(1)

UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Desarrollo de un demostrador de vehículos conectados con delegación de tareas (*computation offloading*) y computación en el borde de la red (*edge computing*)

Autor:

Alfredo García Martín

Tutor:

Dr. D. Juan Carlos Aguado Manzano

VALLADOLID, JULIO 2025

TRABAJO FIN DE GRADO

TÍTULO:	Desarrollo de un demostrador de vehículos conectados con delegación de tareas (*computation offloading*) y computación en el borde de la red (*edge computing*)
AUTOR:	**Alfredo Garcia Martin**
TUTOR:	**Dr. D. Juan Carlos Aguado Manzano**
DEPARTAMENTO:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

Presidente:	Dr. Ramón José Durán Barroso
Vocal:	Dr. Ramón de la Rosa Steinz
Secretario:	Dr. J. Carlos Aguado Manzano
Suplente:	Dr. Ignacio de Miguel Jiménez
Suplente:	Dr. Noemí Merayo Álvarez

FECHA:

CALIFICACIÓN:

Resumen de Trabajo Fin de Grado

En este Trabajo Final de Grado se ha llevado a cabo la actualización y migración del sistema del vehículo DeepRacer de ROS1 a ROS2, asegurando la compatibilidad de nuestros propios nodos y desarrollando nuevos módulos adicionales. Para ello, se ha recurrido a fuentes como la documentación oficial de Amazon, repositorios en GitHub y distintos foros especializados. Además, se han establecido conexiones mediante el protocolo MQTT para facilitar el intercambio de mensajes con el vehículo, mejorando la comunicación y el procesamiento de datos en tiempo real. Este trabajo se enmarca dentro del ámbito de los vehículos conectados con delegación de tareas (*computation offloading*) y computación en el borde de la red (*edge computing)*, evaluando el impacto de estas mejoras en la eficiencia del sistema y en la capacidad del vehículo para ejecutar algoritmos de conducción autónoma de forma distribuida.

Abstract

In this Grade's Thesis, the system of the DeepRacer vehicle has been updated and migrated from ROS1 to ROS2, ensuring compatibility with our own nodes and developing additional new modules. To achieve this, official Amazon documentation, GitHub repositories, and specialized forums have been used as references. Additionally, MQTT connections have been established to enable message exchange with the vehicle, enhancing real-time communication and data processing. This work falls within the field of connected vehicles with task delegation (computation offloading) and edge computing, evaluating the impact of these improvements on system efficiency and the vehicle's ability to execute autonomous driving algorithms in a distributed manner.

Keywords

ROS2, DeepRacer, Vehículos Conectados, MQTT

Agradecimientos

Quiero expresar mi más sincero agradecimiento a mi tutor por brindarme la oportunidad de desarrollar este proyecto y por su constante apoyo a lo largo del proceso. Su orientación, paciencia y conocimientos han sido fundamentales para llevarlo a cabo, y su compromiso ha sido una fuente de inspiración para mí.

A mis compañeros, gracias por compartir ideas, dudas y también risas durante este camino. Su compañía ha hecho que los desafíos fueran más llevaderos y que cada avance, por pequeño que fuera, se sintiera aún más gratificante.

A mi familia, por estar siempre ahí, creyendo en mí incluso cuando yo dudaba. Su apoyo incondicional y su confianza han sido el pilar que me ha sostenido en los momentos más exigentes de este trabajo.

Finalmente, quiero agradecer a la Universidad de Valladolid por asumir los costes económicos del proyecto, permitiendo que esta investigación se llevara a cabo en las mejores condiciones posibles.

Este trabajo ha sido financiado por el Ministerio de Ciencia, Innovación y Universidades, la Agencia Estatal de Investigación y por FEDER/UE (proyecto PID2023-148104OB-C41 financiado por MICIU/AEI/10.13039/501100011033 y por FEDER/UE).

This work has been funded by Ministerio de Ciencia, Innovación y Universidades, Agencia Estatal de Investigación and by ERFD/EU (grant PID2023-148104OB-C41 funded by MICIU/AEI/10.13039/501100011033 and by ERFD/EU).

Índice

Índice de figuras

Índice de tablas

1.Introducción

1.1. Motivación del proyecto

El avance en la robótica y la inteligencia artificial ha impulsado la evolución de los vehículos autónomos, dando lugar a plataformas de desarrollo como AWS DeepRacer, una herramienta desarrollada por Amazon para la experimentación con aprendizaje por refuerzo y conducción autónoma orientada a ámbitos docentes. AWS DeepRacer integra algunas piezas de tecnología que son consideradas esenciales en el mundo de la robótica como es el caso de ROS (Robot Operating System) (2). ROS es realmente un ecosistema de desarrollo orientado a robots y que, al igual que los sistemas operativos, evoluciona con cada nueva versión publicada. Esta evolución tecnológica conlleva la necesidad de actualización, especialmente en lo que respecta a los sistemas operativos y arquitecturas de *software* subyacentes. En este contexto, la comunidad ha migrado de ROS1 a ROS2, lo que ha generado la necesidad de adaptar el ecosistema de DeepRacer para aprovechar las mejoras en rendimiento, modularidad y comunicación que ofrece ROS2.

En este Trabajo Fin de Grado (TFG), se toma como punto de partida un proyecto previo en el que se desarrollaron y añadieron nuevos nodos al sistema del vehículo DeepRacer, utilizando para ello el entorno de desarrollo ROS1 (3). A partir de esa base, el presente trabajo se centra en la actualización integral de la arquitectura del vehículo, migrando de ROS1 a una implementación completamente funcional en ROS2. Esta transición no solo facilita la compatibilidad con versiones y librerías más recientes, sino que también asegura que el DeepRacer pueda seguir siendo empleado en contextos de investigación y desarrollo que demandan soluciones modernas, especialmente en áreas emergentes como la robótica avanzada y la computación en el borde (Edge Computing).

A diferencia de proyectos anteriores centrados en la conducción autónoma con DeepRacer utilizando su entorno estándar (4), en este caso el reto ha sido reimplementar la estructura de *software* desde cero, asegurando la operatividad de todos los módulos esenciales, incluyendo el control del vehículo, la captura y procesamiento de datos de sensores, y la comunicación con servidores externos. Esta migración ha requerido una profunda revisión de la arquitectura del sistema, el análisis de compatibilidad de los nodos personalizados que habían sido desarrollados previamente, y la implementación de nuevos módulos capaces de integrarse de manera nativa con ROS2.

Este trabajo surge como una iniciativa del tutor Juan Carlos, quien identificó la necesidad de actualizar el DeepRacer del laboratorio para mantener la compatibilidad con las soluciones más recientes de Amazon y acceder a nuevas funciones y proyectos futuros. De esta manera, el proyecto no solo contribuye al avance del conocimiento en vehículos autónomos y conectados, sino que también sienta las bases para futuras investigaciones dentro del laboratorio.

En este TFG se han utilizado diversas fuentes de información, incluyendo documentación oficial de Amazon, repositorios de código en GitHub, foros especializados y la experiencia compartida por la comunidad de DeepRacer. A través de este proceso, se ha logrado no solo actualizar el vehículo, sino también mejorar su funcionalidad, optimizando la gestión de recursos y asegurando que pueda seguir siendo una plataforma relevante para experimentación e investigación en el ámbito de la robótica móvil.

El resultado final de este trabajo permite disponer de un DeepRacer totalmente funcional en ROS2, con todos los componentes esenciales operativos y con mejoras en la comunicación y el procesamiento de datos. Además, este proyecto abre la puerta a futuras integraciones con tecnologías emergentes en el ámbito de la conducción autónoma, la computación distribuida y los sistemas inteligentes de transporte.

1.2. Objetivos

Este Trabajo de Fin de Grado tiene dos objetivos principales. En primer lugar, la migración completa del sistema del vehículo Amazon DeepRacer de ROS1 a ROS2, asegurando la compatibilidad con todos los nodos personalizados previamente desarrollados. En segundo lugar, la implementación de nuevos módulos y funcionalidades que la conducción autónoma del vehículo, así como su capacidad frente a situaciones cambiantes de luz.

Los objetivos específicos que cubren este proyecto son los siguientes:

- Migración de la arquitectura *software* del vehículo DeepRacer desde ROS1 a ROS2, asegurando la operatividad de todos los nodos previamente desarrollados.
- Adaptación y actualización de los nodos propios del sistema, modificándolos para que sean compatibles con ROS2 y aprovechando las mejoras que ofrece esta nueva versión.
- Desarrollo e integración de nuevos nodos, optimizando el sistema de control y permitiendo una mejor comunicación entre los componentes del vehículo.
- Pruebas y validación del sistema mediante la ejecución de recorridos autónomos y la verificación de la correcta transmisión de datos entre los diferentes módulos del sistema. Y desarrollo del algoritmo de conducción.

1.3. Etapas y métodos

Para llevar a cabo este proyecto, se han seguido una serie de fases:

- Análisis de la arquitectura del DeepRacer y su *software*: Se ha realizado un estudio detallado tanto del sistema original del vehículo basado en ROS1 como de las integraciones realizadas en el proyecto anterior. Además, se han analizado los cambios necesarios para su migración a ROS2, así como las implicaciones técnicas que conlleva dicha transición.
- Migración progresiva de nodos: Se han trasladado y adaptado uno por uno los nodos del sistema a ROS2, verificando su correcto funcionamiento.
- Desarrollo de nuevos módulos: Se han creado nodos adicionales que mejoran la capacidad del DeepRacer para comunicarse y procesar datos en tiempo real.
- Despligue de cliente MQTT en el DeepRacer para su control: En la versión anterior del vehículo, el estado del mismo se controlaba a través del protocolo MQTT. Al instalar desde cero todo el sistema del DeepRacer ha sido necesario estudiar cómo funcionaba este control y el despliegue del cliente en el vehículo.
- Pruebas de rendimiento: Se han realizado mediciones de latencia y estabilidad para evaluar el impacto de la actualización.
- Validación del sistema en entornos controlados: Se han realizado pruebas de conducción autónoma y de comunicación en escenarios simulados y reales.

- Evaluación de compatibilidad con futuras actualizaciones de Amazon: Se ha asegurado que el sistema actualizado pueda seguir recibiendo mejoras y nuevas funciones de Amazon DeepRacer.

1.4. Recursos

Para el desarrollo de este proyecto, se ha utilizado el siguiente material electrónico y *software*:

Vehículo Amazon DeepRacer con pack de expansión Evo, equipado con:

- **Cámara estéreo y sensores de profundidad:** Una cámara monocular de 4 megapíxeles con lente gran angular que captura imágenes en tiempo real, esenciales para la percepción visual y la toma de decisiones.
- **Unidad de procesamiento**: El vehículo incorpora un módulo de cómputo con un procesador Intel Atom x5 a 1.6 GHz, 4GB de RAM y almacenamiento flash interno. Este sistema ejecuta una versión de Ubuntu y soporta la implementación de modelos de aprendizaje profundo.
- **El sistema de alimentación del vehículo está compuesto por dos baterías**: una de 1.100 mAh destinada al servomotor y al sistema de tracción, y una power bank externa que reemplaza la batería original de 13.600 mAh para alimentar el ordenador. Esta modificación permite una mayor autonomía y fiabilidad durante las pruebas, al separar claramente la fuente de energía del procesamiento respecto al sistema de movimiento
- **Conectividad**: Dispone de conexión Wi-Fi 802.11ac y tres puertos USB, lo que facilita la integración con otros dispositivos y la ampliación de sus capacidades con periféricos adicionales.
- **Unidad de Medición Inercial (IMU)**: Equipada con un giroscopio y acelerómetro Bosch BMI160, permite medir la orientación y la aceleración del vehículo.
- **LiDAR opcional**: En la versión Evo, el DeepRacer cuenta con un sensor LiDAR 2D con un radio de escaneo de 360 grados y un alcance de 12 metros, lo que mejora la capacidad de detección de obstáculos.
- **Servidor de desarrollo**: En este caso, se trata de **un ordenador personal** que ha sido fundamental para realizar pruebas y simulaciones.
- ***Software***:
    - ROS2 (Foxy Fitzroy), con herramientas de depuración y análisis de nodos.
    - Plataforma de simulación AWS DeepRacer, utilizada para pruebas previas antes de la implementación en hardware.
    - Sistema de comunicación MQTT, con un bróker Mosquitto instalado directamente en mi portátil para gestionar la comunicación entre el vehículo y otros sistemas.

Además, se han utilizado recursos adicionales del laboratorio, tales como:

- Pantalla externa, ratón, teclado y cables de conexión para facilitar el desarrollo.
- **Batería adicional para el DeepRacer**, diferente a la suministrada originalmente, proporcionando mayor autonomía en las pruebas.
- **Unidad USB de 64 GB**, utilizada para almacenar configuraciones y datos del proyecto.

Por otro lado, también se ha contado con una **maqueta de pruebas previamente fabricada en otros proyectos**, compuesta por:

- Goma negra de 5mm para simular la carretera.
- Fieltro negro de 90 cm de anchura para simular la carretera.
- Placas de PVC.
- Césped artificial.
- Pintura mate de diferentes colores.

1.5. Estructura del Proyecto

La estructura del presente trabajo se ha organizado en capítulos que reflejan el desarrollo técnico y conceptual del proyecto desde sus bases hasta su implementación completa. En el primer capítulo se presentan la introducción, la motivación, los objetivos y los métodos seguidos, junto con los recursos utilizados. El segundo capítulo, dedicado a la historia del arte, explica las bases tecnológicas sobre las que se apoya el proyecto, centrándose en ROS (Robot Operating System), comparando sus versiones ROS1 y ROS2, y detallando aspectos de comunicación como el uso de MQTT y Mosquitto.

A continuación, el capítulo 3 describe la infraestructura del vehículo, incluyendo tanto el hardware del Amazon DeepRacer como su *software* base. El capítulo 4 desarrolla el proceso de migración e integración a ROS2, explicando los cambios en nodos, servicios y tópicos. El capítulo 5 se centra en la implementación de algoritmos de conducción autónoma, especialmente la detección de carril con visión artificial. En el capítulo 6 se integran todos estos algoritmos en un sistema de percepción completo. Finalmente, el capítulo 7 presenta las conclusiones generales y propone líneas futuras de mejora, seguido de las referencias y anexos que completan el trabajo.

2.Estado del Arte

Esta sección presenta una revisión detallada de los fundamentos tecnológicos para el desarrollo del proyecto, enfocándose en la evolución del sistema operativo robótico ROS desde su primera versión hasta ROS 2, así como en la integración de protocolos de comunicación como MQTT en entornos robóticos. En particular, se analiza su aplicación en plataformas como el vehículo autónomo Amazon DeepRacer. La sección se divide en dos bloques: primero, se estudia la evolución de ROS, destacando las mejoras de ROS 2; luego, se exploran las estrategias de migración desde ROS 1 y el uso de tecnologías de comunicación distribuidas como MQTT.

2.1. Introducción a ROS y su papel en la robótica

El Robot Operating System (ROS) es una infraestructura de código abierto que facilita el desarrollo de *software* modular en aplicaciones robóticas. Fue introducido en 2007 gracias a mezcla de investigación dentro de la universidad y desarrollo industrial.

ROS permite la creación de nodos que se comunican entre sí mediante un sistema de mensajes estandarizados, promoviendo la reutilización de código y la abstracción del hardware. Sin embargo, ROS 1 presenta limitaciones importantes, como la dependencia de un nodo maestro (roscore), ausencia de soporte para tiempo real y una limitada capacidad de escalabilidad y seguridad.

El desarrollo de ROS fue impulsado por la universidad de Stanford y el centro de investigación Willow Garage, posteriormente mantenido por Open Robotics, consolidándose como un estándar en la robótica en entornos académicos, de investigación y de desarrollo.

2.2. Arquitectura de ROS1 y sus principales características

ROS 1 emplea una arquitectura hibrida que combina dos modelos: El maestro-esclavo y el paradigma publicador/suscriptor. Es un sistema formado por un nodo[1] maestro, conocido como roscore, que cumple una función central, al permitir a los demás nodos registrarse, suscribirse y de suscribirse y ofrecer sus servicios a otros nodos. Gracias a este nodo central, se establece una base para la comunicación en el sistema.

La comunicación entre estos nodos se basa en las publicaciones y suscripciones entre estos mismos, los publicadores generan y envían mensajes mientras que los suscriptores reciben la información y actúan en consecuencia a ella, haciendo que el sistema sea escalable y eficiente. Ofrece tres mecanismos de comunicación (5):

- **Tópicos**: Publicación/suscripción asíncrona.
- **Servicios**: Comunicación síncrona de tipo solicitud-respuesta.
- **Acciones**: Comunicación asíncrona con retroalimentación para tareas prolongadas.

[1] Un nodo es una unidad de procesamiento que ejecuta una tarea especifica dentro del sistema, como leer sensores, controlar un motor o procesar datos.

Pese a su flexibilidad, presenta desventajas críticas:

- Un único punto de fallo (roscore), es decir, si falla el nodo principal deja de funcionar.
- Falta de soporte para sistemas en tiempo real, al ir evolucionando, ya no hay apoyo a versiones anteriores.
- Comunicación mediante TCPROS y UDPROS, con control limitado sobre la calidad del servicio (QoS)

2.3. Introducción y arquitectura de ROS2

ROS 2 surge como respuesta a las deficiencias de ROS 1, adoptando el middleware DDS (Data Distribution Service) (6) que es un estándar de comunicación descentralizada y robusta. DDS facilita configuraciones avanzadas como tolerancia a fallos, latencia garantizada, y descubrimiento automático de nodos, cumpliendo así con los requisitos de sistemas distribuidos en tiempo real, incluyendo además nativamente políticas de *Quality of Service (QoS)* (7).

ROS 2 introduce mejoras fundamentales:

- Eliminación del nodo maestro.
- Soporte multiplataforma (Linux, Windows, macOS).
- Compatibilidad con sistemas embebidos y redes heterogéneas.
- Configuración granular de QoS (latencia, fiabilidad, persistencia), Las políticas de calidad de servicio (QoS) son un conjunto de configuraciones que controlan cómo se comunican los nodos en ROS 2, permitiendo adaptar su comportamiento en función de los requisitos del sistema distribuido.
- Mayor seguridad mediante autenticación y cifrado (SROS2)

2.4. Comparación entre ROS1 y ROS2

A continuación, se presentan las diferencias clave entre ROS1 y ROS2 en detalle:

Arquitectura de Comunicación

ROS1 utiliza un modelo maestro-esclavo coordinado a partir de un único punto central, el nodo roscore, que gestiona el resto de los nodos. Si roscore falla, el sistema pierde la capacidad de registrar nuevos nodos. ROS2, en cambio, elimina esta dependencia utilizando DDS, que permite una comunicación distribuida y robusta.

Soporte para Sistemas de Tiempo Real

ROS1 no está diseñado para sistemas de tiempo real, lo que dificulta su uso en aplicaciones industriales críticas. ROS2 introduce control sobre la QoS, permitiendo definir políticas de comunicación que garantizan tiempos de respuesta predecibles, Esto representa una mejora significativa para aplicaciones donde la latencia y la fiabilidad desempeñan papeles importantes.

Compatibilidad con Diferentes Sistemas Operativos

ROS1 está optimizado principalmente para Linux, lo que restringe su uso en plataformas con otros sistemas operativos. Aunque existen implementaciones no oficiales para Windows y macOS, su funcionalidad es limitada. ROS2 en cambio, expande su compatibilidad a Windows y macOS, facilitando su adopción en entornos heterogéneos

y permitiendo así, el desarrollo y despliegue de aplicaciones robóticas en entornos más diversos.

Escalabilidad y Seguridad

La arquitectura de ROS1 dificulta la implementación en sistemas de gran escala. ROS2, gracias a DDS, permite implementar sistemas con múltiples robots que operan de manera distribuida y con mecanismos de autenticación y encriptación.

En la Tabla 1 se presenta una comparación entre ROS2 y ROS2 en cuanto a aspectos claves como escalabilidad, seguridad y arquitectura.

Característica	ROS1	ROS2
Arquitectura	Maestro-esclavo	Descentralizada (DDS)
Tiempo Real	No	Sí
Compatibilidad	Principalmente Linux	Linux, Windows, macOS
Calidad del Servicio	Limitada	Avanzada
Escalabilidad	Baja	Alta
Seguridad	Limitada	Cifrado y autenticación

Tabla 1. Comparación ROS1 vs ROS2

En la versión original del DeepRacer se empleaba ROS 1 Kinetic como sistema base. Sin embargo, con la actualización realizada en este proyecto, se ha adoptado ROS 2 Foxy Fitzroy, una distribución más moderna y con soporte a largo plazo (LTS).

En la Tabla 2 se realiza una comparativa entre ambas versiones que recoge aspectos clave como el año de lanzamiento, la finalización del soporte oficial y los sistemas operativos y lenguajes compatibles.

Característica	ROS 1 Kinetic	ROS 2 Foxy Fitzroy
Año de lanzamiento	2016	2020
Fin de soporte (EOL)	Abril 2021	Mayo 2023
Sistema operativo principal	Ubuntu 16.04 (Xenial)	Ubuntu 20.04 (Focal)
Lenguajes soportados	C++, Python 2.7	C++, Python 3
Mantenimiento oficial	Finalizado	LTS (soporte extendido)

Tabla 2 comparación de las especificaciones de ROS vs ROS2 usados es el proyecto

2.5. Opciones para la migración de ROS1 a ROS2

La migración de ROS1 a ROS2 es un desafío que requiere una planificación cuidadosa debido a las diferencias entre las arquitecturas de ambos sistemas. Existen varias estrategias para llevar a cabo esta transición:

- **Uso de ros1_bridge** (8)**:** Permite la coexistencia de nodos ROS1 y ROS2 mediante un nodo que actuaría como puente de comunicación. Es útil cuando se necesita compatibilidad temporal, pero introduce una capa adicional de complejidad, ya que requiere compilar y gestionar múltiples workspaces y definir posibles reglas personalizadas cuando los mensajes no coinciden exactamente

- **Conversión progresiva de paquetes:** Consiste en portar gradualmente los paquetes ROS1 a ROS2, adaptándolos uno por uno. Aunque permite una transición controlada, puede generar inconsistencias si algunos componentes permanecen en ROS1 por mucho tiempo.
- **Reescritura completa de los nodos en ROS2:** En esta estrategia, se eliminan los nodos ROS1 y se re-implementan desde cero en ROS2, aprovechando todas las mejoras de la nueva versión.

En este proyecto, se ha optado por la reescritura completa de los nodos en ROS2, lo que ha permitido optimizar el código y mejorar su integración con las nuevas funcionalidades. Esta decisión se basó en las siguientes ventajas:

- **Aprovechamiento total de ROS2:** Al escribir los nodos desde cero, se pueden utilizar directamente las mejoras en calidad de servicio (QoS), soporte para sistemas de tiempo real y la arquitectura descentralizada basada en DDS, ya explicada anteriormente.
- **Mayor estabilidad y mantenimiento a largo plazo:** ROS1 ya no recibe actualizaciones principales, por lo que una migración completa garantiza la compatibilidad futura con nuevos desarrollos.
- **Eliminación de dependencias innecesarias:** Al reescribir los nodos, se ha podido optimizar la estructura del *software*, eliminando código obsoleto y mejorando la eficiencia del sistema.
- **Mayor flexibilidad para el desarrollo de nuevos nodos:** La migración no solo ha implicado replicar la funcionalidad existente, sino también el desarrollo de nuevos nodos para mejorar la arquitectura del sistema.

Mas adelante se detallará el proceso llevado a cabo, incluyendo la adaptación del código, las herramientas utilizadas y las pruebas realizadas para validar la nueva implementación.

