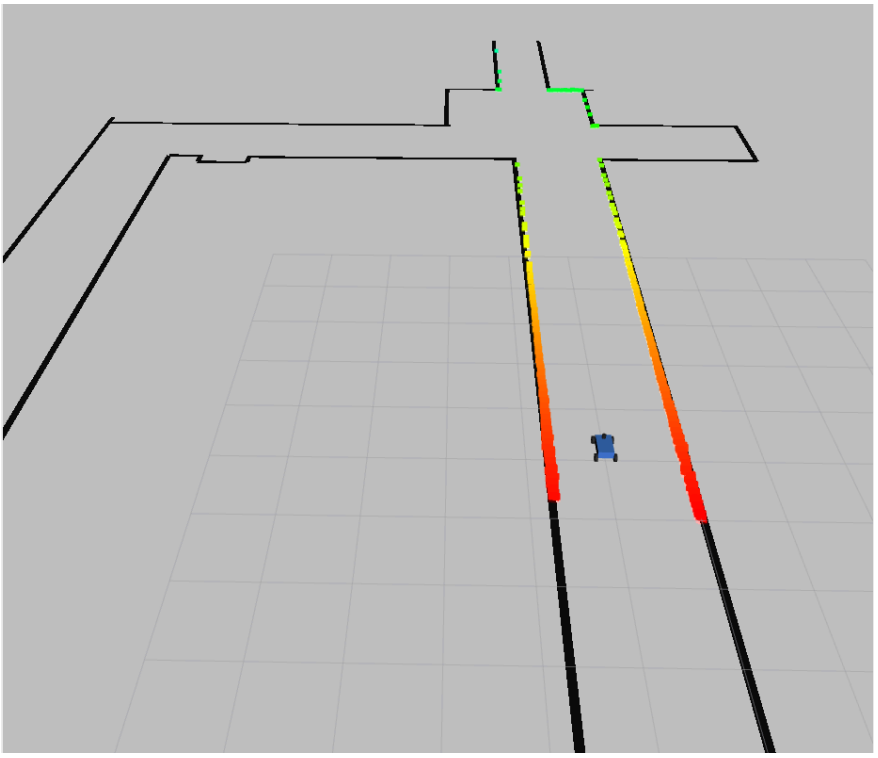

**Figura 7: Configuración base de RViz**

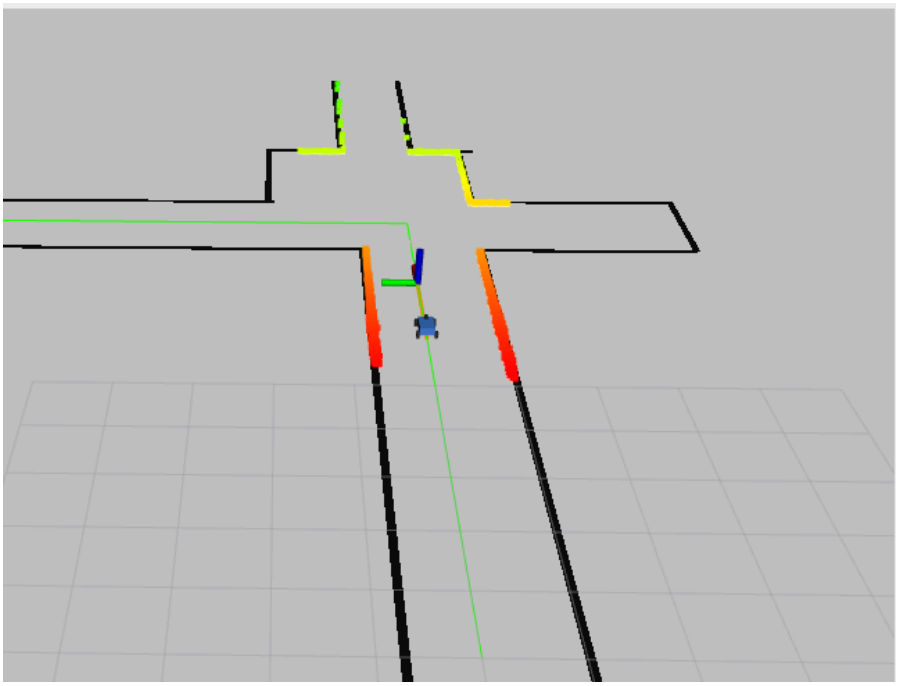

**Figura 8: Configuración avanzada para control**

Se continuará el análisis del paquete por los tópicos relacionados con su funcionamiento. De nuevo, sólo se tratarán de explicar los nodos destacados a la hora de trabajar en nuestro proyecto. Entre los múltiples tópicos disponibles en el entorno de simulación, existen algunos que resultan imprescindibles para la ejecución y supervisión del sistema, así como otros útiles durante las fases de prueba y depuración. A continuación se describen los más relevantes:

- **/scan (sensor_msgs/LaserScan)**: Proporciona la nube de puntos generada por el sensor LIDAR frontal simulado. Este tópico es clave para la percepción del entorno y se utiliza como entrada para los algoritmos de detección de obstáculos y de evasión reactiva en tiempo real.

- **/ego_racecar/odom (nav_msgs/Odometry)**: Contiene la estimación de la posición y velocidad del vehículo (llamado **ego racecar**). Es fundamental para la navegación, ya que informa al sistema de control sobre el estado actual del robot dentro del entorno simulado.

- **/drive (ackermann_msgs/AckermannDriveStamped)**: Es la salida del controlador, en forma de comandos de velocidad longitudinal y ángulo de dirección, siguiendo la cinemática de Ackermann. El vehículo ejecuta los comandos que recibe a través de este canal.

- **/map (nav_msgs/OccupancyGrid):** este tópico está disponible cuando se utiliza planificación basada en mapas, permitiendo el acceso a la representación del entorno en forma de grilla de ocupación. No se empleó activamente en todos los módulos del proyecto, pero se podrían implementar funcionalidades adicionales con ayuda de este tópico.

- **/tf y /tf_static (tf2_msgs/TFMessage):** Son tópicos internos del sistema de transformaciones de ROS. Permiten conocer las relaciones espaciales entre los diferentes frames del robot y del entorno. Su correcto funcionamiento es indispensable para garantizar la coherencia entre sensores, odometría y comandos de control.

Además de los tópicos descritos, también se han hecho uso de otros tópicos auxiliares con el fin de depurar código y conducir pruebas controladas a lo largo del desarrollo. Algunos de éstos son los siguientes:

- **/clock:** Publica el tiempo simulado. Es utilizado internamente por ROS para sincronizar nodos cuando la simulación no se ejecuta en tiempo real. Es crucial cuando se trabaja con use_sim_time := true. Para este proyecto se han simulado en tiempo real la mayoría de casos.

- **/clicked_point:** Permite seleccionar puntos en RViz. Es útil como herramienta de depuración, para marcar puntos de referencia visuales durante la simulación, por ejemplo.

- **/initialpose:** Permite reiniciar la posición estimada del robot, normalmente usado en combinación con herramientas de localización como AMCL. No se debe publicar aquí manualmente, pero puede emplearse desde RViz para pruebas controladas de localización.

Con esto se da por terminada el análisis del paquete de simulación, basado en los repositorios **f1tenth_gym_ros** y **f1tenth_simulator** del usuario **RoboRacer (formerly F1Tenth) Autonomous Racing Community** en GitHub. También se ha hecho uso del repositorio **rocker** del usuario **osrf** en GitHub para la ejecución de imágenes de docker soporte local personalizado inyectado para compatibilidad con tarjetas gráficas como NVIDIA. Enlaces respectivos:

- **f1tenth_simulator**: https://github.com/f1tenth/f1tenth_simulator.git

- **f1tenth_gym_ros**: https://github.com/f1tenth/f1tenth_gym_ros.git

- **rocker**: https://github.com/osrf/rocker.git

## 3.2. Paquete de generación de trayectoria

El paquete **trajectory_agent** es responsable de generar la trayectoria de referencia para el robot dentro del proyecto. En concreto, proporciona al sistema un camino cerrado predefinido, compuesto por cinco tramos rectilíneos en distintas direcciones, que el robot debe seguir. Esta trayectoria planificada complementa al paquete de simulación (**f1tenth_gym_ros**), actuando como la entrada deseada para el controlador del robot. Como se vio en el apartado anterior, el mapa consta de varios pasillos interconectados, por lo que la trayectoria tendrá al robot tomando curvas más abiertas o cerradas dependiendo de cómo esté posicionado respecto a las esquinas. Además, siempre se mantendrá a una distancia segura de la pared y lo más centrado en la trayectoria posible.

El primer fichero creado en este paquete fue **trajectory_node.py**, que se encarga de generar la trayectoria dictada por el método **generate_trajectory**. Este código crea un nodo llamado **trajectory_node** que publica en el tópico **/planned_trajectory** la trayectoria generada a una frecuencia dictada por el método **timer_callback**. Al lanzar dicho nodo en una terminal en paralelo a la representación visual de simulación, el nodo de generación de trayectoria publica su información en el tópico, por lo que con la configuración avanzada de RViz se dibujaría sobre el mapa a tiempo real.

