Universidad de la República
Facultad de Ingeniería

Exploración Robótica Colaborativa de Interiores

Memoria de proyecto presentada a la Facultad de Ingeniería de la Universidad de la República por

Ricardo Ercoli (IE), Fausto Navadian (IE), Joaquín Urrisa (IC)

en cumplimiento parcial de los requerimientos para la obtención del título de Ingeniero Electricista o Ingeniero en Computación.

Tutor

Pablo Monzón ..... IIE, Universidad de la República
Facundo Benavides ..... InCo, Universidad de la República

Tribunal

Mauricio Gonzalez ..... IIE, Universidad de la República
Ewelina Bakala ..... InCo, Universidad de la República
Martin Llofriu ..... InCo, Universidad de la República

Montevideo
martes 31 octubre, 2023

Exploración Robótica Colaborativa de Interiores, Ricardo Ercoli (IE), Fausto Navadian (IE), Joaquín Urrisa (IC).

Esta tesis fue preparada en LaTeX usando la clase iietesis (v1.1).
Contiene un total de 163 páginas.
Compilada el martes 31 octubre, 2023.
http://iie.fing.edu.uy/

Agradecimientos

Queremos expresar nuestro sincero agradecimiento a quienes hicieron posible esta tesis de grado. En primer lugar, a Federica, Fernanda, Renata, nuestros familiares y amigos, cuyo apoyo, cariño y paciencia nos ha impulsado a lo largo de este proyecto y de nuestra carrera académica.

También deseamos reconocer a nuestros tutores, Pablo Monzón y Facundo Benavides, por su orientación experta y dedicación para guiar nuestro trabajo de investigación.

Extendemos nuestro agradecimiento a los integrantes del InCo por sus valiosas opiniones y colaboración en varios aspectos del trabajo.

Por último, agradecemos a la empresa Focus por permitirnos utilizar sus instalaciones para realizar pruebas y por brindarnos las herramientas necesarias para desarrollar un robot real. Su apoyo fue esencial para lograr nuestros objetivos en este proyecto.

Esta página ha sido intencionalmente dejada en blanco.

Resumen

Este proyecto se centra en la creación de mapas 2D de ambientes interiores desconocidos mediante la implementación de un sistema multi-robot cooperativo. El objetivo principal es reducir, en comparación a un sistema con un único robot, el tiempo total de exploración. Se focaliza en la robustez del sistema, eliminando el punto de falla único al trabajar con un sistema des-centralizado.

Se construye una flota de robots, donde la plataforma motora se basa en un robot educativo preexistente, incorporándole el hardware necesarios para lograr el mapeo colaborativo. Se diseñaron los elementos estructurales para la integración de cada uno de los componentes mecánicos y eléctricos.

Partiendo del trabajo realizado para el robot educativo, se modifica el firmware de control motor y se le agrega un módulo de comunicación con el fin de cumplir con los requerimientos del proyecto. Estas modificaciones se basan en los modelos obtenidos para el robot desarrollado.

En cuanto a la solución de alto nivel, se utiliza el *framework* ROS2, aprovechando implementaciones ya existentes para la localización, mapeo y navegación. Siguiendo el enfoque de exploración basada en fronteras, se desarrolla un algoritmo de agrupación de fronteras, utilizando los centroides de dichas agrupaciones como objetivos de navegación. Para la distribución de dichos objetivos se implementa un método de subastas des-centralizado.

Como metodología de trabajo se utiliza el simulador Gazebo, con el fin de desarrollar y testear las soluciones en un ambiente controlado, previo a la implementación física. Se realizan y presentan pruebas de alto nivel, tanto de la solución física como simulada.

Se concluye que la exploración colaborativa reduce los tiempos de exploración en comparación con la exploración individual. También se destaca que el uso de una flota coordinada produce resultados superiores en comparación con una flota no coordinada. En conjunto, estos resultados respaldan la importancia de la colaboración y la coordinación en la exploración.

Esta página ha sido intencionalmente dejada en blanco.

Tabla de contenidos

Tabla de contenidos

Tabla de contenidos

Capítulo 1

Introducción

La presente investigación se enmarca en el desarrollo de la tesis de grado titulada «Exploración Robótica Colaborativa de Interiores», o **ICRE** por su sigla en inglés, para la carrera de Ingeniería Eléctrica de Ricardo Ercoli y Fausto Navadian, y para la carrera de Ingeniería en Computación de Joaquín Urrisa. Este trabajo aborda aspectos fundamentales de la exploración robótica colaborativa. En particular se trabaja en la construcción, el modelado físico-matemático y el control de un robot, se desarrollan algoritmos con el fin de permitir el comportamiento cooperativo entre múltiples robots y se integran algoritmos de navegación y mapeo. Uno de los objetivos principales es el de derivar conclusiones relacionadas a las diferencias de trabajar con el simulador o en el entorno real.

1.1. Introducción al proyecto

Para el desarrollo de este proyecto, se ensamblaron tres robots que se utilizaron para llevar a cabo las pruebas de campo. Todo el trabajo realizado se encuentra documentado en dos repositorios diferentes en la plataforma gitlab.fing. Uno de los repositorios se enfoca en los aspectos relacionados a la parte motora de la plataforma[1], como el control de los motores y el ensamblaje del robot, mientras que el otro repositorio[2] abarca las temáticas de alto nivel como lo son la realización de Simultaneous Localization And Mapping (SLAM), la comunicación entre robots o la división de tareas, entre otras. Cada uno de estos repositorios cuenta con una documentación específica en forma de README.md. Además, este documento constituye el último componente del proyecto, en el cual se exponen los aspectos teóricos, el desarrollo del proyecto, las decisiones tomadas, los resultados obtenidos y las conclusiones alcanzadas a lo largo del trabajo.

[1]Repositorio ESP: https://gitlab.fing.edu.uy/ricardo.ercoli/icre
[2]Repositorio ROS2: https://gitlab.fing.edu.uy/fausto.navadian/icre-ros2

1.1.1. Motivación

En los últimos años, la exploración robótica ha ganado una importancia significativa debido a su amplio rango de aplicaciones en diversos campos como exploración de entornos hostiles, entre los que se destaca la exploración espacial, robots de servicio, para el auxilio o incluso para la industria, en lo que se llama "Industria 4.0". Los robots autónomos han demostrado ser herramientas valiosas para la exploración de entornos desconocidos y peligrosos, superando las limitaciones de los exploradores humanos en términos de resistencia física y habilidades especializadas. Por otra parte, la autonomía evita enfermedades en los operadores generadas por la teleoperación de los robots. La colaboración entre múltiples robots autónomos en tareas de exploración ha surgido como un área de investigación emocionante y prometedora [1].

1.1.2. Contexto y Justificación

La exploración robótica colaborativa se basa en la idea de que múltiples robots trabajando en conjunto puedan lograr resultados superiores en comparación a un solo robot actuando de manera aislada [1]. Estas soluciones ofrecen ventajas tales como una mayor eficiencia en la exploración, reducción del tiempo de exploración y una mejor cobertura del área de interés. Además, la colaboración entre robots permitiría la compensación de fallas individuales, la mejora de la robustez del sistema y la posibilidad de realizar tareas más complejas y especializadas.

1.1.3. Objetivos

El objetivo de este proyecto es diseñar y construir una flota de tres robots holonómicos[3] que de forma colaborativa logren realizar tareas de mapeo 2D de manera autónoma. En particular, estos robots deben ser construidos tomando como punto de partida la plataforma robótica «Robotito» desarrollada por el grupo MINA[4] y la solución debe ser desarrollada sobre el framework Robot Operating System 2 (ROS2) [2].

Con el fin de alcanzar este objetivo general, se establecieron los siguientes objetivos específicos:

- Estudiar antecedentes y el estado del arte en las temáticas abordadas, en particular relacionados a exploración autónoma colaborativa.
- Comprender la dinámica y el control de los movimientos del robot físico.

[3]Los robots holonómicos son aquellos que pueden moverse en cualquier dirección sin necesidad de cambiar la orientación de su cuerpo. Se describe más en detalle esta característica en la Sección 3.1.2.

[4]El MINA es un grupo dentro del Instituto de Computación de la Universidad de la República enfocado en temáticas de robótica móvil y gestión de redes de computadoras. MINA proviene de: Network Management / Artificial Intelligence

- Ensamblar tres robots partiendo de la plataforma móvil provista y modificarla para que los robots tengan las capacidades motoras, sensoriales y computacionales que permitan llevar a cabo el desarrollo de mapas 2D de manera colaborativa.
- Utilizar el entorno de desarrollo ROS2 en las plataformas Odroid de los robots. Instalar y configurar las bibliotecas necesarias para la comunicación y el intercambio de información entre los robots.
- Configurar y controlar el sistema motor y el sensor LiDAR (Light Detection and Ranging) de los robots utilizando ROS2.
- Integrar a los robots algoritmos de SLAM [3] que tengan la capacidad de aprovechar la colaboración entre los robots. Es decir, que se pueda construir un mapa en base a las medidas de todos los robots o que cada robot pueda construir su propio mapa obteniendo información tanto de sus sensores como de los otros robots.
- Realizar pruebas y simulaciones utilizando entornos virtuales para validar el funcionamiento y la eficacia del sistema multi-robot en la generación de mapas 2D.
- Validar los resultados obtenidos en entornos reales mediante la generación de un mapa del entorno del Instituto de Computación (InCo) como caso de estudio.
- Documentar de manera detallada el proceso de desarrollo del sistema multi-robot, incluyendo la descripción de los componentes utilizados, la metodología de implementación, los algoritmos desarrollados y los resultados obtenidos.
- Utilizar técnicas y metodologías modernas de gestión de proyectos que faciliten la planificación y desarrollo de las diferentes tareas. A su vez, utilizar herramientas de control de versiones que permitan paralelizar el trabajo y mantener diferentes versiones durante el avance del proyecto.

1.1.4. Alcance

En esta sección se detallará todo lo que, al momento de comenzar con el proyecto, se definió que estaría tanto dentro como fuera del alcance del proyecto.

Se ensamblaron tres robots utilizando la plataforma Robotito como base y que se equiparon con una placa Odroid y un sensor LiDAR para proporcionar capacidades motoras, de cómputo y sensoriales adecuadas.

Se incluyó dentro del alcance el desarrollo de un sistema de mapeo 2D autónomo y colaborativo en entornos simulados. Se utilizaron los robots ensamblados como plataforma de hardware para realizar pruebas físicas en entornos reales. Se decidió que el sistema debería ser desarrollado utilizando ROS2 como plataforma de software.

Se realizaron simulaciones en entornos estándar. Además, se llevó a cabo el mapeo de un piso del InCo como caso de estudio.

No estuvo dentro del alcance el diseño o desarrollo de nuevos elementos, como el sistema de control de la plataforma motora.

1.1.5. Supuestos

En cuanto a los supuestos del proyecto, se considera que se cuenta con todos los materiales necesarios para el ensamblaje de los robots, incluyendo la plataforma Robotito, las placas Odroid y los sensores LiDAR. Asimismo, se asume que la plataforma robótica utilizada ya tiene resuelto los problemas de control del movimiento.

Se parte del supuesto de que existe un antecedente donde se ha logrado establecer la comunicación entre la placa Odroid y el microcontrolador encargado del control motor.

Por último, se supone que ROS2 es compatible con la plataforma Odroid. Además, se espera que sea posible establecer una comunicación efectiva entre los robots utilizando ROS2 y Data Distribution Service (DDS) como middleware para la comunicación.

1.1.6. Validez de los supuestos

Si bien se logró contar con todos los materiales necesarios para ensamblar los robots, la plataforma motora no contaba con el control del movimiento resuelto sino que se tuvo que implementar parte de la solución. Este trabajo se describe en la Sección 4.

Con respecto a la comunicación entre el Odroid y la plataforma motora, se puede afirmar que es factible aunque no se haya encontrado un antecedente el cual realice una comunicación serial entre el Odroid y el microcontrolador de la plataforma. Se encontraron antecedentes y referencias que realizaban comunicaciones seriales con cada uno y se tomaron como apoyo para realizar las tareas de diseño e implementación de la solución.

El último supuesto resultó ser el más problemático. Ya que por más de que se verificó que ROS2 es compatible con el Odroid, con el middleware de comunicación de ROS2 no se logró establecer una comunicación entre robots tal que permita el despliegue completo de la solución implementada. Este punto es analizado y descrito en la Sección 7.2.5

1.2. Introducción al Documento

El propósito de este documento es brindar al lector una introducción al desarrollo de «**Rogel**», un robot diseñado y desarrollado como parte de este proyecto de grado. Rogel se basa en Robotito [4], una plataforma robótica desarrollada por el grupo MINA [5], que se utiliza como plataforma motora, como se mencionó

previamente. Sin embargo, con el fin de poder utilizar dicha plataforma se realizaron modificaciones tanto en el firmware como en la adición de componentes de hardware necesarios para cumplir con los objetivos del proyecto.

La estructura del documento se divide en los siguientes capítulos:

- Introducción: Se presenta el contexto en el que se desarrolla este trabajo, así como aspectos generales del robot y del tema en cuestión.
- Resumen: Se proporciona un resumen de cada uno de los capítulos del documento.
- Hardware: Se describe cada uno de los componentes físicos que conforman a Rogel. Se detallan los elementos de hardware utilizados, sus características, principios de funcionamiento y las funciones que desempeñan en el sistema. También se presenta la conexión entre cada una de las partes.
- Modelado y Caracterización: En este bloque se presentan los cálculos teóricos que respaldan el modelado del robot. Además, se detallan aspectos prácticos, como el comportamiento real de los componentes y las especificaciones finales del robot.
- Firmware: Esta sección describe el trabajo realizado en el firmware de Robotito para adaptarlo como una plataforma motora que cumple con los requisitos funcionales de Rogel. También se describe el protocolo de comunicación utilizado para la integración con la aplicación principal.
- Software: Aquí se explica el funcionamiento de alto nivel de Rogel, los módulos implementados, las funcionalidades y sus características. Se abordan temas centrales del proyecto, como la navegación, SLAM, la exploración colaborativa entre robots, la división de tareas y la comunicación intra e interrobot.
- Simulación: En este módulo se explica el proceso y trabajo realizado para generar simulaciones del robot y del entorno que representen la realidad. Se presentan resultados de dichas simulaciones para distintos escenarios.
- Resultados: Se describen los experimentos realizados para verificar el correcto funcionamiento del robot, recopilar información sobre sus características y realizar ajustes necesarios. También se presentan las conclusiones particulares de cada una de las prácticas y simulaciones realizadas.
- Conclusiones: Se exponen las conclusiones generales del proyecto y del trabajo realizado. Se proponen aspectos a mejorar, posibles cambios futuros y líneas de trabajo interesantes.

1.3. Antecedentes

En el campo de la robótica, existen numerosos antecedentes que resultan relevantes debido al gran interés y trabajo dedicado a esta área. En esta sección, se

presentarán casos que se aproximan a la temática de investigación. Específicamente, se abordan antecedentes relacionados con robots omnidireccionales, sistemas multi-robot, el Turtlebot [6] y Robotito [4].

1.3.1. Rovert

Como antecedente de un robot omnidireccional, se destaca el trabajo realizado en el proyecto de grado por Santiago Bernheim, Agustín Costa y Andrés De Luca, titulado «Robot Autónomo Móvil Terrestre con Énfasis en SLAM y Fusión Sensorial» [7]. En este proyecto, se construyó un robot llamado Rovert y se implementó un sistema capaz de realizar SLAM para entornos desconocidos. El robot construido es un carro de cuatro ruedas omnidireccionales equipado con sensores y actuadores para interactuar con su entorno (ver Figura 1.1).

Figura 1.1: Robot Rovert. Extraída de [7].

1.3.2. Exploración colaborativa

En relación a los sistemas multi-robot enfocados en la exploración colaborativa, se hace referencia al trabajo realizado por María Victoria Díaz y Sergio Robaudo en su proyecto titulado «Exploración Colaborativa con Butiá» [8]. En este trabajo, se introduce la temática de la exploración colaborativa a través de simulaciones y se plantean las bases para su implementación en el mundo real. Una de las principales diferencias entre este proyecto y el nuestro es que su implementación fue prácticamente simulada realizando una prueba de concepto en un robot físico y personalizado, sin utilizar entornos de desarrollo conocidos y orientados a la robótica, como ROS.

1.3.3. TurtleBot

TurtleBot [6] es una plataforma robótica diseñada para la investigación y la educación en robótica. Este es el único antecedente consultado que proviene de una fuente externa a la Universidad de la República, originalmente fue desarrollado por

Melonee Wise y Tully Foote en Willow Garage y actualmente es mantenido por Open Robotics. El proyecto combina hardware y software de código abierto. A su vez, este cuenta con versiones físicas como la que se ve en la Figura 1.2 y también con modelos para realizar simulaciones en entornos como Gazebo [9].

Entre los usos se encuentran algunas de las temáticas que en este proyecto se trabajarán como lo son la navegación autónoma [10] y la exploración colaborativa [11].

Figura 1.2: Turtlebot3 Waffle. Extraída de [6].

1.3.4. Robotito

Finalmente se introduce a Robotito [4], ver Figura 1.3, un robot de bajo costo construido para trabajar con niños de entre 4 y 7 años y que, por el desarrollo que tuvo la plataforma, nos permite tener varios robots iguales a disposición para realizar las experimentaciones. Originalmente este robot estaba programado para reaccionar a su entorno y ofrecía a los desarrolladores tanto la posibilidad de modificar estas reacciones como de desarrollar sus propios scripts para acceder a las funcionalidades del hardware. El robot cuenta con LEDs (Light Emitting Diode), el sistema motor, sensores de distancia y de luz.

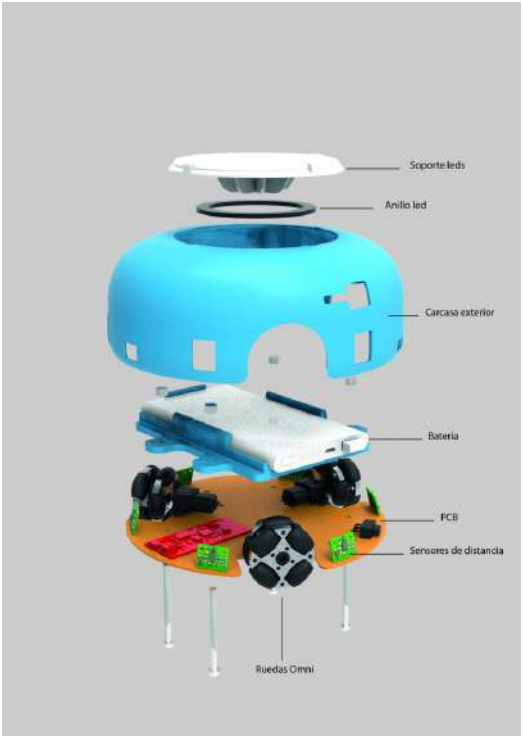


Figura 1.3: Robotito. Extraído de [4].

El firmware desarrollado por el MINA que permite controlar el sistema motor se trata de una solución sencilla, escalable y que permite reportar al detalle su funcionamiento y los recursos disponibles para otro tipo de tareas. Dado que en nuestro caso solo se trabajará con el sistema motor de Robotito, es decir, no se utilizarán los sensores de distancia ni las luces LED, se decidió recurrir a dicha implementación como punto de partida.

Capítulo 2

Hardware

En esta sección se detallarán los elementos de hardware utilizados para la construcción del robot. Se introducirán los tres grupos distintos que se muestran en la Figura 2.1. En primer lugar, se abordará la plataforma motora, compuesta principalmente por las ruedas, los motores, los encoders y los controladores de los motores. A continuación, se describirán los componentes utilizados para el cómputo y sensado. Por último, se proporcionarán detalles sobre la fuente de energía utilizada.

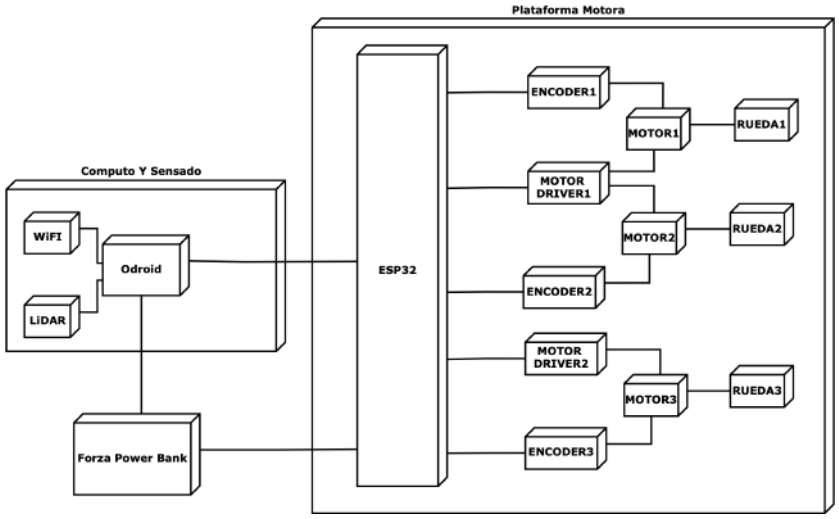


Figura 2.1: Descomposición del sistema en diferentes grupos.

Como se puede ver en la figura Rogel se puede dividir en tres sub-sistemas:

- El encargado del control de los motores y el calculo de odometría, la plataforma motora.
- El bloque que se encarga de la lógica y tareas de alto nivel que se representa como cómputo y sensado.

- Finalmente la fuente de energía que se representa con el modelo de la batería utilizada.

2.1. Plataforma Móvil basada en Robotito

En la presente sección se introduce la plataforma móvil utilizada en Rogel basada en el proyecto Robotito. En las siguientes sub-secciones se describen los componentes de hardware utilizados en la plataforma partiendo desde el PCB (Printed Circuit Board) hasta el microcontrolador. Además, se incluyen en algunos casos, descripciones del comportamiento físico del componente.

2.1.1. PCB y chasis

La plataforma móvil se encuentra montada sobre un PCB donde se conectan todos los elementos electrónicos que componen a Robotito. El propio PCB oficia de chasis de superficie circular de 15 cm de diámetro a la cual se le realizaron 3 muescas que distan entre ellas 120°para colocar las ruedas. Ver Figura 2.2.

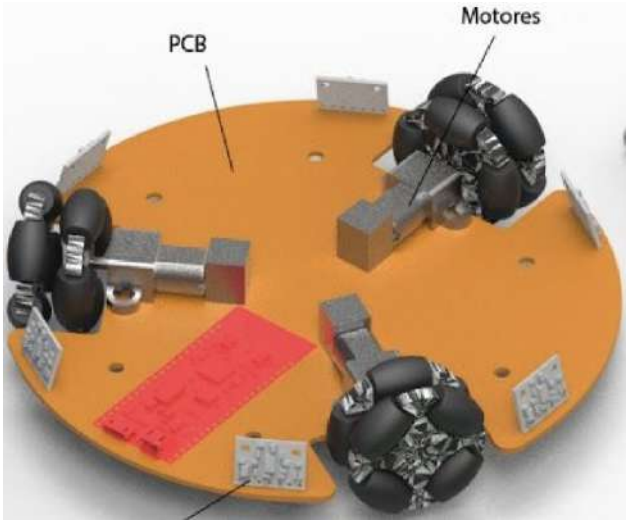


Figura 2.2: Diagrama 3D de la base de Robotito. Extraído de [4].

Los componentes que conforman a Robotito se disponen siguiendo el esquemático de la Figura 2.3. Los principales componentes son las ruedas acopladas a los motores junto con sus respectivos encoders, los drivers de los motores y el microcontrolador.

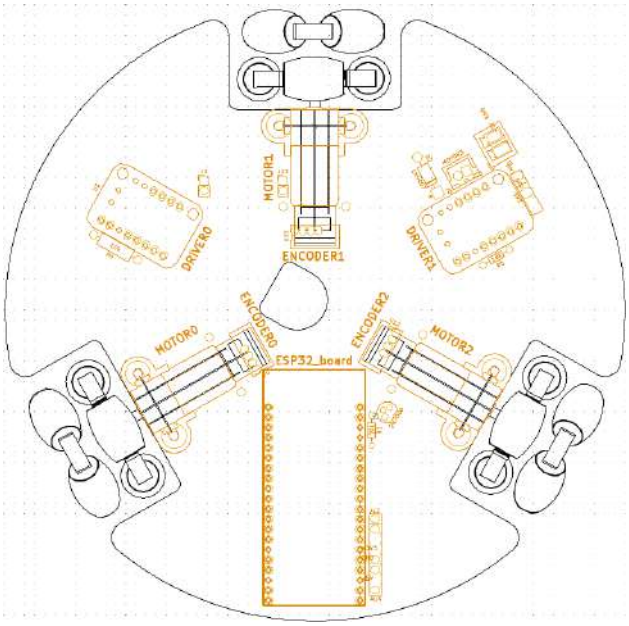


Figura 2.3: Esquemático de la base de Robotito. Extraído de [12].

2.1.2. Ruedas

Se utilizaron ruedas omnidireccionales de 19 mm de radio de aluminio con rodillos de goma [13]. Una rueda se dice omnidireccional cuando permite el movimiento en la dirección de rodamiento natural y también en la dirección paralela al eje de la rueda. Las ruedas que se utilizaron constan de 2 discos de aluminio dispuestos en paralelo, entre los cuales se intercalan 8 rodillos de goma como se muestra en la Figura 2.4.

Figura 2.4: Imagen de la rueda omnidireccional utilizada. Extraído de [13].

Esta configuración de rodillos permite que la rueda minimice la fricción al ser

sometida a una fuerza paralela al eje. Además, la combinación de 3 ruedas dispuestas según se muestra en la Figura 2.2 le da al robot la característica de holonomía. Esto significa que, estando en una posición determinada, el robot tiene la capacidad de desplazarse en cualquier dirección mediante una combinación adecuada de las velocidades de los tres motores [14].

La Figura 2.5a muestra un robot con capacidad holonómica dado que en cualquier momento puede desplazarse en cualquier dirección como se justifica en [7], mientras que la Figura 2.5b muestra un robot que carece de esta característica ya que, por ser diferencial, para realizar un movimiento en la dirección normal al giro de las ruedas debe avanzar o girar en su lugar previo a dirigirse a dicho lugar.

(a) Robot holonómico (b) Robot diferencial

Figura 2.5: Robots según su característica de movimiento

Para acoplar las ruedas al eje de los motores se utilizaron los *hubs* de aluminio [15] que se muestran en la Figura 2.6.

Figura 2.6: *Hub* utilizado para acoplar las ruedas al eje del motor. Extraído de [15].

2.1.3. Motores

Los motores utilizados son los motores Micro Metal Gear [16] con una reducción aproximada de 50:1 y eje extendido. Estos motores fueron instalados en el chasis

mediante un anclaje de plástico [17], tal como se muestra en la Figura 2.7. En la Tabla 2.1 se pueden apreciar las especificaciones técnicas de estos motores.

Figura 2.7: Anclajes de los motores. Extraído de [17].

Tamaño	(10x12x26) mm
Peso	9,5 g
Diámetro del eje	3 mm
Relación de engranajes exacta	51,45 : 1
Tensión nominal	6 V
Rango de tensiones de trabajo	$(-6, 6)V$
Corriente de vacío	100 mA
Corriente de arranque	1600 mA
Velocidad en vacío	590 rpm
Torque de arranque	0,106 Nm

Tabla 2.1: Especificaciones de motores trabajando con una alimentación de 6 V

Si bien todas las especificaciones de los motores están basadas en una tensión de alimentación de 6 V, la plataforma motora es alimentada por una batería de 5 V. Igualmente, al ser motores de corriente continua o Direct Current (DC), las características cuando se lo alimenta con una tensión distinta se pueden obtener como una función de los valores que se muestran en la Tabla 2.1. La justificación teórica y más detallada se puede conocer en [18] . Cualitativamente lo que sucede es que cada motor tendrá menor torque y velocidad y mayor consumo.

2.1.4. Encoders

En la extensión del eje de cada motor, se colocó un Encoder Magnético [19] con dos sensores de efecto Hall. Este componente se agrega para registrar el giro de cada una de las ruedas y así obtener una medida del desplazamiento del robot. Se

lo llama sensor de efecto Hall pues se vincula al fenómeno físico que se utiliza para determinar el movimiento. Estos encoders son componentes claves en el diseño y funcionamiento del robot ya que son una fuente de información de desplazamiento que se utiliza para la odometría[1]. Las dimensiones del encoder se ilustran en la Figura 2.8, estas dimensiones son compatibles con las del eje extendido de cada motor.

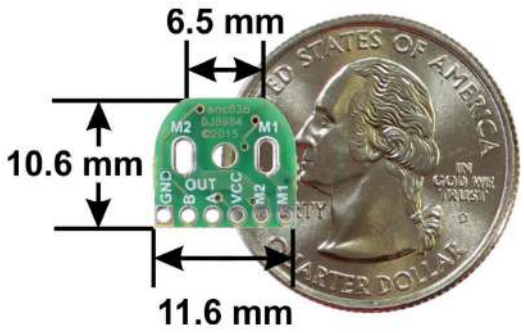


Figura 2.8: Tamaño del encoder. Extraído de [19].

Un encoder magnético como el utilizado en este proyecto se comporta como se muestra en la Figura 2.9. Como se ve en la figura, cuando el encoder gira, las dos señales A y B varían su valor entre 0 y 1 dependiendo del estado relativo entre los 6 imanes distribuidos a lo largo de la circunferencia y los dos lectores.

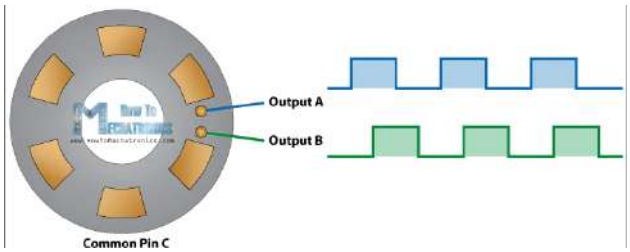


Figura 2.9: Funcionamiento eléctrico de un encoder. Extraído de [20].

El encoder utilizado tiene la capacidad de detectar hasta 12 pulsos por cada vuelta de eje del motor al igual que el del ejemplo de la figura anterior. Debido que el encoder posee una resolución (σ) de 12 cuentas, que las ruedas utilizadas tienen un radio (r) de 1,9 cm y que la reducción teórica del eje del motor a las ruedas (κ) es 50:1, se obtiene una resolución teórica (Δx) para la medida del desplazamiento de una rueda a partir de la medida del encoder según la ecuación 2.1.

$$\Delta x = \frac{2 * \pi * r}{\sigma * \kappa} = 0,019\ cm \tag{2.1}$$

[1]El concepto de odometría se explica en detalle en la Sección 4.2.1.4

2.1.5. Controladores

Para controlar el sentido del giro de los motores se utilizaron dos controladores DRV8833 [21], tal como se muestra en la Figura 2.10. Estos controladores cuentan con dos puentes H [22] en paralelo cada uno, lo que implica que el robot dispone de un puente H sin utilizar. Cada puente H se controla mediante dos entradas digitales, lo que permite que las ruedas puedan girar en sentido horario o antihorario. Además, si esto se combina con el uso de pines PWM (Pulse Width Modulation) del microcontrolador para controlar la velocidad del motor, se obtiene un motor al cual se le puede controlar tanto el sentido de giro como la velocidad. Este tipo de pines y forma de modulación se describen en 2.1.6.

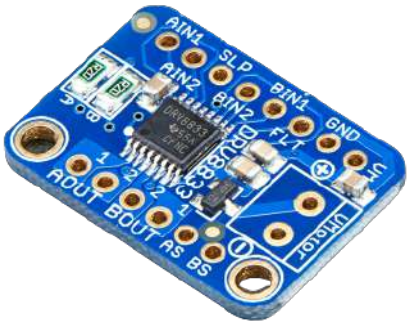


Figura 2.10: Driver para motores DRV8833. Extraído de [21].

2.1.6. Microcontrolador

El control de movimiento y la comunicación entre la plataforma motora y la SBC (Single Board Computer) se realizan mediante el uso de un microcontrolador SparkFun ESP32 Thing, ver Figura 2.11. Este microcontrolador, diseñado principalmente para aplicaciones de Internet of Things (IoT), ofrece una amplia gama de capacidades que proporcionan al robot una gran versatilidad. Es un módulo que cuenta con características necesarias como para poder cumplir con las tareas que se buscan realizar. El microcontrolador está equipado con hardware que permite establecer comunicación Serial para generar un canal de comunicación con el Odroid, trabajar con pines PWM para poder fijar la velocidad de los motores y también permite trabajar con frecuencias de reloj lo suficientemente altas como para cumplir con las tareas de comunicación, cómputo, sensado y actuación en tiempo y forma. Para obtener más información acerca de este microcontrolador, se puede consultar [23], las especificaciones se detallan en la Tabla 2.2 y su hoja de datos está disponible en [24].

Figura 2.11: Microcontrolador SparkFun ESP32 Thing. Extraído de [23].

Almacenamiento	4 MB de flash
Frecuencia de reloj	Hasta 240 MHz
Pines	28 GPIO
Pines PWM	16
RAM	520 kB interna
Tensión de operación	3.0 V a 3.6 V

Tabla 2.2: Especificaciones del SparkFun ESP32 Thing

Como se puede ver en la tabla, el microcontrolador posee 16 pines con funcionalidad PWM, cualquiera de los cuales permite controlar la velocidad de los motores [25]. La modulación por ancho de pulso es una técnica de control que se utiliza entre otras cosas para variar la velocidad de los motores DC. Consiste en conmutar el estado de un pin que alimenta o maneja al motor. El período de la señal y el ciclo de trabajo se pueden ajustar para controlar la velocidad del motor. En la Figura 2.12 se observa un ejemplo de señal PWM donde se muestra tanto el período de pulso (T) como el ciclo de trabajo (D).

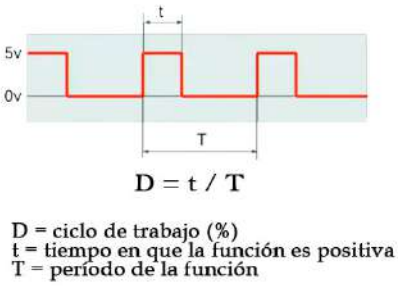


Figura 2.12: Modulación por ancho de pulso.