2.6. Implementación de una comunicación eficiente con MQTT

MQTT (Message Queuing Telemetry Transport) es un protocolo ligero basado en el modelo publicador-suscriptor, ampliamente utilizado en sistemas IoT debido a su eficiencia y bajo consumo de ancho de banda. En este proyecto, se ha elegido MQTT como mecanismo de comunicación entre los componentes del sistema debido a sus ventajas sobre otras opciones, como WebSockets o ROSbridge (en este caso se han utilizado *sockets*).

Esta decisión responde, en parte, a la continuidad con el proyecto anterior (3), donde ya se utilizaba MQTT como protocolo principal de intercambio de datos entre el vehículo y el sistema de administración. Dado que sigue cumpliendo los requisitos de eficiencia, simplicidad e interoperabilidad, se ha mantenido su uso en la implementación actual. A continuación, se describe la arquitectura empleada, que, en este caso, la implementación se ha realizado manteniéndola como el proyecto anterior:

Bróker MQTT en servidor externo: Se ha utilizado un ordenador externo como servidor de mensajes (bróker) encargado de gestionar la comunicación entre los diferentes clientes.

Cliente MQTT en el mismo ordenador: Se ha instalado un cliente MQTT en el mismo ordenador en el que se ha desplegado el bróker para interactuar directamente con el coche,

permitiendo la monitorización y depuración de los mensajes enviados. Para ello, se ha utilizado MQTT Explorer, una herramienta visual que facilita la gestión y visualización de los tópicos de comunicación.

Cliente MQTT en el coche: En el vehículo, se ha implementado un cliente MQTT utilizando una librería de Python, lo que permite recibir comandos y enviar información al bróker. Este protocolo se utiliza para transmitir datos no críticos, como el estado de la red o el nivel de batería.

Mensajería UDP para comunicación con el coche: Se utilizan mensajes UDP para transmitir comandos y datos de sensores entre el sistema de control y el vehículo. Asimismo, este protocolo se emplea para la transmisión de imágenes comprimidas JPG con paquetes fragmentados si fuese necesario, esta forma de enviar las imágenes minimiza la latencia en la transmisión de video (9). También se envían las lecturas del LIDAR hacia el servidor en la nube. Se ha optado por este protocolo debido a sus ventajas en entornos robóticos:

- **Menor latencia**: UDP es un protocolo sin confirmación de entrega, lo que reduce el tiempo de transmisión de datos.
- **Menor consumo de recursos**: No requiere mantener una conexión establecida, lo que es beneficioso en sistemas con hardware limitado.
- **Mayor tolerancia a pérdidas de paquetes**: En aplicaciones robóticas, donde es más importante la actualización frecuente de datos que la recepción garantizada de cada mensaje, UDP es una mejor opción.

En el proyecto anterior, se utilizaba el protocolo MQTT, combinado con un bróker Mosquitto, una herramienta ligera y de código abierto que actúa como intermediario en la comunicación MQTT, para gestionar la transmisión de mensajes entre el vehículo y el servidor (10). Dada su eficiencia y compatibilidad con sistemas de bajo consumo, se mantuvo esta arquitectura como base en las etapas iniciales del presente trabajo. La descripción técnica completa se detalla más adelante, en el capítulo dedicado a la.

3. Infraestructura del Vehículo

El Amazon DeepRacer es un vehículo autónomo a escala desarrollado por Amazon Web Services (AWS) con el objetivo de proporcionar una plataforma accesible para la experimentación en inteligencia artificial, aprendizaje por refuerzo y robótica. Su diseño compacto, combinado con un potente sistema de procesamiento y una serie de sensores avanzados, lo convierte en una herramienta ideal para el desarrollo de algoritmos de navegación autónoma y percepción del entorno.

3.1. Especificaciones técnicas del Amazon DeepRacer

El DeepRacer está equipado con un conjunto de componentes de hardware que le permiten operar de manera autónoma y procesar datos en tiempo real, componentes que fueron mencionados en la introducción.

A pesar de su diseño avanzado, el DeepRacer presenta algunas limitaciones. Por ejemplo, no incorpora un sistema de geoposicionamiento por GPS ni un sensor de velocidad integrado. Esto impone ciertas restricciones en el desarrollo de aplicaciones que requieren una mayor precisión en la localización y control del movimiento.

En proyectos anteriores, se llevaron a cabo modificaciones para mejorar la autonomía del vehículo y ampliar su capacidad de procesamiento. Estas mejoras incluyeron la sustitución de la batería de litio por una de mayor capacidad y la integración de sensores adicionales, permitiendo así, optimizar la estabilidad del sistema y facilitar la incorporación de nuevos algoritmos de navegación autónoma.

3.2. *Software* original

El Amazon DeepRacer viene con un sistema operativo basado en Linux, preconfigurado para ejecutar modelos de aprendizaje por refuerzo y permitir la programación de comportamientos autónomos. Originalmente, el vehículo está equipado con **Ubuntu 16.04.3 LTS** y una versión de **ROS (Robot Operating System) Kinetic**, proporcionando un entorno flexible para el desarrollo de aplicaciones robóticas, que se mantuvo y sobre la que se trabajó en proyectos anteriores.

3.3. Características del *software*

ROS es una plataforma de *software* que facilita la programación de robots mediante una arquitectura modular basada en nodos. Estos nodos pueden comunicarse entre sí a través de un sistema de mensajería basado en tópicos (publicador-suscriptor) o mediante llamadas a servicios (RPC).

En el DeepRacer, ROS gestiona el acceso a los sensores, la ejecución de comandos de control y la implementación de algoritmos de aprendizaje automático. Para interactuar con ROS en el vehículo, es necesario importar variables de entorno utilizando el script `setup.bash`, ubicado en `/opt/aws/deepracer/`.

Ejemplo de comando para inicializar ROS:

```
source /opt/aws/deepracer/setup.bash
```

3.4. Nodos ROS preinstalados

El sistema de *software* del DeepRacer en su versión original está compuesto por múltiples nodos ROS que gestionan distintas funcionalidades del vehículo. Algunos de los más relevantes incluyen:

- **`/battery_node`**: Monitoriza el nivel de batería y publica datos sobre el estado de la energía.
- **`/control_node`**: Gestiona los modos de conducción (manual, calibración y autónomo), enviando comandos a los motores y servomecanismos.
- **`/inference_engine`**: Responsable de la ejecución del modelo de aprendizaje automático para la conducción autónoma.
- **`/navigation_node`**: Procesa datos de percepción y genera comandos de navegación en base a la información de los sensores.
- **`/rplidarNode`**: Publica datos del sensor LiDAR en el tópico `/scan`, permitiendo la detección de obstáculos.
- **`/sensor_fusion_node`:** Integra información de múltiples sensores para mejorar la precisión en la detección del entorno.
- **`/webserver`:** Proporciona una interfaz web para el control manual del vehículo y la carga de modelos de IA.

Cada uno de estos nodos desempeña un papel crucial en la operación del DeepRacer, facilitando la comunicación entre los distintos subsistemas del vehículo.

La Tabla 3 muestra un resumen de las publicaciones, suscriptciones y servicios ofrecidos por los nodos preinstalados en el Deepracer, útil para comprender mejor la organización de los datos y sus comunicaciones en el sistema. Ha sido elaborada a partir de la inspección del sistema original de Amazon y la documentación aportada por Ignacio (3) (3).

Nodo	Publicaciones	Suscripciones	Servicios

`/battery_node`	`/rosout`		`/battery_level`
			`/battery_node/get_loggers`
			`/battery_node/set_logger_level`

`/control_node`	`/rosout`	`/auto_drive`
		`/calibration_drive`
		`/manual_drive`
			`/autonomous_throttle`
			`/car_state`
			`/control_node/get_loggers`
			`/control_node/set_logger_level`
			`/enable_state`
			`/get_car_cal`
			`/get_car_led`
			`/model_state`
			`/set_car_cal`
			`/set_car_led`

`/inference_engine`	`/rosout`		`/inference_engine/get_loggers`
			`/inference_engine/set_logger_level`
			`/inference_sate`

			/load_model
/media_engine	/display_mjpeg /rosout /video_mjpeg		/media_engine/get_loggers /media_engine/set_logger_level /media_state
/model_optimizer	/rosout		/model_optimizer/get_loggers /model_optimizer/set_logger_level /model_optimizer_server
/navigation_node	/auto_drive /rosout	/rl_results	/load_action_space /navigation_node/get_loggers /navigation_node/set_logger_level /navigation_throttle
/rosout	/rosout_agg	/rosout	/rosout/get_loggers /rosout/set_logger_level
/rplidarNode	/rosout /scan		/start_motor /stop_motor /rplidarNode/get_loggers /rplidarNode/set_logger_level
/sensor_fusion_node	/overlay_msg /rosout /sensor_msg	/scan /video_mjpeg	/configure_lidar /sensor_data_status /sensor_fusion_node/get_loggers /sensor_fusion_node/set_logger_level
/servo_node	/rosout		/get_cal /get_led_state /servo_cal /servo_gpio /servo_node/get_loggers /servo_node/set_logger_level /servo_state /set_led_state /set_raw_pwm
/software_update	/rosout		/begin_update /console_upload_model /get_device_info /get_otg_link_state /software_update/get_loggers /software_update/set_logger_level /software_update_get_state /software_update_status /verify_model_ready
/web_video_server	/rosout	/display_mjpeg	/web_video_server/get_loggers /web_video_server/set_logger_level

/webserver	/manual_drive /rosout		/webserver/get_loggers /webserver/set_logger_level

Tabla 3 Nodos en Ubuntu 16.8[2]

3.5. Situación de la configuración ROS en el vehículo

En la versión original del DeepRacer basado en ROS1 Kinetic, el sistema venía preconfigurado con una serie de nodos que gestionaban diferentes aspectos del funcionamiento del vehículo. Sin embargo, para la implementación de un nuevo *software* de control autónomo, se decidió deshabilitar varios de estos nodos preinstalados con el fin de evitar interferencias y optimizar el uso de los recursos computacionales del vehículo. Estos cambios están descritos en el Trabajo Fin de Máster de Ignacio Royuela (3).

Por completitud, y para facilitar al lector la compresión de los cambios que se realizarán en este trabajo, se describen a continuación los cambios que se operaron en la versión original del Deepr Racer.

3.5.1. Nodos originales deshabilitados

Se desactivaron los siguientes nodos debido a que no se utilizaban las herramientas de aprendizaje automático de Amazon ni la interfaz web del vehículo:

- **`/interference_engine`** y **`/model_optimizer`**: Encargados de la optimización y gestión del modelo de aprendizaje automático del vehículo.
- **`/navigation_node`** y **`/sensor_fusion_node`**: Relacionados con la navegación asistida basada en aprendizaje automático y fusión de datos sensoriales.
- **`/webserver`** y **`/web_video_server`**: Proporcionaban acceso a una interfaz web para visualizar información y controlar el vehículo remotamente.
- **`/software_update`**: Administraba las actualizaciones del sistema, deshabilitado para evitar problemas de compatibilidad con el nuevo *software* desarrollado.

La desactivación de estos nodos se llevó a cabo editando el archivo de lanzamiento `/opt/aws/deepracer/share/launch/deepracer.launch`, asegurando que solo se ejecutaran los nodos necesarios para el nuevo sistema.

3.5.2. Nodos originales manteniendo su Funcionalidad

Los nodos que permanecieron activos estaban directamente relacionados con la conducción y percepción del vehículo:

- **Nodos de control del vehículo**: `/servo_node` y `/control_node` gestionaban la dirección, aceleración y frenado.
- **Nodo de recepción de imágenes**: el nodo `/media_engine`, encargado de procesar las imágenes de la cámara del vehículo para la toma de decisiones.
- **Nodo de informacion LIDAR**: `/rplidarNode` que se encarga de gestionar los datos del LIDAR.

[2] I. Royuela, Nodos ROS preinstalados en el vehículo junto a los tópicos en los que publican y a los que se suscriben, y los servicios que ofrecen., Trabajo Fin de Máster, Universidad De Valladolid, 2022.

- **Nodo de sensores:** Permitía la recopilación de datos provenientes del nodo `/sensor_fusion_node`, incluyendo información del sensor de inercia (IMU), el LiDAR (a través del tópico `/scan` publicado por `/rplidarNode`) y el nivel de batería (desde `/battery_level`, publicado por `/battery_node`).

3.5.3. Nodos incorporados no originales

El nuevo *software* desarrollado para el vehículo por Ignacio Royuela (3) introdujo tres nodos adicionales diseñados para mejorar la comunicación y el control autónomo:

Nodo de comunicaciones con el administrador: Permitía la conexión con un servidor externo utilizando el protocolo MQTT, facilitando el envío y recepción de comandos e información del vehículo.

Nodo de control autónomo del vehículo: Procesaba imágenes y datos sensoriales en tiempo real para ejecutar el algoritmo de guiado autónomo.

Nodo de control externo del vehículo: Gestionaba el control remoto en tiempo real a través de una conexión de red mediante el protocolo UDP.

Estos nuevos nodos fueron programados en Python 3.7 y se integraron con el sistema ROS preexistente, asegurando un funcionamiento eficiente y flexible del DeepRacer modificado.

3.5.4. Procesamiento de Imágenes

En la configuración original del Deepracer el procesamiento de imágenes se realiza utilizando la librería `CvBridge` (11), que permite convertir los mensajes de tipo imagen de ROS en objetos de OpenCV, lo que facilita su posterior análisis y tratamiento. Esta conversión es fundamental para tareas como la detección de carriles, reconocimiento de objetos o el cálculo de errores de trayectoria.

El nodo /media_engine es el encargado de publicar el video de la cámara delantera y los datos se convierten internamente usando CVBridge en el nodo de aprendizaje por IA, desactivado, o en el control del vehículo. Por lo que se mantiene la herramienta, pero se adapta a la estructura de ROS2.

3.6. Desarrollo de la Maqueta

El banco de pruebas, ya constituido, se basa en el **Trabajo de Fin de Grado (TFM) de Ignacio Royuela** y comprende la maqueta y el vehículo autónomo.

En dicho trabajo se diseñó una maqueta modular a escala 1:16 para simular un entorno urbano flexible y realista. Su estructura permite evaluar diversos escenarios viales sin reconstrucciones constantes. La primera versión, con goma negra sobre cartón, presentaba problemas ópticos y de estabilidad, por lo que se mejoró con fieltro negro y base plástica. Los carriles de 30 cm de ancho se ajustaron para optimizar el guiado del vehículo, reduciendo el grosor de las líneas laterales para mayor realismo.

Ilustración 1 Maqueta

4. Migración e Integración del Sistema DeepRacer en ROS2

4.1. Actualización

En el marco de desarrollo del presente trabajo, fue imprescindible llevar a cabo una actualización del *software* del AWS DeepRacer. Esta tarea no solo tenía como objetivo garantizar la compatibilidad con las versiones más recientes de las herramientas y librerías utilizadas en el entorno de desarrollo de Amazon, sino también optimizar el rendimiento del vehículo en sus procesos de aprendizaje y ejecución de modelos de conducción autónoma.

Para comprender adecuadamente el proceso de actualización, se llevó a cabo una investigación detallada sobre la arquitectura del DeepRacer y su integración con ROS (Robot Operating System). Se estudiaron en profundidad los nodos originales que conformaban la estructura de comunicación del vehículo, los sensores y mecanismos involucrados, así como los cambios que suponía la transición de ROS1 Kinetic a ROS2 Foxy Fitzroy.

4.1.1. Fases del proceso de actualización

La actualización del AWS DeepRacer se realizó en varias fases, cada una de ellas con especial atención a la integridad del sistema y a la prevención de posibles fallos que pudieran comprometer su funcionamiento.

4.1.2. Respaldo de la versión previa

Antes de proceder con la actualización, se realizó una copia de seguridad completa del sistema operativo original (Ubuntu 16.04 Xenial Xerus) y de los archivos de configuración del DeepRacer. Este respaldo fue esencial, ya que cualquier error en el proceso de actualización podía dejar el vehículo inoperativo. Para esta tarea se utilizó una unidad USB de 64GB proporcionada por la escuela, en la cual se almacenaron tanto la imagen del sistema como los archivos de configuración personalizados.

Ilustración 2 pincho para la copia de seguridad y para el reseteo

4.1.3. Verificación de la versión actual del *software*

Antes de iniciar el procedimiento, se verificó la versión del *software* instalada en el DeepRacer a través de la consola de administración del dispositivo. Para ello, se accedió a la sección de configuración, donde se encontraba el apartado "About", que indicaba la versión del sistema en ejecución. Esto permitió confirmar que el dispositivo aún utilizaba

la pila de *software* basada en Ubuntu 16.04 y ROS1 Kinetic, lo que hacía necesaria la actualización.

4.1.4. Preparación del medio de instalación

Para llevar a cabo la actualización, fue necesario preparar un medio de instalación USB con la versión personalizada de Ubuntu 20.04 Focal Fossa y las herramientas específicas del DeepRacer, incluyendo Intel® OpenVINO™ toolkit 2021.1.110, ROS2 Foxy Fitzroy y Python 3.8

Todos estos paquetes *software* se pueden descargar directamente desde Amazon en una carpeta comprimida, en la parte de actualización del Deepracer, que contiene todos los archivos necesarios. Una vez descargada, basta con descomprimirla y copiar su contenido en el dispositivo UBS correspondiente (12).

El proceso de preparación de este medio de instalación consistió en:

- Descargar la imagen ISO personalizada proporcionada por Amazon AWS.
- Formatear la unidad USB con dos particiones:
  - Una partición FAT32 de 4GB, utilizada como partición de arranque.
  - Una partición NTFS de al menos 18GB, donde se almacenarían los archivos de actualización.
- Hacer la unidad USB *bootable* mediante la herramienta UNetbootin.
- Copiar los archivos de actualización en la partición NTFS.

4.1.5. Instalación del nuevo sistema operativo

Una vez preparado el medio de instalación, se procedió a conectar el DeepRacer a un monitor mediante un cable HDMI y se utilizaron un teclado y un ratón USB para interactuar con la interfaz del BIOS del dispositivo.

El procedimiento de actualización consistió en los siguientes pasos, insertar la unidad USB en un puerto disponible del DeepRacer, acceder al BIOS del dispositivo presionando repetidamente la tecla "ESC" durante el arranque, seleccionar la opción "Boot From File" y navegar hasta la partición de arranque del USB, iniciar la instalación desde el archivo BOOTx64.EFI.

Esperar a que el sistema inicie el proceso de actualización automáticamente. Durante este proceso, se ejecutaron diversos scripts que instalaron el nuevo sistema operativo, los controladores y las herramientas de desarrollo necesarias.

Al finalizar la instalación, el dispositivo se reinició automáticamente, indicando que el proceso había concluido con éxito.

Después de completar la actualización del AWS DeepRacer, el sistema operativo Ubuntu 20.04 Focal Fossa con ROS 2 Foxy Fitzroy y las herramientas necesarias como Intel OpenVINO™ Toolkit y Python 3.8 estarán en funcionamiento. Para poder acceder a los nodos y gestionar las funciones del vehículo, es necesario abrir una terminal y ejecutar ciertos comandos para configurar el entorno adecuado. Los pasos básicos para iniciar la interacción con los nodos incluyen el uso de los siguientes comandos:

```
source /opt/ros/foxy/setup.bash
```

```
source /opt/aws/deepracer/lib/ setup.bash
```

Estos comandos cargan el entorno de ROS 2 y aseguran que las bibliotecas necesarias del AWS DeepRacer estén disponibles para su ejecución.

Además, el sistema permite realizar actualizaciones en los nodos utilizando el repositorio oficial de GitHub de Amazon (13). Estos repositorios son mantenidos por la comunidad y por el equipo de Amazon, y ofrecen una manera de personalizar y extender las funcionalidades del vehículo. Para actualizar o modificar nodos, se debe crear un paquete nuevo, copiar las dependencias necesarias y compilarlo correctamente en el dispositivo, esto se explicará en detalle más adelante. Sin embargo, debido a que ha pasado un tiempo considerable desde que Amazon realizó una actualización oficial, existen ciertas dificultades a la hora de implementar nuevas modificaciones. En particular, algunos nodos presentan incompatibilidades con la versión más reciente de los paquetes y no funcionan correctamente, lo que puede generar más fallos que mejoras.

En el intento de realizar actualizaciones y modificaciones, se experimentó con varios nodos que no operaban como se esperaba, lo que dificultó su integración y uso. No obstante, algunos de los nodos más importantes, como los relacionados con la percepción, la navegación y el control, continúan operando de manera estable tras la actualización, aunque la implementación de cambios más complejos sigue siendo un desafío debido a la falta de soporte o actualizaciones regulares por parte de Amazon.

Para ilustrar los cambios y nuevos nodos disponibles después de la actualización, se incluye la Tabla 4 donde se detallan las funciones de los nuevos nodos. Estos nodos reflejan las modificaciones y adiciones realizadas, lo que permite una mayor personalización y control sobre el comportamiento del AWS DeepRacer. Estos cambios no solo optimizan el rendimiento del vehículo, sino que también ofrecen nuevas oportunidades para su ajuste y experimentación.

Nodos ROS2	Tópicos Publicados	Tópicos Suscrito
/Battery Node	Ninguno	Ninguno
/Ctrl_node	/ctrl_pkg/raw_pwm Tipo mensaje: ServoCtrlMsg /ctrl_pkg/servo_msg Tipo mensaje: ServoCtrlMsg	/deepracer_navigation_pkg/auto_drive Tipo mensaje: ServoCtrlMsg /webserver_pkg/calibration_drive Tipo mensaje: ServoCtrlMsg /webserver_pkg/manual_drive Tipo mensaje: ServoCtrlMsg
/Inference	/inference_pkg/rl_results Tipo InferResultsArray	/sensor_fusion_pkg/sensor_msg Tipo mensaje: EvoSensorMsg
/camera_node	/camera_pkg/video_mjpeg Tipo CameraMsg /camera_pkg/display_mjpeg Image	Ninguno
model/optimizer	Ninguno	Ninguno
/Navegation_node	/deepracer_navigation_pkg/auto_drive Tipo ServoCtrlMsg	/inference_pkg/rl_results Tipo mensaje InferResultsArray
/rosout	No existe	No existe
/rplidar_node	/scan tipo sensor_msgs/LaserScan	Ninguno

/sensor_fusión_node	/sensor_fusion_pkg/overlay_msg Tipo imagen /sensor_fusion_pkg/sensor_msg Tipo EvoSensorMsg	/rplidar_ros/scan Tipo LaserScan /camera_pkg/video_mjpeg Tipo CameraMsg /camera pkg/display mjpeg Imagen
/servo_node	Ninguno	/ctrl_pkg/servo_msg Tipo ServoCtrlMsg /ctrl_pkg/raw_pwm Tipo ServoCtrlMsg
/software_update	/deepracer systems pkg/software update pct Tipo SoftwareUpdatePctMsg	/usb_monitor_pkg/usb_file_system_notification Tipo USBFileSystemNotificationMsg /deepracer systems pkg/network connection_status Tipo NetworkConnectionStatus
/web_video_server	Ninguno	Ninguno
/webserver_publisher_node	/webserver_pkg/calibration_drive Tipo ServoCtrlMsg /webserver_pkg/manual_drive Tipo ServoCtrlMsg	/deepracer systems pkg/software update pct Tipo SoftwareUpdatePctMsg
/model_loader_node	Ninguno	/usb_monitor_pkg/usb_file_system_notification Tipo USBFileSystemNotificationMsg
/network_monitor_node	/deepracer_systems_pkg/network_connection_status Tipo NetworkConnectionStatus	/usb_monitor_pkg/usb_file_system_notification Tipo USBFileSystemNotificationMsg
/ otg_control_node	Ninguno	Ninguno
/status_led_node	Ninguno	Ninguno
/usb_monitor_node	/usb_monitor_pkg/usb_file_system_notification Tipo USBFileSystemNotificationMsg	Ninguno
/async_web_server_cpp	Ninguno	Ninguno
/device_info_node	Ninguno	Ninguno

Tabla 4. Nodos Actuales de Amazon para el Deepracer

Asimismo, se ha elaborado un diagrama con el objetivo de representar de forma visual la estructura del sistema, Ilustración 3, mostrando los nodos involucrados, los tópicos que publican o suscriben, y las relaciones entre ellos. Esta representación facilita la comprensión del flujo de datos y la interacción entre los distintos componentes del DeepRacer.