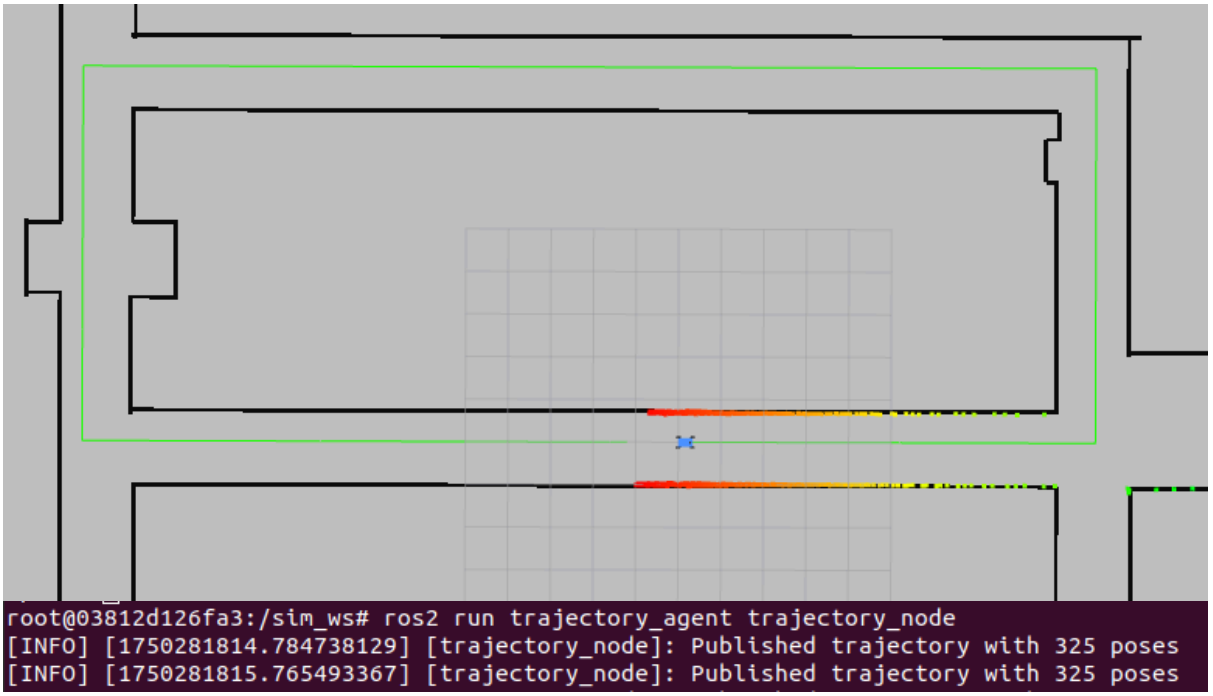


**Figura 9: Trayectoria representada sobre mapa (publicada cada 1s)**

A continuación se explicará detalladamente cómo funciona el nodo **trajectory_node**, haciendo hincapié en las clases de los mensajes y en los métodos internos de dicho nodo. El cálculo y publicación de mensajes se ha separado en tres principales pasos:

1. **Definición de las poses clave:** Se calculan cinco puntos de la clase **geometry_msgs/PoseStamped**, uno para cada extremo de tramo de la trayectoria. Cada **PoseStamped** incluye la posición (x, y, z) y la orientación (cuaternión) en un instante dado. Estas poses se eligen de modo que al conectar sucesivamente los tramos rectilíneos se forme una ruta cerrada.

2. **Construcción del mensaje Path:** Se crea un **objeto nav_msgs/Path** y se asigna su **header.frame_id** (por ejemplo **"map"**). A continuación, se llena la lista **poses** del mensaje con los cinco **PoseStamped** definidos. La especificación de ROS define **nav_msgs/Path** como un arreglo de poses (**PoseStamped**) que constituye la trayectoria planificada. Para cerrar la trayectoria, se añade (por práctica común) la primera pose al final del arreglo.

3. **Publicación periódica:** El nodo utiliza un temporizador (**self.create_timer**) con un intervalo fijo para invocar periódicamente la rutina de publicación. En cada llamada del callback, el **Publisher** publica el mensaje **Path** completo en el tópico **/planned_trajectory**.

Por último, el tópico **/planned_trajectory** transmite la trayectoria calculada en forma de mensaje **nav_msgs/Path**. Este mensaje contiene un encabezado (con marcos de tiempo y referencia) y un arreglo de **PoseStamped** que describen la ruta completa, que servirá como referencia de entrada para el controlador del robot. Dicho controlador interpretará la secuencia de poses como la ruta deseada.

## 3.3. Paquete de control por Pure Pursuit

Con todos los preparativos necesarios completados, el paquete de control por Pure Pursuit es el siguiente paso lógico en el desarrollo del trabajo. Este paquete contiene otros componentes del trabajo, como por ejemplo el lanzador global o los ficheros de resultados, pero en esta sección se explicará en específico la realización del nodo de control y todas las funcionalidades implementadas en éste.

Para empezar con el paquete **pure_pursuit_controller**, el script de configuración del paquete ha sido modificado para incluir las funcionalidades tanto del nodo de control **pure_pursuit_node** como el lanzador maestro que se verá más adelante (**master_launch.launch.py**).

Al igual que con la trayectoria, se describirá el proceso de creación del fichero correspondiente al nodo de control (**pure_pursuit_node.py**) paso a paso.

### 3.3.1 Control Pure Pursuit básico

La primera etapa en el desarrollo del controlador consistió en implementar una versión básica del algoritmo Pure Pursuit para el seguimiento de trayectorias en robots con geometría tipo Ackermann. Esta versión inicial se encarga de recibir una trayectoria objetivo, calcular un punto de avance (lookahead) apropiado, y generar comandos de velocidad y dirección en función de dicho punto. Este algoritmo se puede dividir en tres grandes bloques de funcionalidad:

**1. Recepción de odometría y trayectoria:**
El nodo se suscribe al tópico **/odom** para obtener la posición y orientación actual del robot, y a un tópico de tipo **nav_msgs/Path** que contiene la trayectoria a seguir. Para evitar procesar múltiples veces la misma trayectoria, se implementa un mecanismo de detección de repeticiones basado en un hash MD5 generado a partir de las coordenadas de los puntos.

**2. Bucle de control y lógica Pure Pursuit:**
El control se ejecuta dentro de un bucle periódico donde se calcula el punto de lookahead más adecuado, y se determina el ángulo de giro necesario para alcanzarlo. El cálculo se activa únicamente si el nodo ha recibido al menos una posición actual (**current_pose**) y una trayectoria válida (**trajectory**).

Al iniciar el seguimiento, se guarda el tiempo de inicio, útil para evaluar el desempeño más adelante. A continuación, se llama a **find_lookahead_point()** para buscar el punto objetivo. Si no se encuentra un punto válido (por ejemplo, si el robot ya ha llegado al final), se publica un comando de parada. Si se encuentra un punto válido, se convierte del sistema de coordenadas globales al sistema de referencia local del robot (**base_link**). Esta transformación permite calcular la curvatura hacia el punto objetivo con la fórmula del controlador Pure Pursuit. Finalmente, se publica el mensaje de control con la velocidad y ángulo calculados al tópico **/drive**.

**3. Cálculo del punto de lookahead**
La función **find_lookahead_point()** implementa la lógica para encontrar el primer punto de la trayectoria que esté al menos a una distancia **lookahead_distance** del robot. Primero se busca el punto más cercano al robot dentro de la trayectoria, y luego se recorre hacia adelante hasta encontrar uno suficientemente alejado. El parámetro **lookahead_distance** se podrá ajustar durante la evolución del proyecto para mejorar el rendimiento del control.

## 3.3.2 Visualización del punto de lookahead y la línea de guía en RViz

Con el objetivo de facilitar la depuración y evaluación visual del comportamiento del controlador Pure Pursuit, se añadieron elementos gráficos en RViz para mostrar el punto de lookahead y la línea de avance que une la posición actual del robot con dicho punto. Esto permite verificar si el controlador selecciona correctamente los objetivos a lo largo de la trayectoria.

Este añadido fue más sencillo que implementar el controlador desde cero, pero aún así se deben tener en cuenta los diferentes procesos realizados para conseguir este avance:

**1. Publicadores de visualización:**
Durante la inicialización del nodo, se crearon dos nuevos publicadores:

- Un publicador de tipo **PoseStamped** en el tópico **/lookahead_point**, que representa gráficamente el punto objetivo actual.

- Un publicador de tipo **visualization_msgs/Marker** en el tópico **/lookahead_line**, que dibuja una línea entre el robot y el punto de lookahead.

**2. Publicación del punto y línea de avance:**
Dentro del bucle de control, después de encontrar un punto de lookahead válido, se publica este punto (**target**) para su visualización. Además, se genera y publica un marcador tipo **LINE_STRIP** que conecta la posición actual del robot con el punto de lookahead. Esto se implementa en la función auxiliar **publish_lookahead_line()**.

**3. Generación del marcador de línea:**
La función **publish_lookahead_line()** utiliza el tipo **Marker.LINE_STRIP** para representar una línea en RViz. Esta línea está definida por dos puntos: la posición actual del robot y el punto de lookahead seleccionado. El resto de parámetros conforman el apartado visual y de permanencia del objeto.

**4. Configuración en RViz:**
Para visualizar estos elementos en RViz, es necesario:

- Añadir una visualización de tipo Pose y suscribirse al tópico /lookahead_point.
- Añadir una visualización de tipo Marker y suscribirse al tópico /lookahead_line.
- Asegurarse de que ambos mensajes estén referenciados al mismo sistema de coordenadas (normalmente map) y de que la hora de los mensajes coincida con el reloj del sistema (use_sim_time si se está usando simulación).

Estos cambios se reflejan en el nuevo script de configuración de RViz **gym_bridge_pp.rviz.**

### 3.3.3 Velocidad adaptativa en función de la cercanía al objetivo

Para mejorar la estabilidad y precisión del seguimiento de la trayectoria, se incorporó una lógica de velocidad adaptativa, que permite reducir la velocidad del vehículo a medida que se aproxima al final de la trayectoria. Esta estrategia minimiza oscilaciones y sobrepasos, y es especialmente útil en entornos con curvas cerradas o trayectorias pequeñas.

El primer paso consiste en calcular la distancia entre la posición actual del robot y el punto de lookahead, transformando ambos al mismo sistema de referencia (en este caso, se trabaja directamente en el marco global **map**, por lo que no se requiere transformación explícita).

Tras el cálculo de la distancia se procede con el ajuste dinámico de velocidad. La velocidad deseada del vehículo se ajusta según la distancia al punto objetivo. Si el robot está a menos de 1 metro del punto de lookahead, se reduce la velocidad máxima; de lo contrario, se mantiene una velocidad nominal más alta. Esto se implementa mediante una lógica condicional simple **if-else**. Esta velocidad ajustada se utiliza más adelante en la generación del mensaje de control **AckermannDriveStamped**, lo que garantiza que el robot reduzca su velocidad de forma automática conforme se aproxima al final de la trayectoria.

