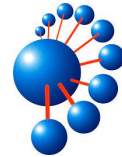




DEPARTAMENTO DE INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE CONCEPCIÓN



LOCALIZACIÓN E IDENTIFICACIÓN DE OBJETOS EN TIEMPO REAL CON ROBOT TURTLEBOT3

POR
PAMELA CARINA PINILLA BANCHERO

Memoria presentada para la obtención del título de
INGENIERA CIVIL INFORMÁTICA

Patrocinantes: JULIO ERASMO GODOY DEL CAMPO
IGNACIO SANHUEZA RUIZ

Concepción, Septiembre 2025

Agradecimientos

Gracias a Valentina y a Miguel por acompañarme en los momentos lindos y no tan lindos.

Gracias a mi mamá, mi papá, mi hermano y mi primo, los amo.

Gracias a mis amistades que siempre están ahí.

Y gracias a mí por lograr todo esto y más.

Índice

1. Introducción	4
1.1. Objetivo general y específicos	5
1.1.1. Objetivo General	5
1.1.2. Objetivos Específicos	5
2. Revisión bibliográfica	6
2.1. Turtlebot3	6
2.2. NVIDIA Jetson Nano	8
2.3. ROS 2	9
2.4. SLAM	11
2.5. Navegación con ROS	12
2.6. Docker	13
2.7. Protocolo de comunicación UDP	14
2.8. Detección de objetos con YOLOv5	14
2.9. Visión robótica	16
3. Metodología	17
3.1. Ensamblado del robot Turtlebot3	18
3.2. Instalación de dependencias	20
3.2.1. Instalación de ROS 2	21
3.2.2. Instalación de SLAM Toolbox y Navigation2	22
3.2.3. Configurar Docker e instalar YOLOv5	22
3.3. Detección de objetos en tiempo real	24
3.3.1. Transmisión de datos a través de protocolo UDP	24
3.3.2. Implementación de la detección de objetos	25
3.4. Creación de nodos	25
3.4.1. Nodo que recibe información	26
3.4.2. Nodo que localiza objetos en el mapa	26
3.4.3. Nodo de navegación	26
3.5. Flujo de trabajo	27
4. Experimentos y resultados	30
4.1. Mapa 1	32
4.1.1. Prueba 1: Un objeto	32
4.1.2. Prueba 2: Dos objetos	33
4.1.3. Prueba 3: Dos objetos	35
4.1.4. Prueba 4: Tres objetos	36
4.1.5. Prueba 5: Tres objetos	38
4.1.6. Resumen general de análisis cuantitativo (Mapa 1)	39
4.2. Mapa 2	40
4.2.1. Prueba 1: Dos objetos	40
4.3. Otros experimentos	41
4.3.1. Prueba 1 anexa: Mapa 1, dos objetos	41
4.3.2. Prueba 2 anexa: Mapa 2, dos objetos	42
4.4. Análisis de los resultados	43

5. Conclusiones	45
5.1. Conclusiones por ejes evaluados	45
5.2. Limitaciones	45
5.3. Trabajos Futuros	46
5.4. Experiencia y aprendizajes	47
Bibliografía	48
6. Anexo	52
6.1. Descarga y configuración de contenedor de Docker	52
6.2. Código de prueba de cámara dentro del contenedor <i>camera_test.py</i>	53
6.3. Código de detección de objetos <i>detect.py</i>	54
6.4. Script de respaldo de contenedor de Docker <i>dockerRespaldo.sh</i>	58
6.5. Código de nodo que recibe información de YOLOv5 <i>listener_yolov5.py</i>	60
6.6. Código de nodo que localiza objetos en el mapa <i>map_updater.py</i>	61
6.7. Código de nodo para que el robot navegue <i>navigator.py</i>	65

1. Introducción

Hoy en día, gradual y progresivamente, la robótica cognitiva se ha convertido en un aspecto fundamental en la vida cotidiana de las personas. Este campo combina métodos de la robótica, la inteligencia artificial y las ciencias cognitivas para diseñar sistemas completos de robots inteligentes [8]. Por ejemplo, las aspiradoras Roomba de IRobot (Ver figura 1.a) [29] emplean una combinación de elementos de la robótica cognitiva para mapear habitaciones e incluso hogares completos, lo que les permite realizar una limpieza eficiente. Estas aspiradoras integran una serie de sensores que facilitan la realización de la tarea. Además, existen vehículos autónomos como Waymo de Google (Ver figura 1.b) [20], que gracias a la robótica cognitiva, contribuyen a reducir los accidentes de tráfico y las muertes relacionadas con estos [58].



(a) Aspiradora Roomba. Obtenido desde [29].



(b) Auto autónomo Waymo. Obtenido desde [20]

Figura 1: Robots en la vida diaria

Por otro lado, la robótica cognitiva podría abordar otras necesidades de las personas, como proporcionar asistencia a personas enfermas entregándoles medicamentos donde se encuentren, servir como asistentes para tareas cotidianas en el hogar a adultos mayores, personas con movilidad reducida o simplemente personas que necesiten ayuda. Un ejemplo concreto de esto es el sistema presentado por Guda et al. (2024) [18], un robot portátil y adaptable que asiste en la alimentación de personas con movilidad reducida. Este sistema puede montarse en una silla de ruedas y controlarse a través de una aplicación web, utilizando visión por computador para identificar alimentos y detectar la boca del usuario, lo que permite una alimentación autónoma, segura y centrada en las necesidades reales del usuario.

Las soluciones presentes en la literatura para asistir a personas con movilidad reducida, en la mayoría de los casos, se limitan al uso de un brazo robótico o de algún sistema robótico asistido, cuyo espacio de trabajo es restringido y no permite el desplazamiento hacia el lugar donde se encuentra la persona que requiere atención. Por esta razón, en la presente memoria de título se propone una solución alternativa, en la cual el robot asistente es capaz de desplazarse por el entorno mediante navegación semántica autónoma [14], la cual permite al robot entender el entorno de una forma parecida al humano, pudiendo desplazarse por espacios habitados por personas, esquivando obstáculos, entendiendo que existe un entorno, y de esta forma asistir al usuario en cualquier ubicación en la que se encuentre. Para ello, se utiliza un robot TurtleBot3 de tamaño mediano, modelo “Waffle”, junto con una cámara web conectada por USB y un dispositivo NVIDIA Jetson Nano para la detección e identificación de objetos en tiempo real. Con estos componentes, se desarrolla un sistema en el cual el robot debe construir un mapa asociando su ubicación dentro de él, la cual se desarrolla con SLAM [15], al mismo tiempo que detecta y localiza los objetos en el mapa. Posteriormente, el usuario puede solicitar que el robot se dirija hacia un objeto específico, ante lo cual el sistema genera una ruta para que el robot navegue y alcance el objeto solicitado.

1.1. Objetivo general y específicos

1.1.1. Objetivo General

Implementar una solución robótica para que el TurtleBot3 pueda cumplir una función de asistencia simulada, reconociendo objetos, asociándolos a una ubicación en un mapa generado por el robot y planeando rutas para llegar a un objeto seleccionado por el usuario.

1.1.2. Objetivos Específicos

- Lograr que el TurtleBot3 logre realizar un ambiente previamente mapeado con ayuda de la cámara Realsense d435.
- Lograr que el TurtleBot3 se localice en un mapa.
- Lograr que el TurtleBot3 asocie objetos a etiquetas.
- Asociar ubicación de los objetos en el mapa.
- Implementar A* integrado con ROS2 para planificar la ruta desde el robot hasta el objeto seleccionado por el usuario.
- Integrar los componentes desarrollados en la solución final.

2. Revisión bibliográfica

En el desarrollo de sistemas robóticos autónomos, la capacidad de percibir el entorno, identificar objetos y navegar de forma eficiente hacia ellos representa un desafío clave dentro del campo de la robótica cognitiva. Estos sistemas requieren la integración de diversas disciplinas como la visión computacional, la navegación autónoma, entre otras tecnologías que se articulan a través de arquitecturas de software como el Robot Operating System 2 [34] (ROS 2).

Se han desarrollado proyectos en navegación semántica [14], en los cuales se intenta que los robots entiendan e interactúen con su entorno de manera más humana, utilizando robots Turtlebot3. Por ejemplo, *TurtleBot3 Object Tracker* [17], que utiliza la versión anterior de ROS 2, ROS, para crear un paquete donde el robot puede seguir objetos utilizando YOLO [46]. Otro proyecto relacionado es *Real-Time Object Detection and Navigation with TurtleBot3 and Qualcomm RB5* [4] el cual integra múltiples tecnologías, como SSD MobileNet model, para la detección de objetos en tiempo real, Whisper model para la transcripción de audio y ROS para la integración, las cuales permiten que el TurtleBot3 pueda identificar, localizar y navegar hacia objetos especificados mediante comandos de voz. Por último, *TurtleBot3 Red Brick Finding* [5] es un proyecto donde el TurtleBot puede explorar de forma autónoma un laberinto, ya sea desconocido o preexplorado, utilizando sensores integrados para detectar ladrillos rojos.

El presente marco teórico aborda las tecnologías utilizadas en esta memoria de título, tomando como referencia proyectos desarrollados con robots TurtleBot3. Los casos anteriormente mencionados ilustran cómo es posible implementar soluciones efectivas de detección de objetos en tiempo real y navegación autónoma hacia objetivos detectados, utilizando herramientas como OpenCV, modelos de deep learning como YOLO, y técnicas de planificación y navegación.

2.1. Turtlebot3

Turtlebot [39] es una plataforma de robótica estandarizada desarrollada por ROS (Robotic Operative Sistem) para la educación e investigación. Existen cuatro versiones de Turltebot, entre ellas Turtlebot3 [36], que es una plataforma de robótica estandarizada, desarrollada por Robotis. Estos robots son pequeños, asequibles y personalizables, de bajo costo, de desarrollo flexible sin tener que sacrificar funcionalidad y calidad. Entre los Tutlebot3 existen dos modelos diferentes: *burger* y *waffle pi*. Para efectos de esta memoria de título se utilizará el modelo *waffle pi* (Ver Figura 2), debido a que su mayor tamaño facilita la reubicación de sus componentes, lo cual resulta beneficioso para el desarrollo del proyecto.

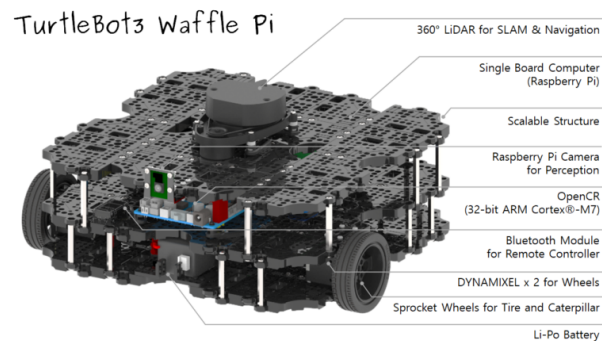


Figura 2: Robot waffle pi y sus componentes. Obtenido desde [36].

	Nombre	Cantidad
Motores	DYNAMIXEL (XM430-W210-T)	2
Boards	OpenCR1.0	1
Boards	Raspberry Pi	1
Boards	USB2LDS	1
Controles remotos	BT-410 Set (Bluetooth 4, BLE)	1
Controles remotos	RC-100B (Remote Controller)	1
Sensores	LDS-01	1
Memoria	MicroSD Card	1
Powers	SMPS 12V5A	1
Powers	LIPO Battery 11.1V 1,800mAh	1
Powers	LIPO Battery Charger	1

Cuadro 1: Especificaciones técnicas Turtlebot3

El robot Turtlebot3 waffle pi está equipado con un módulo de control de código abierto llamado OpenCR [38], el cual posee múltiples sensores y dos tipos diferentes de entradas de alimentación intercambiables entre sí.

Los sensores que posee el OpenCR son los siguientes:

- **Giroscopio de 3 ejes (Velocidad angular):** La función del giroscopio es medir la velocidad angular en los tres ejes espaciales, para que así el robot pueda detectar y controlar su orientación y movimientos rotacionales.
- **Acelerómetro de 3 ejes (Inclinación):** Detecta la aceleración lineal en los ejes midiendo la dirección de gravedad, midiendo movimientos lineales y cambios en la velocidad del robot.
- **Procesador de movimiento digital (DMP en inglés) – ICM-20648:** Es un procesador interno especializado que fusiona datos del acelerómetro y del giroscopio para ofrecer una estimación más precisa de la orientación y los movimientos en tiempo real.

Todos estos elementos conforman lo que se conoce como *Unidad de Medida Inercial (IMU)* que permite estimar la orientación (ángulos de rotación como roll, pitch, yaw) del robot y su movimiento en el espacio. La IMU es fundamental para la navegación, el control de estabilidad, el SLAM o para mantener trayectorias suaves.

Otro componente importante del TurtleBot3 son los motores DYNAMIXEL, los cuales incorporan encoders, un tipo de sensor que permite medir cuántas vueltas ha dado el eje del motor. Esta información es leída por la placa OpenCR, que, combinándola con parámetros como el radio de las ruedas y la distancia entre ellas, calcula la *odometría* del robot, la cual corresponde a una estimación de la posición y orientación del robot, basada en el movimiento de sus ruedas.

La CPU que viene por defecto con el Turtlebot3 es la Raspberry Pi 3 Model B+ [30] (Ver figura 3). Esta contiene un sistema operativo basado en Linux (Raspberry Pi OS) y es muy útil para utilizarlo en proyectos de robótica.



Figura 3: Raspberry Pi 3 Model B+. Obtenido desde [30]

Por último, el Turtlebot3 contiene un sensor de distancia láser 2D llamado LiDAR (LDS-01) [37] (Ver figura 4) capaz de detectar información en 360 grados. Estos datos, el robot los recopila y los utiliza para realizar mapas y navegar en ellos.



Figura 4: LiDAR LDS-01. Obtenido desde [37].

2.2. NVIDIA Jetson Nano

NVIDIA Jetson Nano [12] (Ver figura 5) es un computador pequeño, cuya arquitectura es ARM64 [13] (Cuadro 2), que facilita el desarrollo de diferentes proyectos que tengan relación con Inteligencia Artificial (IA); entre ellas, permite ejecutar múltiples redes neuronales en paralelo, clasificar imágenes, detectar objetos, segmentar y procesar voz.



Figura 5: NVIDIA Jetson Nano. Obtenido desde [12].

GPU	NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores
CPU	Quad-core ARM Cortex-A57 MP-Core processor
Memoria	4 GB 64-bit LPDDR4, 1600MHz 25.6 GB/s
Almacenamiento	16 GB eMMC 5.1

Cuadro 2: Especificaciones técnicas Jetson Nano

Para mejorar el rendimiento de las aplicaciones en la Jetson Nano existe CUDA [11], que es una plataforma de computación paralela de NVIDIA que permite ejecutar tareas en la GPU para mejorar el rendimiento de aplicaciones de inteligencia artificial, visión artificial y procesamiento de datos. También, CUDA es compatible con ROS 2, ayudando a optimizar el procesamiento en robots autónomos. Además, la GPU de la Jetson Nano está diseñada para trabajar a base de la eficiencia energética, promoviendo el rendimiento en su máximo potencial.

Jetson Nano incluye, además, varias bibliotecas optimizadas para trabajar con CUDA, entre ellas se mencionan las más relevantes para este proyecto:

1. OpenCV [7]: es una biblioteca de visión por computador la cual se utiliza en tareas como: detección de rostros, filtros de imágenes, seguimiento de objetos y procesamiento de video en tiempo real. Al usar OpenCV con CUDA se pueden acelerar significativamente operaciones relacionadas a la biblioteca.
2. Pythorch [42]: es una biblioteca de código abierto para deep learning el cual puede utilizar CUDA para acelerar cálculos en la GPU de la Jetson Nano con una versión específica para la arquitectura ARM.
3. Torchvision: es una librería de PyTorch para procesamiento de imágenes y modelos preentrenados. Las transformaciones o procesamientos de imágenes pueden utilizar CUDA para obtener mayor eficiencia. YOLO puede ejecutarse con aceleración de GPU en la Jetson Nano.

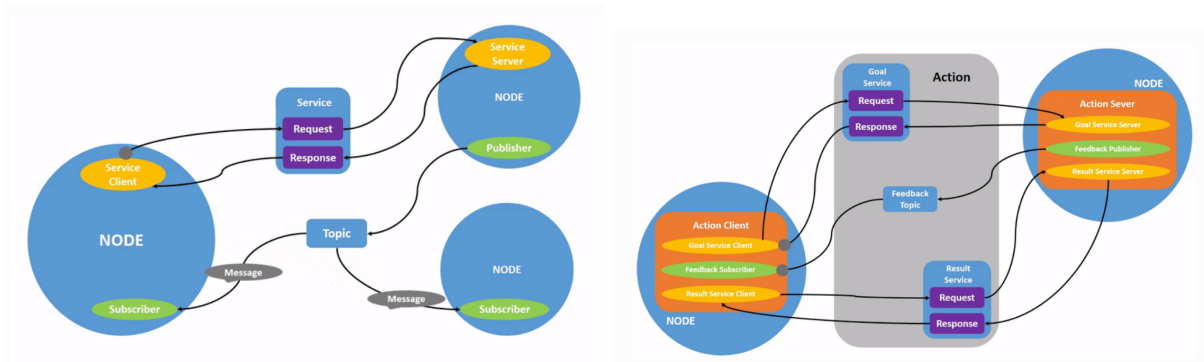
2.3. ROS 2

Robot Operating System (ROS) [34] es una herramienta de código abierto que posee un conjunto de bibliotecas de software y herramientas para desarrollar aplicaciones robóticas. ROS 2, que es la segunda versión de ROS, actúa como un middleware flexible que permite el diseño distribuido y modular de sistemas complejos, promoviendo la reutilización de componentes y la escalabilidad de los proyectos.

En ROS 2, las distintas tareas de una aplicación robótica se organizan en nodos, que son unidades de ejecución contenidas dentro de un paquete ROS. Cada nodo cumple una función específica y puede comunicarse con otros mediante diversos mecanismos: tópicos, servicios y acciones (Ver figura 6).

- **Nodos** [48]: Son procesos independientes que se encargan de ejecutar funciones particulares. Cada nodo puede interactuar con el entorno y con otros nodos mediante mensajes estructurados. Estos mensajes pueden intercambiarse de distintas formas, dependiendo del tipo de interacción que se necesite.
- **Tópicos** [52]: Es una de las formas más comunes en que los nodos se pueden comunicar. Estos actúan como un bus de mensajes en el que un nodo puede publicar datos, mientras otros pueden suscribirse a estos tópicos para recibir la información. Esta comunicación es asíncrona y unidireccional (nodo publicador a nodo suscriptor), ideal para flujos constantes de información como sensores o comandos de control.

