ULPGC Universidad de Las Palmas de Gran Canaria | eite ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO DE FIN DE GRADO

Sistema de detección de obstáculos mediante ROS y procesamiento embebido en Jetson Nano

Titulación: Grado en Ingeniería de Tecnologías de la Telecomunicación

Mención: Telemática

Autor: Airam López Mendoza

Tutor: Itziar Goretti Alonso González

Fecha: **10 de junio del 2025**

Resumen

Este Trabajo Fin de Grado se centra en el desarrollo de un sistema embebido basado en ROS y la plataforma Jetson Nano, orientado a la clasificación y detección de barreras arquitectónicas. Para ello, se utilizó ROS Noetic como entorno de desarrollo, implementado sobre la plataforma de procesamiento embebido Jetson Nano. Se trabajó en la integración de sensores clave, como una cámara ToF y un LiDAR, complementados con un sistema de visión por computador basado en el algoritmo YOLO para la detección de objetos. Se implementaron nodos específicos para la cámara ToF, el mapeo, YOLO, el control motorizado, la comunicación y el LiDAR.

En la primera fase del proyecto, se llevó a cabo la configuración inicial del hardware y software necesarios (ROS y Jetson Nano) y la integración de los sensores. Se logró controlar con éxito la cámara ToF y se integró el LiDAR al sistema, verificando el funcionamiento simultáneo de todos los sensores. Posteriormente, se desarrolló e incorporó un modelo de detección de objetos, entrenando un algoritmo YOLO para identificar barreras arquitectónicas y otros objetos relevantes. Además, se implementó un script que utiliza la información de detección de YOLO junto con un flujo adicional basado en n8n.

El sistema completo, incluyendo sus componentes de detección, fue probado mediante evaluaciones independientes que permitieron trabajar con resultados parciales, debido a las limitaciones del hardware disponible. Se realizó una validación manual de la detección de barreras, confirmando su funcionamiento con un margen de error reducido. Asimismo, se implementó un prototipo complejo de conducción autónoma que integra la detección desarrollada. Estos resultados iniciales validan el desarrollo e implementación de los componentes clave del sistema de detección y control, evidenciando su potencial aplicación práctica.

Abstract

This Final Degree Project focused on the development of an embedded system based on ROS and the Jetson Nano platform, aimed at the classification and detection of architectural barriers. For this purpose, ROS Noetic was used as the development environment, implemented on the embedded processing platform Jetson Nano. The project involved integrating key sensors such as a ToF camera and a LiDAR, complemented by a computer vision system based on a YOLO neural network for object detection. Specific nodes were implemented for the ToF camera, mapping, YOLO, motor control, communication, and LiDAR.

In the first phase of the project, the initial configuration of the necessary hardware and software (ROS and Jetson Nano) and sensor integration were carried out. Successful control of the ToF camera was achieved, and the LiDAR was integrated into the system, verifying the simultaneous operation of all sensors. Subsequently, an object detection model was developed and incorporated by training a YOLO neural network to identify architectural barriers and other relevant objects. Additionally, a script was implemented that uses the detection information from YOLO together with an additional workflow based on n8n.

The complete system, including its detection components, was tested through independent evaluations that allowed working with partial results due to the limitations of the available hardware. Manual validation of the barrier detection was performed, confirming its operation with a low margin of error. Furthermore, a complex prototype of autonomous driving integrating the developed detection was implemented. These initial results validate the development and implementation of the key components of the detection and control system, demonstrating its potential practical application.

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a mi familia, y en especial a mi madre, por su constante apoyo y por estar siempre a mi lado en cada paso del camino. Asimismo, quiero dar las gracias a mis amigos por su apoyo incondicional, en concreto a una persona muy especial entre ellos, cuya cercanía y ánimo han marcado una diferencia significativa a lo largo de este proceso.

Un agradecimiento muy especial a mi abuelo, que siempre ha esperado con ilusión poder ver este logro cumplido. Su paciencia y confianza en mí han sido una fuente constante de motivación.

Tampoco puedo dejar de mencionar a todos los profesores, compañeros y tutores que han contribuido a mi formación y crecimiento durante estos años. En particular, quiero agradecer a quienes me han acompañado durante este último curso, ya que gracias a su orientación y colaboración ha sido posible la realización de este Trabajo de Fin de Grado.

Tabla de contenido

Índice de Tablas

Índice de Figuras

Definición de acrónimos

Tabla 1 Tabla de acrónimos

RGB-D	Red Green Blue-Depth
LiDAR	Light Detection and Ranging
ToF	Time of Flight
YOLO	You Only Look Once
ROS	Robot Operating System
SLAM	Simultaneous localization and mapping
ARM	Advanced RISC Machines
CUDA	Compute Unified Device Architecture
cuDNN	CUDA Deep Neural Network
IMUs	Inertial Measurement Units
RTAB-Map	Real-Time Appearance-Based Mapping
LTS	Long-Term Support
SDK	Software Development Kit
DK	Development Kit
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CNN	Convolutional Neural Networks
SSD	Single Shot Detector
TF	Transformations Framework
URDF	Unified Robot Description Format

XML	Extensible Markup Language
UART	Universal Asynchronous Receiver-Transmitter
I2C	Inter-Integrated Circuit166175
GPIO	General Purpose Input/Output
CSI	Camera Serial Interface
RANSAC	Random Sample Consensus
IAGen	Inteligencia Artificial Generativa
API	Application Programming Interface
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
IDE	Integrated Development Environment

Capítulo 1. Introducción y Objetivos

En este capítulo se contextualiza el Trabajo Fin de Grado realizado, centrado en la detección de barreras arquitectónicas mediante sensores y visión artificial en plataformas embebidas. Se presentan los antecedentes tecnológicos que sustentan el proyecto, y se definen los objetivos generales y específicos. Asimismo, se describe brevemente la estructura del documento y el uso de herramientas basadas en inteligencia artificial que han sido aplicadas durante el desarrollo del sistema.

1.1 Introducción

La detección de obstáculos y, en particular, de barreras arquitectónicas [1] representa uno de los principales retos en la robótica móvil orientada a la asistencia personal y la mejora de la accesibilidad. A pesar de los avances en técnicas de percepción y navegación autónoma, la detección fiable de este tipo de obstáculos sigue siendo un problema complejo, especialmente en entornos dinámicos o con limitaciones de hardware.

Tradicionalmente, se han empleado cámaras RGB-D [2] o sensores LIDAR [3] de alto rendimiento para abordar este problema. Sin embargo, su uso en plataformas embebidas con recursos limitados plantea importantes desafíos. En este Trabajo Fin de Grado se propone una solución basada en una Jetson Nano [4], que integra una cámara ToF [5] con capacidad de proporcionar tanto información de profundidad como señal RGB, un sensor LIDAR adicional para refinar la precisión, y un sistema de visión artificial basado en el algoritmo YOLO [6] para la detección de objetos.

El sistema ha sido desarrollado utilizando ROS [7] como entorno de integración, permitiendo la sincronización entre sensores y la toma de decisiones en tiempo real. La complejidad del trabajo radica en la necesidad de combinar múltiples fuentes de información heterogénea en un sistema embebido, manteniendo un rendimiento adecuado y una fiabilidad suficiente para identificar de forma precisa barreras arquitectónicas en escenarios reales.

1.2 Antecedentes

En la actualidad, el desarrollo de sistemas robóticos embebidos ha ganado un gran impulso, especialmente en el ámbito de la robótica de servicio y asistencia. La necesidad de hacer frente a los desafíos de accesibilidad en entornos urbanos y domésticos ha motivado el diseño de soluciones inteligentes capaces de detectar barreras arquitectónicas, como escaleras, desniveles o mobiliario urbano, que dificultan la movilidad de personas con discapacidad.

En este contexto, la integración de sensores de percepción avanzados ha sido clave. El uso de cámaras ToF (Time-of-Flight), que ofrecen mapas de profundidad en tiempo real, se ha consolidado como una alternativa eficiente y menos costosa frente a otros sensores RGB-D convencionales. Asimismo, la incorporación de sensores LIDAR permite una mayor precisión en la medición de distancias, especialmente en condiciones lumínicas complejas o superficies reflectantes.

Por otro lado, los avances en visión artificial mediante redes neuronales convolucionales han permitido desarrollar modelos como YOLO (You Only Look Once), capaces de detectar objetos con gran rapidez y precisión. Estas tecnologías, combinadas, ofrecen un potencial significativo para construir sistemas de detección robustos en entornos no estructurados.

En cuanto al software, el uso de ROS (Robot Operating System) ha sido determinante en la evolución de la robótica. Este framework facilita la integración modular de componentes, la gestión de sensores heterogéneos y la implementación de algoritmos en tiempo real. Su compatibilidad con plataformas embebidas como Jetson Nano ha abierto la puerta a soluciones de bajo coste y alta capacidad de procesamiento, adaptadas a escenarios reales.

Este Trabajo Fin de Grado aborda el diseño e implementación de un sistema embebido que, mediante el uso de ROS, una cámara ToF con canal RGB, un LIDAR y el algoritmo YOLO, permite la detección precisa de obstáculos y barreras arquitectónicas en entornos cotidianos.

1.3 Objetivos

Este proyecto tiene como propósito la creación de un sistema de navegación autónoma basado en ROS sobre la plataforma Jetson Nano, con capacidad para generar mapas del entorno, detectar obstáculos y barreras arquitectónicas, y desplazarse de forma autónoma en interiores. El sistema se apoya en tecnologías de visión artificial y sensores de profundidad para interpretar el entorno en tiempo real, facilitando la toma de decisiones en la navegación.

Desde el inicio, el vehículo debe ser capaz de moverse de forma completamente autónoma, recopilando información del entorno mediante sensores, generando mapas mediante algoritmos SLAM [8], y reaccionando ante obstáculos o cambios en el entorno.

Los objetivos iniciales que se plantearon alcanzar en este proyecto son los siguientes:

- **Objetivo 1.** Implementar y configurar el entorno de desarrollo ROS sobre la plataforma Jetson Nano, incluyendo los paquetes necesarios para la integración y gestión de sensores, cámaras ToF y control del movimiento.
- **Objetivo 2.** Desarrollar el sistema de detección, clasificación y gestión de obstáculos y barreras arquitectónicas, utilizando sensores.
- **Objetivo 3.** Evaluar el sistema en distintos entornos mediante la generación de mapas y pruebas de obstáculos en distintos escenarios.

Sin embargo, durante el desarrollo se identificó que, para alcanzar una mayor precisión en el levantamiento de mapas y una navegación autónoma más robusta, era necesario complementar las cámaras ToF con un sensor LIDAR. Esta adición permitió una mayor fidelidad en la representación del entorno y una respuesta más precisa ante obstáculos.

Además, surgieron problemas con el hardware con el que se trabajaba, el cual era insuficiente para llevar a cabo este proyecto.

De esta manera, los objetivos finales se reformularon de la siguiente manera:

- **Objetivo 1.** Implementar y configurar el entorno de desarrollo ROS sobre la plataforma Jetson Nano, integrando sensores de profundidad como cámaras ToF y LIDAR, así como el control autónomo del movimiento.

- **Objetivo 2.** Desarrollar el sistema de detección, clasificación y gestión de obstáculos y barreras arquitectónicas, utilizando sensores.

- **Objetivo 3.** Diseñar la arquitectura software general del sistema en ROS y desarrollar el código correspondiente para la integración de nodos, la gestión de la información de los sensores y el control del movimiento, incluyendo el diseño de la lógica de conducción autónoma.

1.4 Contenidos

El presente proyecto se estructura en seis capítulos que abarcan de forma integral tanto los fundamentos teóricos como la implementación práctica del sistema de navegación autónoma desarrollado. La organización de los contenidos ha sido diseñada para ofrecer una visión progresiva y coherente del proceso, desde la motivación inicial hasta la evaluación de resultados y conclusiones.

En el **capítulo I** se presenta una introducción general al proyecto, en la que se detallan el contexto, los antecedentes, los objetivos del trabajo y los contenidos tratados. También se aborda el uso complementario de herramientas de inteligencia artificial generativa como apoyo en el desarrollo del proyecto.

El **capítulo II** ofrece una revisión de los conceptos y tecnologías clave utilizados. Se analizan los fundamentos del sistema operativo robótico ROS (Robot Operating System), la plataforma Jetson Nano, los principios de navegación autónoma, la técnica de mapeo SLAM, y la integración de sensores como el LIDAR y las cámaras ToF y RGB-D. Además, se profundiza en técnicas de detección de objetos mediante redes neuronales, como YOLO.

En el **capítulo III** se describe la arquitectura general del sistema y los componentes físicos utilizados en el robot móvil. Se explican las características de la plataforma Jetson Nano, los sensores empleados, y el modelo de comunicación entre dispositivos. Asimismo, se detalla la configuración de los nodos en ROS Noetic Ninjemys [9] para el procesamiento de imágenes y el levantamiento de mapas.

El **capítulo IV** se centra en el diseño e implementación del sistema dentro del entorno ROS. Se describe la estructura del workspace, la configuración de los paquetes utilizados y el desarrollo de los nodos encargados de integrar los sensores y controlar el robot. También se expone el flujo de ejecución del sistema desde el arranque hasta el desplazamiento autónomo con detección de obstáculos.

En el **capítulo V** se describen las pruebas planificadas y los resultados obtenidos en el proceso de validación del sistema. Inicialmente, se había previsto evaluar el comportamiento del robot en distintos entornos, la calidad de los mapas generados y la detección de obstáculos y barreras arquitectónicas. Sin embargo, debido a limitaciones de potencia en el dispositivo Jetson Nano, no fue posible ejecutar completamente el script de conducción autónoma ni generar los mapas finales con las barreras señaladas. A pesar de ello, se realizaron algunas pruebas parciales, como la detección de objetos mediante YOLO, un prototipo avanzado de conducción autónoma y la verificación del funcionamiento conjunto de los sensores. Se documenta el cumplimiento parcial de ciertos objetivos, así como los elementos que no pudieron validarse por completo.

Por último, el **capítulo VI** presenta las conclusiones del trabajo. Se resumen los principales logros del proyecto, se identifican las limitaciones encontradas y se proponen posibles mejoras y líneas de trabajo futuras que podrían permitir una ampliación de las funcionalidades del sistema o su adaptación a nuevos contextos de aplicación.

1.5 Uso de IA Generativas

Siguiendo las directrices establecidas en las *"Recomendaciones sobre el uso de la IAGen en la ULPGC"* [10] , aprobadas por el Consejo de Gobierno Extraordinario de la ULPGC [11] el 6 de junio de 2024, en la elaboración de este Trabajo Fin de Grado se ha hecho uso de inteligencia artificial generativa de forma responsable y ética, equiparable al empleo de consultas avanzadas en motores de búsqueda o en foros técnicos especializados.

Dado que la documentación relativa a la integración de ROS Noetic Ninjemys con ciertos componentes recientes —como la cámara Azure Kinect [12], modelos de detección como YOLOv7 [13], o configuraciones en sistemas embebidos como Jetson Nano— puede ser limitada o fragmentada, se utilizaron herramientas como ChatGPT para obtener explicaciones preliminares, aclarar conceptos técnicos y recibir ejemplos que sirviesen como referencia durante el desarrollo. En ningún caso estas respuestas fueron tomadas como soluciones definitivas, sino como guías iniciales sujetas a verificación, adaptación y validación por parte del autor.

Además, se recurrió a estas herramientas para realizar tareas de apoyo editorial, tales como la corrección de estilo, la reescritura de apartados para mejorar la claridad del texto y la revisión orto tipográfica, de forma similar a las funciones ofrecidas por plataformas como Microsoft Word Office 365 [14].

Capítulo 2. Estado del Arte

En este capítulo se comienza por introducir el concepto de barreras arquitectónicas, dado que representan el eje central del problema que se pretende abordar en el presente proyecto. Se analiza su impacto tanto en la movilidad de personas con discapacidad, lo que justifica la necesidad de desarrollar sistemas capaces de detectarlas y actuar en consecuencia. A continuación, se abordan las tecnologías clave y los fundamentos teóricos que sustentan el desarrollo del sistema. Se estudia el middleware ROS Noetic Ninjemys como base para la robótica autónoma, junto con su ecosistema de nodos y paquetes. También se analizan las plataformas embebidas como la Jetson Nano, así como los sensores empleados para la percepción del entorno, incluyendo cámaras RGB-D, sensores ToF y LiDAR. Además, se revisan las técnicas de mapeo y localización basadas en SLAM, y se examinan los enfoques actuales para la detección de objetos mediante redes neuronales, con especial atención al uso de modelos como YOLO en sistemas robóticos.

2.1 Barreras arquitectónicas

Las barreras arquitectónicas son aquellos elementos del entorno construido que dificultan o impiden la libre circulación y el acceso de las personas, especialmente de aquellas con movilidad reducida o discapacidades. Estas barreras pueden estar presentes tanto en espacios exteriores como interiores, y abarcan desde obstáculos estructurales como escaleras sin alternativa accesible, hasta mobiliario mal ubicado o diferencias de nivel sin rampas ni señalización adecuada.

En el contexto del diseño de entornos accesibles, la eliminación o mitigación de estas barreras constituye una prioridad para garantizar la inclusión y seguridad. En robótica móvil, estas barreras representan también un desafío técnico, ya que pueden impedir el desplazamiento fluido de dispositivos autónomos, especialmente si no se detectan ni gestionan adecuadamente en tiempo real.

En espacios interiores, las barreras arquitectónicas son diversos obstáculos o características del entorno que dificultan o impiden la movilidad de personas, especialmente aquellas con discapacidad o movilidad reducida. A continuación, se describen las principales barreras presentes en interiores, junto con sus soluciones más comunes:

- **Escaleras:** Representan un obstáculo importante para personas en silla de ruedas o con dificultades de movilidad.
  Soluciones: Rampas accesibles, ascensores, plataformas elevadoras o sistemas de salvaescaleras.

- **Desniveles o escalones pequeños**: Incluso pequeñas diferencias de altura pueden suponer un impedimento.
  Soluciones: Rampas suaves con pendiente adecuada, señalización táctil o auditiva.

- **Puertas estrechas**: Pueden dificultar el paso, especialmente de sillas de ruedas o andadores.
  Soluciones: Puertas automáticas, ensanchamiento de accesos, eliminación o rebaje de umbrales.

- **Pasillos estrechos y mobiliario mal ubicado**: Reducen el espacio navegable y pueden ser un obstáculo físico.
  Soluciones: Reorganización del mobiliario, señalización adecuada, espacios libres mínimos según normativa.

- **Mobiliario fijo y temporal**: Elementos como estanterías, archivadores, papeleras o sillas desplazadas que bloquean el paso.
  Soluciones: Planificación del espacio, uso de mobiliario modular, almacenamiento adecuado.

- **Pavimentos resbaladizos o con textura inadecuada**: Pueden provocar caídas.
  Soluciones: Uso de pavimentos antideslizantes, señalización, alfombrillas de seguridad.

- **Mostradores demasiado altos**: Dificultan la interacción de personas en silla de ruedas o de baja estatura.
  Soluciones: Incorporación de zonas accesibles más bajas o mostradores con doble altura.

No obstante, es importante señalar que algunas de estas barreras no podrán ser abordadas dentro del presente proyecto. Esto se debe tanto a la falta de los recursos técnicos necesarios (como sensores especializados o modelos de IA entrenados para la detección de condiciones del suelo), como a las limitaciones de tiempo y alcance propias de este

trabajo. Concretamente, las barreras relacionadas con el estado y tipo de pavimento (por ejemplo, superficies resbaladizas, irregulares o inestables), así como la presencia de mostradores altos, no serán detectadas ni evaluadas por el sistema desarrollado. Si bien su inclusión sería deseable para lograr un análisis más integral de la accesibilidad del entorno, su implementación requeriría un desarrollo adicional considerable que excede el marco establecido para este proyecto.

2.2 ROS

El Sistema Operativo de Robot (ROS, por sus siglas en inglés) es un framework de código abierto ampliamente utilizado en el ámbito de la robótica. Está diseñado para facilitar el desarrollo de aplicaciones robóticas complejas mediante una arquitectura modular basada en nodos que se comunican entre sí de forma distribuida. Esta estructura permite dividir las funcionalidades del sistema en procesos independientes, que pueden ser desarrollados, ejecutados y depurados por separado, favoreciendo la escalabilidad y el mantenimiento del software.

El corazón de ROS es el ROS Master [15], un nodo central que gestiona la información sobre otros nodos, permitiendo su descubrimiento y comunicación. Los nodos pueden intercambiar datos a través de topics [16] (temas), que son canales de comunicación para mensajes de tipo publicador-suscriptor, así como mediante servicios y acciones, lo que brinda una gran flexibilidad para diseñar interacciones síncronas o asíncronas.

La Ilustración 1 representa el funcionamiento interno del sistema ROS, destacando el papel central del ROS Master como nodo de coordinación. En este esquema, se visualiza cómo diferentes nodos se registran en el ROS Master y establecen comunicación entre ellos mediante tópicos, servicios y acciones. Cada nodo puede actuar como publicador o suscriptor, así como ofrecer o consumir servicios y acciones. Además, el uso del servidor de parámetros permite compartir información global de configuración. Este modelo distribuido facilita una arquitectura flexible, escalable y eficiente para el desarrollo de aplicaciones robóticas complejas.

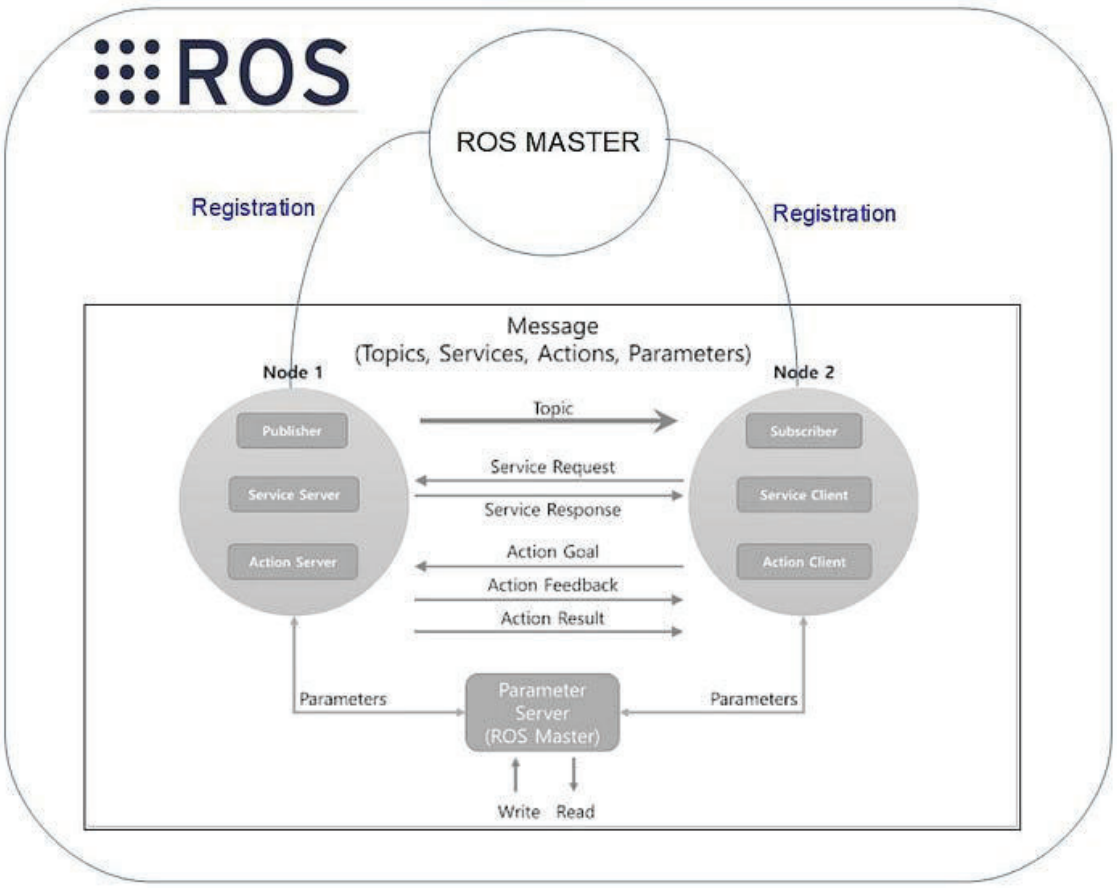

Ilustración 1 Esquema del funcionamiento de ROS [17]

Gracias a su enfoque modular, ROS ha sido empleado en numerosos proyectos de robótica móvil, manipulación, visión por computador y sistemas autónomos [18]. Por ejemplo, se ha utilizado en vehículos terrestres y aéreos para tareas como el mapeo en 3D [19], la navegación autónoma en entornos dinámicos [20], la detección de obstáculos, o la interacción con humanos. Herramientas como *rviz* [21] para visualización o *rosbag* [22] para el registro y análisis de datos son también parte del ecosistema ROS, facilitando las etapas de desarrollo y validación.