Puede parecer un cambio poco significativo, pero gracias a él se notó durante la simulación un aumento de precisión al finalizar respecto a la versión sin este ajuste. También se observó mejor control en curvas y mayor seguridad en general, ya que al permitir realizar aproximaciones suaves y ayudar a mantener estabilidad se detectaron menos obstáculos dada la misma trayectoria.

### 3.3.4 Detección y evasión reactiva de obstáculos

Para dotar al robot de capacidades de navegación más robustas, se añadió un mecanismo de detección y evasión reactiva de obstáculos utilizando el sensor LIDAR y una estrategia de desvío basada en reglas. Además, se incorporó un marcador visual en RViz para representar la posición del obstáculo frontal más cercano en tiempo real.

Empezando desde el inicio, en el constructor del nodo se añadió la suscripción al tópico **/scan**, que proporciona los datos del sensor LIDAR en forma de mensajes **sensor_msgs/LaserScan**. También se inicializaron parámetros clave para la detección como la distancia mínima de seguridad y el rango angular frontal a analizar, además de parámetros de evasión como flags indicadores de maniobrabilidad a la izquierda y derecha del rango angular.

El método **lidar_callback()** se encarga de procesar todos los datos declarados anteriormente. Por una parte, se tiene el procesamiento del LIDAR y detección frontal, encargado de analizar los rangos LIDAR dentro de un sector frontal de -30° a +30° para detectar obstáculos próximos. También se analiza si existen huecos libres en los laterales (±60° a ±90°) que permitan ejecutar maniobras evasivas. Por último, Cuando se detecta un obstáculo, se calcula su posición relativa en el marco **base_link**, y se publica un marcador de tipo esfera roja en el tópico **/obstacle_marker**, usando mensajes **visualization_msgs/Marker**. Este marcador permite visualizar en RViz la posición del obstáculo detectado directamente frente al robot, facilitando la depuración y análisis visual del comportamiento del sistema.

Dentro del bucle de control principal (**control_loop()**), si se detecta un obstáculo, se detiene la lógica de seguimiento de trayectoria y se ejecuta una maniobra evasiva sencilla o, en caso de no haber espacio a ninguno de los lados, se detiene en seco y comunica la situación acabando prematuramente la trayectoria.

## 3.3.5 Parámetros de evaluación del control

Con el objetivo de evaluar cuantitativamente el desempeño del controlador Pure Pursuit en cada prueba, se incorporaron mecanismos de registro automático de métricas clave. Estas incluyen el **tiempo total de trayectoria**, el **error de seguimiento acumulado** (**RMSE**), el **número de colisiones detectadas** y la **detección de fallos de finalización**. A continuación, se describe cómo fueron implementadas paso a paso en el nodo.

**1. Conteo de colisiones**
El nodo se suscribe a los datos del LIDAR a través de **lidar_callback()**. En este callback, si se detecta un obstáculo frontal (según lo descrito en el paso anterior), se evalúa si ha pasado suficiente tiempo desde la última colisión registrada para evitar duplicaciones.

**2. Cálculo del error de seguimiento (RMSE)**
Durante cada iteración de **control_loop()**, se calcula la distancia mínima entre la posición actual del robot y la trayectoria deseada (**self.trajectory**). Este error puntual se acumula para el cálculo posterior del RMSE. Además, se calcula el RMSE parcial cada ciclo, y se guarda junto con el tiempo transcurrido en un archivo CSV para su análisis posterior.
**3. Detección de finalización y evaluación**
Cuando se detecta que se ha alcanzado el objetivo (mediante **reached_goal()**), se detiene el vehículo, se calcula el tiempo total de trayectoria y se imprime el RMSE medio acumulado.

**4. Detección de fallo de finalización**
Si el robot no alcanza el objetivo dentro de un tiempo máximo permitido (por ejemplo, 60 segundos), se marca la prueba como fallida. Al finalizar una trayectoria (exitosa o no), se reinician las métricas para permitir nuevas pruebas.

Una vez se termina la simulación, se crea un archivo CSV llamado **results_YYMMDD_HHMMSS.csv** (año, mes, día, hora, minuto, segundo) en el directorio **sim_ws/src/pure_pursuit_controller/results**. Con dicho archivo se podrán graficar y comparar los diferentes casos probados en este proyecto.

### 3.3.6 Lectura de argumentos para configuraciones de ruido, bias y latencia

Para facilitar la comparación entre diferentes condiciones experimentales, se integró en el nodo **pure_pursuit_node.py** la posibilidad de modificar su comportamiento mediante parámetros configurables desde el archivo de lanzamiento. En concreto, se añadieron tres parámetros booleanos:

- **use_noisy_odom**: determina si se debe utilizar la odometría con ruido generada por el nodo de modificación **noisy_odom_publisher** (que se analizará en el siguiente apartado) en lugar de la odometría ideal.

- **use_drive_raw**: controla si el nodo debe publicar comandos en el tópico **/drive_raw** de forma que el nodo de modificación **steering_bias_node** (analizado en el siguiente apartado) introduzca sesgo en la dirección recibida por el control.

- **use_latency**: activa una frecuencia de control reducida para simular efectos de latencia o bajo rendimiento (5 Hz en lugar de 20 Hz).

Estos parámetros se declaran y leen al inicio del método especial **__init__()** del nodo, y a partir de los valores leídos se ajustan tres aspectos clave del comportamiento del nodo. Con esta arquitectura modular y parametrizable, se facilita el análisis sistemático del desempeño del controlador bajo diversas condiciones, lo que resulta especialmente útil para tareas de validación y comparación de algoritmos.

## 3.4. Paquete de inyección de errores

El último paquete creado dentro del espacio de trabajo del proyecto fue el encargado de inyectar errores en odometría y sesgo en los comandos de conducción, **odom_modifiers**. Este paquete consta principalmente por dos nodos encargados de realizar las funciones descritas anteriormente, declarados en **setup.py** como **noisy_odom_publisher** y **steering_bias_node** respectivamente.

### 3.4.1 Nodo de inyección de ruido gaussiano a la odometría

El primero en el proceso de creación fue el nodo **noisy_odom_publisher**, cuyo comportamiento se determina en el script **noisy_odom_publisher.py** y cuyos parámetros se pueden cambiar para emular una mayor o menor cantidad de ruido en la odometría. Este nodo se añadió para simular errores de percepción típicos en sensores reales (por ejemplo, localización inexacta debido a ruido en GPS, IMU o LIDAR), por lo que se desarrolló para introducir ruido gaussiano a la señal de odometría del robot. Gracias a esta adición, se puede evaluar la robustez del controlador Pure Pursuit frente a incertidumbre en la estimación del estado.

**1. Implementación del nodo**
Para comenzar, el nodo se suscribe a la odometría ideal publicada en el tópico **/ego_racecar/odom**. Éste está implementado como una clase que hereda de **Node**, y realiza las operaciones clave en su método **odom_callback**.

**2. Parámetros de ruido y modificación del mensaje**
Se definieron dos desviaciones estándar para controlar la magnitud del ruido introducido, una para la posición y otra para la orientación (**pos_noise_std** y **yaw_noise_std**). Estos valores pueden ser modificados si se desea ajustar el nivel de error simulado. Una vez definidos, el proceso de inyección de ruido se lleva a cabo de la siguiente manera:

1. Copiar el mensaje original recibido desde **/ego_racecar/odom**
2. Perturbar la posición sumando ruido gaussiano independiente a x e y
3. Perturbar la orientación: Primero se convierte el cuaternión original a ángulos de Euler (**roll**, **pitch**, **yaw**) utilizando **scipy.spatial.transform**. Tras eso se aplica un ruido gaussiano al ángulo **yaw** y, por último, se reconstruye un nuevo cuaternión a partir de los ángulos modificados.
4. Publicación del mensaje modificado: De esta forma, se garantiza que la odometría publicada por el nodo tenga características similares a las de sensores reales, incluyendo errores aleatorios tanto en la posición como en la orientación del vehículo

**3. Ejecución**
Este nodo puede ejecutarse directamente o incluirse como parte del archivo de lanzamiento maestro (**master_launch.launch.py**). La odometría modificada puede ser utilizada por otros nodos del sistema, en específico el controlador **pure_pursuit_node**, activando la opción **use_noisy_odom**.

### 3.4.2 Nodo de inyección de error sistemático en la dirección

En sistemas reales, los mecanismos de dirección pueden estar desalineados debido al uso prolongado, calibración incorrecta o defectos mecánicos. Para estudiar el impacto de este tipo de fallos en el comportamiento del vehículo, se diseñó un nodo (**steering_bias_node**) que simula un sesgo sistemático (bias) en el ángulo de dirección, emulando así una dirección permanentemente desviada respecto al centro ideal. Este tipo de error afecta directamente la trayectoria del vehículo, incluso si el controlador calcula correctamente el ángulo de giro deseado.

**1. Implementación**
Para comenzar, el nodo se suscribe a las señales de control de dirección sin procesar, publicadas en el tópico **/drive_raw**. En su constructor, el nodo establece aparte de eso un valor fijo de desplazamiento (**offset**) en 0.1 radianes (~5.7°), que se suma sistemáticamente al ángulo de dirección y la publicación del mensaje modificado en el tópico **/drive**.

**2. Modificación del mensaje**
Cada vez que se recibe un mensaje de dirección, se modifica directamente el campo **steering_angle** sumando el sesgo definido. De esta manera, incluso si el controlador determina correctamente el ángulo deseado, el vehículo ejecutará una dirección desviada en todos los comandos, lo que permite observar efectos como deriva sistemática, errores acumulativos y desviaciones en trayectorias nominales.

**3. Ejecución**
Al igual que el nodo de inyección de ruido en la odometría, su integración es sencilla dentro de cualquier sistema de simulación modular. Simplemente se redirige la salida del controlador hacia **/drive_raw** y se deja que el nodo de bias se encargue de modificarla antes de enviarla al simulador mediante **/drive**. Este proceso también se puede automatizar haciendo uso del lanzador con el valor adecuado en el argumento **use_drive_raw**.

# 4. Lanzador y control de versiones

En este breve apartado se compartirá el control de versiones manualmente realizado a lo largo de este trabajo para asegurar puntos clave del desarrollo del control, de la generación de trayectorias, de la generación de errores y de la unificación de nodos en el lanzamiento.