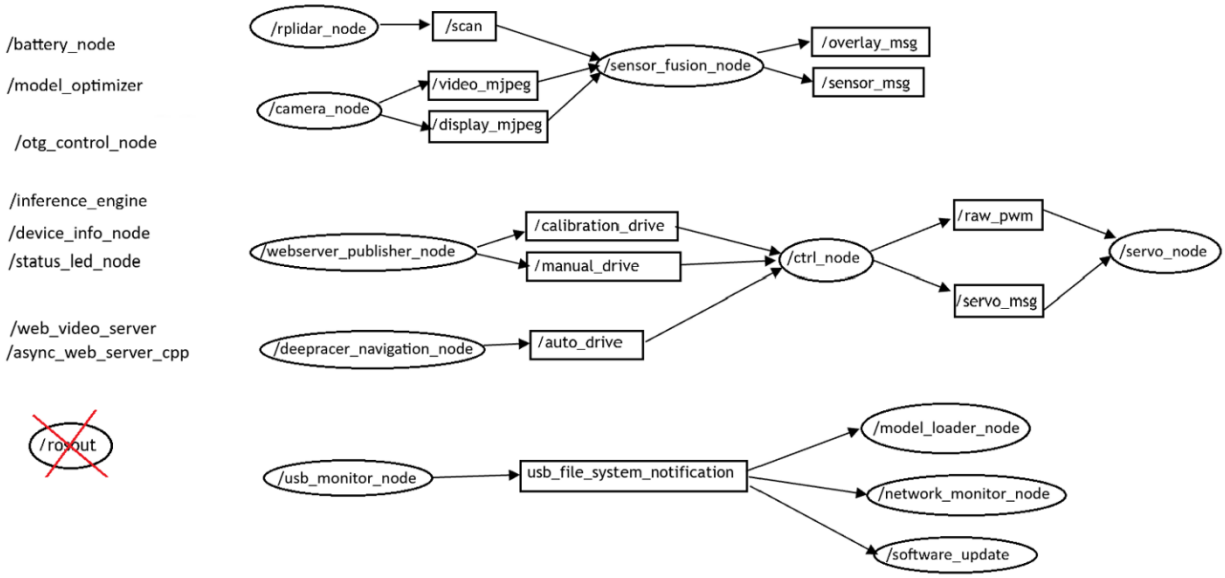


Ilustración 3 diagrama de nodos deepracer

4.2. Tópicos Publicados y Suscritos en ROS2 – Cambios, Mejoras y Consecuencias

El paso de ROS1 a ROS2 ha traído consigo una transformación profunda en la forma en que los nodos se comunican mediante tópicos. Aunque la lógica funcional en muchos casos se mantiene, ahora los nodos interactúan con una arquitectura más modular, robusta y flexible. Uno de los cambios más significativos es la desaparición del nodo maestro (roscore), lo que ha eliminado un posible punto único de fallo y ha dado paso a una arquitectura basada en DDS (Data Distribution Service), permitiendo que los tópicos funcionen de forma más distribuida y eficiente, incluso en entornos más grandes o multi-robot.

Ahora, nodos como el `ctrl_node` no solo publican en tópicos como `/ctrl_pkg/raw_pwm` para un control de bajo nivel mediante señales PWM, sino que también utilizan `/ctrl_pkg/servo_msg` para enviar comandos procesados y más refinados, mejorando el control del vehículo. Lo que se refleja con el ejemplo de publicación anterior es la separación que permite un diseño más limpio y modular, donde las tareas se desacoplan, facilitando la depuración, el mantenimiento y la posibilidad de reutilizar componentes en distintos contextos.

Otro ejemplo interesante es el `software_update`, que ahora no solo publica el porcentaje de progreso de las actualizaciones mediante `/deepracer_systems_pkg/software_update`, sino que también escucha eventos desde `/usb_monitor_pkg/usb_file_system_notification` y `/deepracer_systems_pkg/network_connection_status`. Este enfoque basado en eventos le otorga una mayor autonomía, permitiéndole reaccionar automáticamente ante cambios del sistema sin intervención manual constante.

También hay mejoras notables en nodos como `sensor_fusion_node`, que ahora se suscribe a varios flujos de datos como imágenes y escaneos LIDAR, lo que le permite fusionar datos sensoriales de múltiples fuentes de forma más natural y eficaz. Lo mismo

sucede con `servo_node`, que ahora opera prácticamente de forma completamente reactiva mediante los tópicos, sin depender tanto de servicios como en ROS1.

En resumen, el sistema ha ganado en claridad, escalabilidad y reactividad. El uso de tópicos en ROS2 no solo facilita la implementación, sino que habilita nuevas formas de trabajar más limpias y desacopladas, sentando las bases para sistemas autónomos más inteligentes y robustos. Los servicios de la nueva versión se encuentran en la Tabla 5.

NODO	Servicios
/Battery Node	battery_level/BatteryLevelSrv Se llama para obtener la información actual del nivel de la batería del vehículo en el rango de [0 a 11].
/Ctrl_node	**Servicios** vehicle_state enable_state get_car_cal set_car_cal set_car_led get_car_led get_ctrl_modes model_state autonomous_throttle is_model_loading **pide servicio auxliar a** /model_optimizer_pkg/model_optimizer_server /inference_pkg/load_model /inference_pkg/inference_state /servo_pkg/servo_gpio /servo_pkg/set_calibration /servo_pkg/get_calibration /servo_pkg/get_led_state /servo_pkg/set_led_state /deepracer_navigation_pkg/navigation_throttle /deepracer_navigation_pkg/load_action_space
/Inference	load_model inference_state
/camera_node	media_state
model/optimizer	model_optimizer_server
/Navegation_node	action_space_service throttle_service
/rosout	No existe

/rplidar_node	/stop_motor /start_motor
/sensor_fusión_node	sensor_data_status configure_lidar
/servo_node	servo_gpio set_calibration get_calibration set_led_state get_led_state
/software_update	software_update_check begin_update software_update_state **pide servicio auxiliar a** /status_led_pkg/led_blink /status_led_pkg/led_solid /usb_monitor_pkg/usb_file_system_subscribe /usb_monitor_pkg/usb_mount_point_manager
/web_video_server	
/webserver	**pide servicio auxiliar a** ctrl_pkg/vehicle_state /ctrl_pkg/enable_state /ctrl_pkg/get_car_cal /ctrl_pkg/set_car_cal /device_info_pkg/get_device_info /i2c_pkg/battery_level /sensor_fusion_pkg/sensor_data_status /ctrl_pkg/set_car_led /ctrl_pkg/get_car_led /ctrl_pkg/get_ctrl_modes /deepracer_systems_pkg/verify_model_ready /sensor_fusion_pkg/configure_lidar /ctrl_pkg/model_state /ctrl_pkg/is_model_loading /deepracer_systems_pkg/console_model_action /deepracer_systems_pkg/software_update_check /deepracer_systems_pkg/begin_update /deepracer_systems_pkg/software_update_state /ctrl_pkg/autonomous_throttle /deepracer_systems_pkg/get_otg_link_state

/model_loader_node	verify_model_ready console_model_action **pide servicio auxiliar a** /status_led_pkg/led_blink /status_led_pkg/led_solid /usb_monitor_pkg/usb_file_system_subscribe /usb_monitor_pkg/usb_mount_point_manager /model_optimizer_pkg/model_optimizer_server
/network_monitor_node	**pide servicio auxiliar a** /status_led_pkg/led_blink /status_led_pkg/led_solid /usb_monitor_pkg/usb_file_system_subscribe /usb_monitor_pkg/usb_mount_point_manager
/ otg_control_node	Ejecuta scripts para inicializar o deshabilitar la conexión Ethernet por USB
/status_led_node	led_solid led_blink
/usb_monitor_node	usb_file_system_subscribe usb_mount_point_manager
/async_web_server_cpp	Ninguno
/device_info_node	get_device_info

Tabla 5. Servicios de los Nodos

4.3. Servicios en ROS2 – Cambios, Modernización y su Impacto en el Sistema

Aunque los servicios en ROS2 siguen cumpliendo la misma función general que en ROS1, es decir, permiten la comunicación síncrona entre nodos, su definición, implementación y uso han sido modernizados para ajustarse al nuevo sistema. En ROS2, la introducción de rclcpp y rclpy[3] (1) ha transformado la manera en que los servicios son escritos, lanzados e invocados. Se ha apostado por una mayor claridad y un mejor manejo de errores y dependencias.

Uno de los grandes cambios es el desplazamiento del protagonismo de los servicios hacia los tópicos. Nodos que antes dependían completamente de servicios, como `servo_node`, ahora han transferido buena parte de sus funcionalidades a comunicaciones asincrónicas basadas en tópicos, permitiendo una integración más suave con el resto de nodos.

Eso no significa que los servicios hayan desaparecido. Al contrario, ahora se utilizan con más criterio. Por ejemplo, `usb_monitor_node` sigue ofreciendo funciones críticas como la gestión de archivos y montajes USB, pero estas se han encapsulado de forma que los servicios sean llamados sólo cuando es estrictamente necesario. Esto permite que los

[3] Nota: rclcpp y rclpy son las bibliotecas cliente que permiten programar nodos de ROS2 en C++ y Python, respectivamente.

servicios se utilicen de forma más eficiente, reservando su uso para tareas puntuales o críticas donde es vital garantizar la respuesta del otro nodo.

Además, la estructura del código fuente ha sido reorganizada, centralizando dependencias y configuraciones en archivos como `CMakeLists.txt` y `package.xml`, más adelante son explicados, lo que mejora el mantenimiento y reduce errores de configuración. El cambio de archivos de lanzamiento en XML a *scripts* Python también contribuye a un mayor control, algo especialmente útil en el despliegue de servicios con parámetros dinámicos.

En definitiva, aunque los servicios en ROS2 han perdido algo de protagonismo frente a los tópicos, lo han ganado en calidad. Son más seguros, más fáciles de mantener y, sobre todo, están mejor integrados en una arquitectura moderna, preparada para los desafíos de la robótica distribuida del futuro.

4.4. Cambios en los nodos propios.

Como vimos en el capítulo anterior, el vehículo actualmente cuenta con tres nodos que no son originales, o también podríamos llamarlos propios. Estos tres nodos se incorporaron en el trabajo de Ignacio Royuela para permitir la conducción autónoma del vehículo con medios propios.

Durante la migración de estos nodos del vehículo DeepRacer de ROS1 a ROS2, uno de los primeros cambios que fue necesario aplicar en los tres nodos fue la transformación de su estructura de programación. En ROS1, el código se escribía de forma procedural, es decir, como un conjunto de funciones sueltas con variables globales. En cambio, ROS2 promueve una arquitectura basada en clases, lo que nos obligó a encapsular la lógica de cada nodo dentro de clases que heredan de `rclpy.node.Node`. Es decir, en lugar de tener funciones independientes sueltas en el *script*, ahora estas se integran como métodos dentro de clases como `AutonomousControlNode`, `CloudControlNode` o `AdminCommunicationsNode`. Esto ha mejorado la organización del código, ha hecho más sencillo reutilizarlo y ha reducido errores al eliminar el uso de variables globales.

Además de reestructurar el código en clases, también fue necesario cambiar la forma en que se crean y destruyen las suscripciones y publicaciones. En ROS1 se usaban funciones como `rospy.Subscriber` y `rospy.Publisher`, pero en ROS2 usamos `self.create_subscription()` y `self.create_publisher()`, que son métodos asociados al objeto del nodo. Esto permite que todo esté controlado desde dentro de la clase y que el sistema gestione mejor los recursos, evitando suscripciones activas innecesarias cuando el nodo ya no las necesita. Cuando ya no queremos recibir más datos, ahora podemos llamar a `self.destroy_subscription()` para liberar la memoria correctamente. Estos cambios permiten evitar errores que antes eran comunes cuando un nodo se reiniciaba o se apagaba de manera parcial.

Uno de los aspectos más importantes en la lógica de los nodos ha sido la gestión de servicios. En ROS1, para llamar a un servicio como por ejemplo el de calibración del volante o el control de los LEDs, se usaba `rospy.ServiceProxy`. En ROS2, en cambio, usamos clientes y servidores de servicios mediante `self.create_client()` y `self.create_service()`, y las llamadas se hacen de forma asíncrona con `call_async()`. Esta forma actualizada de trabajar permite que el nodo no se bloquee mientras espera una respuesta. Así, por ejemplo, si el servicio de los LEDs está momentáneamente no disponible, el nodo sigue funcionando normalmente y puede reintentar más adelante, sin colgarse. Además, sí se necesita esperar la respuesta para

continuar, podemos usar `rclpy.spin_until_future_complete(self, future)` para hacerlo de forma segura.

La comunicación entre los distintos componentes del sistema también ha mejorado gracias a la eliminación del nodo maestro. En ROS1, era obligatorio tener un nodo central (roscore) que coordinara a los demás. En ROS2, esto desaparece gracias al uso de DDS (Data Distribution Service), que permite que los nodos se comuniquen entre sí sin depender de un único punto. Esto es útil en el DeepRacer, ya que ahora se podrían tener nodos funcionando en distintos dispositivos sin problemas.

Por ejemplo, podríamos ejecutar nodos desde un servidor o portátil, como una cámara externa que le proporcione información de su entorno haciendo que los datos se integren con los del coche en tiempo real, lo que mejora la escalabilidad y la tolerancia a fallos del sistema. Esto podría abrir la puerta a la sustitución del protocolo MQTT por ROS2 en el envío de las imágenes si se considerara interesante.

Una novedad interesante en ROS2 ha sido la configuración del sistema de calidad de servicio, conocido como QoS (Quality of Service). Cuando se publican datos como los comandos de giro y aceleración, ahora se puede definir con qué fiabilidad y frecuencia deben enviarse. Esto es especialmente útil en sistemas como el del DeepRacer, donde las conexiones pueden no ser estables. Por ejemplo, al publicar mensajes del tipo `ServoCtrlMsg`, se añade un perfil con `QoSProfile(depth=10)`, que indica cuántos mensajes se guardan en cola si no se pueden entregar al instante.

La forma de tratar con las imágenes también ha cambiado ligeramente. Seguimos usando `CvBridge` para convertir los mensajes de ROS en imágenes de OpenCV, pero su uso ahora se encuentra encapsulado dentro de las clases y adaptado al nuevo tipo de mensaje `CameraMsg`, que es propio de ROS2. Esto nos obligó a cambiar cómo accedemos a las imágenes, usando `msg.images[0]` y aplicando `self.bridge.imgmsg_to_cv2(...)`. Al hacerlo dentro del nodo como un método y no como una función externa, el mantenimiento del código es más sencillo y modular.

En cuanto al control por red, uno de los retos fue asegurar que los nodos que envían y reciben datos por *sockets* UDP lo hagan de manera no bloqueante. En ROS1, una lectura del *socket* con `recvfrom()` podía congelar el nodo si no llegaban datos. En ROS2, esto se resolvió usando `select.select()` para comprobar si hay datos antes de intentar leer, evitando así bloqueos innecesarios. Este cambio mejora la fiabilidad del sistema en escenarios reales, donde las comunicaciones pueden ser inestables o interrumpirse.

También se mejoró la forma de publicar mensajes periódicos, como los del sensor IMU. En ROS1, esto se hacía con `threading.Timer`, pero ahora se puede utilizar `create_timer()` dentro del nodo para lanzar acciones repetidas sin interferir con el hilo principal. Aunque algunos casos se siguen resolviendo con procesos separados mediante `multiprocessing.Process`, estos ahora se lanzan desde dentro de los nodos y se controlan con atributos como `self.MQTT_periodic_process` y `self.periodic_process_launched`. Esto evita que se creen procesos duplicados o que queden procesos activos después de apagar el nodo.

Otro cambio clave ha sido en la trazabilidad y el manejo de errores. En vez de imprimir mensajes con `print()`, en ROS2 se usa `self.get_logger().info()` o `self.get_logger().error()`, que permite ver los mensajes desde cualquier herramienta compatible con ROS2. Así, si algo falla, es mucho más fácil saber qué pasó y dónde ocurrió. Además, muchos fragmentos del código están ahora envueltos en `try-`

`except`, lo que evita que un pequeño error detenga todo el nodo. Esto ha sido fundamental, por ejemplo, al procesar imágenes o al intentar llamar a un servicio que puede no estar disponible.

El sistema de control por la nube también ha ganado en flexibilidad. Ahora, cuando se recibe una orden desde el servidor remoto para comenzar a enviar datos, se inicializan suscripciones y *sockets* solo si no estaban activos, y se destruyen correctamente cuando se recibe la orden de parar. Esto se hace desde el método `handle_enable_cloud_control()` dentro del `CloudControlNode`. Si el sistema intenta volver a activar o desactivar algo que ya está activo o inactivo, se devuelve un mensaje como "Already done", evitando errores o comportamientos indeseados.

Además, algunos detalles como la transmisión de datos con `pickle` y `struct`, aunque se han mantenido, ahora se encapsulan dentro de métodos como `send_data()` y están protegidos por un `Lock`, lo que evita que varios hilos accedan al *socket* al mismo tiempo. Esto mejora la seguridad del sistema frente a errores por acceso simultáneo, algo que en ROS1 era difícil de controlar.

También se mejoró el arranque automático de los servicios más importantes. Por ejemplo, en el nodo del administrador, al iniciarse el sistema se activa automáticamente la cámara y se pone el vehículo en modo manual mediante las llamadas `self.set_manual_mode()` y `self.llamar_servicio_video()`. Esto evita que el operador tenga que hacerlo manualmente cada vez que se reinicia el robot.

En la versión original con ROS1, la clase Net_artemis, encargada de obtener información de red como la IP pública, la intensidad de la señal Wi-Fi o los resultados de escaneos, se encontraba en un archivo independiente y era importada directamente como un módulo externo en todos los nodos que la necesitaban. Tras la migración a ROS2, se decidió integrar esta clase directamente dentro del nodo AdminCommunicationsNode. Este cambio se hizo para centralizar su uso, ya que solo este nodo es el encargado de gestionar la conexión con el servidor y de publicar información de red mediante MQTT. Al incluir Net_artemis en el mismo archivo, se facilita su mantenimiento, se reduce la dependencia externa y se asegura que esté siempre disponible en el entorno donde realmente se utiliza.

Finalmente, el sistema de indicadores LED se ha vuelto más preciso y robusto. En ROS2, se encapsuló el uso del servicio `SetLedCtrlSrv` dentro del método `led_state(...)`, que verifica que el servicio esté disponible antes de usarlo. Los colores del LED siguen indicando el estado del vehículo (verde para local, azul para nube, amarillo para administrador, etc.), pero ahora se actualizan de forma más fiable y con menos errores. Esto ayuda a entender visualmente en qué estado se encuentra el coche en cada momento.

En resumen, la migración a ROS2 ha implicado una transformación profunda pero beneficiosa para los tres nodos principales del DeepRacer. Se ha mejorado la organización del código, la gestión de recursos, la robustez frente a errores, y la capacidad de integración con otros sistemas distribuidos, lo que hace que el vehículo esté mejor preparado para enfrentar entornos reales y más complejos.

Para que el coche encienda los nodos automáticamente sin que tengamos que iniciarlos manualmente cada vez, hemos modificado el archivo del servicio `ScriptServicioArtemis.sh`. Antes, los nodos se ejecutaban directamente con bash, pero eso daba errores porque no se cargaban bien los entornos ni se usaban permisos de administrador. Ahora, usamos sudo `bash -c` para ejecutar cada nodo por separado, asegurando que se cargan correctamente los archivos `setup.bash` de ROS y Deepracer

antes de iniciar cada nodo. Así, el servicio se activa bien al arrancar el coche y todo funciona sin problemas. Este archivo se encuentra en `/etc/systemd/system`.

Tambien se modifica el archivo `ARTEMIS.service` para que el servicio se ejecute como root.

State	Explanation	Command to reach the state	Vehicle reaction
Start-Up	`ARTEMIS.service` starts running. All parameters of the vehicle.conf configuration file are retrieved, the steering system is calibrated and the connection to the MQTT broker is attempted	Start the vehicle pressing the button and wait about 1 minute	
AM-Cloud	The vehicle transmits the images to the server configured in the `vehicle.conf` file and waits to receive address control messages	`AM-Cloud`	
AM-Off	The vehicle is awaiting further orders through MQTT. It maintains its position.	`AM-Off`	
AM-Local	The vehicle begin to process the images and control the vehicle according to the autonomous driving algorithm installed in it	`AM-Local`	
Calibration	The vehicle calibrates the steering system with the new parameters received through MQTT	`Calibrate maxvalue midvalue minvalue`	
Unknown command	The vehicle has received an unknown command via MQTT and ignores it.	Any other command not specified in this documentation.	
No connection	The vehicle fails to establish a connection with the MQTT server and repeatedly retries to establish a connection	Vehicle disconnected from the network or `vehicle.conf` misconfigured.	

Ilustración 4 colores del coche según estados[4]

[4] Imagen sacada del repositorio de GitHub del proyecto Artemis.