En trabajos similares al presente, ROS ha sido empleado para combinar técnicas de mapeo simultáneo y localización (SLAM) con sistemas de navegación autónoma, mediante sensores como cámaras RGB-D, LiDAR o cámaras de tiempo de vuelo (ToF). En particular, frameworks como RTAB-Map [23] permiten generar mapas tridimensionales en tiempo real, que pueden ser utilizados por planificadores de trayectoria integrados en ROS, como Nav2 [24], para realizar desplazamientos autónomos en entornos no estructurados.

Algunos de los proyectos relevantes que se pueden destacar en el contexto del uso de ROS son el Trabajo de Fin de Grado “Automatización del posicionamiento y seguimiento de trayectorias de un vehículo autónomo basado en ROS sobre planos previamente

levantados” [25], centrado en la navegación autónoma y el procesamiento de datos sensoriales; el proyecto YOLOv5 Autonomous Driving Scenario [26], que constituye la principal fuente de inspiración para este trabajo, al combinar detección de objetos mediante redes neuronales con planificación de trayectorias; y el uso del robot TurtleBot [27], una plataforma educativa y de investigación ampliamente utilizada para el desarrollo de soluciones de mapeo y navegación en entornos controlados, la cual ha servido como base en numerosos proyectos académicos y de prototipado rápido con ROS.

2.3 ROS Noetic Ninjemys

ROS Noetic Ninjemys es la última distribución oficial del sistema operativo para robots dentro de la familia ROS 1, lanzada en mayo de 2020 y diseñada específicamente para ser compatible con Ubuntu 20.04 LTS. Esta versión representa la culminación de una década de desarrollo en ROS 1, integrando mejoras de rendimiento, estabilidad y compatibilidad con Python 3 [28], que la convierten en una de las versiones más robustas y completas de este sistema.

Su elección para el presente proyecto no fue casual, sino el resultado de un análisis basado en diversos factores. Entre ellos, el hecho de que ROS Noetic continúa siendo mantenido oficialmente hasta mayo de 2025 ofrecía, en su momento, ciertas garantías en cuanto a soporte, documentación y comunidad activa. No obstante, esta misma fecha marca también el fin de su soporte, lo que limita su viabilidad a largo plazo. Aunque frente a versiones anteriores como ROS Melodic o ROS Kinetic, Noetic presentaba ventajas claras —mejor compatibilidad con bibliotecas modernas, mayor soporte para arquitecturas ARM como la Jetson Nano, e integración más fluida con herramientas recientes gracias a su soporte exclusivo para Python 3—, la proximidad del fin de su ciclo de vida representa una desventaja significativa en términos de escalabilidad y mantenimiento futuro del sistema.

Además, ROS Noetic es una de las pocas versiones que dispone de controladores estables y bien documentados para la cámara utilizada en este trabajo, la Azure Kinect, sin necesidad de recurrir a adaptaciones complejas o migraciones prematuras a ROS 2, donde muchas bibliotecas aún no han alcanzado la madurez deseada. Esta disponibilidad de drivers ha sido fundamental para garantizar una integración directa y eficaz con el sistema de percepción del robot.

Por otro lado, la similitud con el proyecto de referencia “YOLOv5 Autonomous Driving Scenario” también ha sido determinante. Este proyecto base emplea ROS 1 como entorno

de desarrollo, y adoptar la misma versión facilita la comprensión y adaptación del código fuente, los flujos de trabajo y la estructura modular del sistema.

Aunque ROS 2 ofrece ventajas a largo plazo como una arquitectura más segura, comunicación en tiempo real y soporte nativo para múltiples sistemas operativos, su adopción en ciertos entornos aún está en proceso de consolidación, especialmente en lo relativo a compatibilidad con hardware específico como sensores y cámaras. Por tanto, ROS Noetic se presenta como la opción más equilibrada entre estabilidad, compatibilidad y funcionalidad para los objetivos planteados en este trabajo.

2.4 Plataforma Jetson Nano

La NVIDIA Jetson Nano es una plataforma de computación de alto rendimiento orientada al desarrollo de sistemas embebidos con capacidades de inteligencia artificial, visión por computadora y robótica. Desde su lanzamiento, ha sido ampliamente adoptada en proyectos educativos, prototipos industriales y sistemas autónomos gracias a su bajo coste, consumo energético reducido y una excelente relación potencia/precio.

La Jetson Nano cuenta con una GPU NVIDIA Maxwell de 128 núcleos, acompañada de una CPU ARM Cortex-A57 de cuatro núcleos, que permite ejecutar modelos de deep learning [29], procesamiento de imágenes y otras tareas intensivas en cálculo en dispositivos de pequeño tamaño. Está diseñada específicamente para trabajar con herramientas de desarrollo como CUDA [30], cuDNN [31] y TensorRT [32], lo cual la hace ideal para proyectos que requieren inferencia en tiempo real.

Uno de los aspectos clave que ha impulsado su uso en robótica es su total compatibilidad con ROS (Robot Operating System), así como el soporte oficial para sistemas operativos como Ubuntu 18.04 y 20.04, que permiten instalar desde versiones antiguas de ROS como Melodic hasta ROS Noetic, utilizada en este trabajo.

A lo largo de los últimos años, se han documentado múltiples casos de uso que demuestran la eficacia de esta plataforma en entornos reales. Entre los ejemplos más destacados se encuentran robots móviles para tareas de inspección industrial, vehículos autónomos a pequeña escala con visión por computadora, drones con navegación visual, y proyectos académicos centrados en la detección de objetos, navegación y SLAM. Su compatibilidad con frameworks como TensorFlow [33], PyTorch [34] o OpenCV [35] ha facilitado aún más su integración en soluciones de IA en tiempo real.

En este trabajo se ha optado por utilizar la Jetson Nano como plataforma de desarrollo con el objetivo de dotar al sistema de capacidades de autonomía, siguiendo además el enfoque del proyecto de referencia mencionado anteriormente (*YOLOv5 Autonomous Driving Scenario*). Esta elección permite no solo replicar y adaptar un modelo ya validado en otro entorno, sino también comprobar en la práctica la utilidad real de la Jetson Nano en sistemas de robótica móvil con visión por computador, navegación y procesamiento en tiempo real dentro de un contexto académico.

2.5 Navegación Autónoma y SLAM

La navegación autónoma es uno de los pilares fundamentales en el desarrollo de vehículos robóticos móviles. Su objetivo es permitir que el robot sea capaz de desplazarse por un entorno desconocido sin intervención humana, tomando decisiones en tiempo real a partir de los datos percibidos por sus sensores. Este proceso requiere la combinación de diversas tecnologías, entre las que destaca el mapeo, la localización, la planificación de rutas y la evasión de obstáculos.

Una de las técnicas más empleadas para alcanzar esta autonomía es SLAM (Simultaneous Localization and Mapping), que permite al robot construir un mapa del entorno al mismo tiempo que estima su propia posición dentro de él. Esta técnica es especialmente útil en entornos previamente desconocidos, donde no se dispone de un mapa predefinido.

La Ilustración 2 muestra un esquema representativo del funcionamiento del algoritmo SLAM. En él, se observa cómo el robot (representado por triángulos) se desplaza por el entorno mientras realiza estimaciones de su posición $\mathbf{x_k}$ y observa referencias del entorno conocidas como *landmarks* o puntos de referencia $\mathbf{m_j}$. Las trayectorias reales y estimadas del robot se muestran diferenciadas, al igual que las mediciones de movimiento ($\mathbf{u_k}$) y observaciones sensoriales ($\mathbf{z_{k,j}}$). Este gráfico refleja la incertidumbre inherente al proceso y cómo el sistema ajusta su estimación de la trayectoria y el mapa mediante técnicas de

optimización, contribuyendo así a una navegación más precisa y robusta en entornos desconocidos.

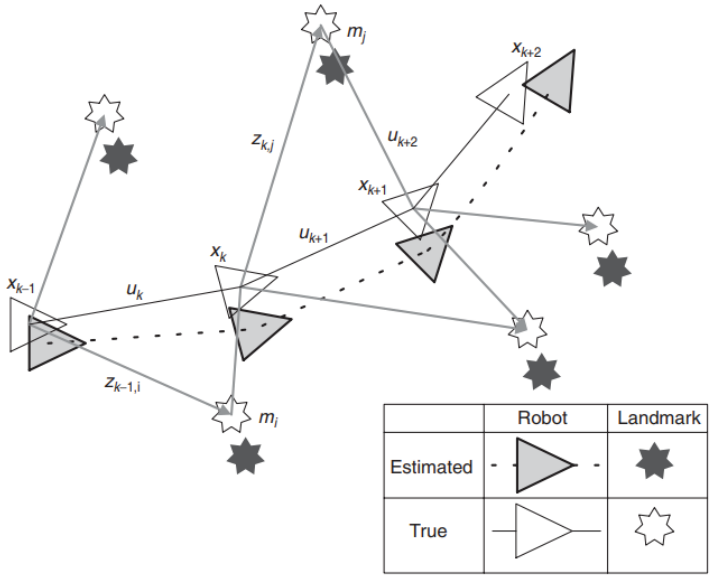

Ilustración 2 Esquema de funcionamiento de SLAM [36]

El motivo por el cual SLAM se ha consolidado como una de las técnicas más empleadas para alcanzar la autonomía radica en su capacidad de resolver simultáneamente dos problemas fundamentales en robótica móvil: la construcción de un mapa del entorno y la estimación precisa de la posición del robot dentro de ese mapa. En situaciones reales, especialmente en entornos desconocidos o dinámicos, no siempre se dispone de un plano previo del entorno ni de sistemas externos de localización como el GPS, por lo que es esencial que el propio robot sea capaz de generar esta información de manera autónoma. SLAM permite esa autosuficiencia, integrando datos en tiempo real provenientes de sensores heterogéneos y combinándolos mediante algoritmos de filtrado y optimización. Gracias a su flexibilidad y adaptabilidad, esta técnica se ha convertido en un estándar en aplicaciones de robótica de exploración, rescate, logística y servicios en interiores, donde la navegación fiable y contextualizada es imprescindible.

El SLAM combina información proveniente de múltiples sensores como cámaras RGB-D, LiDAR, IMUs [37] o sensores de ultrasonido para generar representaciones espaciales precisas del entorno. En el contexto de este trabajo, la cámara Azure Kinect permite la recolección de datos de profundidad, lo cual resulta ideal para ser empleado por sistemas de SLAM tridimensionales.

Dentro del ecosistema ROS, existen varios paquetes que permiten implementar SLAM, como gmapping [38], cartographer [39] o RTAB-Map. Este último ha sido el elegido en el

presente proyecto debido a su capacidad para realizar mapeo en 3D en tiempo real, su integración con ROS Noetic y su compatibilidad con sensores RGB-D como el Azure Kinect. RTAB-Map (Real-Time Appearance-Based Mapping) no solo permite generar mapas tridimensionales navegables, sino que además incorpora técnicas de detección de bucles (loop closure) y optimización de grafos, elementos esenciales para mejorar la precisión del mapa generado a lo largo del tiempo.

Gracias a SLAM y a herramientas como RTAB-Map, es posible dotar al robot de la capacidad de desplazarse de forma inteligente por entornos cambiantes, evitando obstáculos y planificando rutas óptimas sin necesidad de GPS. Esto es especialmente valioso en interiores o zonas con poca cobertura satelital, donde el mapeo visual se convierte en el principal medio de localización.

En suma, la integración de SLAM dentro del sistema de navegación autónoma del robot constituye una pieza clave para lograr una interacción fluida, segura y adaptativa con su entorno.

2.6 Sensores LiDAR

Los sensores LiDAR (Light Detection and Ranging) han sido durante años una de las tecnologías más empleadas para la generación de mapas tridimensionales precisos en robótica. Estos dispositivos funcionan emitiendo pulsos láser y midiendo el tiempo que tarda la luz en regresar al sensor después de impactar con los objetos del entorno. Esta técnica permite generar una nube de puntos extremadamente precisa y detallada, incluso a largas distancias.

Los LiDAR se utilizan ampliamente en vehículos autónomos, drones, robots móviles y sistemas de inspección industrial, gracias a su alta precisión, resistencia a condiciones de iluminación adversas y su capacidad para proporcionar un modelo fiel del entorno. Dependiendo de su arquitectura, pueden ser de tipo 2D (empleados en planos horizontales) o 3D (capaces de escanear en múltiples direcciones y planos).

A pesar de sus numerosas ventajas, los sensores LiDAR también presentan ciertas limitaciones, como el alto coste en comparación con cámaras RGB-D y la necesidad de procesamiento intensivo para interpretar sus datos. Por ello, en algunos casos se opta por su integración con otros sensores complementarios, como cámaras o IMUs, para enriquecer la percepción del entorno.

En el contexto de este proyecto, se decidió incluir un sensor LiDAR, concretamente el modelo RPLIDAR S2 [40], debido a que el uso exclusivo de la cámara Azure Kinect para realizar simultáneamente el mapeo y la navegación autónoma resultaba demasiado complejo para el marco temporal limitado de un Trabajo de Fin de Grado. La incorporación del LiDAR ha permitido simplificar notablemente la generación del mapa en tiempo real y la localización precisa dentro del entorno, complementando así los datos visuales proporcionados por la cámara. El RPLIDAR S2, gracias a su diseño compacto, su capacidad de escaneo 360° y su buena integración con ROS, se presenta como una solución eficaz y asequible para este tipo de aplicaciones de robótica móvil.

2.7 Cámaras ToF y RGB-D

Las cámaras basadas en tecnología Time-of-Flight (ToF) y RGB-D se han consolidado como herramientas clave en la percepción tridimensional del entorno, especialmente en robótica móvil y sistemas de navegación autónoma. Estas cámaras combinan la adquisición de color (RGB) con información de profundidad (D), lo que les permite capturar no solo las texturas visuales del entorno, sino también su geometría espacial.

Las cámaras ToF emiten una señal de luz infrarroja que rebota en los objetos del entorno, calculando el tiempo que tarda en regresar al sensor para estimar la distancia. Esta tecnología destaca por su alta precisión a corta distancia, así como por su capacidad de funcionar en condiciones de baja iluminación. Las cámaras RGB-D, por su parte, pueden utilizar distintas tecnologías de profundidad, como ToF, luz estructurada o visión estéreo, ofreciendo una integración completa entre imagen y distancia.

La Ilustración 3 representa el principio de funcionamiento de una cámara basada en tecnología Time-of-Flight (ToF). En el esquema se aprecia cómo el emisor (VCSEL) proyecta un haz de luz infrarroja hacia un objeto, el cual refleja dicha señal. El sensor receptor capta el rebote de esa señal, y mediante un chip de procesamiento de profundidad se calcula el tiempo que ha tardado en regresar, lo que permite estimar la distancia. Todo este proceso está coordinado por una arquitectura interna que incluye componentes ópticos de transmisión y recepción, gestión de energía, drivers láser y una capa de software que integra firmware, SDKs y aplicaciones. Esta arquitectura permite que la cámara proporcione información de profundidad precisa en tiempo real, incluso en condiciones de iluminación adversas, lo que la convierte en una herramienta esencial para tareas de percepción 3D en robótica móvil.

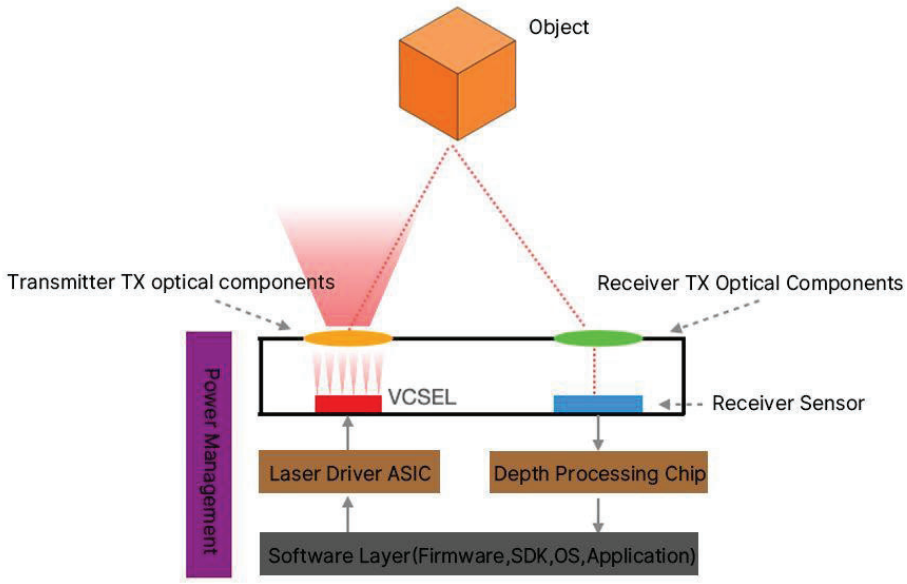

Ilustración 3 Esquema del funcionamiento de cámaras ToF [41]

En el ámbito de la robótica, estas cámaras se emplean en tareas como detección de obstáculos, navegación en interiores, SLAM tridimensional, interacción hombre-máquina y reconocimiento de objetos. Además, su compatibilidad con frameworks como ROS ha permitido una rápida adopción y experimentación en entornos de desarrollo reales.

Una de las cámaras más destacadas en esta categoría es la Azure Kinect, desarrollada por Microsoft. Este dispositivo integra una cámara RGB de alta resolución, un sensor de profundidad basado en ToF, un conjunto de micrófonos y sensores IMU, convirtiéndolo en una solución todo-en-uno para percepción espacial. La Azure Kinect se ha utilizado en diversos proyectos de investigación, incluyendo navegación robótica, modelado 3D y aplicaciones de visión artificial. En este trabajo, se optó por este dispositivo debido a su alta resolución, disponibilidad de drivers compatibles con ROS Noetic y la capacidad para ofrecer diferentes tipos de imágenes lo cual favorece la implementación de YOLO.

2.8 YOLO

En los últimos años, la detección de objetos en tiempo real ha avanzado considerablemente gracias a la aparición de modelos de redes neuronales convolucionales (CNN) [42] optimizados para velocidad y precisión. Uno de los algoritmos más destacados en este campo es YOLO (You Only Look Once), propuesto por Joseph Redmon en 2016. A diferencia de otros métodos de detección basados en regiones, como R-CNN [43] o SSD [44], YOLO realiza la detección en una sola pasada de red neuronal, lo que permite alcanzar altas velocidades de inferencia sin sacrificar significativamente la precisión.

El enfoque de YOLO divide una imagen en una cuadrícula y predice simultáneamente las clases de objetos y sus ubicaciones dentro de cada celda, utilizando regresión directa. Esto permite detectar múltiples objetos de distintas clases en una misma imagen con muy bajo retardo, convirtiéndolo en una herramienta especialmente útil para aplicaciones en vehículos autónomos, robótica móvil, vigilancia, drones, y sistemas de asistencia al conductor (ADAS).

La Ilustración 4 proporciona un claro ejemplo de cómo YOLOv7 opera en un escenario de tráfico real. Se pueden observar los cuadros delimitadores (bounding boxes) generados por el modelo, que identifican y clasifican diversos objetos como "car" (coche), "person" (persona) y "traffic light" (semáforo). Cada cuadro está acompañado de una etiqueta que indica la clase del objeto detectado y un valor numérico que representa la confianza de la predicción, como 0.92 para un coche o 0.90 para una persona. Este ejemplo visual demuestra la capacidad de YOLO para procesar escenas complejas con múltiples elementos dinámicos, un requisito fundamental para aplicaciones de visión por computadora en el ámbito de la robótica y los vehículos autónomos.

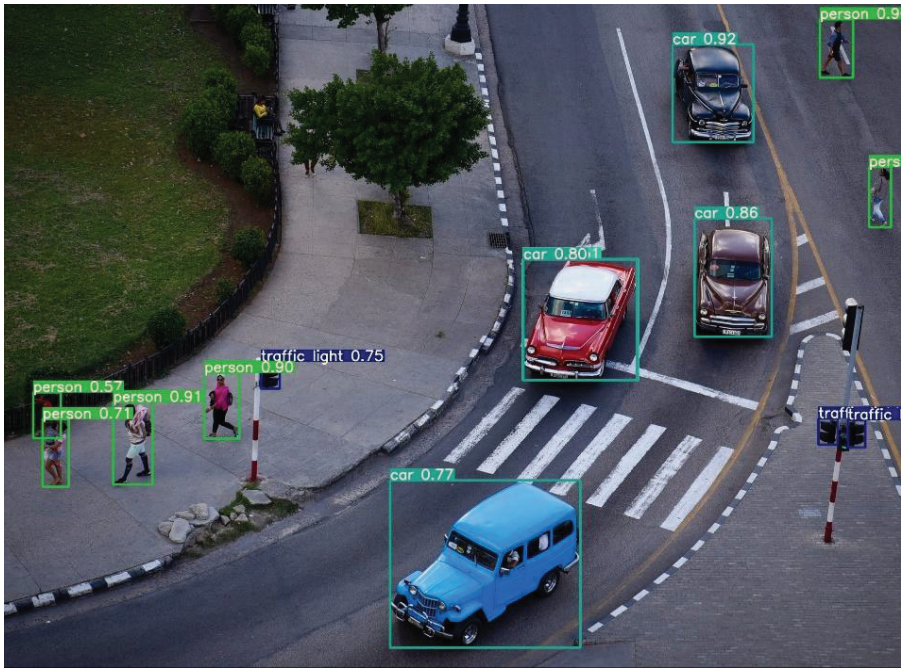

Ilustración 4 Ejemplo del funcionamiento de YOLO [45]

Con el tiempo, se han desarrollado múltiples versiones de YOLO, desde YOLOv1 hasta YOLOv8, cada una mejorando aspectos como la precisión, velocidad y compatibilidad con dispositivos de hardware limitados. Versiones como YOLOv4 y YOLOv5 destacaron por su facilidad de uso y adaptabilidad en proyectos de código abierto, mientras que YOLOv7, lanzada en 2022, se consolidó como una de las versiones más equilibradas entre rendimiento y eficiencia computacional.

YOLOv7 introduce mejoras en la arquitectura backbone y en los métodos de entrenamiento, lo que permite obtener una excelente precisión con tiempos de inferencia competitivos incluso en dispositivos con recursos limitados como una Jetson Nano. Esta versión además facilita la implementación de detección en tiempo real en combinación con frameworks como ROS, lo que permite integrarla de forma natural en sistemas de navegación autónoma.

En este proyecto se ha optado por utilizar YOLOv7 debido a su rendimiento sobresaliente en tareas de detección en tiempo real, su compatibilidad con CUDA/cuDN y su arquitectura flexible, que permite ajustar el modelo a distintos niveles de exigencia según el hardware disponible. Además, esta versión mantiene una cierta continuidad con el proyecto base de referencia, facilitando la adaptación del código y la integración con ROS Noetic.