## 4.1. Versiones anteriores

En la carpeta **Trabajo CPR Docs** se encuentran varios documentos informativos del camino de desarrollo tomado, además de versiones archivadas de las diferentes versiones del control y del espacio de trabajo completo. Empezando por los documentos, el flujo de trabajo según éstos fue el siguiente:

1. **Setup.txt**: Documento con la configuración necesaria para comenzar a trabajar y procesos que sólo deberán realizarse una vez.
2. **Sim.txt**: Documento con los pasos para iniciar la simulación y teleoperación.
3. **Topics.txt**: Información de los tópicos de la simulación por f1tenth.
4. **Traj.txt**: Documento con los pasos para lanzar la generación de trayectoria.
5. **Ctrl.txt**: Documento con los pasos para lanzar el control. Además, incluye la lista de parámetros modificables para hacer “fine tuning” y otra lista de las versiones archivadas de controladores **pure_pursuit_node.py** (guardadas en la carpeta **Pure_Pursuit_versions**).
6. **Error.txt**: Documento con la creación e implementación inicial de errores de modelado (más adelante se implementó con el uso de argumentos en el comando **launch**).
7. **Upgrade.txt**: Esquema de mejora de sistema base / Plan general de evolución del sistema. Cada propuesta está valorada con un nivel de prioridad diferente según su importancia, y se marcará como hecha si se termina de implementar en el sistema.
8. **Comp_metrics.txt**: Documento con las métricas utilizadas para evaluar el desempeño del controlador bajo diferentes pruebas. También está escrito cómo configurar los ficheros de resultados para crear gráficas con los códigos correspondientes.
9. **Results.txt**: Documento con los resultados de simulaciones de cada configuración, con el método de prueba realizado para cada una y con los pasos para evaluar individualmente o en comparación con otras pruebas los resultados deseados.

Notoriamente se encuentran en la misma carpeta tres diferentes espacios de trabajo archivados:

- **sim_ws_backup1**: Workspace base con paquetes y nodos básicos creados.
- **sim_ws_backup2**: Workspace completo con todas las mejoras realizadas antes de la unificación de lanzadores
- **sim_ws_backup_final**: Workspace completo con unificación de lanzadores y gráficas.

Por último, la carpeta **Pure_Pursuit_versions** contiene versiones archivadas de controladores **pure_pursuit_node.py**, que vienen descritas en el fichero **Ctrl.txt**.

## 4.2. Versión definitiva

Con la finalidad de agilizar el proceso de pruebas, evaluación y comparación se realizaron diferentes archivos auxiliares. Éstos se encuentran ubicados en el paquete de control **pure_pursuit_controller**, donde la mayoría de las optimizaciones se programan.

Por una parte, se mejoró significativamente el número de acciones necesarias para lanzar la simulación con todos los nodos y configuraciones necesarias gracias al fichero **master_launch.launch.py**. Este código se encarga no solo de lanzar de forma unificada todos los nodos de la configuración base (sin errores), sino también de leer argumentos de entrada y cambiar dicha configuración según los valores introducidos. Las opciones de configuración son las siguientes:

- **use_noisy_odom**: Activa la odometría con ruido, lanzando el nodo de inyección de ruido correspondiente y suscribiendo el controlador al tópico de odometría ruidosa.
- **use_drive_raw**: Activa la publicación en **/drive_raw** por parte del control y lanza el nodo de inyección de sesgo correspondiente, haciendo que se publique los comandos de control sesgados a la simulación.
- **use_latency**: Activa el modo con latencia, reduciendo significativamente la frecuencia de control del nodo **pure_pursuit_controller**.

Por otra parte, se introdujeron herramientas de representación gráfica y comparación para los ficheros de resultados creados tras cada iteración del lanzamiento de simulación completo. Estos ficheros (explicados en la sección “Parámetros de evaluación del control”) pueden analizarse individualmente mediante el código **analizar_resultados.py**, o compararse entre sí (con un total de cuatro ficheros representados) mediante **comparar_resultados.py**. Ambos hacen uso de la biblioteca **Matplotlib**, que debe instalarse en cada contenedor nuevo que se cree en el docker.

El código **analizar_resultados.py** se utiliza para evaluar el rendimiento individual de la configuración de control respectiva. Al ejecutarse, se crean de salida dos gráficos en el mismo directorio: Uno representa la evolución del error de seguimiento (RMSE) respecto al tiempo (**grafica_rmse_vs_tiempo.png**) y otro representa los valores estadísticos más interesantes de dicho error de seguimiento (**grafica_estadisticas_rmse.png**).

El código **comparar_resultados.py** se debe ejecutar únicamente con otros cuatro ficheros de resultados en su mismo directorio. Estos ficheros deben estar organizados alfabéticamente de forma que queden en este orden de configuraciones: sin error, con ruido en la odometría, con sesgo de dirección y con latencia de control. Si se cumplen los requisitos, este proceso da como salida un gráfico de evolución del error de seguimiento (RMSE) respecto al tiempo con los cuatro resultados simultáneamente representados (**grafica_comparativa_rmse.png**).

En la carpeta **graphs** se guardarán las gráficas a analizar en los siguientes apartados de la memoria, mientras que en la carpeta **tests** se archivarán los resultados de las simulaciones evaluadas y comparadas.

# 5. Resultados

En esta última gran sección de la memoria se evaluará el rendimiento de cada simulación según una serie de métricas de comparación. Primero se explicarán los criterios de evaluación de cada configuración y las formas de representación elegidas para lograrlo, y tras eso se expondrán los resultados visualmente en los apartados correspondientes.

## 5.1. Métricas de comparación

En este proyecto se han elegido medir las siguientes métricas de desempeño clave:

- **Tiempo total [seg]**: Tiempo que tarda el robot en completar la trayectoria.
- **Error de seguimiento [m]**: Desviación media entre el robot y la trayectoria deseada y evolución de la desviación parcial por cada bucle de control.
- **Número de colisiones**: Cuántas veces se detecta un obstáculo o pared a lo largo de la trayectoria.
- **Fallo de finalización [true/false]**: Si no completa la trayectoria en el tiempo establecido (por atasco, desvío excesivo, seguimiento subóptimo, etc.).

Estos cuatro parámetros se comunican por el terminal al final de cada trayectoria completada exitosamente, y se representarán en una tabla junto a los valores obtenidos de cada configuración para su comparación. El error de seguimiento parcial por bucle de control se utilizará para graficar su evolución respecto al tiempo. Además, de ésta última métrica se extraerán los valores máximo, mínimo y medio para su representación.

## 5.2. Análisis individuales

En este apartado se mostrarán todas las gráficas evaluativas representadas para cada configuración. Cada gráfica estará archivada dentro de la carpetas **graphs** en el directorio de su configuración respectiva (**sin errores**, **noisy odom**, **steering bias** o **latency**). Las gráficas de este apartado se realizaron mediante la script **analizar_resultados.py**.