La velocidad de un motor está directamente relacionada con el ciclo de trabajo de la señal que lo alimenta, es por esto que se utiliza PWM para controlar la

velocidad de motores DC. Esta relación entre la velocidad y el ciclo de trabajo se debe a que al variar el ciclo de trabajo de la señal de control, se puede variar la corriente media que se suministra, lo que permite controlar la velocidad del motor. Esto se debe a que los motores DC, debido a aspectos físicos y constructivos, ofician de filtro pasa bajos de la señal de control por la presencia de inductancias y resistencias inherentes al componente y principalmente por la inercia mecánica del motor. Con este filtrado se obtiene la componente de continua de la señal de alimentación del motor que es quien determina la velocidad de giro. De esta forma, al aumentar el ciclo de trabajo se tiene un mayor nivel de continua mientras que al disminuirlo se tiene un menor nivel de continua, reflejándose en un aumento y disminución de la velocidad del motor respectivamente.

2.1.7. Ensamblaje

En esta sección se detalla cómo es el ensamblaje de cada uno de los componentes en el PCB de la plataforma. En la Figura 2.13 se pueden ver cada una de las partes mencionadas y los componentes que las conectan.

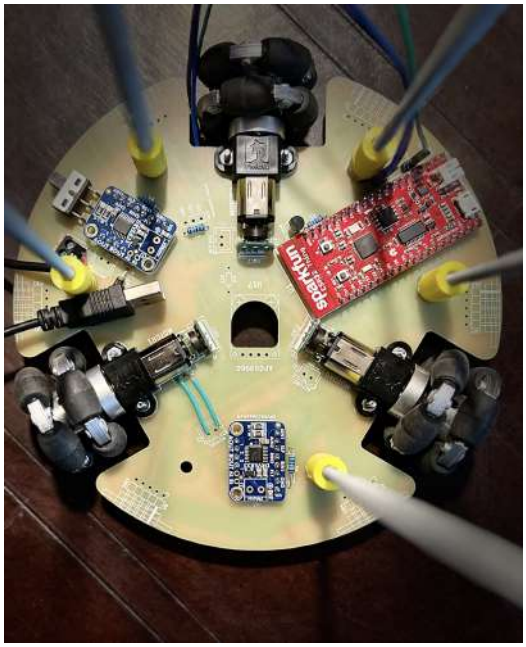


Figura 2.13: Plataforma motora con sus pilares para permitir la adición de nuevos pisos.

Como se ve en la figura, todos los componentes se sueldan al PCB o chasis, mientras que los motores además se fijan con el anclaje de plástico. Por su parte, las ruedas se fijan al eje del motor. En la misma figura se muestra cómo se añaden al chasis los elementos estructurales que permiten luego la integración de los demás niveles donde se encuentran los restantes componentes de hardware que se detallarán en la Sección 2.2.

La integración de todos estos elementos de hardware es fundamental para dar forma al robot y permitir su funcionamiento coordinado. El chasis y los elementos estructurales garantizan la correcta disposición y sujeción de cada componente, asegurando un ensamblaje robusto.

Además, la forma de ensamblaje dio lugar al nombre del robot «Rogel». Se le determinó este nombre debido al ensamblaje en distintos pisos que el robot tiene dándole un aspecto similar a un postre uruguayo llamado rogel, que consiste de una torre con varias capas de masa apilada. Es curioso mencionar que el llamar a los robots siguiendo nombres de comida es algo que se realiza en el mundo de la robótica, por ejemplo el Turtlebot posee modelos como el Burger [26] o el Waffle [27].

2.2. Cómputo y Sensado

En esta sección, se detallarán las características del hardware relacionado con el cómputo y el sensado de Rogel. Es importante destacar que los componentes introducidos en esta sección no formaban parte originalmente de Robotito. El principal objetivo de estos componentes es proporcionar capacidades sensoriales y el nivel de cómputo necesario para llevar a cabo las tareas de exploración autónoma y colaborativa.

En la Sección 2.2.1, se introducirá la plataforma de cómputo utilizada, el Odroid N2+. Esta SBC cumple el rol de unidad central de procesamiento de Rogel, esta será la encargada de realizar todo el cómputo de algoritmos y procesamiento necesario para llevar a cabo las funcionalidades introducidas en la Sección 1.

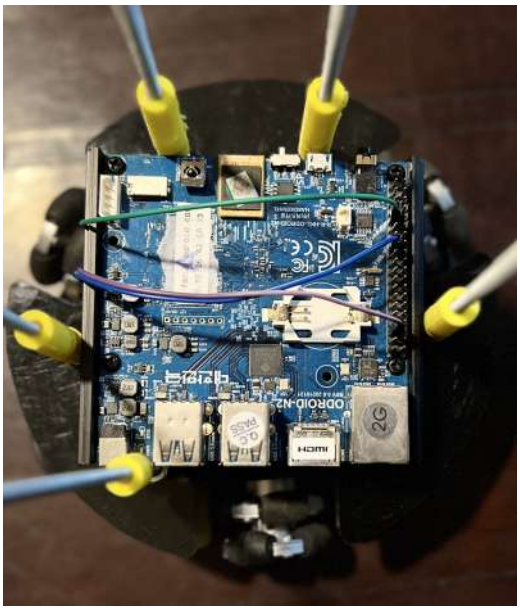
En la Sección 2.2.2, se introducirá el LiDAR utilizado en el proyecto. Este sensor permite obtener información del entorno y es esencial para la generación de mapas. Los detalles técnicos y las características del LiDAR se describirán en dicha sección.

2.2.1. Odroid

Con el objetivo de ejecutar software con altos requerimientos de hardware que sería imposible de ejecutar en un microcontrolador, se utilizó una SBC que se comunica a través de una conexión serial con el microcontrolador, mediante un protocolo que se describe en la Sección 4.2.1.5. Además, otro motivo por el cual se utilizó esta arquitectura en la que se trabaja en dos placas, es que diferentes funcionalidades se ejecutan a distintas frecuencias como el control de los motores, el SLAM o la comunicación, y que de trabajar en una sola placa resultaría más complejo de resolver adecuadamente. A su vez, separar el control de los motores es una forma de trabajo comúnmente utilizada para evitar que los consumos excesivos o picos de corriente afecten al centro de cómputo principal. La SBC seleccionada para este proyecto es el Odroid N2+, el cual se encuentra disponible en el laboratorio del MINA y cuenta con los recursos adecuados (ver Tabla 2.3) para instalar Ubuntu y ejecutar ROS2. El uso de ROS2 permite aprovechar los paquetes existentes y desarrollar nuestros propios algoritmos y técnicas para cumplir

con las tareas necesarias para lograr los objetivos planteados. En la Figura 2.14a se muestra el Odroid N2+, mientras que en la Figura 2.14b se muestra la SBC acoplada a Robotito.

(a) Odroid N2+. Extraído de [28].



(b) Odroid ensamblado a Rogel.

Si bien previo al desarrollo de los componentes de software es difícil predecir las características mínimas necesarias para llevar a cabo el procesamiento requerido por la aplicación a desarrollar, se cree que con las características que se mencionan en la Tabla 2.3 y otras en [28] además de ejemplos de uso más exigentes como en [29], se puede esperar que se cumpla con los niveles mínimos de cómputo requeridos.

Procesadores	Quad-core Cortex-A73 y Dual-core Cortex-A53
Memoria	2 GiB
Almacenamiento	eMMC o microSD
Puertos I/O	4 x USB 3.0, 1 x UART
Alimentación	12 V / 2 A

Tabla 2.3: Especificaciones del Odroid

Por otra parte, para poder contar con comunicación de forma inalámbrica entre los robots y entre cada robot y nuestras computadoras, se utilizó un *dongle* Wi-Fi conectado a uno de los puertos USB (Universal Serial Bus) de cada Odroid. Esto permitió que cada robot se conecte a una red local y los mensajes que desde él se transmitan sean recibidos por todos los participantes de la red.

2.2.2. LiDAR

Para obtener información de distancia y utilizarla para evitar obstáculos y construir mapas, se utiliza lo que se conoce como LiDAR. Un LiDAR es un sistema óptico utilizado para determinar la distancia de objetos remotos. El principio de funcionamiento de este dispositivo se basa en el uso de pulsos láser para medir el tiempo que tarda la luz en viajar desde el emisor hasta el objeto y volver al receptor.

Un LiDAR se compone de: un láser, un escáner y un receptor de luz. El láser emite pulsos de luz, estos pulsos de luz se dirigen hacia el objeto de interés mediante un escáner que puede moverse. Cuando los pulsos alcanzan un objeto, parte de la luz es reflejada de regreso hacia el escáner. El receptor detecta estos retornos y mide el tiempo que tarda en recibirlos (Δt). Utilizando la velocidad de la luz como referencia (v_{luz}) se puede obtener la distancia del objeto (Δx) según la ecuación 2.2 (distancia = velocidad x tiempo).

$$\Delta x = \frac{v_{luz} * \Delta t}{2} \tag{2.2}$$

En este proyecto se trabajó con tres RPLidar A1 [30] debido a que eran los que se encontraban disponibles en el Instituto de Computación. Estos sensores permiten medir distancias en un rango de 360°con las características que se detallan en la Tabla 2.4.

Distancia máxima (m)	12
Distancia mínima (m)	0.15
Precisión	1 % de la medida
Frecuencia de muestreo (kHz)	8
Resolución angular (°)	1

Tabla 2.4: Especificaciones del RPLidar A1.

En la Figura 2.15 se muestra uno de los sensores acoplado al robot.

Figura 2.15: LiDAR ensamblado a Rogel.

Como se muestra en la figura, debido a la forma de ensamblar los LiDARs se encuentran ubicados a una altura de 16.5 cm respecto al suelo. Por lo tanto, al construir mapas basados en las mediciones de estos sensores, es importante tener en cuenta esta altura. El mapa resultante representará un plano horizontal elevado 16.5 cm con respecto al suelo. Es importante destacar que cualquier obstáculo que no se encuentre a esta altura no será detectado por el robot.

Estos obstáculos podrían detectarse si se utilizaran otro tipo de sensores. Por ejemplo, el modelo original de Robotito contaba con un anillo de sensores de distancia que se podrían utilizar para este propósito. Sin embargo, el manejo y uso de estos sensores se dejó fuera del alcance del proyecto por simplicidad por lo que no se trabajó con ellos en esta implementación.

2.3. Gestión de energía

Para satisfacer los requerimientos de alimentación del Odroid, que opera a 12 V, y de la plataforma móvil, que requiere 5 V provenientes de un puerto USB, se decidió utilizar el *power bank* de Forza [31] que provee ambas tensiones de un mismo banco de baterías, como se muestra en la Figura 2.16.

Figura 2.16: Batería utilizada. Extraído de [31].

La elección de este *power bank* se basó en su capacidad para proporcionar los dos niveles de tensión necesarios para alimentar el robot en su totalidad con una sola batería. Se consideró que debido a los grandes picos de corriente generados por los motores no sería buena idea alimentar la plataforma motora directamente de algún puerto USB del Odroid. Las características más relevantes del *power bank* se detallan en la Tabla 2.5. Otra observación importante es que, como se muestra en la tabla, su capacidad es lo suficientemente alta como para garantizar una operación prolongada sin interrupciones. Más detalles sobre el consumo y autonomía de Rogel se exponen en la Sección 3.3.5.

Voltajes de salida (V)	5/9/12
Capacidad (W)	14
Corriente de salida nominal (A)	1
Protección contra sobre corriente (A)	1.2
Peso (kg)	0.4

Tabla 2.5: Especificaciones del *power bank* Forza

Además de cumplir con los requerimientos de consumo, de potencia necesaria y de tensiones de entrega. Esta fuente de energía posee la ventaja de centralizar la alimentación de todo el sistema desde una mima batería. Esta característica simplifica tanto aspectos mecánicos, como eléctricos y también de funcionamiento. Por ejemplo, como aspecto mecánico se tiene una única batería que acoplar, por lo que es mas sencillo distribuir el peso de manera uniforme. Con respecto a aspectos eléctricos, al estar alimentado de una sola batería se posee una misma tierra común en todo el sistema y se evitan posibles problemas eléctricos. Finalmente, en cuanto a la logística, el tener una única batería simplifica el manejo del usuario de la carga de la batería.

2.4. Conexionado

En esta sección se describen detalladamente las conexiones entre los diversos componentes de Rogel. El diagrama de conexiones se presenta en la Figura 2.17, donde se representan tanto las conexiones de alimentación como las señales de datos.

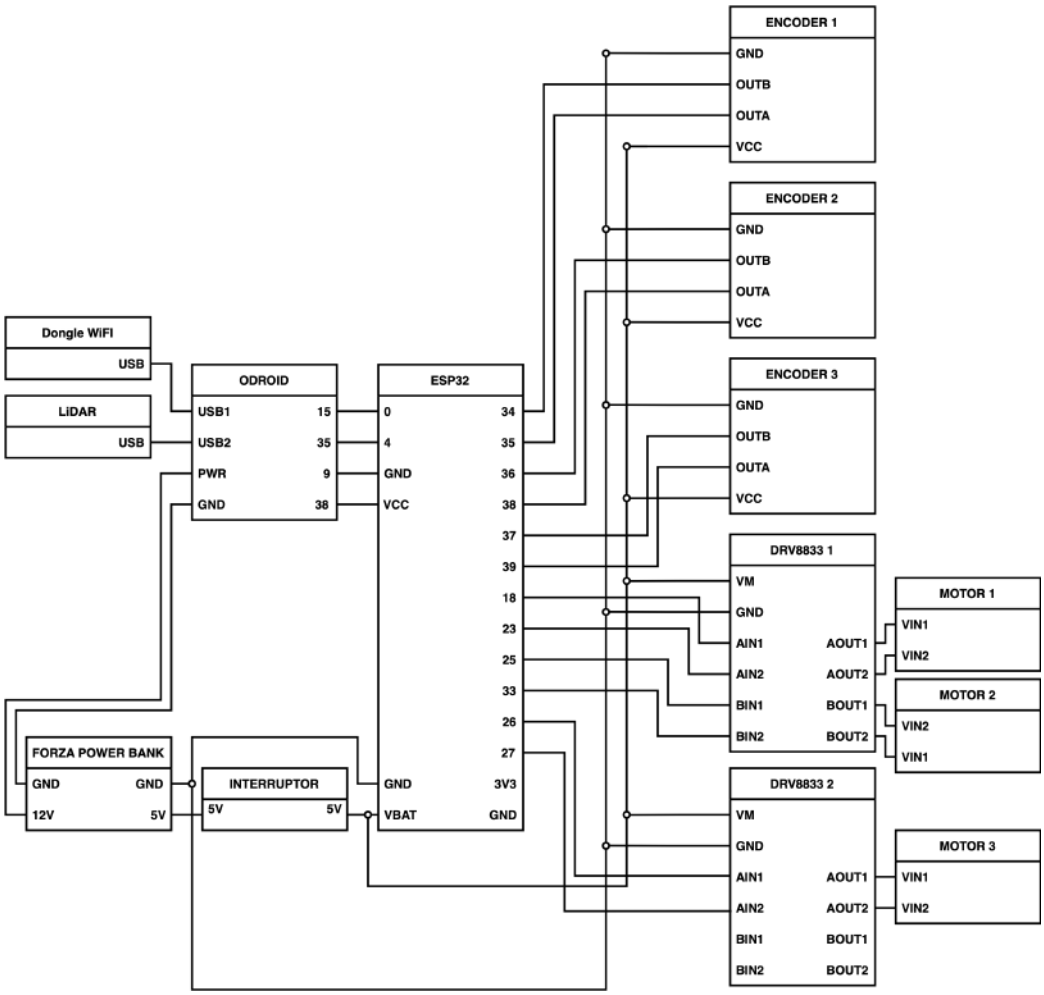


Figura 2.17: Diagrama de conexión de Rogel.

Como se puede observar en el esquema de la Figura 2.17 y como se mencionó previamente en la Sección 2.3, la alimentación del robot proviene de una única batería que cuenta con dos salidas independientes. Una salida de 5 V alimenta la plataforma motora a través de un puerto USB que se puede habilitar o deshabilitar mediante el uso de un interruptor. La otra salida utilizada es la de 12 V, que alimenta el Odroid mediante un conector jack de 5.5x2.5 mm.

En el esquema también se muestra la comunicación UART (Universal Asynchronous Receiver/Transmitter) entre la plataforma motora y el Odroid. Esta comunicación se da a través de los pines {0, 4, GND, VCC} del ESP y los pines

{15, 35, 9, 38} del Odroid. Los datos que se transmiten del ESP al Odroid son enviados por la línea que conecta el pin 0 del ESP con el 15 del Odroid, mientras que los datos desde el Odroid al ESP son enviados por la línea que conecta el 4 del ESP y el 35 del Odroid. Además, para garantizar que los niveles lógicos del ESP y Odroid sean los mismos, se conectan el VCC del ESP al pin 38 del Odroid que es el nivel lógico alto y el pin GND del ESP al 9 del Odroid que es el nivel lógico bajo del Odroid. Como se ve, no es necesario utilizar un adaptador lógico, ya que los niveles lógicos de ambos componentes coinciden.

Los pines asignados para la lectura de los encoders son los pines {34, 35, 36, 37, 38, 39}, los cuales son GPIO (General Purpose Input/Output) genéricos. Las conexiones de los drivers de los motores se realizan utilizando los pines que son capaces de realizar modulación por ancho de pulso (PWM), lo que permite regular la velocidad de los motores, por lo tanto se utilizaron los pines {18, 23, 25, 26, 27, 33}. Además, las salidas de los drivers se conectan directamente a los motores para controlar su movimiento.

Por otro lado, el LiDAR se alimenta a través del puerto USB del Odroid, y también se utilizan las conexiones USB para enviar los datos de distancia desde el LiDAR al Odroid.

Finalmente, el *dongle* WiFi se conecta a un puerto USB del Odroid.

Capítulo 3

Modelado y Caracterización

El presente capítulo tiene como objetivo detallar tanto el proceso de modelado físico-matemático del robot utilizado en el proyecto como la caracterización física de sus componentes y su comportamiento.

El modelado físico-matemático del robot es un proceso fundamental que permite describir su comportamiento y movimiento mediante ecuaciones matemáticas y modelos físicos. Estas ecuaciones se reflejan en el control del robot y son la base para todos los algoritmos que se describen en el Capítulo 4.

Por otro lado, la caracterización física de los componentes del robot juega un papel esencial al transformar los resultados y especificaciones teóricas en valores concretos y medibles. Este proceso es de vital importancia, ya que permite desarrollar módulos de control teniendo en cuenta el comportamiento real de cada componente del robot. De esta manera, se garantiza que el robot cuenta con una base sólida para llevar a cabo las funciones de alto nivel que se detallan en el Capítulo 5.

3.1. Modelado del robot

En esta sección se abordará el proceso de modelado del robot siguiendo el enfoque de [32]. El objetivo principal es desarrollar un modelo que describa el comportamiento del robot. Es importante mencionar que en todo momento se busca utilizar el modelo más sencillo posible del robot que aún sea capaz de aproximar adecuadamente el comportamiento real del mismo. Esto permite obtener un modelo práctico y manejable, sin sacrificar la precisión y la validez de las aproximaciones realizadas.

En el proceso de modelado se deben considerar distintos factores como las dimensiones y geometría del robot y las propiedades de los componentes mecánicos y electrónicos. Asimismo, se deben considerar factores como fuerzas, inercias y fricciones que afectan el movimiento del robot.

3.1.1. Sistema de coordenadas

Con el propósito de representar las variables de interés del robot, como su posición y velocidad, de manera sencilla y coherente, se definen los sistemas de coordenadas presentados en la Figura 3.1. En este diagrama se detallan los ejes de referencia asociados a cada rueda (O_{wi}), siendo cada uno de ellos una rotación de 120°con respecto a los demás según el eje $\hat{Z}$, debido a la construcción particular del robot. También se considera el referencial del robot (O_r) y el sistema de coordenadas absoluto (O).

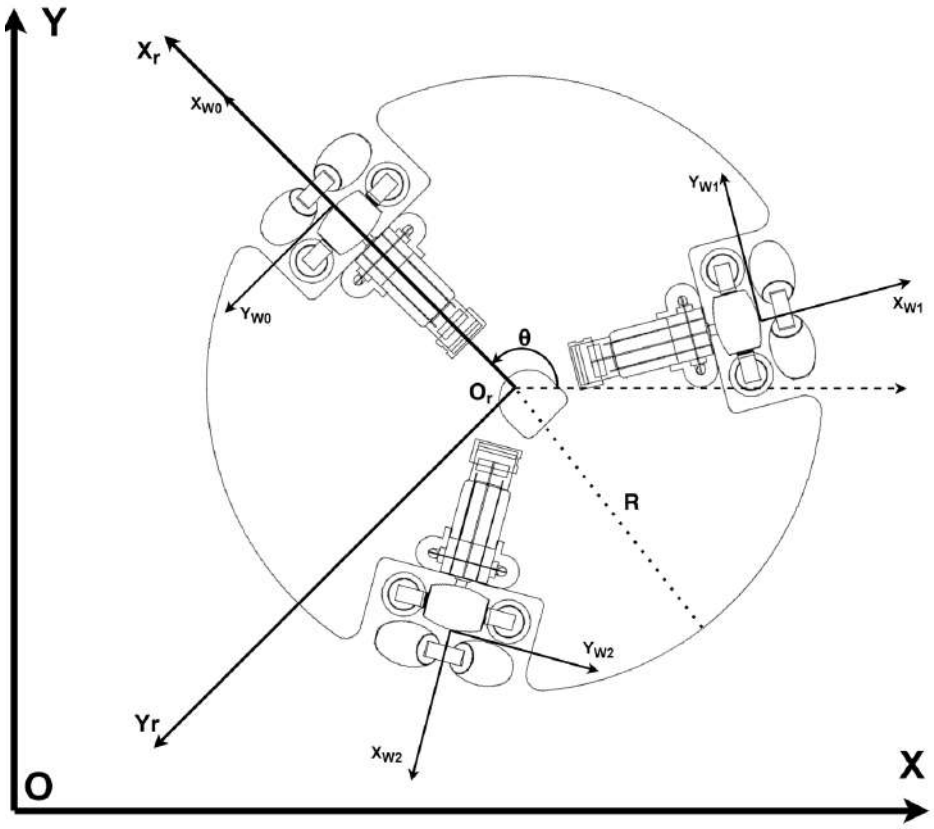


Figura 3.1: Sistemas de coordenadas del robot.

Como se muestra en la Figura 3.1, los ejes de referencia solidarios a cada rueda se encuentran orientados de manera idéntica al sistema de coordenadas asociado al robot. Esto permite expresar la velocidad del robot en cualquiera de estos ejes de coordenadas y también deducir las ecuaciones necesarias para cambiar la referencia de los parámetros de interés.

3.1.2. Holonomía

La holonomía de un sistema mecánico es un concepto que describe la relación entre los grados de libertad controlables y los grados de libertad de movimiento

del sistema. Un robot se considera holónomo si la cantidad de grados de libertad controlables es igual al número de grados de libertad de movimiento, lo que implica que puede desplazarse en todas las direcciones de movimiento sin restricciones y de forma independiente.

Por el contrario, un robot no-holónomo posee menos grados de libertad controlables que grados de libertad de movimiento. Esto significa que el robot no puede moverse en todas las direcciones de movimiento sin llevar a cabo maniobras adicionales, como sucede, por ejemplo, con un auto.

Desde una perspectiva matemática, un sistema se considera holonómico si todas las restricciones de su movimiento a las que está sometido pueden expresarse mediante ecuaciones que dependen únicamente de las variables de posición de los grados de libertad y del tiempo. Estas ecuaciones pueden escribirse de la forma:

$$f(x_1, x_2, ..., x_n, t) = 0 \tag{3.1}$$

Donde $x_1, x_2, ..., x_n$ son las variables de posición del sistema y t es la variable temporal.

Como se demostrará en la Sección 3.1.5, el robot es holónomo ya que las ecuaciones de movimiento quedan completamente descritas por tres variables de posición que se introducen en la Sección 3.1.3 y la variable temporal.

3.1.3. Cinemática

Las ecuaciones que provienen de la cinemática describen el movimiento del robot respecto a parámetros constructivos del mismo. Estas ecuaciones no tienen en cuenta las causas que producen el movimiento, por ejemplo fuerzas o torques.

Previo a comenzar se introduce la nomenclatura que se utilizará para el resto de la sección, estas definiciones siguen lo que se ilustra en la Figura 3.1.

- r: radio de las ruedas
- R: radio del robot
- m: masa del robot
- $\dot{\Phi}_i = \omega_i$: velocidad angular de la rueda i respecto al eje $\hat{x}_{\omega_i}$
- θ : ángulo de rotación entre el referencial absoluto y el referencial del robot.
- x: desplazamiento del robot según el eje $\hat{X}$
- $\dot{x}$: velocidad del robot en el referencial absoluto según el eje $\hat{X}$
- y: desplazamiento del robot según el eje $\hat{Y}$
- $\dot{y}$: velocidad del robot en el referencial absoluto según el eje $\hat{Y}$
- $\dot{\theta}$: velocidad angular del robot en el referencial absoluto según el eje $\hat{Z}$

Con estas definiciones se puede obtener la cinemática del robot, donde en la ecuación 3.2 se muestra cómo se pueden calcular las velocidades angulares de las ruedas a partir de las velocidades del robot en el referencial absoluto.

$$\begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\end{bmatrix} = \frac{1}{r} * T(\theta) \begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix}, \text{ donde } T(\theta) = \begin{bmatrix}-\sin(\theta) & \cos(\theta) & R\\ -\sin(\theta+\frac{2\pi}{3}) & \cos(\theta+\frac{2\pi}{3}) & R\\ -\sin(\theta-\frac{2\pi}{3}) & \cos(\theta-\frac{2\pi}{3}) & R\end{bmatrix} \quad (3.2)$$

es la matriz de rotación entre el marco de referencia absoluto y el solidario al robot.

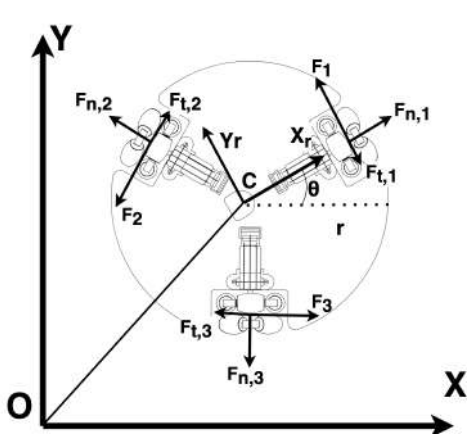
Como la matriz $T(\theta)$ es no-singular para $\theta \in \mathbb{R}$, se puede obtener la matriz inversa $T^{-1}(\theta)$ y con ella la cinemática inversa. Con el uso de esta matriz se pueden calcular las velocidades del robot en el referencial absoluto en función de las velocidades angulares de las ruedas en sus referenciales como se muestra en la ecuación 3.3.

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = r * T^{-1}(\theta) \begin{bmatrix}\omega_1\\ \omega_2\\ \omega_3\end{bmatrix}, \quad (3.3)$$

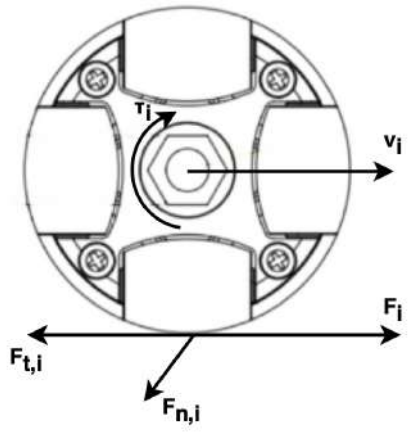
$$\text{donde } T^{-1}(\theta) = \frac{2}{3}\begin{bmatrix}-\sin(\theta) & -\sin(\theta+\frac{2\pi}{3}) & -\sin(\theta-\frac{2\pi}{3})\\ \cos(\theta) & \cos(\theta+\frac{2\pi}{3}) & \cos(\theta+\frac{2\pi}{3})\\ \frac{1}{2R} & \frac{1}{2R} & \frac{1}{2R}\end{bmatrix}$$

3.1.4. Dinámica

A continuación se deducirán las ecuaciones dinámicas que determinan el comportamiento del robot. Éstas describen la relación entre el movimiento del robot y los factores físicos que lo afectan tales como las fuerzas, masa y energía. En las Figuras 3.2a y 3.2b se pueden ver el diagrama de cuerpo libre del robot y un diagrama de la dinámica de la rueda respectivamente.



(a) Diagrama de cuerpo libre del robot.



(b) Diagrama de la dinámica de la rueda.

Las ecuaciones de la dinámica se obtienen de aplicar tanto la segunda ley de Newton como la segunda cardinal utilizando los diagramas presentados anteriormente, resultando las ecuaciones 3.4 y 3.5 respectivamente.

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \frac{1}{m} \sum_{i=1}^{3} \vec{F_i} + \vec{F_{t,i}} + \vec{F_{n,i}} \tag{3.4}$$

$$\ddot{\theta} = \frac{R}{J} \sum_{i=1}^{3} (\vec{F_i} - \vec{F_{t,i}}) \tag{3.5}$$

Donde $\vec{F_i}$, $\vec{F_{t,i}}$, $\vec{F_{n,i}}$ y J son: la fuerza de tracción de la rueda i, la fricción en la dirección longitudinal de la rueda i, la fricción en la dirección normal de la rueda i y el momento de inercia del cuerpo respectivamente. En nuestro caso, $J = \frac{mR^2}{2}$

Se define el vector de variables de estados $C = \begin{bmatrix} x, & y, & \theta \end{bmatrix}^T$, que es también la posición del robot en el sistema de coordenadas global. Si además se definen $F = \begin{bmatrix} F_1, & F_2, & F_3 \end{bmatrix}^T$, $F_t = \begin{bmatrix} F_{t,1}, & F_{t,2}, & F_{t,3} \end{bmatrix}^T$ y $F_n = \begin{bmatrix} F_{n,1}, & F_{n,2}, & F_{n,3} \end{bmatrix}^T$ se obtiene de 3.4 y 3.5 la siguiente ecuación:

$$M\ddot{C} = T(\theta)^T (F - F_t) + N(\theta)^T F_n, \tag{3.6}$$

$$\text{donde } N(\theta) = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ cos(\theta + \frac{2\pi}{3}) & sin(\theta + \frac{2\pi}{3}) & 0 \\ cos(\theta - \frac{2\pi}{3}) & cos(\theta - \frac{2\pi}{3}) & 0 \end{bmatrix} \text{ y } M = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & J \end{bmatrix}$$

Considerando que el peso del robot se distribuye uniformemente en las tres ruedas, se pueden obtener las siguientes dos ecuaciones para las fricciones de rodamiento:

$$\begin{aligned} \|F_{t,i}\| &= \mu_{t,i} \frac{mg}{3} \\ \|F_{n,i}\| &= \mu_{n,i} \frac{mg}{3} \end{aligned} \tag{3.7}$$

Donde $\mu_{t,i}$ y $\mu_{n,i}$ son los coeficientes de rozamiento entre la rueda i y el piso en las direcciones tangenciales y normales respectivamente.

Por otra parte, se puede escribir la fuerza de tracción F_i como se muestra a continuación, donde τ_i es el torque que realiza el motor y r el radio de la rueda. Si se considera un modelo lineal para el torque en función de la tensión de alimentación de la forma: $\tau_i = a_i u_i - b_i \omega_i$, se puede sustituir el torque obteniendo la segunda igualdad presentada en la ecuación 3.8, donde u_i, a_i y b_i son la tensión inducida en el motor y dos constantes de proporcionalidad respectivamente.

$$F_i = \frac{\tau_i}{r} = \frac{a_i u_i - b_i \omega_i}{r} \tag{3.8}$$

De esta forma, uniendo las ecuaciones 3.2, 3.6, 3.7 y 3.8 se obtiene la ecuación que modela la dinámica del robot en forma matricial:

$$\ddot{C} = M^{-1}T(\theta)^T I_\alpha U - M^{-1}T(\theta)^T I_\beta T(\theta)\dot{C} + M^{-1}\{N(\theta)^T F_n - T(\theta)^T F_t\} \quad (3.9)$$

$$\text{Donde: } I_\alpha = \begin{bmatrix} \frac{a_1}{r} & 0 & 0 \\ 0 & \frac{a_2}{r} & 0 \\ 0 & 0 & \frac{a_3}{r} \end{bmatrix}, U = \begin{bmatrix} u_1, & u_2, & u_3 \end{bmatrix}^T, I_\beta = \begin{bmatrix} \frac{b_1}{r^2} & 0 & 0 \\ 0 & \frac{b_2}{r^2} & 0 \\ 0 & 0 & \frac{b_3}{r^2} \end{bmatrix},$$

$$F_n = \frac{mg}{3}\begin{bmatrix} \mu_{n,1} & 0 & 0 \\ 0 & \mu_{n,2} & 0 \\ 0 & 0 & \mu_{n,3} \end{bmatrix} \text{ y } F_t = \frac{mg}{3}\begin{bmatrix} \mu_{t,1} & 0 & 0 \\ 0 & \mu_{t,2} & 0 \\ 0 & 0 & \mu_{t,3} \end{bmatrix}$$

3.1.5. Ecuaciones del Movimiento

Una vez deducidas las ecuaciones de la dinámica y cinemática del robot, es posible obtener las ecuaciones de movimiento que regirán el comportamiento del robot y que serán utilizadas para el control del mismo.

En el caso del robot diseñado, se demostrará que es suficiente con las ecuaciones de la cinemática para el control del mismo. Esto se puede demostrar si el robot cumple con las siguientes condiciones:

- El robot debe moverse sin deslizamiento entre sus ruedas y el piso.
- El tiempo de establecimiento para un cambio de velocidades (escalón) en los motores debe ser despreciable en comparación al cambio en la velocidad.