- **Acciones** [50]: A diferencia de los tópicos, los servicios ofrecen una comunicación sincrónica y bidireccional. Consisten en una solicitud por parte de un nodo cliente y una respuesta procesada por un nodo servidor. Estos son útiles cuando se requiere realizar una solicitud puntual.
- **Servicios** [51]: Diseñadas para realizar operaciones que pueden tardar un tiempo en completarse, como navegar desde un punto A hacia un punto B. Permiten enviar una meta, recibir retroalimentación durante la ejecución y obtener el resultado al finalizar.



(a) Nodos comunicándose por servicios y tópicos. (b) Nodos comunicándose por acciones. Obtenido desde [48]. Obtenido desde [50]

Figura 6: Tipos de comunicación en ROS 2

ROS 2 tiene múltiples herramientas integradas que resultan de utilidad. En esta ocasión se mencionarán las fundamentales para la realización de este proyecto:

- **Gestión de paquetes y compilación:** El sistema utiliza *colcon* como herramienta de compilación y gestión de dependencias [53], lo que facilita la construcción de múltiples paquetes dentro de un workspace.
- **Simulación y pruebas:** Herramientas como *rviz2* [49] permiten visualizar sensores, transformaciones espaciales y estados del sistema, siendo fundamentales para la realización de pruebas.
- **Otras herramientas:** ROS 2 incorpora utilidades como *ROS 2 topic*, *ROS 2 service* y *ROS 2 node* que ayudan a explorar y depurar el sistema de forma interactiva.

Las transformaciones TF [47] en ROS2 son un sistema fundamental para manejar las relaciones espaciales entre distintos marcos de referencia del robot y su entorno. En el caso del TurtleBot3, permiten convertir coordenadas desde la cámara, el LiDAR o los sensores a un marco común, como la base del robot (*base_link*) o el mapa global (*map*). Gracias a estas transformaciones, es posible integrar de forma coherente la percepción, la localización y la navegación, ya que cada sensor entrega información en su propio marco de referencia. Sin TF, el robot no podría localizar correctamente los objetos detectados ni calcular trayectorias de manera precisa dentro del espacio de navegación. Además, este sistema permite que los nodos publiquen y consulten transformaciones entre distintos frames mediante una estructura de datos llamada *tf_buffer*, que almacena las transformaciones a lo largo del tiempo. Para recuperar estas transformaciones, los nodos utilizan un *tf_listener*, el cual consulta el buffer y entrega la transformación deseada. Estos componentes son fundamentales en aplicaciones como SLAM, navegación, manipulación robótica y detección de objetos, ya que permiten convertir coordenadas de un sistema de referencia a otro.

2.4. SLAM

SLAM (Simultaneous Localization and Mapping) [15] es una técnica mediante la cual un robot construye un mapa del entorno mientras se localiza en este (Ver figura 7). Es fundamental para que el robot navegue en entornos desconocidos sin depender de un mapa previo. Los objetivos principales de SLAM son la localización, la cual estima la posición y orientación del robot dentro del entorno, y mapeo, que es construir un mapa del entorno con ayuda de sensores.

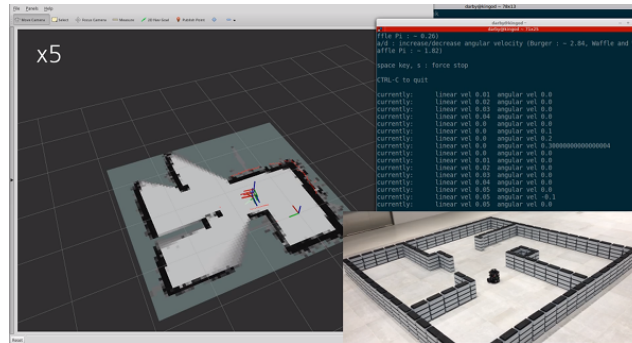


Figura 7: Mapa generado por SLAM. Obtenido desde [54]

Para la realización de este proyecto se utilizó SLAM Toolbox [31], que es un paquete para realizar SLAM en ROS 2, especialmente optimizado para LiDAR en 2D. Fue desarrollado por Steve Macenski y es ampliamente utilizado en robots móviles como el TurtleBot3.

Entre las características de SLAM Toolbox se pueden destacar las siguientes:

- Mapeo **síncrono**, donde el mapa se actualiza solo cuando el robot se detiene o en pausas específicas.
- Mapeo **asíncrono**, donde el mapa se actualiza en tiempo real. En esta memoria de título se utilizó un mapeo asíncrono, ya que se necesita la información del mapa constantemente para actualizar la localización de objetos en el mapa.
- Guardar y cargar de mapas optimizados.
- Corrección de la trayectoria del robot.

Los tópicos de SLAM Toolbox permiten obtener información específica que puede ser útil al momento de correr el nodo. Entre los tópicos se pueden destacar los siguientes:

Tópico	Descripción
/tf	Transformaciones entre marcos (ej. map $\rightarrow$ odom).
/map	Mapa generado del entorno.
/scan	Datos del LiDAR.

Cuadro 3: Tópicos SLAM Toolbox

2.5. Navegación con ROS

La navegación robótica es el proceso mediante el cual un robot es capaz de desplazarse de un punto a otro de forma autónoma, evitando obstáculos y tomando decisiones basadas en sus sensores y en un mapa.

En ROS 2 el paquete de navegación es Navigation2 (Nav2) [32] el cual permite a los robots móviles navegar en entornos conocidos (con mapas previos) o generados dinámicamente (con SLAM), planeando rutas en mapas 2D utilizando algoritmos como A* o Dijkstra [33]. Fue desarrollado por el equipo de Open Navigation y es compatible con múltiples plataformas robóticas como TurtleBot3 (Ver figura 8).

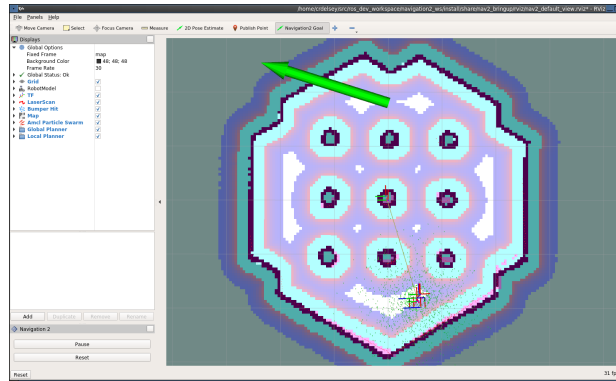


Figura 8: Turtlebot3 utilizando Nav2 visualizado RVIZ2

Entre las herramientas principales de Nav2 se pueden destacar las siguientes:

- Cargar, publicar y almacenar mapas.
- Localizar el robot en un mapa
- Planificación de recorridos.
- Controlar el robot para que siga el camino y ajusta colisiones dinámicamente.
- Seguir puntos de referencia secuenciales.

Los tópicos de Nav2, al igual que con SLAM Toolbox, permiten obtener información específica que puede ser útil al momento de correr el nodo. Entre los tópicos se pueden destacar los siguientes:

Tópico	Descripción
/map	Mapa estático del entorno.
/amcl_pose	Estimación de la posición del robot por amcl
/scan	Datos del LiDAR.
/goal_pose	Posición objetivo del robot
/tf, /tf_static	Transformaciones entre map, odom, y base_link

Cuadro 4: Tópicos Navigaton2

Finalmente, Nav2 ofrece acciones específicas para comunicarse. La más utilizada en este proyecto es “navigate_to_pose” la cual se utiliza para navegar a una coordenada específica.

2.6. Docker

Docker [6] es una plataforma que permite desarrollar, empaquetar y ejecutar aplicaciones dentro de contenedores, los cuales son entornos ligeros, portables y reproducibles. Un contenedor aísla aplicaciones del sistema operativo, lo que facilita la portabilidad del software y evita conflictos entre dependencias.

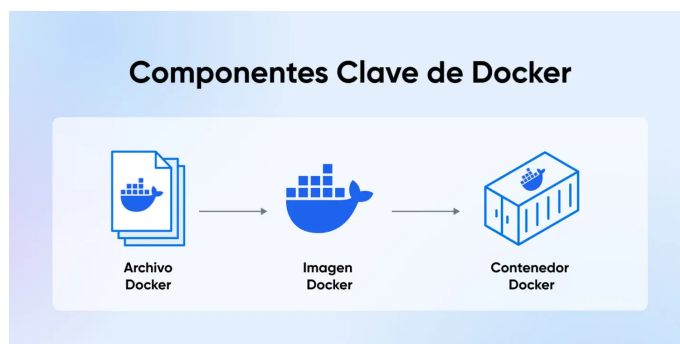


Figura 9: Componentes clave de Docker

Para poder entender qué es un contenedor y cómo se utiliza en esta memoria de título, primeramente debemos comprender qué es una imagen. La imagen de Docker es un archivo inmutable que contiene todo lo necesario para ejecutar una aplicación: el sistema operativo base, bibliotecas, dependencias, código fuente y configuraciones.

Un contenedor es una instancia en ejecución de una imagen (Ver figura 9) [2]. Cuando se lanza una imagen, se crea un contenedor que corre de manera aislada. Los contenedores pueden modificarse durante la ejecución (instalar paquetes, cambiar archivos, etc.) (Ver cuadro 5).

Concepto	Imagen	Contenedor
Estado	Estático (no cambia)	Dinámico (puede modificarse durante su ejecución)
Función	Plantilla para crear contenedores	Entorno de ejecución basado en una imagen
Persistencia	No guarda cambios por sí sola	Puede guardar cambios si se convierte en imagen
Ejecución	No puede ejecutarse directamente	Se ejecuta como proceso aislado

Cuadro 5: Diferencias entre una imagen y un contenedor en Docker

El uso de contenedores Docker en proyectos de robótica resulta especialmente útil, ya que en dispositivos con recursos limitados, como la Jetson Nano, permite instalar librerías, software y otras herramientas sin preocuparse por la compatibilidad directa con el sistema operativo del equipo. En el caso específico de la Jetson Nano, NVIDIA proporciona soporte oficial para Docker, con integración directa con sus bibliotecas como CUDA, permitiendo así ejecutar cargas de trabajo aceleradas por GPU dentro de contenedores.

2.7. Protocolo de comunicación UDP

UDP (User Datagram Protocol) [43] es un protocolo de comunicación de la capa de transporte del modelo OSI [62], que permite enviar datos entre dispositivos a través de una red.

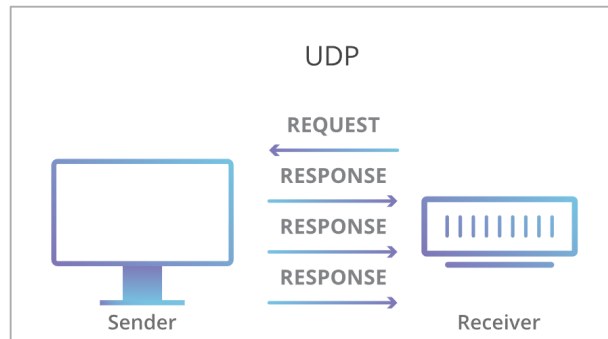


Figura 10: User Datagram Protocol. Obtenido desde [10].

Este protocolo no está orientado a la conexión, es decir, no requiere una conexión previa entre cliente y servidor (Ver figura 10) [10]. Esto es útil en contextos donde se necesita transmitir datos de manera rápida y sin esperar una respuesta, como por ejemplo, enviar información desde un contenedor de Docker a un nodo de ROS 2.

Utilizar UDP prioriza la velocidad sobre la confiabilidad: es ideal para el envío de datos pequeños y frecuentes, incluso si esto implica tolerar pérdidas ocasionales de paquetes.

2.8. Detección de objetos con YOLOv5

YOLO (You Only Look Once) [46] es un algoritmo de detección de objetos, el cual se destaca de sus competidores porque requiere “ver” la imagen una sola vez; esto hace que YOLO sea más rápido.

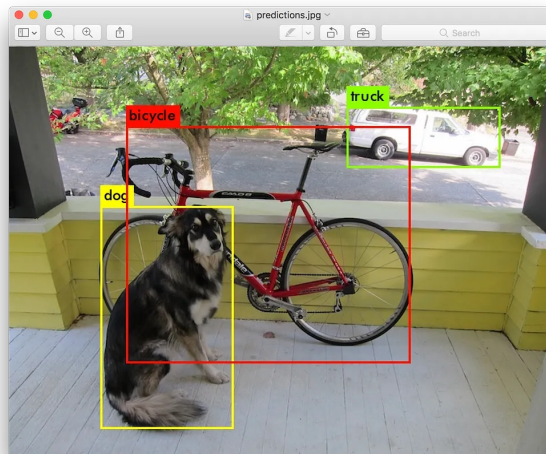


Figura 11: Imagen de ejemplo de YOLO. Obtenido desde [16].

YOLO realiza la detección dividiendo la imagen en **una cuadrícula de $S \times S$ celdas**. En cada celda, el modelo predice **N posibles bounding boxes**, las cuales son cajas delimitadoras donde se encuentra el objeto detectado (Ver figura 11), calculando la probabilidad de que cada una contenga un objeto. Como resultado, **se generan $S \times S \times N$ cajas**. Posteriormente, se descartan aquellas cuya probabilidad sea inferior a un umbral previamente definido. A las cajas restantes se les aplica la técnica de **non-maximum suppression**, que consiste en una serie de pasos destinados a eliminar detecciones erróneas, duplicadas o imprecisas [16]. Al finalizar, se tiene una imagen con las bounding boxes asociadas a los objetos detectados (Ver figura 11) [16].

YOLOv5 (You Only Look Once v5) [21] fue desarrollada y mantenida por Ultralytics, y es la evolución de las versiones anteriores de YOLO con importantes mejoras en rendimiento, velocidad y modularidad.

Para entrenar y evaluar este tipo de modelos, se utilizan conjuntos de datos estandarizados como COCO (Common Objects in Context) [28] (Ver figura 12), un conjunto de datos de detección, segmentación y subtítulo de objetos a gran escala. COCO proporciona miles de imágenes anotadas con objetos comunes en contextos realistas, las cuales sirven de ayuda para medir el rendimiento de algoritmos de visión por computador como YOLOv5.

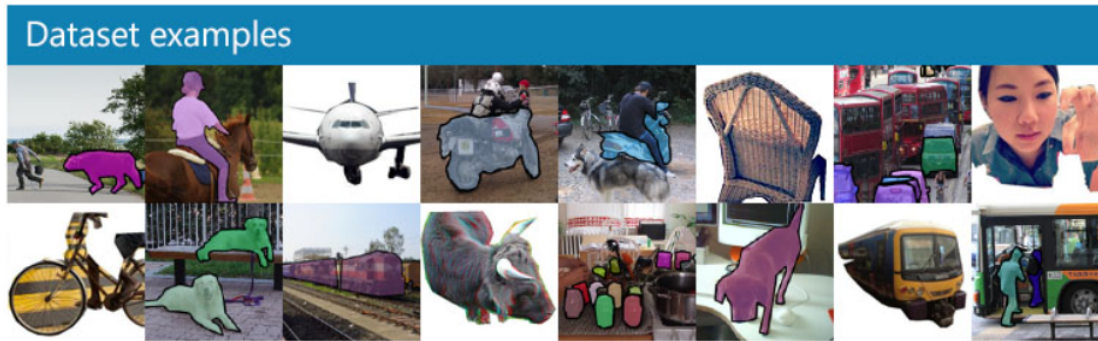


Figura 12: Ejemplos de dataset de COCO. Obtenido desde [28].

Por último, YOLOv5 destaca por su arquitectura eficiente y versátil, lo que ha permitido su amplia adaptación en aplicaciones de visión por computador. Este modelo ofrece varias versiones (Nano, Small, Medium, Large y Extra Large) que equilibran número de parámetros, precisión y velocidad. Según la documentación oficial de Ultralytics [59], **YOLOv5s**, que es el modelo utilizado en este proyecto, alcanza un mean Average Precision (mAP) de 37.4 % sobre el conjunto de validación COCO val2017 (un benchmark estándar en detección de objetos), con una velocidad de inferencia de aproximadamente 1.92 ms sobre GPU con TensorRT y sólo 24.0M de parámetros. En la misma documentación se compara con **EfficientDet-d0** el cual obtiene un mAP de 34.6 % pero con una velocidad cerca de 3.92 ms, lo que muestra que YOLOv5s ofrece mayor eficiencia. Además, otros estudios respaldan que YOLOv5 supera en precisión y rapidez a modelos como MobileNet-SSDv2 [61], consolidando su ventaja para aplicaciones de detección en tiempo real.

2.9. Visión robótica

La visión computacional, como lo relatan Sucar y Gómez en su libro “Visión Computacional” [57], emula la capacidad que tienen los seres vivos de poder observar el entorno mediante imágenes adquiridas por una cámara web.

Específicamente, los robots observan el entorno y lo interpretan, calculando distancias, detectando objetos, clasificando y actuando en función a la información visual. A esto se llama **Visión robótica**, la cual es una subárea de la visión computacional.

Para que un robot sea capaz de percibir e interpretar su entorno de manera efectiva, es necesario considerar diversos componentes fundamentales. En primer lugar, se utilizan **cámaras** de distintos tipos, las cuales constituyen los “ojos” de los sistemas de visión robótica, ya que permiten capturar la información visual que será posteriormente procesada. Entre las tecnologías más comunes se encuentran:

1. **Cámaras estéreo** [25], que utilizan dos sensores separados para obtener imágenes desde diferentes ángulos y estimar la profundidad mediante disparidad estereoscópica.
2. **Cámaras de luz estructurada** [26], que proyectan un patrón de luz sobre el entorno y calculan la geometría de los objetos a partir de las deformaciones del patrón capturado.
3. **Sistemas de triangulación láser** [60], que emiten un haz láser y miden su posición reflejada en un sensor para determinar distancias con alta precisión.

Cada una de estas tecnologías presenta ventajas y limitaciones dependiendo del entorno y la tarea: las cámaras estéreo son eficientes en exteriores con buena iluminación, la luz estructurada es más precisa en interiores y entornos controlados, mientras que la triangulación láser ofrece mediciones muy exactas, pero con menor velocidad de adquisición de datos.

La **iluminación** también desempeña un papel crucial, dado que la mayoría de las cámaras requieren condiciones lumínicas adecuadas para generar imágenes útiles en la estimación de métricas e interpretación del entorno. Asimismo, el **software de procesamiento de imágenes** resulta esencial, ya que posibilita la interpretación y el análisis de los datos, transformando las imágenes capturadas en información relevante. Finalmente, los **sistemas de control** integran la visión artificial con otros subsistemas del robot, facilitando la ejecución coordinada de tareas y aplicaciones.