### **A)** Configuración sin errores

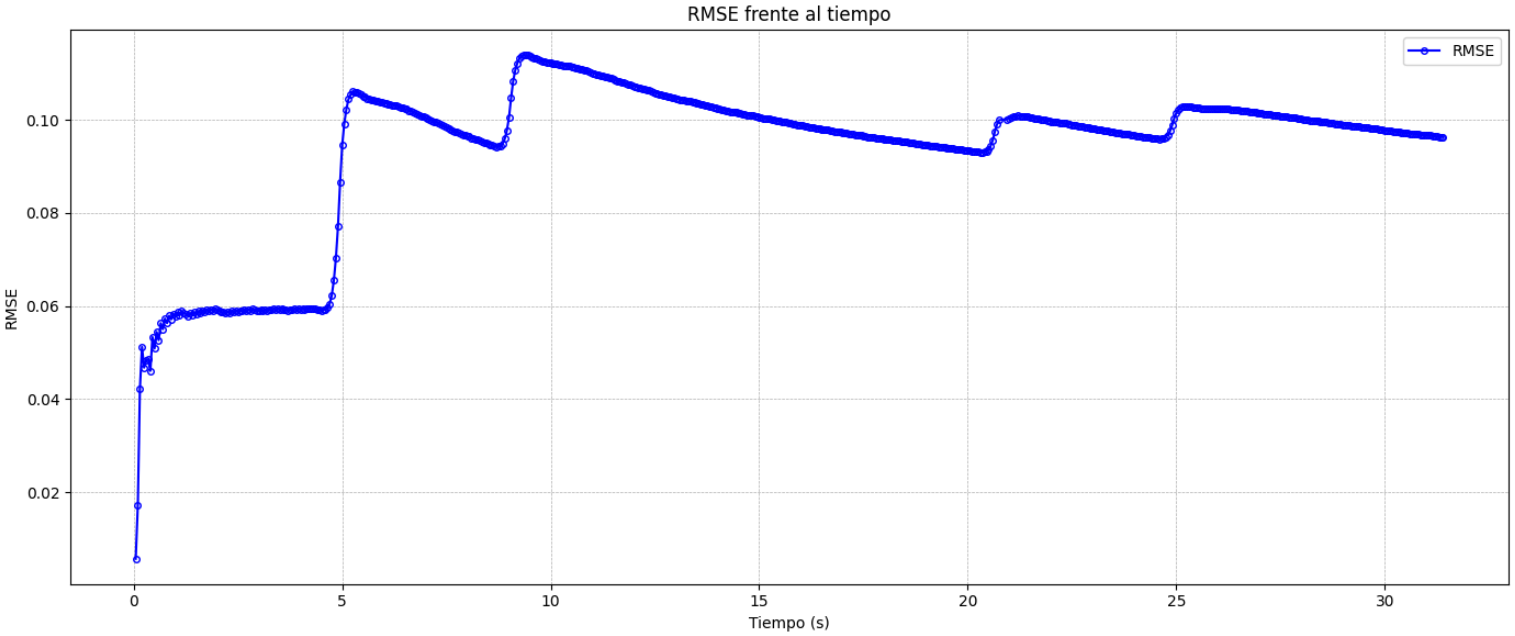


**Figura 10: Error de seguimiento frente al tiempo (caso sin errores)**

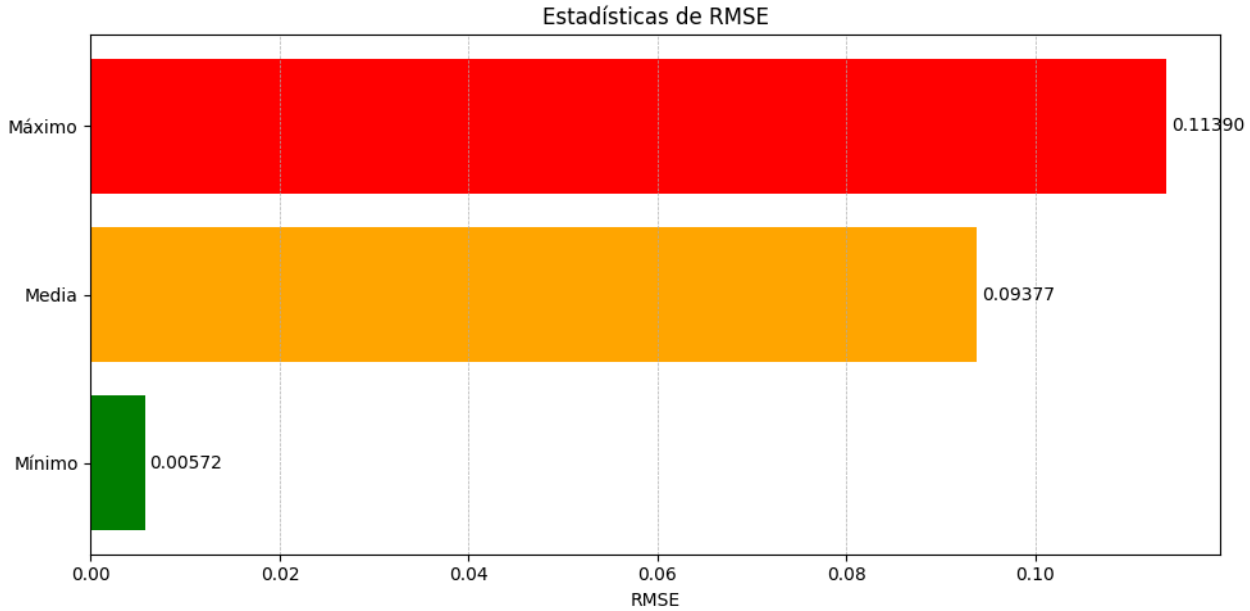


**Figura 11: Estadísticas del RMSE (caso sin errores)**

Los valores de las métricas de desempeño clave en este experimento fueron las siguientes:

- Trayectoria completada en **31.40 segundo**s
- Error de seguimiento (RMSE): **0.092 metros**
- Número de obstáculos encontrados: **1**
- Fallo de finalización: **False**

El vídeo demostrativo del experimento está subido en YouTube siguiendo este enlace.

**B)** Configuración con ruido en odometría

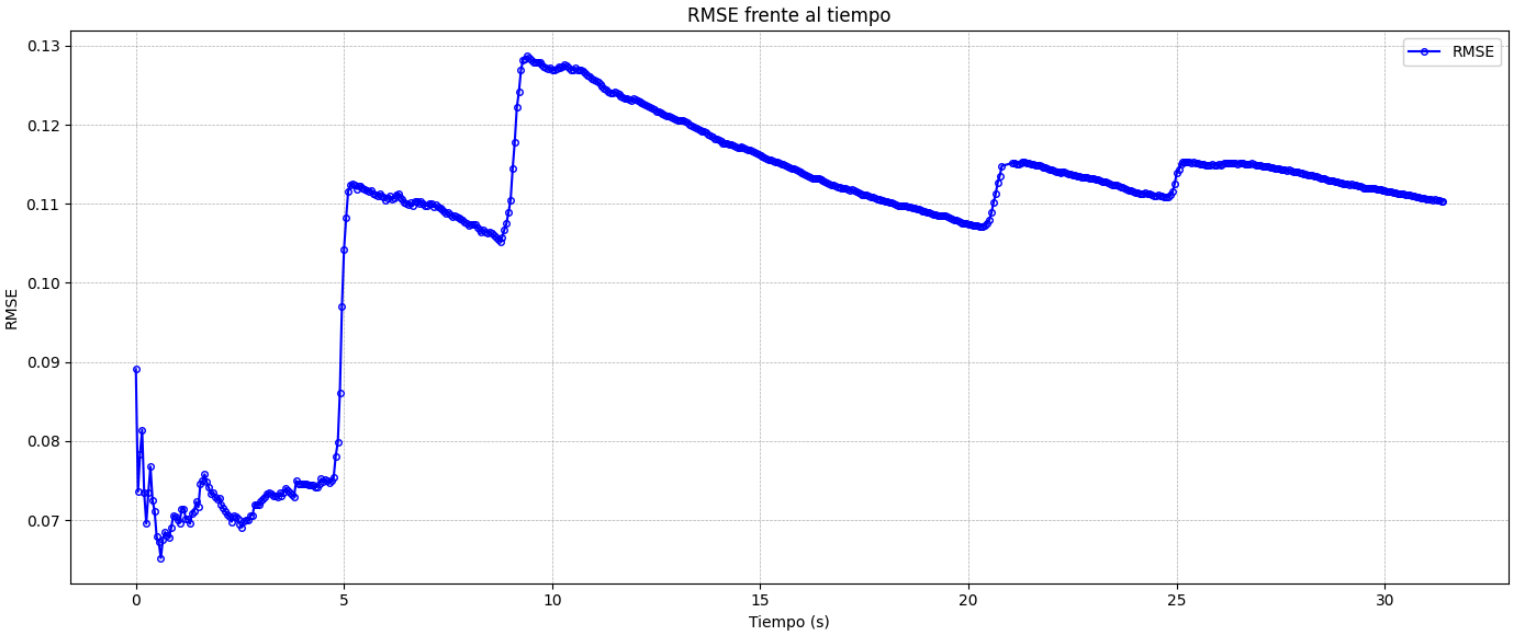


**Figura 12: RMSE frente al tiempo (caso con ruido en odometría)**

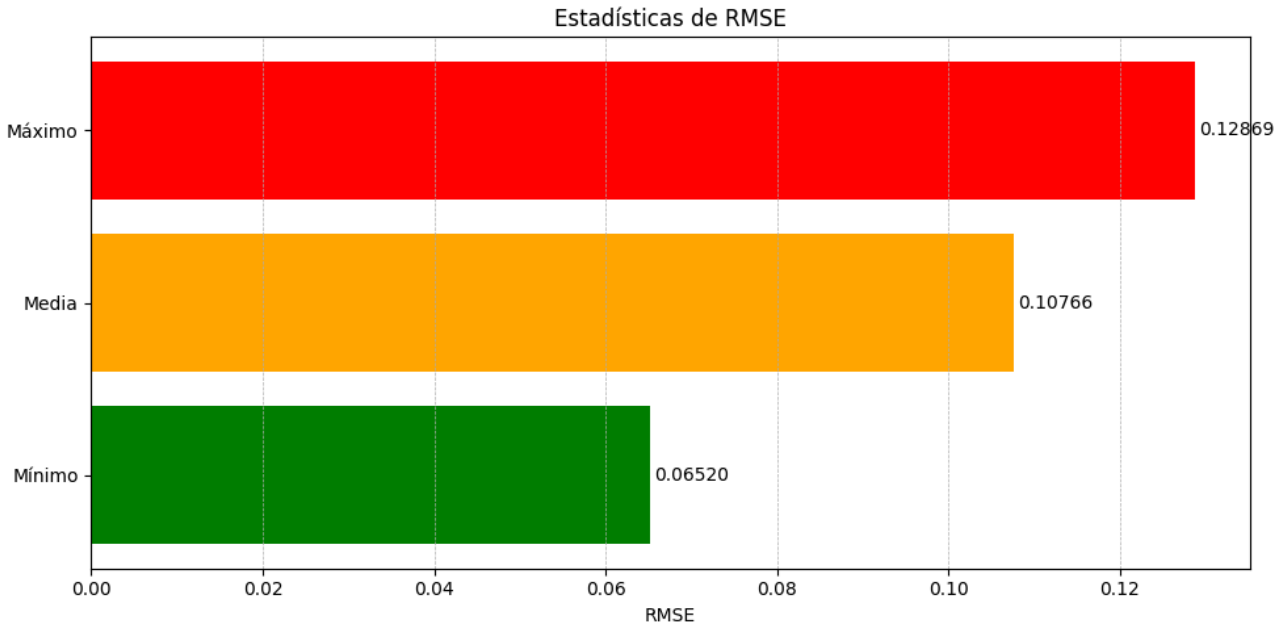


**Figura 13: Estadísticas del RMSE (caso con ruido en odometría)**

Los valores de las métricas de desempeño clave en este experimento fueron las siguientes:

- Trayectoria completada en **31.40 segundo**s
- Error de seguimiento (RMSE): **0.110 metros**
- Número de obstáculos encontrados: **1**
- Fallo de finalización: **False**

Como se puede comprobar en estos resultados, con un nivel moderado de ruido en odometría no se ve afectado prácticamente el controlador. Pese a eso, sí que se nota cómo el error RMSE medio aumenta, sobre todo debido a que la mínima es bastante superior a la del caso anterior.

El vídeo demostrativo del experimento está subido en YouTube siguiendo este enlace.