Capítulo 3. Desarrollo del sistema

En este capítulo se presenta el desarrollo completo del sistema robótico, abarcando tanto la arquitectura general como los componentes físicos y su integración. Se describen las características técnicas de la plataforma Jetson Nano, los sensores utilizados, entre ellos la cámara Azure Kinect y el LiDAR RPLIDAR S2, así como el modelo de comunicación entre dispositivos. Además, se detallan los nodos implementados en ROS Noetic para la percepción, el mapeo y la navegación autónoma, destacando el uso de técnicas como SLAM y la detección de objetos mediante YOLOv7.

3.1 Arquitectura general del sistema

La arquitectura del sistema ha sido diseñada bajo un enfoque modular que prioriza la escalabilidad, el mantenimiento y la integración de nuevas funcionalidades. Cada módulo representa una unidad funcional independiente que se comunica con el resto del sistema a través de interfaces ROS bien definidas, lo cual permite una implementación desacoplada y reutilizable. Esta estrategia facilita la adaptación del sistema a distintos escenarios operativos, garantizando un desempeño robusto y flexible.

En el caso del presente proyecto, los módulos principales que conforman el sistema son: percepción, localización y mapeo, detección de objetos, control de movimiento, y navegación autónoma. Estos módulos se implementan y coordinan sobre una base física compuesta por una Jetson Nano, una cámara Azure Kinect DK, un LIDAR RPLIDAR S2, y un chasis con ruedas mecanum controladas por Arduino UNO [46]. La Jetson Nano centraliza la computación y el procesamiento de sensores, mientras que el Arduino se encarga del control de los motores a través de comunicación serial.

El módulo de percepción se fundamenta en la integración de la Azure Kinect y el LIDAR. La Azure Kinect publica datos RGB, profundidad y nube de puntos a través del nodo Azure_Kinect_ROS_Driver, permitiendo la adquisición en tiempo real de información visual y espacial del entorno. El LIDAR, por su parte, proporciona escaneos 2D que complementan la detección de obstáculos. Ambos sensores están sincronizados dentro del entorno ROS, generando un flujo continuo de datos que alimenta los módulos de mapeo, navegación y detección.

El módulo de localización y mapeo está basado en rtabmap_ros, un paquete de ROS que implementa SLAM visual y/o láser. Este módulo combina las imágenes de profundidad y RGB de la cámara con los escaneos del LIDAR para construir un mapa 2D/3D del entorno

mientras estima en tiempo real la posición y orientación del robot. Esto permite la navegación en entornos no estructurados, con capacidades de realineación y cierre de bucle.

El módulo de detección de objetos utiliza un nodo personalizado que ejecuta YOLOv7 optimizado para Jetson Nano con CUDA. Este nodo suscribe a los tópicos de imagen RGB de la Azure Kinect, realiza inferencias en tiempo real y publica las detecciones como mensajes ROS, incluyendo su posición proyectada en coordenadas del mapa mediante transformaciones TF y uso de la profundidad. Además, se visualizan mediante marcadores en RViz.

El módulo de control de movimiento consiste en un nodo ROS que recibe comandos de velocidad y los transforma en instrucciones específicas para las ruedas mecanum. Este nodo aplica la cinemática diferencial adecuada al chasis mecanum, calcula las velocidades individuales para cada motor y las envía a la placa Arduino Uno mediante comunicación serial, utilizando un protocolo simple y robusto.

El módulo de navegación autónoma está encargado de integrar los mapas generados por RTAB-Map con el estado de detección y posición para planificar rutas y ejecutar desplazamientos seguros. Utiliza planificación global y local con ROS Navigation Stack adaptada al robot mecanum, enviando comandos de velocidad al nodo de control. Puede modificar su trayectoria en tiempo real en respuesta a detecciones de obstáculos o cambios en el entorno.

Todos estos módulos están interconectados mediante ROS Noetic, corriendo en la Jetson Nano. La arquitectura permite el intercambio eficiente de información entre sensores, motores y nodos de decisión, facilitando la operación autónoma del sistema. La comunicación con el Arduino Uno se realiza por puerto Serial UART, mientras que la publicación y suscripción de datos entre nodos ROS permite una coordinación modular, extensible y mantenible.

3.2 Descripción hardware del vehículo autónomo

Para el desarrollo de este proyecto se ha utilizado una combinación de hardware que abarca desde componentes básicos hasta elementos de procesamiento y percepción más complejos. Dada la heterogeneidad de estos dispositivos, ha sido esencial organizarlos según su función específica dentro del sistema, lo que ha permitido una integración eficaz, coherente y adaptada a los requerimientos técnicos del proyecto.

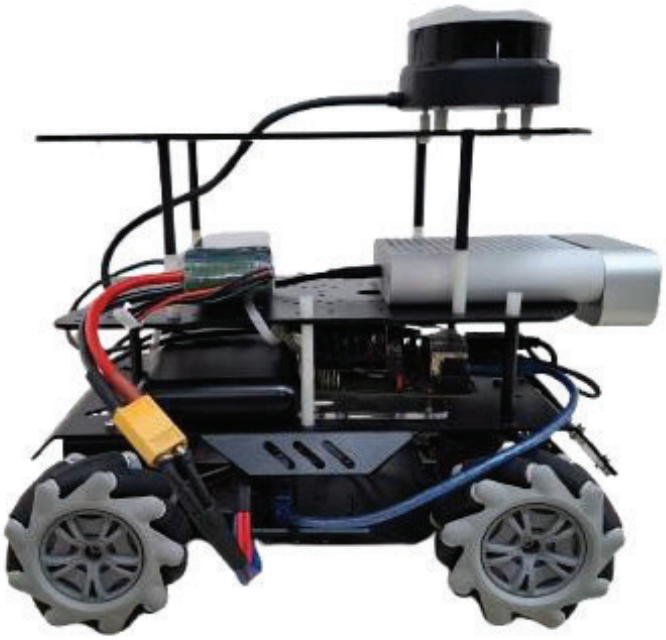

Ilustración 5 Vista general del vehículo

En la ilustración 5 se muestra el diseño final del vehículo, en el que se destacan los principales elementos físicos montados sobre su chasis. Cada uno de estos componentes ha sido ubicado de forma estratégica para maximizar la funcionalidad, la estabilidad y la eficiencia del sistema. El vehículo incorpora sensores de percepción avanzados como un sensor LiDAR, situado en la parte superior para captar de manera óptima la información del entorno, así como una cámara RGB conectada directamente a la Jetson Nano. La movilidad omnidireccional se consigue mediante el uso de ruedas mecanum acopladas a motores controlados a través de una placa Arduino. La unidad principal de procesamiento está representada por una Jetson Nano, que gestiona los procesos de percepción, detección, mapeo y navegación en tiempo real mediante el sistema operativo ROS Noetic.

El sistema de alimentación está dividido en dos fuentes de energía independientes: una batería dedicada a los motores y otra destinada al sensor LiDAR y a la Jetson Nano. Esta última, a su vez, provee alimentación directa a la cámara conectada, garantizando un funcionamiento estable y continuo del sistema de percepción.

El chasis del vehículo está estructurado en múltiples niveles, optimizados para distribuir adecuadamente los distintos módulos. En la base se encuentran los motores y las ruedas mecanum, asegurando una plataforma de desplazamiento sólida y se ubica la placa de control motriz (Arduino). En el siguiente nivel se encuentra la Jetson Nano y la batería que alimenta esta y el LiDAR. En el nivel intermedio, se encuentra la cámara y batería que alimenta la parte motorizada. Finalmente, en la parte superior del vehículo se localiza el LiDAR, cuya altura les permite obtener datos sin obstrucciones del entorno cercano. Esta disposición general favorece la percepción fiable del entorno y una navegación autónoma eficiente, incluso en escenarios dinámicos o con obstáculos.

3.3 Plataforma de desarrollo: Jetson Nano

Durante gran parte del desarrollo del sistema se utilizó una Jetson Nano disponible. Esta plataforma resultó útil para realizar pruebas iniciales, familiarizarse con el entorno de trabajo y adaptar los distintos módulos del sistema, gracias a su compatibilidad con Ubuntu 20.04 y ROS Noetic.

No obstante, pronto se detectaron serias limitaciones debidas a su hardware, ya que se trataba del modelo con 2 GB de memoria RAM, insuficiente para ejecutar simultáneamente procesos exigentes como la detección de objetos mediante YOLOv7, el mapeo SLAM y el procesamiento de imágenes provenientes de la cámara Azure Kinect. Estas restricciones impedían un funcionamiento fluido del sistema, ocasionando bloqueos del sistema operativo y una drástica reducción del rendimiento general.

Por este motivo, se optó por una actualización a la Jetson Orin Nano de 16 GB, una versión considerablemente más potente y adecuada para los requerimientos del proyecto. Esta nueva plataforma permitiría alcanzar un rendimiento estable en la ejecución concurrente de todas las tareas críticas del sistema, como se resume en la siguiente tabla:

Tabla 2 Especificaciones de Jetson Orin Nano (16 GB) [47]

Característica	Jetson Orin Nano (16 GB)
CPU	8 núcleos ARM Cortex-A78AE
GPU	NVIDIA Ampere: 1024 núcleos CUDA + 32 Tensor
Memoria RAM	16GB 128-bit LPDDR5 102.4GB/s
Almacenamiento	128GB NVMe SSD + Tarjeta microSD
Aceleración por hardware	CUDA, cuDNN, TensorRT
Sistema operativo compatible	Ubuntu 20.04 / JetPack SDK
Rendimiento en IA	157 TOPS
Conectividad	GPIO, I2C, SPI, UART, CSI, USB 3.0

Si bien existen otras plataformas de desarrollo como la Raspberry Pi 4 [48], la BeagleBone AI [49] o incluso controladoras más específicas como las basadas en STM32 [50] para tareas embebidas, la elección de la Jetson Nano, y posteriormente su evolución a Jetson Orin Nano, ofreció una solución intermedia entre potencia y simplicidad de integración. Frente a opciones como la Raspberry Pi, se gana en capacidad de cómputo para tareas de inteligencia artificial y visión por computador gracias a su GPU integrada, aunque a costa de un mayor consumo energético y complejidad de configuración. Comparado con plataformas industriales más avanzadas, como las basadas en NVIDIA Xavier o PCs embebidos, se sacrifica rendimiento absoluto, pero se gana en accesibilidad, tamaño reducido y coste contenido, lo cual encaja con los objetivos y limitaciones de un Trabajo Fin de Grado.

Sin embargo, la Jetson Orin Nano no llegó a tiempo para ser integrada en el desarrollo efectivo del sistema, lo que supuso una limitación clave del proyecto. Esta situación impidió comprobar de forma adecuada el funcionamiento conjunto de todos los módulos planificados, y condicionó la validez de algunos resultados que no pudieron ser evaluados bajo las condiciones óptimas previstas.

3.4 Cámara Azure Kinect

La Azure Kinect DK es una cámara avanzada desarrollada por Microsoft, equipada con sensores que permiten capturar tanto imágenes en color (RGB) como datos de profundidad (Depth). Gracias a su diseño orientado a aplicaciones de inteligencia artificial, visión por computador y robótica, esta cámara ha sido empleada en el presente proyecto como principal fuente de percepción del entorno.

La Azure Kinect proporciona simultáneamente varios tipos de datos:

- Imagen RGB de alta resolución (hasta 3840x2160 píxeles), útil para tareas de visión como la detección de objetos mediante modelos como YOLOv7.
- Mapa de profundidad (Depth Map), basado en tecnología Time-of-Flight (ToF), que permite calcular la distancia a cada punto del entorno.
- Nube de puntos 3D, generada a partir de los datos de profundidad, útil para mapeo y navegación.
- IMU integrada, que ofrece datos de aceleración y rotación, aunque en este proyecto no ha sido utilizada de forma activa.

Esta combinación de sensores permite realizar simultáneamente tareas de detección de objetos y generación de mapas del entorno tridimensional.

Para integrar la cámara Azure Kinect en el entorno ROS Noetic, se utilizó el paquete oficial Azure_Kinect_ROS_Driver, el cual permite publicar los diferentes tipos de datos en tópicos ROS estándar. Fue necesario compilar el SDK de Azure Kinect y adaptar algunos parámetros en los archivos de lanzamiento para que funcionaran correctamente en la plataforma Jetson. Se configuraron los siguientes elementos clave:

- Resolución de la imagen RGB y profundidad.
- Sincronización entre las imágenes RGB y el mapa de profundidad, para poder fusionar ambos en algoritmos de visión o SLAM.
- Publicación de los datos en tiempo real, permitiendo su uso por parte de otros nodos del sistema.

Gracias a esta integración, la cámara pudo ser utilizada como fuente principal para los módulos de detección de objetos y mapeo.

Frente a otras cámaras similares como la Intel RealSense D435i [51] o la Orbbec Astra [52], la Azure Kinect ofrece varias ventajas clave:

- Mayor resolución y precisión en los datos RGB y de profundidad.
- Rango de detección más amplio, lo que la hace adecuada para entornos más grandes o espacios interiores con obstáculos.
- Calibración más estable y calidad óptica superior.
- Mejor integración en aplicaciones con redes neuronales, gracias al formato de datos compatible y soporte en plataformas de IA como Jetson.

Estas características hicieron que fuera una opción especialmente adecuada para este proyecto, donde se requería una cámara capaz de proporcionar datos fiables y detallados en tiempo real, tanto para la navegación autónoma como para la detección de objetos mediante visión artificial.

Sin embargo, durante el proceso de integración se detectaron algunas complicaciones técnicas importantes. El uso de una arquitectura ARM como la de Jetson Nano y Orin obligó a emplear versiones específicas de ciertas librerías del SDK oficial, lo que limitaba la compatibilidad con versiones más recientes. Además, una de las funciones destacadas del dispositivo, el body tracker, generaba conflictos con otras bibliotecas del sistema dependiendo de su versión, llegando a provocar errores críticos si no se usaban exactamente las versiones compatibles. Esta doble restricción impuso una única combinación funcional de librerías, que tuvo que ser identificada tras diversas pruebas y búsquedas en documentación no oficial y foros técnicos.

3.5 LiDAR RPLIDAR S2

El RPLIDAR S2 es un sensor de escaneo láser 2D desarrollado por SLAMTEC [53], diseñado específicamente para tareas de mapeo y navegación en entornos móviles. Este tipo de sensor es ampliamente utilizado en robótica debido a su precisión, velocidad de escaneo y facilidad de integración con sistemas como ROS.

El RPLIDAR S2 utiliza una técnica de medición láser por tiempo de vuelo (ToF), escaneando en un plano horizontal con una frecuencia de rotación de hasta 15 Hz. El sensor cubre un campo de visión de 360 grados con un rango de detección de hasta 30 metros en condiciones ideales, y genera hasta 32.000 muestras por segundo, lo que le permite capturar detalles finos del entorno.

Gracias a su resolución angular mejorada respecto a versiones anteriores (como el A1 o A2), ofrece una reconstrucción más precisa de los contornos y obstáculos, lo que se traduce en mapas más detallados y fiables.

El LIDAR es un componente fundamental en los sistemas de localización y mapeo simultáneo (SLAM). Su capacidad para generar mapas en tiempo real basados en las distancias a obstáculos lo hace especialmente útil para:

- Generar mapas 2D del entorno en interiores.
- Detectar obstáculos a corta y media distancia.
- Calcular trayectorias libres de colisión para la navegación autónoma.
- Corregir la posición del robot cuando se combina con algoritmos de odometría visual o inercial.

Además, el uso del LIDAR permite una mayor precisión en el levantamiento de mapas y facilita el movimiento autónomo, ya que proporciona información completa del entorno en 360 grados, y no únicamente en la dirección hacia la que apunta la cámara. Esto resulta clave para la navegación robusta en espacios complejos o dinámicos.

La integración del RPLIDAR S2 en ROS se realizó mediante el paquete oficial rplidar_ros, el cual permite publicar los datos del escáner láser en el tópico /scan, ampliamente compatible con nodos de mapeo y navegación como gmapping, hector_slam o rtabmap_ros.

3.6 Placa controladora y robot motorizado

Para el control de los motores del robot se ha utilizado una placa controladora QGPMarker MotorShield v5.3, la cual actúa como controlador intermedio entre los motores y un Arduino Uno R3, que viene integrado en el sistema motorizado adquirido. Esta placa está basada en el controlador TB6612FNG [54], ampliamente utilizado en proyectos de robótica por su capacidad para manejar múltiples motores de corriente continua (DC) de forma eficiente.

La Ilustración 6 muestra un esquema detallado del QGPMarker MotorShield v5.3, resaltando sus principales componentes y conexiones. Se pueden identificar claramente los terminales para la conexión de hasta cuatro motores, etiquetados como "Motor 1" a "Motor 4", junto con sus respectivas conexiones de tierra (GND) y los pares de pines para la señal de control (A y B). Asimismo, la imagen indica las conexiones para los "Servos", lo

que sugiere la capacidad de la placa para controlar también este tipo de actuadores. Es crucial notar la sección de "Alimentación de los motores" y las opciones "Abierto: Alimentación externa a los motores" y "Cerrado: Arduino alimenta a los motores", lo que subraya la flexibilidad de la placa para ser alimentada de forma independiente, algo vital para manejar las demandas de corriente de los motores sin sobrecargar el Arduino. Esta representación visual facilita la comprensión de la arquitectura de control de los motores y su integración en el sistema robótico.

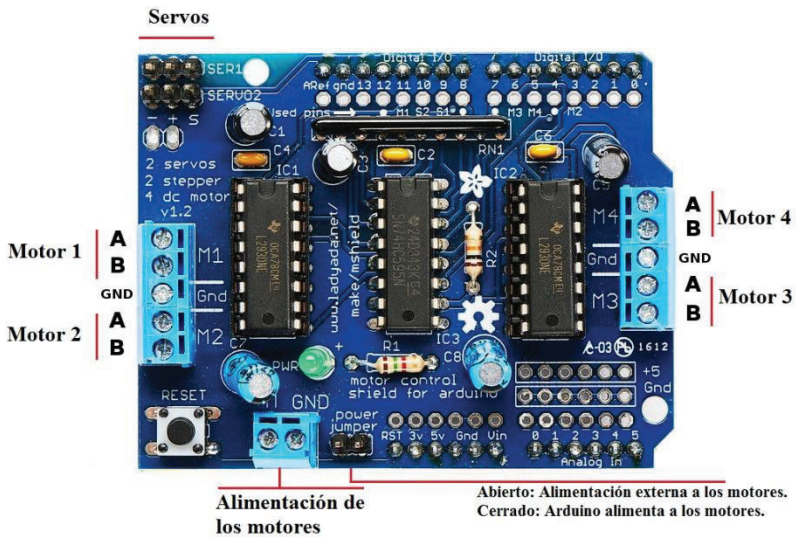

Ilustración 6 QGP Marker Motor Shield v5.3

El QGPMarker MotorShield v5.3 permite el control simultáneo de hasta cuatro motores DC, gracias a sus dos puentes H TB6612FNG. Cada canal puede proporcionar una corriente continua de aproximadamente 1.2A con picos de hasta 3A, permitiendo un rendimiento adecuado para plataformas móviles de tamaño medio. La placa soporta tensiones de alimentación separadas para la lógica y los motores, ofreciendo flexibilidad y protección en la operación.

La comunicación entre la Jetson Nano y el sistema de control se realiza a través del Arduino Uno R3, el cual recibe comandos desde ROS mediante una conexión serial. Para ello, se ha desarrollado un nodo ROS en Python que publica instrucciones de velocidad y dirección en un tópico específico, que posteriormente son traducidas y enviadas al Arduino a través del puerto /dev/ttyUSB0.

El Arduino interpreta estos comandos y genera las señales PWM necesarias para controlar los motores a través de la placa. Este diseño modular permite mantener ROS como sistema de control de alto nivel, mientras el Arduino gestiona el control físico de los motores a bajo nivel.

El robot implementa una configuración 4WD con ruedas mecanum [55], lo que le proporciona movilidad omnidireccional. Gracias a la disposición en ángulo de los rodillos de estas ruedas, el sistema puede desplazarse no solo hacia adelante y atrás o girar sobre su eje, sino también deslizarse lateralmente o realizar movimientos diagonales. Esta característica es especialmente útil para la navegación en espacios reducidos o con obstáculos, ya que permite maniobras precisas sin necesidad de realizar giros amplios.

3.7 Nodos de percepción

El sistema de percepción del robot se encarga de procesar y combinar la información de los sensores visuales y de distancia para permitir la detección de objetos, el mapeo del entorno y la navegación autónoma. Como se ilustra en la Ilustración 7, los nodos clave incluyen el Azure_Kinect_ROS_Driver para datos de color y profundidad, el rplidar_ros para la información LiDAR, y el rtabmap_ros que integra estos datos junto con la odometría para la construcción del mapa. Complementariamente, se incorpora un nodo de detección en tiempo real basado en YOLOv7. A continuación, se detallan el funcionamiento y la configuración de cada uno de estos componentes esenciales.

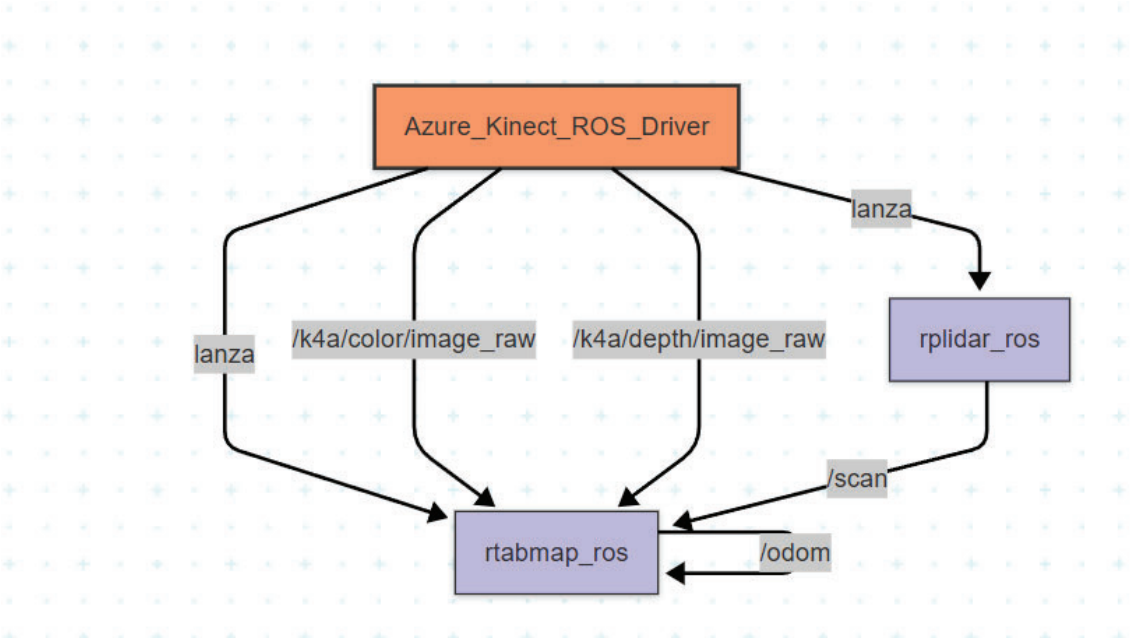

Ilustración 7 Enlace entre los nodos de precepción

3.7.1 Nodo Azure_Kinect_ROS_Driver

La cámara Azure Kinect DK se integró en el sistema mediante el nodo proporcionado por el paquete Azure_Kinect_ROS_Driver, que actúa como puente entre el SDK oficial del dispositivo y el entorno ROS. Este nodo tiene como objetivo principal publicar en tiempo real los datos de imagen, profundidad y nube de puntos para ser consumidos por otros nodos del sistema, como los encargados de mapeo, detección o navegación.