En el proyecto desarrollado por Asadullah (2017) [3], se presenta una metodología para **estimar la distancia a un objeto utilizando una cámara web**. El procedimiento se basa en una etapa de calibración, en la que se emplea un objeto de ancho conocido ubicado a una distancia fija, lo que permite calcular la distancia focal de la cámara. Este parámetro resulta esencial para realizar estimaciones posteriores, ya que establece una relación entre el tamaño aparente de un objeto en la imagen (en píxeles) y su distancia real. La distancia focal se define como la distancia entre el centro óptico del objetivo y el plano focal, es decir, el sensor de imagen. Cabe destacar que cada lente posee una distancia focal específica, lo que influye directamente en la forma en que se proyectan los objetos sobre el sensor (Sony, s.f.) [56].

Para implementar esta metodología, resulta fundamental el uso de herramientas computacionales que faciliten el procesamiento de imágenes, como lo es la librería OpenCV (Open Source Computer Vision Library) [19]. Esta librería de código abierto proporciona una amplia gama de funciones para la manipulación de imágenes y videos, incluyendo la detección de objetos, el cálculo de dimensiones en píxeles y la aplicación de transformaciones geométricas.

3. Metodología

El objetivo de esta memoria de título es que el robot Turtlebot3 pueda generar un mapa, detectar objetos y localizarlos en dicho mapa, para que finalmente pueda navegar de forma autónoma hacia los objetos detectados (Ver figura 13), los cuales son solicitados por el usuario. Por ello, se busca resolver, por una parte, problemas relacionados con la integración de componentes electrónicos y, por otra, desafíos en el análisis de imágenes y la generación de rutas en un robot Turtlebot3.



Figura 13: Ejemplificación del objetivo del proyecto

Para lograr la integración de los distintos componentes del sistema, se exploraron múltiples configuraciones y métodos experimentales. A continuación, se describen las principales alternativas evaluadas, junto con sus respectivas limitaciones y decisiones tomadas:

1. **Configuración del sistema operativo:** Inicialmente se utilizó Ubuntu 18.04, versión compatible con la Jetson Nano. Sin embargo, debido a la necesidad de utilizar versiones más recientes de ROS 2, se opta por instalar Ubuntu 20.04, en su versión adaptada para la Jetson Nano.
2. **Implementación nativa con cámara RealSense y Nav2:** Instalación de todos los componentes directamente en la Jetson Nano, integrando la cámara Intel RealSense d435 y utilizando Nav2 por su compatibilidad con ROS 2. No obstante, esta opción se descartó debido a la falta de soporte completo de la cámara RealSense bajo la versión 4.6 del JetPack, lo que impedía su correcto funcionamiento.
3. **Implementación nativa con cámara web:** Se reemplazó la cámara RealSense por una cámara web convencional, manteniendo la instalación local de los demás módulos. Esta configuración también se descartó debido a problemas de compatibilidad entre las dependencias de YOLOv5 y la versión de JetPack.
4. **YOLOv5 en contenedor Docker, SLAM y navegación externos:** Se aisló el módulo de detección en un contenedor Docker, mientras que SLAM y Nav2 se ejecutan directamente sobre ROS 2 en el sistema host. Esta opción resultó funcional, permitiendo la detección de objetos y la comunicación fluida entre el contenedor y ROS 2.
5. **Localización de objetos en el mapa con Vector3Stamped():** Se implementó la localización de objetos en el mapa mediante mensajes del tipo Vector3Stamped(), el cual realiza una transformación vectorial ayudando a localizar la posición del objeto. Esta solución se descartó por su imprecisión al representar la posición real de los objetos en el entorno.

-
6. **Localización de objetos en el mapa con `PointStamped()`:** Esta técnica solo es efectiva si se conserva la primera detección, ya que, al rotar el robot, los objetos rotan con él. Debido a la complejidad que implicaría una solución más robusta y a las restricciones de tiempo disponibles para la gestión de memoria, se optó por guardar únicamente la primera detección y utilizar esta técnica para la localización de objetos en el mapa.
 7. **Generación del mapa, detección y posterior navegación:** En esta alternativa, se realizó la detección de objetos durante la fase de mapeo, se almacenaba la información en el mapa, se guardaba el mapa y finalmente se ejecutaba la navegación con Nav2. Se toma la decisión de utilizar esta herramienta ya que al momento de planear rutas en mapas 2D, este utiliza A*. Sin embargo, se identificó un inconveniente crítico: al guardar el mapa, su orientación cambiaba, lo que comprometía la coherencia del sistema, ya que al realizar la navegación a objetos se necesita que esté exactamente igual a cuando se creó ya que las coordenadas deben ser las mismas.
 8. **Mapeo y navegación simultáneos:** Se activó la navegación de forma paralela a la construcción del mapa, permitiendo una transición fluida entre ambas fases. Sin embargo, durante la navegación hacia los objetos detectados, el robot no consideraba los objetos como obstáculos, lo que afectaba el desempeño del sistema.
 9. **Mapeo y navegación simultáneos evitando objetos:** Finalmente, se integró una función adicional al código de actualización de objetos en el mapa (*map_updater*) encargado de marcar los objetos detectados como obstáculos dentro del mapa. Esta configuración resultó efectiva, ya que permite una navegación fluida y segura, con esquivas activas de objetos detectados durante el recorrido.

La metodología se divide en distintos apartados, destacando la importancia de cada uno de forma individual. En primer lugar, se aborda el **ensamblado del robot**, ya que este se entrega desarmado y se le deben incorporar componentes adicionales. A continuación, se detalla la **instalación de las dependencias**, una etapa compleja que requirió un análisis minucioso para garantizar el funcionamiento correcto e integrado del sistema. Posteriormente, se desarrolla la etapa de **detección de objetos** en tiempo real, que permite al robot identificar objetos en el ambiente y transmitir la información captada por la cámara. Esta información es utilizada en la **creación de nodos**, donde se programan tres nodos distintos, cada uno con una funcionalidad específica. Finalmente, se describe el **flujo de trabajo**, explicando cómo se integran todas las áreas implementadas para cumplir con los objetivos del proyecto.

3.1. Ensamblado del robot Turtlebot3

Los robots Turtlebot3 disponibles para las y los estudiantes que se encuentran en la universidad son los modelos **Waffle** y **Burguer**. En una primera instancia, se tomó la decisión de utilizar el modelo **Waffle**, en el cual se hizo el cambio de Raspberry Pi por una NVIDIA Jetson Nano, ya que su capacidad para trabajar con procesamiento de imágenes es mucho mayor en relación a la primera computadora. Este proceso se llevó a cabo con la ayuda del tutorial creado por Sj-Amani [55], el cual explica paso a paso como conectar el OpenCR a la Jetson Nano.

Como primera etapa para realizar esta memoria de título, se realiza un diseño 3D para sostener la cámara web (Ver figura 14). Este considera sus dimensiones, manteniéndola en la parte del frente del robot y así poder facilitar la detección de objetos mientras el robot se desplaza.

Para regular la temperatura de la Jetson Nano, se toma la decisión de agregar un ventilador (Ver figura 15), ya que al momento de correr modelos de detección de objetos, ésta se sobrecalienta y puede ser perjudicial para la vida útil de la computadora.

Luego de añadir el ventilador, se realiza un cambio del soporte del LiDAR, ya que las dimensiones del Turtlebot3 Waffle no permiten que quede completamente asegurado a la base del robot. Por lo anterior, se añade una pieza de un Turtlebot3 Burguer para sostener el LiDAR y así poder mantener la ventilación de la Jetson Nano sin problemas.

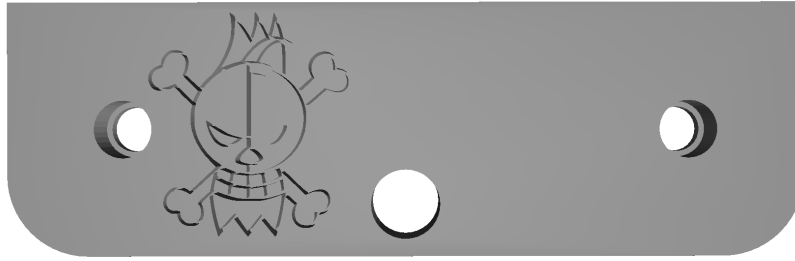


Figura 14: Diseño impresión 3D de soporte de cámara por Sebastián Retamal



Figura 15: Ventilador Jetson Nano

Con estos cambios se obtiene un robot modificado, el cual se utiliza para realizar esta memoria de título (Ver figura 16).

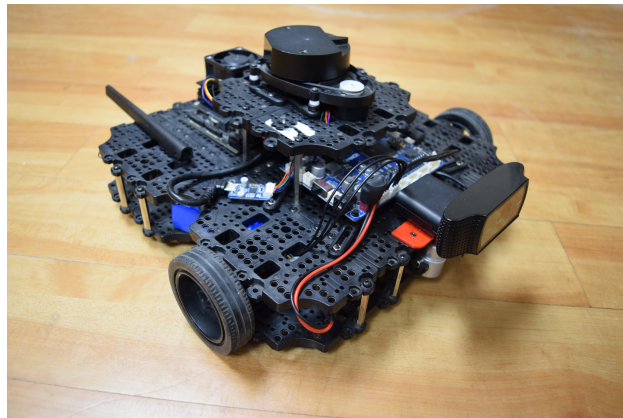


Figura 16: Imagen del robot con todos sus componentes

3.2. Instalación de dependencias

Para la instalación del sistema operativo en la Jetson Nano, se utilizó la imagen proporcionada en el repositorio de Qengineering [45]. Esta imagen está basada en Ubuntu 20.04 y ha sido específicamente adaptada para dispositivos con arquitectura ARM64, como la Jetson Nano. Una de las principales razones para elegir esta imagen es que incluye soporte para CUDA y otras herramientas necesarias para aprovechar la aceleración por hardware de la GPU integrada en la Jetson, lo cual es fundamental para aplicaciones de visión por computadora e inteligencia artificial.

Además, se instaló NoMachine [41] en la Jetson Nano, siguiendo el tutorial específico de la página web de JetsonHacks [22]. NoMachine es una herramienta de acceso remoto que permite conectarse a la interfaz gráfica del sistema operativo de la Jetson desde otro equipo, facilitando la interacción con el entorno de escritorio sin necesidad de conectar un monitor, teclado o ratón directamente al dispositivo. Esto es especialmente útil en entornos de desarrollo donde se trabaja de forma remota o se desea ahorrar espacio físico.

Luego de la instalación del sistema operativo, se realizaron algunas configuraciones adicionales para optimizar el funcionamiento de la Jetson Nano.

1. Aumentar el tamaño del disco: La imagen del sistema operativo utilizada venía con una partición de almacenamiento reducida por defecto. Para ampliar la partición principal y aprovechar todo el espacio disponible en la tarjeta microSD, se ejecutaron los siguientes comandos:

```
sudo apt update
sudo apt install gparted
sudo resize2fs /dev/mmcblk0p1
```

Este procedimiento permitió redimensionar la partición principal (/dev/mmcblk0p1) y así evitar problemas relacionados con espacio insuficiente en disco al instalar dependencias.

2. Instalación y configuración del ventilador: Esto se realiza para mejorar la gestión térmica del dispositivo, utilizando el repositorio llamado jetson-fan-ctl [44].

```
sudo ./install.sh
```

Este script permite controlar el encendido y apagado automático del ventilador de la Jetson Nano según la temperatura del sistema, evitando sobrecalentamientos y contribuyendo a la eficiencia energética del dispositivo. La instalación y configuración se realizó siguiendo las instrucciones del repositorio, lo que permitió tener un sistema de enfriamiento activo y dinámico sin necesidad de intervención manual.

3. Preparación de placa OpenCR: Se siguieron los pasos descritos en la documentación oficial de Robotis [35]. Este procedimiento permite configurar correctamente la placa controladora OpenCR para su uso con ROS 2, asegurando la comunicación entre la Jetson Nano y el TurtleBot3.

Durante el proceso, se identificó que el sistema no otorgaba permisos de acceso al puerto serie /dev/ttyACM0, el cual es necesario para interactuar con la placa. Para solucionar esto, se ejecutó el siguiente comando para agregar el usuario actual al grupo dialout, que tiene los permisos necesarios para acceder a dispositivos serie:

```
sudo usermod -a -G dialout $USER
```

Después de ejecutar este comando, fue necesario cerrar sesión y volver a iniciarla para que los cambios surtieran efecto.

Finalmente, se verificó que el usuario tuviera los permisos correctos sobre el puerto serie con:

```
ls -l /dev/ttyACMO
```

Una vez confirmados los permisos, se continuó con el procedimiento de carga del firmware y configuración del OpenCR según lo indicado en el manual oficial.

3.2.1. Instalación de ROS 2

Para la instalación del sistema de robótica ROS 2 en la Jetson Nano, se siguió el procedimiento oficial disponible en la documentación de ROS 2 Foxy: <https://docs.ros.org/en/foxy/Installation.html>. Esta versión fue seleccionada por varias razones que la hacen particularmente conveniente para esta plataforma:

- Cuenta con soporte desde junio del 2020 hasta junio del 2023, lo que garantiza actualizaciones de seguridad y estabilidad durante una cantidad de tiempo prolongada, ideal para proyectos que requieren robustez.
- Es compatible de forma nativa con arquitecturas ARM de 64 bits, como la utilizada por la Jetson Nano.
- Es una de las versiones más estables y ampliamente soportadas por la comunidad, especialmente en proyectos con TurtleBot3.

Para la instalación y configuración de Foxy se siguieron los siguientes pasos:

1. Configurar la localización del sistema (si es necesario):

```
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8
```

2. Agregar las fuentes del repositorio de ROS 2:

```
sudo apt update && sudo apt install curl gnupg2 lsb-release
sudo curl -sSL
    https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc |
    apt-key add -
sudo sh -c 'echo "deb http://packages.ros.org/ROS 2/ubuntu $(lsb_release
    -cs) main" > /etc/apt/sources.list.d/ROS 2-latest.list'
```

3. Instalar ROS 2 Foxy:

```
sudo apt update
sudo apt install ros-foxy-desktop
```

4. Configurar el entorno:

```
echo "source /opt/ros/foxy/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Para no tener que configurar el entorno cada vez que se utilice una terminal, se edita el archivo *bashrc* y así, cada vez que se abra una nueva terminal, se carguen las configuraciones iniciales:

```
nano ~/.bashrc
Export TURTLEBOT3_MODEL=waffle
Ctrl+O, luego Enter para guardar.
Ctrl+X para salir
source ~/.bashrc
```

6. Instalar herramientas de desarrollo y dependencias adicionales:

```
sudo apt install -y python3-colcon-common-extensions python3-rosdep
python3-argcomplete
```

3.2.2. Instalación de SLAM Toolbox y Navigation2

Con el objetivo de realizar mapeo y navegación autónoma utilizando SLAM (Simultaneous Localization and Mapping), se siguió el tutorial de Robotics Back-End, disponible en https://roboticsbackend.com/ROS2-nav2-generate-a-map-with-slam_toolbox/

Este procedimiento permite utilizar el paquete *slam_toolbox* con ROS 2 Foxy para generar un mapa en tiempo real usando el LiDAR del TurtleBot3, y luego emplear los paquetes de navegación (*nav2*) para planificar rutas.

Los pasos a seguir para la instalación son los siguientes:

1. Instalar los paquetes de navegación Nav2 y su entorno de ejecución:

```
sudo apt install ros-foxy-navigation2 ros-foxy-nav2-bringup
```

2. Instalar los paquetes del TurtleBot3 compatibles con ROS 2 Foxy:

```
sudo apt install ros-foxy-turtlebot3*
```

3. Instalar *slam_toolbox*, el paquete encargado de realizar SLAM:

```
sudo apt install ros-foxy-slam-toolbox
```

Con esta configuración, la Jetson Nano queda lista para ejecutar SLAM con *slam_toolbox* y realizar navegación autónoma, utilizando el modelo TurtleBot3 Waffle.

3.2.3. Configurar Docker e instalar YOLOv5

En este proyecto, por una parte, se utiliza ROS 2 Foxy, que es compatible con Python 3.8, lo cual es de gran utilidad en la creación de nodos. Por otra parte, las dependencias necesarias para la utilización de YOLOv5 (OpenCV, Pytorch y TorchVision) compatibles con JetPack 4.6, solo son compatibles con Python 3.6, lo cual genera discordancia con la versión de ROS 2 y la utilización de YOLOv5.

Para facilitar el uso de YOLOv5 en la Jetson Nano, se optó por utilizar un contenedor Docker que incluyera las dependencias complejas de instalar manualmente. Todas estas bibliotecas fueron seleccionadas en versiones compatibles con JetPack 4.6:

- OpenCV
- PyTorch
- TorchVision

El uso de Docker permite aislar el entorno de trabajo y simplificar la gestión de dependencias, reduciendo errores de instalación. Para descargar la imagen y configurar el contenedor, se siguen los pasos detallados en el Anexo 6.1.

Previo a la instalación de YOLOv5 dentro del contenedor, se decide realizar un pequeño código de prueba para corroborar que los accesos a cámara y GUI funcionen de manera correcta, el cual contiene lo siguiente:

- Se importa la librería cv2 (OpenCV) la cual es una librería de computación visual para el procesamiento de imágenes en Python.
- Se intenta abrir la cámara con `cv2.VideoCapture(0)`, donde 0 es el índice de la cámara predeterminada.
- Si la cámara no puede abrirse, se imprime un mensaje de error.
- En caso contrario, se inicia un bucle que:
 - Lee continuamente los fotogramas de la cámara.
 - Muestra en pantalla la imagen capturada en tiempo real.
 - Sale del bucle si se presiona la tecla q.
- Finalmente, se liberan los recursos y se cierra la ventana con `destroyAllWindows()`.

Este test permitió confirmar que tanto el acceso a dispositivos como la visualización gráfica estaban correctamente habilitados dentro del contenedor, condiciones esenciales para ejecutar modelos de visión por computador como YOLOv5.

Para realizar la detección de objetos, se optó por utilizar YOLOv5, una de las arquitecturas más eficientes y precisas en tareas de visión por computador en tiempo real. La instalación se llevó a cabo dentro del contenedor Docker previamente configurado para asegurar compatibilidad con JetPack 4.6, PyTorch y OpenCV. A continuación, se describen los pasos seguidos:

1. Crear carpeta para la instalación:

```
mkdir yolov5_memoria
cd yolov5_memoria
```

2. Clonar repositorio oficial de YOLOv5 y dirigirse a la rama de la versión 6.0:

```
git clone https://github.com/ultralytics/yolov5.git
cd yolov5
git checkout v6.0
```

La versión 6.0 se elige por su compatibilidad con JetPack 4.6.

3. Instalar dependencias listadas en requirements.txt

```
pip3 install -r requirements.txt
```

Comentando previamente las líneas de `torch`, `torchvision` y `opencv-python` ya que estas vienen previamente instaladas en la imagen de Docker.

4. Instalación de pycocotools, librería para trabajar con COCODataset

```
pip3 install pycocotools
```

Librerías importantes instaladas:

- **PyTorch y TorchVision:** Framework principal para deep learning en *YOLOv5*. Ya estaban preinstaladas en el contenedor con versiones específicas compatibles con *Jetson Nano* y *CUDA de JetPack 4.6*, por lo que se omitió su reinstalación para evitar conflictos.
- **OpenCV:** Biblioteca esencial para la manipulación de imágenes y acceso a la cámara. También estaba previamente instalada, y se mantiene su versión optimizada para la Jetson Nano.
- **pycocotools:** Es una herramienta clave para trabajar con el formato de anotaciones del *dataset COCO*, ampliamente usado en detección de objetos. Es requerida por *YOLOv5* para realizar evaluaciones, métricas y análisis de datos.
- Otras bibliotecas como *numpy*, *matplotlib*, *pandas* y *tqdm*, que facilitan el procesamiento de datos, visualización de resultados y monitoreo del progreso durante la inferencia o entrenamiento.

3.3. Detección de objetos en tiempo real

Como se mencionó con anterioridad, en este proyecto se utiliza YOLOv5 como modelo de detección de objetos en tiempo real, debido a su equilibrio entre precisión, velocidad y facilidad de implementación en sistemas como la Jetson Nano. Este algoritmo cuenta con múltiples variantes que permiten ajustarse a las limitaciones de hardware.

Se opta por utilizar un modelo preentrenado en el conjunto de datos COCO, *YOLOv5s (small)*, con el fin de evitar los altos costos computacionales y temporales asociados al entrenamiento desde cero, lo que permite mantener un mejor rendimiento en computadoras con GPU limitada como lo es la Jetson Nano.

YOLOv5 es compatible con PyTorch, uno de los frameworks incluidos y optimizados previamente en el contenedor Docker del sistema. Esta compatibilidad evita problemas derivados de versiones específicas de CUDA y reduce la complejidad de configuración en el entorno de la Jetson.

3.3.1. Transmisión de datos a través de protocolo UDP

La transmisión de datos resultantes de la detección se realiza mediante protocolo UDP, en lugar de emplear la comunicación estándar mediante nodos ROS 2. Esta decisión obedece a varios factores técnicos:

1. Dentro del contenedor Docker en ejecución se utiliza Python 3.6, que no es compatible con ROS 2 Foxy, el cual requiere como mínimo Python 3.8. Esto impide el uso directo de `roslaunch` y otros componentes de ROS 2 desde el entorno de detección.
2. En el caso de utilizar ROS 2 dentro del contenedor, no es posible realizar comunicación ROS 2 entre diferentes versiones del framework (por ejemplo, entre Dashing y Foxy), ya que no existe compatibilidad binaria entre ellas.

3. La comunicación vía UDP permite una implementación más liviana y directa, sin dependencias adicionales, siendo también más eficiente en términos de latencia en redes locales cuando se trata de paquetes pequeños y frecuentes.

Por estas razones, se ha implementado un emisor UDP dentro del contenedor Docker, el cual transmite las detecciones codificadas en formato JSON a través de la red.

3.3.2. Implementación de la detección de objetos

Este programa corresponde a un código en Python que detecta objetos con YOLOv5, estima la distancia aproximada a estos objetos y envía esa información a través de la red usando UDP.

En una primera instancia, se carga un modelo previamente entrenado (yolov5s.pt) que puede detectar múltiples objetos en imágenes. El modelo se ejecuta en GPU si está disponible, lo que mejora el rendimiento. Luego, se abre la cámara web utilizando la librería OpenCV y se configura para capturar video. Si la cámara no se abre correctamente, el programa termina. Después, se definen tamaños reales aproximados (en centímetros) de algunos objetos específicos del dataset de COCO. Estos datos se usan para estimar a qué distancia se encuentra un objeto detectado, comparando su tamaño real con su tamaño aparente en la imagen y así poder calcular la distancia focal para poder hacer estas estimaciones con mayor precisión.

Luego se programa el bucle principal, el cual captura el fotograma de la cámara para poder procesar la imagen y que el modelo pueda analizarla. Este, al detectar objetos, entrega las posiciones de los píxeles de las bounding boxes, las etiquetas y el nivel de confianza. Cuando un objeto está bien detectado, se calcula la distancia. Finalmente, se dibujan las bounding boxes, con las etiquetas, la confianza y la distancia en metros, para así ser enviados por UDP en formato JSON (Ver figura 17).



Figura 17: Detección de objetos con YOLOv5 y COCO Dataset

Además, para facilitar el uso del contenedor y cuidar el respaldo de los datos, se desarrolla un script para guardar un respaldo de las versiones del contenedor (Anexo 6.4).

3.4. Creación de nodos

ROS 2 es fundamental en muchas aplicaciones de la robótica, ya que permite modularizar las funcionalidades según se estime conveniente. Además, los nodos pueden comunicarse entre sí a través de mecanismos, lo cual es útil para enviar y recibir información.

En este proyecto, se desarrollan tres nodos: `listener_yolov5.py`, que es un nodo que recibe la información que proviene del Docker; `map_updater.py` nodo que actualiza la información de los objetos detectados asignando las coordenadas al mapa; y `navigator.py`, nodo que genera una acción para que el robot navegue hacia el objeto deseado.

3.4.1. Nodo que recibe información

Este programa es un nodo de ROS 2 (Robot Operating System) que escucha detecciones de objetos enviadas por una red local mediante UDP, y luego las publica dentro del sistema ROS 2, para que otros nodos puedan usarlas.

En resumen, este código actúa como un puente entre la detección por red y el ecosistema ROS 2.

3.4.2. Nodo que localiza objetos en el mapa

Este programa corresponde a un nodo de ROS 2 llamado `map_updater`, cuyo objetivo es actualizar un mapa con la posición de objetos detectados por YOLOv5. El código recibe las detecciones, las ubica dentro del mapa y visualiza en RViz2. Además, guarda esta información para su uso posterior.

El nodo se suscribe a tres fuentes de información. Primero, al tópico `/map`, que entrega el mapa del entorno generado por SLAM Toolbox. Este mapa se guarda internamente para poder agregarle los objetos. Segundo, se suscribe al tópico “detecciones”, donde llegan las detecciones de objetos provenientes de YOLOv5. Cada mensaje contiene una lista de objetos detectados, con su clase, la posición del bounding box en la imagen y una estimación de distancia. Tercero, el nodo publica en el tópico `/visualization_marker`, el cual permite visualizar los objetos en RViz2 como etiquetas con el nombre del objeto. Y cuarto, el nodo que publica el tópico `/virtual_obstacles`, que permite crear obstáculos virtuales para que el robot sepa que en los objetos que reconozca existe un obstáculo.

Para cada objeto detectado proveniente del JSON de objetos que llegan desde el nodo que recibe la información desde el Docker, se obtiene su nombre, esto se hace a partir del identificador de clase (usando la lista de clases del dataset COCO). Si el objeto ya fue registrado antes, lo ignora para evitar duplicados. Luego, se calcula la posición del centro del objeto en la imagen y, usando una fórmula basada en trigonometría y el campo de visión de la cámara (FOV), estima su posición relativa respecto al robot. Esta posición se transforma a coordenadas globales del mapa utilizando las transformaciones TF disponibles en ROS 2, que permiten saber cuál es la posición del robot en el entorno y aplicar ese mismo desplazamiento al objeto detectado. Si la transformación es exitosa, se publica un marcador visual en el mapa con el nombre del objeto, y se guarda su ubicación en una base de datos interna, la cual se guarda finalmente en un archivo JSON llamado `objects_db.json`, dentro de una carpeta “maps” ubicada junto al script.

3.4.3. Nodo de navegación

Este programa es un nodo de ROS 2 llamado `object_navigator`, cuyo propósito es hacer que el robot pueda navegar de manera autónoma hacia la ubicación de un objeto detectado previamente. Para lograrlo, utiliza un archivo JSON que contiene las coordenadas de los objetos en el mapa, y se conecta con el servidor de acciones de Nav2 para enviarle objetivos de navegación.

Al iniciarse, el nodo carga el archivo `objects_db.json`, que contiene un diccionario con los nombres de los objetos detectados como claves y sus coordenadas en el mapa como valores.

La función principal del nodo es `navigate_to_object`, que recibe como entrada el nombre del objeto. Si el objeto existe, se obtienen sus coordenadas (x, y) en el mapa y se construye un mensaje de tipo **NavigateToPose**, que es la acción que utiliza Nav2 para definir una meta de navegación. Utilizar acciones de ROS 2, es la mejor opción para este caso, ya que una acción tiene una “meta” (las coordenadas), un “feedback” (El robot está yendo a la coordenada) y un “resultado” (El robot llega o no a la meta). En este caso, se usa una orientación fija con un ángulo yaw de 0 radianes, es decir, el robot mirará hacia el este una vez que llegue al objeto. Esta orientación se convierte en un cuaternión mediante una función auxiliar (`quaternion_from_euler`) para ser compatible con ROS.

Antes de enviar la meta, el nodo espera a que el servidor de acciones de Nav2 esté listo. Una vez conectado, se envía la meta de forma asíncrona. Si el objetivo es aceptado, se espera también de manera asíncrona a que se complete la navegación. Cuando finaliza, se muestra un mensaje indicando si la meta fue alcanzada correctamente o si hubo algún error.

El programa incluye un menú interactivo en consola (Figura 18), en el que se listan los objetos disponibles y se le pide al usuario que escriba el nombre del objeto al que desea que el robot se dirija. Si el usuario escribe “salir” o simplemente presiona Ctrl+C, el nodo se detiene. Esto permite probar la navegación en diferentes objetos.

```
jetson@nano:~/object_detection_route_planning$ ros2 run yolov5_slam_nav navigato
r
1752089678.444567 [0] navigator: using network interface wlan0 (udp/172.20.10.2
) selected arbitrarily from: wlan0, docker0
[INFO] [1752089679.510364245] [object_navigator]: Objetos disponibles en: ['bott
le', 'cup']