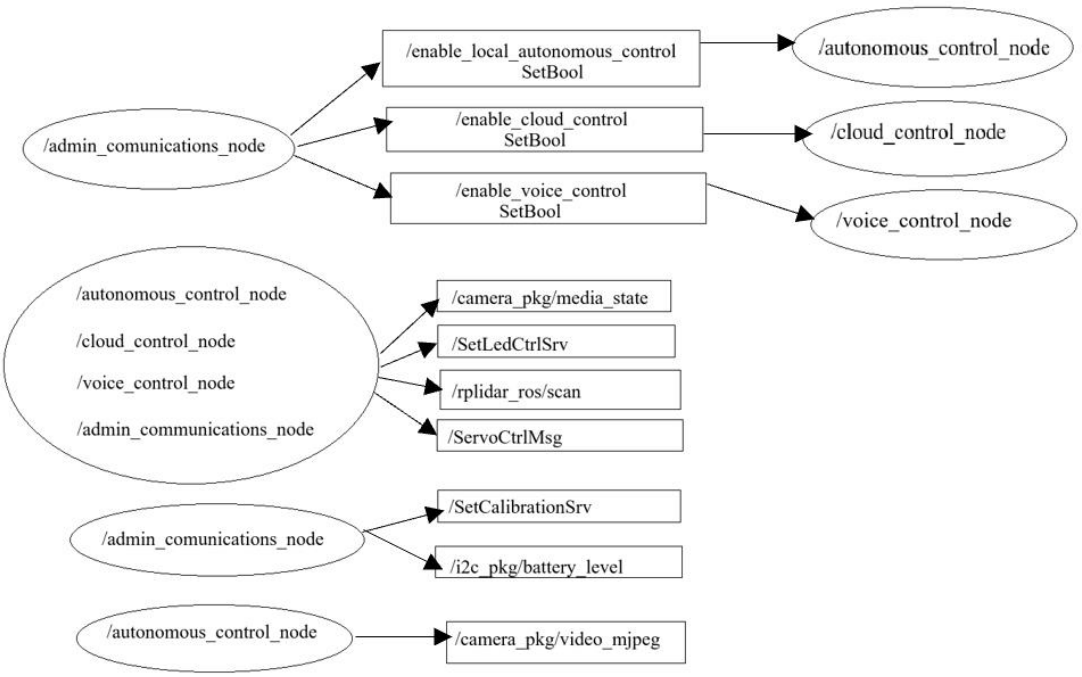


Ilustración 5 diagrama de comunicación de los nodos nuevos con los servicios

```
#!/bin/bash

sleep 4

sudo bash -c "source /opt/ros/foxy/setup.bash; source /opt/aws/deepracer/llb/setup.bash;  source /home/deepracer/soft_artemis/install/setup.bash; ros2 run soft_artemis admincomunication_node.py" &
sudo bash -c "source /opt/ros/foxy/setup.bash; source /opt/aws/deepracer/llb/setup.bash;  source /home/deepracer/soft_artemis/install/setup.bash; ros2 run soft_artemis cloud_control_node_UDP.py" &
sudo bash -c "source /opt/ros/foxy/setup.bash; source /opt/aws/deepracer/llb/setup.bash;  source /home/deepracer/soft_artemis/install/setup.bash; ros2 run soft_artemis autonomous_control_node.py" &
sudo bash -c "source /opt/ros/foxy/setup.bash; source /opt/aws/deepracer/llb/setup.bash;  source /home/deepracer/soft_artemis/install/setup.bash; ros2 run soft_artemis voice_control_node.py" &
```

Ilustración 6 inicio del servicio ARTEMIS.service

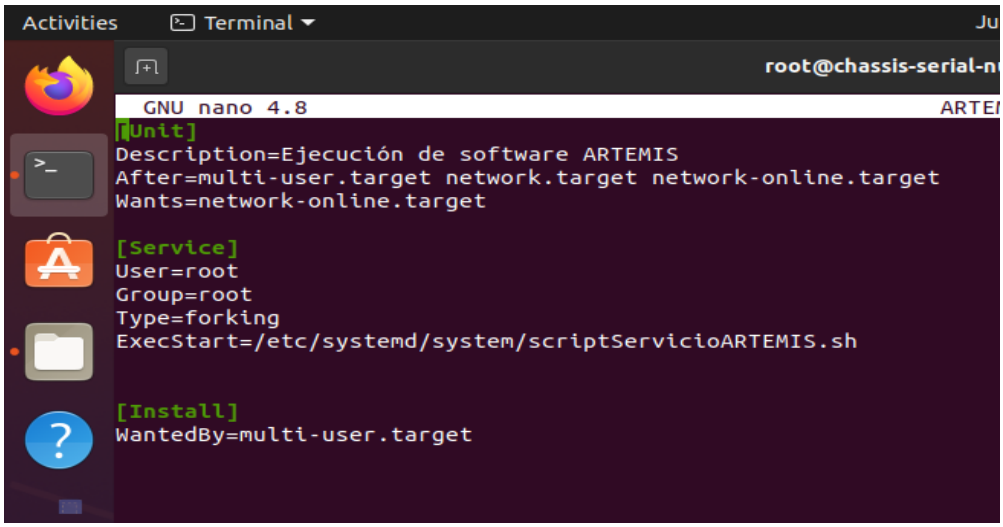


Ilustración 7 archivo ARTEMIS.service

4.5. Integración de Control por Voz en el Sistema DeepRacer

Como parte del proceso de evolución e interacción avanzada con el vehículo autónomo DeepRacer, se ha desarrollado un nuevo módulo que permite el control por comandos de voz, abriendo una vía más intuitiva, accesible y natural de comunicación entre el usuario y la máquina. Este desarrollo no solo mejora la experiencia de uso del sistema, sino que también aporta valor pedagógico y práctico en entornos de demostración, investigación o accesibilidad.

La arquitectura propuesta para este nuevo sistema se compone de dos elementos complementarios y distribuidos: un cliente de reconocimiento de voz ejecutado en un portátil, y un nodo ROS2, ubicado en el DeepRacer, que interpreta y ejecuta las órdenes recibidas.

En primer lugar, el cliente de control por voz, desarrollado en Python, utiliza la popular librería speech_recognition para capturar audio en tiempo real mediante el micrófono del portátil y transformarlo en texto mediante el uso de un motor de reconocimiento de voz basado en la API de Google (para que funcione correctamente se deberá dar permisos a Google para utilizar el micrófono, de otra forma no funcionará). El sistema está configurado para operar en idioma español y reconoce un conjunto limitado de comandos: “avanzar”, “frenar”, “izquierda”, “derecha”, “retroceder” y “salir”. También sería posible implementarlo para otros idiomas y conseguir que reconozca tanto inglés como español, por ejemplo. Una vez reconocido el comando, se transmite al vehículo DeepRacer utilizando un *socket* de red sobre el protocolo UDP o TCP, dependiendo de la configuración del nodo ROS en el robot (en este caso será UDP).

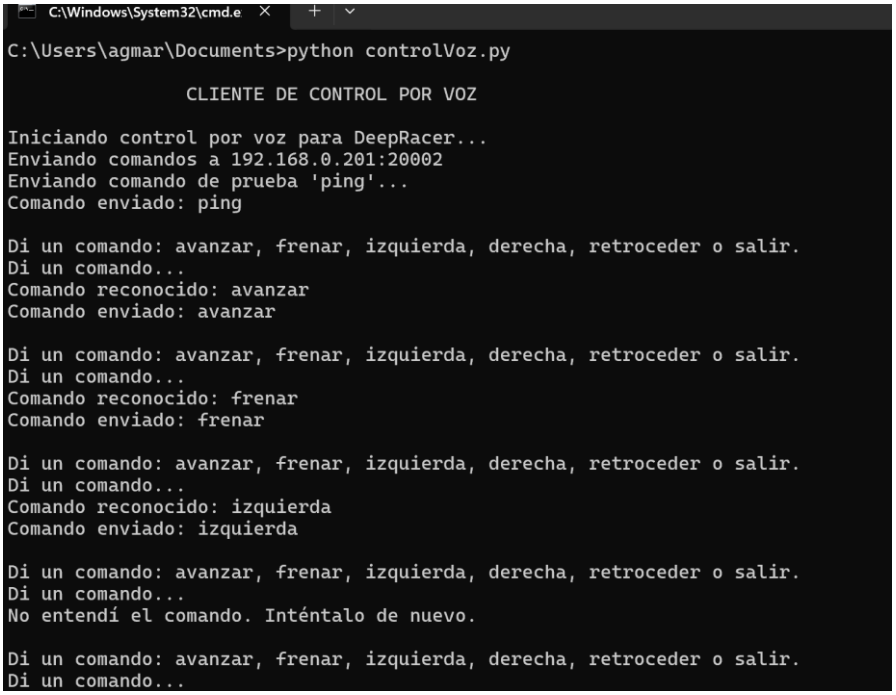


Ilustración 8 Ejecución en portátil del código controlVoz.py

Este enfoque distribuye la carga computacional del procesamiento de voz fuera del DeepRacer, evitando sobrecargar el procesador del vehículo, lo cual es especialmente relevante en sistemas embebidos con recursos limitados. Además, esta separación permite

futuras integraciones con otros asistentes virtuales o sistemas embebidos sin necesidad de modificaciones en el nodo ROS del vehículo.

En el extremo receptor se encuentra el nodo `VoiceControlNode`, implementado en ROS2, cuyo objetivo es recibir los comandos enviados desde el portátil, interpretar su significado, y traducirlos en acciones concretas sobre el vehículo. Este nodo crea un *socket* TCP que se mantiene escuchando conexiones entrantes. Una vez establecida la conexión, se activa un temporizador que verifica constantemente si existen nuevos mensajes. En caso de recibir un comando de voz válido, este se procesa mediante el método execute_command, que contiene una lógica de mapeo entre el texto recibido y las acciones físicas a realizar (como acelerar, frenar o girar).

Es importante destacar que el nodo ROS2 incluye un servicio llamado `/enable_voice_control`, que permite habilitar o deshabilitar dinámicamente el control por voz. Este mecanismo introduce una capa adicional de seguridad y control, ya que el sistema no reaccionará a comandos de voz a menos que dicho modo esté explícitamente activado, evitando así acciones no deseadas durante otras fases del uso del vehículo (por ejemplo, durante una demostración o prueba autónoma).

La decisión de utilizar mensajes ROS estándar (`ServoCtrlMsg`) para el control del vehículo garantiza una integración total con el resto de los componentes del sistema, lo cual permite mantener la modularidad y la coherencia del ecosistema. Además, el sistema aprovecha la infraestructura de ROS2 para ofrecer un comportamiento reactivo, trazabilidad mediante logs (`get_logger().info()`), y fácil escalabilidad a más comandos o funcionalidades futuras.

Este módulo no solo aporta una solución tecnológica funcional, sino que también constituye un claro ejemplo de cómo el reconocimiento de voz puede integrarse en sistemas robóticos de forma eficiente y modular**.** A nivel educativo, permite explorar conceptos como comunicación distribuida, procesamiento de voz, programación en red, y diseño orientado a servicios.

La posibilidad de controlar el DeepRacer mediante voz se traduce también en una mejora en términos de accesibilidad, ya que facilita su manejo por parte de usuarios con movilidad reducida o en situaciones donde el uso de dispositivos de entrada tradicionales (teclado, ratón) no es viable.

4.6. Organización e Implementación del Paquete soft_artemis en ROS2

En el contexto de la migración del sistema del vehículo autónomo DeepRacer a ROS2, se tomó la decisión de reestructurar toda la lógica de los nodos dentro de un paquete Python dedicado, con el objetivo de mejorar la mantenibilidad, modularidad y escalabilidad del sistema. Este paquete fue creado bajo el nombre `soft_artemis` y responde a las buenas prácticas actuales en el desarrollo de *software* sobre ROS2, especialmente en lo que se refiere a la gestión de nodos, reutilización de código y organización estructurada (14).

Para su creación se utilizó el comando:

```
ros2 pkg create --build-type ament_python soft_artemis
```

que genera una plantilla básica con los archivos y estructuras necesarias para que ROS2 reconozca y maneje el paquete como un módulo Python autónomo. Esta plantilla se

adaptó posteriormente para incorporar todos los nodos funcionales del sistema, así como los módulos auxiliares y archivos de configuración que estos necesitan.

El resultado de esta configuración puede observarse en la estructura del paquete, donde encontramos archivos fundamentales como `package.xml`, que actúa como descriptor del paquete ante el sistema ROS2, definiendo sus dependencias, nombre, versión y mantenedores, y `setup.py`, que se encarga de declarar los puntos de entrada y *scripts* ejecutables necesarios para lanzar los nodos mediante ros2 run.

Junto a estos, se incluye una carpeta con el mismo nombre del paquete, `soft_artemis`, que es el núcleo funcional de la implementación. Esta carpeta contiene tanto los *scripts* correspondientes a los distintos nodos como otros módulos reutilizables. La presencia del archivo `__init__.py` en esta carpeta es lo que permite tratarla como un módulo Python completo, habilitando así las importaciones internas entre archivos. Esta característica ha resultado clave para mantener el sistema ordenado, ya que ha permitido, por ejemplo, que clases como `artemis_autonomous_car`, encargada de encapsular la lógica de navegación y control autónomo, puedan ser utilizadas desde cualquier nodo mediante una sencilla instrucción de importación (from . import artemis_autonomous_car). Este nivel de reutilización es fundamental para evitar la duplicación de código y asegurar un comportamiento coherente entre los distintos componentes del sistema.

Entre los archivos presentes en el paquete se encuentran los nodos principales del sistema: `autonomous_control_node.py`, encargado del control autónomo local, `cloud_control_node_UDP.py`, que se encarga de recibir y ejecutar comandos desde un servidor remoto en la nube, y `admin_communications_node.py`, responsable de la conexión con el servidor MQTT, la gestión de estados del vehículo, y la calibración del sistema de dirección. A estos se ha sumado un nodo adicional, `voice_control_node.py`, que introduce una nueva funcionalidad de control por voz, lo cual añade una capa adicional de interacción humana directa con el vehículo.

Dentro del paquete también se incluyen archivos con funcionalidades específicas, como `bmi160.py`, que gestiona las lecturas del acelerómetro y giroscopio integrados en el vehículo, y `vehicle.conf`, un archivo de configuración en formato estándar que almacena información persistente como la calibración del sistema de dirección, las direcciones IP de los servidores y otros parámetros relevantes del sistema.

Un caso particular es el del archivo `net_artemis.py`, que contiene funciones dedicadas a obtener parámetros de red como la intensidad de señal WiFi, la IP pública o la interfaz de red activa. Este módulo fue diseñado inicialmente para ser importado como una clase en el nodo de administración, pero durante las pruebas se detectaron errores en la carga del módulo cuando se hacía la importación directa desde el interior de la clase principal. Por este motivo, y dado que su funcionalidad solo era requerida por un nodo concreto, se optó por incluir sus funciones directamente en el archivo `admin_communications_node.py`, fuera de la clase principal. Esta solución, aunque menos elegante desde el punto de vista de la abstracción, permitió evitar conflictos de ejecución y asegurar la estabilidad del sistema durante su funcionamiento continuo.

En términos de arquitectura de *software*, este enfoque basado en paquetes modulares permite que el sistema sea fácilmente mantenible, y que nuevos nodos o funcionalidades puedan añadirse sin alterar el funcionamiento de los nodos existentes. De igual manera, la estructura adoptada hace posible instalar el paquete en otros entornos ROS2 con tan solo declarar sus dependencias y ejecutar su instalación, lo que habilita su distribución en equipos de desarrollo o robots adicionales.

Esta forma de organizar el código no solo responde a las recomendaciones oficiales de Open Robotics, sino también a las prácticas más estables y sostenibles del desarrollo de *software* distribuido en robótica. La documentación oficial de ROS2 respalda este enfoque, destacando la importancia de construir paquetes autocontenidos, con puntos de entrada bien definidos y configuraciones explícitas a través de archivos como `setup.py` y `package.xml`. Además, el uso de módulos internos y el aprovechamiento de Python como lenguaje de *scripting* principal está alineado con guías ampliamente aceptadas en la comunidad Python para la creación de paquetes.

El resultado final es un sistema más ordenado y reutilizable, en el que cada nodo cumple una función bien delimitada y se integra dentro de un ecosistema coherente, tanto desde el punto de vista de ROS2 como desde una perspectiva lógica.

4.7. Instalación y configuración de Mosquitto

Como se introdujo en el capítulo anterior, el protocolo MQTT permite gestionar la comunicación entre el vehículo y el servidor externo. A través de este protocolo, se intercambian mensajes relacionados con el estado del coche, como la activación de modos de conducción, el envío de datos de sensores o la calibración remota. Por ello, en esta sección se describe el proceso de instalación y configuración del bróker Mosquitto, necesario para que esta comunicación pueda establecerse correctamente.

La instalación de Mosquitto puede realizarse fácilmente en sistemas basados en Linux utilizando los gestores de paquetes correspondientes, en nuestro caso lo hemos instalado en Windows a través de su página oficial.

Una vez instalado, es necesario **modificar el archivo de configuración de Mosquitto** (15)**,** ya que por defecto solo admite conexiones por TCP y no define ningún puerto ni comportamiento asociado a UDP. Para ello, se edita el archivo de configuración principal:

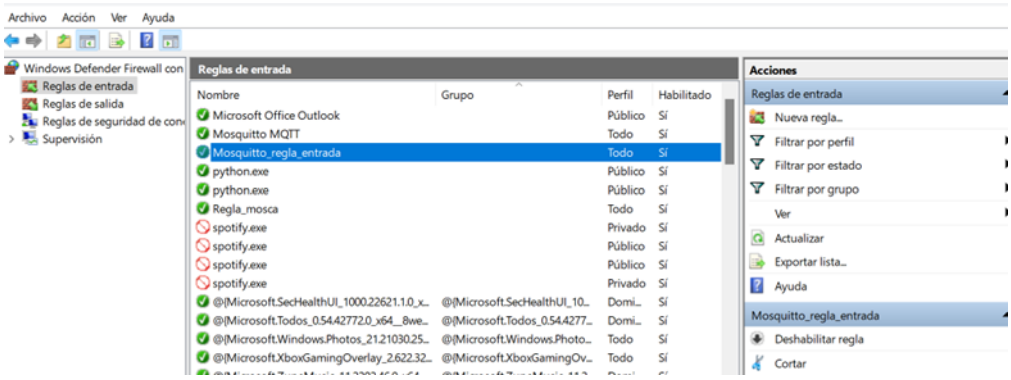
```
mosquitto/mosquitto.conf
```

En este archivo es necesario añadir manualmente una entrada que especifique el comportamiento deseado. En nuestro caso, se ha añadido una sección para permitir comunicaciones sobre UDP, utilizando el siguiente formato:

```
listener 1883
allow_anonymous true
```

Con esta configuración, el broker Mosquitto escuchará tanto en el puerto 1883 para conexiones estándar, permitiendo así una mayor flexibilidad en la forma en que los nodos y clientes (como MQTT Explorer o el nodo de comunicaciones del vehículo) se conectan al sistema.

Además de configurar correctamente el archivo mosquitto.conf, es fundamental garantizar que el sistema operativo permita el tráfico de red en los puertos que se han definido para la comunicación MQTT. Para ello, será necesario crear una regla de entrada y otra de salida en el firewall, adaptadas al puerto utilizado y al protocolo correspondiente (ya sea TCP o UDP).



La razón por la que estas reglas son necesarias es porque, por defecto, muchos sistemas operativos modernos restringen todo el tráfico entrante por motivos de seguridad. Sin estas reglas, los clientes MQTT (como el nodo ROS del vehículo o MQTT Explorer desde otro equipo) no podrían establecer conexión con el broker, incluso si este está configurado correctamente. Del mismo modo, sin una regla de salida, el servidor no podrá responder adecuadamente, provocando fallos de comunicación, bloqueos o pérdidas de mensajes.

Una vez configurado correctamente, Mosquitto puede ejecutarse directamente desde la terminal. Para ello, basta con situarse en el directorio de configuración (si se quiere usar un archivo distinto al predeterminado) y ejecutar:

```
mosquitto -c mosquitto.conf -v
```

El parámetro `-v` activa el modo "verbose", lo que permite ver en tiempo real los mensajes que se reciben y transmiten, así como las conexiones que se establecen o finalizan, proporcionando un mecanismo muy útil de depuración y verificación del sistema.

Finalmente, la combinación de Mosquitto como broker y MQTT Explorer como cliente de desarrollo y análisis proporciona una plataforma robusta y visualmente clara para gestionar toda la infraestructura de comunicación del sistema, desde comandos de movimiento hasta lecturas de sensores o estado de calibración.

En el proceso de puesta en marcha del sistema de comunicación mediante MQTT, es bastante habitual encontrarse con ciertos errores recurrentes que pueden dificultar el correcto establecimiento de la conexión entre el DeepRacer y el servidor Mosquitto. Estos fallos no necesariamente se deben a errores en el código, sino que suelen estar relacionados con la configuración de red, los permisos del sistema o la gestión de servicios en segundo plano.

Para detectar situaciones de fallo, se puede abrir el símbolo del sistema con privilegios de administrador y ejecutar:

```
netstat -ano | findstr :1883
```

Este comando permite identificar si hay alguna conexión activa en el puerto 1883, que es el utilizado por defecto por MQTT. Si se confirma que el puerto está en uso y se desea liberar, se puede forzar el cierre del proceso correspondiente con el siguiente comando, sustituyendo PID por el identificador obtenido previamente:

```
taskkill /PID [PID] /F
```

4.8. Problemas con el Sensor BMI160 en el DeepRacer: Detección, Puerto Dinámico y Solución

En los sistemas de conducción autónoma como el DeepRacer, la precisión en la lectura de datos inerciales es clave para mantener una conducción estable, especialmente cuando se navega en entornos con curvas cerradas o superficies irregulares. Para ello, el DeepRacer cuenta con un sensor inercial BMI160, un componente fundamental que combina un acelerómetro y un giroscopio en un solo chip. Este sensor permite medir tanto la aceleración como la velocidad angular del vehículo, ofreciendo datos críticos para estimar su orientación, detectar derrapes, realizar correcciones de trayectoria y complementar sistemas como la visión por computador o el LIDAR.

El BMI160 se comunica con la placa de procesamiento mediante el bus I2C, una interfaz serie que permite que múltiples sensores compartan el mismo canal físico diferenciándose por sus direcciones. Tradicionalmente, en versiones anteriores del *firmware* de DeepRacer, este sensor se encontraba conectado al bus I2C número 5 con una dirección fija: 0x68, que corresponde a su dirección por defecto. Esto permitía que el sistema operativo y los *scripts* de control localizasen sin dificultad el sensor y accedieran a sus lecturas mediante una ruta de comunicación conocida y constante.

Sin embargo, al actualizar a versiones recientes del sistema, se ha detectado un cambio inesperado en el comportamiento del hardware: aunque el sensor sigue operando bajo la dirección 0x68, ahora aparece conectado a un bus I2C diferente, típicamente el I2C 1 o el I2C 2, en función del entorno o configuración específica del hardware. Este cambio, aparentemente menor, provoca un fallo crítico en los módulos que dependen del sensor, ya que el código original intenta acceder al BMI160 exclusivamente a través del bus I2C 5, donde ya no se encuentra. El resultado es una incapacidad para inicializar el sensor, lo que impide la obtención de datos inerciales y, por tanto, afecta directamente a la estabilidad del sistema de navegación y control.

4.8.1. Dificultad para detectar el sensor: Limitaciones de herramientas estándar

Intentar localizar manualmente el sensor en estos nuevos entornos se convirtió en un desafío no trivial. Las herramientas clásicas como i2cdetect, que muestran un mapa general de dispositivos conectados a un bus I2C, resultaron ineficaces en muchas pruebas: o bien no listaban el sensor, o bien mostraban resultados confusos que no permitían confirmar con certeza su presencia activa. Esto podría deberse a estados de reposo del sensor, condiciones de acceso al bus restringidas, o incluso a configuraciones dinámicas del firmware base.

La herramienta que finalmente permitió identificar de manera confiable la conexión del sensor fue **i2cdump**. Usando un comando como i2cdump 1 0x68, donde el número representa el bus I2C y la dirección 0x68 es la del sensor, fue posible visualizar el contenido de sus registros internos en tiempo real. La presencia de un patrón reconocible en los registros confirmaba que el sensor estaba activo y operativo en ese bus. Este procedimiento tuvo que repetirse para diferentes buses (i2cdump 2 0x68, i2cdump 3 0x68, etc.) hasta encontrar en cuál de ellos había sido conectado dinámicamente el BMI160 en ese arranque concreto.

4.8.2. Dirección móvil y necesidad de adaptación en el *software*

La raíz del problema reside en que la asignación del bus I2C no es fija en esta nueva versión del sistema operativo de DeepRacer. Esto implica que la dirección física del sensor (0x68) se mantiene constante, pero la "ruta lógica" para alcanzarlo cambia, y el código debe estar preparado para ello. El archivo responsable de manejar la comunicación con el BMI160 suele ser un *script* Python llamado bmi160.py, donde se especifica explícitamente el número del bus I2C a usar.

Para restaurar el funcionamiento del sensor, fue necesario editar manualmente este archivo, modificando la línea que inicializa la conexión I2C, por ejemplo:

self.bus = smbus.SMBus(1) # Puede que cada vez que se arranque cambie el número del puerto.