El cumplimiento de ambas condiciones implica que, en cada momento, se está imponiendo directamente la velocidad de cada rueda, en lugar de una fuerza para alcanzar dicha velocidad. Si bien en la literatura se puede encontrar que utilizar la cinemática como primera aproximación en general es un buen modelo de la realidad, para validar estas hipótesis se realizaron los experimentos detallados en las Secciones 3.3.3.2 y 3.3.3.1.

En conclusión, el modelo para las ecuaciones del movimiento que se considera para el desarrollo de cada uno de los módulos implementados fue el obtenido a partir de la cinemática, es decir, utilizando la ecuación 3.10.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = r * T^{-1}(\theta) \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \quad (3.10)$$

Es importante notar que la ecuación anterior describe la velocidad del robot en un instante dado de tiempo. Esta no es aplicable directamente para obtener la posición del robot, ya que el parámetro θ depende del tiempo y se requiere realizar una integración numérica para calcular su posición. La ecuación 3.11 representa la forma de realizar dicha integración numérica, donde T_s es el tiempo entre el cálculo del término $k-1$ y el k. Se utilizará esta última ecuación tanto para calcular la

posición del robot como para estimar el desplazamiento mediante la odometría y lograr la localización del robot en su entorno.

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_k = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_{k-1} + \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}_k * T_s \tag{3.11}$$

3.2. Control del motor DC

3.2.1. Modelado motor DC

En esta sección se presenta el modelado físico matemático de un motor de corriente continua o motor DC. Este modelo es clave debido a que el control de las velocidades del robot se realizan mediante el control de cada uno de estos motores. Por ende, es necesario conocer las características de la respuesta de estos motores. El modelado se llevó a cabo siguiendo como referencia el trabajo realizado por parte de dos de los integrantes del grupo para una asignatura de la carrera. Como se puede ver en la Figura 3.3 el modelado del sistema se puede dividir en dos dominios, el eléctrico en la parte izquierda del diagrama y el mecánico en el lado derecho.

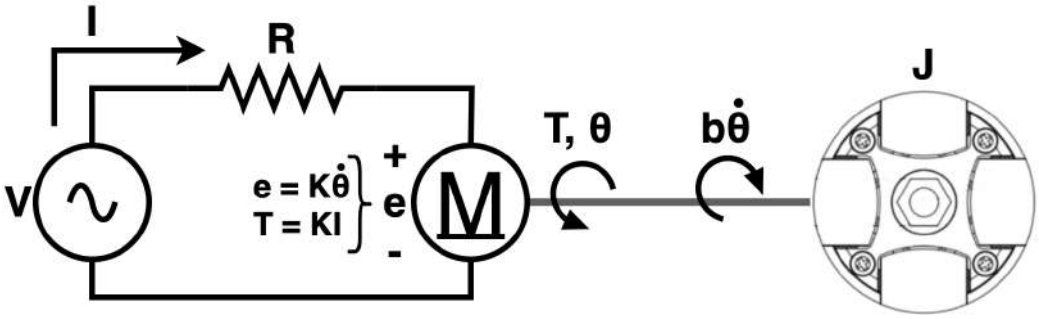


Figura 3.3: Modelo del sistema motor.

Siguiendo el esquema anterior, se definen las variables de interés del sistema:

- V: voltaje en bornes del motor.
- R: resistencia interna del motor.
- e: caída de potencial en el motor.
- I: corriente de excitación del motor.
- T: torque generado por el motor.
- $\dot{\theta}$: velocidad angular del eje del motor.
- J: momento de inercia visto desde el eje motor.

- b: coeficiente de viscosidad en el eje.

Si se aplican la ley de Kirchoff para la malla y la segunda cardinal para el sistema de la Figura 3.3, se obtienen las ecuaciones 3.12 y 3.13 respectivamente.

$$V = RI + e \tag{3.12}$$

$$J\ddot{\theta} + b\dot{\theta} = T \tag{3.13}$$

Además, si se utiliza el modelo para la constante de proporcionalidad del motor (K) expresado en 3.15 y 3.14, se obtienen relaciones entre el torque en el eje del motor y la corriente de excitación del motor en la ecuación 3.15 y entre la caída de potencial del motor y la velocidad angular en el eje del motor en la ecuación 3.14.

$$e = K\dot{\theta} \tag{3.14}$$

$$T = KI \tag{3.15}$$

Si se combinan las ecuaciones 3.12 con 3.14 y 3.13 con 3.15 se obtiene el siguiente sistema de ecuaciones:

$$\begin{cases} V = Ri + K\dot{\theta} \\ J\ddot{\theta} + b\dot{\theta} = KI \end{cases} \tag{3.16}$$

Con el fin de estudiar el sistema, se aplica la transformada de Laplace[1] [33] al sistema anterior obtiene el sistema de ecuaciones 3.17. Al resolver el sistema de ecuaciones en el dominio de Laplace se obtiene la transferencia del sistema $H(s)$ dada por la ecuación 3.18

$$\begin{cases} V(s) = RI(s) + s\theta(s)K \\ s\theta(s)(Js + b) = KI(s) \end{cases} \tag{3.17}$$

$$H(s) = \frac{\dot{\theta}(s)}{V(s)} = \frac{s\theta(s)}{V(s)} = \frac{K}{R(Js + b) + K^2} \tag{3.18}$$

Es importante notar que el modelo que se obtuvo es el de un sistema de primer orden, el cual en la práctica se ajusta bien a la realidad. Sin embargo, se podría complejizar el modelo considerando efectos de segundo orden como la inductancia en la malla del motor para obtener un modelo más representativo de la realidad. Pero como ya se mencionó al comienzo de este capítulo, el objetivo no es modelar de manera perfecta cada fenómeno sino que utilizar el modelo más simple que permita representar matemáticamente un sistemas físico, y por ello se decide trabajar con el modelo de primer orden.

[1] La Transformada de Laplace es una técnica matemática que se emplea para analizar sistemas lineales que evolucionan en el tiempo. Permite transformar ecuaciones diferenciales en el dominio del tiempo (t) en ecuaciones algebraicas en el dominio de la frecuencia compleja (s).

3.2.2. Respuesta al escalón

La respuesta al escalón en sistemas dinámicos es un análisis fundamental para evaluar cómo un sistema responde a una entrada de tipo escalón[2] [34]. Este estudio ofrece una visión profunda de la dinámica y el comportamiento temporal del sistema en diversas disciplinas y aporta información que caracteriza al sistema en general. Dentro de las características más importantes de la respuesta al escalón se tienen las que se muestran en la Figura 3.4 y mencionan a continuación:

- **Sobretiro (M_p):** es la diferencia entre el valor máximo alcanzado por la respuesta y el valor final de estado estacionario. Este puede indicar la capacidad que tiene un sistema para manejar perturbaciones repentinas; además proporciona información sobre la estabilidad del sistema. En sistemas de primer orden como nuestro caso, el sobretiro es cero.

- **Tiempo de pico (t_p):** es el tiempo en el cual se da el sobretiro.

- **Tiempo de Establecimiento (t_s):** es el tiempo que tarda la respuesta en entrar y permanecer dentro de una banda de tolerancia alrededor del valor final de estado estacionario. Indica cuánto tiempo lleva al sistema alcanzar su estado estable después de la aplicación del escalón. Un tiempo de establecimiento más corto define la característica de que un sistema es de respuesta rápida. La banda de tolerancia puede situarse en general entre un 10, 5 o 2 % del valor final, en este proyecto se considera 5 % para dicha banda.

- **Tiempo de Subida (t_r):** es el tiempo que lleva a la respuesta aumentar desde el 10 % hasta el 90 % del valor final de estado estacionario. Representa la rapidez con la que el sistema pasa de valores bajos a valores altos después del cambio en la entrada. Un tiempo de subida más corto indica una respuesta más rápida y un sistema más reactivo a los cambios.

- **Tiempo de Retardo (t_d):** es el intervalo entre la aplicación del escalón y el momento en el cual el sistema alcanza en su salida un 50 % de su valor asintótico final. Representa el tiempo que el sistema necesita para comenzar a reaccionar al cambio en la entrada. Un tiempo de retardo prolongado puede afectar la capacidad del sistema para seguir cambios rápidos en la entrada.

[2]En general se conoce como escalón a la entrada que es cero para todo tiempo menor que cero y uno para todo tiempo mayor o igual a cero

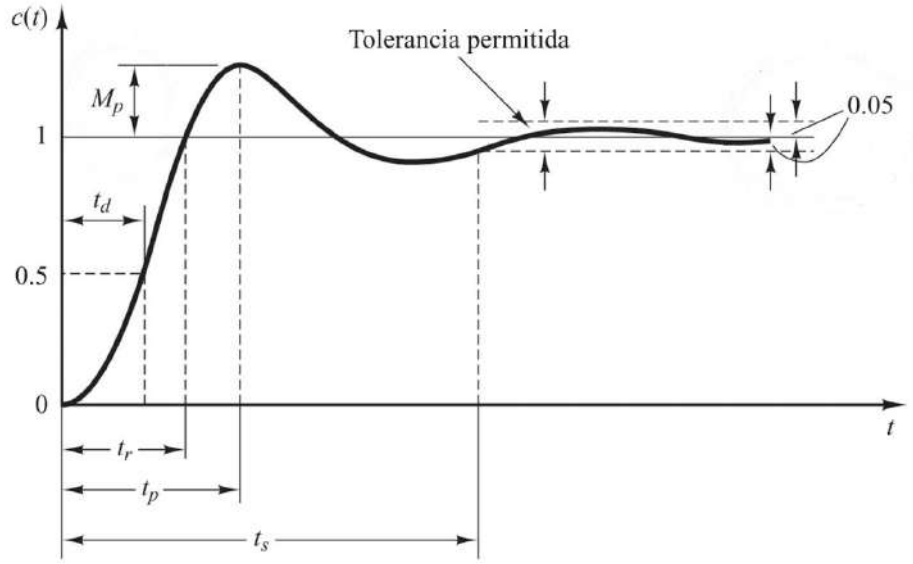


Figura 3.4: Respuesta al escalón de un sistema de orden mayor o igual a 2. Extraído de [35].

En la Figura 3.4 se muestra la respuesta al escalón de un sistema de orden 2. En esta figura se muestran gráficamente las definiciones que se mencionaron en el punteo anterior. Si bien el sistema motor modelado es un sistema de orden 1, es importante introducir estos conceptos para un sistema de orden mayor previo a la descripción del método de control que se menciona en la Sección 3.2.3 con el fin de conocer las precauciones a tener en cuenta para mitigar efectos no deseados: como sobretiro excesivamente alto o un tiempo de asentamiento muy lento, entre otros.

3.2.3. Controlador PID

En esta sección se introduce el método utilizado para controlar la velocidad en los motores. La elección de utilizar un controlador Proporcional Integral Derivativo (PID) se justifica en la simpleza que requiere a la hora de programar y además la experiencia previa que los integrantes del grupo poseen trabajando con controladores de este tipo. A su vez, el uso de PIDs aporta herramientas matemáticas muy poderosas ya que es un estándar en muchas industrias y se obtiene un buen desempeño utilizándolos.

El controlador PID es una técnica de control que se clasifica como un controlador de lazo cerrado. En un sistema de lazo cerrado, la salida se mide y se compara con una señal de referencia o valor deseado. La diferencia entre estas dos señales se utiliza como entrada para el controlador, que genera una señal de control aplicada al sistema para ajustar su salida, tal como se muestra en la Figura 3.5. Este proceso se repite continuamente, permitiendo que el sistema se mantenga en un estado deseado.

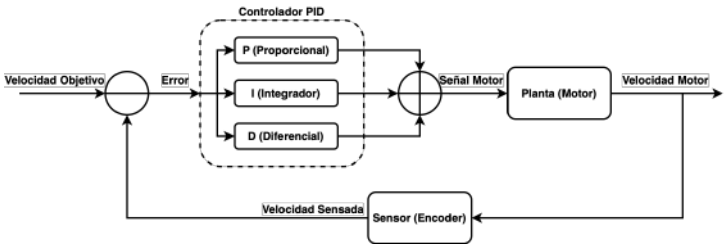


Figura 3.5: Diagrama de bloques del controlador PID.

Por ende podemos escribir la señal motor, es decir la señal que ingresa a la planta u como:

$$u = K_P * e + K_I * \int_t e \; dt + K_D \frac{\partial e}{\partial t}$$

Donde e es la diferencia entre la velocidad sensada y la velocidad objetivo, K_P, K_I y K_D son las constantes de proporcionalidad. La anterior ecuación puede reescribirse en función de variables temporales para los sumandos diferencial e integral como se muestra en la ecuación 3.19. Como se puede ver, este cambio es simplemente una redefinición de las variables de proporcionalidad para poder trabajar de manera más intuitiva con ellas.

$$u = K_P * (e + \frac{1}{T_I} * \int_t e \; dt + T_D \frac{\partial e}{\partial t}) \tag{3.19}$$

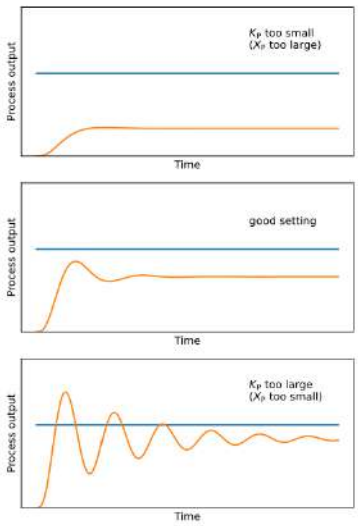
Uno de los aspectos más importantes de los controladores PID es la correcta determinación de cada constante. El valor que se le asigne a cada una de las constantes modifica el comportamiento resultante del sistema. Con el fin de estudiar en detalle y comprender los objetivos de realizar el ajuste de las constantes primero se introducen conceptos necesarios sobre el control de sistemas. Luego se modela el sistema físico e introducen algunas características de dicho modelo y finalmente se introducirá el método utilizado para determinar los valores de las constantes K_P K_I y K_D

Para determinar estas constantes se aplicaron métodos prácticos basados en la experiencia que existe entre los antecedentes cuando se trabaja con controladores PID. Entre ellos, se encuentra por ejemplo el método de Zieglers-Nichols[3]. Sin embargo, en nuestro trabajo se decidió determinar las constantes realizando un trabajo iterativo dado que entre la bibliografía [36] se marca que para sistemas donde se puedan realizar estos métodos se obtienen mejores resultados.

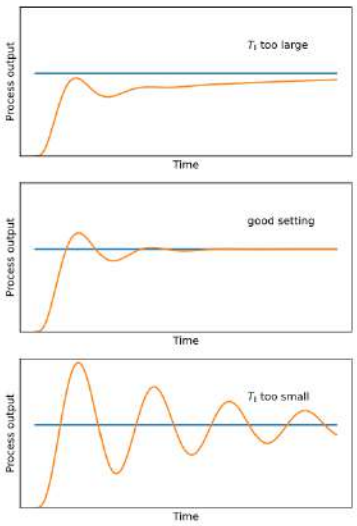
El método que se siguió fue basado en [36]. Allí se mencionan una serie de pasos con los cuales luego de iterar se obtiene un controlador que logra estabilizar el sistema minimizando el error. Este método consiste en trabajar con los diferentes valores de las constantes de manera constructiva, es decir, primero la constante proporcional (Kp), luego la constante integral (Ki) y finalmente la derivativa (Kd)

[3]El método de Zieglers-Nichols es un método heurístico muy conocido para determinar los valores de las constantes de un controlador PID. En general se utiliza como un punto de partida con el cual luego se realiza un ajuste fino con algún otro método iterativo

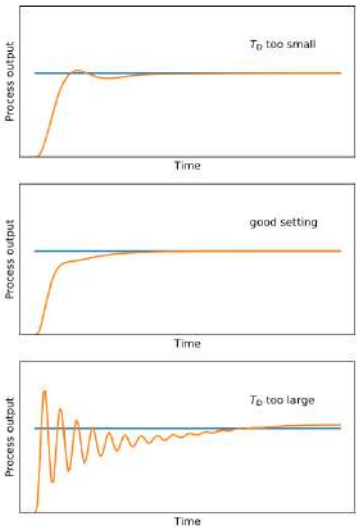
de ser necesaria. Para ello se comienza con un valor pequeño, se registra el comportamiento de la salida y, sabiendo a qué se debe cada salida como se muestra en las Figuras 3.6a, 3.6b y 3.6c, se toman decisiones para las constantes Kp, Ki y Kd respectivamente. Es importante aclarar que para la determinación de cada una de las constantes es necesario mantener fijo el valor de las anteriores.



(a) Determinación de K_P, controlador proporcional.



(b) Con K_P fija se determina T_I, controlador proporcional-integral (PI).



(c) Con K_P y T_I fijas se determina T_D, controlador proporcional-integral-derivativo (PID).

Figura 3.6: Respuesta de la planta en función al controlador y valor de constantes utilizadas. Extraídas de [36].

Como se puede observar en la figura anterior, se comienza trabajando con la constante proporcional siguiendo como guía la Figura 3.6a hasta que se obtiene

una respuesta sin oscilaciones sostenidas pero que presenta sobretiro apreciable y error asintótico[4] también considerable.

Una vez determinada la constante K_P se procede a trabajar sobre la constante T_I. En este paso se busca anular el error asintótico que presentaba el sistema en el paso anterior. Es por ello que se busca obtener una respuesta con sobretiro, pero que en este caso anule el error asintótico de forma rápida sin presentar oscilaciones sostenidas o que perduren con el tiempo.

Finalmente, cuando se tienen tanto K_P como T_I determinados se trabaja con T_D. Con esta constante se busca estabilizar y eliminar las oscilaciones y el sobretiro presente en el paso anterior. Si el valor de T_D es muy pequeño casi no se nota su aporte pero al agrandarlo se podría llegar a desestabilizar el sistema.

En nuestro caso, a la hora de llevar a cabo el control se constató que el beneficio de implementar un controlador PID no era apreciable comparado con el uso de un controlador PI, es decir la respuesta obtenida en la práctica fue la misma. Por ende, se decidió por el controlador PI por ser uno más sencillo y consumir menos recursos computacionales. Sin embargo, se dejó la posibilidad de transformarlo en un controlador PID de manera muy sencilla.

3.3. Caracterización

Esta sección tiene como objetivo caracterizar el comportamiento en la práctica de los componentes desarrollados y utilizados en el proyecto. Para ello se presentan experimentos de diferentes componentes para lograr obtener los resultados buscados.

Se busca comprobar y verificar los modelos físico-matemáticos realizados en las Secciones 3.1 y 3.2. También se busca determinar el comportamiento de los sensores, actuadores y componentes utilizados.

Con el fin de generar un contexto previo a los experimentos se detalla cual es el objetivo y la justificación de porqué se realiza cada una de los experimentos o medidas.

- **Sección 3.3.1:** Se describen las características físicas del robot como los son el peso y las dimensiones.
- **Sección 3.3.2:** Se presenta un experimento para determinar la relación entre una vuelta del eje del motor y la cantidad de tics producidos por el encoder.
- **Sección 3.3.3:** En esta sección se presentan diferentes experimentos para determinar el rango de velocidades del robot. Además, se buscan comprobar las hipótesis presentadas en la sección 3.1.5

[4]El error asintótico representa la diferencia constante que se registra entre el valor objetivo y la respuesta del sistema cuando el tiempo tiende a infinito teóricamente ó en la práctica cuando transcurre un tiempo considerable.

- **Sección 3.3.4:** En esta sección se estudia las características del sensor LiDAR utilizado. Se busca caracterizar el rango mínimo, máximo y desviación de una medida
- **Sección 3.3.5:** En esta sección se presenta un experimento para determinar el consumo del robot.

3.3.1. Características Físicas

En esta sección se documentan las características físicas del robot. En la Tabla 3.1 se pueden ver las más importantes.

Las características físicas del robot son importantes ya que permiten determinar entornos en los cuales el robot no sería capaz de cumplir su función. Por ejemplo, entornos donde existan pozos o baches de tamaño mayor al *clearance*[5], o donde existan obstáculos que no permitan pasar por su altura.

Característica	Valor
Peso	1093 g
Altura máxima	17,5 cm
Altura del LiDAR	16,5 cm
Radio del robot	7,75 cm
Clearance	1 cm

Tabla 3.1: Tabla de características físicas del robot

3.3.2. Relación motor-encoder

Este experimento buscó determinar la relación que existe entre un giro completo de la rueda y la cantidad de pasos que produce el encoder, para conocer más sobre el manejo del encoder referirse a la Sección 4.2.1.1. Si bien ya se contaba con la información teórica de este valor como se presentó en la Sección 2.1, se quiso realizar un experimento para corroborar esta relación. Por lo tanto, se decidió realizar un experimento sencillo con el fin de determinar experimentalmente dicha relación.

El experimento consistió en girar 10 de veces cada rueda y llevar un contador de la cantidad de pasos producidos por el encoder. Con ello, al dividir finalmente la cantidad de pasos entre las 10 vueltas se obtiene la relación buscada, siendo esta la cantidad de pasos por vuelta que se producen al girar una rueda.

El set-up del experimento se muestra en la Figura 3.7, en el se pueden ver la computadora donde se recibe la información de la cantidad de pasos registrados, la plataforma motora a la cual se le hace girar la rueda y la marca en la rueda para registrar tanto el punto de partida como de finalización del giro. Se decidió realizar 10 vueltas ya que se consideró una cantidad suficiente para que las condiciones de borde sean despreciables en el resultado final.

[5]Distancia entre el punto más bajo del robot y el suelo

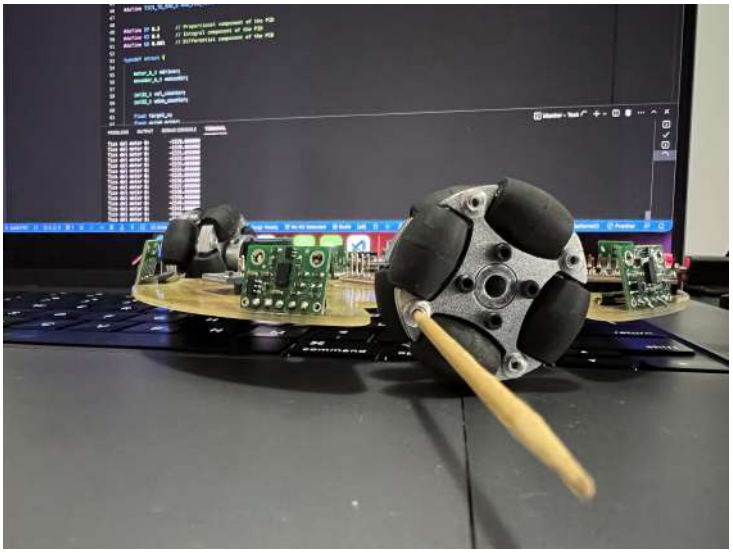


Figura 3.7: Set-up del experimento

Debido a que en un principio no era evidente que este resultado sería igual para los diferentes pares motor-encoder se repitió el mismo experimento para cada uno de los diferentes pares, obteniendo los resultados que se muestran en la Tabla 3.2. La nomenclatura $Rueda_{i,j}$ corresponde a la rueda i del robot j.

$Rueda$	$\frac{Pasos}{10\ vueltas}$	$\frac{Pasa}{vuelta}$
$0,0$	1543	154
$1,0$	1542	154
$2,0$	1543	154
$0,1$	1542	154
$1,1$	1544	154
$2,1$	1543	154
$0,2$	1542	154
$1,2$	1541	154
$2,2$	1542	154

Tabla 3.2: Resultados del experimento realizado para diferentes pares encoder-motor.

Como se puede ver en la tabla anterior, el valor obtenido para los pasos por vuelta para todas las ruedas es de 154. En la Tabla 3.3 se compara el resultado teórico con el experimental observando una diferencia prácticamente despreciable.

$\frac{Pasos}{vuelta}$ Teóricos	$\frac{Pasos}{vuelta}$ Experimental
150	154

Tabla 3.3: Comparación entre los pasos por vuelta teóricos y experimentales.

Como resultado de este experimento se constata entonces que siguiendo las especificaciones de los fabricantes se tiene un error de menos de un 5 %, igualmente se decidió trabajar con el resultado obtenido en la práctica debido a que para todos los pares motor-encoder se registró el mismo valor.

3.3.3. Caracterización de velocidades

En esta sección se presentarán experimentos que buscan determinar el rango de velocidades para los cuales el robot cumple con las hipótesis presentadas en la Sección 3.1.5, es decir:

- El tiempo de establecimiento para un cambio de velocidades (escalón) en los motores debe ser despreciable en comparación al cambio en la velocidad.
- El robot se mueve sin deslizamiento entre sus ruedas y el piso.

Además, en la Sección 3.3.3.3 se presenta una particularidad encontrada a la hora de configurar el control de las velocidades.

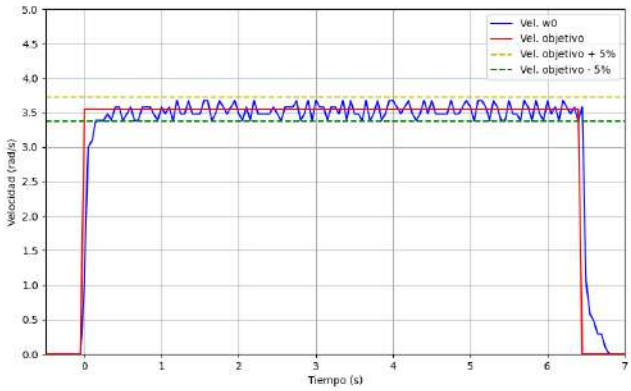
3.3.3.1. Respuesta al Escalón

Con el fin de verificar el cumplimiento de la hipótesis sobre el tiempo de establecimiento del sistema, se desarrolló y llevó a cabo el experimento que se detalla en esta sección.

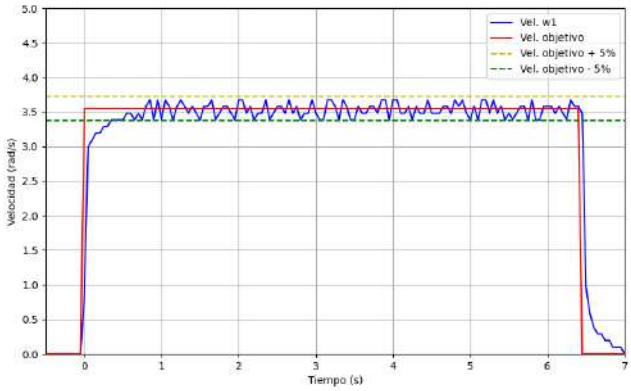
En el experimento se impone un escalón de velocidad a cada rueda del robot desde el reposo y se mide el tiempo de respuesta. Para ello se impone una velocidad al sistema y se registra la respuesta obtenida. Es importante aclarar que el muestreo de la velocidad del robot se realiza cada 50 ms, mientras que el ciclo de control, el cual se describe en la Sección 4.2.1.3, se realiza cada 5 ms.

El escalón que se impone sobre el sistema es: desde el reposo $\omega = 0\frac{rad}{s}$, se impone un escalón de velocidad de rotación según el eje $\hat{Z}$, $\omega = 1\frac{rad}{s}$ y se estudia el tiempo de asentamiento. Esta velocidad sobre el robot corresponde a una velocidad sobre los motores de $\omega_{w_i} = 3{,}55\frac{rad}{s}$.

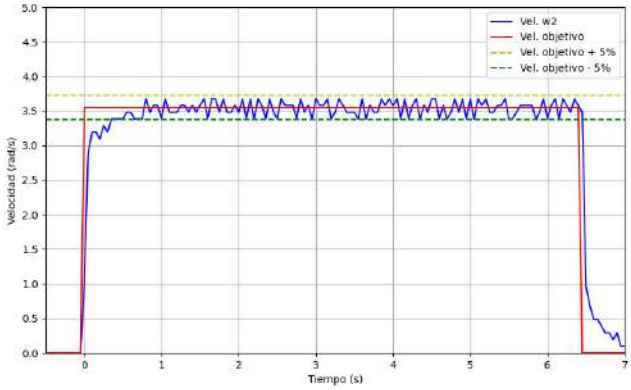
En nuestro caso se toma al tiempo de asentamiento (t_s) como el tiempo necesario para que la salida del sistema ingrese y permanezca dentro de un $\pm 5\,\%$ del valor final. En las Figuras 3.8a, 3.8b y 3.8c las respuestas al escalón normalizadas para cada uno de los motores.



(a) Motor 0.



(b) Motor 1.



(c) Motor 2.

Figura 3.8: Respuesta al escalón normalizada de cada motor.

Como se puede ver en las figuras, el tiempo de asentamiento en ningún caso

excede los 500 ms, si se calcula el tiempo de asentamiento promedio se obtiene $\bar{t_s} = 280\ ms$. Por otra parte se puede ver que no se tiene un sobretiro apreciable. Además, el tiempo de subida promedio obtenido es de $\bar{t_r} = 210\ ms$ y el tiempo de retardo promedio es $\bar{t_D} = 50\ ms$. Todos los valores temporales se pueden observar en la Tabla 3.4.

Tiempo de asentamiento (ms)	280
Tiempo de subida (ms)	210
Tiempo de retardo (ms)	50

Tabla 3.4: Características promedio de la respuesta al escalón

3.3.3.2. Deslizamiento de ruedas

Este experimento se diseñó y realizó para ver si dadas las características del robot, se podría determinar un rango de velocidades para las cuales la plataforma se desplace sin que se produzcan deslizamientos en ninguna de las ruedas.

Para medir el deslizamiento se configuraron dos movimientos, uno hacia adelante y otro hacia atrás. El experimento consistió de 5 pulsos de velocidad constante hacia adelante durante 2 segundos cada uno, deteniendo el robot entre pulsos. Una vez realizados los 5 pulsos de movimiento hacia adelante, se realizó un solo movimiento hacia atrás de 10 segundos. Este tipo de movimiento es útil para determinar si en los momentos de arranque y parada las ruedas del robot patinan. Para poder afirmar que no hubo deslizamiento, la posición final del robot debería ser prácticamente la misma que la posición de inicio.

Se tomaron como referencia medidas en el piso que marcan los momentos de parada entre pulsos, además de la posición inicial y la final. Se realizó el experimento para 5 velocidades distintas: [0.1, 0.15, 0.2, 0.25, 0.3] $\frac{m}{s}$.

Es importante notar que este experimento es muy dependiente del piso en el cual se utilice el robot. Es por ello que esta experiencia se repitió en dos ambientes distintos: un piso de madera y otro de baldosas.

Como era de esperarse, los resultados dependieron fuertemente del piso en el cual se realizaran. Para el piso de madera se logro trabajar sin ningún problema hasta las velocidades mas altas. Mientras que en el caso del piso de las baldosas ya cuando se impusieron velocidades de 0.25 $\frac{m}{s}$ ya se registraron deslizamientos debido a las ranuras y uniones del piso, además de que el piso es mas resbaladizo para el robot.

Como resultado, se puede decir que para cualquier velocidad menor a $v = 0{,}25\ \frac{m}{s}$ no se observó un deslizamiento excesivo en movimientos hacia adelante y atrás en las ruedas para ningún tipo de suelo. Sin embargo, como esta no es la dirección de máxima velocidad para ninguna rueda particular, se decidió que la velocidad máxima del robot en cualquier dirección sea:

$$\|v_{max}\| = 0{,}2\ \frac{m}{s}$$

Si se quisiera replicar el experimento para alguna de las direcciones de máxima velocidad de una rueda, se debería trabajar con velocidades impuestas en la dirección paralela a alguno de los ejes Y_{w_i} según se muestra en la Figura 3.1.

3.3.3.3. Configuración de la frecuencia del PWM

A la hora de configurar el PWM se constataron dos consideraciones prácticas a tener en cuenta para determinar la frecuencia de trabajo. En [37] y [38] se ven tanto consideraciones teóricas como prácticas para lograr comprender el funcionamiento y cómo configurar correctamente el ciclo de trabajo y frecuencia del PWM. Como peculiaridades más notorias se tiene que:

- Al aumentar la frecuencia, la velocidad del motor disminuye aún cuando el ciclo de trabajo se mantiene.
- En un rango de frecuencias de trabajo bajas, el motor produce ruidos o zumbidos que pueden ser molestos.

Se recomienda en general trabajar con frecuencias apenas superiores al rango audible para evitar las perturbaciones auditivas del motor pero que los efectos de aumentar la frecuencia no sean apreciables en la práctica. Por ello, se decidió trabajar con una frecuencia para el PWM de 20 kHz.

3.3.4. Caracterización del sensor LiDAR

Para el sensado de distancia, como ya se mencionó en la Sección 2.2, se utiliza un RPLidar A1. Se procedió a realizar una serie de pruebas de diferentes medidas para caracterizar dicho sensor y así conocer en detalle su funcionamiento. Se eligieron tres casos de estudio en los que evaluar el desempeño del LiDAR: medidas cercanas a la distancia mínima de medición, medidas cercanas a la distancia máxima de medición y por último medidas dentro de estos límites, que son indicados en la hoja de datos [39].

El procedimiento para realizar la experiencia es análogo para cada caso. Para la obtención de las medidas se ubicó el LiDAR a cierta distancia de una pared para su dirección 0^{o}, como se muestra en la Figura 3.9, y de esta forma se buscó determinar la precisión de las medidas. El rango de medición presente en la hoja de datos va desde los 15 cm a los 12 m, es por esto que se utilizó una distancia de 15 cm, luego 12 m y 3 m para cada caso respectivamente. Vale la pena aclarar que la figura antes mencionada muestra la disposición utilizada para las pruebas con la distancia mínima y no se agregan para las distancias mayores pues no aportan valor. Se pueden obtener detalles de cómo recrear las pruebas realizadas en el Anexo B.

Figura 3.9: Disposición utilizada para tomar medidas con el LiDAR.

Para distancias cercanas al valor mínimo de medición, se logra tener una lectura de media 14.97 cm y desviación estándar de 0.05 cm por lo que se puede decir que es una medida exacta y precisa. Se dice exacta pues dista 0.03 cm, un 0.2 %, de los 15 cm esperados. Por otro lado, es precisa ya que, como se puede observar en el histograma de la Figura 3.10, de 140 valores tomados 102 corresponden a medidas muy cercanas entre sí alrededor de los 15 cm. A su vez, si bien los demás valores se encuentran por debajo de la media menos la desviación estándar, estos no se alejan demasiado y esto se ve reflejado en que el valor de la desviación estándar es bajo siendo un 0.3 % de la media. Se concluye entonces que se tiene una baja dispersión en las medidas.