Objetos detectados disponibles para navegación:
- salir
- bottle
- cup

Ingrese el nombre del objeto al que desea navegar:bottle
```

Figura 18: Menú interactivo del nodo de navegación

3.5. Flujo de trabajo

Para lograr la creación del mapa, la detección de objetos, su ubicación dentro del entorno y la posterior navegación hacia ellos, es fundamental seguir una secuencia específica de procesos. En esta sección se analiza el flujo de operaciones y procedimientos (Figura 19) que permite la integración de cada una de estas funcionalidades.

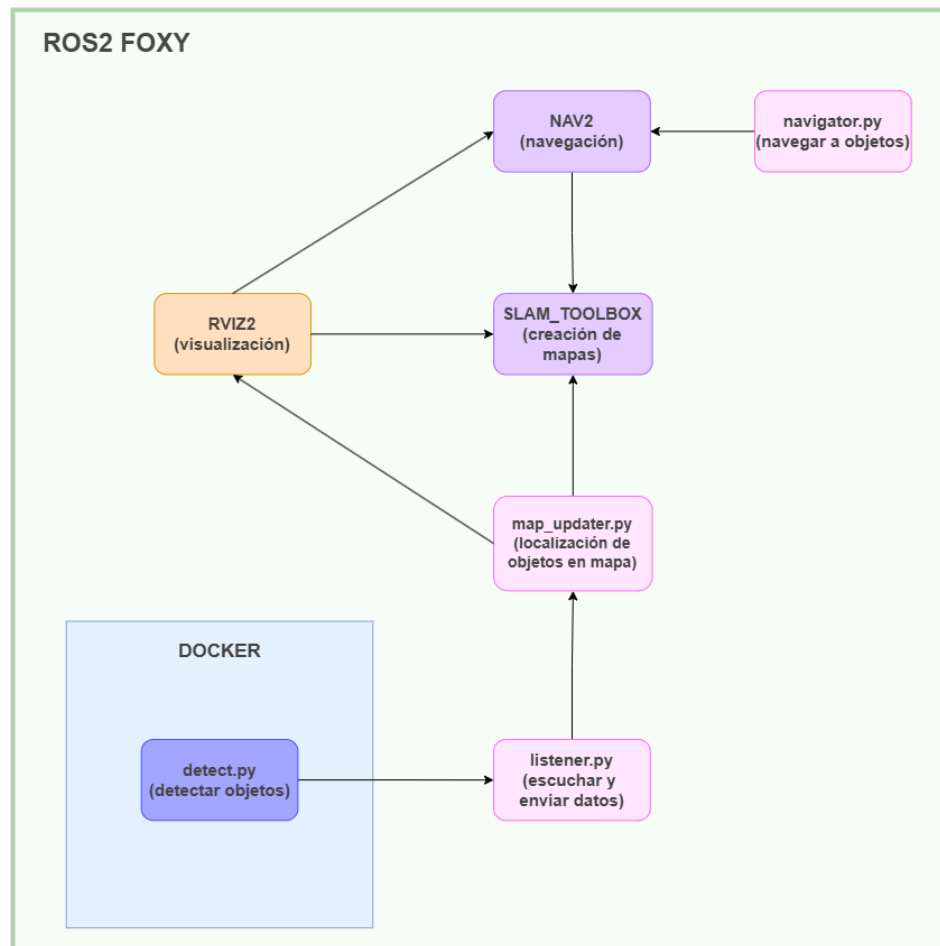


Figura 19: Diagrama de flujo general del sistema de navegación semántica desarrollado

En resumen, ROS 2 se emplea como middleware que permite la coordinación eficiente entre los distintos módulos del robot. Los nodos de ROS 2 facilitan la integración del sistema de mapeo basado en SLAM, del módulo de detección de objetos con YOLOv5 y del framework de navegación Nav2. SLAM genera y actualiza el mapa del entorno en tiempo real, mientras que YOLOv5 identifica estos objetos y los envía a través de UDP al nodo que recibe esa información, este la envía al nodo que localiza objetos dentro del mapa. A su vez, Nav2 utiliza esta información combinada para planificar trayectorias seguras y ejecutar la navegación autónoma del Turtlebot3 hacia los objetos detectados, evitando obstáculos. Esta arquitectura refleja un enfoque de integración poco explorado en la literatura, que combina percepción, localización y control de manera coordinada, y destaca la flexibilidad de ROS 2 como plataforma modular para sistemas robóticos complejos en entornos reales.

4. Experimentos y resultados

En esta sección se presentan los experimentos realizados para evaluar el sistema propuesto, incluyendo las etapas de detección, localización y navegación hacia objetos. Los experimentos se agrupan por fases y se analizan de manera individual, incluyendo el análisis cuantitativo y cualitativo por cada prueba.

Para el análisis cuantitativo, se considera lo siguiente:

- **Detección de objetos:** Para esto se busca evaluar si el robot detecta los objetos relevantes. Para calcular esto se utiliza la **Precisión** [23], la cual es una proporción de predicciones positivas entre las que realmente lo son. Esto se utiliza ya que existen ocasiones en donde el robot puede detectar objetos que realmente no están ahí y se quiere considerar las detecciones de objetos que realmente lo son.

$$\text{Precisión} = \frac{TP}{TP + FP}$$

- TP : Verdaderos positivos
- FP : Falsos positivos

Como umbrales de referencia se adoptaron criterios heurísticos usados en la literatura y guías prácticas: precisión ≥ 0.80 como indicador de desempeño operativo en robots móviles. Además, se compara el resultado experimental propio con resultados reportados en la literatura para entornos similares, como en *YOLOv5s pre-entrenado en Duckietown* mostró 88 % de precisión [40].

- **Éxito:** Precisión $\geq 80\%$
 - **Aceptable:** $70\% \leq \text{Precisión} < 80\%$
 - **Fallo:** Precisión $< 70\%$
- **Localización de objetos:** En este caso se busca que el objeto sea localizado de la forma correcta en el mapa. Para calcular el error de localización se utiliza el **Error Euclidiano medio** [1], el cual se utiliza para comparar la posición real del objeto con la posición estimada por el robot.

$$\text{Error Euclidiano Medio} = \frac{1}{N} \sum_{i=1}^N \sqrt{(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2}$$

- N : Número total de muestras
- (x_i, y_i) : Posición real del objeto i
- $(\hat{x}_i, \hat{y}_i)$: Posición estimada del objeto i

En base a los experimentos, se entiende que el éxito en la localización será con un error menor o igual a 10 centímetros, se considerará aceptable un error mayor a 10 centímetros, pero menor a 25 centímetros, y se entiende como fallo cuando la localización del objeto en el mapa es mayor a 25 centímetros.

- **Éxito:** Error euclidiano medio ≤ 10 cm
- **Aceptable:** $10 \text{ cm} < \text{Error euclidiano medio} \leq 25$ cm
- **Fallo:** Error euclidiano medio > 25 cm

- **Navegación:** Se requiere que el robot navegue hacia el objeto. Pero se consideran dos casos: que el robot navegue hacia el objeto y que el robot navegue hacia el objeto sin colisionar con otro objeto. Con respecto a los umbrales de éxito y fallo, en la literatura no existe explícitamente estos tipos de tasas con robots móviles y Nav2, así que se realizó una comparación con otros algoritmos y en base a eso se designaron los casos de éxito y fallo para cada tasa de navegación. En estudios en robots móviles (CrowdSteer, VLM-Social-Nav, PPO en Turtlebot, ISN framework), la tasa de navegación exitosa varía entre 65–100 %, mientras que la tasa de navegación exitosa sin colisiones suele encontrarse entre 65–85 % [27].

En el primer caso, se busca calcular la **Tasa de navegación exitosa** [24], la cual entrega el porcentaje de veces en la cual el robot llegó a su objetivo.

$$\text{Tasa de navegación exitosa} = \frac{\text{objetivos alcanzados}}{\text{total de intentos}} \times 100 \%$$

- **Objetivos alcanzados:** Cantidad de metas a las que el robot llegó con éxito
- **Total de intentos:** Número total de veces que se intentó llegar a un objetivo

Umbrales definidos según literatura:

- **Éxito:** Tasa de navegación exitosa $\geq 90 \%$
- **Acceptable:** $65 \% \leq \text{Tasa de navegación exitosa} < 90 \%$
- **Fallo:** Tasa de navegación exitosa $\leq 65 \%$

Para el segundo caso se calcula la **Tasa de navegación exitosa sin colisiones** [9], que entrega el porcentaje de veces en que el robot navega y llega a su objetivo sin chocar con otro objeto.

$$\text{Tasa de éxito sin colisiones} = \frac{\text{número de objetivos alcanzados sin colisiones}}{\text{número total de intentos de navegación}} \times 100 \%$$

- **Número de objetivos alcanzados sin colisiones:** Casos en que el robot llegó al objetivo sin colisionar con un objeto
- **Número total de intentos de navegación:** Total de trayectorias intentadas hacia los objetivos

Umbrales definidos en base a literatura:

- **Éxito:** Tasa de navegación exitosa sin colisiones $\geq 85 \%$
- **Acceptable:** $65 \% \leq \text{Tasa de navegación exitosa sin colisiones} < 85 \%$
- **Fallo:** Tasa de navegación exitosa sin colisiones $\leq 65 \%$

Como complemento, se creó un video con cada uno de los experimentos descritos en esta sección, el cual se puede visualizar mediante este enlace: https://youtu.be/-jdjhPkm_uY.

4.1. Mapa 1

4.1.1. Prueba 1: Un objeto

Descripción del experimento En este experimento, se evaluó un objeto (cup). Esto con el objetivo de evaluar la detección, localización y posterior navegación. Es el experimento con menor complejidad, ya que no existen otros objetos como obstáculos que el robot debe considerar para su navegación.

Resultados

- **Detección:** detecta el único objeto adecuadamente, no existen falsos positivos.
- **Localización:** localiza el objeto en el mapa de forma correcta, tiene un cierto desfase en la localización.
- **Navegación:** Se considerará el análisis de navegación por etapas, ya que se intenta navegar más de una vez.
 1. Intento 1: Se inicia nodo de navegación, pero el robot no comienza a navegar. Se observan errores en Nav2 “send_goal failed” y en SLAM una advertencia que explica que se recibe de forma tardía el frame “base_scan” el cual entrega la información proveniente del LiDAR. Por esto, se opta por reiniciar Nav2.
 2. Intento 2: Al reiniciar Nav2 se puede realizar la correcta navegación hacia el objeto sin inconvenientes adicionales.

Análisis cuantitativo

- **Detección de objetos**
 - Objetos reales: 1 (cup)
 - Objetos detectados: 1 (cup)
 - Precisión: **100 %**
 - Distancia promedio de detección: **40 cm**
- **Localización**
 - cup:
 - Localización real: (0.48, 0.01)
 - Localización mapa: (0.48, 0.11)
 - Error euclidiano medio: **10 cm**
- **Navegación**
 - Navegación exitosa:
 - Objetos alcanzados: 1
 - Total de intentos: 2
 - Navegación exitosa sin colisiones:
 - Objetos alcanzados sin colisiones: 1
 - Total de intentos: 2
 - Tasa de navegación exitosa: **50 %**
 - Tasa de éxito sin colisiones: **50 %**

- Tiempo promedio de navegación: **18 segundos**

Visualización del experimento



Figura 23: Comparación entre el entorno real y el mapa generado con el objeto detectado.

4.1.2. Prueba 2: Dos objetos

Descripción del experimento En este experimento, se evaluaron dos objetos (cup y bottle). Esto con el objetivo de evaluar la detección, localización y posterior navegación. Además, se busca evaluar el comportamiento del robot al tener más de un objeto en su entorno.

Resultados

- **Detección:** El robot se tarda en detectar “bottle”. Esto pasa ya que se utiliza un modelo pequeño y la botella no es de un color característico. Se detectan los objetos de forma correcta.
- **Localización:** localiza el objeto en el mapa de forma correcta, tiene un cierto desfase en la localización.
- **Navegación:** Se considerará el análisis de navegación por etapas, ya que se intenta navegar más de una vez.
 1. Intento 1, navegación hacia “cup”: Se lanza Nav2 después de realizar el mapa y la localización, y antes de navegar. Al momento de iniciar la navegación llega de forma exitosa al objeto, pero al rotar choca con este.
 2. Intento 2, navegación a “bottle”: No navega. Al revisar la consola no existen errores. Este comportamiento es verosímil debido a la latencia que existe entre sensores por la sobrecarga del sistema.
 3. Intento 3, navegación a “bottle”: Luego de esperar unos segundos, se intenta realizar la navegación nuevamente, la cual resulta exitosa. Además, no choca con la botella al rotar, ya que esta la puede detectar el LiDAR y lo considera como un obstáculo.

Análisis cuantitativo

■ Detección de objetos

- Objetos reales: 2 (cup, bottle)
- Objetos detectados: 2 (cup, bottle)
- Precisión: **100 %**
- Distancia promedio de detección: **25 cm**

■ Localización

- cup:
 - Localización real: (0.47, 0.4)
 - Localización mapa: (0.47, 0.5)
- bottle:
 - Localización real: (1.64, 0.03)
 - Localización mapa: (1.64, 0.03)
- Error euclidiano medio: **5 cm**

■ Navegación

- Navegación exitosa:
 - Objetos alcanzados: 2
 - Total de intentos: 3
- Navegación exitosa sin colisiones:
 - Objetos alcanzados sin colisiones: 1
 - Total de intentos: 3
- Tasa de navegación exitosa: **66,6 %**
- Tasa de éxito sin colisiones: **33,3 %**
- Tiempo promedio de navegación: **11,5 segundos**

Visualización del experimento

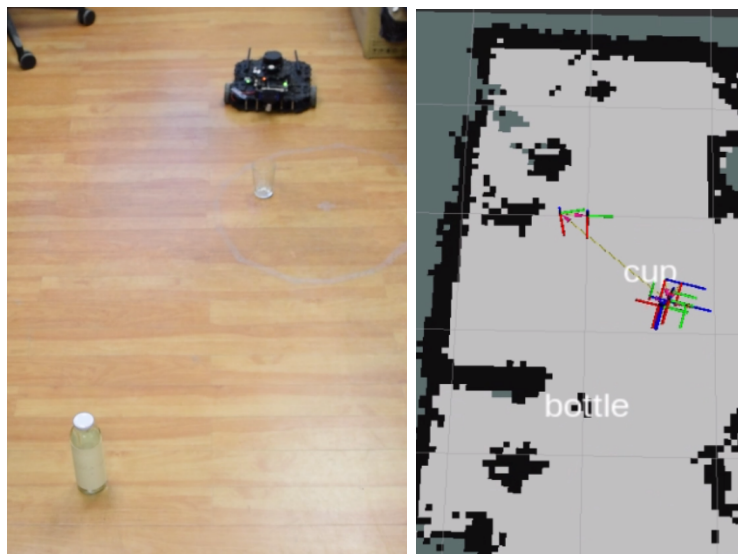


Figura 24: Comparación entre el entorno real y el mapa generado con los objetos detectados.

4.1.3. Prueba 3: Dos objetos

Descripción del experimento Se evaluaron dos objetos (cup y bottle), replicando el experimento anterior. Esto con el fin de evaluar la reproducibilidad de los resultados en condiciones (que deberían ser) idénticas.

Resultados

- **Detección:** buena detección, no hay dificultades ni tampoco falsos positivos.
- **Localización:** hay un pequeño desfase en la localización de “bottle”. El comportamiento es verosímilmente a la forma en que se calcula la distancia.
- **Navegación:** Se considerará el análisis de navegación por etapas, ya que se intenta navegar más de una vez.
 1. Intento 1, navegación a “bottle”: Al momento de ejecutar el código de navegación, el robot no comienza a navegar. Al igual que en experimentos anteriores se observan errores en Nav2 “send_goal failed” y en SLAM una advertencia que explica que se recibe de forma tardía el frame “base_scan” el cual entrega la información proveniente del LiDAR. Por esto, se opta por reiniciar Nav2.
 2. Intento 2, navegación a “bottle”: El robot llega al objeto, pero como existe un desfase en la localización de este objeto, navega hasta unos centímetros antes de llegar a la meta real. En el momento que se termina de navegar, Nav2 entrega un error que dice lo siguiente: “Failed to make progress”. Eso ocurre también cuando se utiliza 2D goal pose de Rviz2.
 3. Intento 3, navegación a “cup”: llega al objeto de forma correcta a un costado del objeto.
 4. Intento 4, navegación a “bottle”: El robot no comienza a navegar. Se observan errores en Nav2 “send_goal failed” y en SLAM una advertencia que explica que se recibe de forma tardía el frame “base_scan” el cual entrega la información proveniente del LiDAR. Por esto, se opta por reiniciar Nav2.
 5. Intento 5, navegación a “bottle”: Llega al objeto de forma correcta.

Análisis cuantitativo

- **Detección de objetos**
 - Objetos reales: 2 (cup, bottle)
 - Objetos detectados: 2 (cup, bottle)
 - Precisión: **100 %**
 - Distancia promedio de detección: **35 cm**
- **Localización**
 - cup:
 - Localización real: (0.4, 0.28)
 - Localización mapa: (0.45, 0.28)
 - bottle:
 - Localización real: (1.57, 0.62)
 - Localización mapa: (1.42, 0.32)
 - Error euclidiano medio: **20 cm**

■ Navegación

- Navegación exitosa:
 - Objetos alcanzados: 3
 - Total de intentos: 5
- Navegación exitosa sin colisiones:
 - Objetos alcanzados sin colisiones: 3
 - Total de intentos: 5
- Tasa de navegación exitosa: **60 %**
- Tasa de éxito sin colisiones: **60 %**
- Tiempo promedio de navegación: **15,6 segundos**

Visualización del experimento

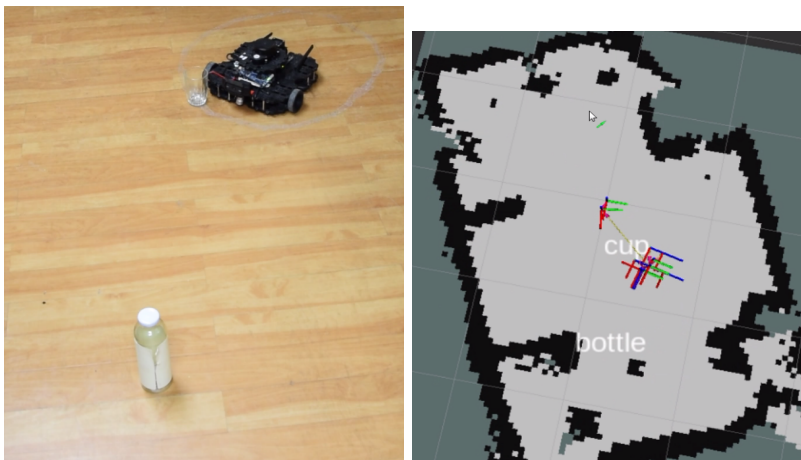


Figura 25: Comparación entre el entorno real y el mapa generado con los objetos detectados.

4.1.4. Prueba 4: Tres objetos

Descripción del experimento En este experimento, se evaluaron tres objetos (cup, bottle y orange). Esto con el objetivo de evaluar la detección, localización y posterior navegación, aumentando la complejidad al añadir un objeto más, analizando cómo el robot se comporta al aumentar la cantidad de objetos en el mapa.

Resultados

- **Detección:** buena detección, no hay dificultades ni tampoco falsos positivos.
- **Localización:** existen errores grandes en la localización por sobre todo “bottle”.
- **Navegación:** Se considerará el análisis de navegación por etapas, ya que se intenta navegar más de una vez.
 1. Intento 1, navegación a “orange”: Al iniciar la navegación, no esquiva a “cup”. Llega al objetivo pero a 25 cm de distancia aproximadamente, por error de localización.
 2. Intento 2, navegación a “bottle”: Al igual que el caso anterior, llega al objetivo pero a 25 cm de distancia aproximadamente, por error de localización.
 3. Intento 3, navegación a “cup”: Se repiten los casos anteriores, sin ningún error, pero la llegada es con desfase.

Análisis cuantitativo

■ Detección de objetos

- Objetos reales: 3 (cup, bottle, orange)
- Objetos detectados: 3 (cup, bottle, orange)
- Precisión: **100 %**
- Distancia promedio de detección: **41,6 cm**

■ Localización

- cup:
 - Localización real: (0.4, 0.02)
 - Localización mapa: (0.48, 0.17)
- bottle:
 - Localización real: (0.99, 0.97)
 - Localización mapa: (0.47, 0.93)
- orange:
 - Localización real: (0.007, 0.72)
 - Localización mapa: (0.007, 0.32)
- Error euclidiano medio: **36 cm**

■ Navegación

- Navegación exitosa:
 - Objetos alcanzados: 3
 - Total de intentos: 3
- Navegación exitosa sin colisiones:
 - Objetos alcanzados sin colisiones: 2
 - Total de intentos: 3
- Tasa de navegación exitosa: **100 %**
- Tasa de éxito sin colisiones: **66.6 %**
- Tiempo promedio de navegación: **14,6 segundos**

Visualización del experimento



Figura 26: Comparación entre el entorno real y el mapa generado con los objetos detectados.

4.1.5. Prueba 5: Tres objetos

Descripción del experimento En este experimento, se evaluaron tres objetos para detectar y localizar. Además, en este experimento se añade una mejora para esquivar objetos que no son detectados por el LiDAR, que consta en añadir puntos virtuales para que el robot los considere como obstáculos aunque no pueda detectarlo con los sensores.

Resultados

- **Detección:** se realizan dos falsos positivos.
- **Localización:** Existen errores al localizar, pero no tan grandes como en el experimento anterior.
- **Navegación:** Se considerará el análisis de navegación por etapas, ya que se intenta navegar más de una vez.
 1. Intento 1, navegación a “orange” y “cup”: Hay un error al iniciar la navegación de “Aborting handle”, se le atribuye al límite de los componentes de hardware.
 2. Intento 2, navegación a “cup”: Llega al objetivo pero al momento de rotar colisiona con el objeto.
 3. Intento 3, navegación a “orange”: No navega. Mismo error que en intento uno. Se reinicia Nav2.
 4. Intento 4, navegación a “orange”: Llega al objetivo pero con un desfase mínimo por el error de localización. Al final de la navegación aparece error “Aborting handle”.
 5. Intento 5, navegación a “bottle”: Al intentar navegar aparece error de “Aborting handle”. Se le da unos minutos al sistema y se intenta nuevamente. Navega correctamente al objetivo.

Análisis cuantitativo

- **1. Detección de objetos**
 - Objetos reales: 3 (cup, bottle, orange)
 - Objetos detectados: 5 (cup, bottle, orange, mouse, keyboard)
 - Precisión: **60 %**
 - Distancia promedio de detección: **45 cm**
- **2. Localización**
 - cup:
 - Localización real: (1.27, 0.32)
 - Localización mapa: (1.18, 0.39)
 - bottle:
 - Localización real: (1.3 , 1)
 - Localización mapa: (1.44, 0.81)
 - orange:
 - Localización real: (0.12, 0.8)
 - Localización mapa: (0.12, 0.85)
 - Error euclidiano medio: **18 cm**

■ 3. Navegación

- Navegación exitosa:
 - Objetos alcanzados: 3
 - Total de intentos: 5
- Navegación exitosa sin colisiones:
 - Objetos alcanzados sin colisiones: 2
 - Total de intentos: 5
- Tasa de navegación exitosa: **60 %**
- Tasa de éxito sin colisiones: **40 %**
- Tiempo promedio de navegación: **11,3 segundos**

Visualización del experimento

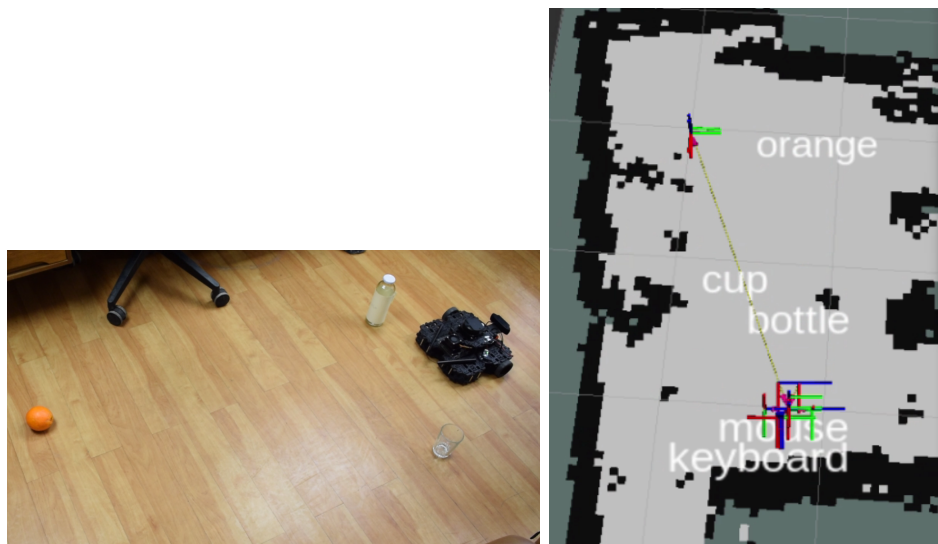


Figura 27: Comparación entre el entorno real y el mapa generado con los objetos detectados.

4.1.6. Resumen general de análisis cuantitativo (Mapa 1)

Prueba	Detección (%)	Distancia detección promedio	Error de localización (cm)	Tiempo promedio de navegación	Navegación exitosa (%)	Navegación exitosa SC(%)
1	100	40	10	18	50	50
2	100	25	5	11,5	66,6	33,3
3	100	35	20	15,6	60	60
4	100	41,6	36	14,6	100	66,6
5	60	45	18	11,3	60	40
Promedio	92	37,32	17,8	14,2	67,32	50

Cuadro 6: Resumen de análisis cuantitativo por prueba mapa 1.

4.2. Mapa 2

4.2.1. Prueba 1: Dos objetos

Descripción del experimento En este experimento, se evaluaron dos objetos para detectar y localizar. Para realizar este experimento, se decide despejar el espacio en el que el robot se va a desplazar, para evitar inconvenientes externos que provoquen errores.

Resultados

- **Detección:** Se genera un falso positivo.
- **Localización:** En un inicio se puede observar una buena localización, a simple vista no se logra detectar errores. Cuando pasa el tiempo y el mapa se va actualizando, el error en la localización aumenta pero no es mayor.
- **Navegación:** Se considerará el análisis de navegación por etapas, ya que se intenta navegar más de una vez.
 1. Intento 1, navegación a “bottle”: Navega exitosamente al objeto.
 2. Intento 2, navegación a “cup”: Al intentar navegar, el robot no se mueve, pero en consola se puede observar que la meta fue alcanzada con éxito. Este comportamiento se atribuye al hecho de que el robot se encuentra cerca de un obstáculo lo cual suele confundirlo. Se toma la decisión de mover al robot con teleop de forma manual, para así poder separar el área de seguridad del robot del obstáculo. También se reinicia Nav2.
 3. Intento 3, navegación a “cup”: Navega correctamente al objeto, con un leve desfase. Este desfase de localización aumenta con el tiempo por la falta de recursos que provoca el retardo del análisis de los sensores.
 4. Intento 4, navegación a “bottle”: Para realizar este intento, primero se mueve el robot a otra área del mapa para que el robot pueda generar una ruta diferente, más complicada y evitando obstáculos. Este intento de navegación se logra exitosamente.
 5. Intento 5, navegación a “cup”: Al intentar navegar, en la consola se puede observar un error al iniciar de “Aborting goal”. Se toma la decisión de reiniciar Nav2.
 6. Intento 6, navegación a “cup”: Navega de forma exitosa.

Análisis cuantitativo

- **1. Detección de objetos**
 - Objetos reales: 2 (cup, bottle)
 - Objetos detectados: 3 (cup, bottle, bird)
 - Precisión: **66.6 %**
 - Distancia promedio de detección: **35 cm**
- **2. Localización**
 - cup:
 - Localización real: (0.5, 0.16)
 - Localización mapa: (0.5, 0.16)

- bottle:
 - Localización real: (2.31, -1.9)
 - Localización mapa: (2.31, -1.9)
- Error euclidiano medio: **0 cm**

■ 3. Navegación

- Navegación exitosa:
 - Objetos alcanzados: 4
 - Total de intentos: 6
- Navegación exitosa sin colisiones:
 - Objetos alcanzados sin colisiones: 4
 - Total de intentos: 6
- Tasa de navegación exitosa: **66,6 %**
- Tasa de éxito sin colisiones: **66,6 %**
- Tiempo promedio de navegación: **35,75 segundos**

Visualización del experimento

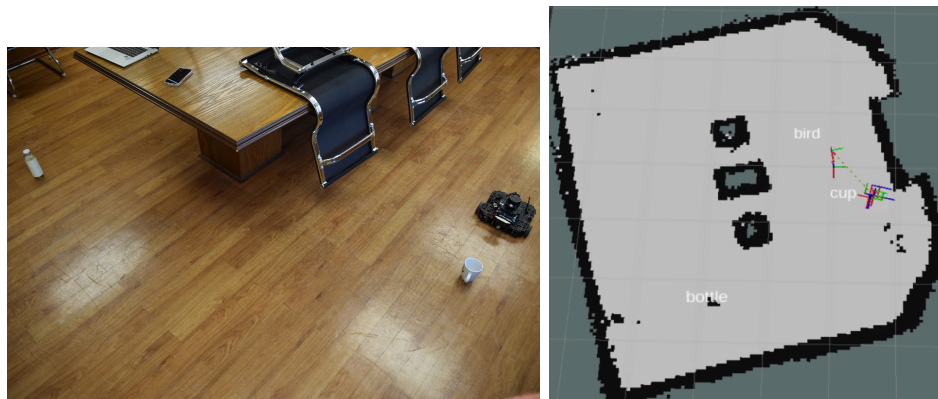


Figura 28: Comparación entre el entorno real y el mapa generado con los objetos detectados.

4.3. Otros experimentos

4.3.1. Prueba 1 anexa: Mapa 1, dos objetos

Descripción del experimento En este experimento no se comienza a hacer un mapa desde cero, sino que se utiliza uno de un experimento anterior. Se ejecutan los códigos sobre esa base y se analizan dos objetos.

Resultados

- **Detección:** hubo solo la detección de “bottle”.
- **Localización:** “bottle” se localiza de forma correcta, pero “cup” se encuentra en el mapa, pero realmente no se considera en la localización ya que estaba en el mapa previo.
- **Navegación:** al momento de realizar la navegación se evidencia la falta de objetos en la base de datos donde se guardan las localizaciones. Esto ocurre porque se inicia el experimento en un mapa previo y eso conlleva a la pérdida de datos anteriores al experimento actual.

Este experimento no se considera para el cálculo de resultados, ya que no aporta información relevante a nivel numérico. A nivel práctico y de aprendizaje, se considera siempre realizar pruebas desde cero.

Visualización del experimento

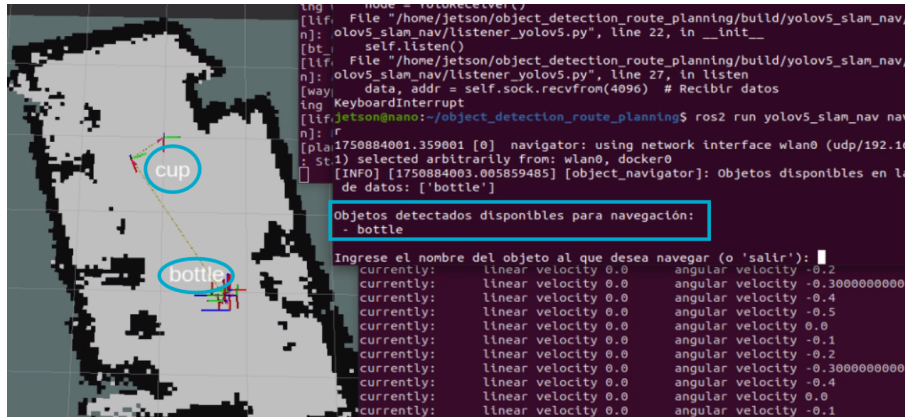


Figura 29: Mapa con objetos localizados y consola que muestra solo un objeto localizado

4.3.2. Prueba 2 anexa: Mapa 2, dos objetos

Este experimento no se considera para el análisis cuantitativo porque existen varios errores externos que dificultan el cálculo. Aun así, se agrega la información relacionada a este, ya que ayuda a entender que la calidad del espacio debe ser la más apropiada para el Turtlebot3, como por ejemplo, eliminar del espacio obstáculos que el LiDAR no pueda detectar.

- Detección de objetos: correcta detección.
- Localización de objetos en el mapa: error mínimo al localizar los objetos.
- Navegación hacia objetos:
 - Intento 1 navegación a “bottle”: No navega, la consola de Nav2 entrega un error llamado “Aborting handle”. Se reinicia Nav2.
 - Intento 2 navegación a “bottle”: Comienza a navegar, pero se atasca con la base de una silla. Con intervención, el robot pudo volver a desplazarse y llega al objetivo. Esta acción repercute en el mapa, ya que se genera un obstáculo que no existe en el mapa en forma de línea.
 - Intento 3 navegación a “cup”: Al intentar navegar, en la consola de Nav2 aparece un error llamado ERROR NAVIGATION FAILED. Se cree que pasa por la cercanía al obstáculo, ya que el robot tiene un área de seguridad y suele confundirse si se encuentra próximo a algún objeto u obstrucción. Esta falla ocurre por el inconveniente de la base de la silla. Se opta por movilizar el robot con teleop de forma manual, alejarlo del obstáculo y reiniciar Nav2.
 - Intento 4 navega a “cup”: Navega de forma correcta hasta que se queda atrapado con un cable. Esto ocasiona que no pueda terminar la navegación.

Visualización del experimento

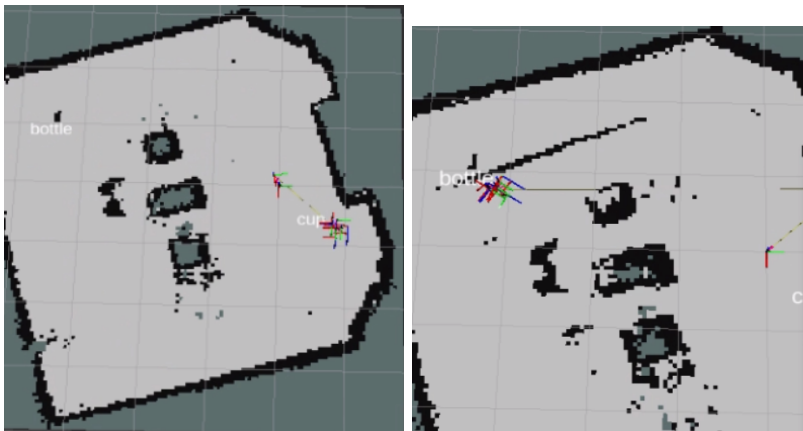


Figura 30: Mapa antes y después del inconveniente con base de silla

4.4. Análisis de los resultados