El nodo se ejecuta a través del archivo de lanzamiento driver.launch, ubicado dentro del paquete, y es el encargado de inicializar los sensores y establecer los parámetros necesarios para su funcionamiento. El script principal que gestiona este nodo es el archivo node/k4a_ros_device_node.cpp, el cual instancia el dispositivo, configura el modo de captura y lanza los tópicos de ROS. Este nodo se compila como un ejecutable llamado k4a_ros_device_node, definido en el CMakeLists.txt del paquete.

Los principales tópicos publicados por este nodo son:

- /k4a/rgb/image_raw: imagen RGB capturada por la cámara.
- / k4a /depth/image_raw: imagen de profundidad sin procesar.
- / k4a /pointcloud: nube de puntos generada a partir de los datos de profundidad y color.
- /tf: transformaciones entre los distintos marcos de referencia de la cámara.

El archivo de configuración utilizado para definir el comportamiento del nodo fue params/params.yaml, donde se especificaron parámetros clave como:

- depth_enabled: true
- color_enabled: true
- point_cloud: true
- sensor_sn: número de serie de la cámara (opcional, si hay múltiples dispositivos)
- camera_fps: 15, para reducir la carga computacional en la Jetson Nano
- rgb_resolution: 720P, para reducir la carga computacional en la Jetson Nano

El archivo launch/k4a.launch fue modificado para adaptarse a las necesidades del sistema, deshabilitando el uso del micrófono y ajustando los parámetros mencionados. También se creó otro script basado en el principal llamado slam_rtabmap.launch, pero que cumpliera las necesidades para nuestro proyecto y que pudiera lanzar a su misma vez el nodo rtabmap_ros y el del LiDAR, para ello se tuvieron que remapear tópicos y facilitar su integración con otros módulos de ROS.

En ejecución, el nodo se encarga de abrir el dispositivo Azure Kinect, configurar los sensores según los parámetros definidos y comenzar la publicación continua de los datos. Además, gestiona la calibración intrínseca de la cámara y la transforma en parámetros compatibles con los mensajes ROS estándar (sensor_msgs::CameraInfo), lo que facilita la posterior proyección de coordenadas 3D.

Esta implementación permite centralizar el acceso a los datos del sensor en un único nodo confiable, modular y reutilizable, cumpliendo con las exigencias del sistema y sirviendo como base para múltiples funcionalidades de percepción en el robot.

3.7.2 Nodo rplidar_ros

El nodo del plidar_ros se lanza en paralelo desde el nodo de la cámara, publica los datos de escaneo láser en el tópico estándar (/scan). Este nodo también es compatible de forma directa con ROS gracias a los drivers disponibles, y su salida es aprovechada por RTAB-Map para complementar la información visual y así mejorar la precisión en el levantamiento de mapas.

Cabe destacar que el nodo del plidar_ros es especialmente simple, y solo fue necesario realizar una ligera modificación en el script principal para permitir su integración con el resto del sistema. Esto facilitó considerablemente su puesta en marcha dentro del flujo de trabajo general.

Ambos sensores (Azure Kinect y LiDAR) se han configurado para comunicarse con RTAB-Map y proporcionar datos de forma simultánea, haciendo posible una fusión multisensorial durante el proceso de SLAM. Todo el sistema se lanza desde el mismo nodo de inicio de la cámara, lo que garantiza una sincronización correcta entre los diferentes elementos.

3.7.3 Nodo yolov7

Para la detección de objetos en tiempo real, se ha integrado un nodo basado en YOLOv7, adaptado para ejecutarse en la Jetson Nano. Este nodo ha sido configurado para suscribirse al tópico que publica la imagen RGB (/k4a/rgb/image_raw) de la Azure Kinect y procesarla en tiempo real, detectando clases de interés previamente entrenadas.

El modelo de YOLOv7 fue entrenado fuera de la Jetson Nano, en un ordenador con especificaciones de hardware superiores, lo que permitió realizar el entrenamiento de forma óptima y eficiente, evitando las limitaciones de rendimiento de la Jetson Nano. Para ello, se preparó un conjunto de datos compuesto por imágenes reales de entornos interiores, extraídas de la base de datos Open Images [56] para las siguientes clases: puerta, escritorio, mesa, archivador, estantería, encimera, papelera, escaleras, macetero, silla, sofá y persona.

Estas clases fueron seleccionadas por representar elementos comunes en ambientes interiores, que es el entorno principal en el que se enfoca este proyecto. Aunque sería posible ampliar la lista de clases o adaptar el modelo a otros contextos (como exteriores o entornos industriales), se ha optado por una colección razonable que permita trabajar de forma fluida en los recursos disponibles y centrarse en un entorno concreto.

La detección de objetos genera una salida con las coordenadas y clases de los objetos detectados, lo cual puede ser utilizado para la toma de decisiones o como información complementaria para la navegación y la interacción con el entorno. Además, si se requiere, puede integrarse con el sistema de transformaciones espaciales (TF) para conocer la ubicación relativa de los objetos detectados respecto al robot.

Durante el proceso de entrenamiento, se observó que, aunque la descripción general del entorno mediante YOLO ofrecía una funcionalidad valiosa, esta se desviaba ligeramente del objetivo principal del proyecto: la detección de barreras arquitectónicas. Por esta razón, se planteó un nuevo entrenamiento que incluyera no solo los obstáculos, como escaleras, sino también sus posibles soluciones, como rampas y ascensores. Sin embargo, al analizar la base de datos Open Images utilizada originalmente, se comprobó que no contenía imágenes etiquetadas adecuadamente para estas dos nuevas categorías. Para solventar este inconveniente, se recurrió a la plataforma Roboflow [57], que alberga proyectos ya preparados para el entrenamiento de redes neuronales, y desde la cual se pudieron extraer conjuntos de datos en formato YOLOv7 con imágenes específicas de rampas y ascensores, que fueron integradas al conjunto de clases del modelo.

Además, se valoró la posibilidad de entrenar individualmente cada una de las categorías relevantes para la detección de barreras arquitectónicas y sus posibles soluciones. Sin embargo, este enfoque introdujo una complicación que no se había previsto inicialmente: el sistema estaba diseñado para ejecutar un script en paralelo durante la navegación, capturando imágenes cada dos segundos para analizarlas con un único modelo de YOLO. Al aumentar el número de modelos —uno por clase o por grupo de clases—, incluso utilizando la Jetson Orin Nano, se alcanzaban los límites de carga computacional, lo que imposibilitaba realizar la inferencia en tiempo real. Como solución, se desarrolló un nuevo script basado en un enfoque diferido, en el que las imágenes se almacenan periódicamente y se analizan posteriormente, cuando la carga del sistema lo permite. Este cambio permitió mantener una mayor eficiencia y asegurar el rendimiento global del sistema durante la navegación.

La Ilustración 8 muestra un ejemplo real de la detección de una escalera mediante el modelo YOLOv7. En la imagen, se observa una escalera, y el modelo ha logrado identificarla correctamente. Es importante destacar que el cuadro delimitador (bounding box) que YOLO generó originalmente para señalar la escalera se ha remarcado en esta ilustración para mejorar su visibilidad, ya que en la salida original de YOLO podría ser poco perceptible. Esta capacidad de detección de elementos específicos es crucial para el objetivo del proyecto de identificar barreras arquitectónicas y sus soluciones, permitiendo al robot planificar rutas accesibles.

Ilustración 8 Modelo de YOLOv7 entrenado solo en escaleras.

3.7.4 Nodo rtabmap_ros

El nodo rtabmap_ros se encarga de procesar la información proveniente de los sensores para realizar el mapeo y la localización simultánea (SLAM). Este nodo recibe datos tanto de la cámara Azure Kinect como del LiDAR, fusionando la información visual y de escaneo láser para construir mapas tridimensionales precisos y robustos del entorno.

El nodo rtabmap_ros es modular y está bien integrado con ROS, lo que facilitó su implementación dentro del sistema. Se realizó una configuración mínima en el archivo de lanzamiento principal para adaptarlo a las características específicas de los sensores y asegurar una comunicación fluida con ambos dispositivos. Esta sencillez en la integración permitió un despliegue rápido y eficiente en el flujo de trabajo general.

Para que el sistema funcione correctamente, es fundamental realizar las transformaciones de referencia temporales y espaciales (TF) necesarias entre los distintos marcos de referencia de los sensores y el robot. Estas transformaciones garantizan que los datos de los distintos sensores estén correctamente alineados y sincronizados en el espacio y tiempo, permitiendo una fusión multisensorial precisa y la generación de mapas coherentes. La gestión de estas transformaciones se ve significativamente asistida por la integración de un modelo URDF [58] (Unified Robot Description Format) del robot. Este modelo, definido en un archivo XML, describe la geometría, cinemática y propiedades físicas del robot, incluyendo la posición y orientación exactas de todos sus componentes y sensores. Esto permite que herramientas de visualización como RViz representen el robot de forma precisa y que los algoritmos de SLAM, como rtabmap_ros, comprendan la relación espacial entre los datos de los diferentes sensores y la propia estructura del robot, optimizando la precisión del mapeo y la localización.

La Ilustración 9 presenta un ejemplo visual de cómo se definen las transformaciones en un modelo URDF para un robot de base móvil. En la imagen, se aprecian los distintos marcos de referencia (o "frames") asociados a cada componente del robot, como base_link (el origen principal), left_wheel_frontside, right_wheel_backside, etc. Cada uno de estos marcos está representado por un sistema de ejes XYZ (rojo, verde, azul respectivamente), indicando su posición y orientación en el espacio tridimensional. Las líneas que conectan estos marcos (base_link con cada rueda, por ejemplo) representan las uniones definidas en el URDF, que especifican la relación espacial y el tipo de movimiento permitido entre los componentes. Esta representación es vital para la correcta interpretación de los datos de los sensores en relación con el robot y para asegurar que la navegación y el mapeo se realicen con la máxima coherencia espacial.

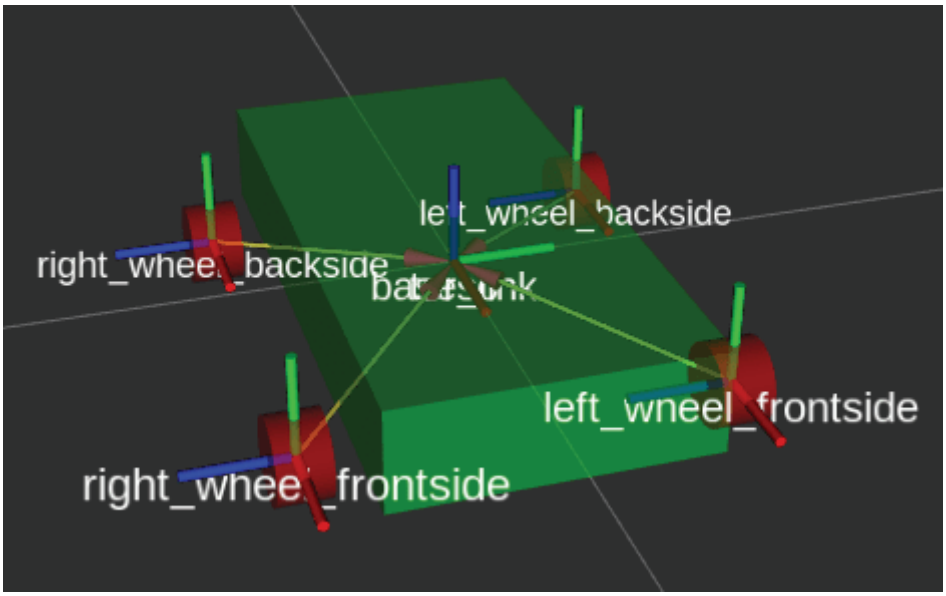

Ilustración 9 Ejemplo de modelo URDF [59]

Gracias a este nodo, se consigue obtener un mapa actualizado y detallado del entorno. Sin embargo, es importante destacar que el encargado de generar y gestionar el uso de dicho mapa es el nodo de conducción autónoma, ya que esta información es vital para que el robot pueda navegar de forma segura y eficiente dentro del espacio.

Además, rtabmap_rosse lanza junto con el nodo de la cámara, asegurando una sincronización adecuada entre la adquisición de datos y el procesamiento, lo que es fundamental para mantener la coherencia temporal de los mapas generados y mejorar la precisión en la navegación autónoma.

3.7.5 Nodo barrier_map

El nodo barrier_map es el componente central que cumple la finalidad principal de este Trabajo Fin de Grado. Su objetivo es generar un mapa integrado y enriquecido que combine la información espacial obtenida por el nodo de conducción autónoma (basado en RTAB-Map) con los datos de detección de objetos y barreras arquitectónicas proporcionados por el nodo de yolov7.

Este nodo se encarga de recibir el mapa final generado cuando la conducción autónoma termina, así como las coordenadas y clasificaciones de objetos detectados en tiempo real. A partir de esta información, sincroniza y fusiona ambos tipos de datos para construir un mapa simbólico único, en el que no solo se representa la geometría y la estructura del entorno, sino que también se marcan explícitamente las barreras arquitectónicas

identificadas (como escaleras, puertas estrechas, etc.) y sus posibles soluciones o elementos alternativos (por ejemplo, rampas de acceso o ascensores).

Con el fin de que el nodo funcionara independientemente del modo de ejecución de YOLO (a tiempo real o con varias capturas una vez terminado el mapeo), este nodo se ejecuta cuando finaliza la ejecución de YOLO. Para ambos casos, el script diseñado revisa la carpeta donde se generan los resultados de YOLO, contabiliza cuántos archivos hay (en caso de detección en tiempo real, suele haber uno; en el otro modo, una por cada peso entrenado) y analiza todas las detecciones presentes de forma secuencial, implementándolas en el mapa simbólico. Como es obvio, dado que puede haber conflictos de detección (por ejemplo, que el mismo objeto sea detectado dos veces si aparece en capturas desde distintos ángulos, o errores que provoquen detecciones redundantes), se desarrolló un sistema que mide la densidad de objetos detectados en una zona concreta y elimina aquellos que son redundantes, evitando duplicados.

El mapa simbólico resultante facilita una comprensión más elevada y semántica del entorno, permitiendo que el robot o cualquier sistema asociado tome decisiones más informadas respecto a la navegación, accesibilidad y adaptación a las limitaciones del espacio. Esto es especialmente útil para entornos cerrados, donde la identificación clara de las barreras arquitectónicas y las vías de solución es crucial para garantizar una movilidad segura y eficiente.

Para asegurar la correcta sincronización de datos, el nodo barrier_map trabaja en conjunto con las transformaciones TF y utiliza los tiempos de estampado (timestamps) de los mensajes para alinear espacial y temporalmente la información recibida. De esta forma, se logra que el mapa final refleje con precisión tanto la distribución física del espacio como la presencia y ubicación exacta de los obstáculos y soluciones detectadas.

En resumen, el nodo barrier_map constituye la capa de interpretación avanzada del sistema, convirtiendo la información sensorial y de detección en un recurso visual y funcional que permite evaluar y superar las barreras arquitectónicas dentro del ámbito del proyecto.

Capítulo 4. Desarrollo del Sistema de Percepción y Navegación Autónoma

En este capítulo se aborda el diseño e implementación del sistema de percepción y navegación autónoma dentro del entorno ROS. Se detalla la organización del workspace y la configuración de los paquetes desarrollados, así como la creación e integración de los nodos responsables de la gestión de sensores, el procesamiento de datos y el control del robot. Además, se explica el flujo general de ejecución del sistema, desde su arranque hasta la puesta en marcha del desplazamiento autónomo con detección de obstáculos, describiendo cómo interactúan los distintos componentes para lograr una operación coordinada.

4.1 Nodo depth_movement

Este nodo constituye, junto con el de la cámara, uno de los elementos principales del sistema, ya que en él se toman todas las decisiones necesarias una vez el dispositivo se encuentra en funcionamiento. Su importancia radica en que centraliza el procesamiento de datos de los sensores y ejecuta las acciones de navegación en tiempo real.

Como se ha mencionado anteriormente, este nodo cuenta con dos modos de operación diferenciados que se verán en detalle en el apartado 4.3. Por ello, en primer lugar, se describe su funcionamiento general, común a ambos modos, y posteriormente se detallan las particularidades de cada uno.

Al iniciarse, el nodo carga las librerías esenciales para su ejecución, entre ellas pyransac3d [63], una biblioteca especializada en el ajuste robusto de modelos geométricos tridimensionales mediante el algoritmo RANSAC [61]. Esta herramienta permite trabajar con nubes de puntos generadas por los sensores, facilitando la segmentación de planos y la detección de obstáculos relevantes para la navegación.

A continuación, el nodo se conecta a los tópicos de ROS previamente lanzados en paralelo por el nodo Azure_Kinect_ROS_Driver, accediendo así a la información de sensores como la cámara y el LiDAR. Una vez establecida esta comunicación, se procede a la definición de múltiples parámetros clave para la conducción autónoma.

Entre estos parámetros se encuentran los umbrales de seguridad, que definen las distancias mínimas a partir de las cuales el vehículo debe detenerse o reducir su velocidad. También se configuran valores que regulan dinámicamente la velocidad del robot en

función del entorno, así como estados internos que permiten llevar un seguimiento preciso tanto del proceso de mapeo como del estado operativo de la navegación (por ejemplo, estados como "STOP" o "INITIAL_SCAN").

Asimismo, se incluyen variables orientadas a optimizar la precisión de los datos sensoriales, ya sea del LiDAR o de la cámara, y se establece la configuración específica para el funcionamiento de YOLO. Esta configuración varía según el formato de uso, pero siempre incluye parámetros como el timestamp, las dimensiones de las imágenes capturadas, y la información sobre el peso o pesos del modelo neuronal que se utilizarán para la detección de objetos.

En conjunto, este nodo permite al sistema interpretar el entorno, planificar movimientos y adaptarse a situaciones diversas de forma autónoma, constituyendo el núcleo funcional del comportamiento inteligente del robot.

4.2 Movimiento autónomo

El movimiento autónomo del robot comienza con un escaneo inicial del entorno en el que se encuentra. Durante esta primera fase, el sistema registra la posición inicial como referencia, lo que permitirá más adelante retornar a ella una vez completada la exploración. Tras finalizar este escaneo, el robot selecciona de forma autónoma la dirección hacia la cual desplazarse, aplicando una serie de decisiones anidadas.

El proceso de decisión parte del análisis de los obstáculos circundantes. En primer lugar, se calcula la distancia a cada obstáculo y se selecciona la dirección que ofrezca mayor amplitud libre. Esta elección, sin embargo, no se basa únicamente en los datos del LiDAR. La cámara RGB-D, gracias a su campo de visión más amplio y su ubicación física por debajo del plano del LiDAR, complementa el análisis mediante el uso de la técnica RANSAC (Random Sample Consensus). Esta técnica permite ajustar modelos geométricos sobre los puntos obtenidos, siendo útil para detectar planos inclinados, escalones o desniveles que podrían representar caídas. Además, la cámara facilita la detección de objetos bajos que no son visibles para el LiDAR, lo que contribuye a evitar colisiones con elementos por debajo de su línea de visión.

Una vez seleccionado el camino, el proceso de toma de decisiones se mantiene en ejecución continua durante todo el recorrido del robot. A medida que el sistema genera el mapa con los datos del LiDAR, evalúa constantemente el entorno hasta que no se detecten nuevas áreas sin explorar (este comportamiento se encuentra definido en el código mediante una condición específica que evalúa la cobertura del mapa). Cuando esto ocurre,

el robot detiene su movimiento, lanza los procesos adicionales del sistema y guarda el mapa generado en una ruta específica, nombrado mediante un identificador basado en el timestamp del momento de finalización. Como medida de seguridad, el sistema también va guardando copias temporales del mapa en intervalos regulares durante la exploración, para evitar la pérdida de información en caso de interrupciones inesperadas.

La Ilustración 10 presenta un fragmento de código Python, específicamente la función check_map_coverage(self), que es la encargada de determinar cuándo el mapeo del entorno ha alcanzado un nivel de cobertura suficiente para finalizar la exploración. Esta función calcula el porcentaje de celdas "conocidas" dentro del mapa total (self.map_data.data), donde una celda conocida se define como aquella que no tiene el valor -1 (generalmente, indicando un estado desconocido o no explorado). El resultado de este cálculo se almacena en self.map_coverage y se imprime por consola para seguimiento (rospy.loginfo). Crucialmente, la función verifica si self.map_coverage es mayor o igual que un umbral predefinido (self.target_coverage). Si esta condición se cumple, se considera que se ha alcanzado la cobertura objetivo, se guarda el mapa (self.save_map()) y la función retorna True, señalizando el fin del proceso de mapeo autónomo.

```python
def check_map_coverage(self):

    """Check if the map has reached target coverage"""
    if self.map_data is None:
        return False

    # Calculate coverage
    total_cells = len(self.map_data.data)
    known_cells = sum(1 for cell in self.map_data.data if cell != -1)
    self.map_coverage = known_cells / total_cells if total_cells > 0 else 0
    rospy.loginfo(f"Map coverage: {self.map_coverage:.2%}")

    if self.map_coverage >= self.target_coverage:
        rospy.loginfo("Target map coverage reached!")
        self.save_map()
        return True
    return False
```

Ilustración 10 Código encargado de dictaminar el final del mapeo

El sistema contempla además la posibilidad de que el robot se quede bloqueado en un entorno saturado de obstáculos. Para gestionar esta situación, se implementa una lógica que cuenta la cantidad de obstáculos detectados en un breve intervalo temporal. Si el robot no puede avanzar, ejecuta un retroceso automático hasta alcanzar una posición más favorable desde la que continuar el proceso de exploración.

Todos los movimientos del robot se gestionan a través de publicaciones a los tópicos ROS "/robot/move/raw" (para la velocidad) y "/robot/move/direction" (para la dirección). Estos comandos son procesados por un nodo específico llamado motor_control, que se ejecuta en paralelo. Su única función es escuchar estos tópicos y enviar las órdenes correspondientes al microcontrolador Arduino encargado del control físico del vehículo. Este nodo también incorpora mecanismos de seguridad ante posibles fallos del nodo depth_movement, así como medidas para evitar la saturación de comandos en el canal de comunicación con el Arduino, asegurando una operación más estable y segura del sistema.

4.3 Modos de funcionamiento

El nodo encargado del movimiento autónomo cuenta con dos modos de funcionamiento, que se seleccionan a través del script de inicio con el que se lanza el sistema: move_robot-ToF-Lidar-YOLO1 o move_robot-ToF-Lidar-YOLO2. Ambos modos comparten la lógica principal de navegación, pero se diferencian en la forma en que integran la detección de objetos mediante YOLO y la gestión de las imágenes capturadas por la cámara.

Modo 1: Detección en Tiempo Real

El primer modo, activado mediante el script move_robot-ToF-Lidar-YOLO1, representa una opción más simple en cuanto la implementación, puesto que este modo no necesita crear una estructura de carpetas donde guardar las imágenes y cambiar el resto de nodos para que esto funcione. En este caso, desde el inicio del sistema se lanza el nodo de detección con el script detec-mapa-YOLO1. Este script se conecta directamente en tiempo real al flujo de vídeo de la cámara y realiza una detección de objetos cada dos segundos, utilizando un único peso de red neuronal previamente definido. Durante la fase de escaneo del entorno, el nodo de YOLO permanece activo realizando estas detecciones periódicas. Una vez finaliza el mapeo, este proceso se detiene automáticamente y se lanza el script mapa_simbolico-YOLO1, cuyo funcionamiento ya fue descrito anteriormente. Este último proceso completa la generación del mapa simbólico y concluye la ejecución del sistema.