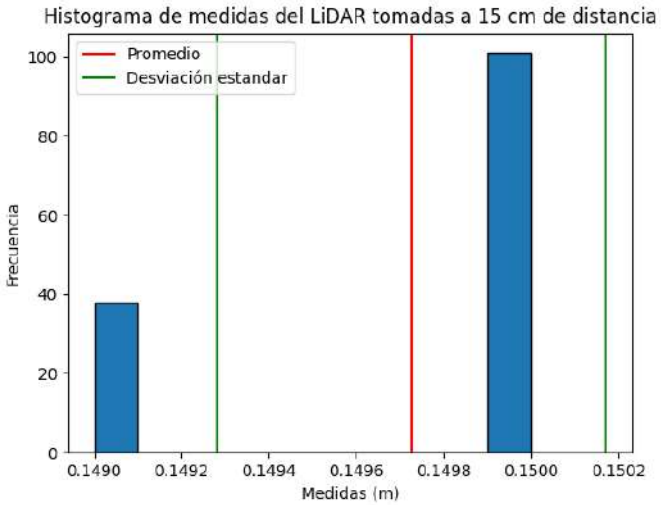


Figura 3.10: Histograma de las medidas del LiDAR tomadas a 15 cm de distancia de una pared.

Luego, para el caso de 3 m de distancia entre el LiDAR y la pared se obtuvo que el valor medio de las medidas fue 3.153 m y la desviación estándar fue de 0.003 m. Nuevamente se tiene que las medidas son precisas, distan poco entre ellas teniendo una dispersión menor a un 0.1 %. Se realizó el histograma presente en la Figura 3.11 en donde se aprecia que las medidas se concentran alrededor de la media. A diferencia del caso anterior, esta serie de medidas no se puede decir que es exacta ya que se tiene un error de un 5 % frente al valor esperado y tampoco coincide con el error que estima el fabricante en su hoja de datos de 1 % de la medida.

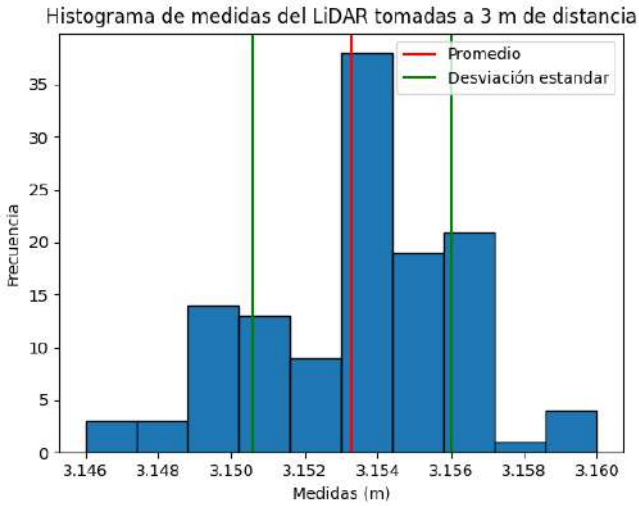


Figura 3.11: Histograma de las medidas del LiDAR tomadas a 3 m de distancia de una pared.

Finalmente, se tuvo que para el caso de medidas de distancia máxima, con el LiDAR a 12 m de una pared, no se logró obtener valores representativos de la medida del dispositivo. En una serie de 160 medidas aproximadamente 10 devuelven un resultado numérico mientras que las restantes devuelven *'inf'*. Se puede concluir que el LiDAR no tiene un buen desempeño en el alcance máximo teórico y que no es posible utilizarlo para un rango tan extenso.

A partir de las lecturas registradas en todas las pruebas se puede obtener una tabla comparativa entre los valores esperados y el valor medio de las medidas tomadas por el láser en cada caso, ver Tabla 3.5. Se puede apreciar que, como ya se mencionó, para una distancia de 12 m el LiDAR no logra tomar una medida.

Valor esperado	15 cm	3 m	12 m
Valor medio de medidas	14.97 cm	3.153 m	-
Desviación estándar	0.05 cm	0.003 m	-
Número de medidas	140	130	160

Tabla 3.5: Valores obtenidos a partir de las medidas del LiDAR en los diferentes casos.

Con el fin de obtener un resultado práctico de cuál sería el máximo de medición admisible, se realizó la siguiente experiencia. Se fue disminuyendo la distancia entre el LiDAR y la pared partiendo de los 12 m hasta encontrar un valor aproximado en el que las medidas adquiridas sean representativas de lo que se quería medir. Se obtuvo que a partir de aproximadamente los 8 m de distancia de la pared el LiDAR retorna en una serie de medidas valores en su mayoría numéricos. Para apreciar esto se construyó la Tabla 3.6 donde en un comienzo se tiene que la mayor parte de las medidas tomadas por el LiDAR no eran medidas válidas y luego, a medida que desciende la distancia entre el sensor y el objeto a medir, comienzan a prevalecer los valores de medidas válidos por sobre los valores no numéricos. Se tomó como criterio que tener un 20 % de medidas no numéricas, es decir 80 % de valores válidos, es suficiente para poder determinar el valor medido por el dispositivo y resulta entonces que el rango máximo de medición en la práctica es de 8 m.

Distancia a medir	# medidas	# *'inf'*	% de valores *'inf'*
12 m	160	150	94 %
10 m	137	115	84 %
9.5 m	146	115	79 %
9 m	170	116	68 %
8.75 m	160	89	56 %
8.5 m	159	38	24 %
8.25 m	164	34	21 %
8 m	159	32	20 %
7.75 m	163	34	20 %
6.5 m	143	26	18 %

Tabla 3.6: Porcentaje de mediciones *'inf'* en una serie de medidas según la distancia a medir.

Como conclusión se extrae que el LiDAR se comporta bien a pequeñas distancias, cercanas a su rango mínimo de medición, mientras que tiende a empeorar su desempeño cuando aumenta la distancia hacia los objetos que se quieren medir. No obstante, si bien se tiene un error presente, teniendo en cuenta la aplicación en la que se utilizará el sensor esto no será un problema grave. Ya que el robot cuando requiere de una medida certera y de mayor exactitud es en el caso en que se acerca a objetos, es decir cuando disminuye la distancia entre el LiDAR y los objetos del entorno, y ese es el caso en que mejor trabaja el dispositivo.

3.3.5. Consumo de batería

En esta sección se busca caracterizar el consumo de batería del robot. Determinar el consumo promedio de la plataforma así como los picos de consumo es crucial para conocer el tiempo de vida de la batería dada su capacidad nominal, así como también los picos de consumo que la batería debe soportar. Para ello se utilizó el analizador de consumo Otii Arc 3 [40].

Debido a que el robot se alimenta de dos niveles de tensión diferentes para la plataforma motora y la plataforma de cómputo se dividió el análisis en dos secciones.

3.3.5.1. Consumo de la plataforma motora

Para determinar el consumo de la plataforma motora se trabajó con el diagrama de conexión que se ve en la Figura 3.12. En él se puede ver que la parte motora del robot se alimenta directamente del Otii mientras que la plataforma de cómputo y sensado se alimenta de la batería incluida en el robot.

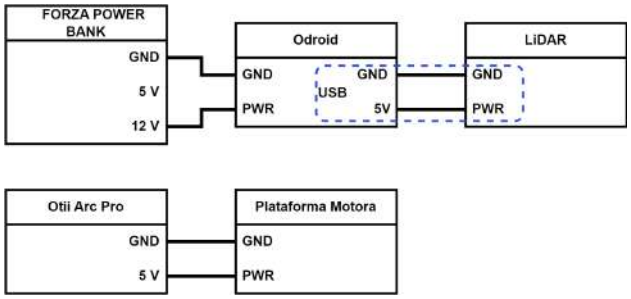


Figura 3.12: Diagrama de conexión para la determinación del consumo de batería de robotito.

En esta práctica se tomaron medidas de consumo para los motores parados, durante el arranque y en movimiento para determinar los diferentes consumos que se podrían observar luego en su uso. En la Figura 3.13 se observa el perfil de consumo de la plataforma motora como respuesta de un escalón de velocidad.

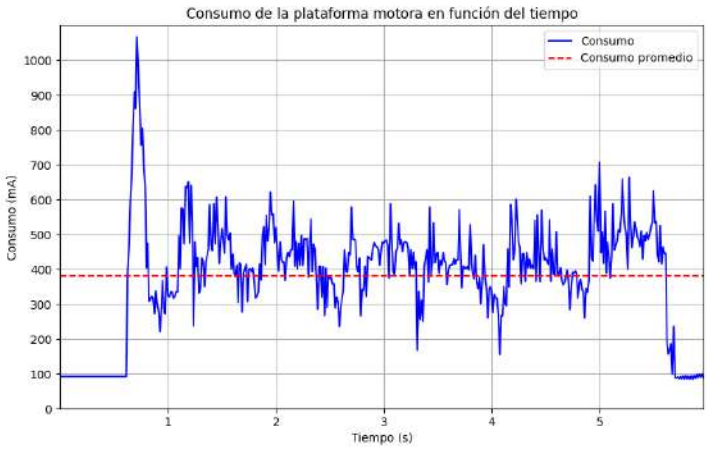
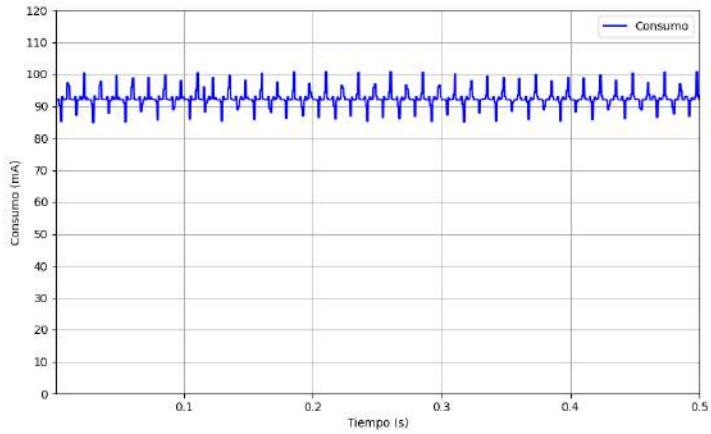
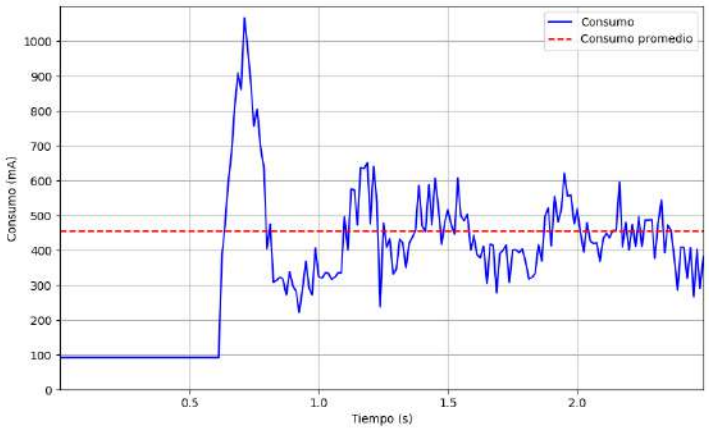


Figura 3.13: Perfil de consumo como respuesta a un escalón de velocidad

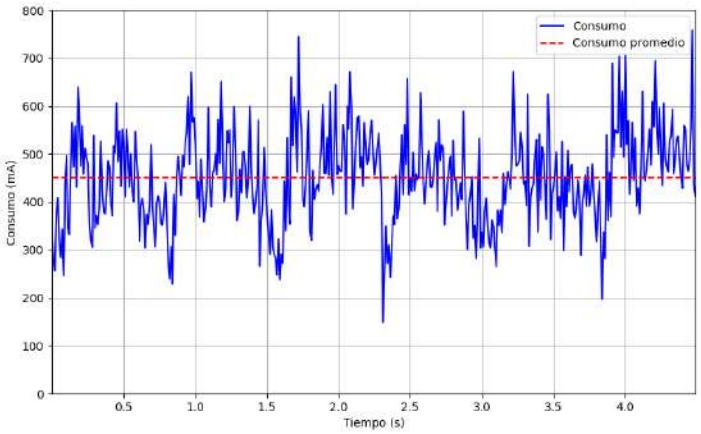
A partir del consumo ilustrado en la imagen anterior se obtienen las Figuras 3.14a, 3.14b y 3.14c donde se muestran los perfiles de consumo para los tres casos mencionados.



(a) Quieto



(b) En arranque



(c) En movimiento

Figura 3.14: Perfiles de consumo de la plataforma motora.

Los resultados ilustrados en las anteriores figuras se resumen en la Tabla 3.7 donde se exponen los valores de consumo mínimo, máximo y promedio en cada caso.

	Máximo	Promedio	Mínimo
Quieto	100 mA	93 mA	84 mA
Arranque	1.07 A	455 mA	92 mA
En movimiento	758 mA	450 mA	149 mA

Tabla 3.7: Resultados de consumos plataforma motora.

3.3.5.2. Consumo de la plataforma de cómputo y sensado

En esta sección se tiene como objetivo determinar el consumo de la plataforma de cómputo. Al igual que en la Sección 3.3.5.1, para lograr determinar el consumo se generó un diagrama de conexión que se ve en la Figura 3.15. En él se puede ver que la parte motora del robot en este caso se alimenta directamente de la batería incluida en el robot, mientras que la plataforma de cómputo y sensado se alimenta utilizando una técnica que permite al Otii poder medir el consumo buscado. Como se ve en la Figura 3.15, se tiene que trabajar con una fuente de tensión en serie al Otii puesto que no es capaz de entregar los 12 V que necesita el Odroid para su alimentación.

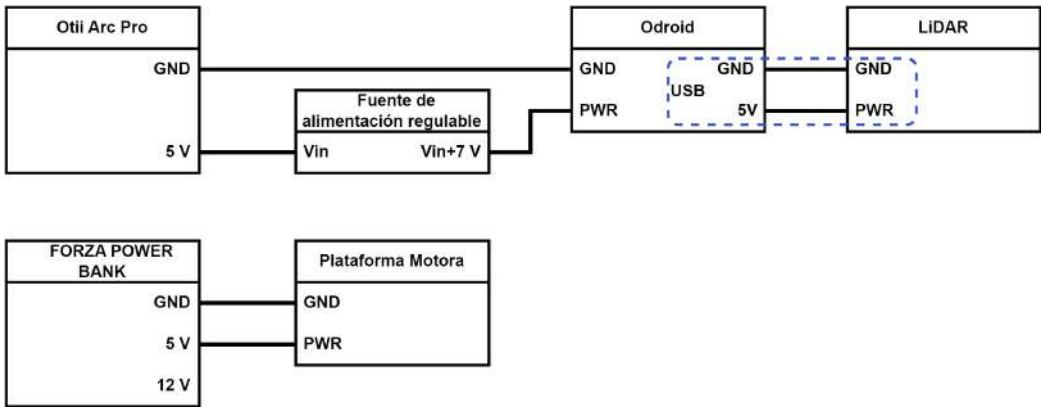


Figura 3.15: Diagrama de conexión para la determinación del consumo de batería de robotito.

Se tomaron las medidas de consumo que se observan en la Figura 3.16. Los mismos resultados se resumen en la Tabla 3.8 donde se muestra el consumo mínimo, máximo y promedio.

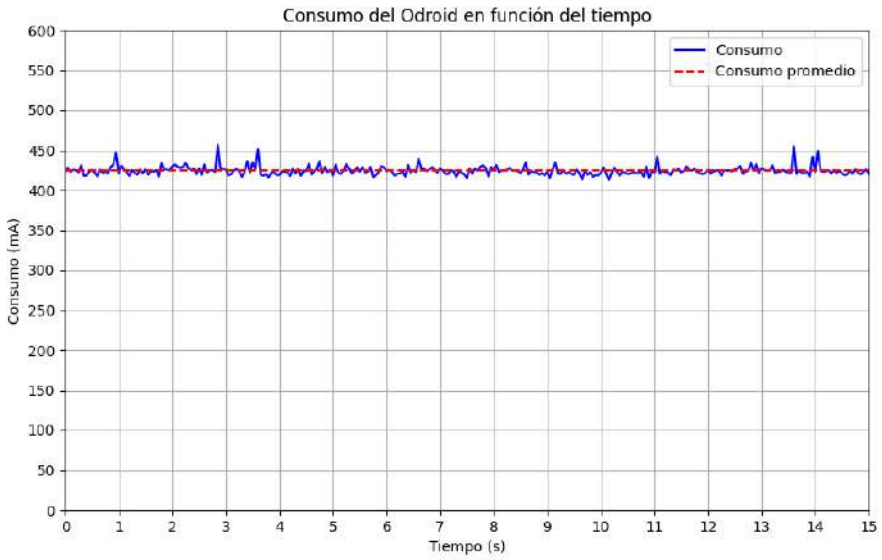


Figura 3.16: Consumo registrado para la plataforma de cómputo y sensado.

Máximo	457 mA
Promedio	424 mA
Mínimo	412 mA

Tabla 3.8: Resultados de consumos para la plataforma de cómputo y sensado.

Como conclusión de estos experimentos se deriva un requerimiento para los picos de consumo de la batería, así como también de la capacidad nominal. Es decir, la batería debe ser capaz de soportar picos de hasta 1.07 A para su salida de 5 V y 457 mA para su salida de 12 V. Además, como ejemplo en la Ecuación 3.20 se calcula la capacidad (C) de la batería necesaria si se quisiera que el robot se encuentre durante una hora en movimiento y con la plataforma de cómputo prendida. Es importante mencionar que la capacidad de la batería en este caso se calculo en Wh debido a que esta debe entregar diferentes niveles de tensión y por ende sería incorrecto determinar la capacidad en unidades de mAh.

$$C = (450\ mA * 5\ V + 424\ mA * 12\ V) * 1hora = 7{,}338\ Wh \tag{3.20}$$

La batería utilizada en este proyecto cuenta con una capacidad nominal de 14,8 Wh[6] por lo que se puede suponer que el tiempo máximo de servicio sería de 2,02 horas. Además, se comprueba que es capaz de soportar los picos de corriente ya que las protecciones accionan a partir de 1,2 A.

[6] `https://forza-ups-frontend.s3.amazonaws.com/media/docs/DC-140USB-ENG.pdf`

Capítulo 4

Firmware

En este capítulo se presentan los conceptos e implementaciones de firmware desarrollados para obtener una plataforma motora que cumpla con las necesidades del proyecto. Como se mencionó en el Capítulo 2, la plataforma motora se basa en Robotito. Sin embargo, debido a que se trata de un robot diferente con distintos objetivos, especificaciones y características, fue necesario realizar modificaciones e implementar componentes de firmware adicionales para lograr la plataforma motora deseada.

El capítulo comienza con una introducción al concepto de Sistema Operativo para Tiempo Real o Real Time Operating System (RTOS) que fue la base sobre la cual se desarrolló la arquitectura y los diferentes componentes de la plataforma. Por último, se presenta la solución de firmware implementada y se examinan en detalle cada uno de los componentes desarrollados. Para cada componente se proporcionan justificaciones, aspectos teóricos relevantes y detalles de su implementación.

4.1. RTOS

Un sistema operativo en tiempo real es un tipo de sistema operativo diseñado específicamente para ejecutar aplicaciones que requieren un tiempo de respuesta predecible y confiable. Un RTOS brinda un control preciso sobre los recursos del sistema, como el tiempo de procesamiento y la memoria. Además, estos sistemas ofrecen mecanismos de prioridad y planificación de tareas para asegurar que las tareas críticas se ejecuten en momentos determinados.

Una de las características fundamentales de un RTOS es su estructura de tiempo determinista, lo que implica que el tiempo de respuesta de una tarea puede predecirse con exactitud. Esta característica resulta crucial en aplicaciones críticas en tiempo, como el control de procesos industriales, sistemas aeroespaciales y automotrices, entre otros, donde una falla en el tiempo de respuesta puede tener consecuencias graves.

En el caso particular de este proyecto, se optó por utilizar el RTOS llamado freeRTOS [41] para desarrollar los componentes de firmware de la plataforma. Esta elección se basa en la necesidad de garantizar con precisión el momento donde se

ejecutan tareas críticas como el control de los motores o la generación de la odometría. Es muy importante garantizar los tiempos de ejecución para estas tareas ya que son altamente sensibles a variaciones temporales, como se menciona más adelante en este capítulo. Además, el microcontrolador utilizado (ESP32) provee un entorno de desarrollo o SDK [1] llamado Espressif IoT Development Framework (ESP-IDF), el cual provee una implementación de freeRTOS modificada para las características de hardware del microcontrolador. Se puede encontrar más información sobre freeRTOS y de ESP-IDF en el Apéndice A

4.2. Firmware de Robotito

En esta sección se introduce el software desarrollado para controlar el hardware que se presentó en la Sección 2.1. Previo a entrar en la descripción detallada del software desarrollado, se explican algunos conceptos que dan marco para comprender la metodología que se siguió a la hora de implementar el firmware.

El desarrollo de los componentes se realizó básicamente con una perspectiva "Bottom-Up", es decir, del más bajo nivel hacia arriba. Se buscó desarrollar siguiendo el modelo de capas, donde una capa provee servicios a su capa superior y utiliza los servicios que brinda su capa inferior. De esta forma, se obtiene un trabajo con mayor modularidad e independencia de los módulos desarrollados. Se desarrolló desde los componentes que interactúan con el hardware, que se los llamarán Hardware Abstraction Layer (HAL), hasta el más alto nivel donde se abstrae incluso la concepción de robot.

4.2.1. Componentes Desarrollados

Los componentes desarrollados fueron: las HAL para el sensado de los encoders y el control de los motores, una librería con implementación de funciones matemáticas matriciales, un controlador PID (Proporcional-Integral-Derivativo) para el control de velocidad de cada motor, un módulo capaz de generar la odometría del robot, la implementación del protocolo de comunicación con la SBC y finalmente la integración de todos los módulos. En la Figura 4.1 se muestran estos componentes y sus relaciones.

[1] SDK, "Software Development Kit" en inglés, es un conjunto de herramientas, bibliotecas, documentación y recursos que permite crear aplicaciones de software para plataformas específicas simplificando y agilizando el proceso.

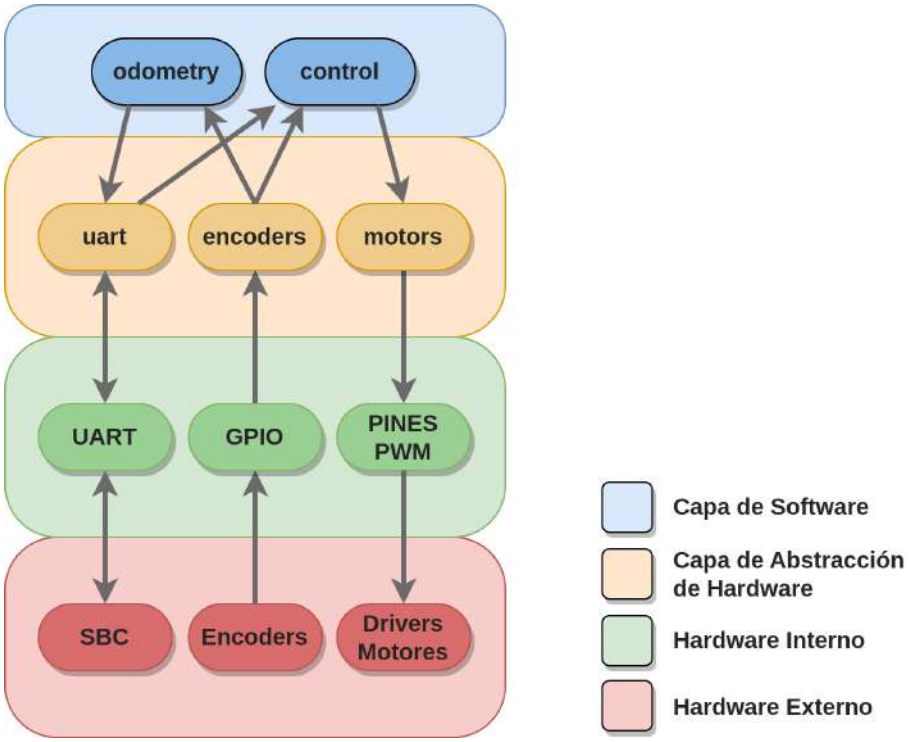


Figura 4.1: Diagrama jerárquico de los módulos. Cada color representa una capa diferente.

Si bien el desarrollo se realizó siguiendo la perspectiva "Bottom-Up", el orden en el cual se procederá a describir los módulos es diferente, con el fin de seguir un hilo a la hora de describir las diferentes funcionalidades. Se comienza describiendo la HAL de los encoders (Sección 4.2.1.1) y de los motores (Sección 4.2.1.2), para luego proceder con el módulo que realiza el control de las velocidades del robot (Sección 4.2.1.3). Se sigue describiendo el módulo encargado de realizar la odometría (Sección 4.2.1.4) y se finaliza con la implantación del protocolo de comunicación serial desarrollado (Sección 4.2.1.5).

4.2.1.1. Encoders

El módulo de manejo de los encoders proporciona a los módulos de una capa superior una forma de interactuar con el hardware de manera transparente. Este módulo implementa una máquina de estados sencilla la cual traduce el comportamiento del encoder a sus respectivos pasos.

Siguiendo una máquina de estados que tenga en cuenta el nivel de las dos salidas lógicas del encoder, A y B, se puede generar un evento cuando se dé un paso ya sea en un sentido o el otro. La máquina de estados queda constituida por los estados que se presentan en la Tabla 4.1.

Posición	Pin A	Pin B
Paso 1	0	0
1/4	1	0
1/2	1	1
3/4	0	1
Paso 2	0	0

Tabla 4.1: Máquina de estados del encoder

El manejo de los encoders utilizando una máquina de estados posee grandes ventajas. La primera es que se puede interpretar de manera muy rápida si el movimiento es en un sentido o el otro dependiendo de si la primera señal en cambiar a nivel alto es el Pin A o el Pin B. Otra ventaja es que el uso de una máquina de estados resuelve el problema de los rebotes con su implementación ya que en el peor caso se quedaría conmutando entre estados intermedios sin generar cambios de pasos.

La única desventaja apreciable que se encontró al trabajar el encoder con una máquina de estados es que los pasos que produce esta implementación son al cabo de 4 tics. Es decir, la HAL de los encoders interrumpe y genera un paso luego de 4 tics. De todas formas, se puede configurar la HAL para trabajar con doble precisión y entonces generar un paso cada dos tics y de esta forma no perder tanta información.

Para cada uno de los encoder se debe instanciar una variable del tipo encoder_h_t la cual almacena la información necesaria para controlar los dispositivos físicos. Este tipo de dato está constituido con la información que se ve a continuación:

- Pines X e Y del microcontrolador que están conectados a los pines A y B del encoder respectivamente.
- Estado del encoder, siguiendo los estados presentados en la Tabla 4.1
- Un contador para conocer la cantidad de tics
- La función de callback para atender los eventos del encoder.

Con el fin de registrar cada uno de los eventos y configurar cada uno de los encoders se tienen las siguientes funciones:

- **encoder_setup:** Esta función inicializa un encoder a partir de dos pines y genera todas las configuraciones necesarias en el microcontrolador para poder reaccionar al comportamiento de este componente.
- **encoder_unsetup:** Libera recursos, borrando toda configuración realizada para un encoder en particular.
- **encoder_register_callback:** Esta función registra la función de callback a la cual se recurre al detectar un tic, es decir un paso en la máquina de estados.

Si bien este módulo conservó el diseño e implementación proveniente del firmware original de robotito, se considera de gran utilidad explicar el funcionamiento para entender la globalidad del proyecto.

4.2.1.2. Control de motores

Al igual que la HAL para los encoders, esta HAL provee una interfaz sencilla para el manejo de los motores. Es una abstracción en firmware del hardware de los drivers para los motores presentados en el Capítulo 2 y de la implementación para el manejo de los pines del microcontrolador utilizando PWM que provee el SDK.

Se basa en la estructura motor_h_t en la cual se almacena la información necesaria para realizar el manejo de cada uno de los motores en lazo abierto. Este módulo impone velocidades de giro a un motor y dirección de giro, sin implementar el control de la velocidad. La velocidad se impone haciendo uso del periférico del ESP que implementa PWM en sus pines.

Este módulo se compone de las siguientes funciones:

- **motor_install:** Esta función almacena en la estructura motor_h_t dos pines con los cuales se maneja al motor, una unidad de PWM, dos puertos de la unidad de PWM los cuales se conectan a los pines y una referencia al timer con el cual se controlarán los tiempos.
- **motor_set_speed:** Asigna una velocidad al motor en forma de duty cycle.
- **motor_stop:** Función para detener al motor.
- **motor_start:** Configura e inicializa la unidad de PWM seleccionada en motor_install con los parámetros, lo que resulta en iniciar al motor.

Al igual que el módulo de manejo de encoders, se conservó el diseño e implementación proveniente del firmware original de robotito y se añade la descripción pues se considera de gran utilidad explicar el funcionamiento para entender la globalidad del proyecto.

4.2.1.3. Control PID

Este módulo es el responsable de que el movimiento del robot se realice de manera controlada. En él se implementa un controlador PID como el que se muestra en la Figura 4.2 para regular la velocidad de cada motor en función de un valor deseado. Para lograr esto, se utilizan los servicios de las dos HAL descritas anteriormente, que se encargan de manejar los motores y obtener información de los encoders.

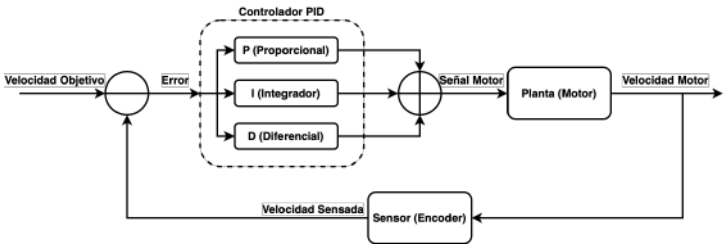


Figura 4.2: Diagrama de bloques del controlador PID.

En un controlador PID se pueden relacionar las diferentes constantes con diferentes naturalezas del error. El término proporcional (P) está directamente relacionado con el error actual y representa la magnitud del ajuste necesaria en función del error actual. Cuando más grande es el error, el término proporcional genera un ajuste proporcionalmente mayor. La constante integral (I) considera la acumulación de errores pasados a lo largo del tiempo. Esto es útil para corregir errores persistentes o desviaciones de estado estacionario. El control integral permite corregir el error acumulado a lo largo del tiempo, mejorando la precisión del sistema. Por último, el ajuste derivativo (D) se enfoca en la tasa de cambio del error. Este ayuda a prever la tendencia del error y permite anticipar en el tiempo ajustes.

A nivel conceptual, el módulo implementa el Algoritmo 1. Para cada motor, se requiere una velocidad objetivo, el error acumulado, el error previo y las constantes del controlador. Utilizando esta información, se obtiene la velocidad actual de cada motor haciendo uso de la HAL encoder. Con esa velocidad, se calcula la diferencia entre la velocidad objetivo y la medida. Con el error entre la señal objetivo y la medida se actualiza el error acumulado y se calcula la variación temporal del error. Finalmente, utilizando las constantes de proporción correspondientes a cada componente, se obtiene una nueva velocidad de control y utilizando la HAL de manejo de motores se fija dicha velocidad. Este procedimiento se repite de forma continua en intervalos de tiempo T definidos.

Algorithm 1: Controlador PID

Input: target_vel, acum_error, prev_error, KP, KI and KD

for *motor in motors* **do**
- $actual\_vel = GetTicsPerSec(motor)$
- $error = target\_vel - actual\_vel$
- $acum\_error = acum\_error + error * T$
- $rate\_error = \frac{(error - prev\_error)}{T}$
- $SetMotorSpeed(motor,\ KP * error + KI * acum\_error + KD * rate\_error)$
- $prev\_error = error$

end

Wait(T)

4.2.1.4. Odometría

La odometría es el proceso mediante el cual se busca estimar la posición y velocidad de un robot en función de la información de rotación de ruedas. La función principal de este módulo es suministrar información de velocidad y posicionamiento del robot a las capas superiores a partir del uso de los servicios de la HAL de los encoders. El proceso operativo se describe conceptualmente en el Algoritmo 2.

En primer lugar, se obtiene la velocidad promedio de cada rueda en el intervalo temporal entre ejecuciones consecutivas del algoritmo mediante el uso del encoder correspondiente. En segundo lugar, se realiza una transformación utilizando las velocidades de todas las ruedas para obtener las velocidades en el marco de referencia del robot. Finalmente, utilizando la velocidad del robot y asumiendo que la velocidad se mantuvo constante en todo el intervalo de tiempo T, se calcula el desplazamiento realizado en dicho intervalo, asumiendo que no existe desplazamiento entre las ruedas y el suelo. Este método es una buena aproximación de la realidad si el tiempo T es pequeño en comparación al cambio de velocidades del sistema y si se puede considerar la velocidad constante en dicho intervalo. A partir de la información de velocidad y desplazamiento, se construye la odometría del robot siguiendo la ecuación 3.11.

Algorithm 2: Odometría

Input: x,y,ϕ

for *encoder in encoders* **do**
  $vel[encoder] = GetSpeed(encoder)$
end
$(\dot{x},\ \dot{y},\ \dot{\phi}) = GetRobotSpeedFromMotorSpeed(vel)$
$x = x + \dot{x} * T$
$y = y + \dot{y} * T$
$\phi = \phi + \dot{\phi} * T$
$odom = (x, \dot{x}, y, \dot{y}, \phi, \dot{\phi})$
Wait(T)

4.2.1.5. Comunicación Serial

Este módulo se desarrolló con el objetivo de establecer un canal de comunicación serial para transmitir la información necesaria entre el Odroid y el ESP. La comunicación desde el ESP al Odroid tiene como propósito el comunicar los resultados de la odometría para que la SBC pueda trabajar con ellos. Además, en el otro sentido, la comunicación desde el Odroid al ESP tiene como propósito enviar comandos de velocidad para imponer sobre el robot.