Después de realizar este cambio, era imprescindible recompilar el paquete correspondiente, y finalmente reiniciar el servicio asociado al BMI160 para que el cambio surtiera efecto.

4.8.3. Importancia operativa del BMI160 en el sistema

La ausencia del BMI160 no solo supone una pérdida de datos de orientación: en muchas configuraciones, este sensor también participa indirectamente en tareas de sincronización de sensores, estimación de velocidad (por integración del acelerómetro), detección de inclinaciones que puedan afectar el comportamiento de la cámara, e incluso en rutinas de seguridad como la parada automática ante movimientos inesperados. En proyectos avanzados de DeepRacer, donde se implementan estrategias de navegación basadas en fusión sensorial (por ejemplo, combinando visión, LIDAR e IMU), la falta de datos del BMI160 hace que el sistema reaccione, quedándose desactivado, puesto que el nodo Admin_comunication_node depende directamente de él. En este sentido, su correcto funcionamiento no es opcional, sino un requisito indispensable para la estabilidad del control autónomo.

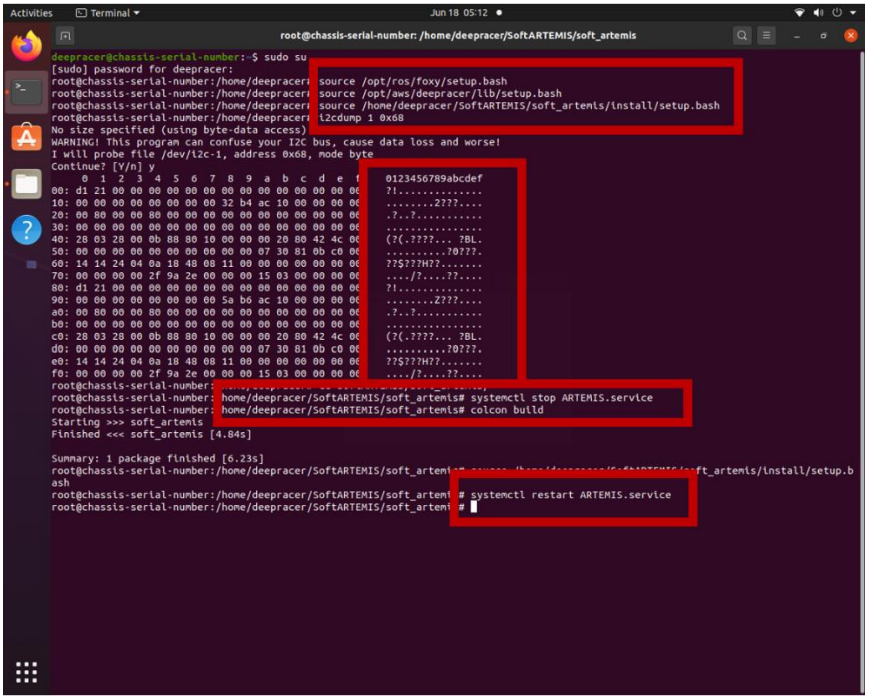


Ilustración 9 Pasos para comprobar y actualizar el bus BMI160

5. Implementación de un Nuevo Algoritmo de Conducción Autónoma Basado en Detección de Bordes

5.1. Justificación del cambio

En la versión inicial del sistema de conducción autónoma desarrollado para el vehículo DeepRacer, la detección de la pista se basaba en la localización de un único camino central a partir de la identificación de contornos de color en un espacio transformado de perspectiva. Esta aproximación, aunque efectiva en condiciones óptimas, presentaba ciertas limitaciones prácticas que motivaron la necesidad de una actualización.

Uno de los principales problemas detectados fue la alta dependencia de las condiciones de iluminación. En situaciones de luz ambiental cambiante, sombras intensas o zonas poco iluminadas, el filtrado de color en el espacio HSV podía fallar, provocando errores en la identificación de la trayectoria o incluso la pérdida completa de referencias visuales. Además, el enfoque anterior estaba basado en encontrar el centro de una sola línea de referencia, lo que lo hacía especialmente vulnerable en pistas amplias o en curvas pronunciadas, donde la única línea detectada podía desaparecer parcial o totalmente del campo de visión de la cámara.

En base a estos problemas observados durante las pruebas reales, se optó por desarrollar un nuevo algoritmo de percepción visual, más robusto y versátil, que se apoyara en técnicas de procesamiento de bordes en lugar de la segmentación de color. El objetivo era lograr un sistema más resistente a cambios de iluminación, capaz de detectar la geometría completa de la pista (no solo una línea central) y que aprovechara la presencia de múltiples líneas de la carretera para calcular una trayectoria más fiable y estable.

5.2. Detección de Líneas y Curvas mediante Canny, Hough y Ventana Deslizante

5.2.1. Introducción

En el desarrollo de sistemas de conducción autónoma, la detección precisa de los límites de la pista es un componente crítico para garantizar una navegación segura y eficiente. En este trabajo, se implementa un algoritmo de visión por computadora en Python para un vehículo DeepRacer, utilizando técnicas avanzadas de procesamiento de imágenes como el detector de bordes Canny (16), la transformada de Hough para identificación de líneas, el uso de ventana deslizante para identificar el circuito y el controlador Stanley para el seguimiento de trayectorias.

El objetivo principal es procesar imágenes capturadas por la cámara del DeepRacer para extraer las líneas de la pista, interpretar su geometría (rectas, curvas) y ajustar la dirección y velocidad del vehículo de manera autónoma. A continuación, se explican los fundamentos teóricos de cada algoritmo utilizado.

5.2.2. Detección de Bordes con el Algoritmo de Canny

En el ámbito del procesamiento de imágenes, detectar bordes es una tarea crucial, ya que permite realizar operaciones fundamentales como la segmentación de regiones, la

identificación de objetos y el análisis de entornos visuales. Entre los diversos métodos existentes, el algoritmo de Canny, introducido en 1986 por John F. Canny, se ha mantenido como una de las técnicas más fiables y precisas, gracias a su resistencia al ruido y su capacidad para delinear bordes nítidos, continuos y coherentes.

La esencia del algoritmo consiste en encontrar las zonas de una imagen donde la intensidad de los píxeles varía bruscamente, ya que estas transiciones suelen marcar los contornos de objetos o estructuras. Canny desarrolló su algoritmo siguiendo tres principios:

- Alta detección: asegurar que todos los bordes reales se detecten mientras se minimizan los bordes espurios, es decir, lo que intenta es evitar detectar como borde algo que en realidad no lo es.
- Precisión de localización: posicionar los bordes detectados lo más cerca posible de su ubicación real.
- Respuesta única: evitar la aparición de múltiples detecciones para un mismo borde.

Estos criterios han permitido que el método siga siendo ampliamente utilizado, incluso en aplicaciones modernas que abarcan desde sistemas robóticos hasta tecnologías médicas.

5.2.3. Etapas del Algoritmo de Canny

La ejecución del algoritmo se estructura en varios pasos fundamentales (17):

Filtrado Gaussiano para reducción de ruido

En primer lugar, la imagen se suaviza utilizando un filtro Gaussiano. Este paso tiene como objetivo eliminar las variaciones de alta frecuencia (ruido), que podrían generar detecciones falsas. El parámetro σ (desviación estándar) controla el grado de suavizado: valores mayores eliminan más ruido, pero pueden difuminar detalles importantes; valores menores conservan más bordes, pero arriesgan la aparición de ruido.

Cálculo del gradiente de intensidad

Posteriormente, se calcula el gradiente de intensidad en cada punto de la imagen, determinando tanto su magnitud como su dirección. Esto se realiza comúnmente mediante operadores como Sobel o Prewitt, que aproximan las derivadas parciales. La magnitud indica cuán marcado es el borde, mientras que la dirección señala hacia dónde se produce el cambio más brusco.

Supresión de no máximos

Para refinar los bordes detectados, se aplica una técnica de supresión de no máximos. Consiste en eliminar todos los píxeles que no sean máximos locales en la dirección del gradiente, afinando así los bordes a líneas de un solo píxel de grosor.

Aplicación de doble umbral y seguimiento por histéresis

Esta fase utiliza dos umbrales, uno alto y uno bajo:

- Los píxeles con gradiente mayor que el umbral alto se clasifican como bordes fuertes.

- Aquellos cuyo valor se encuentra entre ambos umbrales son considerados bordes débiles y se conservarán únicamente si están conectados a bordes fuertes.
- Los píxeles con gradientes por debajo del umbral bajo son descartados.

El seguimiento por histéresis permite mantener la coherencia de los contornos, evitando fragmentaciones causadas por ruido o cambios de iluminación.

Cierre de contornos (Opcional)

En implementaciones extendidas, puede añadirse un paso adicional para conectar extremos de bordes próximos, asegurando el cierre de contornos incompletos mediante el seguimiento de máximos de gradiente.

5.2.4. Uso del Algoritmo de Canny en el DeepRacer

En el proyecto DeepRacer, el algoritmo de Canny se emplea para identificar los bordes de la pista de forma robusta, incluso bajo condiciones de iluminación variables. Al basarse en la intensidad de los píxeles en lugar de su color, el sistema se vuelve menos susceptible a factores como sombras, cambios de luz o deterioro de las marcas en el suelo.

Los bordes detectados se utilizan como base para procesos posteriores, como la aplicación de la Transformada de Hough para trazar las líneas que definen el recorrido, y en el método de la ventana deslizante para ayudar la propia identificación de la pista. Finalmente, esta información geométrica se traduce en instrucciones de movimiento (dirección y velocidad) mediante el Controlador Stanley.

Esta combinación de métodos tradicionales, aunque conceptualmente simple, ha demostrado ser sumamente eficaz para vehículos autónomos ligeros como el DeepRacer, ofreciendo un rendimiento sólido y eficiente con un coste computacional reducido.

5.3. Ejemplo de Aplicación del Filtro de Canny Paso a Paso

Para comprender de manera más intuitiva el funcionamiento del algoritmo de Canny y visualizar cómo evoluciona una imagen a lo largo de las diferentes etapas del proceso, se diseñó un pequeño script en Python. Este programa permite aplicar el filtrado de bordes y mostrar claramente cada fase, desde la imagen original hasta la obtención del mapa final de bordes.

Para este ejemplo práctico, se utilizó una fotografía de un circuito, simulando la perspectiva que podría captar la cámara de un DeepRacer desde una vista aérea. Esta elección de imagen busca aproximar de forma realista las condiciones de trabajo de nuestro algoritmo de conducción autónoma.

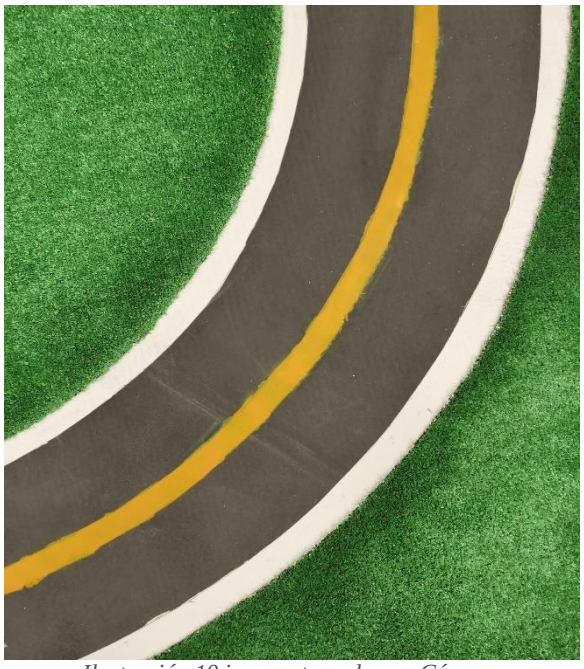


Ilustración 10 imagen tomada con Cámara.

Antes de aplicar el algoritmo de Canny, se procedió a mejorar el contraste de la imagen utilizando una técnica de ecualización de histograma. Este paso adicional es muy útil, ya que permite resaltar mejor las diferencias de intensidad entre las zonas de interés, facilitando una detección de bordes más precisa, especialmente en entornos donde la iluminación puede ser desigual o donde las marcas del suelo no tienen un contraste muy pronunciado.

El flujo de procesamiento implementado fue el siguiente:

- Carga de la imagen en escala de grises.
- Mejora del contraste mediante ecualización de histograma.
- Suavizado de la imagen utilizando un filtro Gaussiano para reducir el ruido.
- Cálculo del gradiente de intensidad (magnitud y dirección) mediante operadores de Sobel.
- Aplicación de la técnica de supresión no máxima para afinar los bordes detectados.
- Realización del doble umbralado y la histéresis para obtener el mapa final de bordes mediante cv2.Canny.

Cada uno de estos pasos fue visualizado mediante gráficos para ilustrar el efecto progresivo de las operaciones sobre la imagen.

Gracias a esta representación paso a paso, resulta mucho más sencillo apreciar la importancia de cada etapa del algoritmo de Canny y entender cómo contribuyen individualmente a la detección robusta de bordes en imágenes reales.

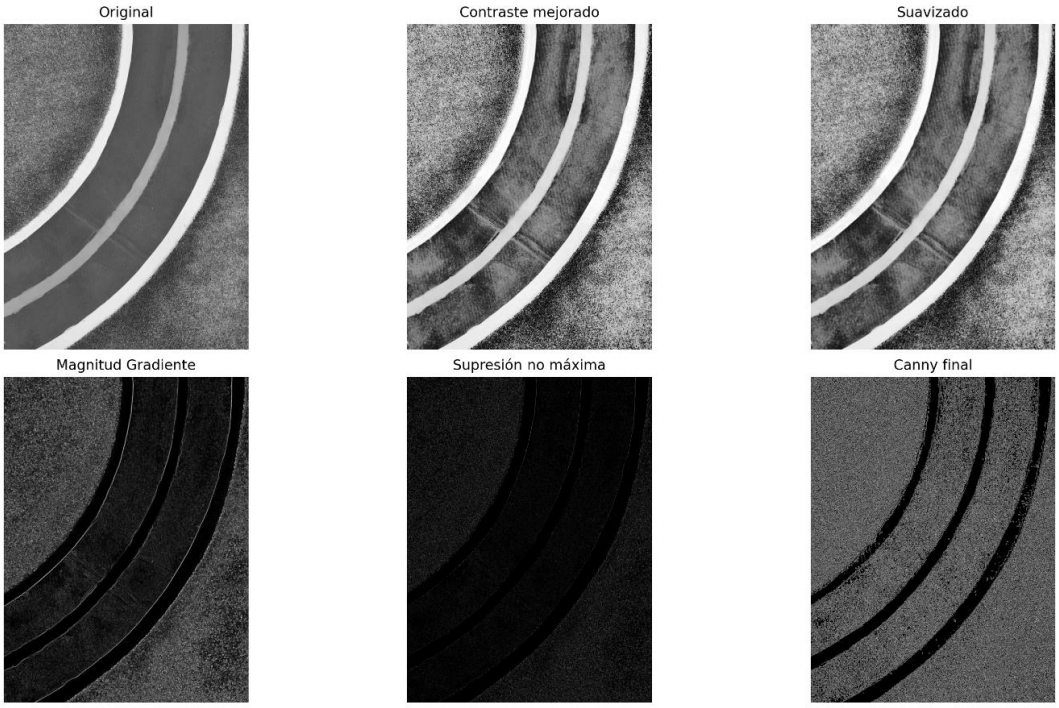


Ilustración 11 imagen procesada por e algoritmo de canny.

5.4. Transformada de Hough: Fundamentos y Aplicación en la Detección de Líneas y Curvas[5]

Dentro del campo de la visión artificial, uno de los retos fundamentales es lograr la detección precisa de estructuras geométricas en imágenes que, inevitablemente, contienen ruido, deformaciones o información parcial. Para abordar este desafío, la Transformada de Hough se presenta a priori como una herramienta extraordinariamente robusta, capaz de detectar líneas, curvas y otras figuras geométricas incluso en condiciones adversas.

La idea principal de este método es cambiar el problema de buscar formas en una imagen por otro más fácil: buscar puntos altos en una especie de "mapa" de parámetros, llamado espacio de Hough (18). En este espacio, cada punto que forma parte de un borde ayuda a marcar las posibles líneas o figuras que podrían pasar por él. Cuando varios puntos coinciden, se puede identificar si hay una línea, un círculo u otra forma en la imagen.

5.5. Origen y Motivaciones

La Transformada de Hough fue propuesta por Paul Hough en 1959 para detectar líneas rectas en imágenes. Más adelante, en 1972, Duda y Hart mejoraron esta técnica usando coordenadas polares, lo que permitió representar mejor las líneas verticales, que daban problemas con las fórmulas antiguas.

Hoy en día, se usa sobre todo esta versión polar, en la que una línea recta se describe con dos valores:

ρ (rho): la distancia desde el origen hasta la línea.

θ (theta): el ángulo que forma la línea con el eje horizontal.

El funcionamiento básico del algoritmo se puede resumir en cuatro pasos:

- Detectar los bordes: se usa un filtro como Canny para encontrar los bordes de la imagen.
- Convertir al espacio de Hough: cada punto detectado se transforma en una curva en un nuevo espacio llamado (θ, ρ).
- Contar coincidencias: se guarda cuántas curvas pasan por cada punto del espacio.
- Buscar los máximos: donde más coincidencias hay, seguramente hay una línea en la imagen original.

Ilustración 12 Detección líneas en una sopa de letras (realizada mediante un código de Python)

Este proceso permite detectar líneas incluso cuando están fragmentadas o parcialmente ocultas, lo que la convierte en una técnica extremadamente robusta frente a ruido o defectos en la imagen.

5.5.1. Variantes: Hough Probabilística y Transformada para Círculos

Para mejorar la eficiencia computacional en imágenes grandes o complejas, se introdujo la Transformada de Hough Probabilística. En esta versión, en lugar de considerar todos los píxeles de borde, se selecciona aleatoriamente un subconjunto de ellos para realizar el voto, reduciendo enormemente el número de cálculos sin comprometer la calidad de los resultados en la mayoría de los casos (19).

Por otro lado, extendiendo el concepto original, surge la Transformada Circular de Hough, diseñada para identificar círculos en una imagen. En este caso, en lugar de buscar líneas rectas, el algoritmo intenta encontrar formas circulares analizando los bordes. Para hacerlo de forma más eficiente, también tiene en cuenta la dirección del borde, lo que ayuda a reducir errores y detectar mejor los círculos reales en la imagen.

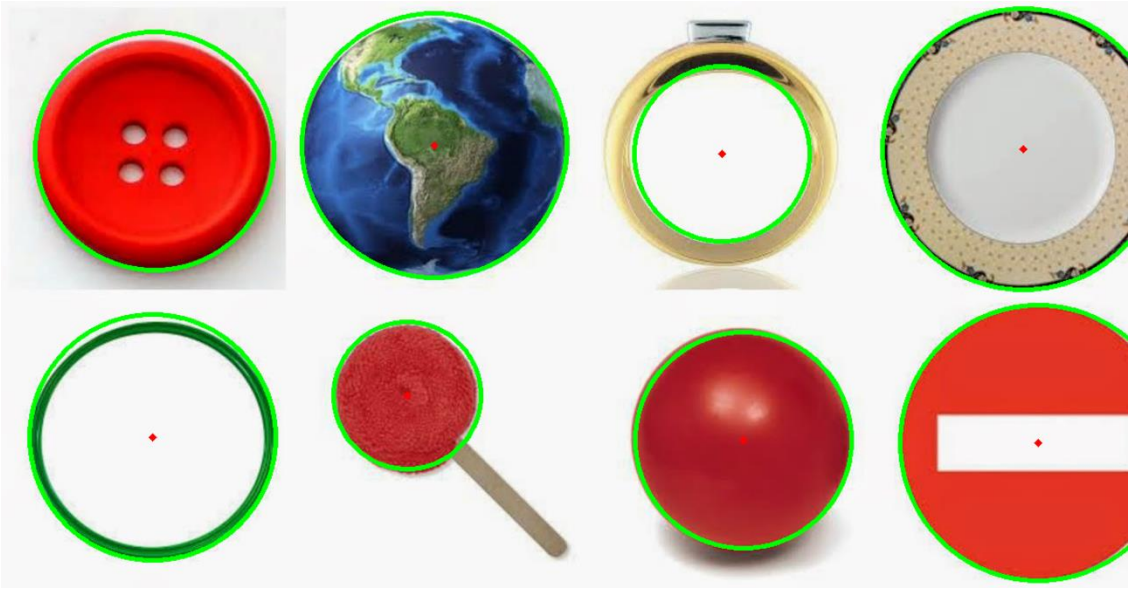


Ilustración 13 Ejemplo transformado de hough circular

5.6. Aplicación de la Transformada de Hough en DeepRacer

Durante las primeras fases del desarrollo, se evaluó la posibilidad de emplear la Transformada de Hough para la detección de los límites laterales de la pista en el vehículo autónomo DeepRacer. Esta decisión se fundamentó en la capacidad teórica del algoritmo para detectar líneas rectas incluso en imágenes con ruido o bordes fragmentados, lo cual parecía prometedor para entornos reales con iluminación variable o marcas parcialmente borradas

En este enfoque inicial, se utilizaba Canny para detectar bordes y posteriormente aplicar la Transformada de Hough con el objetivo de extraer los segmentos correspondientes a las líneas izquierda y derecha del circuito. A partir de estas líneas, se estimaba el centro de la pista y se derivaban las correcciones necesarias mediante un controlador Stanley.

Sin embargo, en las pruebas prácticas se evidenciaron ciertas limitaciones importantes que condicionaron su utilidad real. Si bien el algoritmo ofrecía resultados aceptables en tramos rectos o con líneas bien definidas, su rendimiento se degradaba significativamente en curvas suaves, zonas mal iluminadas o con marcas poco contrastadas. Estas limitaciones llevaron a reconsiderar el uso de Hough como base para la percepción visual del vehículo, tal y como se expone a continuación.

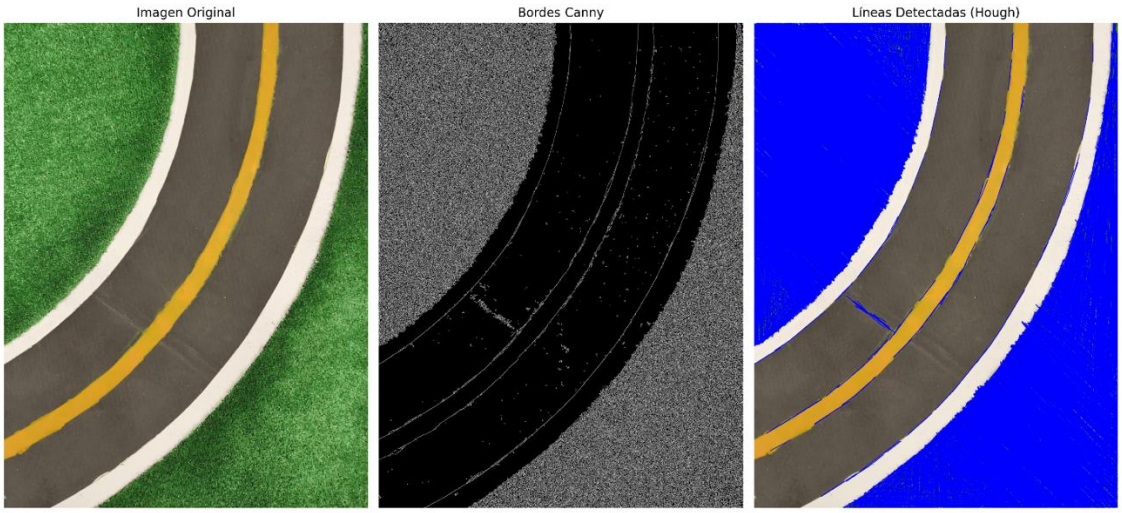


Ilustración 14 detección de líneas en la pista (foto hecha con el móvil)

5.7. Consideraciones finales

Aunque inicialmente se optó por utilizar la Transformada de Hough para la detección de líneas, su rendimiento en escenarios reales con curvas cerradas y líneas irregulares resultó ser insuficiente. A pesar de combinarla con preprocesamiento mediante el detector de bordes de Canny, la Transformada de Hough no ofrecía una segmentación fiable en curvas suaves ni una continuidad adecuada para el cálculo de trayectorias.