Modo 2: Captura y Procesamiento Asíncrono

El segundo modo, iniciado mediante el script move_robot-ToF-Lidar-YOLO2, es más complejo y potente, y por tanto se le ha dedicado más tiempo de desarrollo. En este caso, en lugar de realizar detección en tiempo real, el sistema toma capturas de la cámara cada dos segundos durante todo el proceso de mapeo. Estas imágenes se almacenan en una carpeta específica nombrada con un timestamp que identifica la sesión.

Una vez finaliza la fase de escaneo y se genera el mapa, se lanza el script detec-mapa-YOLO2. A este script se le proporcionan como parámetros el peso de YOLO que se desea utilizar, el timestamp de la sesión y la carpeta donde se encuentran almacenadas las capturas. El script analiza cada imagen no solo con YOLO para identificar objetos, sino que también intenta estimar la profundidad de las detecciones para mejorar su integración en el mapa.

Este proceso de análisis se repite tantas veces como pesos distintos se hayan definido en el script move_robot-ToF-Lidar-YOLO2. Una vez completado el análisis con todos los modelos, se ejecuta mapa_simbolico-YOLO2, cuyo comportamiento es idéntico al de su homólogo en el modo 1.

Este segundo modo ofrece la ventaja de almacenar las capturas durante la navegación, lo que permite aplicar tecnologías adicionales de forma asincrónica. Estas herramientas complementarias se pueden ejecutar incluso después de la detección inicial, y en esta implementación se integran mediante comunicaciones MQTT. Cuando estas tecnologías finalizan su ejecución y cierran sus procesos, el script principal también concluye su actividad, finalizando así el ciclo completo del sistema.

4.4 Flujo de Ejecución del Sistema

Conociendo ya los nodos de ROS que conforman el sistema, así como sus funciones y configuraciones, es posible describir el flujo de ejecución general del sistema desde el arranque hasta la obtención de los mapas finales.

El proceso comienza accediendo de forma remota a la Jetson Nano, la cual se encuentra montada en el robot. A continuación, se procede a cargar el programa correspondiente al Arduino mediante el entorno de desarrollo Arduino IDE. Este paso no es estrictamente necesario si el código ya se encuentra cargado previamente, pero se recomienda realizarlo por razones de seguridad y control del sistema. Cabe señalar que el código del Arduino no se documenta en este proyecto, ya que ha sido proporcionado por el fabricante de la placa de control, y únicamente se requiere conocer cómo establecer la comunicación con él desde ROS.

Una vez listo el entorno, se abren tres consolas para lanzar los distintos procesos del sistema:

En la primera consola se ejecuta el comando:

roslaunch slam_rtabmap.launch

Este lanzamiento inicia todos los sensores necesarios (LiDAR, cámara Azure Kinect, etc.) y lanza RViz para la visualización en tiempo real tanto de los datos obtenidos como del proceso de mapeado.

En la segunda consola se ejecuta:

python3 motor_controller.py

Este script mantiene el nodo motor_control a la espera de recibir información del nodo principal de movimiento, para luego transmitirla al Arduino y controlar así el desplazamiento del robot.

En la tercera consola se lanza el nodo principal del sistema de navegación autónoma, eligiendo uno de los dos modos de funcionamiento:

python3 move_robot-ToF-Lidar-YOLO1

o bien:

python3 move_robot-ToF-Lidar-YOLO2

según se desee utilizar un enfoque de detección en tiempo real o basado en capturas temporales.

Una vez finalizados todos los procesos, se pueden visualizar los mapas generados. Por un lado, se dispone del mapa métrico generado por el LiDAR y RTAB-Map en formato .yaml. Por otro, se cuenta con el mapa simbólico enriquecido con detecciones de objetos, generado a partir de los resultados de YOLO y exportado en formato .json, lo que permite una representación estructurada y detallada del entorno explorado.

La Ilustración 11 representa el diagrama de conectividad de los nodos y tópicos de ROS que orquestan el funcionamiento del sistema autónomo. En este esquema, se visualiza cómo el Azure_Kinect_ROS_Driver y rplidar_ros publican datos crudos de imagen y profundidad (/k4a/depth/image_raw, /k4a/color/image_raw) y escaneo LiDAR (/scan) respectivamente. Estos tópicos, junto con la información de odometría (/odom), son consumidos por rtabmap_ros para generar el mapa (/map). El nodo depth_movement es el centro de control del movimiento, recibiendo el mapa y lanzando, a su vez, la detección de objetos con yolov7 y la generación del mapa_simbolico. Finalmente, depth_movement envía comandos de velocidad (/robot/move/raw) y dirección (/robot/move/direction) al nodo

motor_control, que interactúa directamente con los actuadores del robot. Este diagrama es crucial para comprender el flujo de información y la interdependencia entre los diferentes componentes del sistema ROS, desde la percepción hasta la actuación.

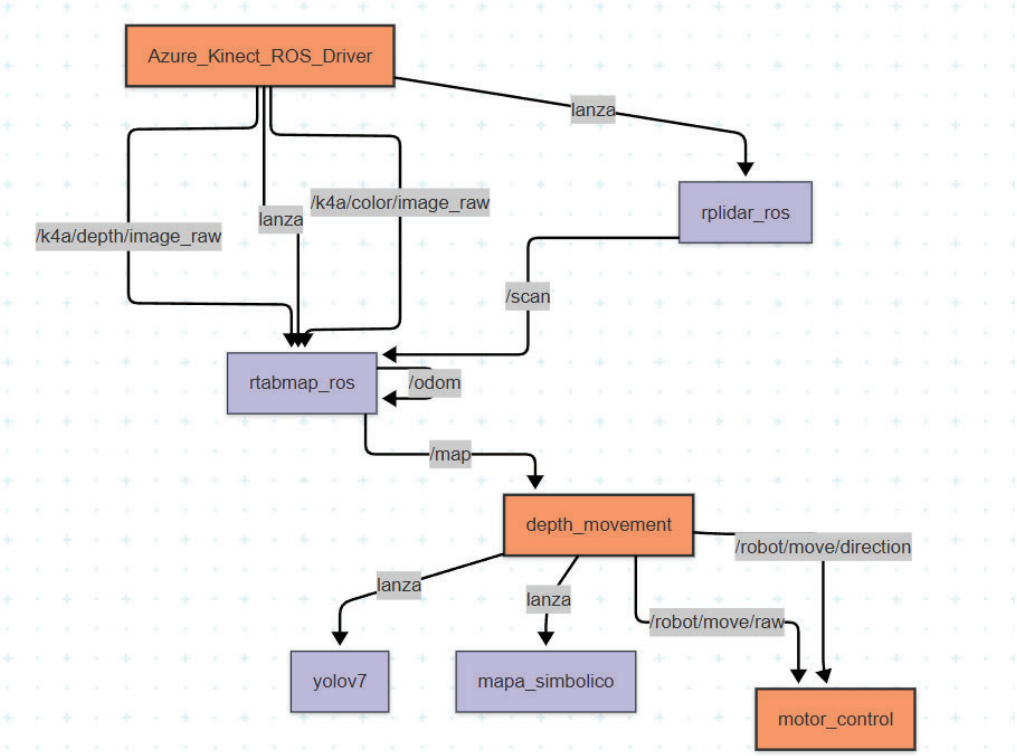

Ilustración 11 Conectividad de los nodos y tópicos

4.5 Script de conducción autónoma

Uno de los aspectos que ha requerido mayor dedicación a lo largo del proyecto ha sido el desarrollo del código encargado de la lógica de conducción autónoma. Este bloque de código representa el núcleo funcional del sistema y condensa gran parte del trabajo técnico realizado, desde la integración de sensores hasta la toma de decisiones en tiempo real. Su diseño modular y escalable tiene como objetivo no solo adaptarse a las limitaciones actuales del hardware, sino también facilitar futuras ampliaciones cuando se disponga de plataformas más potentes. Por ello, se ha procurado dotar al script de un comportamiento robusto y lógico, contemplando escenarios variados de navegación, detección de obstáculos y control del movimiento.

En la ilustración 12 que se muestra a continuación se puede observar una de las partes clave del sistema: la inicialización de parámetros y la obtención de la posición inicial del robot en el marco global (map). Esta transformación se realiza mediante TF, y permite situar al robot correctamente en el entorno antes de iniciar cualquier tarea de navegación. A continuación, se definen múltiples parámetros fundamentales como los ángulos de visión

frontal y lateral del LiDAR, las velocidades máximas y mínimas, y distancias críticas para la seguridad. También se configuran valores relacionados con el comportamiento del sistema, como el intervalo de chequeo del mapa, los tiempos mínimos y máximos de rotación, y el umbral de cobertura objetivo en la exploración. Estos parámetros son esenciales para el correcto funcionamiento del sistema, ya que regulan su respuesta ante distintas condiciones del entorno y aseguran una conducción autónoma eficiente dentro de los límites operativos del hardware disponible.

```
self.initial_pose = None
# Wait until tf is ready
try:
    rospy.loginfo(
        "Esperando transformada inicial de 'map' a 'base_link'...")
    self.tf_listener.waitForTransform(
        "map", self.base_frame, rospy.Time(0), rospy.Duration(10.0)
    )
    trans, rot = self.tf_listener.lookupTransform(
        "map", self.base_frame, rospy.Time(0)
    )
    self.initial_pose = PoseStamped()
    self.initial_pose.header.frame_id = "map"
    self.initial_pose.header.stamp = rospy.Time.now()
    self.initial_pose.pose.position.x = trans[0]
    self.initial_pose.pose.position.y = trans[1]
    self.initial_pose.pose.position.z = trans[2]
    self.initial_pose.pose.orientation.x = rot[0]
    self.initial_pose.pose.orientation.y = rot[1]
    self.initial_pose.pose.orientation.z = rot[2]
    self.initial_pose.pose.orientation.w = rot[3]
    rospy.loginfo("Posición inicial registrada.")
except Exception as e:
    rospy.logerr(f"No se pudo obtener la posición inicial: {e}")

# Core movement and navigation parameters
# 30 grados a cada lado para la zona frontal
self.front_angle = math.radians(30)
# 90 grados para las zonas laterales
self.side_angle = math.radians(90)
self.max_speed = rospy.get_param("~max_speed", 200)
self.min_speed = rospy.get_param("~min_speed", 100)
self.min_safe_distance = rospy.get_param("~min_safe_distance", 1.2)
self.critical_distance = rospy.get_param("~critical_distance", 0.5)
self.map_check_interval = rospy.get_param("~map_check_interval", 5)
self.rotation_min_time = rospy.get_param("~rotation_min_time", 1.5)
self.rotation_max_time = rospy.get_param("~rotation_max_time", 3.0)
self.target_coverage = rospy.get_param("~target_coverage", 0.95)
self.frontier_search_distance = rospy.get_param(
    "~frontier_search_distance", 3.0
)
self.lidar_safety_angle = math.radians(
    rospy.get_param("~lidar_safety_angle", 30)
)
self.scan_timeout = rospy.Duration(
    rospy.get_param("~scan_timeout", 5.0))
```

Ilustración 12 Fragmento de inicialización de variables

En la ilustración 13 se muestra un segundo fragmento de inicialización donde se definen los parámetros relacionados con la detección de escaleras y la integración de la red neuronal YOLO. Los parámetros del detector de escaleras han sido configurados con valores conservadores para minimizar falsos positivos, lo que resulta esencial para evitar

situaciones de riesgo en entornos reales. Entre ellos se incluyen umbrales para la altura del suelo, el tamaño de una caída detectable, la distancia mínima considerada como peligrosa, y la activación o no del modo de evitación automática de escaleras. Por otro lado, se integran múltiples modelos entrenados de YOLOv7, cada uno especializado en una clase de objeto (como rampas, puertas, escaleras o ascensores), lo cual responde al funcionamiento del script en su modo por capturas, en el que se procesan imágenes individuales con pesos distintos según el tipo de detección que se desea realizar. Este enfoque permite una detección robusta y especializada adaptada al entorno. Además, el script establece variables de estado esenciales para controlar la lógica del comportamiento autónomo, tales como el estado de exploración, la cobertura del mapa, el seguimiento de obstáculos consecutivos, y flags de emergencia. Este conjunto de inicializaciones es fundamental para garantizar que el robot tome decisiones adecuadas en función del entorno percibido y que actúe con seguridad y coherencia en escenarios complejos.

```
# Stair detector parameters with more conservative defaults
self.ransac_threshold = rospy.get_param("~ransac_threshold", 0.03)
self.floor_height_threshold = rospy.get_param(
    "~floor_height_threshold", 0.03)
self.drop_threshold = rospy.get_param("~drop_threshold", 0.12)
self.danger_distance = rospy.get_param("~danger_distance", 1.0)
self.camera_frame = rospy.get_param("~camera_frame", "camera_link")
self.avoid_stairs_enabled = rospy.get_param("~avoid_stairs", True)

# YOLO parameters - pesos específicos para ejecución
self.yolo_weights = [
    "/home/jetson/catkin_ws/src/yolov7/runs/new_custom4/weights/best.pt",
    "/home/jetson/catkin_ws/src/yolov7/runs/rampa4/weights/best.pt",
    "/home/jetson/catkin_ws/src/yolov7/runs/puerta4/weights/best.pt",
    "/home/jetson/catkin_ws/src/yolov7/runs/stairs3/weights/best.pt",
    "/home/jetson/catkin_ws/src/yolov7/runs/ascensor3/weights/best.pt",
]
self.yolo_img_size = 640
self.yolo_conf_thres = 0.25
self.yolo_process = None
self.yolo_script_path = self.find_yolo_script()

# State variables
self.running = True
self.exploration_state = "INITIAL_SCAN"
self.last_lidar_time = rospy.Time.now()
self.laser_scan = None
self.initial_scan_start = None
self.map_data = None
self.map_save_path = os.path.expanduser("~/saved_maps")
self.last_map_check_time = rospy.Time.now()
self.map_coverage = 0.0
self.stair_detected = False
self.floor_plane_eq = None
self.danger_zone_detected = False
self.emergency_stop_flag = False
self.current_move_direction = "STOP"
self.current_motor_speeds = [0, 0, 0, 0]
self.movement_lock = False
self.last_movement_command_time = rospy.Time.now()
self.consecutive_obstacles = 0
self.max_consecutive_obstacles = rospy.get_param(
    "~max_consecutive_obstacles", 5
)
```

Ilustración 13 Segundo fragmento de inicialización de variables

La ilustración 14 representa el bucle principal de ejecución del script de exploración autónoma, encargado de coordinar las tareas críticas en tiempo real. En él se definen las acciones que el robot ejecuta de forma continua mientras se encuentra operativo. La rutina comienza con el procesamiento del LiDAR para la navegación y la detección de obstáculos. Si se identifica una escalera o una zona de peligro —determinadas previamente a través del análisis de planos mediante RANSAC y umbrales definidos—, el robot se detiene de inmediato y lanza una maniobra de evasión segura. Además, el sistema controla periódicamente la cobertura del mapa generado y, al alcanzar el umbral predefinido, cambia al estado de retorno a la posición inicial, completando así el ciclo de exploración. Este comportamiento autónomo está pensado para maximizar la cobertura del entorno sin comprometer la seguridad del robot, integrando información sensorial, lógica reactiva y planificación básica de forma eficiente incluso bajo las limitaciones computacionales de la Jetson Nano.

```python
def run(self):
    """Main run loop for the explorer"""
    rospy.loginfo("Starting exploration run")
    rospy.on_shutdown(self.shutdown)

    rate = rospy.Rate(10)  # 10 Hz control loop

    while not rospy.is_shutdown() and self.running:
        # Process LiDAR for obstacle avoidance
        self.process_lidar_for_navigation()

        # Verificar si hay escaleras detectadas
        if self.stair_detected or self.danger_zone_detected:
            rospy.logwarn(
                "¡Escalera o zona de peligro detectada! Deteniendo movimiento"
            )
            self.stop_robot()
            # Esperar un momento antes de intentar maniobrar
            rospy.sleep(1.0)
            # Ejecutar maniobra de evasión
            self.execute_recovery_maneuver()

        # Check if we should return to initial position
        if self.exploration_state == "RETURN_TO_DOCK":
            if self.return_to_initial_position():
                self.exploration_state = "IDLE"

        # Check for map coverage periodically
        if (
            rospy.Time.now() - self.last_map_check_time
        ).to_sec() >= self.map_check_interval:
            if self.check_map_coverage():
                self.exploration_state = "RETURN_TO_DOCK"

        rate.sleep()
```

Ilustración 14 Ejecución principal del código

En la ilustración 15 se implementa la lógica que permite diferenciar entre una simple caída y una escalera mediante el análisis de contornos extraídos de la imagen de profundidad.

El sistema evalúa formas relevantes calculando su relación de aspecto y la proporción entre el área real y el área del rectángulo delimitador (extensión). Estos parámetros geométricos permiten identificar patrones característicos de escaleras, como una relación de aspecto alargada y una cobertura parcial del área delimitada. Si alguno de los contornos cumple con estos criterios, se marca como escalera probable (is_likely_stair). Además, se calcula la distancia mínima desde la cámara a cualquier región identificada como peligrosa, usando directamente los valores del mapa de profundidad. Este enfoque no solo mejora la precisión en la detección de zonas de riesgo, sino que también permite al robot anticipar y evitar situaciones que podrían comprometer su integridad estructural durante la navegación autónoma.

```
# Analyze contour shapes to distinguish stairs from other drops
is_likely_stair = False
for contour in significant_contours:
    # Calculate aspect ratio and area
    x, y, w, h = cv2.boundingRect(contour)
    aspect_ratio = float(w) / h if h > 0 else 0
    area = cv2.contourArea(contour)
    rect_area = w * h
    extent = float(area) / rect_area if rect_area > 0 else 0

    # Stairs typically have specific shape characteristics
    if aspect_ratio > 1.5 and extent > 0.4:
        is_likely_stair = True
        break

# Determine danger zones with distance weighting
self.stair_detected = len(
    significant_contours) > 0 and is_likely_stair
self.danger_zone_detected = False

if self.stair_detected:
    # Calculate minimum distance to any stair region
    min_distance = float("inf")
    for contour in significant_contours:
        # Get all points in the contour
        points = contour.squeeze()
        if points.ndim == 1:
            points = points[np.newaxis, :]

        # Get depths for contour points
        contour_depths = depth_image[points[:,
                                            1], points[:, 0]]
        valid_depths = contour_depths[contour_depths > 0.1]

        if len(valid_depths) > 0:
            contour_min_dist = np.min(valid_depths)
            if contour_min_dist < min_distance:
                min_distance = contour_min_dist
```

Ilustración 15 Fragmento de código de la detección de caídas

En la ilustración 16 se muestra el código encargado de procesar los datos del LiDAR segmentándolos por zonas (frontal, izquierda y derecha) y aplicando un filtrado robusto para mejorar la fiabilidad de las lecturas. Primero se eliminan las lecturas fuera del rango válido del sensor, y posteriormente se utiliza una ventana deslizante para comparar cada

lectura con la mediana local del entorno cercano. Si una lectura difiere significativamente de la mediana —indicativo de un posible valor atípico causado por ruido o reflejos— se descarta. Una vez depurados los datos, se generan máscaras angulares que permiten clasificar cada lectura según su ubicación relativa: frente, izquierda o derecha. Esta segmentación es clave para la toma de decisiones del robot, ya que permite evaluar con precisión en qué dirección se encuentran los obstáculos, facilitando así la navegación autónoma y la evasión de colisiones.

```python
def analyze_lidar_zones(self):
    """Analyze LiDAR data by zones with improved filtering and classification"""
    if self.laser_scan is None:
        return None

    # Get scan data
    ranges = np.array(self.laser_scan.ranges)
    angle_min = self.laser_scan.angle_min
    angle_increment = self.laser_scan.angle_increment

    # Filter invalid readings
    valid_mask = np.logical_and(
        ranges > self.laser_scan.range_min, ranges < self.laser_scan.range_max
    )

    # More efficient outlier filtering using vectorized operations where possible
    window_size = 5
    # Create a view into the data with rolling windows
    # This implementation can be optimized further with specialized libraries like scipy
    filtered_valid_mask = valid_mask.copy()
    for i in range(len(ranges)):
        start_idx = max(0, i - window_size // 2)
        end_idx = min(len(ranges), i + window_size // 2 + 1)
        window = ranges[start_idx:end_idx]
        valid_window = np.logical_and(
            window > self.laser_scan.range_min, window < self.laser_scan.range_max
        )
        # Skip filtering if we have too many invalid readings in the window
        if np.sum(valid_window) < window_size // 2:
            continue

        valid_window_values = window[valid_window]
        if len(valid_window_values) > 0:
            median = np.median(valid_window_values)
            # If value is significantly different from median
            if abs(ranges[i] - median) > 0.5:
                filtered_valid_mask[i] = False  # Mark as invalid

    # Angle array for easier zone calculation
    angles = np.arange(len(ranges)) * angle_increment + angle_min

    # Define zones - front, left, right with overlap for better awareness
    front_mask = np.abs(angles) <= self.front_angle
    left_mask = np.logical_and(
        angles > -self.side_angle, angles < -self.front_angle * 0.5
    )
```

Ilustración 16 Fragmento de código del análisis del LiDAR

En la ilustración 17 aparece el método move, el cual implementa el control principal del movimiento del robot móvil, utilizando ruedas Mecanum y actuando directamente sobre las velocidades individuales de cada una de ellas. Este método recibe una dirección de desplazamiento y un factor de velocidad que modula la intensidad del movimiento. Antes de proceder, realiza comprobaciones críticas: si se detecta una escalera o zona peligrosa, o si está activa una señal de parada de emergencia, el movimiento se cancela

inmediatamente como medida de seguridad. También se verifica si existe un bloqueo de movimiento activo que impida nuevas órdenes.

El cálculo de velocidades se realiza según la dirección recibida. En los desplazamientos rectos (como FORWARD o BACKWARD), todas las ruedas se mueven en la misma dirección. Para los giros (ROTATE_LEFT o ROTATE_RIGHT), las ruedas se activan en sentidos opuestos, con una reducción de velocidad para suavizar la rotación. En los movimientos diagonales (FORWARD_LEFT o FORWARD_RIGHT), se modifica la velocidad relativa de uno de los lados para generar un sesgo direccional.