**C)** Configuración con sesgo de dirección

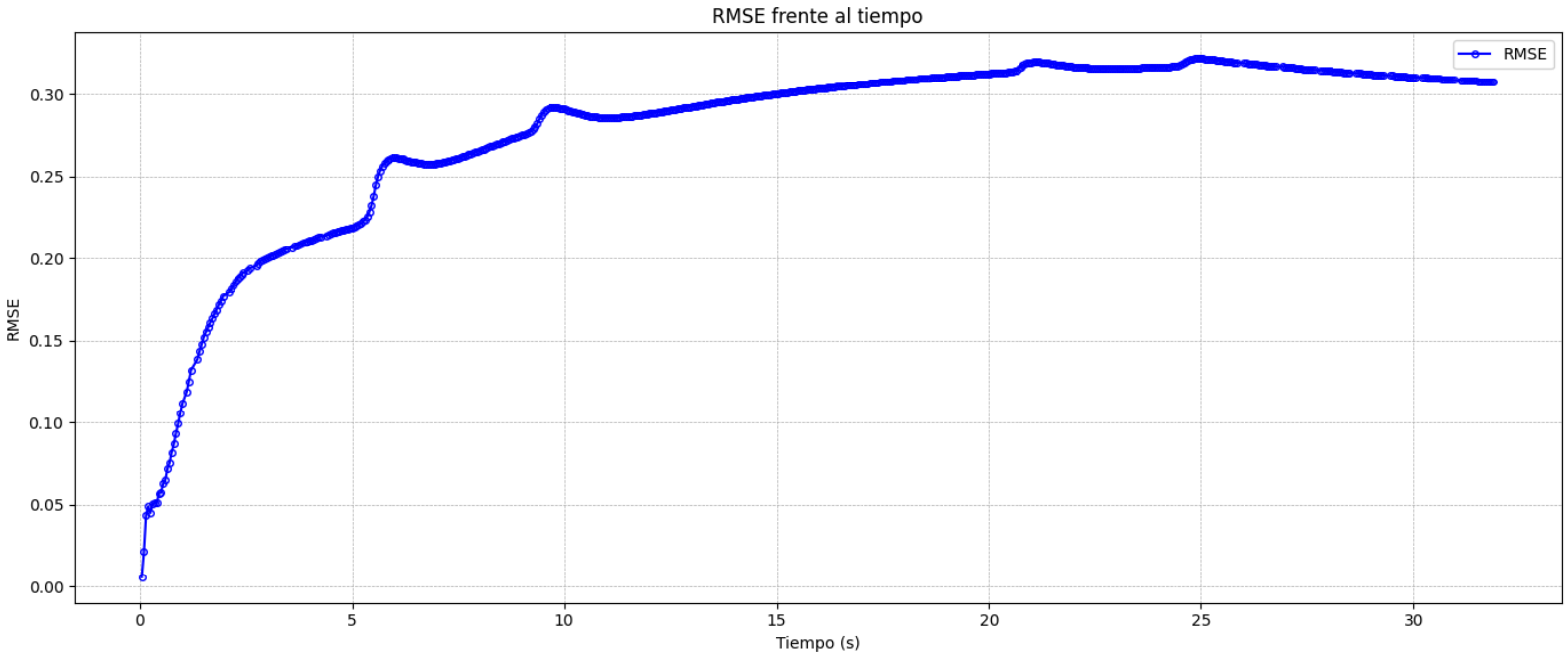


**Figura 14: RMSE frente al tiempo (caso con sesgo de dirección)**

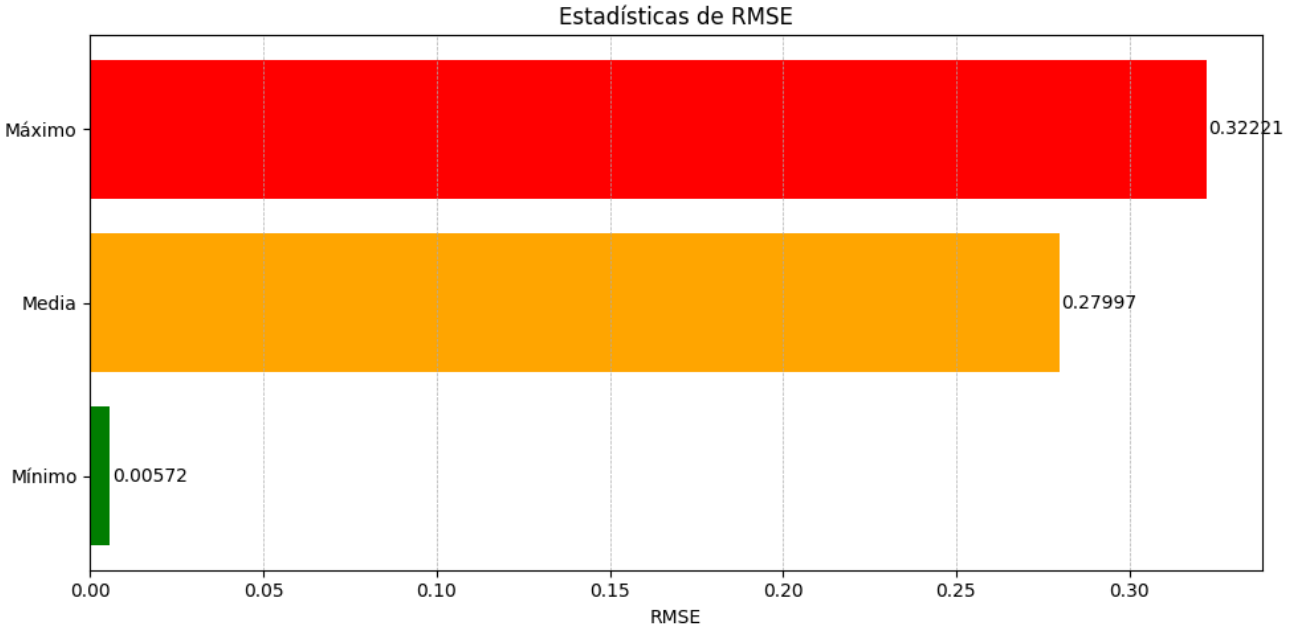


**Figura 15: Estadísticas del RMSE (caso con sesgo de dirección)**

Los valores de las métricas de desempeño clave en este experimento fueron las siguientes:

- Trayectoria completada en **31.90 segundo**s
- Error de seguimiento (RMSE): **0.308 metros**
- Número de obstáculos encontrados: **7**
- Fallo de finalización: **False**

Este experimento muestra cómo un error de calibración en el eje de direccionamiento del robot puede causar un desvío significativo de la trayectoria deseada. También cabe destacar que, al no seguir la trayectoria a través de los pasillos estrechos, el robot ha tenido que maniobrar mucho más frecuentemente para mantener la distancia de seguridad respecto a las paredes.

El vídeo demostrativo del experimento está subido en YouTube siguiendo este enlace.