Con este fin, se implementó un módulo que utiliza un periférico UART [42] del ESP. Este protocolo, como se muestra en la Figura 4.3, consiste en mensajes de longitud de 12 bytes desde la SBC hacia la plataforma motora, que contienen información sobre la velocidad deseada, y mensajes de 24 bytes desde el ESP hacia el Odroid, que contienen información de la odometría. Es importante destacar que este protocolo no cuenta con un número de secuencia ni encabezados, por lo tanto,

el único tipo de mensaje que se espera recibir es el mencionado anteriormente.

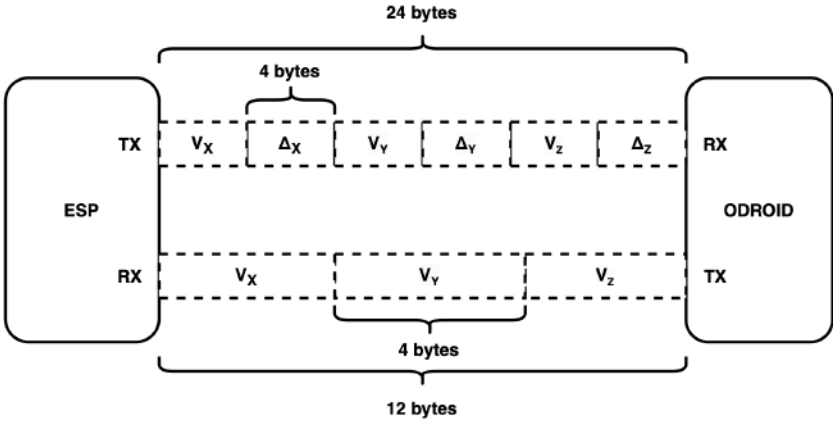


Figura 4.3: Protocolo de comunicación ESP-Odroid.

Cuando el ESP recibe un mensaje proveniente de la Odroid, este módulo le envía la información que se recibió al módulo de control para actualizar la velocidad objetivo de la plataforma motora. De la misma forma, cuando desde la generación de la odómetro se desea reportar la información a la Odroid, se envía un mensaje por UART con esta información.

Capítulo 5

Software

En este capítulo se introduce al lector a los aspectos de software que se relacionan con el proyecto. Este capítulo se divide en dos partes: Primero se exploran las bases teóricas que sustentan la solución de software implementada en este trabajo, Sección 5.1. Luego, en la Sección 5.2, se describe dicha solución. En conjunto, proporcionan los conocimientos fundamentales para comprender y apreciar la solución implementada.

5.1. Conceptos teóricos

En esta sección se presentan los conceptos teóricos que se consideran fundamentales para comprender la solución implementada. Se explican conceptos relacionados al middleware-framework utilizado, que desempeña un rol central en el diseño de la arquitectura de la solución. Además, se detallan conceptos íntimamente ligados con el problema en sí, como la localización, el mapeo, la navegación y la cooperación entre robots.

5.1.1. ROS2

La presente subsección tiene como objetivo introducir conceptos y características principales de ROS2, el cual era un requerimiento para la solución a desarrollar. Se trata de un destacado middleware-framework [43] multi-plataforma utilizado en el ámbito de la robótica. Aunque su denominación, «Robot Operating System», podría sugerir que es un sistema operativo, en realidad se trata de una colección de controladores, librerías y herramientas que brindan infraestructura para el desarrollo de sistemas robóticos complejos. Se trata de un proyecto libre y se encuentra respaldado por una gran comunidad que brinda soporte y desarrollo.

Al finalizar este capítulo, se espera que el lector adquiera una adecuada comprensión de las bases de ROS2, lo que permitirá un mejor entendimiento de la solución que se menciona en la Sección 5.2.

5.1.1.1. Arquitectura

ROS2 se destaca entre otras características por su flexibilidad y modularidad. Estas cualidades son respaldadas por su arquitectura en capas (Figura 5.1), una estructura fundamental que aporta abstracción y modularidad al sistema. Esta arquitectura no solo facilita la interacción y comunicación efectiva entre diversos componentes del sistema, sino que también simplifica la tarea de sustituir o actualizar dichos componentes de manera ágil. Como se presentará más adelante, esto permite desarrollar utilizando distintos lenguajes, lo cual sumado a la compatibilidad multi-plataforma de ROS2, brinda una gran versatilidad. A continuación, se presenta brevemente cada una de las capas.

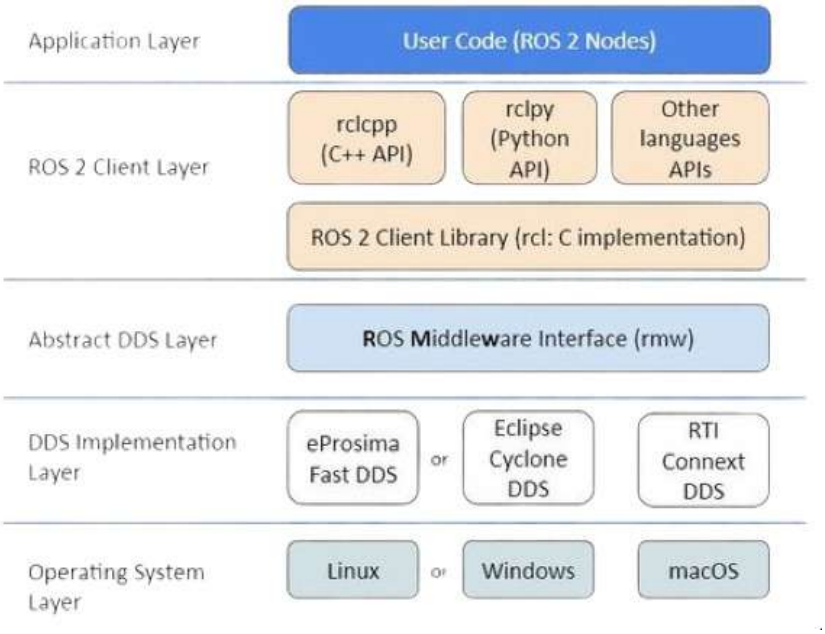


Figura 5.1: Diagrama de la arquitectura de ROS2. Extraída de [44].

Primero se tiene la capa de Aplicación, la capa de mayor nivel de abstracción. En ella se encuentra el código implementado para cada solución, el cual, como se explica más adelante, se organiza en Nodos. El lenguaje en el cual se implementen dichos nodos se encuentra relacionado con la siguiente capa.

La segunda capa se denomina *ROS2 Client Layer*. Se basa, y de allí su nombre, en una biblioteca denominada ROS2 *Client Library* (*rcl*) que implementa la lógica y las funcionalidades centrales de ROS2. A su vez, esta capa cuenta con bibliotecas específicas para los distintos lenguajes que se pueden utilizar en la capa de aplicación. Las bibliotecas específicas encapsulan las funcionalidades de *rcl* con interfaces del correspondiente lenguaje. Esta capa logra que las funcionalidades principales de *rcl* se mantengan con comportamientos similares y coherentes independientemente del lenguaje utilizado. Por defecto, en ROS2 se proveen dos bibliotecas específicas: *rclcpp* para C++ y *rclpy* para Python. Sin embargo, existen otras creadas por la comunidad, por ejemplo *rclnodejs* para NodeJs.

La tercera y cuarta capa están relacionadas con la utilización de DDS/RTPS (Distributed Data Service y Real Time Publish Subscribe Protocol). ROS2 aprovecha la tecnología DDS/RTPS como infraestructura central para la gestión, serialización y transferencia de datos en sus comunicaciones. Aunque los fundamentos subyacentes son diversos [45], en resumen, se utiliza un DDS ya que opera como un middleware integral que ofrece capacidades cruciales para ROS2. Se destaca por su enfoque punto a punto, que contrasta con el enfoque centralizado del ROS1 a través del ROS Master. Los DDS también presentan la ventaja de permitir la configuración de la comunicación mediante el uso de políticas Quality of Service (QoS), otorgando un mayor control sobre cómo se establece la calidad y prioridad de la comunicación. En particular, la capa tres presenta una interfaz común, que abstrae la implementación del DDS a utilizar. Mientras que la capa cuatro es la implementación en sí de dicho DDS. Existen varias implementaciones soportadas [46], como eProsima Fast DDS y Cyclone DDS. Si bien no se profundizará sobre el funcionamiento de las diferentes implementaciones de DDS, ni las diferencias entre ellas, se mencionan brevemente en la Sección 7.2.5.

La última capa interactúa directamente con el hardware y el sistema operativo. Proporciona abstracción del hardware y capacidades en tiempo real. ROS2 aprovecha las características del sistema operativo subyacente para gestionar recursos, planificación de proceso y la comunicación entre ellos.

5.1.1.2. Componentes: Nodos, Tópicos, Servicios y Acciones

Una de las principales características de ROS2, que lo diferencia de otros middleware-frameworks es su organización. Esta se puede describir como un grafo dirigido, donde los vertices representan bloques de computo y las aristas representan el flujo de información. Esto incentiva principalmente que los sistemas implementados sean altamente modulares.

5.1.1.2.1. Nodos

Los nodos son componentes fundamentales dentro del grafo, ya que en ellos es donde se implementan las distintas funcionalidades del sistema. La comunicación entre nodos se realiza a través de tres tipos de interfaces: tópicos, servicios o acciones, que se describen en las Secciones 5.1.1.2.2, 5.1.1.2.3 y 5.1.1.2.4 respectivamente. ROS2 utiliza un lenguaje simplificado, denominado *Interface Definition Language* (IDL), mediante el cual se determina la estructura y el contenido de los mensajes que se utilizan para comunicarse.

5.1.1.2.2. Tópicos

Los tópicos siguen la política de publicador/suscriptor, donde el publicador envía mensajes a determinado tópico y los suscriptores de dicho tópico reciben esos mensajes. Presenta gran versatilidad, ya que permite cualquier combinación de publicadores y suscriptores, como 1 a 1, 1 a N, N a 1 y N a N. Un mismo nodo puede tanto suscribirse como publicar a múltiples tópicos, incluso pueden suscribirse y publicar a un mismo tópico. En el gif de la Figura 5.2 se muestran dos posibles

casos: un caso con un tópico, un nodo que publica y dos nodos que se suscriben, y otro caso con un tópico, dos nodos que publican y dos que se subscriben.

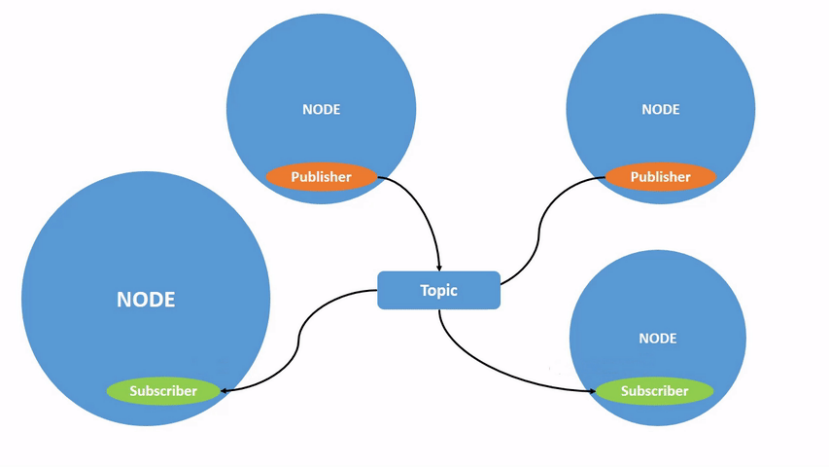


Figura 5.2: Gif de tópicos. Extraído de [2].

5.1.1.2.3. Servicios

Los servicios utilizan una arquitectura cliente-servidor. El cliente realiza una solicitud al servidor, que procesa la información y devuelve un resultado. Los servicios permiten tener un solo servidor pero admiten múltiples clientes. La capacidad de manejar múltiples llamadas concurrentes depende de diversas configuraciones que se explicarán más adelante en la Sección 5.1.1.5. Generalmente, los servicios se utilizan para ejecuciones livianas, que no requieren mucho tiempo para completarse. En este caso, se ve en la Figura 5.3 como los nodos manejan un servicio para lograr la interacción antes descripta.

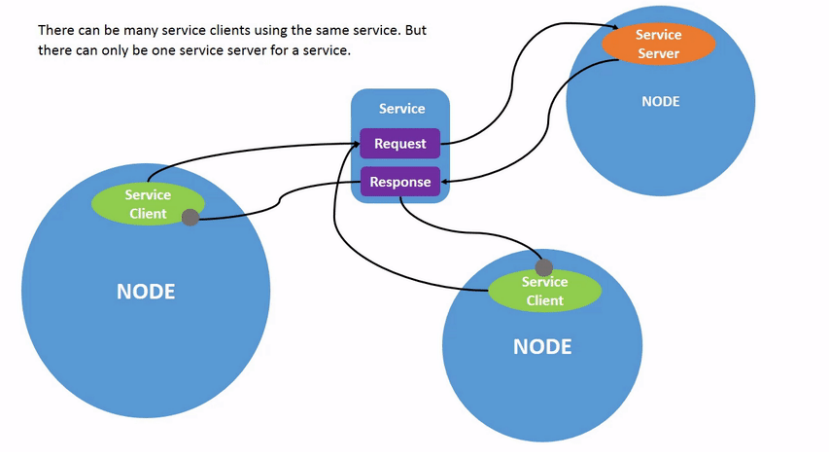


Figura 5.3: Gif de servicios. Extraído de [2].

5.1.1.2.4. Acciones

Las acciones también siguen una arquitectura cliente-servidor. A diferencia de los servicios, brindan retroalimentación mientras se ejecuta la solicitud y permite sobrescribir o cancelarla. Estas características las hacen ideales para procesos prolongados.

Las acciones están compuestas de dos servicios y un tópico, como se observa en la figura 5.4. El servicio *Goal Service* se encarga de recibir los nuevos pedidos enviados por los clientes y retorna si el servidor acepta o no el pedido. En caso afirmativo, el cliente envía a través del servicio *Result Service* el objetivo y el servidor comienza la ejecución, utilizando el tópico *Feedback Topic* para enviar retroalimentación al cliente. Finalizada la ejecución, el servidor envía el resultado como respuesta del servicio *Result Service*, dando por finalizada la comunicación entre el cliente y el servidor.

Un ejemplo para el que comúnmente se utilizan acciones es para la navegación hacia un punto, lo cual puede llevar varios segundos o incluso minutos. Como objetivo se enviaría la posición final deseada, en la retroalimentación se recibiría información sobre el progreso, y en la respuesta se indicaría si se logró o no llegar a la posición deseada.

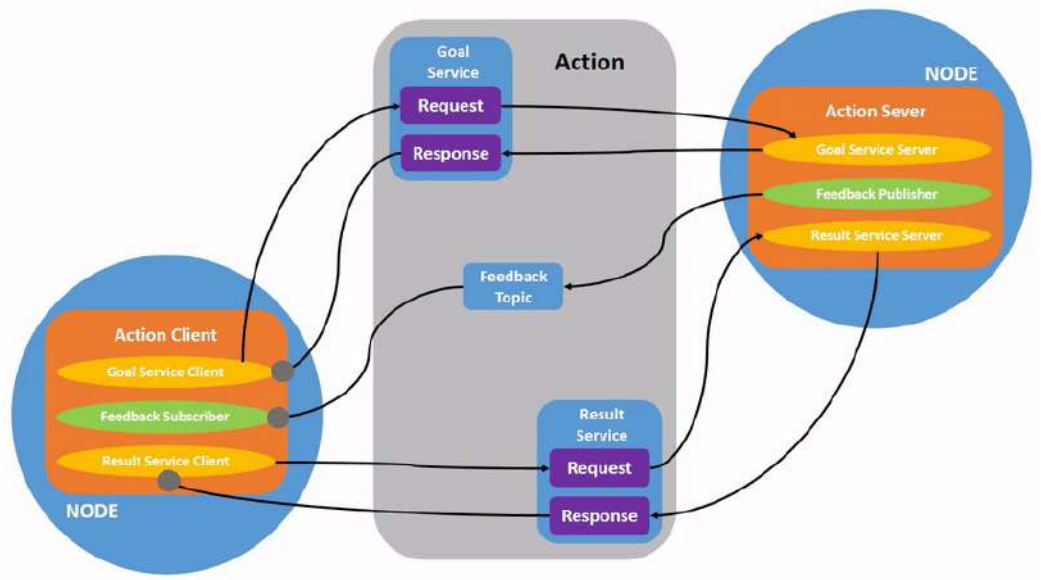


Figura 5.4: Gif de acciones. Extraído de [2].

5.1.1.2.5. Launchfiles

Generalmente los sistemas de ROS2 consisten en la ejecución de muchos nodos, con distintas configuraciones. Si bien es posible ejecutarlos uno a uno, este método no es escalable. El sistema de lanzamiento (*launch system*) de ROS2 soluciona el problema, brindando una herramienta que permite la ejecución y configuración de múltiples nodos con un solo comando. Permite describir completamente el siste-

ma, desde qué nodos deben ejecutarse, dónde deben ejecutarse, qué parámetros y argumentos se deben pasar y cómo deben interactuar entre sí. El sistema de lanzamiento también se encarga de monitorear el estado de los procesos iniciados, y reportar y/o reaccionar ante cambios en el estado de esos procesos. Todo lo anterior se especifica en un archivo de lanzamiento (launchfile), que puede estar escrito en Python, XML o YAML.

5.1.1.3. Estructura

Un proyecto en ROS2 se estructura en un conjunto de paquetes que contienen los componentes necesarios para realizar diversas tareas. La organización básica incluye un "workspace", que es el directorio principal que contiene todos los paquetes relacionados con el proyecto. Cada paquete es una unidad independiente y contiene códigos fuente, recursos y configuraciones específicas.

```
workspace
└── src
    ├── package1
    ├── package2
    ├── ...
    └── packageN
```

Los paquetes pueden estar escritos en diferentes lenguajes de programación, comúnmente C++ o Python. La elección dependerá de las necesidades del proyecto. En general, C++ ha sido más tradicional en la robótica debido a su rendimiento, mientras que Python ha ganado popularidad en la comunidad de ROS2 por su facilidad de uso y flexibilidad. En muchos casos, la combinación de ambos lenguajes puede ser una opción válida, permitiendo aprovechar las ventajas de cada uno para tareas específicas del sistema robótico o aplicación.

Tanto los paquetes en C++, como los paquetes en Python deben contar con un contenido mínimo. Para el caso de C++, debe contar con los archivos *CMakeList.txt* y *package.xml*, que describen las dependencias del proyecto y sus metadatos. Luego, debe contar con un directorio *include*, que contenga un directorio con el nombre del paquete. En dicho directorio se encontrarán los *headers* de los nodos del paquete. La implementación de los nodos se debe encontrar en el directorio *src*.

Para el caso de Python, los paquetes deben contar con el archivo *package.xml*, conteniendo la meta-data del paquete. Deberá contar con dos archivos con el nombre *setup*: uno de tipo *cfg* que se utiliza cuando en el paquete hay ejecutables y otro de tipo *py* que contiene las instrucciones de cómo instalar el paquete. A su vez, se deberá contar con un directorio con el mismo nombre del paquete. Este debe contener un archivo vacío llamado *__init__.py* y la implementación de los nodos del paquete. Por último, debe contener un directorio llamado *resources*, que contenga un archivo -sin tipo ni contenido- con el nombre del paquete.

```
my_c++_package
├── package.xml
├── CMakeList.txt
├── include
│   └── my_c++_package
│       ├── node1.hpp
│       ├── ...
│       └── nodeN.hpp
└── src
    ├── node1.cpp
    ├── ...
    └── nodeN.cpp
```

```
my_pyhton_package
├── package.xml
├── setup.cfg
├── setup.py
├── resource
│   └── my_pyhton_package
└── my_pyhton_package
    ├── __init__.py
    ├── node1.py
    ├── ...
    └── nodeN.py
```

Estos son los requerimientos mínimos para los paquetes de los distintos lenguajes. Existen otros directorios usados comúnmente, pero varían según las necesidades del paquete. Por ejemplo, se puede tener un directorio con el nombre *launch*, el cual contenga los *launchfiles*. Otro directorio comúnmente usado es el directorio *config* para almacenar los archivos de configuración.

5.1.1.4. Frames y TFs

Los marcos de referencia (*frames*) son sistemas de coordenadas que se utilizan para ubicar y describir la posición y orientación de distintos objetos (plataforma robótica, sensores, actuadores, escena). Cada marco de referencia se define por un origen y una orientación específica en el espacio tridimensional. Estos marcos permiten establecer un punto de referencia con el cual se puede medir y relacionar las posiciones y orientaciones de otros elementos del sistema. Además, con ellos se pueden representar diversas partes de un robot, como la base, las articulaciones, extremidades o cualquier componente específico. Es decir, se pueden utilizar para representar la posición y orientación de sensores, objetos del entorno o cualquier otro elemento para el funcionamiento del robot.

Las Transformaciones de Coordenadas (TFs) son una herramienta provista por ROS2 para el desarrollo de aplicaciones robóticas. Permiten representar y gestionar la posición y orientación relativa de distintos elementos dentro de un sistema robótico. Se utilizan para describir las relaciones espaciales entre diferentes marcos de referencia como se ve en la Figura 5.5, allí se ilustra un ejemplo donde se definen diferentes marcos de referencia para un robot con sus TFs correspondientes.

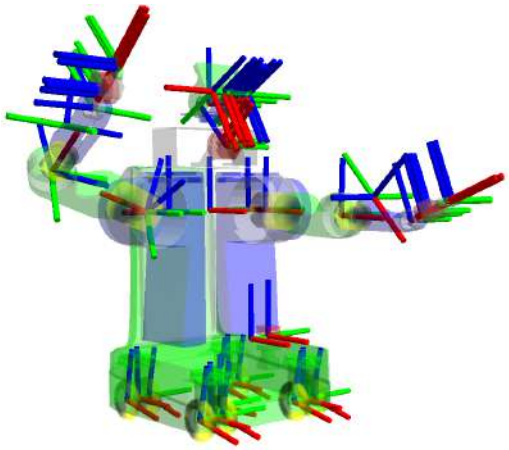


Figura 5.5: Ejemplo de uso para el paquete TF2. Extraída de [2].

Con el uso de TFs se logra representar y mantener marcos de referencia mediante un grafo, donde cada marco de referencia se identifica con un nombre único. Esto permite rastrear cómo se relacionan entre sí estos marcos a lo largo del tiempo, incluso cuando el robot se mueve o cambia su configuración. En el ejemplo de la Figura 5.6 se observa el grafo que se obtiene para dos robots (turtle1 y turtle2) que se relacionan con el mismo marco de referencia global "world"

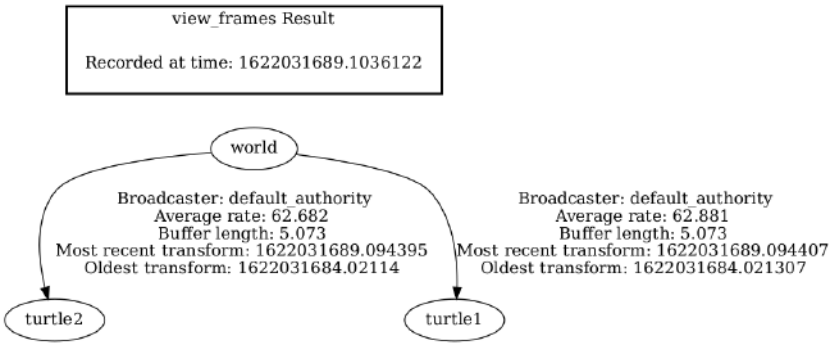


Figura 5.6: Grafo de interacción entre los marcos de referencia de dos Turtlebots. Extraída de [2].

Otra función de las TFs es la transformación de datos. Cuando los sensores del robot generan datos, estos están referidos a su propio marco de referencia local. Para utilizar estos datos de manera conjunta en una aplicación, es necesario transformarlos a un marco de referencia global común. Las TFs permiten realizar estas transformaciones, lo que facilita el procesamiento y la interpretación de los datos en el marco global del robot.

5.1.1.5. Comunicación

Como se vio en la Sección 5.1.1.1, ROS2 utiliza DDS como intermediario para la comunicación. A su vez, en la Sección 5.1.1.2 se vieron los distintos mecanismos

que se puede utilizar para generar comunicación entre nodos. A continuación se presentan conceptos relacionados al funcionamiento y configuración de las comunicaciones en ROS2.

5.1.1.5.1. QoS

Gracias al uso de DDS para la comunicación, ROS2 ofrece políticas de Calidad de Servicio (QoS, por su siglas en inglés), que permiten configurar cualquiera de los tres métodos de comunicación descritos anteriormente. Con la configuración adecuada, la comunicación puede ser por ejemplo tan confiable como TCP o tan *best-effort* como UDP. Estas opciones de configuración brindan la capacidad de adaptar ROS2 a las necesidades específicas de cada proyecto. En la Tabla 5.1 se presentan las diferentes configuraciones que se pueden adoptar para la comunicación.

A continuación se presenta una lista de las políticas configurables y los posibles valores que pueden tomar.

Politica	Valor	Descripción
History	KeepAll	Almacena todos los mensajes, sujeto a los recursos del sistema.
	KeepLast	Almacenan los últimos N mensajes, donde N está dado por la política Depth
Depth	(int)	Cantidad de mensajes a almacenar en caso de que la política History tenga el valor KeepLast
Reliability	Reliable	Garantiza la comunicación, reintentando hasta lograrla
	Best-Effort	Envía el mensaje, sin preocuparse por que sea recibido
Durability	TransientLocal	Para el caso de comunicación mediante tópicos, el publicador es responsable de persistir los mensajes en caso de que hay un suscriptor tardío
	Volatile	No se persiste ningún mensaje
Deadline	(int)	Tiempo máximo esperado entre mensajes. Si un mensaje no se publica dentro del plazo definido, puede considerarse obsoleto o no válido
Lifespan	(int)	Tiempo máximo entre el envió de un mensaje y la recepción del mismo. Si se supera dicho plazo, el mensaje es descartado
Liveliness	Automatic	El sistema considera que todos los publicadores de un nodo siguen vivos durante un período de tiempo (determinado por la política Lease Duration) cada vez que uno de los publicadores del nodo publica un mensaje. Esto significa que si al menos uno de los publicadores sigue activo, el sistema considerará que todos los demás también lo están
	Manual	El sistema considera que un publicador sigue vivo si el propio publicador lo afirma manualmente. El publicador debe indicar explícitamente su vitalidad periódicamente para que el sistema lo considere activo
Lease Duration	(int)	Tiempo máximo que puede pasar sin actividad un publicador para seguir siendo considerado activo

Tabla 5.1: Políticas QoS y sus posibles valores.

5.1.1.5.2. DomainIds y namespaces

Los identificadores de dominio (*DomainIds*) son un mecanismo de ROS2 utilizado por los DDS para subdividir la red. Solo los nodos que se encuentren en el mismo *DomainId* podrán detectarse y comunicarse entre ellos. Esto es de gran utilidad para aislar y así evitar interferencias entre distintos sistemas que usan una misma red.

Los espacios de nombres (*namespaces*) en ROS 2 permiten organizar los nodos, tópicos, servicios y otros recursos de manera jerárquica. Esto facilita la administración y el desarrollo de sistemas robóticos complejos al evitar conflictos de nombres y permitir una estructura organizada.

Para comprender mejor los conceptos, se presenta el siguiente ejemplo: supongamos que se tienen dos robots iguales. Cada robot cuenta con un nodo que se subscribe al tópico */cmd_vel*, en el cual se reciben velocidades que se utilizan para controlar las velocidades de los motores. A su vez, cada robot cuenta con un LiDAR que publica medidas de distancias sensadas al tópico */scan*. Al estar ambos robots suscritos al mismo tópico de velocidad, si se envía una cierta velocidad a uno de ellos, el otro también la recibirá. Por otro lado, ambos publican las lecturas de sus LiDARs al mismo tópico, sin diferenciar de quién es la lectura. Esto se puede ver como un problema para una solución donde se busque independencia entre los distintos robots.

Este problema se puede solucionar utilizando alguno de los métodos mencionados previamente. En estos casos la elección depende de la situación y los resultados que se busquen.

Si lo que se busca es que ambos robots sean independientes y no sepan de la existencia del otro robot, se deberán usar distintos *DomainIds* para cada robot. De esta forma, cada uno ejecuta un entorno de comunicación del DDS.

Por otro lado, si lo que se quiere es que cada robot tenga su tópico de velocidad y su tópico del LiDAR, manteniendo independencia pero pudiendo comunicarse entre sí, la solución será el uso de *namespaces* diferentes para cada robot. Con esta solución, cada robot usaría el tópico */namepsace/cmd_vel* para su velocidad y */namespace/scan* para las medidas sensadas, donde 'namespace' podría ser 'robot1' y 'robot2' respectivamente.

5.1.1.5.3. Paralelización y Sincronismo

Cada nodo en ROS2 es ejecutado utilizando un ejecutor (Executor), los cuales se encargan de utilizar uno o más threads[1] para ejecutar los distintos callbacks[2]. Se tienen tres tipos de ejecutores: SingleThreadedExecutors, StaticSingleThreadedExecutors y MultiThreadedExecutors.

El más simple, y usado por defecto, es el **SingleThreadedExecutor**. Como su nombre indica, utiliza un único thread para la ejecución del nodo, procesando los mensajes recibidos y los eventos de a uno.

[1] Hilos de ejecución

[2] función que se ejecuta tras un evento, por ejemplo cuando un suscriptor recibe un mensaje en el tópico

El **StaticSingleThreadedExecutor** también cuenta con un solo thread. A diferencia del ejecutor anterior, este analiza la estructura del nodo solo una vez, al inicializarse. Esto brinda optimizaciones en el tiempo de ejecución del nodo, privándolo de poder modificar su estructura una vez finalizada la inicialización. Es decir, pierde la posibilidad de, en tiempo de ejecución, suscribirse a tópicos, crear servidores de servicios o de acciones, entre otros.

A diferencia de los dos anteriores, el **MultiThreadedExecutor** permite configurar la cantidad de threads que se quieren utilizar para procesar múltiples mensajes o eventos en paralelo.

En ROS2, el procesamiento de los mensajes y eventos se realiza mediante *callbacks*, los cuales se agrupan en *CallbackGroups* con el fin de brindar cierto nivel de control sobre la paralelización de dichas llamadas. Existen dos tipos de *CallbackGroups*: mutuamente exclusivos y reentrantes. Los *callbacks* que se encuentren en un mismo grupo mutuamente exclusivo no podrán ejecutarse en paralelo, a diferencia de los que pertenecen a un grupo reentrante. Los *callbacks* que se encuentren en distintos grupos siempre se podrán ejecutar en paralelo, independientemente del tipo de cada grupo.

5.1.2. SLAM

La localización y mapeo simultáneo o SLAM por sus siglas en inglés (Simultaneous Location And Mapping) es una técnica que tiene como objetivo que un robot construya y actualice un mapa de su entorno en tiempo real a medida que se desplaza a través de él, sin la necesidad de información de localización externa. En particular, el modelo del entorno que se busca generar podría ser a priori conocido o desconocido, pero es en los entornos desconocidos donde se encuentra el mayor interés y los mayores desafíos.

En otras palabras, SLAM aborda el problema de cómo un robot puede construir un mapa y al mismo tiempo conocer su propia ubicación dentro de ese mapa, utilizando sólo información sensorial (por ejemplo, datos de cámaras, LiDARs, odometría, etc.). Como se puede presumir, es una técnica esencial en la robótica y especialmente en sistemas de navegación autónomos ya que permite crear un modelo del entorno y al mismo tiempo relacionarlo con el robot. Para un robot autónomo el poder crear un modelo y relacionarse en él facilita y abre las puertas para el desarrollo de funcionalidades aún más complejas como la navegación autónoma, la detección de objetivos, entre otros.

Existen diferentes formas de realizar SLAM, ya sea variando los métodos de sensado, el procesamiento de la información, la representación del mapa, entre otros. La elección del tipo de SLAM a utilizar depende de la solución buscada, relacionándose fuertemente con el grado de detalle que se busque representar y los elementos sensoriales con los que se cuente [3][47]. Por ejemplo, para la representación del mapa se pueden usar tanto mapas basados en características (*landmarks*) como grillas de ocupación. En el primer método se utilizan rasgos distintivos en el mapa para ubicarse y almacenar información del entorno con el fin de representar el entorno, por ejemplo siguiendo el ejemplo que se muestra en la Figura 5.7,

un conjunto de árboles, una casa, una rotonda o algún otro elemento del entorno distintivo. Estos modelos en general son menos demandantes computacionalmente pero al mismo tiempo generan una representación del entorno más rudimentaria. Además, en general se debe conocer de antemano las características del entorno para lograr clasificar estos puntos de interés. En estos casos el mapa se puede pensar como un vector que en cada entrada se tiene las coordenadas y orientación relativa del *landmark* correspondiente.