El método garantiza seguridad y estabilidad durante la navegación, permitiendo movimientos precisos y adaptativos según el entorno, siempre bajo control sincronizado para evitar conflictos de concurrencia. Su diseño hace posible que el robot aproveche al máximo la versatilidad de las ruedas mecanum, manteniendo un comportamiento robusto y confiable.

```python
# Improved move method with smoother control
def move(self, direction, speed_factor=1.0, duration=None):
    """Move the robot with improved control and thread safety"""
    with self.movement_command_lock:
        # Check if stairs are detected
        if self.stair_detected or self.danger_zone_detected:
            rospy.logwarn(
                "Movement canceled! Stair or danger zone detected")
            self.stop_robot()
            return False

        if self.movement_lock and direction != "STOP":
            return False

        # Don't allow movement if emergency stop is active
        if self.emergency_stop_flag and direction != "STOP":
            rospy.logwarn("Movement blocked by emergency stop flag")
            return False

        # Calculate speeds based on direction
        if direction == "STOP":
            speeds = [0, 0, 0, 0]
        elif direction == "FORWARD":
            base_speed = int(self.max_speed * speed_factor)
            speeds = [base_speed, base_speed, base_speed, base_speed]
        elif direction == "BACKWARD":
            base_speed = int(self.max_speed * speed_factor)
            speeds = [-base_speed, -base_speed, -base_speed, -base_speed]
        elif direction == "ROTATE_LEFT":
            # Reduced for smoother rotation
            base_speed = int(self.max_speed * 0.7 * speed_factor)
            speeds = [-base_speed, base_speed, -base_speed, base_speed]
        elif direction == "ROTATE_RIGHT":
            # Reduced for smoother rotation
            base_speed = int(self.max_speed * 0.7 * speed_factor)
            speeds = [base_speed, -base_speed, base_speed, -base_speed]
        elif direction == "FORWARD_LEFT":
            base_speed = int(self.max_speed * speed_factor)
            left_speed = int(base_speed * 0.6)  # Left side slower
            speeds = [left_speed, base_speed, left_speed, base_speed]
        elif direction == "FORWARD_RIGHT":
            base_speed = int(self.max_speed * speed_factor)
            right_speed = int(base_speed * 0.6)  # Right side slower
            speeds = [base_speed, right_speed, base_speed, right_speed]
        else:
            rospy.logwarn(f"Unknown direction: {direction}")
            return False
```

Ilustración 17 Fragmento del código encargado del movimiento

Dado que el código completo del proyecto es especialmente extenso y contempla dos modos de funcionamiento diferenciados, se adjunta un enlace al repositorio del proyecto en GitHub [62] para quien desee analizarlo en mayor profundidad. En este anexo se ha tratado de ilustrar una muestra representativa del volumen de declaraciones necesarias, así como parte del flujo de ejecución principal, el mecanismo de detección de caídas, el

análisis parcial del entorno mediante el sensor LiDAR y la lógica esencial del movimiento autónomo del robot. No obstante, quedan fuera de este documento aspectos igualmente relevantes como el proceso completo de mapeo con RTAB-Map, la integración con YOLO para la detección de objetos, la ejecución del script encargado del mapa simbólico, diversos mecanismos de seguridad del vehículo tanto durante su desplazamiento como en la gestión de errores al buscar scripts externos, la llamada a los scripts de comunicación vía MQTT para la prueba de integración con n8n, así como la captura de imágenes condicionada por el modo de operación, entre otras funcionalidades implementadas.

Capítulo 5. Evaluación del Sistema: Pruebas y Limitaciones Técnicas

En este capítulo se detallan las pruebas diseñadas para evaluar la detección de obstáculos, la calidad del mapeo y la navegación autónoma del robot. Si bien las limitaciones de potencia de la Jetson Nano impidieron implementar la ejecución consecutiva de los múltiples sensores, la creación de los mapas y el movimiento del robot de forma conjunta, se llevaron a cabo **pruebas parciales** exitosas como la detección con YOLO y un prototipo de navegación autónoma. Se analiza el **cumplimiento parcial** de los objetivos y se identifican los aspectos no validados.

5.1 Pruebas planificadas

Dado que el sistema está compuesto por múltiples módulos interdependientes, las pruebas diseñadas para su validación se plantearon de forma **modular y progresiva**, siguiendo el orden lógico de desarrollo del proyecto. Estas pruebas se agruparon en tres grandes bloques: **pruebas sensoriales**, **pruebas de movimiento** y **pruebas de mapeo**.

1. Pruebas Sensoriales

Estas pruebas se enfocaron en verificar el correcto funcionamiento de los componentes encargados de la percepción del entorno. Se incluyeron:

- Comprobación del funcionamiento de la cámara en conjunto con el sistema de detección de objetos YOLO.
- Evaluación del entrenamiento del modelo YOLO con las clases seleccionadas.
- Validación del funcionamiento simultáneo de todos los sensores del sistema (cámara, LiDAR, IMU).

2. Pruebas de Movimiento

En este bloque se analizaron aspectos relacionados con la capacidad del robot para desplazarse de forma autónoma y segura. Las pruebas fueron:

- Verificación de las conexiones físicas y funcionamiento de la parte motorizada.
- Evaluación de la capacidad del sistema para detectar obstáculos y reaccionar en tiempo real.

- Comprobación de la detección de desniveles, escaleras o caídas mediante sensores complementarios.
- Validación del movimiento autónomo y las decisiones de navegación.
- Prueba de la capacidad para retornar a la posición inicial registrada al inicio del mapeo.

3. Pruebas de Mapeo

Finalmente, se evaluaría la precisión y coherencia del sistema de mapeo del entorno, tanto en su versión métrica como simbólica. Las pruebas incluyeron:

- Generación de mapas a partir del sensor LiDAR sin errores estructurales.
- Comprobación de la creación del mapa simbólico final con las detecciones procesadas e integradas correctamente.

Esta organización modular permitió una evaluación más clara de cada componente del sistema, facilitando la detección de fallos y la iteración durante el proceso de desarrollo.

5.2 Pruebas Sensoriales

Aunque en apariencia pueda parecer un proceso simple —verificar que la cámara funciona y que el sistema YOLO detecta objetos—, en este proyecto resultó ser un aspecto crítico debido a las condiciones particulares del sistema. En primer lugar, la cámara está instalada a una altura muy baja, lo que puede limitar significativamente el campo de visión y dificultar la detección de ciertos objetos. Además, como se mencionó en su apartado correspondiente, fue necesario configurar la cámara con una calidad mínima para asegurar su funcionamiento en la Jetson Nano, dadas las limitaciones de potencia del dispositivo.

Para validar su eficacia, se realizaron capturas de imágenes desde la altura real de la cámara, comprobando si el sistema era capaz de detectar correctamente los objetos entrenados. En la mayoría de los casos, los resultados fueron satisfactorios. No obstante, se identificó un problema relevante: en ciertos casos, aunque YOLO detectaba correctamente un objeto —por ejemplo, unas escaleras—, la caja delimitadora generada se posicionaba de forma incorrecta, apareciendo en el suelo frente al objeto en lugar de sobre él. Este tipo de error no fue exclusivo de las escaleras, aunque sí fueron, junto con las rampas, las categorías más afectadas por este tipo de imprecisión.

También se realizaron pruebas en tiempo real, sin el uso de imágenes estáticas. Aunque el funcionamiento fue adecuado, estas pruebas estaban condicionadas por la necesidad de mantener conectados tanto la Jetson Nano como la cámara a una fuente de alimentación constante, lo que limitaba la movilidad del robot durante la validación.

En cuanto al entrenamiento del modelo YOLO, no se hará énfasis en los pesos específicos utilizados, sino en el proceso de ajuste que fue necesario para alcanzar una detección óptima. Inicialmente, se entrenaron los modelos durante 100 épocas, pero el resultado arrojaba un margen de error inaceptable. Al duplicar el entrenamiento a 200 épocas, se produjo un fenómeno de sobreajuste (overfitting): la red comenzaba a identificar erróneamente objetos con características similares como si fueran los entrenados. Para corregir esto, se volvió a un entrenamiento de 100 épocas, pero aumentando la cantidad y diversidad de imágenes en el dataset. Este ajuste permitió obtener el modelo final que se utilizó en el proyecto, con un buen equilibrio entre precisión y generalización.

Finalmente, se realizaron pruebas de integración entre los sensores para visualizar correctamente los datos en RViz y asegurar la coherencia del mapa generado. Durante esta fase surgieron errores relacionados con la transformación de marcos de referencia (TF), los cuales impedían una correcta ejecución del sistema. La solución fue crear manualmente los marcos de transformación /base_link y /base_camera, estabilizando así la relación entre los distintos sensores y permitiendo la correcta visualización y fusión de los datos.

La Ilustración 18 muestra una captura de pantalla del entorno de RViz, donde se puede observar el modelo URDF del robot junto con la nube de puntos generada por el sensor LiDAR. Esta visualización es fundamental para verificar la correcta localización del robot en el espacio y la precisión de las lecturas del LiDAR, lo que contribuye a la construcción de un mapa coherente y la navegación autónoma.

Complementariamente, se incluye la ilustración 19 del entorno físico de pruebas, donde se observa el robot montado sobre una de las estaciones de trabajo del laboratorio. Este montaje no corresponde a la versión definitiva del sistema, ya que en ese momento se estaban realizando pruebas de integración de los sensores, como el LiDAR y la cámara RGB-D. La comparación entre el modelo visualizado en RViz y el robot físico permite validar la correcta correspondencia entre el entorno simulado y el real, lo cual es clave para garantizar un funcionamiento preciso del sistema durante la navegación autónoma y la generación del mapa en tiempo real.

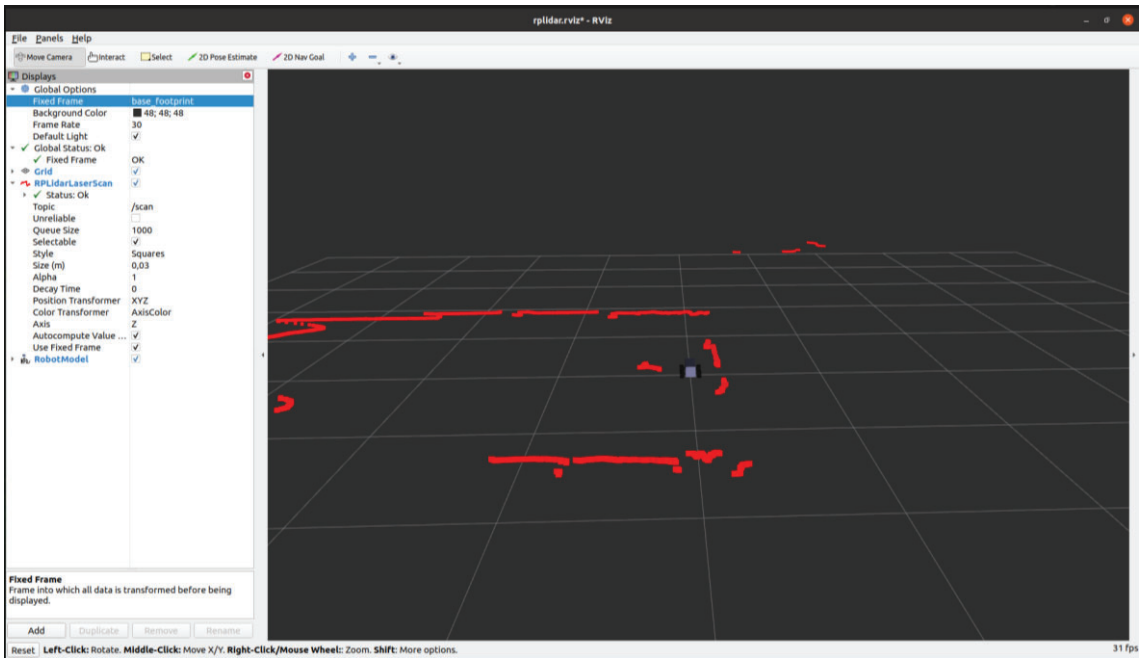

Ilustración 18 Visualización del robot en VRIZ

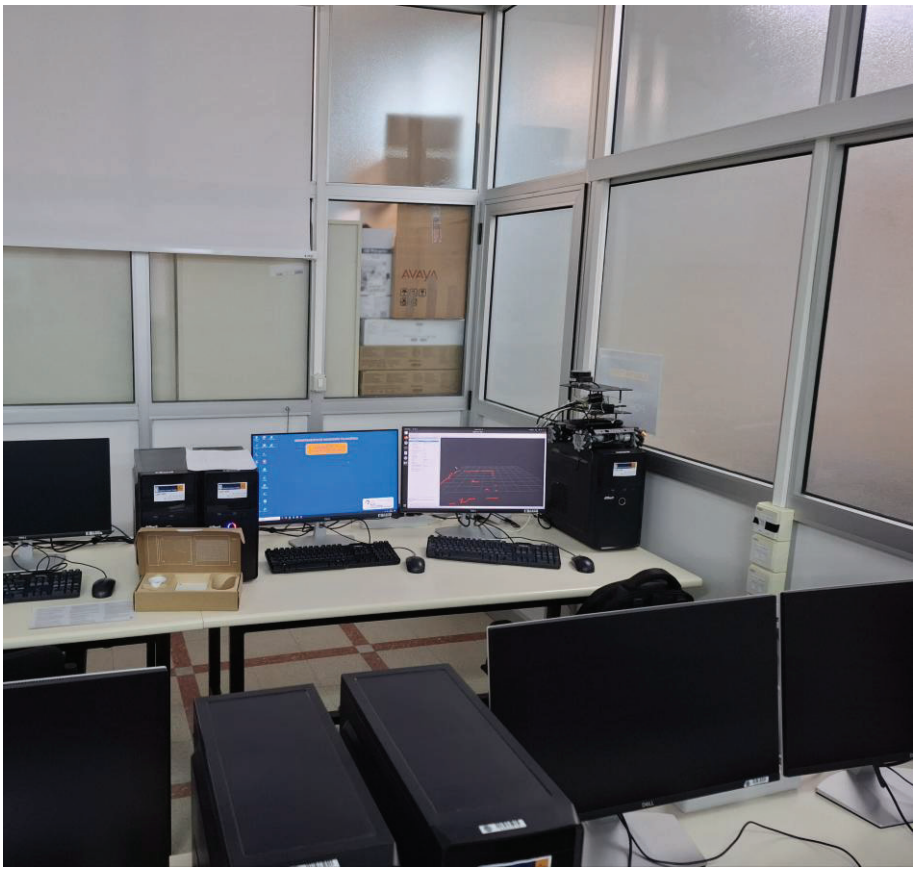

Ilustración 19 Visualización física del entorno mapeado

5.3 Pruebas de Movimiento

Este primer punto fue esencial para garantizar la viabilidad técnica del proyecto, no solo por la conexión final del sistema (como se muestra en la Ilustración 5), sino también para asegurar la integridad física de los componentes. Se buscaba una conexión estable, sin fallos eléctricos, que permitiera una adecuada ventilación del sistema y una disposición óptima de los sensores embarcados.

En primer lugar, se comprobó el estado de las conexiones físicas de los motores y del microcontrolador Arduino. Fue necesario ajustar e incluso rehacer algunas conexiones para garantizar una correcta transmisión de señales. Posteriormente, se evaluaron distintas posiciones para los sensores, teniendo en cuenta criterios funcionales: el LiDAR debía colocarse como el elemento más alto del robot, sin obstáculos alrededor, para permitir un escaneo limpio de 360°, mientras que la cámara debía situarse a una altura y ángulo adecuados para detectar tanto desniveles en el suelo como barreras arquitectónicas frontales.

Una vez definida la distribución de hardware, se procedió a verificar que las señales emitidas por la Jetson Nano fueran correctamente interpretadas por el sistema de control. Estas pruebas se realizaron de forma progresiva: primero enviando comandos manuales desde el IDE de Arduino, luego mediante tópicos de ROS publicados por consola (CMD), y finalmente desde un prototipo inicial del nodo motor_control.

Superada esta etapa, se ejecutó una prueba clave para validar la detección de obstáculos y la capacidad de reacción del robot. Se utilizó un script básico de conducción autónoma, que enviaba una orden de avance continuo y, si la cámara detectaba un obstáculo a 3 metros, activaba una orden de parada, giro a la derecha y nuevo avance. Esta prueba permitió identificar dos fallos críticos en el sistema.

El primer fallo fue la incapacidad de la cámara para detectar obstáculos con suficiente antelación. Aunque se configuró para detener el robot si un objeto se encontraba a 3 metros, la cámara no ofrecía datos fiables a esa distancia y, en muchos casos, fallaba incluso a distancias menores. Este comportamiento se atribuyó a dos factores: la baja potencia de la Jetson Nano, que obligó a reducir al mínimo la calidad de la cámara, y las propias limitaciones del sensor, al tratarse de una cámara ToF con un rango efectivo de apenas 5 metros. Además, se observó que la calidad de la señal visualizada a través del visor oficial de Azure Kinect (k4viewer) era superior a la obtenida desde los tópicos de ROS, lo cual sugiere que ROS reducía aún más la calidad. Ante esta limitación, se optó

por incorporar el LiDAR como sistema de detección principal, permitiendo no solo una detección más robusta, sino también un mapeo más preciso gracias a su visión de 360°.

Las siguientes imágenes muestran una comparación entre la visualización de la cámara Azure Kinect usando la herramienta oficial k4aviewer (Imagen 20) y su funcionamiento integrado en ROS a través de RViz (Imagen 21). En la primera imagen, se observa una nube de puntos generada con alta densidad y precisión, gracias a la configuración óptima que ofrece el visor nativo del SDK de Azure. Por el contrario, en la segunda imagen, al utilizar la cámara dentro del entorno ROS, se aprecia una notable disminución en la calidad de los datos de profundidad. La imagen IR presenta saturaciones, la cámara de color mantiene buena definición, pero la profundidad muestra zonas oscuras y áreas sin datos útiles. Esta comparación permite evidenciar cómo las limitaciones de procesamiento en la Jetson Nano, sumadas a la configuración necesaria para mantener un rendimiento aceptable en ROS, afectan directamente a la fidelidad de los datos de entrada que se utilizan para navegación y mapeo.

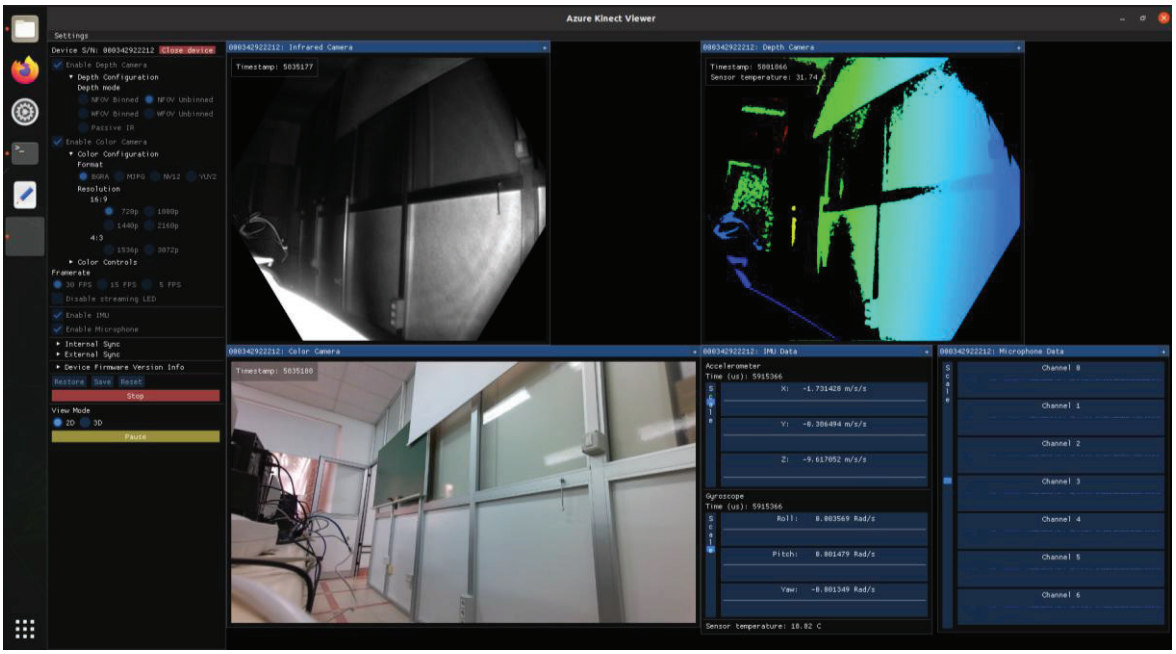

Ilustración 20 Vista de la cámara con k4aviewer

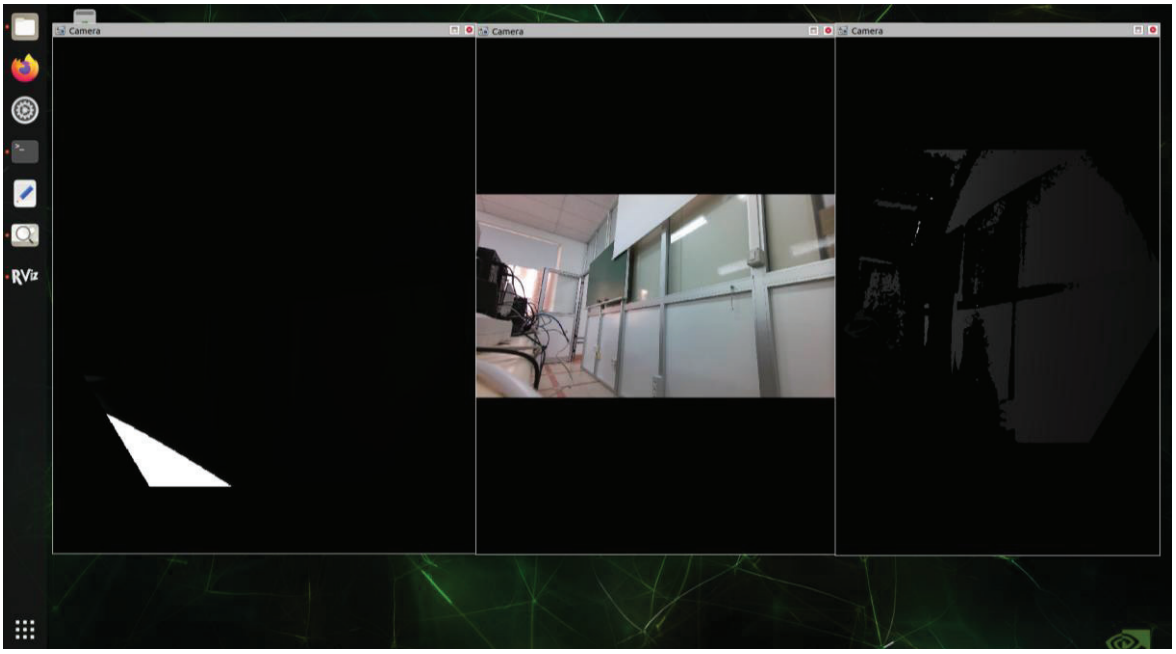

Ilustración 21 Vista de la cámara desde ROS

La Ilustración 22 muestra una prueba inicial de detección de obstáculos utilizando únicamente el LiDAR como mecanismo de percepción, en un escenario estático. En dicha prueba se emplearon dos scripts prototipo que se ejecutaban simultáneamente y registraban su salida por consola, permitiendo verificar la lógica de interacción entre detección y movimiento. A la izquierda de la ilustración, se observa el script move_robot-2_prototype.py, que se encarga de enviar comandos de avance y registrar la detección constante de un obstáculo crítico a 0.26 metros. Esta repetición confirma que el robot se encontraba estático frente a un impedimento fijo, simulando así una condición controlada para la depuración de la lógica. A la derecha, el script motor_controller_prototype.py muestra las decisiones de reacción tomadas ante la detección del obstáculo, como "ROTATE_LEFT" o "ROTATE_RIGHT", evidenciando la correcta interpretación de los datos del LiDAR y la comunicación con el sistema de control motor.

Esta configuración fue clave para validar el funcionamiento básico del sistema antes de su integración completa. Permitió comprobar que los componentes de detección y control podían interactuar adecuadamente y responder a estímulos del entorno, incluso en las

limitaciones de un entorno simulado. Asimismo, esta prueba sirvió para depurar errores lógicos antes de ejecutar pruebas dinámicas más complejas.