### **D)** Configuración con latencia

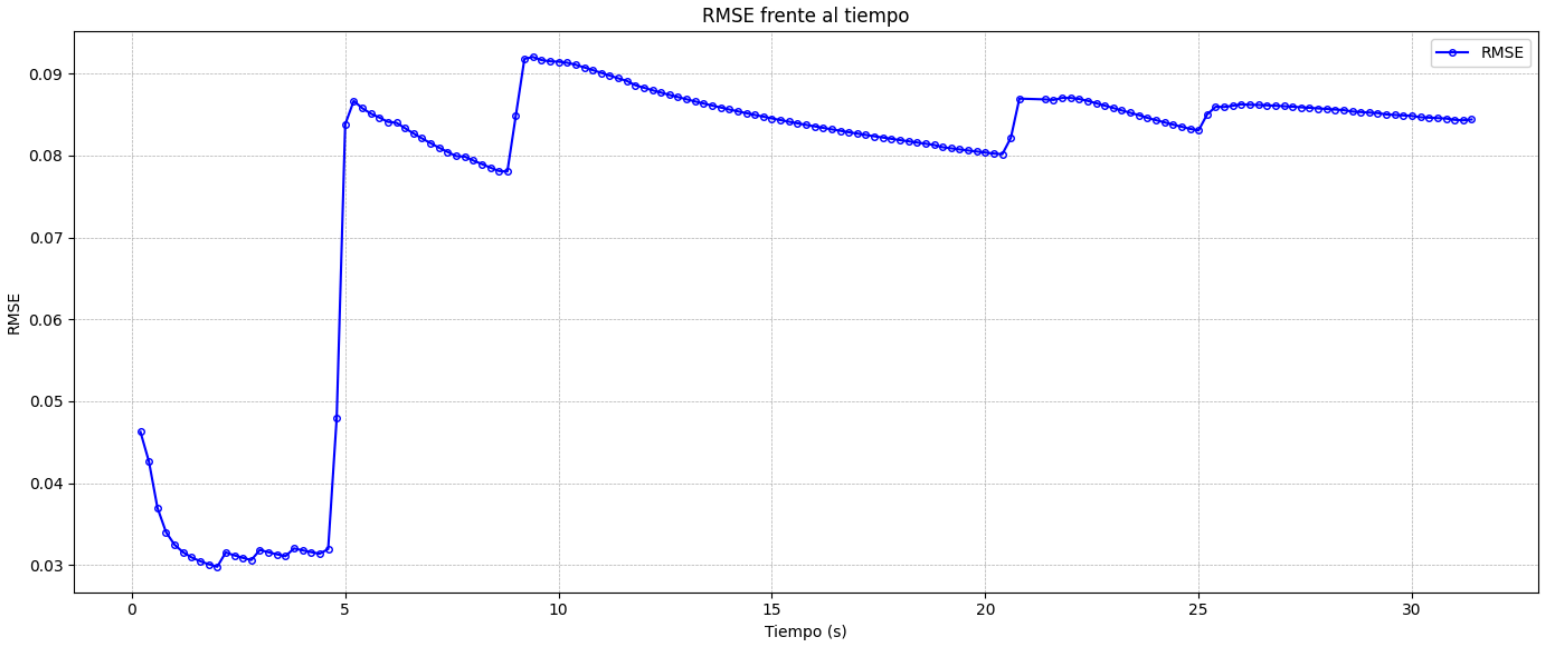


**Figura 16: RMSE frente al tiempo (caso con latencia)**

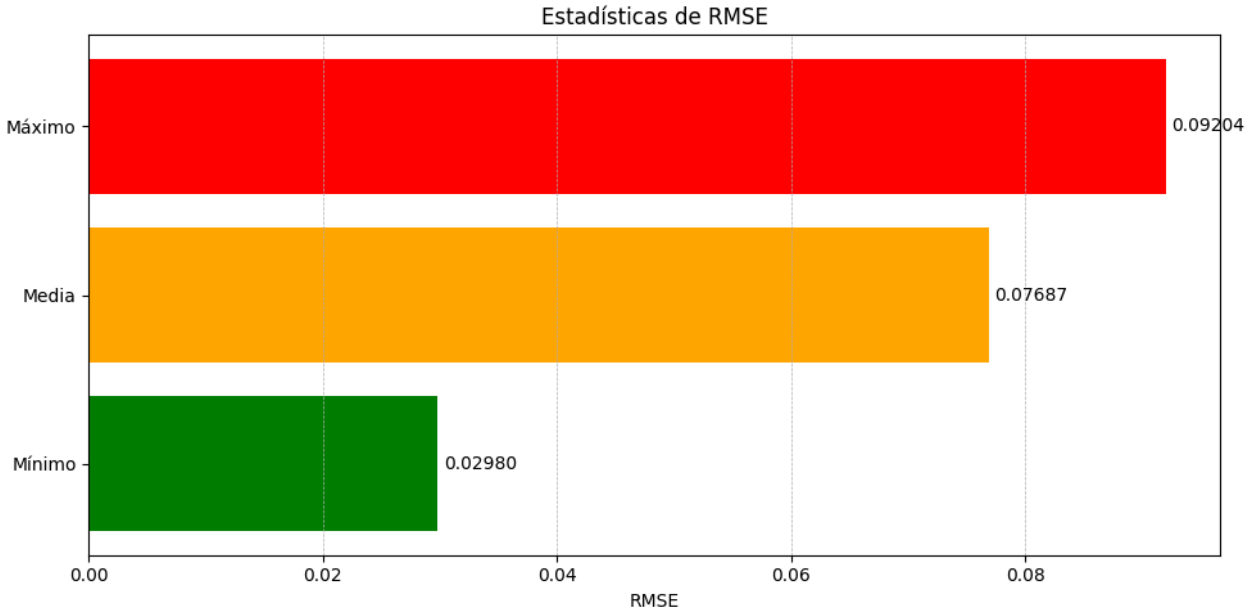


**Figura 17: Estadísticas del RMSE (caso con latencia)**

Los valores de las métricas de desempeño clave en este experimento fueron las siguientes:

- Trayectoria completada en **31.40 segundo**s
- Error de seguimiento (RMSE): **0.084 metros**
- Número de obstáculos encontrados: **1**
- Fallo de finalización: **False**

Uno puede fijarse en estos resultados y darse cuenta de que este experimento ha tenido mejores resultados (según las métricas de comparación) que el experimento sin ningún error. Ésto se debe (además de la variabilidad de prueba a prueba) a que la mayoría de la trayectoria es recta y no sufre por tener menor frecuencia de actualización, mientras que en los giros ha coincidido la frecuencia de actualización con una mejor alineación para tomar las curvas más rápido. Sin embargo, ésto provoca que el error mínimo detectado sea ligeramente mayor que en el caso sin errores.

Se puede concluir, por ende, en que la presencia de latencia por sí sola no causa problemas al sistema de control. Sin embargo, en el siguiente experimento se verá cómo su presencia dificulta de manera significativa el tratamiento de otros errores presentes en el sistema.

El vídeo demostrativo del experimento está subido en YouTube siguiendo este enlace.

**E)** Configuración realista (peor caso)

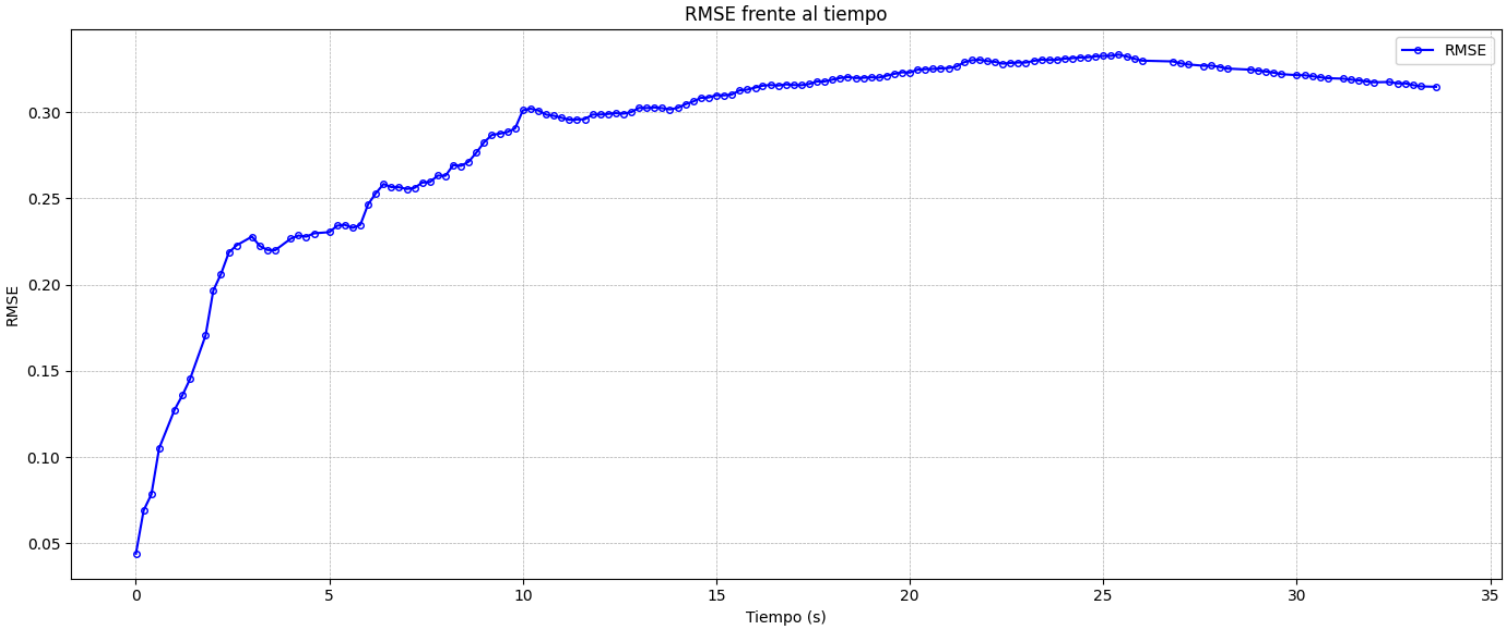


**Figura 18: RMSE frente al tiempo (peor caso)**

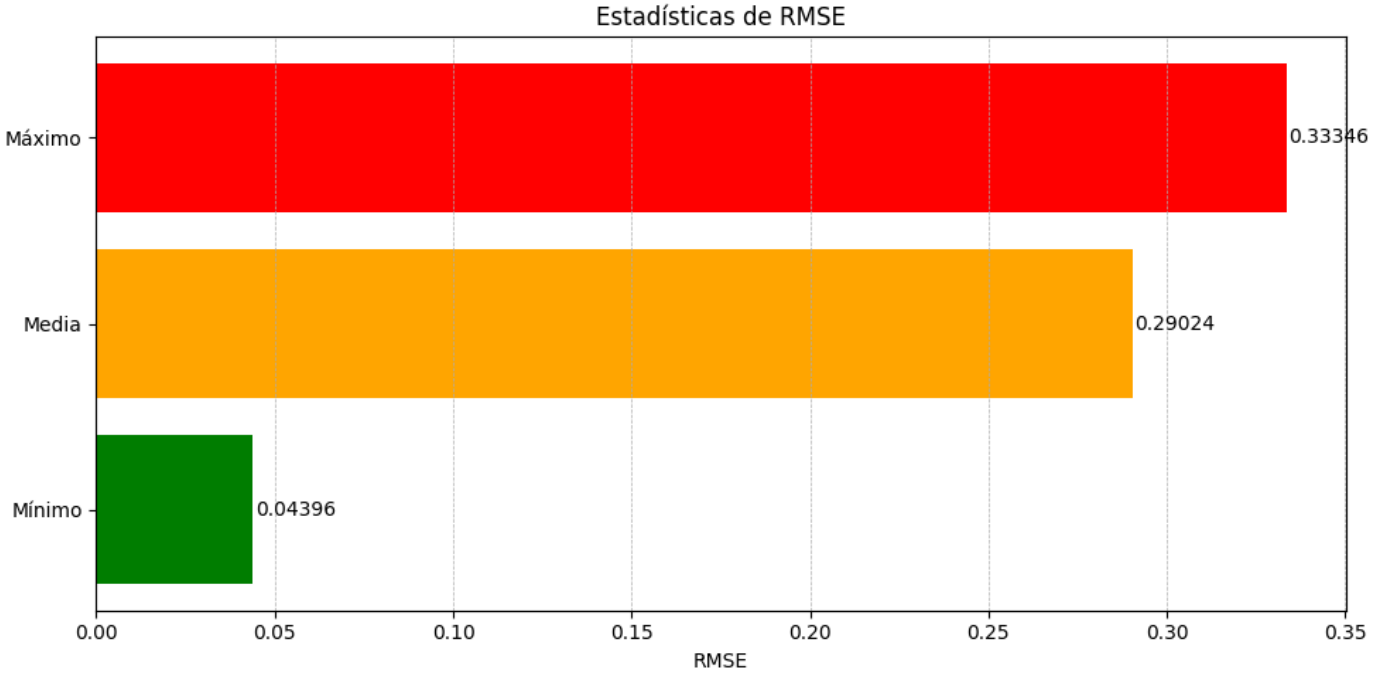


**Figura 19: Estadísticas del RMSE (peor caso)**

Una vez se establece el peor caso posible, en el que todos los errores anteriormente mencionados están presentes, es cuando se puede presenciar la influencia que éstos pueden tener sobre nuestro sistema de control. La robustez de éste sistema se juzgará por su capacidad de mitigación de estas variantes no controlables.

Los valores de las métricas de desempeño clave en este experimento fueron las siguientes:

- Trayectoria completada en **33.60 segundo**s
- Error de seguimiento (RMSE): **0.315 metros**
- Número de obstáculos encontrados: **7**
- Fallo de finalización: **False**

La trayectoria se completa en el tiempo establecido y sin atascos, por lo que se puede considerar de base que el control es suficientemente robusto para aguantar una cantidad moderada de ruido. Se observa que a lo largo del trayecto se comete más error de seguimiento promedio y se realizan más maniobras de evasión de obstáculos que en otros casos, haciendo que el tiempo total de conducción sea significativamente mayor en comparación (~2 segundos).

El vídeo demostrativo del experimento está subido en YouTube siguiendo este enlace.