El principal problema radicaba en que Hough está diseñada para detectar líneas rectas, y aunque es posible adaptarla para curvas mediante variantes como la Transformada de Hough Generalizada o polinomial, estas opciones aumentan considerablemente la complejidad computacional y no se integran fácilmente en sistemas embebidos como DeepRacer. En la práctica, el resultado fue una detección inconsistente, especialmente en tramos curvos o cuando las líneas laterales eran interrumpidas o reflejaban luz.

Por ello, se decidió abandonar esta técnica en favor de un enfoque basado en segmentación por color (HSV) y análisis de contornos, que permite extraer puntos clave de la trayectoria con mayor estabilidad. Este método, combinado con controladores como Stanley, ha demostrado ser más robusto y preciso para el seguimiento de trayectorias en circuitos cerrados.

En la siguiente sección se presenta el diseño definitivo implementado en el vehículo, detallando cómo se integraron las nuevas técnicas de visión y control en el flujo completo de percepción y actuación del DeepRacer.

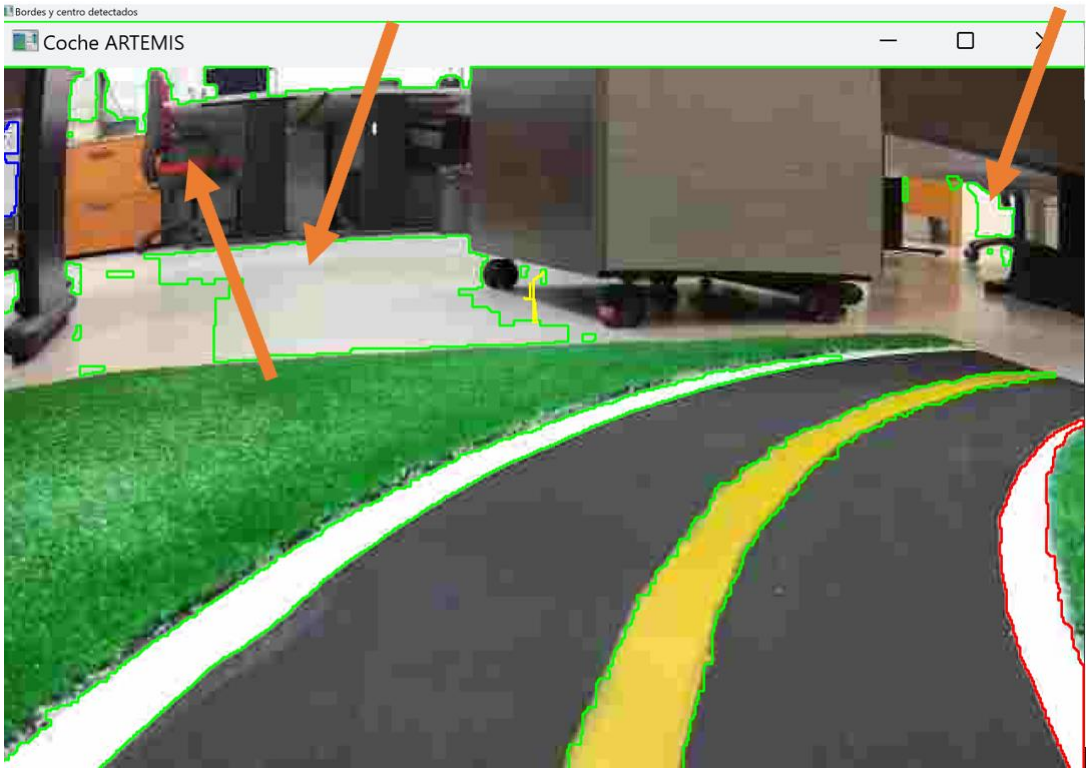


Ilustración 15 deteccion con hough (limitaciones)

5.8. Detección de Carriles mediante el Algoritmo de Ventana Deslizante

Una vez descartada la Transformada de Hough por sus limitaciones prácticas y tras explorar los beneficios del filtrado por bordes con el algoritmo de Canny, se optó por implementar una técnica adicional que mejorara la detección de carriles en condiciones reales: el algoritmo de ventana deslizante (*sliding window*) (20). Este método, ampliamente utilizado en la literatura sobre conducción autónoma, ha demostrado ser especialmente eficaz en contextos donde las líneas de la pista pueden estar parcialmente ocultas, fragmentadas o distorsionadas por la perspectiva (21).

El algoritmo de ventana deslizante se basa en el análisis vertical de una imagen binarizada, comúnmente obtenida tras una transformación de perspectiva (vista cenital) y un filtrado previo por color o bordes. La técnica comienza generando un histograma de intensidades en la mitad inferior de la imagen, lo que permite identificar las posiciones más probables de los carriles izquierdo y derecho.

A partir de estos picos iniciales, se despliegan múltiples ventanas rectangulares hacia arriba en la imagen. En cada nivel, el algoritmo ajusta la posición horizontal de las ventanas en función de la concentración de píxeles blancos, refinando progresivamente la localización de los carriles. Finalmente, se aplica un ajuste polinomial de segundo grado para modelar la trayectoria de cada carril de forma suave y continua.

A diferencia de la Transformada de Hough, que asume geometrías rectas, el algoritmo de ventana deslizante puede adaptarse fácilmente a curvas suaves, bifurcaciones e incluso tramos con marcas borrosas. Además, su lógica basada en la acumulación de píxeles ofrece una mayor tolerancia al ruido visual, siendo menos susceptible a falsos positivos causados por imperfecciones en el suelo o interferencias de iluminación.

En comparación con la segmentación por contornos, la ventana deslizante proporciona una estimación continua del centro del carril, lo que resulta especialmente útil para el cálculo del error lateral (cross track error) en el controlador Stanley. Esto se traduce en una conducción más estable, con menos oscilaciones y mejor anticipación en curvas pronunciadas.

En el presente proyecto, se integró el algoritmo de ventana deslizante tras una conversión a vista de pájaro (*perspective transform*) y la aplicación combinada de filtrado por color (HSV) y detección de bordes (Canny). Esta fusión de métodos permite generar una imagen binaria más robusta, sobre la cual se ejecuta el *sliding window* para localizar los carriles y calcular el centro del camino.

El punto medio entre los carriles detectados se emplea como referencia para estimar la desviación del vehículo respecto a la trayectoria ideal. Esta información alimenta al controlador Stanley, que ajusta tanto el ángulo de giro como la velocidad de avance.

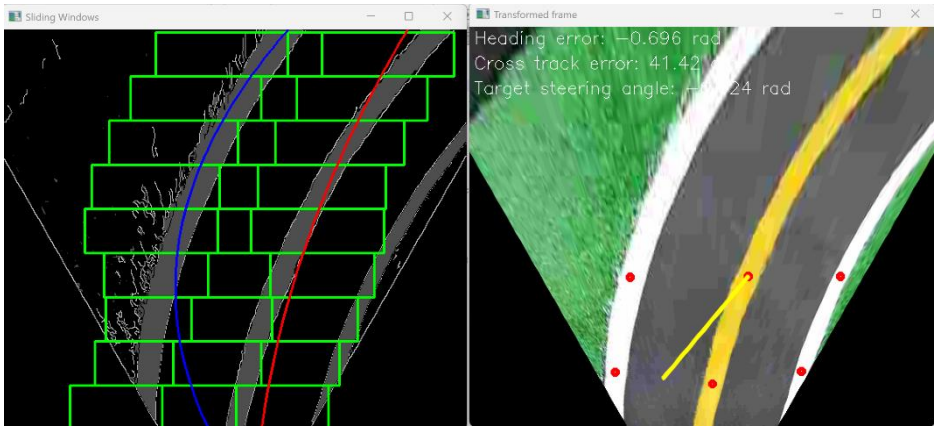


Ilustración 16 Sliding Window

5.9. Controlador Stanley: Seguimiento de Trayectorias

Para convertir las líneas detectadas en acciones de conducción, se utiliza el controlador Stanley (22), un método clásico en vehículos autónomos que ajusta la dirección en función de:

1. **Error de Orientación:** Diferencia entre el ángulo actual del vehículo y el de la trayectoria deseada.
2. **Error Lateral:** Distancia perpendicular entre el centro del vehículo y la línea de referencia.

La ley de control se expresa como: donde es el error de orientación, el error lateral, una constante de ganancia y la velocidad.

Ventajas:

- Suaviza las correcciones bruscas, evitando oscilaciones.
- Es robusto en curvas cerradas y rectas a alta velocidad.

5.9.1. Conclusión

La experiencia con DeepRacer muestra que, si bien los algoritmos clásicos como Hough ofrecen una buena base teórica, su rendimiento práctico puede verse limitado en entornos reales. La combinación de segmentación por color, análisis de contornos y control Stanley ha demostrado ser una solución más robusta y eficiente para la navegación autónoma.

Este trabajo evidencia que, con enfoques bien adaptados y tecnologías accesibles, es posible desarrollar sistemas de conducción autónoma estables y funcionales incluso sin recurrir a modelos complejos basados en aprendizaje con IA.

6. Integración de Algoritmos Clásicos en el Sistema de Percepción del DeepRacer

6.1. Introducción

Tras evaluar las limitaciones del enfoque inicial basado en la Transformada de Hough, se desarrolló un nuevo sistema de percepción visual fundamentado en técnicas más adaptadas a las condiciones reales de un circuito cerrado. Este rediseño se centró en aprovechar la segmentación por color y la extracción de contornos como base para el seguimiento de la pista, logrando una solución más robusta, estable y menos dependiente de parámetros sensibles al entorno.

Este código implementa el comportamiento autónomo de un coche en una clase llamada *artemis_autonomous_car*, que puede seguir una trayectoria usando visión artificial y sensores LIDAR. La clase está diseñada para procesar imágenes captadas por una cámara y datos de distancia obtenidos por el sensor LIDAR, con el fin de tomar decisiones de control tanto en dirección (giro del volante) como en aceleración.

Al inicio, cuando se crea el coche, se establecen los parámetros principales como el tamaño de la imagen capturada, una matriz para cambiar la perspectiva de la imagen, y los valores necesarios para controlar la velocidad y la dirección del vehículo. También se definen los colores que el coche reconocerá en la imagen para detectar las líneas del camino.

El vehículo cuenta con dos tipos de control. El primero es el control longitudinal, que se refiere al movimiento hacia adelante o atrás. Este control usa el sensor LIDAR para medir si hay obstáculos delante. Si el coche detecta un obstáculo demasiado cerca, se detiene automáticamente. Si no hay obstáculos, el coche ajusta su velocidad según la distancia que detecta. Para visualizar estos datos, el programa puede mostrar un mapa en pantalla, donde se representa la posición de los objetos detectados y el coche.

El segundo tipo de control es el lateral, que se encarga de decidir hacia dónde debe girar el coche para mantenerse en la trayectoria correcta. Cada imagen captada por la cámara pasa primero por una transformación de perspectiva para verla desde arriba, como si se tratara de una vista de pájaro. Luego se analizan los colores de la imagen para encontrar las líneas del camino, usando un rango de colores definido al principio. Se aplican mejoras al contraste y operaciones de filtrado para identificar mejor las líneas.

La imagen se divide en tres partes, que representan zonas cercanas, medias y lejanas. En cada zona se buscan contornos, es decir, formas que indican la presencia de líneas. Se calculan los centros de estos contornos y, dependiendo de su cantidad y posición, se eligen dos puntos que representan la trayectoria a seguir. Si el coche tiene una ruta predefinida con bifurcaciones (por ejemplo, girar a la izquierda o derecha), elige los puntos más adecuados para seguir esa ruta.

Con estos dos puntos, el programa calcula dos errores: el error de rumbo, que indica cuánto debe girar el coche, y el error de desviación lateral, que muestra cuánto se ha salido del centro del camino. Luego, se aplica un algoritmo llamado Stanley, que es un método común para que los vehículos autónomos mantengan su trayectoria de forma suave y estable. A partir de ese cálculo, se determina cuánto debe girar el coche. Este valor se ajusta con un poco de suavidad para evitar movimientos bruscos.

La velocidad también se ajusta en función del ángulo de giro. Cuanto más cerrado es el giro, más despacio debe ir el coche. Además, si hay un obstáculo adelante, la velocidad se reduce a cero.

Finalmente, si el usuario activa la opción de mostrar información, el programa dibuja los puntos detectados y la trayectoria elegida en la imagen, así como algunos datos útiles como el valor del giro, la velocidad o el estado del camino. Todo esto permite que el coche se desplace de forma autónoma, tomando decisiones inteligentes a partir de lo que ve y detecta.

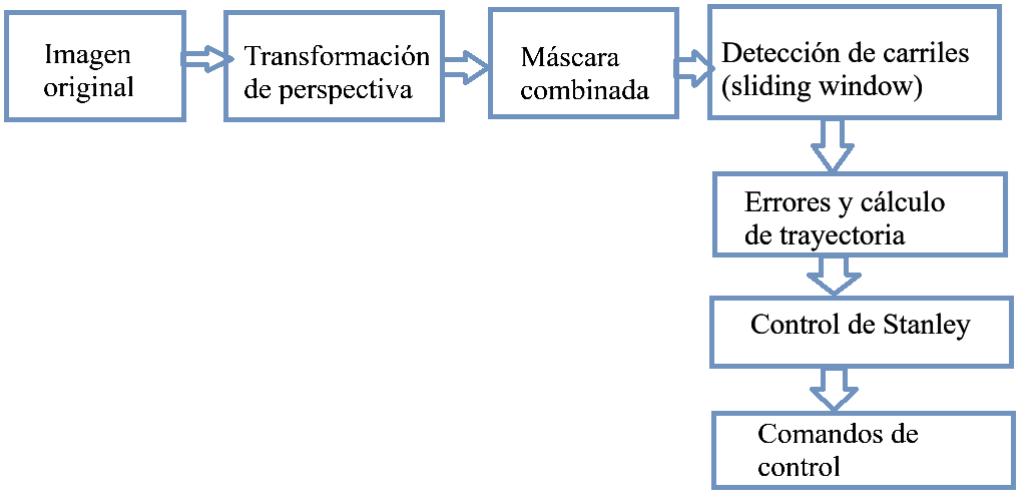


Ilustración 17 Flujo de percepción visual de las funciones

Función / Método	Descripción	Entrada	Salida
__init__	Inicializa parámetros operativos, matriz de perspectiva, constantes del controlador y umbrales de seguridad.	Parámetros de calibración y configuración de pista	Instancia lista para procesamiento
generate_combined_mask	Genera una máscara binaria combinando HSV, RGB y Lab.	Imagen (*frame*)	Imagen binaria mejorada
sliding_window_lane_detection	Detecta carriles mediante ventanas deslizantes sobre la imagen binaria.	Imagen binaria (binary_warped)	Puntos de carril, curvas ajustadas, imagen con anotaciones
set_stop	Permite detener el vehículo estableciendo una bandera de parada.	stop (bool)	Modifica el estado interno
set_battery_level	Actualiza el nivel de batería del sistema.	battery_level (float)	Actualiza una variable de estado
proceso_lidar	Procesa datos del sensor LIDAR para detectar	Lista de distancias LIDAR	Actualiza la variable de

	obstáculos y ajustar la velocidad.		control de velocidad
perspective_transformation	Convierte una imagen frontal en una vista cenital para facilitar la interpretación de la pista.	Imagen (*frame*)	Imagen transformada
calculate_center_contours	Detecta contornos y calcula los centros relevantes para estimar la trayectoria.	Imagen binaria, offset vertical	Número de contornos, centros de masa, ancho del contorno mayor
calculate_trajectory	Calcula los errores de orientación y desplazamiento lateral a partir de contornos detectados.	Imagen transformada	heading_error, cross_track_error, flags de trayectoria
proceso_fotograma	Orquesta todo el procesamiento visual para generar comandos de dirección y velocidad.	Imagen de cámara	steering_control, throttle_control, estado de trayectoria
rad2control	Convierte un ángulo en radianes a un valor normalizado de dirección.	Ángulo en radianes	Valor entre -1 y 1
calculo_stanley	Implementa el controlador Stanley para calcular el ángulo de giro óptimo.	Errores y velocidad	Ángulo de dirección en radianes

Tabla 6. Resumen de funciones de la clase artemis_autonomous_car

6.2. Adaptación del Sistema desde un Entorno de Pista Cerrada hacia un Entorno Controlado con Restricciones Reales

Durante la primera fase de implementación del nuevo algoritmo, el vehículo funcionaba en una pista cerrada sin bifurcaciones, donde el trazado era constante y sin grandes variaciones. Esto permitía utilizar un enfoque más simple de detección de carril, basado en la segmentación del color y la identificación de líneas con técnicas clásicas como "*sliding window*" o máscaras combinadas. Sin embargo, al avanzar hacia un entorno más complejo, con intersecciones y bifurcaciones, fue necesario modificar profundamente el algoritmo de conducción autónoma.

Uno de los principales cambios fue la introducción de una lógica para detectar intersecciones y tomar decisiones en función de una "ruta programada". Esto se refleja en la nueva versión de la clase `artemis_autonomous_car`, donde ahora se usa una lista `self.path` que define el comportamiento esperado en cada bifurcación (por ejemplo, izquierda, recto o derecha). Esta nueva lógica está basada en el código previo desarrollado por Ignacio Royuela (3), del que se han mantenido muchas ideas y estructuras. Sin

embargo, se han realizado ciertos cambios: se añadieron funciones para adaptar mejor la detección de la trayectoria a los cambios de iluminación, y también se ajustaron algunos parámetros visuales, como el punto de referencia en la imagen (`point1 = [self.img_width_center + 80, 430]`), para mejorar la visión desde arriba y detectar la pista con más precisión.

Además, se amplió el sistema de perspectiva de la cámara con respecto a la pista cerrada, utilizando una transformación más agresiva (*birdseye view*), que ya estaba presente en el código anterior, solo se aumentó la resolución, para obtener una vista más amplia del frente del coche. Esta vista permite al vehículo identificar más fácilmente posibles bifurcaciones o caminos alternativos, algo que era innecesario en la pista cerrada original.

También se han introducido importantes mejoras en la detección de trayectorias respecto a la implementación usada en la pista cerrada. En ese entorno más simple, el vehículo se limitaba a identificar contornos en zonas específicas de la imagen y tomaba decisiones básicas basadas en el punto medio detectado.

En el nuevo escenario con bifurcaciones, en cambio, la estrategia se ha sofisticado. Se combinan tres regiones de interés (cercana, media y lejana) para extraer múltiples puntos de referencia que permiten calcular con mayor precisión la dirección de la trayectoria. Esta estructura permite anticiparse a curvas e intersecciones, algo innecesario en un circuito cerrado donde no hay alternativas de recorrido.

Además, el nuevo sistema evalúa el número y posición de los contornos en cada región para decidir entre distintas trayectorias posibles (izquierda, centro o derecha), en función de una ruta programada. Cuando el vehículo detecta bifurcaciones, se activa un sistema de seguimiento temporal que permite asegurar que el desvío se ha realizado correctamente antes de continuar con el próximo objetivo.

Otra diferencia fundamental con la pista cerrada es la incorporación de mecanismos para el cambio de carril, que solo se ejecutan si se identifican con claridad trayectorias en la zona más lejana de la imagen. Esto proporciona una mayor robustez en maniobras más complejas.

6.2.1. Justificación de los Cambios

Los cambios introducidos en el sistema tienen como finalidad adaptar el comportamiento del vehículo autónomo a entornos más representativos de condiciones reales, donde es necesario tomar decisiones dinámicas de navegación, como optar entre distintas trayectorias en una intersección. En un circuito cerrado, este tipo de situaciones no se presentan, por lo que el algoritmo original no contemplaba mecanismos de toma de decisiones en tiempo real ni estructuras de planificación de ruta.

La incorporación de rutas programadas permite definir un comportamiento deseado ante bifurcaciones, lo que habilita la planificación de misiones más complejas y dirigidas. Este enfoque incrementa el grado de autonomía del vehículo y mejora su escalabilidad, haciéndolo aplicable a escenarios reales o simulados de mayor dificultad.

Por otro lado, se han mejorado las capacidades de percepción del sistema frente a condiciones adversas, como iluminación variable u obstrucciones parciales. En lugar de depender exclusivamente de contornos claramente visibles, el sistema ahora es capaz de estimar una trayectoria probable a partir de información parcial, lo que contribuye a una mayor robustez operativa y asegura la continuidad del movimiento en situaciones donde antes se habría producido un fallo.

En la Tabla 7 se recogen las funciones para la pista real con intersecciones, junto con una pequeña descripción. Se pueden ver las funciones que se han mantenido y las que han cambiado con respecto a la pista cerrada.

Función / Método	Descripción
__init__()	Inicializa los parámetros del sistema: imagen, control, color, trayectoria, memoria, etc.
set_stop(stop)	Permite detener o reanudar el vehículo manualmente.
perspective_transformation(img)	Aplica una transformación de perspectiva para ver la imagen desde arriba (vista de pájaro).
proceso_fotograma(img)	Método principal que gestiona la percepción, el cálculo de errores y la generación de comandos.
calculate_trajectory(img)	Detecta la trayectoria mediante contornos y calcula errores de orientación y posición lateral.
calculate_center_contours(img, offset)	Detecta los contornos más grandes y devuelve los puntos centrales de interés.
rad2control(angle)	Convierte un ángulo en radianes en un valor de dirección normalizado entre -1.0 y 1.0.
calculo_stanley(he, cte, vel)	Aplica el controlador Stanley para corregir la dirección del vehículo en función de los errores.
adaptar_rango_hsv(imgHSV)	Ajusta automáticamente los rangos de color HSV según el brillo de la imagen para detectar mejor la línea.
ajustar_gamma(img, gamma)	Aplica corrección gamma para mejorar el contraste y visibilidad de la imagen.

Tabla 7 funciones del código de conducción autónoma por un circuito real

En entornos reales, donde las condiciones de iluminación pueden variar de forma significativa (por ejemplo, en presencia de sombras, sol intenso o zonas oscuras), es fundamental aplicar técnicas robustas de procesamiento de imagen para garantizar un seguimiento fiable del carril.

Como se menciona antes, una de las primeras técnicas que usamos para la correcta visualización y detección del carril es la transformación de perspectiva con `warpPerspective`, que genera una vista cenital de lo que capta la cámara, lo que se conoce como “vista de pájaro”.

Tras aplicar la transformación de perspectiva, se utilizan filtros de segmentación en el espacio de color HSV, que presenta una mayor estabilidad frente a variaciones de iluminación que el modelo RGB. Además, se incorpora el algoritmo CLAHE (Contrast Limited Adaptive Histogram Equalization), el cual mejora el contraste local en regiones oscuras de la imagen, incrementando la visibilidad de los detalles.

Aunque no se realizaron pruebas reales en exteriores, durante el desarrollo del proyecto anterior (3) se detectaron problemas al simular la conducción fuera del laboratorio. En particular, las sombras, la luz intensa del sol o los cambios de iluminación afectaban negativamente a la segmentación del color azul de la pista, provocando errores en la detección de la trayectoria.