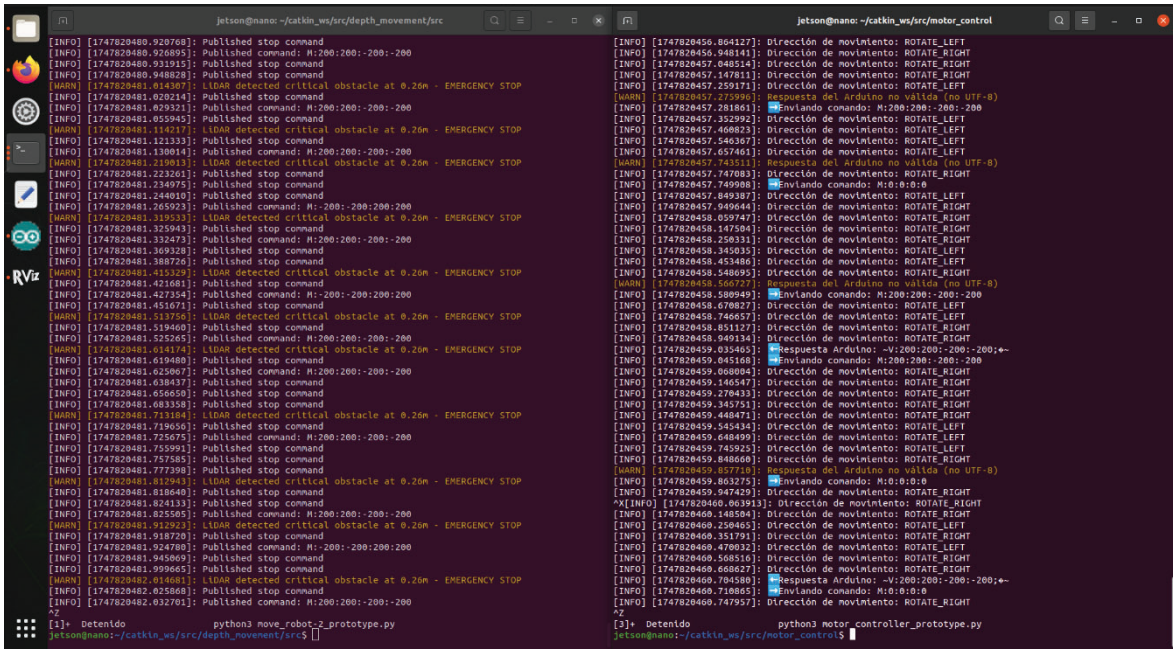

Ilustración 22 Demostración en consola de los scripts prototipos

El segundo problema surgió al intentar hacer que la conducción autónoma fuese más compleja y funcional que la simple maniobra de girar ante un obstáculo. Aunque todavía se trabajaba con prototipos del script final y no se utilizaba mapeo en la navegación autónoma, el sistema no era capaz de soportar la carga generada. Los tres nodos principales —Azure_Kinect_ROS_Driver (con LiDAR y cámara, pero sin mapeo), motor_control y depth_movement con el script de prueba— se ejecutaban en versiones reducidas, pero, aun así, tras unos segundos de funcionamiento correcto, comenzaron a presentarse retrasos en la publicación de los mensajes. Incluso al detener los nodos de detección y control, los mensajes seguían transmitiéndose durante un tiempo, lo que confirmaba una saturación en el sistema provocada por las limitaciones de hardware de la Jetson Nano.

Dado este escenario, no fue posible continuar con las pruebas planeadas. No obstante, se dejó definida la metodología que se habría seguido si el sistema hubiese tenido la capacidad suficiente para ejecutarlas:

- Pruebas de detección de desniveles y escaleras: se habrían realizado inicialmente con el robot estático, colocado en el borde de una mesa para simular un desnivel, observando la respuesta del script de movimiento autónomo a través de consola.

Una vez validado este comportamiento, se trasladaría la prueba a un entorno real con escaleras, siempre garantizando medidas de seguridad para evitar caídas.

- Pruebas de movimiento autónomo completo y retorno a posición inicial: se habría utilizado un entorno simple para validar si el robot era capaz de desplazarse de forma autónoma hasta el final de un trayecto y luego regresar a su punto de origen. La posición inicial se variaría en diferentes repeticiones, y posteriormente se aumentaría la complejidad del entorno para verificar la escalabilidad del sistema.

5.4 Pruebas de Mapeo

El siguiente paso sería validar los mapas generados por el sistema, verificando que ofrecieran una representación precisa y coherente del entorno.

Se planificaron varias pruebas para evaluar la calidad del mapa métrico obtenido a partir de los datos del sensor LiDAR. Estas pruebas incluían:

- Verificación de la precisión dimensional: El robot se habría colocado en entornos controlados con dimensiones previamente medidas (laboratorios, pasillos y salas), comparando posteriormente las distancias y proporciones del mapa generado con las medidas reales.
- Robustez frente a oclusiones temporales: Para simular situaciones reales en entornos dinámicos, se habría introducido la presencia de personas u objetos móviles que bloquearan momentáneamente el campo de visión del sensor LiDAR. El objetivo era comprobar si el sistema mantenía la coherencia del mapa a pesar de estas interrupciones puntuales.

Además, se definieron pruebas para evaluar la integración de las detecciones visuales en el mapa, comprobando que los objetos identificados se ubicaran de forma coherente en la representación espacial. Estas pruebas incluían:

- Correspondencia entre mapa métrico y simbólico: Se pretendía comprobar que cada objeto representado simbólicamente tuviera una correspondencia clara con una estructura presente en el mapa métrico.
- Precisión posicional de los objetos detectados: Se habrían tomado mediciones cruzadas de distancias entre distintos objetos en el mapa simbólico y en el mapa métrico, verificando que las posiciones relativas fueran consistentes.
- Validación de la identificación de objetos: Se habría realizado un análisis cuantitativo para comprobar que los objetos representados en el mapa simbólico

coincidieran con los presentes en el entorno real, evaluando tanto falsos positivos como falsos negativos.

- Detección de elementos clave para la navegación: Se habría valorado la capacidad del sistema para identificar correctamente elementos críticos como escaleras, rampas o zonas peligrosas.
- Consolidación de múltiples detecciones: Finalmente, se habría comprobado que el sistema pudiera unificar distintas detecciones del mismo objeto realizadas desde diferentes puntos de vista, evitando duplicaciones en el mapa simbólico.

5.5 Pruebas Externas

Aunque las funcionalidades descritas en este trabajo de fin de grado abarcan los objetivos principales del proyecto, se desarrolló de forma complementaria una línea de pruebas orientada a mejorar la detección de barreras arquitectónicas y a reducir el margen de error inherente al sistema de detección por YOLO. Esta parte del trabajo no forma parte del núcleo del proyecto, pero su inclusión responde a un enfoque innovador, buscando soluciones viables para detectar elementos arquitectónicos complejos que, por limitaciones técnicas y de tiempo, no pudieron valorarse con precisión (como escaleras sin barandillas, rampas con inclinaciones excesivas o elementos estructurales fijos que obstaculizan el paso).

Para ello, se utilizó n8n [63], una herramienta de automatización de flujos de trabajo de código abierto que permite conectar servicios, APIs y procesos de forma visual. Esta plataforma resultó clave para el diseño de una arquitectura que integrara análisis por IA sobre imágenes capturadas por el sistema. Aunque no se incluirá una descripción técnica detallada de los flujos creados ni de los nodos empleados, ya que quedan fuera del alcance de este trabajo, sí se explica la lógica seguida.

El primer flujo creado en n8n procesaba imágenes previamente almacenadas en la nube. Estas imágenes eran enviadas a un nodo de tipo IA Agent [64], componente que permite interactuar con modelos de inteligencia artificial externos mediante solicitudes definidas. En este caso, el agente se conectaba a Google Gemini, un modelo de lenguaje multimodal que permite razonamientos complejos a partir de texto e imagen. A través de un prompt cuidadosamente diseñado, el sistema analizaba cada imagen buscando exclusivamente la presencia de barreras arquitectónicas previamente definidas. Como salida, se generaba un archivo JSON para cada imagen, conteniendo únicamente valores booleanos (True o False) asociados a cada tipo de barrera detectada. Esta salida permitiría contrastar

posteriormente los resultados obtenidos mediante YOLO, enriqueciendo así la evaluación final del entorno.

Dado que este enfoque no funciona en el caso real, sino mediante imágenes ya guardadas en la nube, se desarrolló un segundo flujo definitivo que permitiera gestionar las capturas tras finalizar el análisis principal. Para ello, fue necesario resolver cómo enviar las imágenes desde el sistema (Jetson Nano) al entorno de ejecución de n8n. Como n8n se encontraba ejecutándose dentro de un contenedor Docker [65] —una tecnología de virtualización ligera que aísla aplicaciones y servicios—, las imágenes generadas por el sistema, ubicadas fuera del contenedor, no podían ser accedidas directamente. Por tanto, se optó por utilizar un broker de mensajería para gestionar la comunicación entre los procesos.

En concreto, se configuró MQTT [66] (Message Queuing Telemetry Transport), un protocolo ligero de mensajería ideal para entornos embebidos y sistemas distribuidos como el utilizado en este proyecto. Se habilitó una instancia MQTT conectada tanto al contenedor de n8n como al sistema host. Para gestionar este flujo, se crearon dos scripts personalizados. El primero se ejecutaba automáticamente cuando YOLO finalizaba el análisis de imágenes; su función era recorrer la carpeta de capturas (pasada como parámetro), enviar cada imagen al flujo de n8n a través del broker MQTT, y mover las imágenes procesadas a una carpeta diferente para evitar su reanálisis. El segundo script, ejecutado en paralelo, recibía a través de MQTT el número total de imágenes tratadas, lo que le permitía cerrarse una vez completado el procesamiento. Por cada imagen analizada, este segundo script guardaba un archivo JSON en la Jetson Nano, completando así el proceso de detección y registro.

Este sistema complementario supuso una mejora significativa en cuanto a capacidad de detección avanzada, permitiendo incorporar un razonamiento contextual y estructural sobre las imágenes del entorno, más allá de lo que las herramientas tradicionales como YOLO pueden ofrecer por sí solas.

En la Ilustración 23 se muestran los dos flujos previamente descritos, siendo el de arriba el que funciona mediante la nube y con una inicialización (trigger) manual y el flujo de abajo siendo el flujo final, el cual se inicia mediante MQTT.

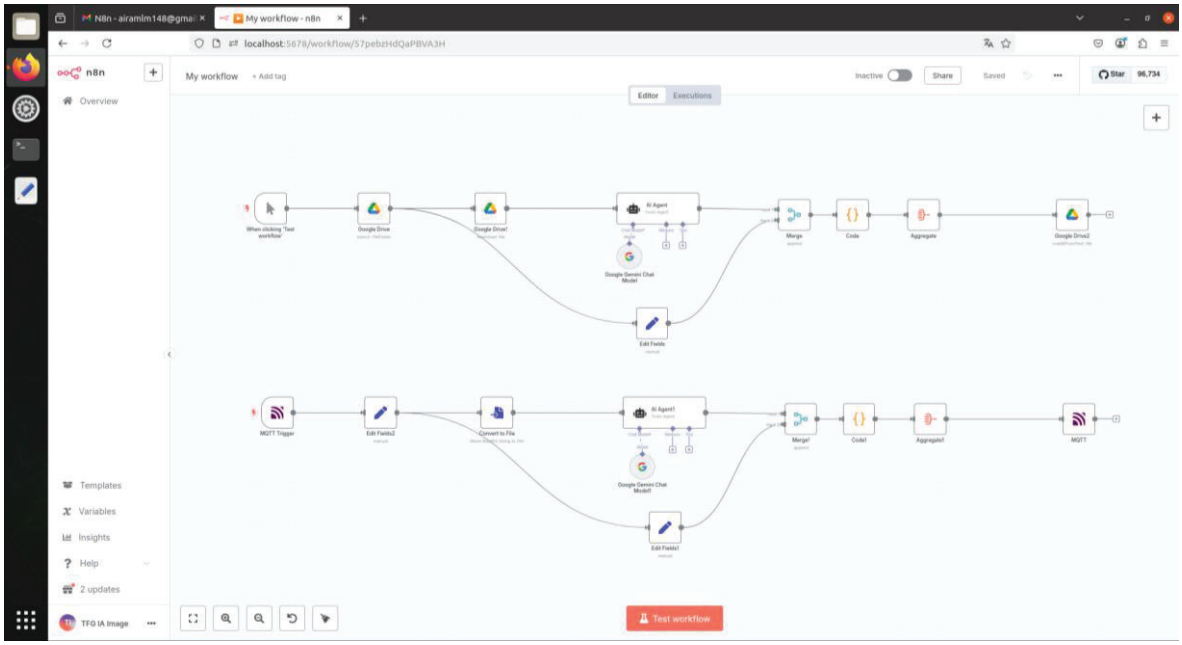

Ilustración 23 Flujos de trabajo en n8n

Capítulo 6. Conclusiones y líneas futuras

El capítulo seis recoge las conclusiones del presente trabajo, en las que se sintetizan los principales logros alcanzados a lo largo del desarrollo del proyecto. Asimismo, se exponen las limitaciones detectadas durante su implementación, tanto a nivel técnico como de alcance, y se plantean diversas propuestas de mejora. Estas sugerencias abren la puerta a futuras líneas de trabajo que permitirían ampliar las funcionalidades del sistema, optimizar su rendimiento o adaptarlo a nuevos entornos y necesidades, favoreciendo así su evolución y aplicabilidad en escenarios más complejos o exigentes.

6.1 Logros alcanzados

A lo largo del desarrollo del proyecto se han conseguido una serie de hitos técnicos que demuestran la viabilidad y funcionalidad del sistema propuesto. Los principales logros son los siguientes:

- **Integración en plataforma embebida:** Se ha logrado implementar con éxito un sistema autónomo en una plataforma de recursos limitados como la Jetson Nano, utilizando ROS como middleware para coordinar los distintos componentes. El sistema integra de forma operativa una cámara ToF, un sensor LiDAR y un robot motorizado.
- **Conducción autónoma basada en sensores:** Se ha desarrollado un sistema de navegación que permite utilizar los datos de los sensores para guiar de forma autónoma al robot. Esta funcionalidad se apoya en la percepción activa del entorno, procesando la información de profundidad y distancia para evitar obstáculos y tomar decisiones de movimiento.
- **Gestión de la cámara mediante ROS:** Se han implementado diferentes usos para la cámara en el entorno ROS, permitiendo su empleo tanto en tiempo real como mediante capturas periódicas. Esto permite su uso dual: durante la navegación para la recolección de datos y para la detección de objetos mediante YOLO.
- **Entrenamiento y personalización de modelos YOLO:** Se han entrenado múltiples modelos de YOLO adaptados a diferentes necesidades del sistema. Además, se ha llevado a cabo la personalización de estos modelos, permitiendo una mayor precisión y rendimiento en la detección de elementos específicos del entorno.

- **Implementación de un sistema externo de detección de barreras:** Se ha desarrollado un mecanismo complementario a la red neuronal para detectar barreras arquitectónicas. Este sistema permite identificar obstáculos que no estaban inicialmente contemplados en los modelos de YOLO, ampliando así el alcance del proyecto más allá de sus objetivos originales.

Estos logros han permitido construir una base sólida para la navegación autónoma y la detección de entornos no accesibles, demostrando la utilidad práctica del sistema en escenarios reales.

6.2 Limitaciones detectadas

Durante el desarrollo del proyecto se han identificado diversas limitaciones, muchas de las cuales han sido mencionadas a lo largo del documento, y que ha afectado al objetivo 3 planteado en el anteproyecto.

La principal limitación ha sido el hardware empleado, concretamente el uso de una Jetson Nano con 2 GB de memoria RAM. Esta capacidad resultó insuficiente para ejecutar de forma simultánea tareas exigentes como la navegación autónoma, el procesamiento de sensores y la inferencia con modelos de YOLO. Aunque se logró implementar un sistema básico de conducción autónoma, este se limitó a comportamientos simples, sin capacidad para realizar mapeo en tiempo real. A pesar de que el script correspondiente al mapeo se encuentra desarrollado, no fue posible probarlo debido a las restricciones de hardware.

Otra limitación significativa se encontró en el entrenamiento de modelos YOLO, que requiere un conjunto de imágenes anotadas en su formato específico. En este proyecto se necesitaban detectar categorías muy concretas, como rampas y ascensores, para las cuales no se encontraron suficientes imágenes disponibles en bibliotecas públicas. Esto generó modelos con escasa capacidad de generalización o sobreajustados, ya que se entrenaron con pocos ejemplos, afectando negativamente la fiabilidad de la detección.

Una alternativa viable habría sido generar un conjunto de datos propio, recopilando imágenes, anotándolas manualmente en el formato YOLO con herramientas específicas y aumentando así la diversidad y volumen del dataset. Sin embargo, este proceso requería una inversión de tiempo considerable, que excedía el marco temporal del proyecto. Por ello, se optó por la solución mediante n8n, tal como se expone en el capítulo correspondiente.

Estas limitaciones no invalidan los logros obtenidos, pero sí marcan el camino hacia futuras mejoras necesarias para alcanzar un sistema más completo, robusto y escalable.

6.3 Conclusión

A la luz de los resultados obtenidos y de todo lo expuesto a lo largo del presente documento, puede afirmarse que la mayoría de los objetivos planteados inicialmente han sido cumplidos o parcialmente alcanzados, ya sea en su forma original o adaptados al contexto y limitaciones del proyecto. A continuación, se detallan cada uno de ellos:

Objetivo 1. Implementar y configurar el entorno de desarrollo ROS sobre la plataforma Jetson Nano, incluyendo los paquetes necesarios para la integración y gestión de sensores, cámaras ToF y control del movimiento.

Este objetivo ha sido cumplido satisfactoriamente, a pesar de las limitaciones impuestas por el hardware. La cámara ToF, aunque inicialmente pensada como el sensor principal para la navegación, demostró ser insuficiente por sí sola. En consecuencia, se añadió un sensor LiDAR que permitió complementar la percepción del entorno, logrando así una integración robusta de ambos sensores junto con el sistema de control del movimiento. Todo ello se ha gestionado de forma coherente dentro del entorno ROS, cumpliendo con los requisitos funcionales previstos.

Objetivo 2. Desarrollar el sistema de detección, clasificación y gestión de obstáculos y barreras arquitectónicas, utilizando sensores.

Este objetivo ha sido también cumplido de manera satisfactoria, logrando no solo implementar la funcionalidad prevista, sino también integrar mejoras y enfoques alternativos que han incrementado el rendimiento general del sistema. Entre estas mejoras destaca la incorporación de un enfoque híbrido de detección, que combina el uso de sensores tradicionales con redes neuronales y métodos externos, alineándose con tendencias actuales en investigación sobre accesibilidad y robótica móvil.

Objetivo 3. Evaluar el sistema en distintos entornos mediante la generación de mapas y pruebas de obstáculos en distintos escenarios.

Este objetivo no pudo cumplirse en su forma original debido a las restricciones de rendimiento de la plataforma Jetson Nano, las cuales imposibilitaron ejecutar procesos simultáneos de mapeo y navegación en tiempo real. Sin embargo, se optó por reformular este objetivo para adaptarlo al marco de trabajo disponible.

Objetivo 3 (reformulado). Diseñar la arquitectura software general del sistema en ROS y desarrollar el código correspondiente para la integración de nodos, la gestión de la información de los sensores y el control del movimiento, incluyendo el diseño de la lógica de conducción autónoma.

Aunque se logró evaluar parcialmente la detección de obstáculos, no fue posible cumplir con el Objetivo 3 en su forma original debido a las limitaciones técnicas encontradas. Por esta razón, dicho objetivo fue reformulado, dando lugar a un nuevo Objetivo 3, el cual sí fue alcanzado con éxito. Este nuevo enfoque no perseguía la validación funcional completa del sistema, sino el diseño e implementación de toda la arquitectura software en ROS, integrando los distintos nodos y componentes —detección, navegación, control de motores, entre otros— de forma estructurada y coherente. El objetivo fue dejar preparado el sistema para facilitar futuras validaciones o migraciones a plataformas con mayor capacidad de procesamiento. Los detalles de esta reformulación y su desarrollo se explican en los capítulos 3 y 4. Como parte de este trabajo, se desarrolló un script de conducción autónoma robusto y modular, capaz de gestionar situaciones complejas y que representa una base sólida sobre la cual podrán construirse futuras ampliaciones del sistema.

En conjunto, estos resultados demuestran que, pese a las restricciones encontradas, el proyecto ha logrado cumplir con sus metas fundamentales, sentando una base sólida para desarrollos futuros.

6.4 Líneas futuras

A partir de las limitaciones identificadas y los logros alcanzados en el presente proyecto, se proponen diversas líneas de trabajo que permitirían ampliar y mejorar el sistema desarrollado. La primera y más inmediata consiste en la ejecución y validación de los scripts diseñados para la conducción autónoma avanzada y la creación de mapas. Aunque dichos scripts han sido implementados, las restricciones de hardware impidieron su correcta evaluación, por lo que sería necesario testear su funcionamiento, corregir posibles errores y realizar los ajustes necesarios.

En segundo lugar, se plantea como línea prioritaria la migración del sistema a ROS 2. Aunque se optó por utilizar ROS Noetic debido a su compatibilidad con el sistema operativo y los dispositivos empleados (como la Jetson Nano), esta versión dejará de recibir soporte oficial en mayo de 2025. Esta migración implicaría identificar los paquetes equivalentes en ROS 2 o, en su defecto, desarrollar nuevas soluciones compatibles, lo que permitiría asegurar la continuidad y escalabilidad del proyecto en el futuro.

Otra línea de trabajo relevante es la profundización en el uso de técnicas de inteligencia artificial, línea ya iniciada en este proyecto. La IA no solo permite la detección de barreras arquitectónicas a partir de imágenes, sino que abre la puerta a nuevas funcionalidades, como la predicción del comportamiento del entorno, la planificación adaptativa del movimiento o el reconocimiento contextual en espacios complejos.

Finalmente, se propone trabajar en el tratamiento y aprovechamiento de los datos recolectados por el robot. Actualmente, el sistema no contempla una infraestructura que permita recibir y procesar estos datos en una plataforma externa. Desarrollar esta capacidad permitiría crear un sistema distribuido, en el cual un dispositivo autónomo se encarga de la recopilación de datos en el entorno, mientras que otro sistema, remoto o en la nube, se encarga del análisis, almacenamiento y visualización de dicha información, facilitando así el uso de los resultados para toma de decisiones o monitorización avanzada.