A continuación, se presenta el análisis de los resultados, estructurado en función de los tres ejes evaluados: detección, localización y navegación. Este se elabora a partir del análisis cuantitativo y cualitativo de los resultados obtenidos en las pruebas realizadas en distintos escenarios, con el objetivo de evaluar el desempeño del sistema propuesto en condiciones reales y controladas. Por otra parte, se analizará el rendimiento de los recursos dentro del sistema.

Detección de objetos: El modelo ligero de YOLOv5 presenta buena precisión en distancias cortas, pero no detecta objetos cuando las distancias son mayores a **37 centímetros** en promedio. Los falsos positivos se le atribuyen a la complejidad del modelo. La precisión de la detección fue de **92 %** en promedio de las pruebas del Mapa 1 y de **66,6 %** en promedio de las pruebas del Mapa 2. El promedio de la precisión de todos los experimentos realizados es del **87,8 %** de precisión.

Localización de objetos: Se evidencia un error sistemático en la estimación de la posición de los objetos en el mapa, con un desfase promedio de **14,8 cm** en dirección hacia el robot. Esto se atribuye a errores en el cálculo de distancia y posibles desincronizaciones entre TF, la imagen y el LiDAR.

Navegación: El sistema de navegación alcanza una tasa de éxito del **67,2 %**, con fallos frecuentes debido a errores en el action server y conflictos con obstáculos no representados adecuadamente en el mapa. Se observó que el rendimiento del sistema mejora significativamente tras reiniciar el módulo de navegación (Nav2) cuando se presentan errores. Por otra parte, la tasa de navegación exitosa sin colisiones alcanza un **52,8 %**, inferior a la anterior. Esto se debe a que, en los primeros experimentos relacionados al mapa 1, el sistema no considera como obstáculos aquellos objetos que el LiDAR no puede detectar (principalmente por estar por debajo de la altura mínima de escaneo), causando colisiones recurrentes. En los experimentos relacionados al mapa 2, se incorpora una corrección de código para incluir estos obstáculos de forma virtual, lo que permite al robot realizar navegaciones sin colisiones. Finalmente, el tiempo promedio de navegación fue de **17,8 segundos**, el cual depende de la complejidad del mapa, el tamaño y la distancia entre un objeto y otro.

Recursos del sistema: El proceso de detección de objetos fue uno de los que consumió más recursos, alcanzando un pico de uso de **CPU del 32,9 %** y una memoria residual de **21 MB**. La ejecución de este script fue intensiva, ya que las tareas de visión por computador suelen requerir una gran cantidad de procesamiento. Por otra parte, SLAM muestra con un uso de **CPU moderado**, alrededor del **24 %**, y un consumo de memoria notablemente alto, alrededor de **2,6 GB**, lo que indica la complejidad del mapa que se construye y actualiza en tiempo real. Los nodos creados para la comunicación y localización en el mapa tuvieron un uso de recursos relativamente bajo, lo que demuestra su eficiencia en la construcción de estos. Finalmente, RViz2 tuvo el uso de **CPU más alto** del sistema, superando el **104 %** (lo que en sistemas de multinúcleos indica un uso intensivo de estos que podrían estar distribuido en varios núcleos) y su **consumo de memoria fue considerable**, alrededor de **4,8 GB**. Esto destaca que la visualización gráfica es una de las tareas más demandantes en este tipo de sistemas.

Visualización de cumplimiento de objetivos: A continuación, se expone una tabla con indicadores y resultados de los experimentos realizados:

Objetivo	Criterios / Indicadores	Observaciones / Resultados
Detectar objetos con la cámara	Detectar objetos relevantes en el entorno con precisión $\geq 80\%$ usando modelo pre-entrenado YOLOv5s	Se utilizó cámara web con cálculo de distancia en código en vez de RealSense D435; precisión promedio 87,8 %
Localizar objetos en el mapa	Errores de localización ≤ 10 cm	Error promedio 14,8 cm, desfase por sincronización entre sensores
Navegar hacia objetos	Alcanzar objetos con y sin colisionar con otros objetos; tasa de navegación exitosa $\geq 90\%$; tasa de navegación exitosa sin colisiones $\geq 85\%$	Tasa de navegación exitosa 67,2 %, tasa de navegación exitosa sin colisiones 52,8 %
Evaluar rendimiento del sistema	CPU $\leq 30\%$ y memoria $\leq 3,5$ GB durante ejecución	CPU pico 32,9 %, memoria 4,8 GB; sistema al borde del colapso durante pruebas

Cuadro 7: Objetivos y criterios de evaluación de los experimentos realizados.

5. Conclusiones

En esta sección se exponen las conclusiones del proyecto de memoria de título, en función de los objetivos planteados y los resultados obtenidos durante el desarrollo experimental. A través del análisis de las etapas de detección, localización y navegación, fue posible identificar tanto los logros alcanzados como las limitaciones del sistema propuesto. Estas conclusiones permiten valorar el desempeño del enfoque implementado y ofrecer una base para futuras mejoras y líneas de investigación.

Cabe destacar, que la integración de componentes en robótica es fundamental para el desarrollo de sistemas autónomos. La combinación de navegación, percepción y visión robótica permite a los sistemas asistir en tareas que van desde el apoyo a personas con movilidad reducida en entornos domésticos, hasta misiones de exploración espacial como la Luna o Marte. Además, el uso de plataformas de tamaño reducido, como la Jetson Nano, resulta relevante en estos escenarios, ya que su bajo consumo energético y tamaño compacto facilitan la movilidad y la integración en espacios limitados. Sin embargo, estas ventajas también traen consigo desafíos técnicos, como restricciones de rendimiento, compatibilidad de software y capacidad de procesamiento, que deben ser cuidadosamente considerados al momento de diseñar e implementar un sistema robótico. Por ello, este tipo de investigación y desarrollo no solo contribuye al avance tecnológico, sino que también promueve soluciones accesibles, replicables y escalables para distintas áreas de la ciencia y la sociedad.

5.1. Conclusiones por ejes evaluados

Dado lo expuesto en el análisis de resultados, se puede afirmar que se cumplieron los objetivos planteados en este proyecto. El sistema logra detectar objetos con una precisión promedio del **87.8 %**, cumpliendo con el porcentaje de **éxito** definido. Por otra parte, el robot localiza las posiciones de los objetos en el mapa con un desfase promedio de **14.8 cm**, valor **aceptable** dentro del umbral descrito. Además el robot logra realizar navegaciones exitosas en un **67,2 %** de los intentos, valor que se considera **aceptable** y logra realizar navegaciones exitosas sin colisiones en un **52.8 %** de los intentos, el cual es un **fallo** para este tipo de navegación. Si bien la tasa de éxito en la navegación se puede mejorar, el sistema fue capaz de completar satisfactoriamente el flujo completo de detección-localización-navegación, cumpliendo así con el objetivo principal del proyecto.

5.2. Limitaciones

El sistema desarrollado presenta resultados funcionales, pero el desempeño se ve condicionado por una serie de limitaciones técnicas y operativas que influyen directamente en su estabilidad y precisión. Uno de los factores más relevantes es la capacidad computacional acotada de la Jetson Nano, la cual condiciona todo el flujo de trabajo. Esta limitación se manifiesta en diversos aspectos: lentitud al iniciar y ejecutar nodos, necesidad de restringirse a modelos de detección ligeros (lo que disminuye la precisión de la detección de objetos) y retardo en la lectura de sensores. Este último punto afecta especialmente las transformaciones de referencia espacial (TF), generando advertencias sobre latencia de frames como **base_scan**, lo que dificulta obtener con precisión la posición actual del robot y compromete la navegación, localización de objetos y la creación del mapa. Algo importante a destacar es que no importa la posición de los objetos en el entorno, el rendimiento limitado de hardware del sistema colabora con el aumento de errores en los experimentos.

Asimismo, se identifican errores recurrentes en el sistema de navegación que están asociados a la sobrecarga de procesos en la Jetson Nano, lo que provoca fallos en el servidor de acciones y respuestas erráticas durante la planificación de rutas. A esto se suma la arquitectura ARM64 de la plataforma, la cual implica desafíos adicionales al momento de instalar librerías y herramientas, ya que muchas de ellas no cuentan con soporte directo o requieren ajustes específicos para funcionar correctamente.

La integración de sensores también representa una dificultad significativa. Aunque se considera que una cámara de profundidad podría mejorar notablemente la estimación de distancias, su integración con la Jetson Nano se descarta debido a la complejidad técnica que representa bajo esta arquitectura.

La autonomía energética, relacionada a la batería del sistema, es otra limitación importante. La batería del TurtleBot3 tiene una duración reducida y no cuenta con un indicador de carga accesible desde el sistema operativo, por lo que se debe estimar manualmente el tiempo restante en cada sesión. Esto dificulta la planificación de pruebas prolongadas y pone en riesgo la pérdida de avances relevantes.

El entorno físico donde el robot va a moverse impone restricciones. Debido a la altura del LiDAR, el robot no detecta objetos de menor tamaño que él, lo cual genera colisiones con obstáculos bajos si no se realiza una compensación manual o virtual de estos elementos.

Estos resultados confirman que la integración de componentes en robótica es un proceso complejo, que exige una planificación detallada, una comprensión profunda de los recursos disponibles y una actitud constante de análisis y adaptación. Las limitaciones encontradas permiten no solo identificar puntos críticos de mejora, sino también aprender a diseñar flujos de trabajo eficientes dentro de un entorno limitado.

5.3. Trabajos Futuros

Si bien el sistema cumple con los objetivos planteados, los resultados obtenidos evidencian diversas oportunidades de mejora que pueden potenciar significativamente su desempeño.

En primer lugar, se recomienda reemplazar el modelo actual de detección por una versión más precisa y optimizada, para que el sistema pueda obtener mejores resultados en cuanto a detección de objetos, para que exista una menor cantidad de falsos positivos y además reducir la carga de recursos en el dispositivo.

Asimismo, la incorporación de una cámara de profundidad facilita una estimación precisa de las distancias, reduciendo errores sistemáticos en la localización de objetos en el mapa. En relación con esto, también se sugiere implementar un mecanismo de localización continua en tiempo real, en lugar de registrar la posición de los objetos únicamente al momento de su primera detección. Esto permite actualizar su ubicación en forma dinámica y mejorar la coherencia espacial del entorno percibido.

Incorporar tecnologías de imagen 3D, resulta beneficioso, como cámaras de profundidad o sistemas de visión estereoscópica, ya que estas permiten obtener información precisa sobre la forma, el tamaño y la ubicación de los objetos en el entorno. Esto resulta útil para mejorar la navegación del robot, al proporcionarle un entendimiento más completo de la geometría del espacio y de los obstáculos presentes. Además, la detección de objetos puede beneficiarse al contar con datos tridimensionales, lo que permitiría diferenciar mejor entre objetos cercanos y lejanos, así como estimar su volumen real. De esta manera, la localización de objetos en el mapa puede ser más exacta, facilitando la planificación de rutas y la interacción segura y eficiente del robot con su entorno.

Por otro lado, el sistema de navegación puede optimizarse mediante una revisión detallada de los casos borde, especialmente en entornos con alta densidad de obstáculos. Es fundamental considerar tanto el área de seguridad del TurtleBot3 como el ángulo de aproximación al objeto destino, con el fin de evitar colisiones y trayectorias poco eficientes.

Algo muy importante a mejorar es el menú interactivo por consola. Se puede crear una aplicación interactiva, la cual el usuario pueda utilizar sin necesidad de saber computación.

Finalmente, se propone investigar estrategias de optimización específicas para arquitecturas ARM64 como la de la Jetson Nano. Esto permitiría ampliar las funcionalidades del sistema sin comprometer su estabilidad, habilitando el desarrollo de soluciones más complejas y robustas en futuros trabajos.

5.4. Experiencia y aprendizajes

La realización de este proyecto me permitió aprender profundamente, tanto a nivel científico como personal. Uno de los aprendizajes más importantes fue comprender la importancia de contar con una estructura clara y objetivos bien definidos desde el inicio. Sin embargo, también entendí que la flexibilidad es esencial: los planes pueden cambiar, y saber adaptarse es parte del proceso. Para lograrlo, es indispensable investigar, leer y formarse constantemente. Aunque la documentación técnica puede ser densa, siempre existe un video, un artículo o una comunidad dispuesta a ayudar a comprender mejor los conceptos.

En este camino descubrí que el aprendizaje en robótica requiere tiempo y paciencia. No es posible dominar temas tan complejos como los nodos, las estimaciones de distancia, los sensores o los marcos de referencia de la noche a la mañana. Es necesario detenerse, revisar, cometer errores y volver a intentarlo, tantas veces como sea necesario. Este proceso no solo me entregó conocimientos técnicos, sino también aprendizajes emocionales. Comprendí que equivocarse es parte del camino, y que, si uno se compromete y persevera, los resultados eventualmente llegan.

También descubrí el valor de conversar con personas que dominan el área. Ningún artículo o tutorial reemplaza la claridad y perspectiva que alguien con experiencia puede entregar. Aprendí que generar redes y nutrirse del conocimiento colectivo es clave para avanzar.

Y quizás lo más valioso: este proyecto me recordó la importancia de rodearse de personas que te valoran y te apoyan. No podría haber terminado esta memoria de título sin el círculo hermoso que me acompaña. Aunque muchas de esas personas estén lejos físicamente, su presencia constante me dio fuerza en los momentos más difíciles. Estoy convencida de que logré concluir este trabajo gracias al cariño, el apoyo incondicional y la confianza que me brindaron.

Bibliografía

- [1] Akyildiz, I. F. and Jornet, J. M. (2020). *Randomly Deployed Wireless Sensor Networks*. Academic Press.
- [2] Andrus, B. (Abril 2, 2024). ¿qué es docker? desglosando esta popular plataforma de aplicaciones. <https://www.dreamhost.com/blog/es/que-es-docker/>.
- [3] Asadullah (2017). Distance measurement using single camera. https://github.com/Asadullah-Dal17/Distance_measurement_using_single_camera.
- [4] avdheshchouhan ges (2024). Real-time object detection and navigation with turtlebot3 and qualcomm rb5. <https://github.com/globaledgesoft/Object-Detection-and-Navigation-with-TurtleBot3-and-Qualcomm-RB5?tab=readme-ov-file#real-time-object-detection-and-navigation-with-turtlebot3-and-qualcomm-rb5>.
- [5] bihonght (2024). Turtlebot3 red brick finding. <https://github.com/bihonght/TurtleBot3-Autonomous-Finding-Object>.
- [6] Boettiger, C. (2015). An introduction to docker for reproducible research, with examples from the r environment. *arXiv preprint arXiv:1410.0846*.
- [7] Bradski, G. and Kaehler, A. (2008). *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media.
- [8] Cangelosi, A. and Asada, M. (2022). *Cognitive Robotics*. Intelligent Robotics and Autonomous Agents series. MIT Press.
- [9] Carrillo-Arce, L. C., Sánchez-Cuevas, P. J., Rodríguez-Ramos, A., and Ollero, A. (2024). A benchmarking dataset for outdoor visual navigation with uavs in real conditions. *Robotics*, 14(4):43.
- [10] Cloudflare, Inc. (n.d.). ¿Qué es el Protocolo de Datagrama de Usuario (UDP)? <https://www.cloudflare.com/es-es/learning/ddos/glossary/user-datagram-protocol-udp/>.
- [11] Corporation, N. (2025a). Cuda toolkit documentation 12.8 update 1. <https://docs.nvidia.com/cuda/>.
- [12] Corporation, N. (2025b). Jetson nano. <https://www.nvidia.com/es-la/autonomous-machines/embedded-systems/jetson-nano/product-development/>.
- [13] Corporation, N. (2025c). Jetson nano. <https://developer.nvidia.com/embedded/jetson-nano>.
- [14] Crespo, J., Castillo, J. C., Mozos, O. M., and Barber, R. (2020). Semantic information for robot navigation: A survey. *Applied Sciences*, 10(2).
- [15] Durrant-Whyte, H. and Bailey, T. (2006). Simultaneous localization and mapping: part i. *IEEE Robotics & Automation Magazine*, 13(2):99–110.
- [16] enrique a. (2018). Detección de objetos con yolo: implementaciones y como usarlas. <https://medium.com/@enriqueav/detecci%C3%B3n-de-objetos-con-yolo-implementaciones-y-como-usarlas-c73ca2489246>.
- [17] goli-mary m (2023). Turtlebot3-object-tracker. <https://github.com/goli-mary-m/TurtleBot3-Object-Tracker>.

-
- [18] Guda, T., Kanumuri, B., Wang, K., Mohan, R., Khojasteh, M. J., Trivedi, M. M., and Riek, L. D. (2024). An adaptable, safe, and portable robot-assisted feeding system. *arXiv preprint arXiv:2403.04134*.
- [19] Howse, J. and Minichino, J. (2020). Learning OpenCV 4 computer vision with python 3, third edition. Code repository on GitHub. <https://github.com/PacktPublishing/Learning-OpenCV-4-Computer-Vision-with-Python-Third-Edition>.
- [20] IRobot (2022). Waymo on uber. <https://store.irobot.cl/producto/aspiradora-y-trapeadora-robot-roomba-i5/>.
- [21] Jocher, G. (2020). YOLOv5 by Ultralytics.
- [22] kangalow (2023). Nomachine -jetson remote desktop. <https://jetsonhacks.com/2023/12/03/nomachine-jetson-remote-desktop/>.
- [23] Kich, V. A., Muttaqien, M. A., Toyama, J., Miyoshi, R., Ida, Y., Ohya, A., and Date, H. (2024). Precision and adaptability of yolov5 and yolov8 in dynamic robotic environments. *arXiv preprint arXiv:2406.00315*.
- [24] Korthals, T., Mohammadi, A., Liu, J., and Behnke, S. (2024). Aerial robot navigation in dense forests using learning-based perception and mapping. *Science Robotics*, 9(94).
- [25] Labs, U. (2025). Stereo vision for depth perception and navigation in robotics. <https://www.unitxlabs.com/resources/stereo-cameras-machine-vision-system-3d-depth-perception/> (consultado August 2025).
- [26] Li, B., Xu, Y., Cappelleri, D., and Zhang, S. (2017). High-accuracy, high-speed 3d structured light imaging techniques and potential applications to intelligent robotics. *Journal of Intelligent & Robotic Systems*. Springer Singapore.
- [27] Liang, J., Patel, U., Sathyamoorthy, A., and Manocha, D. (2020). Crowd-steer: Realtime smooth and collision-free robot navigation in densely crowded scenarios trained using high-fidelity simulation. pages 4193–4200.
- [28] Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., and Dollár, P. (2015). Microsoft coco: Common objects in context.
- [29] LLC, W. (2025). Roomba combo i5+ nueva aspiradora y trapeadora robot 2 en 1 irobot wifi. <https://waymo.com/intl/es/waymo-on-uber/>.
- [30] Ltd, R. P. (2025). Raspberry pi 3 model b+. <https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/>.
- [31] Macenski, S. (2025). slam_toolbox. https://wiki.ros.org/slam_toolbox.
- [32] Macenski, S. and contributors (2023a). Navigation2 (nav2) stack documentation. <https://docs.nav2.org/>. Accessed: 2025-04-22.
- [33] Macenski, S. and contributors (2023b). Planner server. https://docs.nav2.org/setup_guides/algorithm/select_algorithm.html#planner-server. Accessed: 2025-04-22.
- [34] Macenski, S., Foote, T., Gerkey, B., Lalancette, C., and Woodall, W. (2022). Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074.

-
- [35] Mistakes, J. . M. (2025a). Opencr setup. https://emanual.robotis.com/docs/en/platform/turtlebot3/opencr_setup/.
- [36] Mistakes, J. . M. (2025b). Robotis e-manual, features. <https://emanual.robotis.com/docs/en/platform/turtlebot3/features/>.
- [37] Mistakes, J. . M. (2025c). Robotis e-manual, lds-01. https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/.
- [38] Mistakes, J. . M. (2025d). Robotis e-manual, opencr. https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_opencr1_0/.
- [39] Mistakes, J. . M. (2025e). Robotis e-manual, principal. <https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/>.
- [40] Nguyen, T.-K., Vu, L. T., Vu, V. Q., Hoang, T.-D., Liang, S.-H., and Tran, M.-Q. (2021). Analysis of object detection models on duckietown robot based on yolov5 architectures. *International Journal of iRobotics*, 4(4):17–22.
- [41] NoMachine (2025). Nomachine. <https://www.nomachine.com/>.
- [42] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pages 8024–8035.
- [43] Postel, J. (1980). Rfc 768: User datagram protocol. <https://www.rfc-editor.org/rfc/rfc768.txt>. Internet Engineering Task Force.
- [44] Pyrestone (2019). jetson-fan-ctl. <https://github.com/Pyrestone/jetson-fan-ctl>.
- [45] Qengineering (2021). Jetson nano with ubuntu 20.04 os image. <https://github.com/Qengineering/Jetson-Nano-Ubuntu-20-image/tree/main>.
- [46] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection.
- [47] Robotics, O. (2025a). Introducing tf2. <https://docs.ros.org/en/foxy/Tutorials/Intermediate/Tf2/Introduction-To-Tf2.html>.
- [48] Robotics, O. (2025b). Nodes in ros 2. <https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-RS2-Nodes/Understanding-RS2-Nodes.html>.
- [49] Robotics, O. (2025c). rviz2. <https://docs.ros.org/en/humble/p/rviz2/>.
- [50] Robotics, O. (2025d). Understanding actions. <https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-RS2-Actions/Understanding-RS2-Actions.html>.
- [51] Robotics, O. (2025e). Understanding services. <https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-RS2-Services/Understanding-RS2-Services.html>.
- [52] Robotics, O. (2025f). Understanding topics. <https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-RS2-Topics/Understanding-RS2-Topics.html>.
- [53] Robotics, O. (2025g). Using colcon to build packages. <https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Colcon-Tutorial.html>.

-
- [54] Robotis (2025). Slam. <https://emanual.robotis.com/docs/en/platform/turtlebot3/slam/>.
- [55] Sj-Amani (2020). Jetson nano setup for turtlebot3. https://github.com/Sj-Amani/Jetson_Nano_Setup_For_TurtleBot3.
- [56] Sony (n.d.). ¿qué es la distancia focal? <https://www.sony.es/electronics/support/articles/00267921>.
- [57] Sucar, L. E. and Gómez, G. (2012). *Visión computacional*. Instituto Nacional de Astrofísica, Óptica y Electrónica, Puebla, México. <https://ccc.inaoep.mx/~esucar/Libros/vision-sucar-gomez.pdf>.
- [58] Tan, H., Zhao, F., Song, H., and Liu, Z. (2023). Quantifying the impact of deployments of autonomous vehicles and intelligent roads on road safety in china: A country-level modeling study. *International Journal of Environmental Research and Public Health*, 20(5):4069.
- [59] Ultralytics (2023). YOLOv5 vs efficientdet: Speed and accuracy comparison. <https://medium.com/@enriqueav/detecci%C3%B3n-de-objetos-con-yolo-implementaciones-y-como-usarlas-c73ca2489246>. Accessed: 2025-08-09.
- [60] Usamentiaga, R., García, D., et al. (2022). Robot guidance using machine vision techniques in industrial environments: a comparative review. *Baghdad Science Journal*. PMC: comparative review of photogrammetry, stereo, structured light, ToF, triangulation laser for robot guidance.
- [61] Wijaya, R. S., Hasibuan, S., Wibisana, A., Jamzuri, E. R., and Nugroho, M. A. B. (2024). Comparative study of deep learning algorithms between yolov5 and mobilenet-ssdv2 as fast and robust outdoor object detection solutions. In *Proceedings of the 7th International Conference on Applied Engineering (ICAE 2024)*, pages 94–106. Atlantis Press.
- [62] Zimmermann, H. (1980). Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432.

6. Anexo

6.1. Descarga y configuración de contenedor de Docker

1. Identificación de contenedores e imágenes

Obtener el ID y nombre del contenedor en ejecución:

```
docker ps
```

Obtener la imagen asociada a un contenedor específico:

```
docker inspect --format='{{.Config.Image}}' <ID_del_contenedor>
```

2. Abrir el contenedor Docker

Antes de abrir el contenedor, se otorgan permisos de acceso gráfico:

```
xhost +local:root
```

Luego se inicia y accede al contenedor:

```
sudo docker start yolov5_contenedor  
sudo docker exec -it yolov5_contenedor bash
```

3. Cerrar el contenedor y guardar cambios

Para salir del contenedor y guardar su estado actual en una nueva imagen:

```
exit
```

Luego, guardar los cambios realizados:

```
sudo docker commit -m "mensaje_de_cambios" nombre_contenedor nombre_imagen
```

Para verificar que la imagen fue guardada correctamente:

```
sudo docker images
```

4. Crear un contenedor desde una imagen previamente generada

Otorgar permisos de acceso gráfico:

```
xhost +local:root
```

Crear carpeta de trabajo:

```
mkdir workspace
```

Ejecutar una imagen base con soporte NVIDIA (JetPack 4.6)

```
sudo docker run -it --runtime nvidia --network host  
nvcr.io/nvidia/l4t-ml:r32.6.1-py3  
exit
```

Consultar los contenedores creados:

```
sudo docker ps -a
```

Renombrar el contenedor:

```
sudo docker rename <ID_o_nombre_actual> yolov5_contenedor
```

Crear una imagen desde ese contenedor:

```
sudo docker commit yolov5_contenedor yolov5_imagen
```

5. Crear un nuevo contenedor con acceso a la cámara y GUI

Para crear un contenedor basado en la imagen anterior, con acceso a la cámara y entorno gráfico:

```
sudo docker create -it --runtime nvidia --network host \  
--device /dev/video0 \  
-e DISPLAY=$DISPLAY \  
-v /tmp/.X11-unix:/tmp/.X11-unix \  
-v $(pwd):/workspace \  
--name yolov5_contenedor \  
yolov5_imagen
```

6. Volver a iniciar el contenedor

Cada vez que se desee reabrir el entorno:

```
sudo docker start yolov5_contenedor  
sudo docker exec -it yolov5_contenedor
```

Si es necesario eliminar el contenedor:

```
sudo docker rm yolov5_contenedor
```

6.2. Código de prueba de cámara dentro del contenedor *camera_test.py*

```
import cv2  
  
cap = cv2.VideoCapture(0)  
  
if not cap.isOpened():  
    print(" No se pudo abrir la camara")  
else:  
    print(" Camara detectada")  
  
while True:  
    ret, frame = cap.read()  
    if not ret:  
        print(" No se pudo capturar el frame")  
        break  
    cv2.imshow('Camera Test', frame)  
    if cv2.waitKey(1) & 0xFF == ord('q'):  
        break  
  
cap.release()  
cv2.destroyAllWindows()
```

6.3. Código de detección de objetos *detect.py*

```
# -*- coding: utf-8 -*-
import torch
import cv2
import numpy as np
import socket
import json
from models.experimental import attempt_load
from utils.general import non_max_suppression, scale_coords
from utils.datasets import letterbox
import time

COCO_CLASSES = [
    "person", "bicycle", "car", "motorbike", "aeroplane",
    "bus", "train", "truck", "boat", "traffic light",
    "fire hydrant", "stop sign", "parking meter", "bench",
    "bird", "cat", "dog", "horse", "sheep", "cow", "elephant",
    "bear", "zebra", "giraffe", "backpack", "umbrella",
    "handbag", "tie", "suitcase", "frisbee", "skis", "snowboard",
    "sports ball", "kite", "baseball bat", "baseball glove",
    "skateboard", "surfboard", "tennis racket", "bottle",
    "wine glass", "cup", "fork", "knife", "spoon", "bowl",
    "banana", "apple", "sandwich", "orange", "broccoli",
    "carrot", "hot dog", "pizza", "donut", "cake", "chair",
    "sofa", "pottedplant", "bed", "diningtable", "toilet",
    "tvmonitor", "laptop", "mouse", "remote", "keyboard",
    "cell phone", "microwave", "oven", "toaster", "sink",
    "refrigerator", "book", "clock", "vase", "scissors",
    "teddy bear", "hair drier", "toothbrush"
]

UDP_IP = "0.0.0.0"
UDP_PORT = 5005
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

weights = 'yolov5s.pt'
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = attempt_load(weights, map_location=device)
model.eval()