Por esta razón, se han implementado dos mejoras para hacer el sistema más robusto frente a estas variaciones de luz.

La primera fue la implementación de una función de corrección gamma (23). Esta técnica ajusta el brillo de la imagen de forma no lineal, usando una fórmula matemática que mejora la visibilidad en zonas oscuras sin saturar las más brillantes. Gracias a esto, la imagen de entrada se vuelve más uniforme, permitiendo que los filtros de color trabajen mejor.

$$\text{salida} = \left(\frac{\text{entrada}}{255}\right)^{1/\gamma} \times 255$$

Ilustración 18 fórmula gamma

La segunda mejora fue un ajuste dinámico del rango de color HSV. En lugar de usar valores fijos, ahora el sistema calcula el brillo medio de la imagen y adapta automáticamente los límites del color azul que queremos detectar. Así, el mismo algoritmo puede funcionar tanto de día como al atardecer, sin necesidad de recalibrar manualmente.

En nuestro proyecto también usamos un sensor LIDAR, que mide la distancia a los objetos usando rayos láser. Es muy útil para detectar obstáculos como personas, paredes o coches. Sin embargo, el LIDAR no nos ayuda a detectar la pista, por eso usamos principalmente la cámara para seguir el recorrido.

Una vez transformada la imagen a vista de pájaro, se detectan las líneas del carril utilizando segmentación en el espacio de color HSV. La imagen se divide en tres zonas horizontales (lejana, media y cercana) donde se buscan los centros de los contornos.

A partir de estos puntos, se calculan dos errores fundamentales para el control de dirección:

- El **error de rumbo (heading error)** es el ángulo entre dos puntos de referencia y representa la orientación del vehículo respecto a la trayectoria deseada.
- El **error lateral (cross track error)** mide cuánto se desvía lateralmente el vehículo respecto al centro del carril.

Ambos errores se introducen en el algoritmo de Stanley para calcular el ángulo de dirección necesario. Este ángulo es convertido a una señal de control y aplicado al motor de dirección del vehículo, manteniendo así su trayectoria dentro del carril incluso en zonas con bifurcaciones o curvas cerradas. En la Ilustración 19 se puede ver gráficamente.

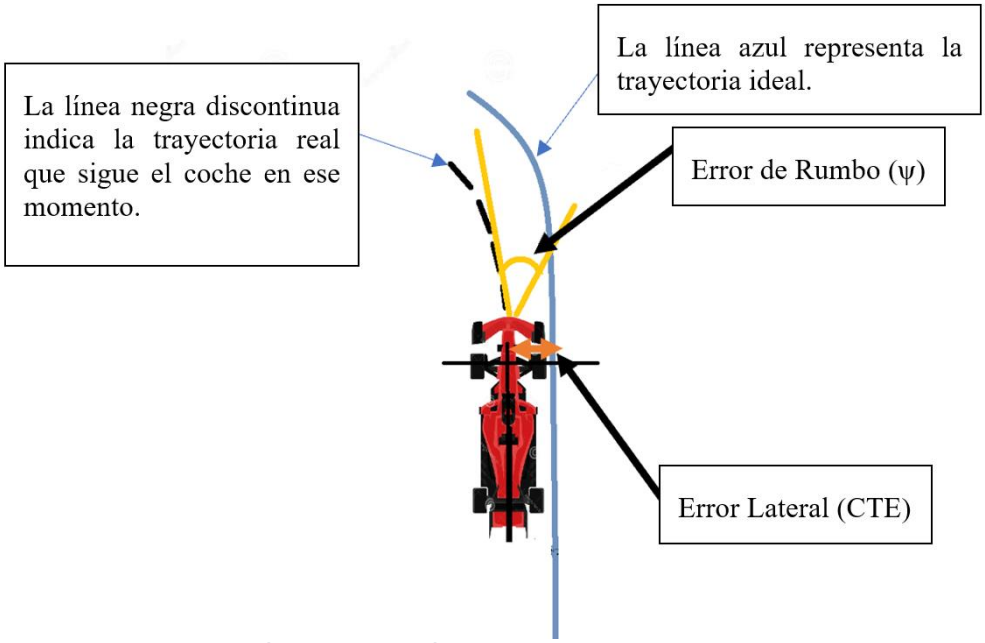


Ilustración 19 Visualización de errores utilizados en el control Stanley

Para calcular el ángulo de dirección necesario que mantenga el vehículo dentro de la trayectoria, el sistema utiliza dos puntos detectados en la imagen: uno cercano al vehículo (punto 1) y otro más alejado (punto 2). A partir de estos dos puntos, se extraen dos errores fundamentales:

- **Error de rumbo (ψ)**: representa el ángulo entre la dirección actual del coche y la línea imaginaria que conecta los dos puntos detectados. Si el vehículo apunta en una dirección distinta a la del camino, este ángulo será diferente de cero.
- **Error lateral (CTE)**: es la distancia perpendicular entre el centro del coche y el centro de la trayectoria. Permite saber cuánto se está desviando lateralmente el vehículo.

Fórmulas con explicación (22)

$\psi = \arctan((x_1 - x_2) / (y_1 - y_2))$
Donde (x_1, y_1) es el punto cercano, y (x_2, y_2) es el punto lejano. El resultado es el heading error en radianes.

$x = x_2 - x\_c + (x_1 - x_2) * k\_cte$
Esta fórmula prolonga virtualmente el vector hacia adelante para estabilizar la trayectoria. x_c es el centro de la imagen, y k_cte una constante ajustable.

cte = -x / pix_por_metro [m]
Convierte el error en píxeles a metros reales. El signo negativo invierte la dirección para respetar el sistema de coordenadas.

$\delta = \psi + \arctan((k\_stanley * cte) / v)$
Esta es la fórmula principal del algoritmo Stanley. Combina los dos errores con la velocidad del coche, v, y ajusta el ángulo de dirección necesario.

Después de detectar las líneas del carril, el sistema analiza los contornos en la imagen y selecciona dos puntos importantes: uno cercano al coche y otro más lejano. Con estos puntos, calcula si el vehículo va en la dirección correcta (error de rumbo) y cuánto se ha desviado del centro del carril (error lateral).

A partir de estos errores, se usa el algoritmo Stanley para calcular el ángulo de giro necesario. Este ángulo indica hacia dónde debe girar el coche para corregir su posición y mantenerse dentro del carril.

Por último, esta información se convierte en una orden de control que se envía al sistema de dirección del coche. Así, el vehículo ajusta su movimiento en tiempo real y puede seguir la trayectoria marcada, incluso si hay curvas, bifurcaciones o cambios de luz.

En la Ilustración 20 se puede ver lo que muestra el coche mientras conduce por la pista.

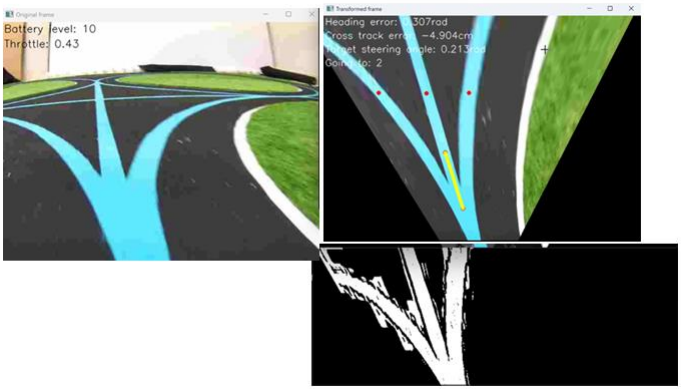


Ilustración 20 Visual de calculate_trajectory(img)

Además de detectar correctamente la pista, el coche autónomo necesita varios parámetros bien ajustados para funcionar de forma estable y segura. Uno de ellos es `k_cross_track_error`, que controla cuánto corrige el coche su posición lateral en el carril. Otro importante es `pixels_in_meter`, que indica cuántos píxeles equivalen a un metro en la imagen. Si estos valores no están bien calibrados, el coche puede girar demasiado o muy poco.

También hay parámetros que se deben cambiar según el circuito. Por ejemplo, el `path` es una lista que indica si el coche debe ir recto, girar a la izquierda o a la derecha en cada cruce. Si el recorrido cambia, esta lista debe actualizarse. Otro parámetro es `minimum_contour_area`, que define el tamaño mínimo de una línea para que se considere parte de la trayectoria. Si el valor es muy bajo, puede detectar ruido; si es muy alto, puede ignorar líneas válidas.

Existen más valores que ayudan al coche a tomar decisiones. Por ejemplo, `frames_counter_switch_fork` y `switch_fork` sirven para controlar cómo se comporta el coche al pasar por cruces. También hay ajustes como `max_larger_width_contour` o `contour_cut_x_position_1` y `contour_cut_x_position_2`, que ayudan a separar trayectorias cuando hay mucha luz o ruido. Finalmente, el parámetro `k_Stanley` ajusta la fuerza con la que el coche corrige su dirección, y `steering_calibration_param` sirve para corregir pequeños desvíos de la dirección.

En resumen, para que el coche autónomo funcione correctamente, no solo es necesario tener una buena visión por cámara, sino también ajustar con cuidado todos estos

parámetros según el entorno, la pista y las condiciones de luz. Esto permite adaptar el sistema a diferentes circuitos y garantizar una conducción más estable, segura y fluida.

Todos los archivos del proyecto, incluyendo el código, configuraciones y documentación adicional, se encuentran disponibles en el repositorio de GitHub.

Artemis/paquete de ROS2 at main · gcoUVa/Artemis

7.Conclusión Final y Líneas Futuras

A lo largo de este Trabajo de Fin de Grado se han cumplido los objetivos planteados inicialmente:

- Se ha migrado el sistema de control del vehículo autónomo Amazon Deepracer desde ROS1 a ROS2, manteniendo la funcionalidad de los nodos personalizados existentes.
- Se han desarrollados nuevos módulos para la conducción autónoma, aprovechando las ventajas de ROS2 y su arquitectura.
- Se ha validado el funcionamiento del sistema en entornos de pruebas, comprobando la capacidad del vehículo para moverse de forma autónoma y responder a instrucciones externas, además de ser capaz de detenerse al detectar objetos a través del LIDAR.
- Estos resultados permiten disponer de una arquitectura más moderna y escalable que podría ser extendida con nuevas funcionalidades en el futuro.

Además de cumplir con estos objetivos, durante el desarrollo del proyecto se incorporaron técnicas importantes. Se añadieron mecanismos para mejorar la detección del carril frente a variaciones de iluminación, como la corrección gamma y la adaptación automática de los rangos HSV.

También se probaron nuevas técnicas para la detección de líneas del carril, como Canny y sliding window, y se comprobó que, aunque son métodos eficaces en condiciones simples, su fiabilidad disminuye cuando hay intersecciones. Esto se debe a que en esos casos los bordes pueden mezclarse o generar trayectorias confusas. Por esta razón, se mantuvo la lógica del proyecto anterior y se ajustaron algunos parámetros clave para asegurar un funcionamiento más estable del sistema.

Como líneas de futuro, se proponen las siguientes:

- Integrar un sistema de conducción basado en aprendizaje automático, que permita al vehículo tomar decisiones a partir de datos visuales y sensores.
- También se podría ampliar el entorno de prueba a escenarios más complejos, como señales de tráfico para que el coche pare o reduzca la velocidad.

En conjunto, el proyecto ha permitido consolidar conocimientos sobre robótica, comunicaciones distribuidas y desarrollo de *software* sobre ROS2, sentado una base para futuros desarrollos.

Artemis/paquete de ROS2 at main · gcoUVa/Artemis

8. Referencias

1. *ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ROS 2.* **Bedard, Christophe, Lutkebohle , Ingo y Dagena, Michel.** s.l. : IEEE Robotics and Automation Letters, 2022, IEEE Robotics and Automation Letters, págs. 6511–6518.

2. *Robot Operating System 2: Design, architecture, and uses in the wild.* **Macenski, Steven, y otros.** 66, s.l. : American Association for the Advancement of Science, 18 de 07 de 2022, Science robotics, Vol. 7, pág. eabm6074.

3. **Royuela, Ignacio.** *Diseño e implementación de un testbed de Edge computing para el soporte de vehículos conectados.* Universidad de Valladolid. 2022. Trabajo Fin de Máster.

4. **Viloria, Ivan.** *Integración de sistemas de posicionamiento indoor para un testbed de Edge Computing para el soporte de vehículos conectados.* s.l. : UNIVERSIDAD DE VALLADOLID, 2022. Trabajo Fin de Master.

5. *ROS: an open-source Robot Operating System.* **Quigley , Morgan, y otros.** 3.2, s.l. : Computer Science Department, Stanford University, Stanford, CA, Willow Garage, University of Southern California, 2009, May, Kobe, Vol. 3, pág. 5.

6. *An Analytical Latency Model of the Data Distribution Service in ROS 2.* **Park, Hyung-Seok, y otros.** 2025, IEEE, págs. 1--10.

7. **Open Robotics.** ROS Documentation. *About Quality of Service settings — ROS 2 Documentation: Foxy .* [En línea] 18 de 07 de 2025. https://docs.ros.org/en/foxy/Concepts/About-Quality-of-Service-Settings.html.

8. **Read the Docs.** readthedocs.io. *ros1_bridge — ros1_bridge latest documentation.* [En línea] 18 de 07 de 2025. https://ros1-bridge.readthedocs.io/en/latest/index.html.

9. **Liu, P.X, y otros.** *An UDP-based protocol for Internet robots.* s.l. : Proceedings of the 4th World Congress on Intelligent Control and Automation (Cat. No.02EX527), 2002. págs. 59-65. Vol. vol.1.

10. **Light, R. A.** Eclipse Mosquitto. *Eclipse Mosquitto.* [En línea] 18 de 07 de 2025. https://mosquitto.org/.

11. **JohnStechschulte.** ROS Wiki. *cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython.* [En línea] 18 de 07 de 2025. https://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython.

12. **Amazon AWS.** AWS Documentation. *Actualice y restaure su DeepRacer dispositivo de AWS.* [En línea] 18 de 07 de 2025. https://docs.aws.amazon.com/es_es/deepracer/latest/developerguide/deepracer-ubuntu-update.html.

13. —. GitHub. *Getting started with AWS DeepRacer OpenSource.* [En línea] 18 de 07 de 2025. https://github.com/aws-deepracer/aws-deepracer-launcher/blob/main/getting-started.md.

14. **ROS Documentation.** ROS Documentation. *Creating a package — ROS 2 Documentation: Foxy documentation.* [En línea] 18 de 07 de 2025.

https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html.

15. **Ahora, canal Youtube: Creatividad.** Youtube . *Instalar Mosquitto 2.0 en Windows 10.* [En línea] 18 de 07 de 2025. https://www.youtube.com/watch?v=cL6n3wafWEQ.

16. *A computational approach to edge detection.* **Canny, John.** s.l. : IEEE Transactions on pattern analysis and machine intelligence, 1986, IEEE Transactions on Pattern Analysis and Machine Intelligence, págs. 679-698.

17. *An improved Canny algorithm with adaptive threshold selection.* **Wang, Yupeng y Li, Jiangyun.** s.l. : EDP Sciences, 2015, EDP Sciences, Vol. 22, pág. 01017.

18. *Sistema de visión por computadora para la detección de objetos esféricos a través de la transformada de Hough.* **Rojas, Teddy V , Sanz, Wilmer y Arteaga, Francisco.** 1, s.l. : Universidad de Carabobo, 2008, revista ingeniería UC, Vol. 15, págs. 77--87.

19. **TutorialsPoint.** TutorialsPoint. *Probabilistic Hough Transform in Scikit-Image.* [En línea] 18 de 07 de 2025. https://www.tutorialspoint.com/scikit-image/scikit-image-probabilistic-hough-transform.htm.

20. **NamidM y NurNils.** GitHub. *Find Line Detection (Image Processing).* [En línea] 18 de 07 de 2025. https://github.com/NurNils/opencv-lane-detection/blob/master/README.md.

21. *Adaptive sliding-window strategy for vehicle detection in highway environments.* **Noh, SeungJong, Shim, Daeyoung y Jeon, Moongu.** 2, s.l. : IEEE Transactions on Intelligent Transportation Systems, 2015, IEEE Transactions on Intelligent Transportation Systems, Vol. 17, págs. 323--335.

22. *Stanley: The robot that won the darpa grand challenge.* **Thrun, Sebastian, y otros.** 9, s.l. : Springer, 2007, Journal of field Robotics, Vol. 23, págs. 661--692.

23. **Rosebrock, Adrian.** PyImageSearch. *OpenCV Gamma Correction - PyImageSearch.* [En línea] 18 de 07 de 2025. https://pyimagesearch.com/2015/10/05/opencv-gamma-correction/.

24. *Control PD difuso de trayectoria para modelo de vehículo mediante la detección de líneas de carril utilizando la transformada de Hough en Matlab.* **Mayhua , Álvarez y André , Mijaíl.** s.l. : Universidad Católica de Santa María, 2022, Universidad Católica de Santa María.