Bibliografía

[1] F. Accessibilitat, «¿Qué son las barreras arquitectónicas?», Farré Accessibilitat. Accedido: 21 de febrero de 2025. [En línea]. Disponible en: https://farre.es/noticias-accesibilidad/que-son-las-barreras-arquitectonicas/

[2] «What are RGBD cameras? Why RGBD cameras are preferred in some embedded vision applications? - e-con Systems». Accedido: 14 de abril de 2025. [En línea]. Disponible en: https://www.e-consystems.com/blog/camera/technology/what-are-rgbd-cameras-why-rgbd-cameras-are-preferred-in-some-embedded-vision-applications/

[3] «¿Qué es LiDAR? | IBM». Accedido: 14 de abril de 2025. [En línea]. Disponible en: https://www.ibm.com/es-es/topics/lidar

[4] «NVIDIA Jetson Nano», NVIDIA. Accedido: 21 de febrero de 2025. [En línea]. Disponible en: https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-nano/product-development/

[5] P. 3D T. S. S. O. LINE, «¿Qué hace un sensor ToF?», Tofsensors. Accedido: 21 de febrero de 2025. [En línea]. Disponible en: https://tofsensors.com/es-es/blogs/noticias/what-does-a-tof-sensor-do

[6] «Descubre los modelosYOLO Ultralytics | Visión computerizada de vanguardia». Accedido: 7 de marzo de 2025. [En línea]. Disponible en: https://www.ultralytics.com/es/yolo

[7] «ROS: Home». Accedido: 21 de febrero de 2025. [En línea]. Disponible en: https://www.ros.org/

[8] «SLAM (localización y mapeo simultáneos) – MATLAB y Simulink». Accedido: 14 de abril de 2025. [En línea]. Disponible en: https://es.mathworks.com/discovery/slam.html

[9] «noetic - ROS Wiki». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://wiki.ros.org/noetic

[10] «La ULPGC y la Inteligencia Artificial Generativa | Servicio de Informática». Accedido: 14 de abril de 2025. [En línea]. Disponible en: https://si.ulpgc.es/ia/la-ulpgc-y-la-inteligencia-artificial-generativa

[11] 78705808, «El Gerente presenta al Consejo de Gobierno las Cuentas Anuales de la ULPGC 2023», ULPGC - Universidad de Las Palmas de Gran Canaria. Accedido: 14 de abril de 2025. [En línea]. Disponible en: https://www.ulpgc.es/noticia/2024/06/06/gerente-presenta-al-consejo-gobierno-cuentas-anuales-ulpgc-2023

[12] «Azure Kinect DK: desarrollo de modelos de IA | Microsoft Azure». Accedido: 14 de abril de 2025. [En línea]. Disponible en: https://azure.microsoft.com/es-es/products/kinect-dk

[13] T. Ramchuen, *theerawatramchuen/Install-Yolo-V7-on-Jetson-nano*. (25 de abril de 2024). Accedido: 7 de marzo de 2025. [En línea]. Disponible en: https://github.com/theerawatramchuen/Install-Yolo-V7-on-Jetson-nano

[14] «Microsoft 365 online gratuito | Word, Excel y PowerPoint». Accedido: 14 de abril de 2025. [En línea]. Disponible en: https://www.microsoft.com/es-es/microsoft-365/free-office-online-for-the-web

[15] «Master - ROS Wiki». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://wiki.ros.org/Master

[16] «Topics - ROS Wiki». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://wiki.ros.org/Topics

[17] L. Cruz, «ROS (Robot Operating System) — Fundamentos», Medium. Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://medium.com/@robtech.impaciente/ros-robot-operating-system-fundamentos-e92478c26e02

[18] «Sistema ROS para el movimiento de Robots - ATRIA Innovation». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://atriainnovation.com/blog/sistema-ros-para-el-movimiento-de-robots/

[19] Ultralytics, «Inicio rápido de ROS». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://docs.ultralytics.com/es/guides/ros-quickstart

[20] «Navegación Autónoma en ROS», Tknika. Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://tknika.eus/cont/navegacion-autonoma-en-ros/

[21] «rviz - ROS Wiki». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://wiki.ros.org/rviz

[22] «rosbag - ROS Wiki». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://wiki.ros.org/rosbag

[23] *introlab/rtabmap_ros*. (7 de marzo de 2025). C++. IntRoLab. Accedido: 7 de marzo de 2025. [En línea]. Disponible en: https://github.com/introlab/rtabmap_ros

[24] «Nav2 — Nav2 1.0.0 documentation». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://docs.nav2.org/

[25] J. S. Cedrés, M. Á. Quintana Suárez, e I. G. Alonso. González, «Automatización del posicionamiento y seguimiento de trayectorias de un vehículo autónomo basado en ROS sobre planos previamente levantados», TFG-EITE de la ULPGC año 2025.

[26] «YOLO v5 : Autonomous Driving Scenario of JetAuto ROS Robot - Hackster.io». Accedido: 21 de febrero de 2025. [En línea]. Disponible en: https://www.hackster.io/490575/yolo-v5-autonomous-driving-scenario-of-jetauto-ros-robot-037583

[27] «TurtleBot». Accedido: 21 de febrero de 2025. [En línea]. Disponible en: https://www.turtlebot.com/

[28] «Python 3.0 Release», Python.org. Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://www.python.org/download/releases/3.0/

[29] «¿Qué es el deep learning? | IBM». Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://www.ibm.com/es-es/topics/deep-learning

[30] «CUDA Toolkit - Free Tools and Training», NVIDIA Developer. Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://developer.nvidia.com/cuda-toolkit

[31] «CUDA Deep Neural Network», NVIDIA Developer. Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://developer.nvidia.com/cudnn

[32] «TensorRT SDK | NVIDIA Developer». Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://developer.nvidia.com/tensorrt

[33] «TensorBoard | TensorFlow». Accedido: 29 de marzo de 2025. [En línea]. Disponible en: https://www.tensorflow.org/tensorboard?hl=es-419

[34] «PyTorch», PyTorch. Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://pytorch.org/

[35] «Home», OpenCV. Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://opencv.org/

[36] «MAS.865 2018 How to Make Something that Makes (almost) Anything». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://fab.cba.mit.edu/classes/865.18/scanning/slam/

[37] «Referencias inerciales - IMUs - AHRS archivos», Sensing, Sensores de Medida. Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://sensores-de-medida.es/medicion/sensores-y-transductores/giroscopos-y-referencias-inerciales/referencias-inerciales-imus-ahrs/

[38] «gmapping - ROS Wiki». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://wiki.ros.org/gmapping

[39] «cartographer - ROS Wiki». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://wiki.ros.org/cartographer

[40] «RPLIDAR S2 - Advanced Triangulation LIDAR for Versatile Use», Gargantua.ai. Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://gargantua.ai/product/gargantua-tof-lidar-rplidar-s2/

[41] «News - Basic Principle and application of TOF(Time of Flight) System», https://www.lumispot-tech.com/. Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://www.lumispot-tech.com/news/tof-time-of-flight-definition-and-principle/

[42] «Red neuronal convolucional (CNN) | TensorFlow Core», TensorFlow. Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://www.tensorflow.org/tutorials/images/cnn?hl=es-419

[43] «¿Qué es R-CNN? Un resumen rápido». Accedido: 29 de marzo de 2025. [En línea]. Disponible en: https://www.ultralytics.com/es/blog/what-is-r-cnn-a-quick-overview

[44] «How single-shot detector (SSD) works?», ArcGIS API for Python. Accedido: 29 de marzo de 2025. [En línea]. Disponible en: https://developers.arcgis.com/python/latest/guide/how-ssd-works/

[45] «YOLOv7 How to Use ? – Best Tutorial simple». Accedido: 15 de abril de 2025. [En línea]. Disponible en: https://inside-machinelearning.com/en/use-yolov7/

[46] «docs.arduino.cc/hardware/uno-rev3». Accedido: 17 de abril de 2025. [En línea]. Disponible en: https://docs.arduino.cc/hardware/uno-rev3/

[47] «NVIDIA Jetson AGX Orin», NVIDIA. Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-orin/

[48] R. P. Ltd, «Raspberry Pi», Raspberry Pi. Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://www.raspberrypi.com/

[49] «BeagleBone® AI», BeagleBoard. Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://www.beagleboard.org/boards/beaglebone-ai

[50] «STM32 Microcontrollers (MCUs) - STMicroelectronics». Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

[51] «Depth Camera D435i – Intel® RealSense™ Depth and Tracking Cameras». Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://www.intelrealsense.com/depth-camera-d435i/

[52] «Astra Series - ORBBEC - 3D Vision for a 3D World». Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://www.orbbec.com/products/structured-light-camera/astra-series/

[53] «RPLIDAR A2: Affordable 2D LIDAR for Robotics and Mapping», Gargantua.ai. Accedido: 16 de abril de 2025. [En línea]. Disponible en: https://gargantua.ai/product/gargantua-triangulation-lidar-rplidar-a2/

[54] L. Llamas, «Controlar dos motores DC con Arduino y driver TB6612FNG», Luis Llamas. Accedido: 17 de abril de 2025. [En línea]. Disponible en: https://www.luisllamas.es/arduino-motor-dc-tb6612fng/

[55] R. Rodriguez, «Ruedas Mecanum: ¿Qué son?, ventajas y aplicaciones», Slind Parts. Accedido: 17 de abril de 2025. [En línea]. Disponible en: https://www.slindparts.com/que-son-las-ruedas-mecanum/

[56] «Open Images V7». Accedido: 17 de abril de 2025. [En línea]. Disponible en: https://storage.googleapis.com/openimages/web/index.html

[57] «Roboflow: Computer vision tools for developers and enterprises». Accedido: 17 de mayo de 2025. [En línea]. Disponible en: https://roboflow.com/

[58] «urdf - ROS Wiki». Accedido: 30 de mayo de 2025. [En línea]. Disponible en: https://wiki.ros.org/urdf

[59] «Robot Operating System: How to Create a Robot Simulation Model», Admantium. Accedido: 30 de mayo de 2025. [En línea]. Disponible en: https://admantium.com/blog/ros04_urdf_tutorial_1/

[60] *pyransac3d: A python tool for fitting primitives 3D shapes in point clouds using RANSAC algorithm*. Python. Accedido: 19 de mayo de 2025. [OS Independent]. Disponible en: https://github.com/leomariga/pyRANSAC-3D

[61] «RPubs - Random sample consensus». Accedido: 19 de mayo de 2025. [En línea]. Disponible en: https://rpubs.com/silvosus/752943

[62] «Leonard148/TFG at completo». Accedido: 30 de mayo de 2025. [En línea]. Disponible en: https://github.com/Leonard148/TFG/tree/completo

[63] «Powerful Workflow Automation Software & Tools - n8n». Accedido: 20 de mayo de 2025. [En línea]. Disponible en: https://n8n.io/

[64] «AI Agents Solutions | IBM». Accedido: 20 de mayo de 2025. [En línea]. Disponible en: https://www.ibm.com/ai-agents

[65] «Docker: Accelerated Container Application Development». Accedido: 20 de mayo de 2025. [En línea]. Disponible en: https://www.docker.com/

[66] «MQTT - The Standard for IoT Messaging». Accedido: 20 de mayo de 2025. [En línea]. Disponible en: https://mqtt.org/

[67] «Honorarios profesionales LE». Accedido: 21 de mayo de 2025. [En línea]. Disponible en: https://www.telecos.zone/libre-ejercicio/honorarios-profesionales-le

[68] Miluska.Jara, «Objetivos y metas de desarrollo sostenible», Desarrollo Sostenible. Accedido: 6 de junio de 2025. [En línea]. Disponible en: https://www.un.org/sustainabledevelopment/es/sustainable-development-goals/

Presupuesto

A continuación, se expone una estimación detallada de los costes asociados a la ejecución de este proyecto. Dado que se trata de un Trabajo Fin de Grado (TFG) enmarcado en un contexto académico y formativo, el presupuesto ha sido elaborado tomando como referencia las tarifas más actualizadas publicadas por el Colegio Oficial de Graduados e Ingenieros Técnicos de Telecomunicación (COITT) [67].

Es importante destacar que, en el ámbito profesional, la evaluación económica de un proyecto debe ajustarse a las indicaciones del Ministerio de Economía y Hacienda, que ha instado a los colegios profesionales a eliminar los baremos orientativos tradicionales en cumplimiento de las directivas europeas. Sin embargo de manera orientativa se ha utilizado su baremo orientativo que se calcula en función de las horas invertidas y factores de reducción.

Este apartado presenta una aproximación al coste derivado de la realización del TFG, considerando los siguientes criterios:

- Amortización de los recursos materiales utilizados
- Coste del trabajo calculado según el tiempo invertido
- Elaboración de la documentación del proyecto
- Aplicación de impuestos y cálculo del coste total

P1. Amortización de recursos materiales

Aquí se analiza el uso de los recursos materiales empleados en el desarrollo del proyecto, tanto en términos de hardware como de software. Para la estimación del coste, se aplica un método de amortización lineal, que asume una depreciación uniforme del valor del material a lo largo de su vida útil. Aunque el ciclo de amortización estándar es de cuatro años, el periodo real de desarrollo del TFG ha sido de cuatro meses, por lo que los cálculos se han ajustado proporcionalmente a este intervalo.

P1.1 Amortización del material hardware

El desarrollo del TFG ha tenido una duración de cuatro meses, lo cual representa un periodo considerablemente más corto que los tres años habitualmente tomados como referencia para la amortización del hardware. Por ello, los costes de amortización reflejan únicamente el valor correspondiente a esos cuatro meses de uso.

La Tabla 2 recoge los principales componentes de hardware empleados durante el proyecto, incluyendo su coste de adquisición y la amortización correspondiente al periodo considerado.

Tabla 3 Tabla de amortización de los recursos hardware

Elemento	Valor de adquisición	Amortización
Ordenador del entrenamiento de YOLO	3.834,38 €	318,25 €
Jetson Nano 2Gb RAM	110 €	9,13 €
Kit Robot Mecanum V3	208,65 €	17,32 €
Kit de separadores de nailon	9,05 €	0,75 €
RPLIDAR S2	665,06 €	55,2 €
Batería HRB 4000 mAh	42,77 €	3,55 €
Placas de aluminio (x2)	41 €	3,4 €
Ordenador personal de trabajo	1.155,98 €	95,94 €
Azure Kinect DK	620,60 €	51,51 €
Total		555.05 €

Se está calculando el porcentaje de amortización proporcional al tiempo de uso del equipo durante el proyecto, concretamente durante 4 meses de los 36 meses (3 años) que normalmente se consideran como vida útil del hardware.

Cálculo del porcentaje:

$$\frac{4\ Meses}{48\ Meses} = \frac{1}{12} \approx 0.8333 = 8.3\%$$

Aplicación sobre el valor del equipo:

$$\text{Valor del equipo} * \ 8{,}3\% = \text{Valor amortizado}$$

El coste total de amortización para los elementos de hardware durante el periodo de desarrollo asciende a **quinientos cincuenta y cinco euros con cinco centimos (555.05 €).**

P1.2 Amortización del material software

En el caso del software utilizado en este Trabajo de Fin de Grado, el cálculo de amortización se basa igualmente en el uso durante 4 meses dentro de un periodo estándar de 3 años. La mayoría del software empleado ha sido de carácter libre y de código abierto, o bien ha sido facilitado por la ULPGC mediante licencias académicas. En particular, se ha utilizado el sistema operativo Ubuntu 20.04 y la plataforma ROS Noetic, ambos completamente gratuitos, lo que ha permitido desarrollar el proyecto sin incurrir en costes adicionales por licencias. Por este motivo, los gastos totales asociados al software ascienden a cero euros (0 €).

P2. Trabajo tarifado por tiempo empleado

Para realizar este proyecto se han invertido alrededor de 300 horas en diseño, desarrollo y elaboración de documentación. Para calcular el valor del trabajo empleado se ha elegido la siguiente ecuación**:**

$$H = C_t * \ 74{,}88 * \ H_n + \ C_t * 96.72 * H_e$$

Siendo:

- H: Honorarios totales recibidos por el proyecto
- $\boldsymbol{Ct}$**:** Factor de corrección dependiendo del número de horas trabajadas
- Hn: Horas trabajadas en horario laboral
- $\boldsymbol{He}$**:** Horas trabajadas fuera del horario laboral (en este proyecto no han existido así que su valor es 0)

Teniendo en cuenta una posible distribución del factor de corrección descrito en la siguiente tabla, el valor es de 0.60.

Tabla 4 Valores del factor de corrección en función a las horas trabajadas

Horas empleadas	Factor de corrección Ct
X < 36	**1**
36 < X < 72	0,90
72 < X < 108	0,80
108 < X < 144	0,70
144 < X < 180	0,65
180 < X < 360	0,60
360 < X < 540	0,55

Según esta tabla, al estar este proyecto finalizado en 300 horas, se deberá escoger el factor de corrección con un valor de 0.60. Dado esto, la formula anterior queda de la siguiente manera:

$$H = 0{,}60 * \ 74{,}88 * \ 300 + \ 0{,}60 * 96.72 * 0 = 13.478{,}40 \ €$$

Los honorarios derivados al tiempo dedicado al proyecto libre de impuestos ascienden a unos **trece mil cuatrocientos setenta y ocho euros con cuarenta céntimos (13.478,40 €).**

P3. Redacción de documentación

Con respecto al coste de la redacción del documento se utiliza la ecuación**:**

$$H = 0{,}07 * P * C_n$$

Teniendo en cuenta que:

- $\boldsymbol{R}$**:** Honorarios por la redacción del documento.
- P: es el presupuesto del trabajo.
- Cn, es el coeficiente de ponderación en función del presupuesto.

El coste final es igual a la suma de los costes de trabajo tarifado por tiempo empleado, calculado anteriormente, y de la amortización de recursos materiales cuya sume se puede observar en la Tabla 4.

Tabla 5 Presupuesto del trabajo tarifado y amortización de los recursos materiales

Descripción	Costes
Amortización de recursos materiales	555.05 €
Trabajo tarifado por tiempo empleado	13.478,40 €
Total	**14.033,45 €**

Debido a que el coeficiente de ponderación para presupuestos menores de 30.050,00€ le estamos asignando el valor de 1.00, el coste de la redacción de documento del TFG es de:

$$H = 0{,}07 * 13.730{,}61 * 1 = 961{,}14\,€$$

Finalmente, el coste de la redacción del proyecto se queda con un valor de **novecientos sesenta y un euros con catorce céntimos (961,14 €).**

P4. Aplicación de impuestos y coste total

Al desarrollo de este Trabajo de Fin de Grado se le aplica el Impuesto General Indirecto Canario (IGIC), que se corresponde con el 7% del valor del presupuesto. El presupuesto total del proyecto se recoge en la Tabla 5.

Concepto	Coste
Amortización de recursos materiales	555.05 €
Trabajo tarifado por tiempo empleado	13.478,40 €
Redacción de documentación	961,14 €
Subtotal (Sin IGIC)	14.994,59 €
IGIC (7%)	1.049,62 €
Total	**16.044,21 €**

El trabajo de Fin de Grado con título "**Sistema de detección de obstáculos mediante ROS y procesamiento embebido en Jetson Nano**" desarrollado en la Escuela de Ingeniería de Telecomunicaciones y Electrónica, de la Universidad de las Palmas de Gran Canaria, tiene un coste de desarrollo total de **dieciséis mil cuarenta y cuatro euros con veintiun céntimos (16.044,21 €),** correspondiente a la suma de las cantidades consignadas a los apartados considerados previamente.

Las Palmas de Gran Canaria, a X de junio de 2025

Firma: Airam López Mendoza

Firmado por LOPEZ MENDOZA AIRAM - ***9211** el día 10/06/2025 con un certificado emitido por

Objetivos de Desarrollo Sostenible

ODS	Grado de relación con los ODS			
	0 No procede	1 Bajo	2 Medio	3 Alto
ODS 1 Fin de la Pobreza	◉	○	○	○
ODS 2 Hambre cero	◉	○	○	○
ODS 3 Salud y Bienestar	○	◉	○	○
ODS 4 Educación de calidad	◉	○	○	○
ODS 5 Igualdad de género	◉	○	○	○
ODS 6 Agua limpia y saneamiento	◉	○	○	○
ODS 7 Energía Asequible y no contaminante	◉	○	○	○
ODS 8 Trabajo decente y crecimiento económico	◉	○	○	○
ODS 9 Industria, Innovación e Infraestructuras	◉	○	○	○
ODS 10 Reducción de las desigualdades	○	◉	○	○
ODS 11 Ciudades y comunidades sostenibles	◉	○	○	○
ODS 12 Producción y consumo sostenibles	◉	○	○	○
ODS 13 Acción por el clima	◉	○	○	○
ODS 14 Vida submarina	◉	○	○	○
ODS 15 Vida de ecosistemas terrestres	◉	○	○	○
ODS 16 Paz, justicia e instituciones sólidas	◉	○	○	○
ODS 17 Alianzas para lograr objetivos	◉	○	○	○

Ilustración 24 Tabla con los objetivos de desarrollo sostenible [68]

La explicación de esta tabla es la siguiente:

Dado que el objetivo del trabajo es lograr que, a través de una cámara ToF (Time of Flight), se pueda reconocer el entorno y procesar dicha información, en concreto las barreras arquitectónicas, esta tecnología podría ser aplicada para asistir a personas con dificultades para identificar su entorno, brindándoles notificaciones o alertas relevantes.

ANEXOS

1 Reproducción del proyecto

En este apartado del anexo se detallan los pasos necesarios para emular el entorno de desarrollo y ejecución utilizado en este proyecto, orientado a futuros trabajos o réplicas que deseen realizarse sobre una plataforma Jetson Nano. Se explican desde la instalación del sistema operativo hasta las consideraciones necesarias respecto al hardware utilizado.

1. Selección e instalación del sistema operativo

La Jetson Nano permite dos métodos oficiales para instalar un sistema operativo:

Uso de tarjeta microSD: Es el método empleado en este TFG. Se descarga la imagen correspondiente y se graba en una tarjeta microSD de al menos 64 GB utilizando herramientas como balenaEtcher o Raspberry Pi Imager.

Uso del SDK Manager de NVIDIA: Requiere un ordenador adicional con sistema operativo Linux. Con este software se puede iniciar el flasheo del sistema conectando la Jetson en modo recuperación (Force Recovery Mode). Esta opción instala un sistema operativo oficial de NVIDIA, basado igualmente en Ubuntu.

Se recomienda utilizar una versión del sistema operativo basada en Ubuntu 20.04, ya que esta es la base requerida para instalar ROS Noetic, utilizado en el presente trabajo.

2. Instalación de ROS Noetic

Una vez instalado el sistema, se procede con la instalación de ROS Noetic, siguiendo los pasos detallados en la documentación oficia.

Tras completar la instalación de ROS, se debe crear el espacio de trabajo (workspace) habitual, llamado catkin_ws:

mkdir -p ~/catkin_ws/src

cd ~/catkin_ws

catkin_make

Para cargar este entorno de forma automática al iniciar una nueva terminal, se añade al archivo .bashrc:

echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc

source ~/.bashrc

3. Clonado y compilación del proyecto

Con el workspace creado, se clona el repositorio del proyecto desde GitHub en la carpeta src:

cd ~/catkin_ws/src

git clone <URL_del_repositorio>

A continuación, se vuelve al directorio raíz y se compila:

cd ~/catkin_ws

catkin_make clean

catkin_make

4. Posibles errores durante la compilación

Durante la compilación del workspace pueden aparecer diversos errores, que se dividen principalmente en dos categorías:

Errores por estructura del proyecto: Los archivos CMakeLists.txt y package.xml definen la estructura esperada. Si esta estructura ha sido alterada o no coincide exactamente con la original, será necesario modificarlos para ajustarlos.

Errores por dependencias: Pueden surgir errores por bibliotecas o paquetes de ROS ausentes o incompatibles. Para resolverlos se recomienda consultar detenidamente el log de errores que se muestra tras la compilación, e instalar las dependencias requeridas. Esto puede hacerse mediante los siguientes comandos:

sudo apt-get install ros-noetic-<paquete-faltante>

o en algunos casos:

pip install <librería>

5. Consideraciones de hardware

Es fundamental tener en cuenta el hardware empleado en el desarrollo original, ya que algunos nodos están configurados específicamente para estos dispositivos:

Cámara (Azure Kinect DK): Si no se dispone del mismo modelo, no es necesario lanzar el nodo específico de este proyecto. No obstante, se recomienda analizar su estructura, ya que en él se centraliza el lanzamiento de todos los sensores. Además, la instalación del SDK de la cámara es independiente del código y debe hacerse manualmente.

Sensor LiDAR (RPLIDAR): El nodo utilizado (rplidar_ros) soporta múltiples modelos de LiDAR, pero será necesario modificar su configuración para adaptarse a la versión concreta del sensor utilizado.

Plataforma motriz: Si se emplea una placa distinta para el control de motores (como un controlador diferente o un sistema de comunicación alternativo), también se debe modificar el nodo encargado de esta interacción, de modo que se adapte a la nueva conexión (UART, I2C, etc.).

Este conjunto de pasos permite replicar el entorno de desarrollo original del proyecto, manteniendo la flexibilidad para adaptarlo a diferentes configuraciones de hardware o software siempre que se respete la compatibilidad entre versiones.

2 Promts de IA Generativa usados

Estos son algunos de los promts usados en la IA generativa:

- Analiza el código, tanto a nivel de redundancias, cosas que tenga mal o mejorables. No quiero que añadas nada que no sea para solucionar un error.
- Coge las siguientes barreras arquitectónicas y soluciones a ellas y ponlo en formato de tabla.

- No es un promt al uso, pero se le ha pasado código externo con el fin de que lo analizara y explicara, útil para conocer el funcionamiento del código del Arduino dado por el fabricante, o librerías que se usan en ROS, como el Lidar.
- Dame las diferencias entre ROS 1, ROS 2.
- Ponte en el punto de vista de una persona con movilidad reducida. Teniendo esto en cuenta, devuélveme una descripción del entorno referente a problemas que puedo encontrar. No añadas nada como si fueras una persona.