WIDTH = 1280
HEIGHT = 720
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, WIDTH)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, HEIGHT)
cv2.namedWindow("Deteccion con COCO Dataset", cv2.WINDOW_NORMAL)

if not cap.isOpened():
    print("No se pudo abrir la camara")
    exit()

print("Camara detectada. Enviando datos por UDP. Presiona 'q' para salir.")
```

```

fps = cap.get(cv2.CAP_PROP_FPS)
if fps == 0:
    fps = 30

frame_interval = 1.0 / fps

REAL_WIDTHS = {
    "bird": 10.9,
    "cat": 20.0,
    "dog": 25,
    "umbrella": 90,
    "handbag": 30,
    "bottle": 7,
    "cup": 8,
    "fork": 2.5,
    "knife": 2.5,
    "spoon": 3,
    "bowl": 20,
    "banana": 15,
    "apple": 8,
    "sandwich": 12,
    "orange": 9,
    "carrot": 14,
    "laptop": 35,
    "mouse": 6,
    "keyboard": 40,
    "cell phone": 7,
    "book": 14,
    "scissors": 8,
    "teddy bear": 25,
    "toothbrush": 3
}

calibrated = False
FOCAL_LENGTH = None

def calcular_focal(ancho_px, ancho_real_cm, distancia_cm):
    return (ancho_px * distancia_cm) / ancho_real_cm

def calcular_distancia(ancho_px, ancho_real_cm, focal):
    return (ancho_real_cm * focal) / ancho_px

while True:
    start_time = time.time()

    ret, frame = cap.read()
    if not ret:
        print("No se pudo capturar el frame.")
        break

    img = letterbox(frame, 640, stride=int(model.stride.max()))[0]
    img = img.transpose((2, 0, 1))[:-1]
    img = np.ascontiguousarray(img)
    img = torch.from_numpy(img).to(device).float() / 255.0
    img = img.unsqueeze(0)

```

```

pred = model(img)[0]
pred = non_max_suppression(pred, 0.25, 0.45, classes
    =[14,15,16,25,26,39,41,42,43,44,45,46,47,48,49,
    51,63,64,66,67,73,76,77,79])

detections = []
for det in pred:
    if len(det):
        det[:, :4] = scale_coords(img.shape[2:], det[:, :4], frame.
            shape).round()
        for *xyxy, conf, cls in det:
            class_id = int(cls)
            label_text = COCO_CLASSES[class_id]
            bbox_width = int(xyxy[2]) - int(xyxy[0])

            distancia_cm = 0
            if label_text in REAL_WIDTHTHS and bbox_width > 0:
                if not calibrated:
                    FOCAL_LENGTH = calcular_focal(bbox_width,
                        REAL_WIDTHTHS[label_text], 40)
                    print("Distancia focal calibrada con {}: {:.2f} px"
                        .format(label_text, FOCAL_LENGTH))
                    calibrated = True
                elif FOCAL_LENGTH is not None:
                    distancia_cm = calcular_distancia(bbox_width,
                        REAL_WIDTHTHS[label_text], FOCAL_LENGTH)
            if conf > 0.7 and distancia_cm < 50:
                detection = {
                    "xmin": int(xyxy[0]),
                    "ymin": int(xyxy[1]),
                    "xmax": int(xyxy[2]),
                    "ymax": int(xyxy[3]),
                    "confidence": float(conf),
                    "class": class_id,
                    "label": label_text,
                    "distance_m": round(distancia_cm / 100, 2) if
                        distancia_cm else 0.4
                }
                detections.append(detection)

            label = f'{label_text} {conf:.2f}'
            if distancia_cm:
                label += f' {distancia_cm / 100:.2f}m'
            cv2.rectangle(frame, (int(xyxy[0]), int(xyxy[1])), (int(
                xyxy[2]), int(xyxy[3])), (0, 255, 0), 2)
            cv2.putText(frame, label, (int(xyxy[0]), int(xyxy[1]) - 10)
                , cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)

```

```
if detections:
    message = json.dumps(detections)
    sock.sendto(message.encode(), (UDP_IP, UDP_PORT))

cv2.imshow('Deteccion con COCO Dataset', frame)
if cv2.waitKey(1) & 0xFF == ord('q'):
    break

elapsed_time = time.time() - start_time
sleep_time = frame_interval - elapsed_time
if sleep_time > 0:
    time.sleep(sleep_time)

cap.release()
cv2.destroyAllWindows()
sock.close()
print("Deteccion finalizada.")
```

6.4. Script de respaldo de contenedor de Docker *dockerRespaldo.sh*

```
#!/bin/bash
# -----
# CONFIGURACION INICIAL
# -----
# Nombres de contenedor e imagen
NEW_CONTAINER_NAME="yolov5_contenedor"
NEW_IMAGE_NAME="yolov5_imagen"

# Carpeta para respaldos
BACKUP_DIR="./respaldos"

# -----
# FUNCION: Verificar si pv esta instalado
# -----
check_pv() {
    if ! command -v pv &>/dev/null; then
        echo "'pv' no esta instalado. Quieres instalarlo? (s/n)"
        read -r respuesta
        if [[ "$respuesta" == "s" || "$respuesta" == "S" ]]; then
            echo "Instalando pv..."
            sudo apt update && sudo apt install -y pv
            if [[ $? -ne 0 ]]; then
                echo "Error al instalar 'pv'. Por favor instalalo manualmente."
                exit 1
            fi
        else
            echo "Continuando sin barra de progreso."
        fi
    fi
}

# -----
# RESPALDO (COMMIT + SAVE CON PV)
# -----
mkdir -p "$BACKUP_DIR"

CONTAINER_ID=$(docker ps -q -f name="$NEW_CONTAINER_NAME")
TIMESTAMP=$(date +"%Y%m%d_%H%M%S")
BACKUP_IMAGE_TAG="${NEW_IMAGE_NAME}:${TIMESTAMP}"
BACKUP_FILE="${BACKUP_DIR}/${NEW_IMAGE_NAME}_${TIMESTAMP}.tar"

echo ""
echo "Iniciando respaldo..."
echo "Contenedor: $NEW_CONTAINER_NAME (ID: $CONTAINER_ID)"
echo "Nueva imagen: $BACKUP_IMAGE_TAG"
echo "Archivo de respaldo: $BACKUP_FILE"

# Crear imagen desde el contenedor
docker commit "$CONTAINER_ID" "$BACKUP_IMAGE_TAG"
if [[ $? -ne 0 ]]; then
    echo "Error al hacer commit."
    exit 1
fi
echo "Commit exitoso."
```

```
# Verificar e instalar pv si se desea
check_pv

echo "Guardando imagen con barra de progreso..."

# Usar pv si esta disponible
if command -v pv &>/dev/null; then
    docker save "$BACKUP_IMAGE_TAG" | pv > "$BACKUP_FILE"
else
    docker save -o "$BACKUP_FILE" "$BACKUP_IMAGE_TAG"
fi

if [[ $? -ne 0 ]]; then
    echo "Error al guardar la imagen."
    exit 1
fi

echo "Imagen guardada en: $BACKUP_FILE"
echo "Respaldo completado con exito."
```


6.5. Código de nodo que recibe información de YOLOv5 *listener_yolov5.py*

```
import rclpy
from rclpy.node import Node
import socket
import json
from std_msgs.msg import String

class YoloReceiver(Node):
    def __init__(self):
        super().__init__('yolo_receiver')

        self.publisher_ = self.create_publisher(String, 'detecciones', 10)

        self.UDP_IP = "0.0.0.0"
        self.UDP_PORT = 5005
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind((self.UDP_IP, self.UDP_PORT))

        self.get_logger().info(f"Escuchando en {self.UDP_IP}:{self.UDP_PORT}")

        self.listen()

    def listen(self):
        while rclpy.ok():
            try:
                data, addr = self.sock.recvfrom(4096)
                self.get_logger().info(f"Datos recibidos de {addr}: {data.decode()}")

                detections = json.loads(data.decode())

                msg = String()
                msg.data = json.dumps(detections)
                self.publisher_.publish(msg)
                self.get_logger().info(f"Detecciones publicadas: {detections}")

            except Exception as e:
                self.get_logger().error(f"Error recibiendo datos: {e}")

def main(args=None):
    rclpy.init(args=args)
    node = YoloReceiver()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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

6.6. Código de nodo que localiza objetos en el mapa *map_updater.py*

```
import rclpy
from rclpy.node import Node
from nav_msgs.msg import OccupancyGrid
from std_msgs.msg import String
from visualization_msgs.msg import Marker
import tf2_ros
import json
import numpy as np
import os
from geometry_msgs.msg import PointStamped
from tf2_geometry_msgs.tf2_geometry_msgs import do_transform_point

class MapUpdater(Node):
    def __init__(self):
        super().__init__('map_updater')

        # Suscripción al mapa de SLAM Toolbox
        self.map_sub = self.create_subscription(OccupancyGrid, '/map', self
            .map_callback, 1)

        # Suscripción a las detecciones de YOLOv5
        self.yolo_sub = self.create_subscription(String, 'detecciones',
            self.yolo_callback, 2)

        # Publicador de marcadores en RViz2
        self.marker_pub = self.create_publisher(Marker, '/'
            'visualization_marker', 5)

        self.obstacle_pub = self.create_publisher(PointCloud2, '/'
            'virtual_obstacles', 10)
        #Llama a publish_virtual_obstacles 2 veces por segundo para
        #que los obstaculos virtuales esten siempre publicados
        self.obstacle_timer = self.create_timer(0.5, self.
            publish_virtual_obstacles)

        # Transformaciones TF
        self.tf_buffer = tf2_ros.Buffer(cache_time=rclpy.duration.Duration(
            seconds=20.0))
        self.tf_listener = tf2_ros.TransformListener(self.tf_buffer, self)

        self.current_map = None
        self.objects_db = {}

    def map_callback(self, msg):
        #Almacena el mapa actual
        self.current_map = msg

    def save_objects_to_file(self):
        #Guarda los objetos detectados en un archivo JSON
        script_dir = os.path.dirname(os.path.realpath(__file__))
        maps_dir = os.path.join(script_dir, "maps")
        os.makedirs(maps_dir, exist_ok=True)

        file_path = os.path.join(maps_dir, "objects_db.json")
        with open(file_path, "w") as file:
            json.dump(self.objects_db, file)
        self.get_logger().info(f"Objetos guardados en {file_path}")
```

```

def publish_virtual_obstacles(self, radius=0.2 , density=8):
    points = []
    for x, y in self.objects_db.values():
        for i in range(density):
            angle = 2 * np.pi * i / density
            px = x + radius * np.cos(angle)
            py = y + radius * np.sin(angle)
            points.append([px, py, 0.0])

    cloud = pc2.create_cloud_xyz32(
        header=self.create_header("map"),
        points=points
    )
    self.obstacle_pub.publish(cloud)

def create_header(self, frame_id):
    from std_msgs.msg import Header
    header = Header()
    header.stamp = self.get_clock().now().to_msg()
    header.frame_id = frame_id
    return header

def yolo_callback(self, msg):
    #Coloca los objetos detectados en el mapa con su nombre
    if self.current_map is None:
        self.get_logger().warn("Mapa no recibido aun, esperando")
        return
    detections = json.loads(msg.data)
    for detection in detections:
        obj_class = detection["class"]
        obj_name = self.get_coco_label(obj_class)

        if obj_name in self.objects_db:
            self.get_logger().info(f"Objeto '{obj_name}' se omite.")
            continue
        # Obtiene coordenadas del centro del objeto,
        #se calcula punto medio de la caja
        obj_x_pixel = (detection["xmin"] + detection["xmax"]) / 2
        obj_y_pixel = (detection["ymin"] + detection["ymax"]) / 2
        #Entrega coordenadas del centro del objeto y devuelve
        #coordenadas del objeto con respecto al robot
        obj_x_robot, obj_y_robot = self.image_to_robot(obj_x_pixel,
            obj_y_pixel, detection["distance_m"])
        #Entrega coordenadas el objeto con respecto al robot
        #y devuelve coordenadas que objeto en el mapa
        obj_x_mapa, obj_y_mapa = self.object_to_map(obj_x_robot,
            obj_y_robot)
        #Si la transformacion fallo, no colocar el marcador
        if obj_x_mapa is None or obj_y_mapa is None:
            continue
        # Publicar el marcador en RViz2
        self.publish_marker(obj_x_mapa, obj_y_mapa, obj_name)
        #Guarda las coordenadas del objeto en la base de datos
        self.objects_db[obj_name] = (obj_x_mapa, obj_y_mapa)
    self.save_objects_to_file()
    self.get_logger().info("Mapa actualizado.")

```

```

def image_to_robot(self, x_pixel, y_pixel, distance):
    #Convierte coordenadas de imagen a coordenadas del robot
    cam_fov = 90
    cam_width = 1280

    angle = (x_pixel / cam_width) * cam_fov - (cam_fov / 2)
    x_robot = distance * np.cos(np.radians(angle))
    y_robot = distance * np.sin(np.radians(angle))

    return x_robot, -y_robot

def object_to_map(self, x_obj, y_obj):
    #Convierte coordenadas del robot a coordenadas globales
    try:
        point_local = PointStamped()
        point_local.header.frame_id = "base_link"
        point_local.point.x = x_obj
        point_local.point.y = y_obj
        point_local.point.z = 0.0
        #Obtiene la pose del robot, la cual es una matriz de
        #transformacion que define donde esta la base del robot
        transform = self.tf_buffer.lookup_transform(
            'map', 'base_link', tf2_ros.Time(),
            timeout=rclpy.duration.Duration(seconds=0.5)
        )
        point_global = do_transform_point(point_local, transform)
        x_obj_map = point_global.point.x
        y_obj_map = point_global.point.y
        return x_obj_map, y_obj_map

    except Exception as e:
        self.get_logger().warn(f"No se pudo obtener la transformacion
            TF: {str(e)}")
        return None, None

def get_coco_label(self, class_id):
    COCO_CLASSES = [
        "person", "bicycle", "car", "motorbike", "aeroplane",
        "bus", "train", "truck", "boat", "traffic light",
        "fire hydrant", "stop sign", "parking meter", "bench",
        "bird", "cat", "dog", "horse", "sheep", "cow",
        "elephant", "bear", "zebra", "giraffe", "backpack",
        "umbrella", "handbag", "tie", "suitcase", "frisbee",
        "skis", "snowboard", "sports ball", "kite",
        "baseball bat", "baseball glove", "skateboard",
        "surfboard", "tennis racket", "bottle", "wine glass",
        "cup", "fork", "knife", "spoon", "bowl", "banana",
        "apple", "sandwich", "orange", "broccoli", "carrot",
        "hot dog", "pizza", "donut", "cake", "chair", "sofa",
        "pottedplant", "bed", "diningtable", "toilet",
        "tvmonitor", "laptop", "mouse", "remote", "keyboard",
        "cell phone", "microwave", "oven", "toaster", "sink",
        "refrigerator", "book", "clock", "vase", "scissors",
        "teddy bear", "hair drier", "toothbrush"
    ]
    return COCO_CLASSES[class_id] if class_id < len(COCO_CLASSES) else
        "Unknown"

```

```

def publish_marker(self, x, y, name):
    #Publica un marcador en RViz2 con el nombre del objeto
    marker = Marker()
    marker.header.frame_id = "map"
    marker.header.stamp = rclpy.time.Time().to_msg()
    marker.ns = "objects"
    marker.id = hash(name) % 1000
    marker.type = Marker.TEXT_VIEW_FACING
    marker.action = Marker.ADD
    marker.pose.position.x = x
    marker.pose.position.y = y
    marker.pose.position.z = 0.0
    marker.scale.z = 0.3
    marker.color.r = 1.0
    marker.color.g = 1.0
    marker.color.b = 1.0
    marker.color.a = 1.0
    marker.text = name

    self.get_logger().info(f"Publicando marcador: {name} en ({x}, {y})")
    self.marker_pub.publish(marker)

def main(args=None):
    rclpy.init(args=args)
    node = MapUpdater()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

6.7. Código de nodo para que el robot navegue *navigator.py*

```
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import PoseStamped
from tf_transformations import quaternion_from_euler
from nav2_msgs.action import NavigateToPose
from rclpy.action import ActionClient
import json
import os

class ObjectNavigator(Node):
    def __init__(self):
        super().__init__('object_navigator')
        self.action_client = ActionClient(self, NavigateToPose, '/navigate_to_pose')
        self.objects_db = self.load_objects_from_file()

    def load_objects_from_file(self):
        file_path = os.path.expanduser("/home/jetson/object_detection_route_planning/src/yolov5_slam_nav/maps/objects_db.json")

        if not os.path.exists(file_path):
            self.get_logger().warn(f"No se encontro el archivo {file_path}")
            return {}

        with open(file_path, "r") as file:
            objects_db = json.load(file)

        self.get_logger().info(f"Objetos disponibles en: {list(objects_db.keys())}")
        return objects_db
```

```

def navigate_to_object(self, obj_name):
    if obj_name not in self.objects_db:
        self.get_logger().warn(f"El objeto '{obj_name}' no fue encontrado")
        return
    x, y = self.objects_db[obj_name]
    yaw = 0.0
    q = quaternion_from_euler(0, 0, yaw)

    self.get_logger().info(f"Navegando hacia '{obj_name}' en ({x}, {y}) con orientacion fija yaw={yaw:.2f} rad")

    goal_msg = NavigateToPose.Goal()
    goal_msg.pose.header.frame_id = 'map'
    goal_msg.pose.header.stamp = self.get_clock().now().to_msg()
    goal_msg.pose.pose.position.x = x
    goal_msg.pose.pose.position.y = y
    goal_msg.pose.pose.orientation.x = q[0]
    goal_msg.pose.pose.orientation.y = q[1]
    goal_msg.pose.pose.orientation.z = q[2]
    goal_msg.pose.pose.orientation.w = q[3]

    self.action_client.wait_for_server()
    self.get_logger().info("Servidor de acciones de navegacion encontrado")

    try:
        self.send_goal_future = self.action_client.send_goal_async(goal_msg)
        self.send_goal_future.add_done_callback(self.goal_response_callback)
    except Exception as e:
        self.get_logger().error(f"Error enviando goal: {e}")

def goal_response_callback(self, future):
    goal_handle = future.result()
    if not goal_handle.accepted:
        self.get_logger().warn('La meta fue rechazada')
        return

    self.get_logger().info('Meta aceptada')
    self.get_result_future = goal_handle.get_result_async()
    self.get_result_future.add_done_callback(self.result_callback)

def result_callback(self, future):
    result = future.result()
    if result:
        self.get_logger().info('Meta alcanzada exitosamente')
    else:
        self.get_logger().warn('Error al alcanzar la meta')

```

```

def main(args=None):
    rclpy.init(args=args)
    node = ObjectNavigator()

    try:
        while rclpy.ok():
            print("\nObjetos detectados disponibles para navegacion:")
            print("0. salir")
            for obj in node.objects_db.keys():
                print(f" - {obj}")
            obj_name = input("\nIngrese el nombre del objeto al que desea
navegar:").strip()
            if obj_name.lower() in ['salir', 'exit', 'q']:
                break
            node.navigate_to_object(obj_name)
            rclpy.spin_once(node, timeout_sec=0.1)
    except KeyboardInterrupt:
        pass

    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```