9. Anexos

Codigo Admin_comunication_node

```
import configparser
import io
import signal
import time
import socket
import sys
import rclpy
from rclpy.node import Node
import paho.mqtt.client as mqtt_client
import pickle
import struct
import cv2
from . import bmi160
from threading import Lock
from multiprocessing import Process, Queue
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import LaserScan
from deepracer_interfaces_pkg.msg import CameraMsg
from deepracer_interfaces_pkg.msg import ServoCtrlMsg
from deepracer_interfaces_pkg.srv import SetLedCtrlSrv
from deepracer_interfaces_pkg.srv import SetCalibrationSrv
from deepracer_interfaces_pkg.srv import BatteryLevelSrv
from deepracer_interfaces_pkg.srv import ActiveStateSrv
from deepracer_interfaces_pkg.srv import VideoStateSrv
from std_srvs.srv import SetBool
import subprocess

class Net_artemis:
    def __init__(self):
        pass
    @staticmethod
    def get_SSID_actual():
        return
subprocess.check_output(["iw","dev","mlan0","link"]).split('\n\t')[
1][6:]
    @staticmethod
    def get_dbm_signal_actual():
        return
int(subprocess.check_output(["iw","dev","mlan0","link"]).split('\n\
t')[5][8:].rstrip(' dBm'))
    @staticmethod
    def scan():
        return
subprocess.check_output(["sudo","iw","dev","mlan0","scan"])
    @staticmethod
    def get_public_IP():
        try:
            public_IP=subprocess.check_output(["upnpc","-
s"]).split("ExternalIPAddress = ")[1].split('\n')[0]
            type_conection="4G/5G"
            if public_IP == "10.0.103.56":
```

```
public_IP=str(subprocess.check_output(["ip","addr"]).split(b'mlan')
[1].split(b'inet')[1].split()[0]).split("'")[0]
                type_conection="Wifi"
        except:

public_IP=str(subprocess.check_output(["ip","addr"]).split(b'mlan')
[1].split(b'inet')[1].split()[0]).split("'")[0]
            type_conection="Wifi"
        return (type_conection,public_IP)

# Leer archivo de configuración
vehicle_config = configparser.ConfigParser(allow_no_value=True)
with
open("/home/deepracer/SoftARTEMIS/soft_artemis/soft_artemis/vehicle
.conf") as config_file:
    config_data = config_file.read()
vehicle_config.read_file(io.StringIO(config_data))

vehicle_ID = vehicle_config.get("mqtt", "vehicle_ID")
mqtt_server_ip = vehicle_config.get("mqtt", "mqtt_server_ip")
mqtt_server_port = vehicle_config.getint("mqtt",
"mqtt_server_port")
cloud_server_ip = vehicle_config.get("cloud_autonomous_driving",
"cloud_server_ip")
cloud_server_port =
vehicle_config.getint("cloud_autonomous_driving",
"cloud_server_port")
max_calibration = vehicle_config.getint("steering_calibration",
"max")
mid_calibration = vehicle_config.getint("steering_calibration",
"mid")
min_calibration = vehicle_config.getint("steering_calibration",
"min")
stream_server_address = (cloud_server_ip, int(cloud_server_port))

class AdminCommunicationsNode(Node):
    def __init__(self):
        super().__init__('admin_communications_node')
        # Variables de instancia
        self.i = 0
        self.type_conection = "Unknown"
        self.sending_data = 0
        self.lidar_subscription = None
        self.video_subscription = None
        self.sock = None
        self.periodic_process_launched = 0
        self.MQTT_periodic_process = None
        self.q_MQTT_periodic_sender = Queue()
        self.status = 1
        self.bridge = CvBridge()
        self.conectado = 0
        self.lock = Lock()
        self.vehicle_config = vehicle_config  # Referencia al
config global

        # Servicios
```

```
        self.led_service = self.create_client(SetLedCtrlSrv,
'/servo_pkg/set_led_state')
        self.calibration_service =
self.create_client(SetCalibrationSrv, '/servo_pkg/set_calibration')
        self.battery_level_service =
self.create_client(BatteryLevelSrv, '/i2c_pkg/battery_level')
        self.enable_local_autonomous_control =
self.create_client(SetBool, '/enable_local_autonomous_control')
        self.enable_cloud_control = self.create_client(SetBool,
'/enable_cloud_control')
        self.enable_voice_control = self.create_client(SetBool,
'/enable_voice_control')
        self.inicioimagenes = self.create_client(VideoStateSrv,
'/camera_pkg/media_state')

        # Iniciar servicio de camara
        self.llamar_servicio_video()

        # Publicadores
        self.pub_manual_drive = self.create_publisher(ServoCtrlMsg,
'/ctrl_pkg/servo_msg', 10)

        # Configurar manejadores de señales
        signal.signal(signal.SIGTERM, self.signal_handler)
        signal.signal(signal.SIGINT, self.signal_handler)

        # Inicializar LEDs
        self.led_state(red=10000000, green=10000000, blue=10000000)
        time.sleep(0.8)

        # Calibración inicial
        self.get_logger().info(f"Calibración:
Max={max_calibration}, Mid={mid_calibration},
Min={min_calibration}")
        time.sleep(0.3)

        # Prueba de servo
        self.pub_manual_drive.publish(ServoCtrlMsg(angle=1.0,
throttle=0.0))
        time.sleep(0.3)
        self.pub_manual_drive.publish(ServoCtrlMsg(angle=-1.0,
throttle=0.0))
        time.sleep(0.3)
        self.pub_manual_drive.publish(ServoCtrlMsg(angle=0.0,
throttle=0.0))
        time.sleep(0.3)

        # Conectar a MQTT
        self.client_MQTT = mqtt_client.Client()
        self.client_MQTT.on_connect = self.on_connect
        self.client_MQTT.on_message = self.on_message
        self.client_MQTT.on_disconnect = self.on_disconnect
        self.connect_mqtt()

    def signal_handler(self, _signo, _stack_frame):
        self.led_state(red=0, green=0, blue=0)
        sys.exit(0)
```

```
    def MQTT_periodic_sender(self):
        print("Proceso lanzado")
        client_MQTT = mqtt_client.Client()
        client_MQTT.connect(mqtt_server_ip, mqtt_server_port, 5)
        while true:
            print("ENVIO PERIODICO")
            acc = bmi160.read_accel()
            client_MQTT.publish(f"{vehicle_ID}/IMU/x", acc[0])
            client_MQTT.publish(f"{vehicle_ID}/IMU/y", acc[1])
            client_MQTT.publish(f"{vehicle_ID}/IMU/z", acc[2])
            if self.type_conection == "Wifi":
                client_MQTT.publish(f"{vehicle_ID}/WIFI_dBm",
Net_artemis.get_dbm_signal_actual())
            #result=self.battery_level()
            #client_MQTT.publish(f"{vehicle_ID}/battery_level",
float(result.voltage))
            time.sleep(0.5)
            if not self.q_MQTT_periodic_sender.empty():
                if self.q_MQTT_periodic_sender.get() == 0:
                    client_MQTT.disconnect()
                    break

    def send_data(self, tipo, serialized_data):
        with self.lock:
            self.sock.sendall(struct.pack('c', tipo) +
struct.pack('>I', len(serialized_data)) + serialized_data)

    def laser_data_stream(self, msg):
        serialized_data = pickle.dumps(msg.ranges[:])
        self.send_data(b'L', serialized_data)

    def camera_data_stream(self, msg):
        if self.i == 0:
            img = self.bridge.imgmsg_to_cv2(msg.images[0], "bgr8")
            endoded, img = cv2.imencode('.jpg', img,
[int(cv2.IMWRITE_JPEG_QUALITY), 30])
            serialized_data = pickle.dumps(img)
            time2 = time.time()
            self.send_data(b'I', serialized_data)
            print(time.time() - time2)
            self.i = 0
        else:
            self.i += 1

    def on_connect(self, client, userdata, flags, rc):
        if rc == 0:
            self.get_logger().info("Conectado")
            self.led_state(red=10000000, green=10000000, blue=0)
            client.subscribe(f"{vehicle_ID}/command")
            client.publish(f"{vehicle_ID}", f"Vehicle {vehicle_ID}
connected")

            (self.type_conection, public_IP) =
Net_artemis.get_public_IP()
            client.publish(f"{vehicle_ID}/type_of_conection",
self.type_conection)
            client.publish(f"{vehicle_ID}/public_ip", public_IP)
```

```
client.publish(f"{vehicle_ID}/steering_calibration/max",
max_calibration)

client.publish(f"{vehicle_ID}/steering_calibration/mid",
mid_calibration)

client.publish(f"{vehicle_ID}/steering_calibration/min",
min_calibration)
            client.publish(f"{vehicle_ID}/mqtt_server_ip",
mqtt_server_ip)
            client.publish(f"{vehicle_ID}/mqtt_server_port",
mqtt_server_port)
            client.publish(f"{vehicle_ID}/cloud_server_ip",
cloud_server_ip)
            client.publish(f"{vehicle_ID}/cloud_server_port",
cloud_server_port)
        else:
            self.get_logger().error("No conectado")

    def on_message(self, client, userdata, msg):
        self.get_logger().info(f"Mensaje recibido: {msg.topic} -
{msg.payload}")
        if msg.payload == b"AM-Local":
            self.get_logger().info("Recibido mensaje AM-Local")
            self.call_enable_cloud_control(False)
            self.call_enable_voice_control(False)
            self.call_enable_local_autonomous_control(True)
            self.led_state(red=0, green=10000000, blue=0)
            self.status = 2

        elif msg.payload == b"AM-Cloud":
            self.get_logger().info("Recibido mensaje AM-Cloud")
            self.call_enable_local_autonomous_control(False)
            self.call_enable_voice_control(False)
            self.call_enable_cloud_control(True)
            self.led_state(red=0, green=0, blue=10000000)
            self.status = 3

        elif msg.payload == b"AM-Voz":
            self.get_logger().info("Recibido mensaje AM-Voz")
            self.call_enable_local_autonomous_control(False)
            self.call_enable_cloud_control(False)
            self.call_enable_voice_control(True)
            self.led_state(red=0, green=10000000, blue=10000000)
            self.status = 4

        elif msg.payload == b"AM-Off":
            self.get_logger().info("Recibido mensaje AM-Off")
            self.call_enable_cloud_control(False)
            self.call_enable_voice_control(False)
            self.call_enable_local_autonomous_control(False)
            time.sleep(0.1)
            self.pub_manual_drive.publish(ServoCtrlMsg(angle=0.0,
throttle=0.0))
            self.led_state(red=10000000, green=10000000, blue=0)
            self.status = 1
```

```
        elif msg.payload == b"Get-Data":
            self.get_logger().info("Recibido mensaje Get-Data")
            bmi160.enable_accel()
            self.client_MQTT.publish('signal', "0")  # Corregido
client -> self.client_MQTT
            if self.periodic_process_launched == 0:
                self.MQTT_periodic_process =
Process(target=self.MQTT_periodic_sender,
name="MQTT_periodic_sender")
                self.MQTT_periodic_process.start()
                self.periodic_process_launched = 1

        elif msg.payload == b"Stop-Data":
            self.get_logger().info("Recibido mensaje Stop-Data")
            if self.periodic_process_launched == 1:
                self.q_MQTT_periodic_sender.put(0)
                self.periodic_process_launched = 0

        elif msg.payload[:9] == b"Calibrate":
            try:
                self.get_logger().info("Recibido mensaje
Calibrate")
                calibration_data = list(map(int,
msg.payload.decode().split()[1:4]))
                self.call_calibration(calibration_data[0],
calibration_data[1], calibration_data[2], 1)
                time.sleep(0.1)

self.pub_manual_drive.publish(ServoCtrlMsg(angle=0.0,
throttle=0.0))
                self.led_state(red=0, green=10000000,
blue=10000000)

                # Guardar configuración
                self.vehicle_config.set('steering_calibration',
'max', str(calibration_data[0]))
                self.vehicle_config.set('steering_calibration',
'mid', str(calibration_data[1]))
                self.vehicle_config.set('steering_calibration',
'min', str(calibration_data[2]))

                with
open("/home/deepracer/SoftARTEMIS/soft_artemis/soft_artemis/vehicle
.conf", 'w') as config_file:
                    self.vehicle_config.write(config_file)


self.client_MQTT.publish(f"{vehicle_ID}/steering_calibration/max",
calibration_data[0])

self.client_MQTT.publish(f"{vehicle_ID}/steering_calibration/mid",
calibration_data[1])

self.client_MQTT.publish(f"{vehicle_ID}/steering_calibration/min",
calibration_data[2])
                time.sleep(0.5)

            except Exception as e:
```

```
                self.get_logger().error(f"Error en calibración:
{e}")
                self.led_state(red=10000000, green=0, blue=0)
                time.sleep(0.2)
                self.led_state(red=0, green=0, blue=0)
                time.sleep(0.2)
                self.led_state(red=10000000, green=0, blue=0)
                time.sleep(0.2)
                self.led_state(red=0, green=0, blue=0)
                time.sleep(0.5)

            # Restaurar el estado de los LEDs según el modo actual
            if self.status == 1:
                self.led_state(red=10000000, green=10000000,
blue=0)
            elif self.status == 2:
                self.led_state(red=0, green=10000000, blue=0)
            elif self.status == 3:
                self.led_state(red=0, green=0, blue=10000000)

        elif msg.payload == b"get-data" and self.sending_data == 0:
            # Inicialización del socket
            self.sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
            self.sock.connect(stream_server_address)
            self.sending_data = 1

            # Creación de suscripciones en ROS2
            self.lidar_subscription = self.create_subscription(
                LaserScan,
                '/rplidar_ros/scan',
                self.laser_data_stream,
                10
            )

            self.video_subscription = self.create_subscription(
                CameraMsg,
                '/camera_pkg/video_mjpeg',
                self.camera_data_stream,
                10
            )

        elif msg.payload == b"no-data" and self.sending_data == 1:
            self.sending_data = 0
            self.destroy_subscription(self.video_subscription)
            self.destroy_subscription(self.lidar_subscription)
            time.sleep(1)
            self.sock.close()
        else:
            self.get_logger().warn("Mensaje desconocido recibido")
            self.led_state(red=10000000, green=0, blue=0)
            time.sleep(0.2)
            self.led_state(red=0, green=0, blue=0)
            time.sleep(0.2)
            self.led_state(red=10000000, green=0, blue=0)
            time.sleep(0.2)
            self.led_state(red=0, green=0, blue=0)
            time.sleep(0.5)
```

```
            # Restaurar el estado de los LEDs según el modo actual
            if self.status == 1:
                self.led_state(red=10000000, green=10000000,
blue=0)
            elif self.status == 2:
                self.led_state(red=0, green=10000000, blue=0)
            elif self.status == 3:
                self.led_state(red=0, green=0, blue=10000000)

    def on_disconnect(self, client, userdata, rc):
        if self.periodic_process_launched == 1:
            self.q_MQTT_periodic_sender.put(0)
            self.periodic_process_launched = 0

        self.led_state(red=10000000, green=0, blue=0)
        self.call_enable_cloud_control(False)
        self.call_enable_local_autonomous_control(False)

        time.sleep(0.1)
        self.pub_manual_drive.publish(ServoCtrlMsg(angle=0.0,
throttle=0.0))
        self.get_logger().info("Desconectado")

    def connect_mqtt(self):
        """Intentar conectar con el servidor MQTT."""
        while self.conectado == 0:
            try:
                self.client_MQTT.connect(mqtt_server_ip,
mqtt_server_port, 5)
                self.conectado = 1
            except Exception as e:
                self.led_state(red=10000000, green=0, blue=0)
                self.get_logger().error(f"No conectado: {e}")
                time.sleep(5)
        self.client_MQTT.loop_start()

    # Métodos para llamar a los servicios
    def led_state(self, red, green, blue):
        """Llamar al servicio de control de LEDs."""
 #       while not
self.led_service.wait_for_service(timeout_sec=1.0):
 #          self.get_logger().info('Servicio de LED no disponible,
esperando...')
 #          return
        request = SetLedCtrlSrv.Request()
        request.red = red
        request.green = green
        request.blue = blue
        self.led_service.call_async(request)


    def call_calibration(self, max_val, mid_val, min_val,
polarity):
        """Llamar al servicio de calibración."""
 #       while not
self.calibration_service.wait_for_service(timeout_sec=1.0):
```

```
 #           self.get_logger().info('Servicio de calibración no
disponible, esperando...')

        request = SetCalibrationSrv.Request()
        request.max = max_val
        request.mid = mid_val
        request.min = min_val
        request.polarity = polarity
        self.calibration_service.call_async(request)

    #def battery_level(self):
        #"""Llamar al servicio de nivel de batería."""
      #  while not
self.battery_level_service.wait_for_service(timeout_sec=1.0):
        #    self.get_logger().info('Servicio de batería no
disponible, esperando...')

        #request = BatteryLevelSrv.Request()
       # future = self.battery_level_service.call_async(request)

        # Corregido - Esperar a que se complete la llamada
        #rclpy.spin_until_future_complete(self, future)

       # return future.result()



    def llamar_servicio_video(self):
#        while not
self.inicioimagenes.wait_for_service(timeout_sec=1.0):
#           self.get_logger().warn("Esperando servicio de estado de
video...")

        req = VideoStateSrv.Request()
        req.activate_video = 1

        self.get_logger().info("Llamando al servicio de estado de
video...")

        future = self.inicioimagenes.call_async(req)

        # Esperar la respuesta antes de continuar (bloqueante, pero
útil en init)
#        rclpy.spin_until_future_complete(self, future)

#        if future.result() is not None:
#            self.get_logger().info("Servicio de estado de video
activado correctamente")
#       else:
#           self.get_logger().warn("El servicio de estado de video
no respondió correctamente")

    def call_enable_local_autonomous_control(self, enabled):
        """Llamar al servicio para control autónomo local."""
 #       while not
self.enable_local_autonomous_control.wait_for_service(timeout_sec=1
.0):
```

```
  #          self.get_logger().info('Servicio de control autónomo
local no disponible, esperando...')

        request = SetBool.Request()
        request.data = enabled
        self.enable_local_autonomous_control.call_async(request)

    def call_enable_cloud_control(self, enabled):
        """Llamar al servicio para control desde la nube."""
#         while not
self.enable_cloud_control.wait_for_service(timeout_sec=1.0):
#             self.get_logger().info('Servicio de control en la nube
no disponible, esperando...')

        request = SetBool.Request()
        request.data = enabled
        self.enable_cloud_control.call_async(request)

    def call_enable_voice_control(self, enabled):
        """Llamar al servicio para control de voz desde la nube."""
#         while not
self.enable_cloud_control.wait_for_service(timeout_sec=1.0):
#         self.get_logger().info('Servicio de control en la nube no
disponible, esperando...')

        request = SetBool.Request()
        request.data = enabled
        self.enable_voice_control.call_async(request)

def main(args=None):
    print("\n\t\tNODO AdminCommunicationsNode\n\n")
    rclpy.init(args=args)
    node = AdminCommunicationsNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Codigo Autonomuos_control_node

```
import rclpy
from rclpy.node import Node
import imutils
import cv2
import time
from . import artemis_autonomous_car
from sensor_msgs.msg import LaserScan
from deepracer_interfaces_pkg.msg import CameraMsg
from deepracer_interfaces_pkg.msg import ServoCtrlMsg
from deepracer_interfaces_pkg.srv import SetLedCtrlSrv
from std_srvs.srv import SetBool
from cv_bridge import CvBridge, CvBridgeError
from rclpy.qos import QoSProfile
import numpy as np

class AutonomousControlNode(Node):
    def __init__(self):
        super().__init__('autonomous_control_node')
        self.aac = None
        self.encendido = False
        self.video_subscription = None
        self.lidar_subscription = None
        self.error = 0
        self.bridge = CvBridge()


        # Publishers
        self.pub_manual_drive = self.create_publisher(ServoCtrlMsg,
'/ctrl_pkg/servo_msg', 10)

        # Services
        self.srv = self.create_service(SetBool,
'enable_local_autonomous_control',

self.handle_enable_local_autonomous_control)

        # Service client
        self.led_service = self.create_client(SetLedCtrlSrv,
'/servo_pkg/set_led_state')

    def handle_enable_local_autonomous_control(self, request,
response):
        if request.data == False and self.encendido == True:
            self.get_logger().info("Local autonomous control off")
            self.encendido = False
            self.destroy_subscription(self.video_subscription)
            self.destroy_subscription(self.lidar_subscription)
            response.success = True
            response.message = 'OK'
            return response

        if request.data == True and self.encendido == False:
            path = [2,2,2,2,2,2,2,2,2,2]
            self.get_logger().info("Local autonomous control on")
            self.aac =
artemis_autonomous_car.artemis_autonomous_car(path)
```

```
            # Actualizar LEDs según el estado de error
            if self.error == 0:
                future = self.led_state(red=0, green=10000000,
blue=0)
            else:
                future = self.led_state(red=10000000, green=0,
blue=10000000)

            self.encendido = True
            self.video_subscription = self.create_subscription(
                CameraMsg,
                '/camera_pkg/video_mjpeg',
                self.CameraDataReceived,
                10)

            self.lidar_subscription = self.create_subscription(
                LaserScan,
                '/rplidar_ros/scan',
                self.LaserDataReceived,
                10)

            response.success = True
            response.message = 'OK'
            return response
        else:
            response.success = False
            response.message = 'Already done'
            return response

    def LaserDataReceived(self, msg):
        if self.aac:
            self.aac.proceso_lidar(list(msg.ranges), False)

    def CameraDataReceived(self, msg):
        velocidad = 0
        k = 1

        img = self.bridge.imgmsg_to_cv2(msg.images[0], "bgr8")
        control_giro, control_acelerador, trayectory_not_found =
self.aac.proceso_fotograma(img, False, 0)
        self.get_logger().info(f"giro: {control_giro}")
        self.get_logger().info(f"acelerador: {control_acelerador}")

        control_msg = ServoCtrlMsg()
        control_msg.angle = float(control_giro - 0.25)
        control_msg.throttle = float(control_acelerador)
        self.pub_manual_drive.publish(control_msg)

        if trayectory_not_found == 1 and self.error == 0:
            self.error = 1
            future = self.led_state(red=10000000, green=0,
blue=10000000)
        if trayectory_not_found == 0 and self.error == 1:
            self.error = 0
            future = self.led_state(red=0, green=10000000, blue=0)

    def led_state(self, red, green, blue):
```

```
        """Llamar al servicio de control de LEDs."""
#        while not
self.led_service.wait_for_service(timeout_sec=1.0):
#           self.get_logger().info('Servicio de LED no disponible,
esperando...')
#            return
        request = SetLedCtrlSrv.Request()
        request.red = red
        request.green = green
        request.blue = blue
        self.led_service.call_async(request)

def main(args=None):
    print("\n\t\tNODO CONTROL EN LOCAL\n\n")
    rclpy.init(args=args)
    autonomous_control = AutonomousControlNode()
    rclpy.spin(autonomous_control)
    autonomous_control.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Codigo Cloud_control_node

```
import configparser
import io
import signal
import sys
import socket
import rclpy
from rclpy.node import Node
import pickle
import struct
import cv2
import threading
import time
from threading import Timer
from threading import Lock
from rclpy.qos import QoSProfile, QoSReliabilityPolicy
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import LaserScan
from deepracer_interfaces_pkg.msg import CameraMsg
from deepracer_interfaces_pkg.msg import ServoCtrlMsg
from deepracer_interfaces_pkg.srv import SetLedCtrlSrv
from deepracer_interfaces_pkg.srv import BatteryLevelSrv
from deepracer_interfaces_pkg.srv import VideoStateSrv
from deepracer_interfaces_pkg.srv import ActiveStateSrv
from std_srvs.srv import SetBool
import select



# Leer configuración
vehicle_config = configparser.ConfigParser(allow_no_value=True)
with
open("/home/deepracer/SoftARTEMIS/soft_artemis/soft_artemis/vehicle
.conf") as config_file:
    vehicle_config.read_file(io.StringIO(config_file.read()))

cloud_server_ip = vehicle_config.get("cloud_autonomous_driving",
"cloud_server_ip")
cloud_server_port =
int(vehicle_config.get("cloud_autonomous_driving",
"cloud_server_port"))
cloud_server_address = (cloud_server_ip, cloud_server_port)


class CloudControlNode(Node):
    def __init__(self):
        super().__init__('cloud_control_node')
        self.bridge = CvBridge()
        self.encendido = False
        self.cloud_server_address = cloud_server_address
        self.lock = Lock()
        self.sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
        self.sock.setblocking(False)  # Configurar socket en modo
no bloqueante
```

```
        self.last_frame_time = time.time()  # Inicializar el tiempo
del último frame
        self.timer_frame_not_received = None  # Timer de detección
de frame no recibido
        self.frames_sent_count = 0  # Contador de frames enviados

        # Crear servicio
        self.s = self.create_service(SetBool,
'enable_cloud_control', self.handle_enable_cloud_control)

        # Crear publisher
        self.pub_manual_drive = self.create_publisher(ServoCtrlMsg,
'/ctrl_pkg/servo_msg', 10)

        # Iniciar el timer de detección de frames perdidos
    #    self.timer_frame_not_received = self.create_timer(0.3,
self.frame_not_received)

        # Iniciar el loop de control en un hilo separado
        self.create_timer(0.01, self.control_loop)  # Ejecutar cada
10 ms

    def handle_enable_cloud_control(self, request, response):
        if request.data and not self.encendido:
            self.get_logger().info("Cloud control on - iniciando
servicios")
            self.encendido = True

            # Creación de suscripciones en ROS2
            self.get_logger().info("Creando suscripciones")
            self.lidar_subscription = self.create_subscription(
                LaserScan,
                '/rplidar_ros/scan',
                self.laser_data_stream,
                10
            )

            self.video_subscription = self.create_subscription(
                CameraMsg,
                '/camera_pkg/video_mjpeg',
                self.camera_data_stream,
                10
            )
            self.get_logger().info("Suscripciones creadas
exitosamente")

            response.success = True

        elif not request.data and self.encendido:
            self.get_logger().info("Cloud control off")
            self.encendido = False

            # Destruir las suscripciones si existen
            if hasattr(self, 'video_subscription'):
                self.destroy_subscription(self.video_subscription)
            if hasattr(self, 'lidar_subscription'):
                self.destroy_subscription(self.lidar_subscription)
```

```
            # Detener el timer de detección de frames perdidos
          #     if self.timer_frame_not_received is not None:
            #
self.destroy_timer(self.timer_frame_not_received)
             #     self.timer_frame_not_received = None

            response.success = True
        else:
            response.success = False

        response.message = "OK"
        return response

    def send_data(self, tipo, serialized_data):
        with self.lock:
            self.sock.sendto(struct.pack('c', tipo.encode()) +
serialized_data, self.cloud_server_address)

 #   def send_data_img(self, tipo, serialized_data):
  #      UDPMAXBYTES = 65000
  #      with self.lock:
    #        self.sock.sendto(struct.pack('c', tipo.encode()) +
struct.pack('B', 0) + serialized_data[0:UDPMAXBYTES],
self.cloud_server_address)
 #            self.sock.sendto(struct.pack('c', tipo.encode()) +
struct.pack('B', 1) + serialized_data[UDPMAXBYTES:(UDPMAXBYTES *
2)], self.cloud_server_address)

    def camera_data_stream(self, msg):
        try:
            self.get_logger().info(f"camera_data_stream ejecutado
con {len(msg.images)} imagenes")
            if msg.images and len(msg.images) > 0:
                imgL = self.bridge.imgmsg_to_cv2(msg.images[0],
"bgr8")
                imgL = cv2.imencode('.jpg', imgL,
[int(cv2.IMWRITE_JPEG_QUALITY), 12])[1]
                serialized_dataL = pickle.dumps(imgL)
                self.send_data('I', serialized_dataL)
                self.last_frame_time = time.time()

        except Exception as e:
            self.get_logger().error(f"Error en camera_data_stream:
{str(e)}")

    def laser_data_stream(self, msg):
        try:
            self.get_logger().info("lidar")
            serialized_data = pickle.dumps(msg.ranges)
            self.send_data('L', serialized_data)
        except Exception as e:
            self.get_logger().error(f"Error en laser_data_stream:
{str(e)}")


    #  def frame_not_received(self):
        #  if self.encendido:
         #    msg = ServoCtrlMsg()
```

```
          #    msg.angle = 0.0
           #  #   msg.throttle = 0.0
         #     self.pub_manual_drive.publish(msg)
            #  time_since_last_frame = time.time() -
self.last_frame_time
           #   self.get_logger().warn(f'Frame no recibido en
{time_since_last_frame:.2f}s, deteniendo vehículo')

    def control_loop(self):
        if self.encendido:
            try:
                # Usar select para evitar bloqueo
                ready, _, _ = select.select([self.sock], [], [],
0.01)
                if ready:
                    data, address = self.sock.recvfrom(99999)
                    if data and len(data) >= 17:
                        # Desempaquetar el tipo de frame (usando
data[0:1])
                        frame = struct.unpack('c', data[0:1])[0]

                        # Agregar logs para debug
                        self.get_logger().info(f"Tipo de frame:
{frame}, longitud datos: {len(data)}")

                        # Si es un comando de control
                        if frame == b'C':
                            # Desempaquetar los valores de control
                            control_giro = struct.unpack('d',
data[1:9])[0]
                            control_acelerador = struct.unpack('d',
data[9:17])[0]

                            # Crear y publicar el mensaje de
control
                            msg = ServoCtrlMsg()
                            msg.angle = control_giro
                            msg.throttle = control_acelerador
                            self.pub_manual_drive.publish(msg)

                            self.get_logger().info(f"Giro:
{control_giro}, Acelerador: {control_acelerador}")
            except BlockingIOError:
                pass  # No hay datos disponibles, continuar sin
errores
            except Exception as e:
                self.get_logger().error(f"Error en control_loop:
{str(e)}")



def main(args=None):
    print("\n\t\tNODO CLOUD CONTROL\n\n")
    rclpy.init(args=args)
    node = CloudControlNode()

    try:
```

```
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()


if __name__ == '__main__':
    main()
```