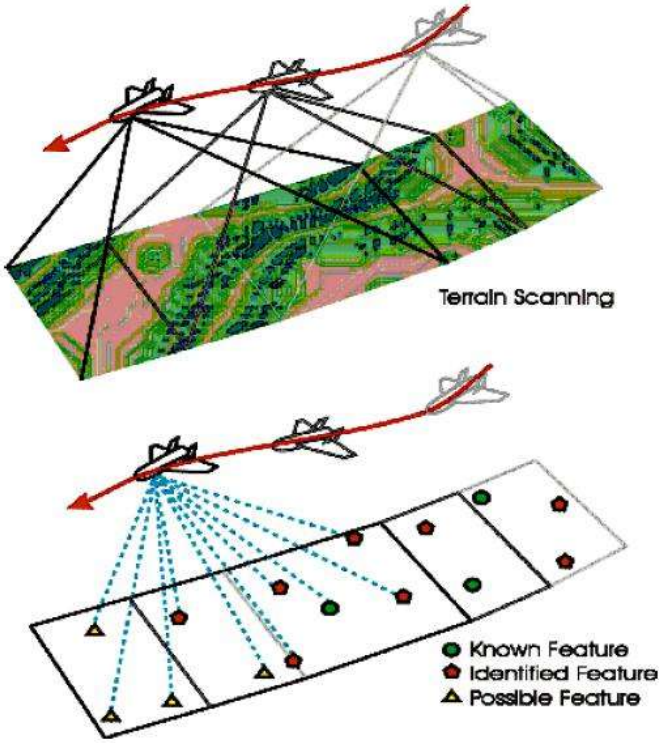


Figura 5.7: Ejemplo de SLAM utilizando características. Extraída de [48]

Por otro lado, los mapas que se basan en grillas de ocupación como el que se muestra en la Figura 5.8 son una representación del entorno por medio de una matriz. En esta matriz cada entrada se refiere a una posición en el mapa donde el valor de la celda almacena la probabilidad de que esa región del espacio se encuentre ocupada. En general, esa probabilidad se compara con un valor umbral y dependiendo del valor se etiqueta como una celda ocupada, libre o desconocida, colores negro y blanco y gris de la figura respectivamente. En contraposición con los mapas basados en características, las grillas de ocupación resultan en mayor demanda computacional pero logran representar obstáculos de manera arbitraria.

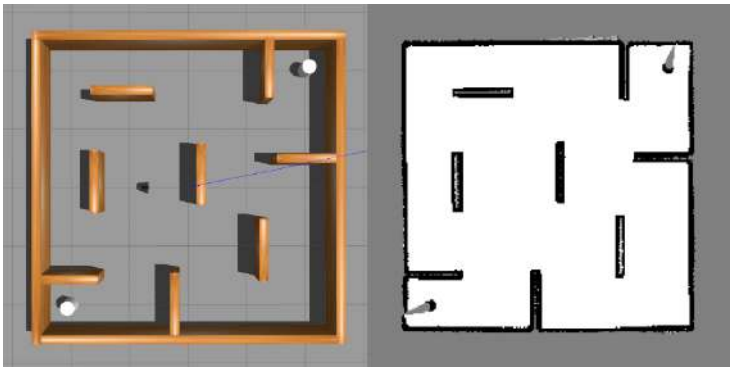


Figura 5.8: Ejemplo de SLAM utilizando grillas de ocupación. Extraído de [49].

Además de generar una representación con mayor grado de detalle, las grillas de ocupación son un formato de mapa que logra interpretarse de manera más evidente por un ser humano, mientras que una representación por características puede, en algunos casos, tornarse complejo de interpretar. En este proyecto se trabaja con un algoritmo de SLAM que representa el entorno en el formato de grillas de ocupación, debido al hardware con el que se cuenta y la experiencia previa. La descripción del algoritmo utilizado se encuentra disponible en la Sección 5.2.3.

5.1.3. Navegación

La navegación representa una habilidad fundamental que permite a los robots planificar y ejecutar rutas hacia objetivos específicos. Este proceso no solo involucra la determinación de la mejor manera de llegar a un destino, sino que también abarca la construcción de mapas del entorno y la capacidad de situarse con precisión en él. Es decir, se puede considerar que SLAM se engloba dentro del concepto de navegación como uno de los pilares fundamentales. En la Figura 5.9 se ilustran los componentes de la navegación.

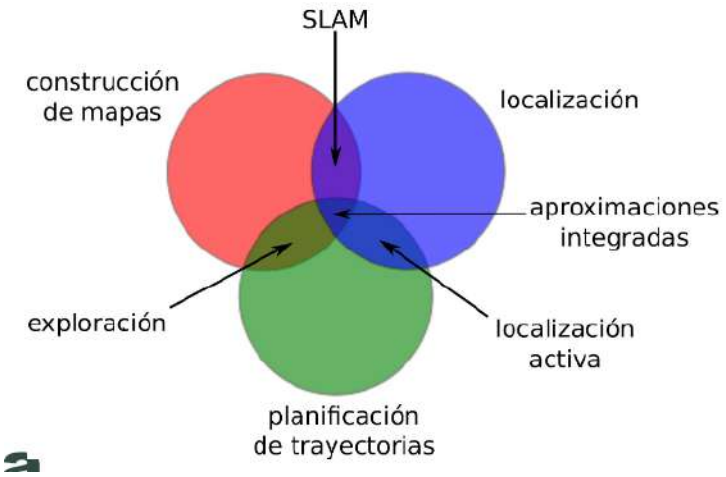


Figura 5.9: Elementos de la navegación. Extraído de [50].

La navegación se puede resumir como la respuesta a las siguientes cuatro preguntas:

1. ¿A dónde debo ir?
2. ¿Dónde estuve?
3. ¿Dónde estoy?
4. ¿Cómo llegar?

La primera pregunta involucra la determinación del destino. Sin embargo, este aspecto en sí no suele ser considerado como parte integral del problema de navegación, la cual se enfoca más en la planificación de cómo llegar al destino y no en la elección del destino en sí. La selección del destino suele ser determinada por input humano o un planificador de misión. En el caso de los robots autónomos esta pregunta se responde con el planificador de misión, un algoritmo que determina cual es la tarea u objetivo que se tiene en determinado momento. Más información sobre la implementación se encuentra disponible en la Sección 5.2.5

La segunda pregunta se relaciona con la construcción de mapas del entorno y en conjunto con la pregunta tres se responden aplicando técnicas de SLAM. Los robots deben ser capaces de explorar y cartografiar su ambiente circundante. Esta tarea implica abordar entornos desconocidos y adaptarse a cambios en el mismo, como la reconfiguración de obstáculos o la alteración de la disposición de los objetos.

La tercera pregunta resalta la necesidad de una precisa localización robótica. Para que un robot pueda seguir una trayectoria planificada o construir un mapa preciso, es esencial que pueda determinar su posición con respecto al entorno. Esto puede lograrse a través de diferentes métodos, como la localización relativa, basada en coordenadas topológicas o incluso en coordenadas absolutas como latitud, longitud y altitud. En general se utilizan técnicas de fusión señorial entre datos de odómetro e información de los sensores para estimar las posiciones del robot como los que se describen en [51].

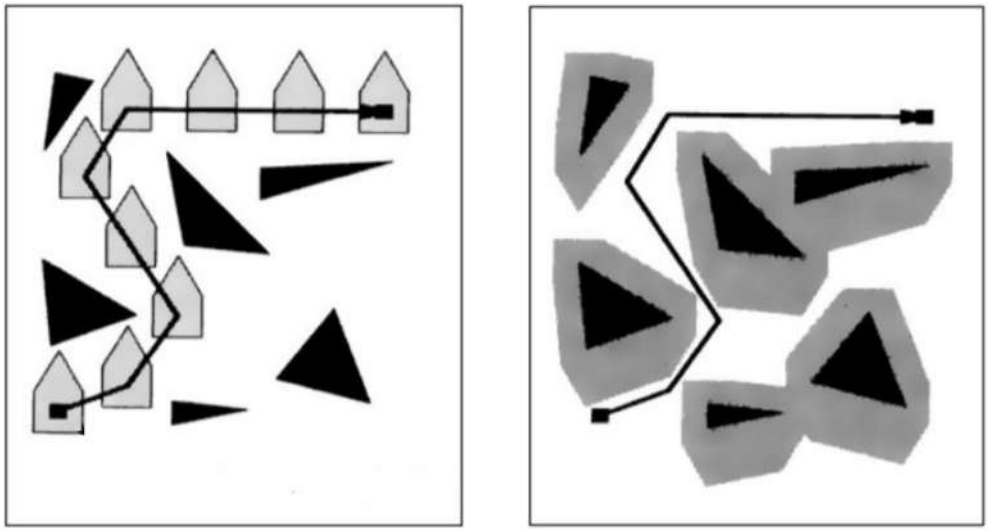
La última pregunta destaca la importancia de la planificación de trayectorias. Esta área se concentra en desarrollar métodos para trazar las rutas óptimas seguras para que un robot logre alcanzar su objetivo. La planificación de trayectorias se subdivide en enfoques que se centran en relaciones cualitativas y enfoques métricos que consideran medidas cuantitativas. En la Sección 5.1.3.3 se especifica más sobre el concepto de la planificación de trayectorias, pero previo a eso se introduce el concepto de espacio de configuración en la siguiente sección (5.1.3.1).

5.1.3.1. Espacio de configuración

Para poder definir e introducir el concepto de espacio de configuraciones es necesario definir el concepto de configuración. La configuración de un robot en un instante dado, q, se define como la especificación del conjunto de todas las variables que definen la posición y orientación del robot en dicho momento. El espacio de configuración de un sistema es el espacio de todas las configuraciones posibles, representa todas las posibles posiciones y estados que el robot puede alcanzar [52].

En la Figura 5.10 se ejemplifica la definición de espacio de configuración y como se puede determinar el espacio de configuración a partir del espacio de trabajo. En

la Figura 5.10a se muestra un robot que se desplaza a través de un entorno con obstáculos. Como se puede ver, este robot posee restringido su movimiento por los obstáculos, su orientación y también posición. En la Figura 5.10b se construye el espacio de configuración, en este caso se considera al robot como un punto pero se "agrandan" los obstáculos considerando las características del robot.



(a) Espacio de trabajo (b) Espacio de configuración

Figura 5.10: Ejemplo de obtención del espacio de configuración a partir del espacio de trabajo. Extraído de [53].

5.1.3.2. Mapas de Costos

Los mapas de costos son la representación en grillas de ocupación del espacio de configuración. Un mapa de costos se representa como una matriz donde cada una de las celdas representa el costo asociado a una determinada posición en el plano. Es decir, cada celda o punto en el mapa de costos está asociado con un valor que indica cuán peligroso es para el robot navegar por esa región. Este costo viene dado por la probabilidad de que en dicha posición exista o no un obstáculo.

En general se utilizan dos mapas de costo: un mapa global y otro local, uno para cada nivel de planeación como se menciona en la Sección 5.1.3.3. Estos mapas, se construyen con la información proveniente de los algoritmos de SLAM, de las características físicas del robot e incluso de otros sensores. A nivel global el mapa de costo contiene información sobre el ambiente y en general se restringe solamente a utilizar información proveniente del SLAM. Por otro lado, el caso local se restringe a una porción reducida del entorno alrededor del robot, es un mapa que podría contar con entradas adicionales a las provenientes únicamente por el SLAM debido a que los robots podrían aportar información sensorial local como con sensores de ultrasonido. Si bien no es el caso de nuestra implementación, en el mapa de costos local se podrían agregar los sensores de proximidad como se mencionó en la Sección 2.2.2 para tener en cuenta obstáculos en diferentes alturas.

En cada uno de los mapas de costos, cada una de las celdas pueden tener un valor entre 0 y 255. Aquellas celdas que tengan costo 0 serán consideradas libres, mientras que las que tengan un valor distinto de 0 tendrán un grado de ocupación. Las celdas que no tengan un valor de costo asociado serán consideradas desconocidas.

Siguiendo el análisis que se realiza en [54], se introduce en la Figura 5.11 un diagrama donde se representa a un robot genérico y el costo asociado a cada celda en función de la distancia de su objeto más próximo. En la representación del robot se considera en negro la celda central del robot, es decir la que contiene al centro del mismo. En rojo se puede ver la silueta del robot, mientras que en azul se ven dos circunferencias: la circunferencia inscrita y la circunscrita. En la figura se introduce el concepto de inflación, es decir asignarle un costo determinado y mayor a cero a celdas que no están ocupadas, pero que a priori se busca evitar o desincentivar el transitar por ellas ya que se estaría cerca de una posible colisión.

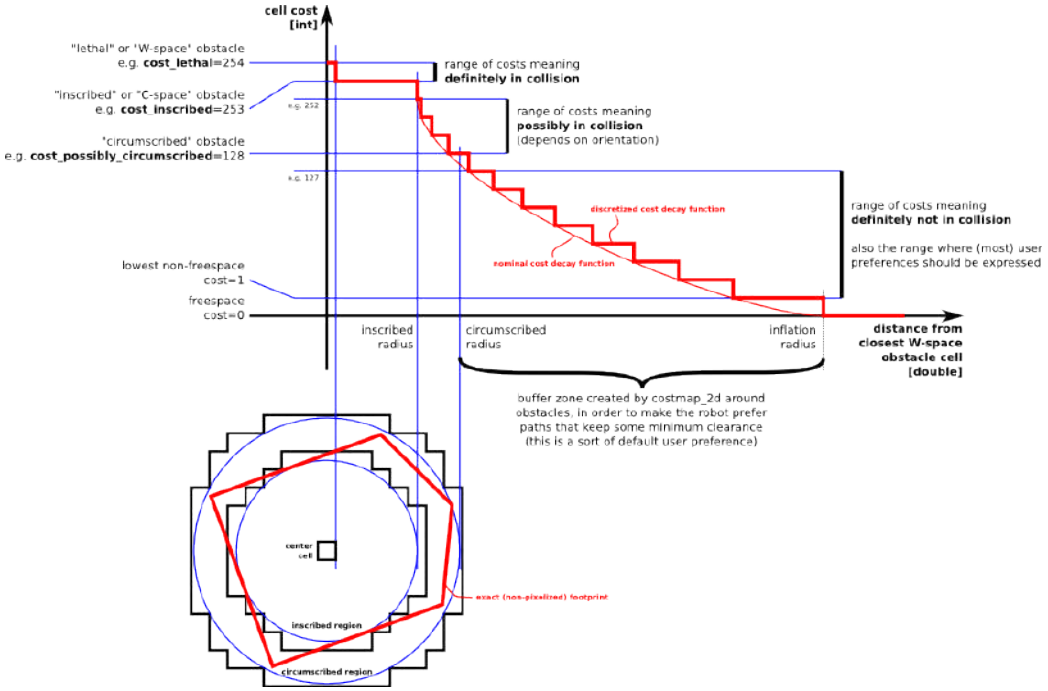


Figura 5.11: Costo asociado a una determinada celda en función de la distancia al objeto más próximo. Extraído de [54].

Se define también el concepto de celda letal. Una celda se considera letal cuando hay un obstáculo y por ende una colisión asegurada, para todos las celdas dentro de la circunferencia inscrita también habrá colisión asegurada y se clasifican también como celdas letales. De manera análoga, se define el concepto de celda inscrita para todas aquellas celdas que se encuentren en la región entre la circunferencia inscrita y la circunscrita de distancia de un objeto. Para este tipo de celda existe una colisión posible, dependiendo de la orientación del robot y por lo tanto un costo elevado.

Finalmente, se asignan costos distintos de cero para determinadas celdas que aunque no se encuentren dentro de la zona de colisión se pretende desincentivar el tránsito por ellas. La idea detrás de esto es desalentar a que el robot circule por las áreas cercanas a los obstáculos. En la Figura 5.11 se puede ver la curva de decaimiento del costo en función de la distancia. Tanto el radio de inflación como el factor de decaimiento son parámetros configurables.

Si analizamos el caso particular de este proyecto, dado que se trabaja con un robot circular y omnidireccional la zona de colisión posible no existe y por ende se cuenta solamente con la zona de colisión asegurada. Es en estos casos donde el margen de seguridad dado por la capa de inflación cobra vital importancia ya que de no considerarlo se podría pretender circular de manera tangencial a los objetos. Práctica que claramente se debe evitar a la hora de trabajar. Otra de las razones por la cual se genera esta zona de inflación alrededor de la celda ocupada es para ganar robustez e intentar encaminar el tránsito del robot por zonas alejadas de obstáculos.

5.1.3.3. Planificación de Trayectorias

Dada la complejidad del problema de planificación de trayectorias, es común utilizar una arquitectura en dos capas para plantear la solución como en [55]. En la Figura 5.12 se ilustra el diagrama de bloques que componen el algoritmo implementado en "move_base".

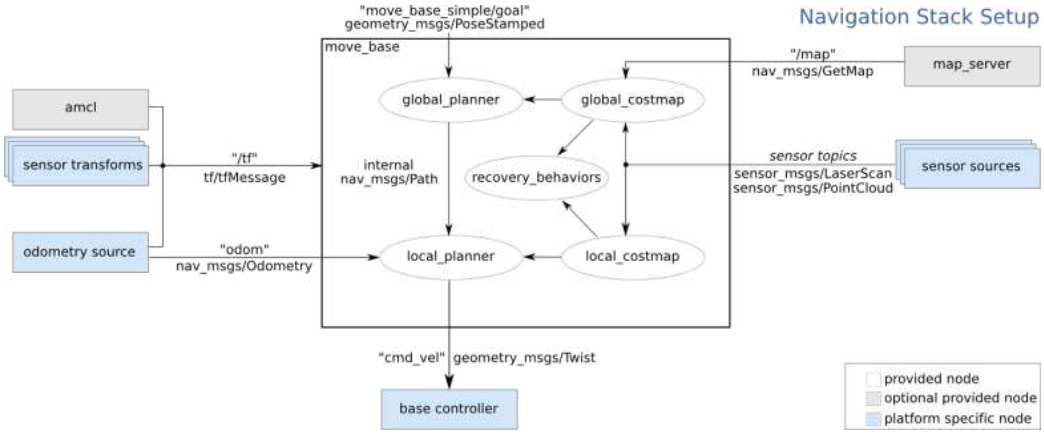


Figura 5.12: Diagrama comportamental de alto nivel de la implementación de "move_base" [55]

Si bien en la figura anterior se ilustra todo el algoritmo implementado para realizar la tarea de navegación, en esta sección se busca describir el trabajo de el "Global Planner" y el "Local Planner". Estas son las dos capas utilizadas para resolver el problema de la planificación de trayectorias.

La capa superior, "Global Planner", tiene como entrada el mapa de costos global y la posición objetivo. Con esta discretización el "Global Planner" construye un grafo que representa el camino desde el punto inicial hasta el objetivo.

Una particularidad de los planificadores globales o de alto nivel es que en general generan una solución sin tener en cuenta la cinemática ni la dinámica del robot, transformando el problema inicial en la búsqueda de caminos óptimos a través de grafos. El planificador global, con sus entradas construye un plan de alto nivel que sirve como entrada para el planificador local. Este plan está dividido en objetivos parciales con los cuales el planificador local puede trabajar.

El "Local Planner" se encarga de calcular las velocidades relativas para el robot. Se envían instrucciones a los actuadores del robot para moverse hacia cada uno de los objetivos parciales, dados por el Global Planner. Para ello en vez de trabajar con el mapa de costos global, se trabaja con una porción más pequeña (una discretización local) pero más precisa del mismo. El objetivo principal de esta capa es lograr alcanzar objetivos incrementales de forma óptima sin colisionar. Es por ello que el "local Planner" tiene como entrada información sensorial con mayor cadencia e incluso datos sensoriales crudos como el caso de la odometría.

5.1.4. Exploración colaborativa

La exploración colaborativa se refiere al proceso en el cual múltiples robots trabajan juntos de manera coordinada para explorar y mapear un entorno desconocido o poco familiar. En lugar de que cada robot opere de manera independiente, la exploración colaborativa busca aprovechar las capacidades y recursos de varios robots para lograr una exploración más eficiente, completa y precisa. Los sistemas con flotas de robots pueden ser centralizados o descentralizados, como se observa en la Figura 5.13.

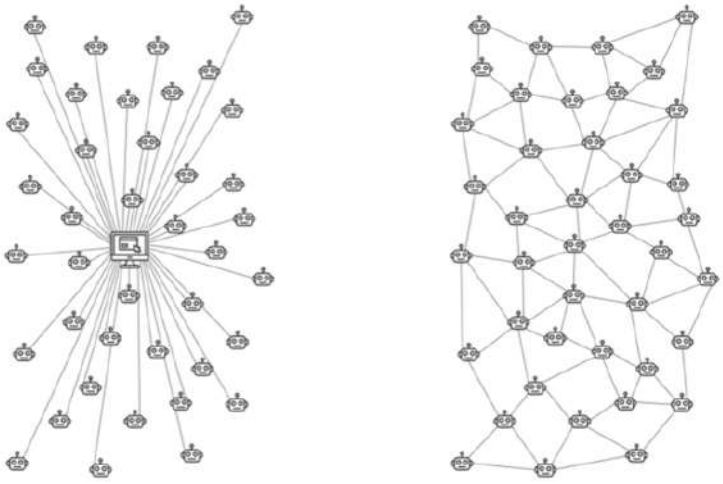


Figura 5.13: Diagrama de un sistema centralizado (izquierda) y uno descentralizado (derecha). Extraído de [56].

Como se ilustra en la imagen, las flotas centralizadas, flota de la derecha, se caracterizan por tener a un robot u otro sistema externo que actúa de coordinador. Cada robot se comunica únicamente con esta unidad central y esta se encarga de la sincronización y coordinación de la flota. Por otro lado, en los métodos descentralizados, flota de la izquierda, los robots se comunican entre sí para intercambiar información y coordinar sus acciones, pero no dependen de una unidad central para tomar decisiones. Cada robot en una flota descentralizada tiene cierta autonomía y capacidad de toma de decisiones, eliminando el riesgo de tener un punto de falla único. Esto permite una mayor flexibilidad y robustez en situaciones cambiantes o en entornos con posibles fallas de comunicación.

En ambos casos, se busca que los robots estén sincronizados y coordinados para evitar colisiones, reducir la ineficiencia en la exploración y maximizar la cobertura del área a explorar.

5.2. Solución Implementada

En esta sección se presenta la solución de software implementada. Como ya se mencionó al lo largo del documento y en particular al comienzo de este capítulo, toda la aplicación de alto nivel se construyó sobre la base de ROS2, en particular la distribución Humble[3], para lograr el mapeo colaborativo descentralizado. En la Tabla 5.2 se pueden apreciar las versiones de algunos de los otros componentes del sistema. Se proporcionará una breve explicación sobre cada uno de los paquetes utilizados, destacando también nodos específicos y relevantes para el funcionamiento del sistema. Es importante mencionar que solo se tomará en consideración la solución no simulada, ya que se dedica el Capítulo 6 al ambiente simulado.

	Robots	Ambiente de simulacion
OS	UbuntuMate 22.0	Ubuntu 22.04
ROS2	Humble	Humble
DDS	Eclipse Cyclone	Eclipse Cyclone
Pyhton	Python 3.10	Python 3.10
C++	11.4	11.4

Tabla 5.2: Versiones utiilizadas en los robots y en los ambientes de simulación.

Se comienza la sección introduciendo ciertas consideraciones a tener en cuenta cuando se trabaja en un entorno con múltiples robots en ROS2 en la misma red en la Sección 5.2.1. Luego de esta sección se describe cada componente de la solución en detalle.

Como se puede ver en el diagrama de la Figura 5.14, la aplicación puede describirse en función de 4 bloques de alto nivel. En esta sección se describirán cada uno de ellos, se mencionan detalles de implementación y configuraciones que se deben tener en cuenta a la hora de trabajar con cada uno de los bloques. Se comenzará

[3]Humble es el nombre que lleva la última versión LTS de ROS2 [2]

describiendo el bloque "ICRE Bringup", en él se engloba todo lo relacionado los sensores y actuadores, así como también configuraciones inherentes a la construcción del robot. Luego se describe en la Sección 5.2.3 el paquete utilizado para realizar las tareas de SLAM. Se sigue explicando el algoritmo de navegación y el paquete utilizado en la implementación en la Sección 5.2.4. Finalmente en la Sección 5.2.5, se describe el método utilizado para determinar los objetivos y como entre los múltiples robots se asignan las tareas.

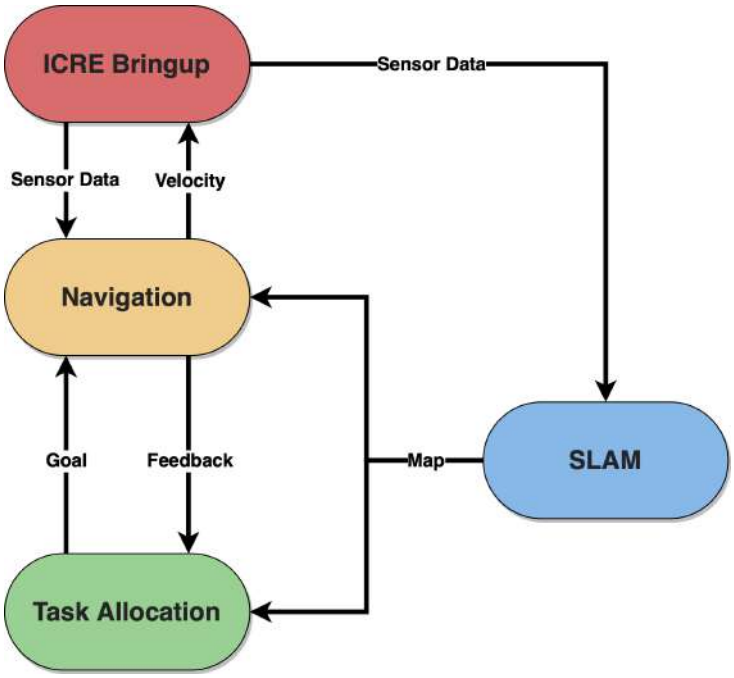


Figura 5.14: Diagrama de alto nivel de la solución implementada

5.2.1. Multi-robot

Al trabajar con múltiples robots, se deben tener en cuenta una serie de configuraciones propias de ROS2 para asegurar el correcto funcionamiento de cada robot. Como se explicó en la Sección 5.1.1.5.2 a la hora de trabajar con múltiples robots existen dos diferentes técnicas provistas por ROS2 para que la información de un robot en particular no llegue hacia los otros o se logre diferenciar entre robots. Dependiendo del objetivo de la aplicación se puede optar por utilizar diferentes identificadores de dominios o espacios de nombres para los diferentes robots. En el caso de este proyecto, donde se busca mapear de manera colaborativa es natural pensar en la solución con *namespaces*.

Para lograr esta configuración se comenzó incorporando el uso de *namespaces* en los paquetes y nodos creados. A su vez, se corroboró que los paquetes externos pueden utilizar dicha funcionalidad. Se descubrió en este proceso que algunos nodos hacen uso de nombres absolutos en ciertos tópicos. Por ejemplo, algunos nodos del paquete "Nav2" se suscriben al tópico absoluto */tf*, en lugar de al tópico

relativo *tf*. Esto generó que, aún cuando se utiliza un *namespace*, el tópico igualmente permanezca */tf*. Por ejemplo, si se toma el caso del tópico *odom* relativo, al usar un *namespace* como "namespace1", este pasaría a ser */namespace1/odom*. Para solucionar esto se utilizaron reglas de *remapping*, convirtiendo los tópicos absolutos a relativos. Con esto se obtiene que cada robot posea todos sus tópicos y árboles de transformadas relativos a su *namespace* y por ende se logra diferenciar la información para que llegue a su correcto destinatario.

Vale la pena mencionar que a partir de este momento se refiere a los tópicos obviando la componente dada por el *namespace* por simplicidad.

5.2.2. ICRE Bringup

En esta sección se describe la implementación del bloque "ICRE Bringup". Como se puede ver en la Figura 5.15, este bloque se encarga de proveer las transformadas al ambiente de desarrollo y también se encarga de proveer una interfaz para enviar y recibir información de los sensores y actuadores del robot.

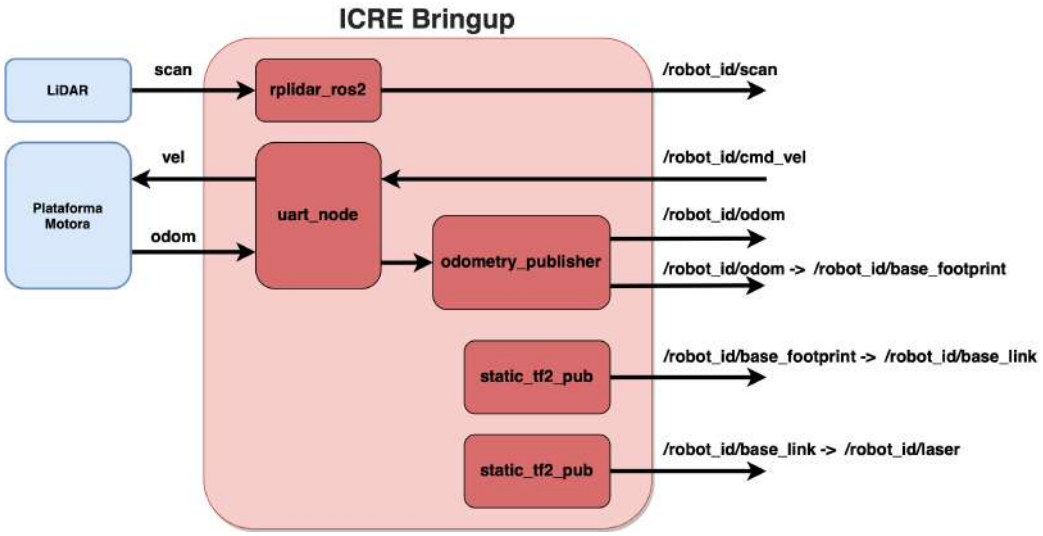


Figura 5.15: Descripción de ICRE Bringup

A continuación se describe en detalle cada uno de los componentes del bloque "ICRE Bringup", también se introduce un Launchfile con el cual es posible ejecutar todo este bloque a la vez.

5.2.2.1. Manejo del LiDAR

5.2.2.1.1. Paquete rplidar_ros2

Con el fin de obtener medidas del LiDAR en el ambiente de ROS2 se utiliza el paquete *rplidar_ros2* desarrollado por Slamtec [57] [58] que es compatible con el LiDAR que se utiliza. Dicho paquete cuenta con un launchfile que se encarga de levantar los nodos necesarios para tomar las medidas del láser y publicarlas en el tópico */scan*.

En este tópico se publica para cada grado de los 360 una medida de distancia o un valor de error ('inf') en caso de no tener una medida válida para ese ángulo. Se puede obtener un error en la medida por diferentes causas, por ejemplo que el objeto más cercano en determinada dirección se encuentre a una distancia mayor que la del rango máximo, que se encuentre a menor distancia que el rango mínimo o que el material del obstáculo absorba la radiación y no permita la reflexión de la luz en su superficie, entre otras causas propias del sensor. En estos casos, el sensor no devuelve información útil a priori de la medida que se está tomando.

5.2.2.1.2. Nodo scan_filter

El nodo scan_filter tiene como objetivo agregar información a las medidas del láser que tengan como valor *'inf'*. Este nodo se subscribe a los mensajes que publica el LiDAR y cambia cada *'inf'* por el valor de distancia equivalente al rango máximo del LiDAR.

Este filtro asume que cada medida que se obtiene con el código de error *'inf'* se deben a que el obstáculo más próximo en la dirección del *'inf'* se encuentra a una distancia mayor al rango máximo del LiDAR.

En las secciones 7.1.1 y 7.2.2 se evalúan los resultados de utilizar este nodo en el entorno simulado y en el robot físico respectivamente. En la Sección 7.1.1 se detalla por qué se consideró utilizar este nodo. Mientras que en la Sección 7.2.2 se presentan las causas por las cuales no se utiliza en las pruebas físicas.

5.2.2.2. Nodo uart_node

El primer componente que se desarrolló completamente por el equipo buscaba generar una comunicación serial entre el Odroid y la ESP utilizando ROS2. El objetivo principal de este nodo era el de generar una interfaz para obtener datos de odometría y también enviar comandos de velocidad a la plataforma motora desde el ambiente de ROS2.

Para lograrlo, se creó el nodo *uart_node*, que cumple dos funciones fundamentales. La primera, enviar las velocidades recibidas en el tópico */cmd_vel* mediante conexión UART hacia la plataforma motora. La segunda, recibir a través de la misma conexión la odometría calculada en la ESP, como se menciona en la Sección 4, y publicarla en el entorno de ROS2.

En este nodo no se realiza el procesamiento de la odometría, sino que simplemente se publica el mensaje recibido desde la ESP en el tópico */uart_received_message*. Esta decisión sigue las buenas prácticas de ROS2, al mantener cada nodo con una única función claramente definida como una unidad computacional independiente.

5.2.2.3. Nodo odometry_publisher

Este nodo es responsable de suscribirse al tópico */uart_received_message* y realizar el procesamiento pertinente para publicar tanto la odometría en el tópico */odom*, como las transformadas en su tópico correspondiente como se observa en la Figura 5.15.

5.2.2.4. Launchfile icre_robot

Dentro del paquete icre_bringup se generó un launchfile que encapsula los dos nodos anteriores, el launch del rplidar y dos nodos extra. Estos nodos extra son los encargados de publicar una transformada estática cada uno: una de *base_footrpint* a *base_link* y otra de *base_link* a *laser*.

5.2.3. SLAM

En esta sección se introducen y describen algunas características del paquete utilizado con el que se realiza SLAM.

Se decidió utilizar el paquete *slam_toolbox* [59] desarrollado principalmente por Steve Macenski[4] pues es el recomendado por el *stack* de navegación de ROS2. Este paquete cuenta con una vasta selección de herramientas enfocadas en el mapeo y localización en mapas bidimensionales. Además, cuando se lo compara con otros algoritmos de SLAM como GMAPPING [61], KARTO [62] o HECTOR_SLAM [63], los resultados que se obtienen son iguales o incluso mejores que los de sus competidores siendo incluso más confiable y brindando mayores parámetros de configuración [64].

En la Figura 5.16, se pueden observar todos los componentes del paquete y cómo interactúan entre sí para lograr realizar el SLAM. El uso de cada uno de los bloques y su configuración se pueden determinar mediante un archivo de configuración de tipo yaml con el fin de adaptar el algoritmo a las características del entorno y del robot con el que se trabaje.