## 5.3. Comparación de resultados

En este último apartado se harán dos comparaciones entre experimentos: una entre los errores cometidos por cada configuración individual, y otra entre el mejor caso (sin errores) y peor caso (todos los errores presentes) posibles.

### 5.3.1 Comparación Sin error - Error en odometría - Sesgo de dirección - Latencia

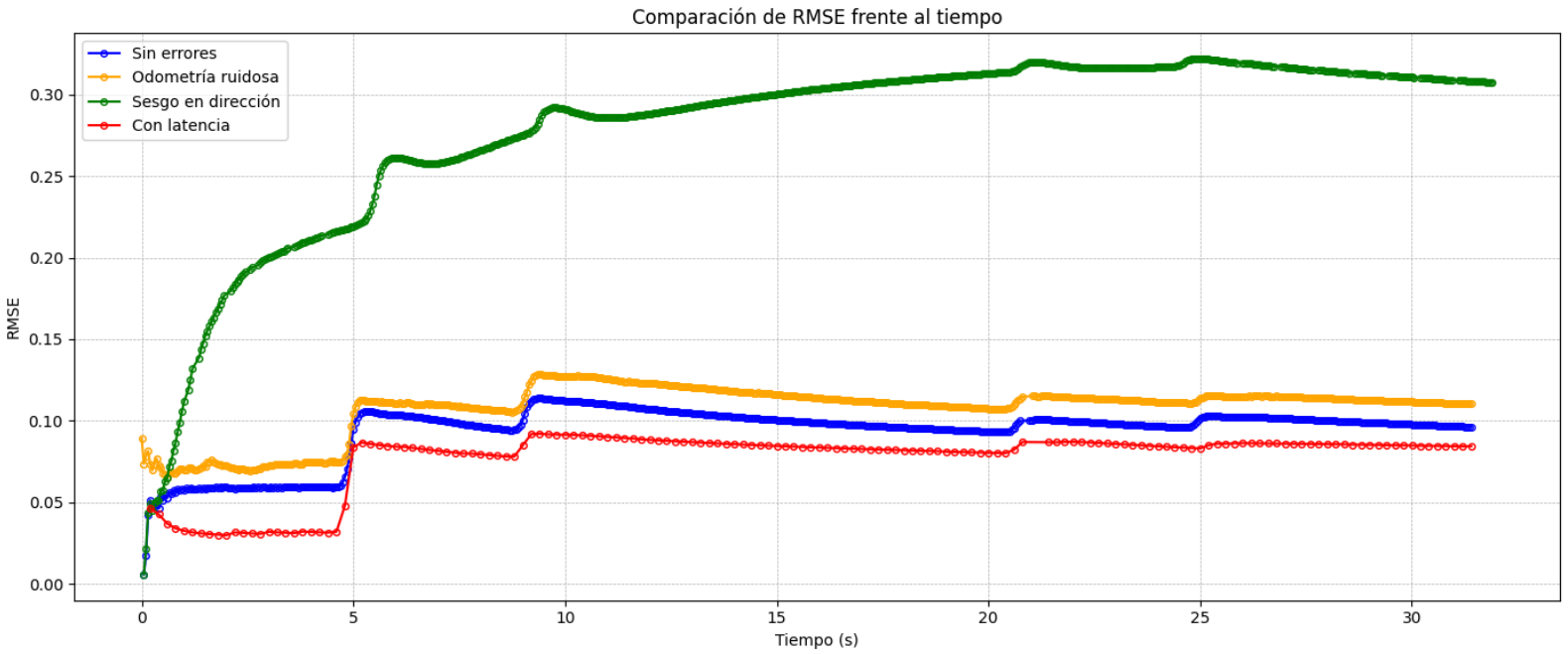


**Figura 20: RMSE frente al tiempo (4 configuraciones)**

En esta figura con todos los experimentos presentados se puede comprobar que el error más influencial por sí sólo es el sesgo de dirección (offset angular), que provoca un error de seguimiento parcial mayor que se acumula a lo largo del trayecto.

Por otra parte, se confirma que el control está bien preparado para soportar error en sensores para el cálculo de la odometría, pudiendo haber aumentado bastante los valores de desviación típica del ruido gaussiano inyectado sin que su rendimiento se aleje mucho del mejor caso.

Por último, se confirma que introducir latencia en el control alterando su frecuencia de actualización no conlleva virtualmente ningún problema en un escenario en el que los sensores y calibración son ideales.

A continuación se presenta una tabla con los parámetros de desempeño clave de cada configuración juntas para comparar su rendimiento:

| Configuración | Sin errores | Error en odometría | Sesgo de dirección | Latencia |
|---|---|---|---|---|
| Tiempo total [seg] | 31.40 | 31.40 | 31.90 | 31.40 |
| Nº obstáculos [uds.] | 1 | 1 | 7 | 1 |
| Fallo de finalización | NO | NO | NO | NO |
| RMSE medio [m] | 0.09377 | 0.10766 | 0.27997 | 0.07687 |
| RMSE mínimo [m] | 0.00572 | 0.06520 | 0.00572 | 0.02980 |
| RMSE máximo [m] | 0.11390 | 0.12869 | 0.32221 | 0.09204 |

De estos valores cabe destacar (que no se haya dicho hasta ahora) que el experimento con sesgo de dirección conserva una estadística de error mínimo de RMSE análoga al experimento sin errores debido a que este sesgo no afecta al vehículo al inicio del recorrido estando ya alineado con la primera recta de la trayectoria, mientras que los otros dos experimentos se pierde más precisión al inicio.

Por último, se ve claramente como en todos los casos se alcanza el valor máximo de error de seguimiento en la última curva (alrededor de los 25 segundos), siendo el vehículo con sesgo de dirección el que más error ha acumulado hasta entonces.

## 5.3.2 Comparación Mejor caso (sin error) - Peor caso (error en odometría, sesgo de dirección y latencia)

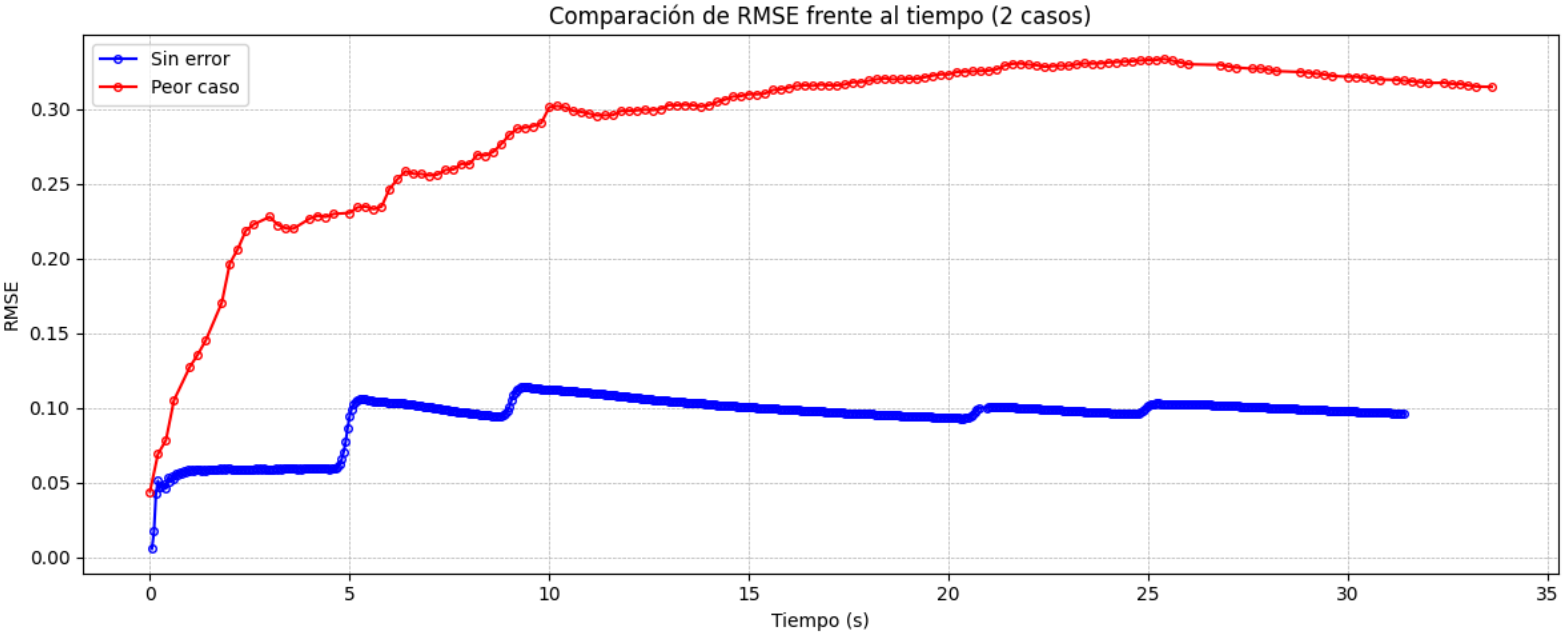


**Figura 21: RMSE frente al tiempo (2 configuraciones)**

| Configuración | Sin errores | Peor caso |
|---|---|---|
| Tiempo total [seg] | 31.40 | 33.60 |
| Nº obstáculos [uds.] | 1 | 7 |
| Fallo de finalización | NO | NO |
| RMSE medio [m] | 0.09377 | 0.29024 |
| RMSE mínimo [m] | 0.00572 | 0.04396 |
| RMSE máximo [m] | 0.11390 | 0.33346 |

En esta última comparación se puede comprobar cómo reacciona el sistema desarrollado en una situación ideal frente a en una situación relativamente realista. Además de tomar más tiempo para realizar la misma trayectoria, la latencia y la mala calibración emulada han complicado al controlador sus intentos de corregir la trayectoria dada una odometría por sensores con defectos. Se observa que la mayor subida de error se da al inicio de la trayectoria, ya que el robot empieza alineado antes de moverse y los errores inyectados provocan una alteración en el camino seguido frente al deseado.

Todos los paquetes, gráficos y resultados de esta memoria estarán subidos en el repositorio:
https://github.com/PatricioDMA/Proyecto_CPR_pdma.git