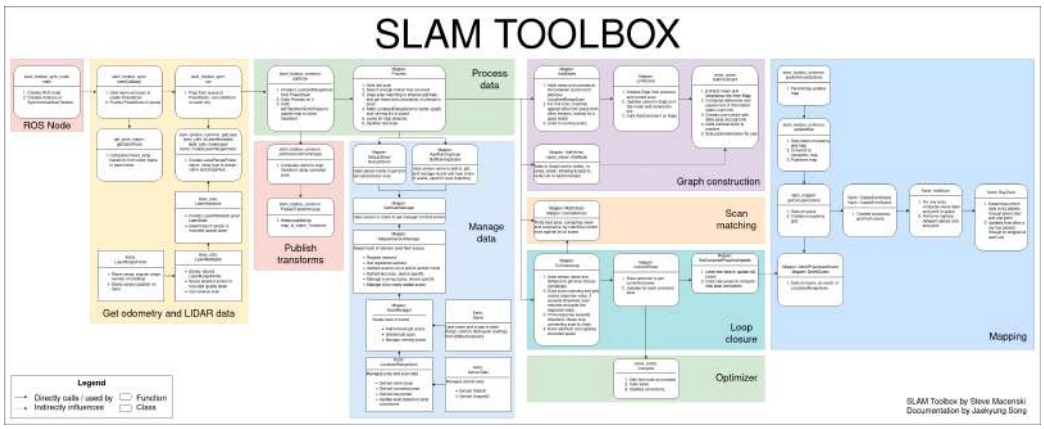


Figura 5.16: Diagrama de componentes de slam_toolbox. Extraído de [59].

Sin embargo, si bien este paquete brinda una excelente solución, fácil de integrar y configurar, no cuenta con ninguna herramienta para el mapeo multi-robot pues no se puede trabajar con *namespaces* ni posee la capacidad de localizar los datos del LiDAR.

[4]Dueño de OpenNavigation, responsable del desarrollo de las principales soluciones de navegación para ROS2 [60]

Es por ello que en lugar de utilizar la versión original, se utiliza una implementación modificada de slam_toolbox. Esta implementación es un *Fork* [5] del proyecto original, creada por Achala Athukorala[6].

En esta versión, en cada robot se ejecuta una instancia del slam_toolbox modificado, cada una bajo un *namespace* único. El paquete implementa una nueva interfaz de ROS2, denominada *LocalizedLaserScan*, que contiene: las medidas del LiDAR, la posición del LiDAR con respecto a la posición del robot que tomó la medida y la posición de dicho robot en el mundo. Como se aprecia en la Figura 5.17, cada robot publica dichos mensajes en un mismo tópico que se comparte por toda la flota. De esta forma, el mapa es generado por cada robot, usando los *LocalizedLaserScans* del resto de los robots.

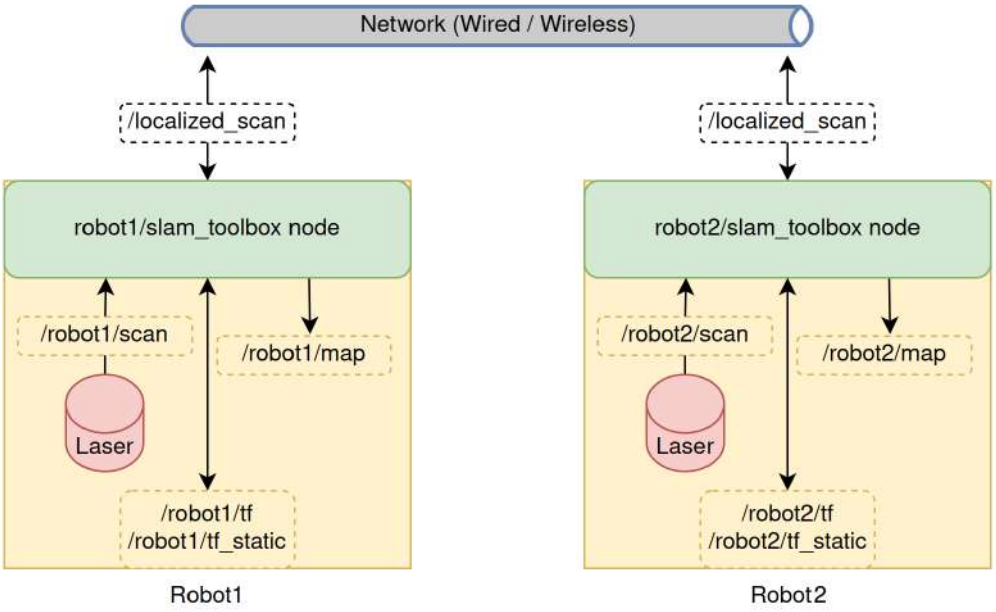


Figura 5.17: Diagrama de la arquitectura utilizada por slam toolbox multi robot. Extraído de [65].

Una limitación práctica de este paquete es que requiere que los robots se inicialicen posicionados y orientados de una manera determinada. Esto se debe a que la única forma que tiene el paquete de referenciar las posiciones de los robots entre sí es a través de scan_matching, y para que este funcione correctamente se necesitan esas dos condiciones.

[5] Fork es un término utilizado al trabajar con repositorios, en el que se realiza una copia del código, para utilizar de punto de partida de un nuevo proyecto.

[6] Ingeniero en robótica dedicado a la investigación en la Universidad de Tecnología y Diseño de Singapur (o SUTD por sus siglas en inglés)

5.2.4. Navegación

En esta sección se presenta la solución adoptada para resolver el problema de la navegación. Se detallan configuraciones realizadas y características de la solución.

Para cumplir con los objetivos de la navegación que se exponen en la Sección 5.1.3, se emplea el paquete Nav2 [66]. Este paquete representa la evolución del conjunto de herramientas de navegación para ROS, ahora adaptado para el entorno de ROS2. Proporciona una diversidad de algoritmos y utilidades que habilitan a los robots a planificar rutas, sortear obstáculos y desenvolverse de manera autónoma y segura en contextos en tiempo real, como se puede ver en su documentación [66].

Nav2 engloba varios tipos de servidores con el fin de cumplir con la navegación, destacándose entre ellos: planners, controllers y recovery. En la Figura 5.18 se ofrece una primera vista de la arquitectura de Nav2, así como también de los datos de entrada y salida que ofrece este paquete. Es importante resaltar que existe la posibilidad de incorporar múltiples complementos (*plugins*) para cada uno de los servidores con el fin de lograr distintos comportamientos.

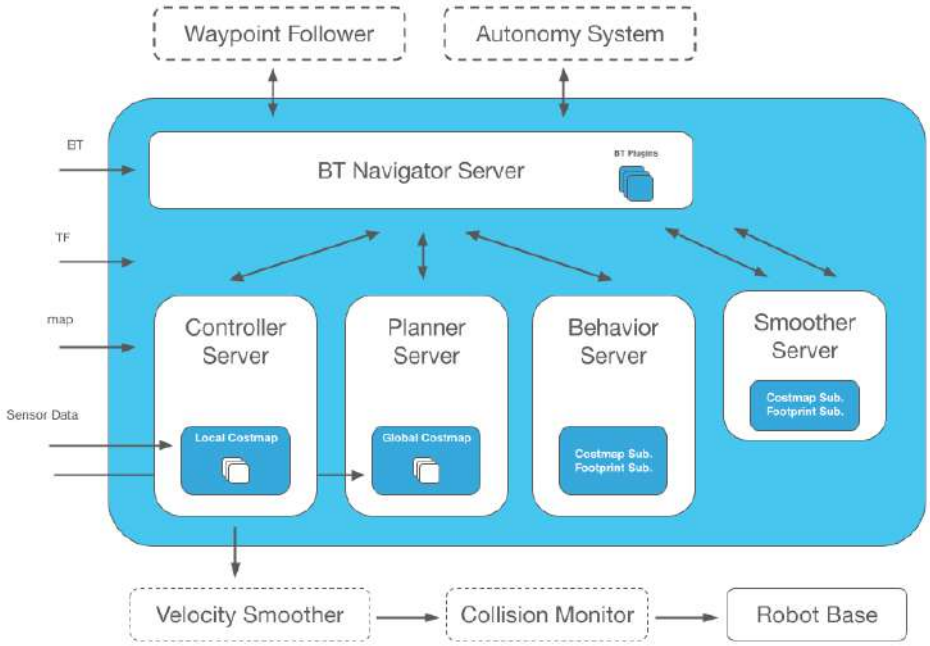


Figura 5.18: Arquitectura del paquete Nav2. Extraído de [66].

Como se puede ver en la Figura 5.18, uno de los componentes principales de Nav2 son los denominados árboles de comportamiento, también conocidos como *Behavior Trees* (BT). Dichos árboles constituyen estructuras jerárquicas y modulares para la ejecución de tareas específicas. Ofrecen una vía para la implementación de aplicaciones compuestas por múltiples etapas o estados. En el contexto de Nav2, los BT se emplean para la configuración de diversos comportamientos de navegación, coordinando las acciones de diferentes servidores.

Además de los BT, Nav2 se compone de distintos servidores para lograr los

objetivos, como se ve en la Figura 5.18 se cuenta con: *controller*, *planner*, *behavior* y *smoother*. Estos servidores contienen además *plugins* asociados a diversos algoritmos destinados a la ejecución de múltiples tareas.

En el núcleo de la navegación se encuentran el *planner* y el *controller*, como ya se mencionó en la Sección 5.1.3. El primero se encarga de determinar la ruta a seguir o plan global, mientras que el segundo se encarga de comandar el robot, siguiendo dicha ruta. Por otro lado, el *smoother* se emplea con el propósito de mejorar la trayectoria calculada por el *planner*. Los servidores de *behavior* contienen diferentes mecánicas a utilizar en distintos escenarios problemáticos para el robot, buscando recuperar su funcionamiento habitual, por lo general se los conoce como servicios de recuperación o *recovery server*.

Estos módulos independientes se comunican con el BT mediante una interfaz de ROS2, ya sea en forma de un servidor de acciones o de servicios. Esto facilita una interacción fluida entre los componentes de navegación y el BT. En la Figura 5.19 se presenta el BT utilizado en la solución implementada. Allí se ilustra cómo se relacionan los distintos servidores y controladores utilizando el *Behavior Tree*.

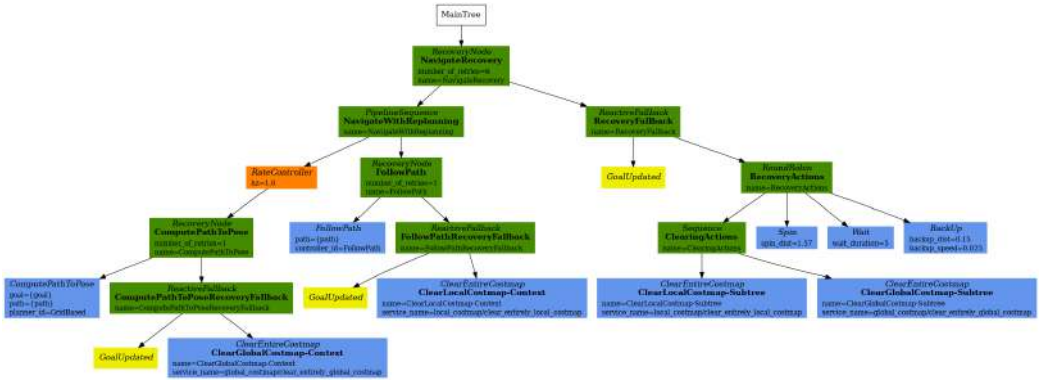


Figura 5.19: BT usado en la solución. Incluye nodos de navegación, planificación y recuperación.

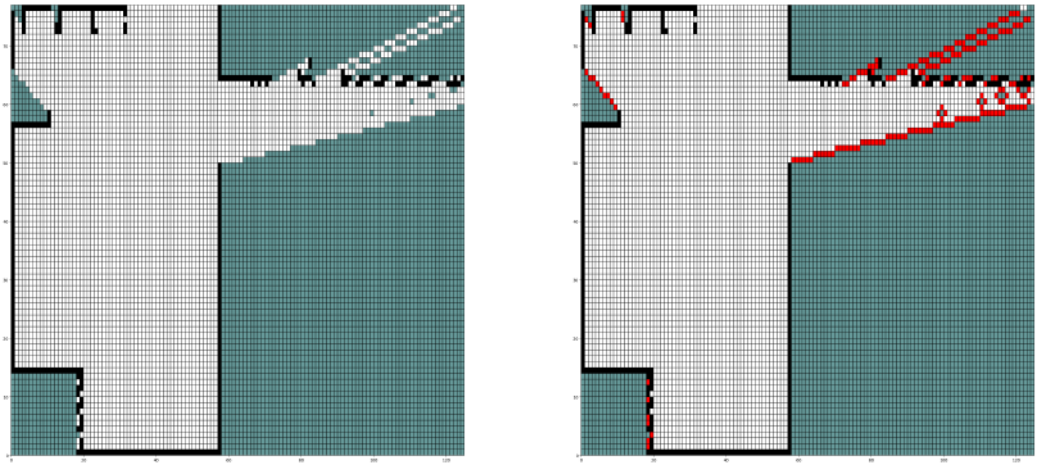
5.2.5. Task Allocation

En esta sección se detalla la implementación de los métodos que se desarrollaron con el fin de que la flota de robots sea una flota autónoma. Con autónoma se refiere a que los robots sean capaces de determinar sus propios objetivos para mapear y que se comuniquen entre ellos con el fin de realizar el mapeo de manera colaborativa, sin intervención humana ni de un planificador de misión externo.

En la Sección 5.2.5.1 se define lo que es una frontera, se describe cómo se pueden determinar y también agrupar. Por otro lado, en la Sección 5.2.5.2 se describe cómo se realiza la planificación y determinación de objetivos de manera colaborativa entre los distintos robots teniendo como entrada las fronteras.

5.2.5.1. Obtención y Agrupación de fronteras

Para la exploración autónoma se decide usar el enfoque de exploración basado en fronteras, siguiendo el trabajo propuesto por Yamauchi [67] [68], el cual



(a) Grilla de Ocupación (b) Fronteras

Figura 5.20: Representación de las etapas del cálculo de fronteras.

presenta que para ganar la mayor cantidad de información nueva del mundo, es necesario moverse a las fronteras. Estas se definen como los puntos limítrofes entre las regiones conocidas y las desconocidas.

La representación usada para el mapa es una grilla de ocupación. Básicamente, se trata de una estructura en la que se divide el espacio en pequeñas celdas o casillas, y a cada celda se le asigna una probabilidad que indica la ocupación o desocupación de esa área del mapa. Las celdas pueden tomar los siguientes valores: -1 para celdas desconocidas, 0 para celdas conocidas que se encuentran vacías y 1 para las celdas conocidas ocupadas. En la Figura 5.20a se puede observar una grilla de ocupación, en donde en gris se tienen las celdas desconocidas, en negro las ocupadas y en blanco las libres.

Se crea el nodo *frontiers_server*, el cual se encarga de calcular y agrupar las fronteras. El nodo contiene un servidor de servicios, que recibe un *request* vacío y retorna un *response* con las fronteras. A su vez, tiene un suscriptor del tópico */map*, el cual se encarga de que el nodo utilice siempre el último mapa generado.

Para el cálculo de las fronteras, se recorre la grilla de ocupación. Por cada celda libre, se chequea si alguna de las celdas vecinas es desconocida. En caso positivo, se marca como frontera. En la Figura 5.20b se puede observar en rojo las fronteras calculadas para la grilla previamente mencionada.

Con el fin de disminuir la probabilidad de que dos o más robots vayan hacia fronteras cercanas, se busca agrupar las fronteras. Para ello, se evalúan tres métodos.

Primero se utiliza el algoritmo *K-Means* [69] de la biblioteca Scikit-Learn [70]. Dado que el algoritmo toma como entrada la cantidad de *clusters*, se inicializa dicho valor en 1 y se itera incrementándolo hasta que, para cada centroide obtenido, la distancia a las fronteras perteneciente al *cluster* sea menor que el rango del LiDAR. En la Figura 5.21 se puede apreciar en azul los centroides obtenidos. Sin embargo, se descarta este algoritmo por dos razones. Primero se observa que los centroides

pueden no ser fronteras (peor aún, pueden quedar en celdas inaccesibles). A su vez, se encontró que al aumentar la cantidad de fronteras, el algoritmo no escala [71].

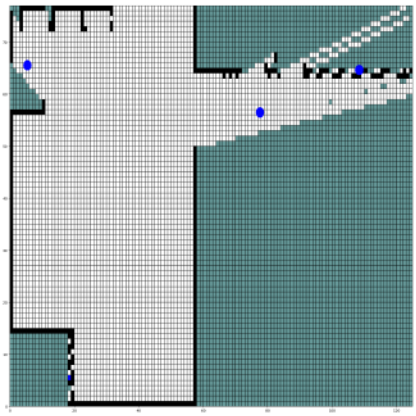


Figura 5.21: Centroides obtenidos usando K-Means

Luego, se evalúa utilizar el algoritmo *Afiinity Propagation* [72], también de la biblioteca Scikit-Learn. Una vez obtenidos los centroides de dicho algoritmo, se realiza un filtro, descartando aquellos con menos de 8 fronteras. En la Figura 5.22 se pueden observar en azul los centroides resultantes tras la agrupación y el filtrado. Una de las principales ventajas de este algoritmo es que todos los centroides son fronteras. Sin embargo, el algoritmo retorna centroides muy próximos, lo que puede resultar en una asignación de tareas ineficiente (es decir, dos robots yendo a posiciones muy cercanas).

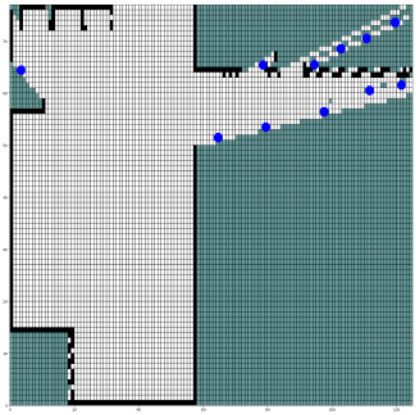
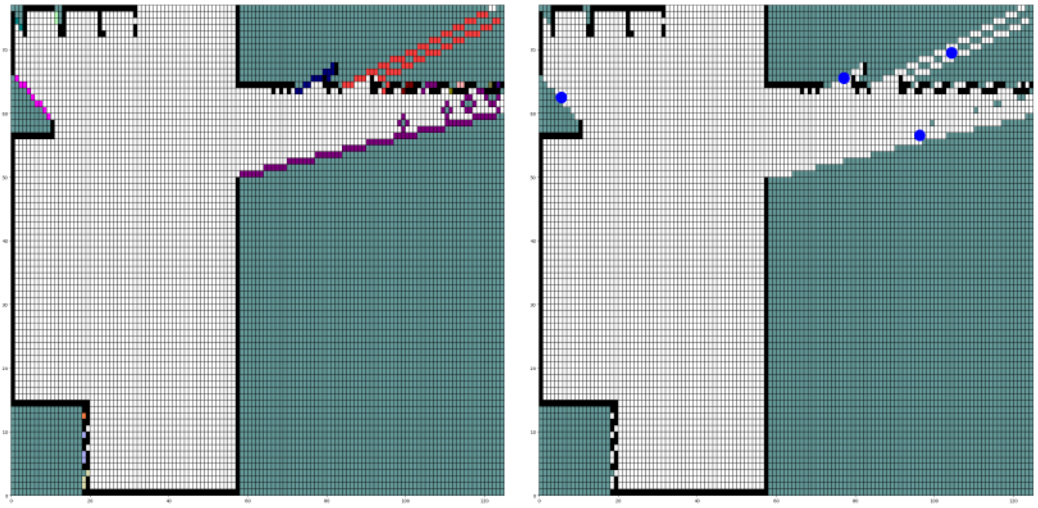


Figura 5.22: Centroides obtenidos filtrando los resultados de Affinity Propagation

Por último, se decide implementar un algoritmo propio, en el que se agrupan las fronteras contiguas en grupos. Se puede encontrar una explicación más detallada del algoritmo de agrupación de fronteras en el Anexo C. Luego se filtran aquellos grupos con menos de 5 fronteras y se subdividen los que tengan más de

100 fronteras. Finalmente, se calcula el centroide de cada grupo. En la Figura 5.23a se pueden apreciar los distintos grupos y en la Figura 5.23b los centroides de cada grupo.



(a) Grupos (b) Centroides

Figura 5.23: Representación del algoritmo de agrupación de fronteras.

5.2.5.2. Método de subasta y asignación de tareas

En la Sección 5.2.5.1 se presenta el concepto de puntos fronteras, los cuales son utilizados como objetivos al momento de realizar la exploración autónoma. Por lo tanto, uno de los grandes componentes de la asignación de tareas radica en cómo distribuir las fronteras objetivo entre la flota de robots.

Para realizar dicha distribución, se opta por implementar un método de subastas, en la que cada robot participante (denominado *bidder*) realiza una oferta por cada una de las fronteras, basándose en la distancia euclídea a la que se encuentra de esta. Luego, quien realiza la subasta (denominado *broker*), le asigna a cada robot una frontera objetivo en función de las ofertas recibidas.

A su vez, dado que uno de los objetivos del proyecto es que la exploración sea descentralizada, se tiene que el encargado de realizar la subasta (denominado *broker*) no es fijo. El primer robot disponible es quien se encargará de recibir las ofertas (incluyendo las suyas) y realizar la asignación de tareas.

Como se puede observar en la Figura 5.24, se implementa una máquina de estados que determina las tareas que debe realizar cada robot. El estado inicial es **IDLE**. Este es el único estado que requiere *input* humano, ya que su única función es esperar por la señal de inicio (mensaje al tópico */start*). De esta forma se puede corroborar la correcta inicialización de cada robot antes de comenzar a explorar, especialmente útil al momento de desarrollar y *debuggear*.

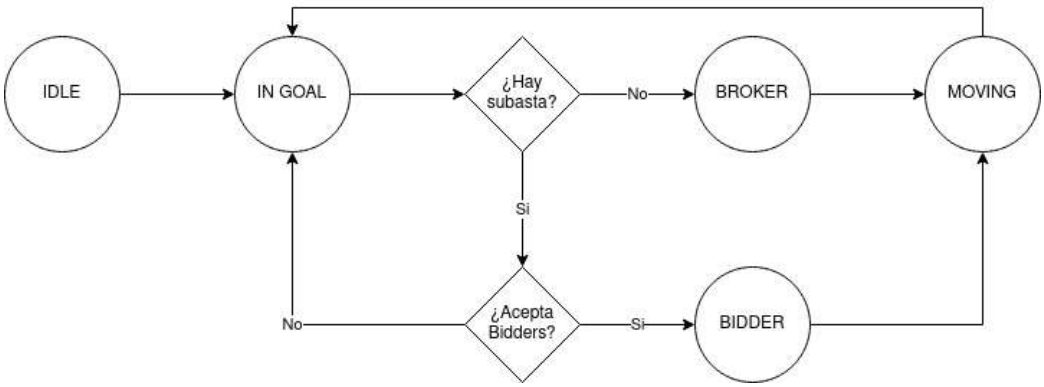


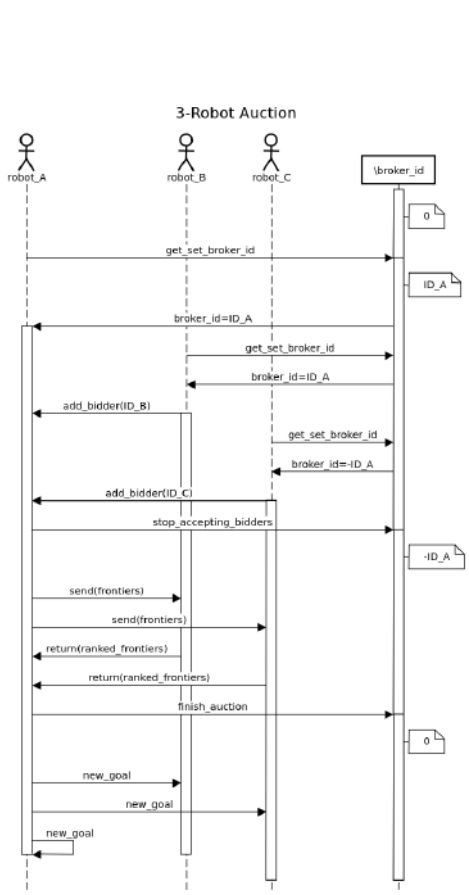
Figura 5.24: Máquina de estados de cada robot.

Una vez recibida la señal de inicio, se pasa al estado **IN GOAL**. Dicho estado representa que el robot alcanzó el último objetivo recibido y está disponible para realizar una nueva exploración. En este estado es donde se determina el rol del robot en la próxima subasta. Existen tres posibles escenarios: no hay una subasta activa, hay una subasta activa que acepta *bidders* y por último está el escenario en que existe una subasta activa pero ya no acepta *bidders*. En el primer caso, al no haber una subasta activa, el robot asume el rol de *broker* para comenzar una nueva subasta y pasa al estado con el mismo nombre. Análogamente, para el segundo caso, el robot pasa al estado *bidder*, comunicándole previamente al *broker* su participación en la subasta. Por último, en el tercer caso, el robot realiza una espera activa hasta la finalización de la subasta, en cuyo momento se repite el proceso de selección de *broker*.

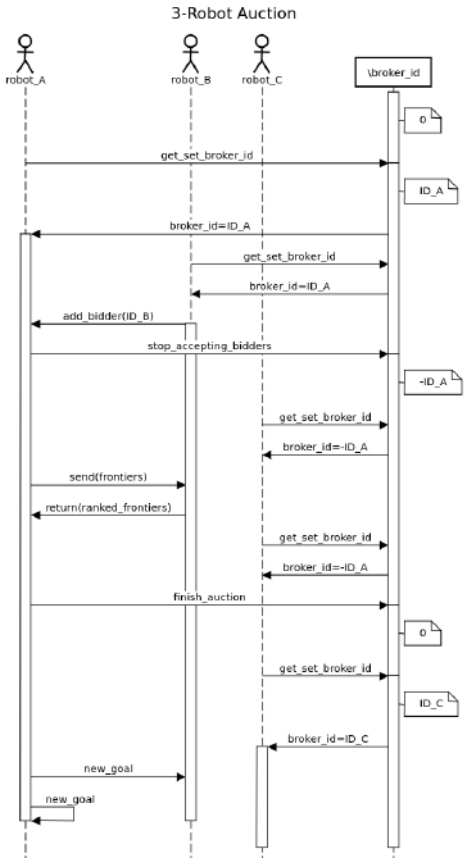
En la Figura 5.25 se pueden ver dos posibles escenarios de este procedimiento para una flota de 3 robots. En el escenario de la izquierda, los tres robots participan de la misma subasta, mientras que en el de la derecha el robotC intenta unirse a la subasta activa de forma tardía no pudiendo participar en esta. Como el *broker* (robotA) ya no acepta más bidders, el robotC debe esperar que este finalice la subasta para poder continuar.

Cabe destacar que el tópico */broker_id* es global, suscrito y publicado por todos los robots. Dicho tópico se encarga de comunicar el identificador del *broker* responsable de la subasta activa. Si se aceptan nuevos *bidders*, el identificador es positivo. En caso contrario, el identificador es negativo. Por último, si no hay una subasta activa se comunica el valor 0.

Como se mencionó previamente, el robot que se encuentre en estado ***Broker*** es el encargado de llevar a cabo la subasta. Primero, espera cierto tiempo a que otros robots se sumen a la subasta. Entre mayor sea el tiempo de espera, mayor es la probabilidad de que la cantidad de robots participantes de cada subasta aumente y por lo tanto mejor sea cada asignación de fronteras. Por otro lado, puede ocasionar esperas innecesarias. En la Sección 7.1.4 se comparan los resultados obtenidos al variar dicho valor. Pasado dicho tiempo, dejan de aceptar adiciones, lo cual se comunica cambiando el valor del tópico */broker_id* como se explicó previamente. Seguido de eso, realiza otra espera, con el fin de evitar condiciones de carrera



(a) A es *broker*, B y C son *bidders*.



(b) A es *broker*, B es *bidder* y C llega tarde a la subasta.

Figura 5.25: Subasta con 3 robots.

entre un robot uniéndose a la subasta y el robot *broker* finalizando el período de aceptación. Finalizado dicho periodo, realiza el cálculo de fronteras, explicado en la sección 5.2.5.1, las cuales envía a todos los participantes de la subasta. Luego, recibe los rankings de las fronteras de cada participante y da por finalizada la subasta, enviando un 0 al tópico. Por último, asigna una frontera a cada robot, basándose en los rankings recibidos y se la envía como próximo objetivo a explorar.

Para el caso del estado ***Bidder***, la primera tarea es comunicarle al *broker* su participación en la subasta. Para eso, envía su identificador al *broker* a través de un servicio de este. Luego, queda a la espera de recibir las fronteras objetivos, las cuales ordena según distancia euclídea y envía de regreso al *broker*. Finalmente, espera hasta recibir la frontera asignada hacia la cual debe navegar en el siguiente estado.

Finalizada la subasta, tanto el *broker* cómo el *bidder* pasan al estado ***Moving***, en el cual se toma la nueva frontera asignada como objetivo de exploración y se navega hacia ella. Para realizar dicha tarea se utiliza un *action server* de Nav2 denominado *nav_to_pose*. Este recibe como entrada la posición a la que se quiere

navegar y retorna el resultado de la navegación (exitosa o fallida). En ambos casos, se pasa al siguiente estado (*InGoal*, ya que en dicho estado se organiza una nueva subasta.

Para implementar el funcionamiento descrito, se crean tres nodos en ROS2. Uno encargado de manejar el estado actual y los cambios de estados del robot, denominado *state_manager_node*. Otro nodo, encargado de realizar las tareas correspondientes de cada estado, denominado *execution_node*. Por último, se tiene el *auction_node* que se encarga de todos los mecanismos relacionados a las subastas, desde la comunicación al tópico */broker_id*, hasta el envío (en caso de *bidder*) o control (en caso de *broker*) de los avisos de participación de una subasta. En la Figura 5.26 se muestra un diagrama simplificado de la interacción de dichos nodos para un robot que organiza una subasta.

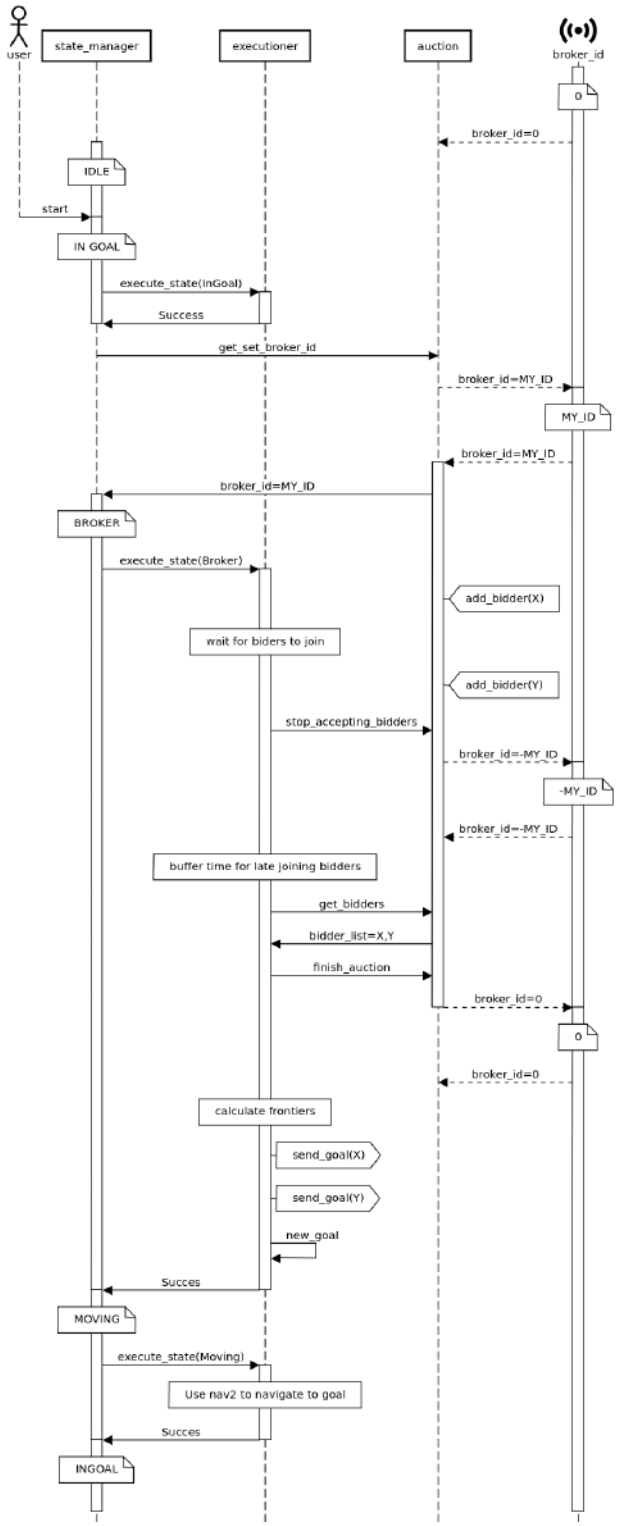


Figura 5.26: Diagrama de alto nivel del funcionamiento de un robot siguiendo la máquina de estados.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 6

Simulación

En esta sección se introduce la herramienta utilizada para simular el comportamiento del robot. Además, se mencionan características necesarias para configurar las simulaciones y se introduce el modelo del robot utilizado.

La simulación proporciona un entorno controlado y repetible para probar algoritmos, validar diseños y analizar el comportamiento de distintas soluciones. En este proyecto, debido a que se trabajó en forma paralela en la implementación de la plataforma física y el desarrollo de la solución de software, todos los algoritmos fueron evaluados primero en el entorno simulado y luego trasladados a los robots físicos.

En la Sección 6.1 se introduce el simulador utilizado y se mencionan características propias de él. En la Sección 6.2 se introduce el modelo del robot utilizado en las simulaciones en el proyecto.

6.1. Gazebo

El simulador utilizado por defecto en ROS2 se llama Gazebo [9], es un simulador físico en 3D de código abierto utilizado en la investigación, desarrollo y pruebas de sistemas robóticos. Proporciona un entorno virtual realista en el que los desarrolladores pueden modelar, simular y experimentar con diferentes robots y escenarios. Cuenta con varios motores de física utilizados para simular la interacción entre objetos en el mundo virtual. Esto incluye la simulación de la dinámica de los robots, la interacción con objetos, la detección de colisiones y la respuesta a fuerzas y torques. Permite crear modelos detallados de robots y entornos utilizando el formato SDF (*Simulation Description Format*) o URDF (*Unified Robot Description Format*), logrando representar la geometría, cinemática y dinámica de sistemas robóticos. Admite agregar sensores y actuadores a los modelos de los robots como cámaras, LiDARs, motores, entre otros [73]. Gazebo cuenta con *plugins*, los cuales pueden ser creados para controlar el comportamiento de los modelos y la interacción con el entorno. Cuenta con una interfaz gráfica (GUI, por sus siglas en inglés) que permite a los usuarios interactuar con el simulador de manera visual, Figura 6.1.

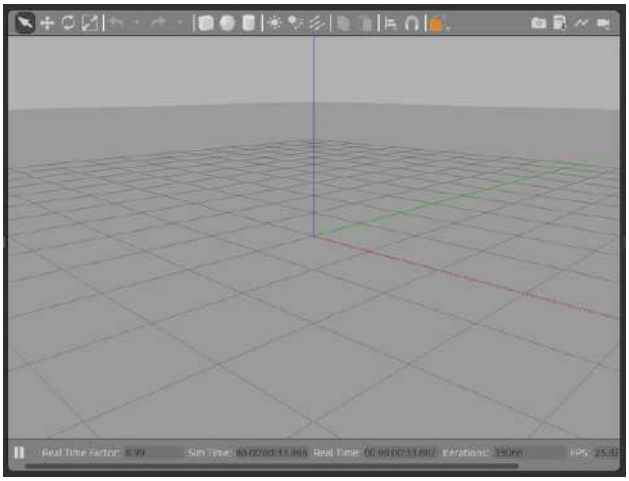


Figura 6.1: Interfaz gráfica de Gazebo al simular un mundo vacío.

6.1.1. Mundo

Para el desarrollo y pruebas simuladas se utilizaron principalmente dos mundos en Gazebo. El primero es un mundo incluido por defecto en la instalación de gazebo, denominado *Playground*. Como se muestra en la Figura 6.2, cuenta con los siguientes elementos: cuatro árboles, dos paredes de ladrillo, tres barreras de cemento, dos barreras de tránsito, dos postes de luz, un cono y un hidrante.

Figura 6.2: Mundo *Playground* visualizado en Gazebo.

Por otro lado, se utilizó un mapa del piso inferior InCo. Si bien el mapa no respeta completamente la escala, la distribución ni el mobiliario real, es una buena aproximación del plano real del InCo. En la Figura 6.3 se presenta dicho mundo

en gazebo.

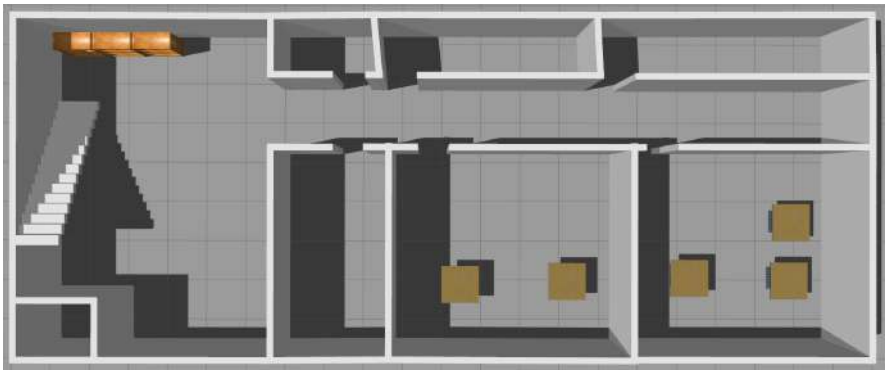


Figura 6.3: Mundo InCo visualizado en Gazebo.

6.2. Modelado del robot

En esta sección se documenta cuales fueron los pasos seguidos para llegar al modelo de robot utilizado en las simulaciones.

El equipo se propuso realizar un modelo propio de *Rogel* con las mismas características físicas que se presentan en la Sección 3.3.1. A pesar de investigar sobre el modelado usando archivos URDF y realizar una prueba de concepto del robot, no fue posible obtener un resultado que se pueda utilizar a la hora de simular. Aspectos físicos del modelo como la matriz de inercia y coeficientes de rozamiento no eran los adecuados resultando en movimientos no deseados por parte del robot. Para no comprometer el tiempo dedicado y lograr avanzar en las simulaciones se descartó el desarrollo del modelo y se procedió a buscar un modelo ya existente que contemple las características deseadas.

Se investigaron varios proyectos, los cuales fueron descartados tanto por problemas de compatibilidad como de funcionamiento. Un ejemplo es el proyecto OpenBase [74] que cuenta con un modelo de un robot omnidireccional de 3 ruedas similar a *Rogel* como se ve en en la Figura 6.4. No obstante, el modelo de dicho proyecto fue desarrollado para ROS1 en lugar de ROS2 y por lo tanto utiliza un *plugin* para su movimiento no compatible con la versión de ROS2, ni con la versión de Gazebo.

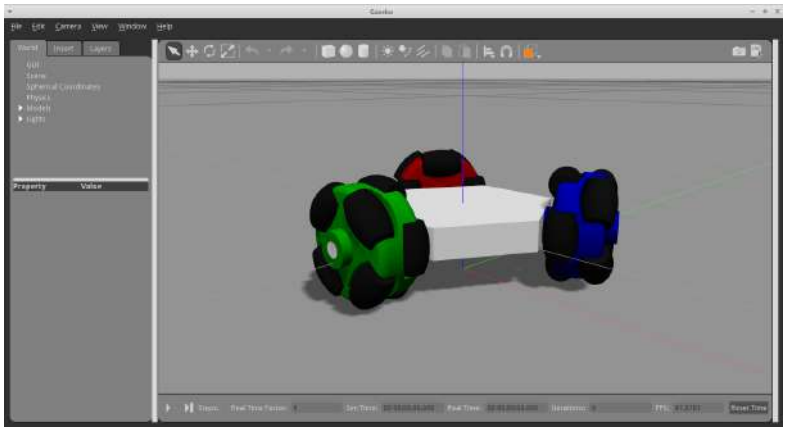


Figura 6.4: Modelo del proyecto OpenBase visualizado en Gazebo.

Finalmente, se decidió por utilizar el modelo del proyecto Linorobot2 [75] que se muestra en la Figura 6.5. Este robot, es un robot holonómico que cuenta con un LiDAR, por lo que cumple con las características principales de Rogel. Sin embargo, el modelo de Linorobot es el de un robot con 4 ruedas mecanum que dan la característica de holonomía, en lugar de un robot de tres ruedas omnidireccionales. Si bien esto no es lo ideal, debido a que no se encontró otro modelo alternativo que represente mejor a Rogel, se asumió esta diferencia y se continuó con este modelo.

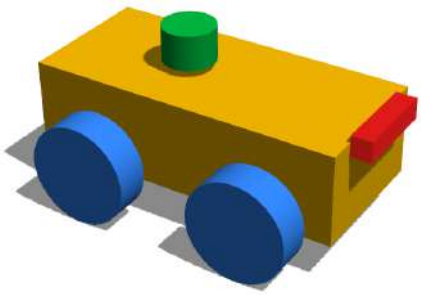


Figura 6.5: Modelo del proyecto Linorobot2 visualizado en Gazebo.

Además de las características ya mencionadas, el modelo de Linorobot cuenta con una cámara frontal que no se utilizó para las simulaciones puesto que Rogel no cuenta con este tipo de sensores.

6.3. Solución Implementada

Se crea un *launchfile* que se encarga de ejecutar todos los componentes necesarios para la simulación.

Se ejecutan para cada robot los mismos nodos utilizados para la solución física explicada en la Sección 5.2 encargados de SLAM, navegación y asignación de tarea.

Luego, en lugar del *launchfile icre_robot*, se utiliza el nodo *spawn_entity* del paquete *gazebo_ros*. Dicho nodo toma de entrada el archivo en formato URDF del modelo del robot descrito anteriormente y lo integra a Gazebo. Éste se encarga de simular el LiDAR, de mover el robot en función de las velocidades publicadas en el tópico *cmd_vel* y de publicar la odometría.

De esta forma, se logra ejecutar la simulación, compartiendo la lógica de la solución con la contraparte física.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 7

Pruebas y Resultados

Este capítulo tiene como objetivo caracterizar el funcionamiento del robot. En particular se presentan diferentes experimentos, su implementación y también los resultados de las pruebas. En este capítulo se incluyen solamente pruebas sobre los algoritmos y soluciones expuestas en el Capítulo 5.

La presentación de las pruebas y resultados se divide en dos bloques temáticos. En la Sección 7.1 se presentan los resultados obtenidos realizando las pruebas en el simulador. Mientras que en la Sección 7.2 se exponen todos los resultados registrados en Rogel. Allí se incluyen particularidades sobre la implementación en un robot real.

Es importante mencionar que de no aclarar en contrario, las pruebas que se exponen en este capítulo se basan en el subsuelo del InCo como entorno de referencia. En las pruebas simuladas se utiliza en general el mundo del InCo introducido en la Sección 6, mientras que las pruebas reales se realizaron en el subsuelo del edificio del InCo. Esto se realiza con el objetivo de minimizar las diferencias que pueden existir entre el entorno simulado y las pruebas físicas.

7.1. Simulado

Esta sección describe las pruebas más relevantes realizadas en el simulador. A lo largo de esta sección se presentarán diferentes experimentos donde se evalúan diferentes configuraciones de parámetros de los algoritmos presentados en la Sección 5. En estos experimentos se utilizó como métrica principal el tiempo total de exploración.

Las pruebas que se presentan para el entorno simulado son:

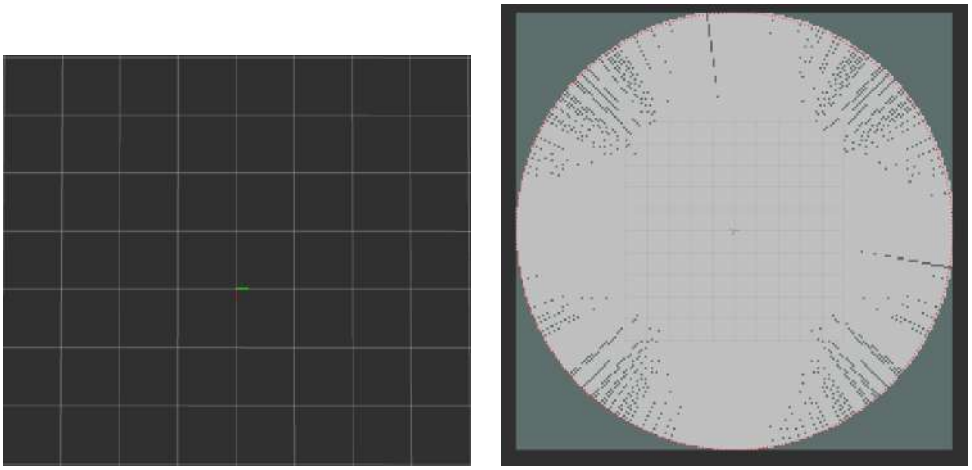
- En la Sección 7.1.1 se evalúa el uso del nodo scan_filter introducido en la Sección 5.2.2.1.2 teniendo en cuenta las consecuencias que tiene en los mapas.
- En la Sección 7.1.2 se evalúan diferentes algoritmos considerados para agrupar las fronteras y determinar los objetivos de la navegación.
- En la Sección 7.1.3 se presentan los resultados que se obtienen al variar la cantidad de robots que componen la flota.

- En la Sección 7.1.4 se expone la influencia de variar el tiempo que el broker de una subasta espera por potenciales bidders antes de comenzar una subasta.
- Finalmente en la Sección 7.1.5 se analiza la solución final para el entorno simulado y se caracteriza utilizando distintas métricas.

7.1.1. Manejo del LiDAR

Esta sección introduce las pruebas relacionadas al uso o no del nodo scan_filter. Como se menciona en la Sección 5.2.2.1.2, este nodo a cada uno de los valores devueltos por el LiDAR como *'inf'* los cambia por el rango máximo del LiDAR.

Lo primero que se puede analizar es el motivo de utilizar este filtro, para ello se presentan las Figuras 7.1a y 7.1b. En estas dos figuras se presentan los mapas generados por el robot en el mundo Open World de gazebo, el presentado en la Figura 6.1.



(a) Sin scan_filter (b) Con scan_filter

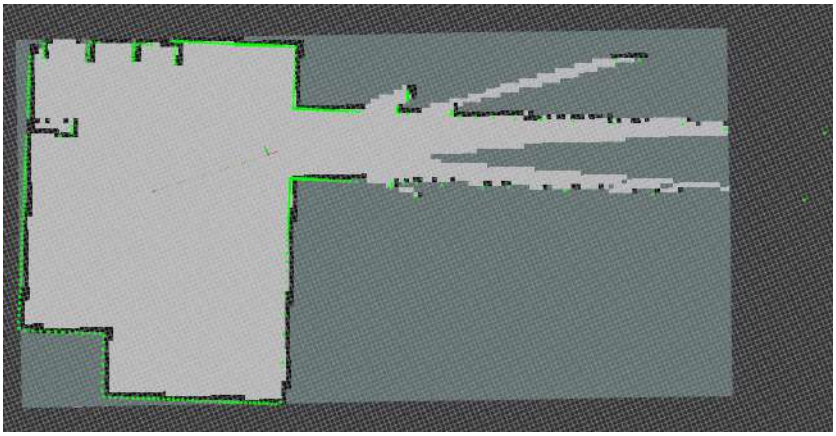
Figura 7.1: Mapas generados del mundo Open World.

Como se ve en la Figura 7.1a, el robot no es capaz de realizar ningún mapa debido a que todas las muestras del LiDAR son *'inf'*. Un comportamiento análogo se obtendría en un entorno donde el robot no tuviera ningún objeto a distancia menor que el rango máximo del LiDAR. Este comportamiento se debe a que el algoritmo de SLAM descarta todo valor no numérico y por ende no posee datos de entrada para generar un mapa. La conclusión que se deriva de este experimento es que ante un escenario análogo al del Open World, la solución que se implementa en este proyecto no sería capaz de determinar una celda frontera, por lo que no se podría enviar un objetivo y por ende el robot no se movería. Por lo tanto, no se cumpliría con el objetivo de generar un mapa de manera autónoma.

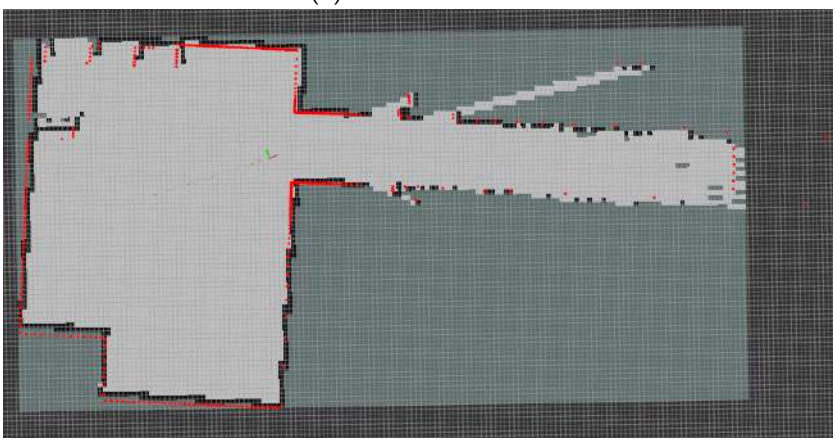
Para evitar este problema se consideró el uso del nodo scan_filter. Como se puede ver en la Figura 7.1b, al utilizar este nodo se genera un mapa alrededor del robot con celdas libres (sin obstáculos) y por ende se logra una representación

más representativa del entorno. Es importante aclarar que el valor necesario para el filtro es exactamente el rango máximo del láser. Esto se debe a cómo interpreta el algoritmo de SLAM las medidas. Si el valor fuese menor al rango máximo, el algoritmo de SLAM interpretaría esa medida como un obstáculo, mientras que si fuese mayor se descartaría la medidas de la misma forma que los *'inf'*.

Se realizó el mismo experimento pero ahora para el mundo del InCo. En la Figura 7.2a se observa una captura sin utilizar el scan_filter, mientras que en la Figura 7.2b utilizando el scan_filter. En estos casos la diferencia se hace visible al final del pasillo.



(a) Sin scan_filter



(b) Con scan_filter

Figura 7.2: Mapas generados del InCo

Para el caso donde no se utiliza el scan_filter, Figura 7.2a, se puede ver que hacia el final del pasillo el mapa no se completa debido a que el final del pasillo excede el rango máximo. Por otro lado, cuando se utiliza el scan_filter, se obtiene el mapa de la Figura 7.2b donde se ve una mayor cantidad de mapa explorado.

Si bien parece intuitivo con los argumentos anteriores que se deba utilizar el scan_filter en la solución final, el equipo igualmente se planteó evaluar las diferencias que se observa a la hora de mapear el InCo utilizando este filtro o no. Para

ello se propuso realizar 10 simulaciones con 3 robots cada una en las que se registre el tiempo de mapeo. Los resultados que se obtuvieron de estas simulaciones se presentan en la Tabla 7.1

1

	Sin filtrado	Con filtrado
Tiempo mínimo (s)	190	152
Tiempo máximo (s)	270	231
Tiempo promedio (s)	226	196

Tabla 7.1: Comparación de tiempos al mapear el InCo con y sin el nodo scan_filter en uso.

Si se observan los resultados que se ven en la Tabla 7.1 se puede apreciar cómo el uso del filtro reduce el tiempo de trabajo promedio en un 7 %. Además, se puede observar cono tanto el tiempo máximo como mínimo también es menor cuando se utiliza el filtrado de las medidas del LiDAR. Es por ello que todos los experimentos que se exponen para la simulación de ahora en más consideran el uso del nodo scan_filter.

7.1.2. Tipos de agrupación

En esta sección se analizan los resultados al variar el método utilizado para agrupar fronteras. Basados en [8] se consideró analizar el uso de *k-means* y de *Affinity Propagation*. Además de estos dos, se consideró también el algoritmo propuesto en la Sección 5.2.5.1 implementado para este proyecto que de ahora en más se referirá a él como *custom*.

Los resultados que se exponen en esta sección, a diferencia del trabajo realizado en [8], no tienen como objetivo comparar en detalle los diferentes algoritmos de agrupación, ni generalizar los comportamientos para cualquier tipo de entorno. Sino que, se pretende plantear diferencias entre los algoritmos y observar los resultados en el tiempo total de exploración en el entorno de trabajo de este proyecto, sin estudiar las justificaciones de fondo.

Previo a proceder con el experimento, es importante detallar cómo se utilizaron cada uno de los algoritmos. Para ello, se expone una breve explicación del procedimiento con el cual se agrupan las fronteras en cada caso. Esta explicación se acompaña a su vez con una representación visual. Vale la pena mencionar que todos los algoritmos utilizaron como entrada la misma grilla de ocupación en estas representaciones.

En la Figura 7.3 se presenta de forma visual los pasos que se utilizan para agrupar las fronteras con *k-means*. Con el fin de agrupar las fronteras utilizando *k-means* primero se deben computar las fronteras a partir de la grilla de ocupación de entrada. Luego, se aplica a esas celdas frontera el algoritmo de clusterización *k-means* de forma iterativa siguiendo el método del codo. Con esta clusterización, se obtiene la agrupación de fronteras buscada [76].

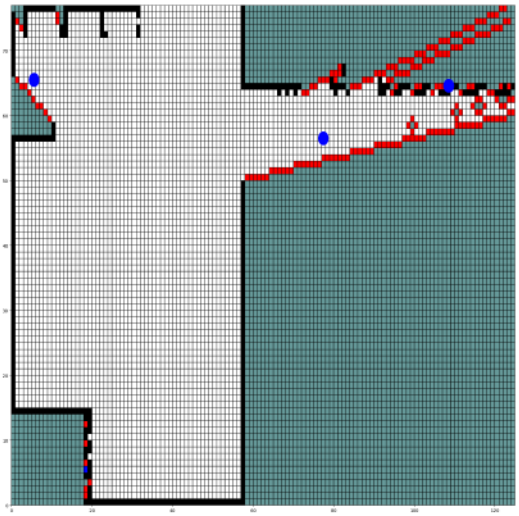


Figura 7.3: Gif de la agrupación de fronteras utilizando *k-means*.

En la Figura 7.3 primero se muestra la grilla de ocupación de entrada, en el segundo frame se observan las fronteras computadas y finalmente en el tercer frame se observan los 4 centroides obtenidos luego de aplicar el algoritmo de *k-means*.

De forma análoga, se introducen los pasos que se realizan para la agrupación cuando se utiliza *Affinity Propagation* en la Figura 7.4. Al igual que para el caso anterior, en el primer frame se muestra la grilla de ocupación de entrada y en el segundo se computan las fronteras. En el tercer frame, dadas las fronteras, se agrupan utilizando *Affinity Propagation* y en el cuarto frame se descartan aquellos grupos que cuenten con menos de diez celdas fronteras. En el ejemplo se descarta el grupo de abajo a la izquierda ya que cuenta con 7 celdas frontera.

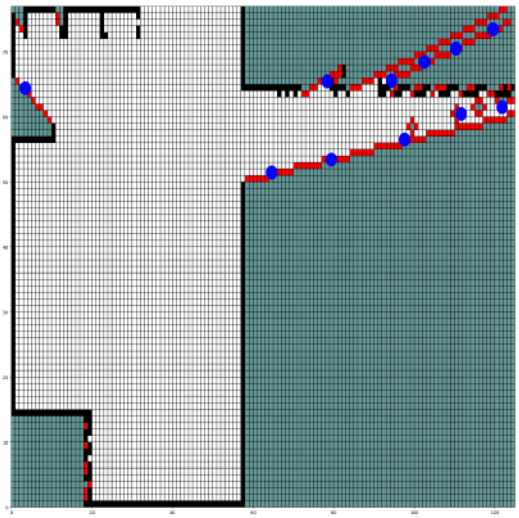


Figura 7.4: Gif de la agrupación de fronteras utilizando *Affinity Propagation*.

Finalmente se presenta el algoritmo *custom* en la Figura 7.5. Nuevamente, los

primeros dos frames se repiten. En el tercer frame se agrupan las fronteras siguiendo el algoritmo detallado en la Sección 5.2.5.1, en el cuarto se filtran aquellos grupos con menos de 5 fronteras y finalmente en quinto frame se calculan los centroides aplicando el promedio.

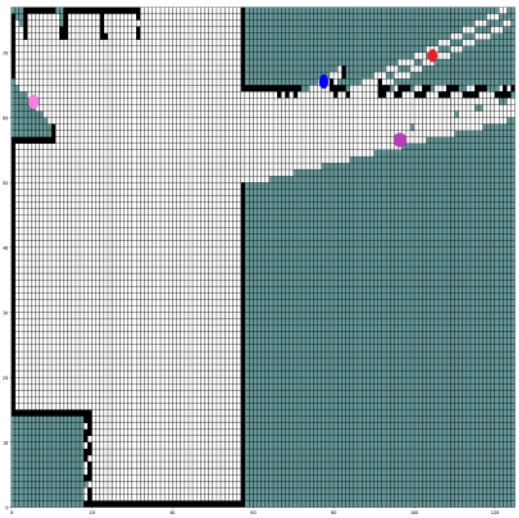
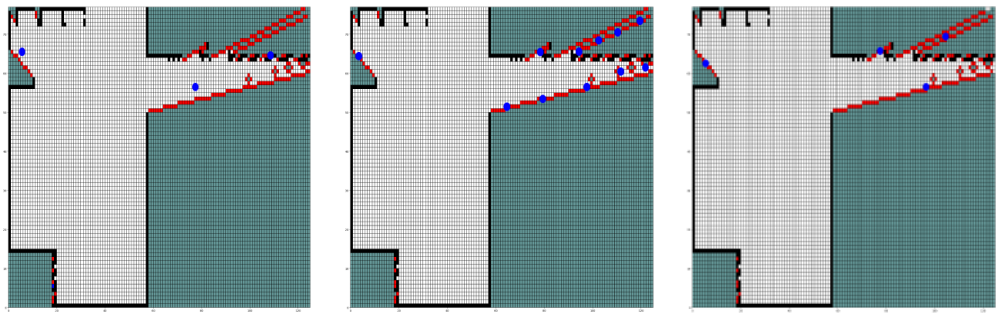


Figura 7.5: Gif de la agrupación de fronteras utilizando el algoritmo *custom*.

En la Figura 7.6 se exponen los grupos que se obtienen utilizando los diferentes métodos. En particular, en la Figura 7.6a se exponen los resultados de *k-means*, en la Figura 7.6b los de *Affinity Propagation* y en la Figura 7.6c los del algoritmo *custom*.



(a) *k-means* (b) *Affinity Propagation* (c) *Custom*

Figura 7.6: Grupos de fronteras para los diferentes métodos de agrupación.

Como se ve en las figuras anteriores, los algoritmos de agrupación de fronteras devuelven resultados evidentemente distintos. Es por ello que se resolvió analizar con cuál de los tres algoritmos se lograba mapear el InCo en menor tiempo. Para ello se realizó el experimento de mapear el InCo con 3 robots 10 veces utilizando cada uno de los algoritmos. Como conclusión de los experimentos se obtiene la Tabla 7.2, donde se exponen los tiempos mínimos, máximos y promedios al mapear con los diferentes algoritmos de agrupación.

	K-Means	Affinity-Propagation	Custom
Tiempo mínimo (s)	185	178	152
Tiempo máximo (s)	315	207	231
Tiempo promedio (s)	223	194	196

Tabla 7.2: Tiempo al mapear el InCo con los diferentes algoritmos de agrupación de fronteras.

Como se puede observar en la tabla comparativa, cuando se utiliza *K-Means* se obtienen los peores resultados e incluso se descartaron algunas pruebas ya que la elección de los centroides resultaba ser muy mala o incluso inalcanzable. Si se compara el uso de *Affinity-Propagation* contra el algoritmo *Custom* se puede ver como los resultados en promedio son similares aunque el algoritmo *Custom* posee más desviación entre el promedio y los extremos. Igualmente, se decidió seguir adelante utilizando el algoritmo *Custom* ya que se trata de un algoritmo desarrollado por el equipo y que, por ende, se tiene más control sobre el proceso mediante el cual se realiza la agrupación de fronteras. Por otro lado, no posee grandes diferencias en su rendimiento si se lo compara con los otros algoritmos probados. Finalmente, el equipo entiende que es posible realizar modificaciones con el fin de obtener mejores resultados teniendo en cuenta otras métricas para la determinación de los centroides por ejemplo.

7.1.3. Cantidad de robots

En esta sección se presentan los resultados que se obtienen cuando se varía el tamaño de la flota. El objetivo principal de esta sección es el de demostrar que la colaboración entre robots logra que se reduzcan los tiempos de generación de mapas. Si bien el número óptimo de robots depende fuertemente de la topología del entorno a explorar, se quiso al menos investigar qué valor corresponde para el mundo del InCo.

Para ello, se varió la cantidad de robots que componen la flota y se realizaron 10 mapas con cada una de estas flotas. La cantidad de robots de la flota se varió entre 1 y 4, no se pudo incrementar más la cantidad de robots en la simulación debido a que la computadora en la cual se corrían no poseía el poder de cómputo necesario para correr con tantos robots. Con estas experiencias se creó la gráfica de la Figura 7.7, donde se exponen los resultados promedios de tiempos al variar la cantidad de robots que componen la flota.

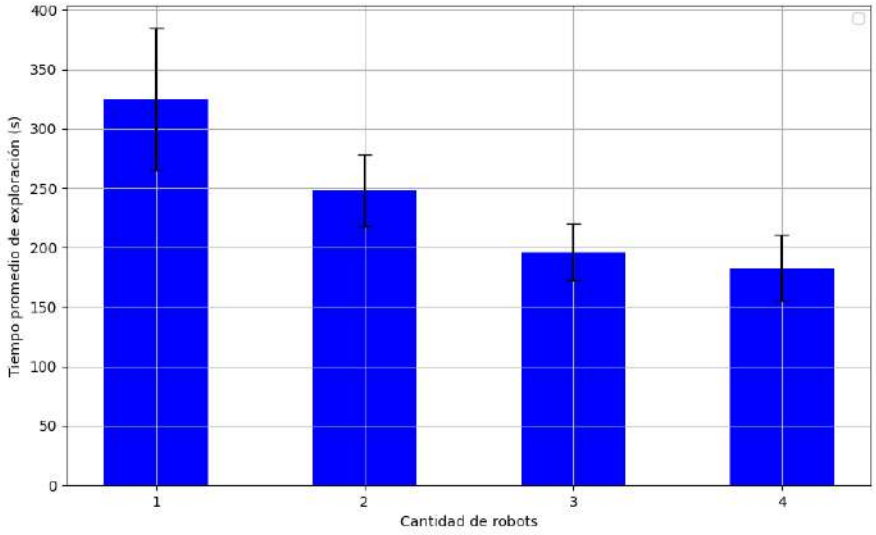


Figura 7.7: Gráfica de tiempo promedio de exploración en función de cantidad de robots de la flota.

Como se puede observar en la Figura 7.7 cuando se aumenta la cantidad de robots se reduce significativamente el tiempo para los primeros incrementos. Por ejemplo, la diferencia promedio de trabajar con uno y con dos robots es de aproximadamente un 30 % de tiempo, una diferencia considerable. Sin embargo, cuando la cantidad de robots excede los 3 robots el tiempo ya no mejora de manera significativa y puede considerarse prácticamente igual. En la Tabla 7.3 se incluyen resultados más en detalle sobre los datos para los experimentos variando la cantidad de robots.

Cantidad de robots	Tiempo mínimo (s)	Tiempo máximo (s)	Tiempo promedio (s)
1	220	416	325
2	216	310	248
3	152	231	196
4	147	213	183

Tabla 7.3: Tiempo al mapear el InCo con los diferentes algoritmos de agrupación de fronteras.

Como se puede observar, al aumentar la cantidad de robots el tiempo de exploración disminuye significativamente, registrando casi la mitad del tiempo cuando se realiza el experimento con cuatro robots en comparación a uno solo, una mejora del 43 %. Como se ve tanto en la Figura 7.7 como en la Tabla 7.3, la tasa de mejora es decreciente con la cantidad de robots. Este mismo fenómeno se documenta en [8], incluso allí se considera que es probable que para cierto tamaño de flota, el tiempo empeore. Esto se debe a que al aumentar la cantidad de robots se

pueden producir colisiones, atascamientos e incluso, los algoritmos de distribución de tareas deben computar mayor cantidad de información al mismo tiempo. Por otra parte, si se considera un robot en particular, que existan más robots en su entorno le restringe su espacio de configuración ya que se tienen mayor cantidad de obstáculos en el entorno.

Se plantea como hipótesis de los resultados de esta experiencia que cada entorno posee una cantidad óptima de robots para realizar tareas de mapeo. Es decir, que si se aumenta o se disminuye el tamaño de la flota, el tiempo de exploración aumenta.

Como en la práctica se cuentan con 3 robots, en la última prueba de la sección se trabajará con esta cantidad.

7.1.4. Tiempo de espera por *bidder*

En esta sección se trabaja sobre el último parámetro que se busca configurar en este proyecto, el tiempo de espera por *bidders*. Al igual que en los experimentos anteriores, se realizaron mapas variando este valor con el fin de encontrar el valor con el cual se reduzca el tiempo promedio de mapeo. En este caso el tiempo de espera por *bidders* se varió entre 4 segundos, 6 segundos y hasta que lleguen todos los robots a la subasta. Los resultados de estas simulaciones se encuentran disponibles en la Tabla 7.4.

Tiempo de espera	Tiempo mínimo (s)	Tiempo máximo (s)	Tiempo promedio (s)
4 s	172	265	202
6 s	152	231	196
Esperar por todos los *bidders*	167	248	201

Tabla 7.4: Tiempo al mapear el InCo para diferentes tiempos de espera de *bidders*.

Como se ve en la Tabla 7.4, el tiempo de espera óptimo determinado en el experimento es de 6 segundos. Igualmente, los resultados que se obtienen al considerar 4 segundos o esperar a que todos los robots se unan no devuelve resultados muy diferentes. Sin embargo, se atribuye que esta paridad en los resultados se debe a que la flota está compuesta simplemente por tres robots y por ende no se debe esperar un tiempo excesivo por algún robot. De todos modos, se cree que si se trabajara con una cantidad mayor de robots estas diferencias serían más notorias, ya que este experimento evalúa la influencia de la colaboración en el tiempo total de exploración.

Un tiempo de espera menor, genera que la coordinación se realice con menor cantidad de robots. Es decir, la probabilidad que dos robots tengan como objetivo un mismo punto o una misma región del mapa aumenta. En particular, si se considera el caso donde el tiempo de espera es 0 segundos, no habría coordinación y cada robot exploraría siendo agnóstico de las tareas que realiza el resto.

Por el contrario, si el tiempo es excesivamente grande se obtiene una solución donde en cada subasta se espera por el total de los robots. En este caso, se obtiene

una división de tareas con mayor coordinación pero donde la probabilidad de que un robot permanezca esperando crece. Además, le quita robustez a la solución debido a que si un robot pierde conexión o por algún motivo falla, toda la exploración se ve cancelada o queda en espera por ese robot.

Como conclusión de esta experiencia se puede decir que existe un óptimo para el tiempo de espera de los *bidders*. Este tiempo depende nuevamente del entorno y de los robots que se consideren. En el caso particular de este proyecto, considerar 6 segundos como tiempo máximo para la espera de *bidders* devuelve los mejores resultados, ya que posee el tiempo mínimo, máximo y promedio menor. Es por ello, que en la Sección 7.1.5 se utiliza este valor en los análisis de la solución implementada.

7.1.5. Solución simulada

En esta sección se exponen los resultados finales utilizando la solución final en el ambiente simulado. En esta sección se trabajan con 3 robots, utilizando scan_filter, el algoritmo *custom* para agrupación de fronteras y 6 segundos para la espera de *bidders*.

En la Figura 7.8 se presenta una de las pruebas realizadas con esta configuración para el mapeo del InCo. En la figura se pueden ver las diferentes etapas de la experiencia, como los robots interactúan para explorar colaborativamente el entorno y la construcción autónoma del mapa. En la Sección 7.2.6 se describe en mayor grado de detalle cada una de las diferentes etapas para el caso de la implementación en el robot Rogel.

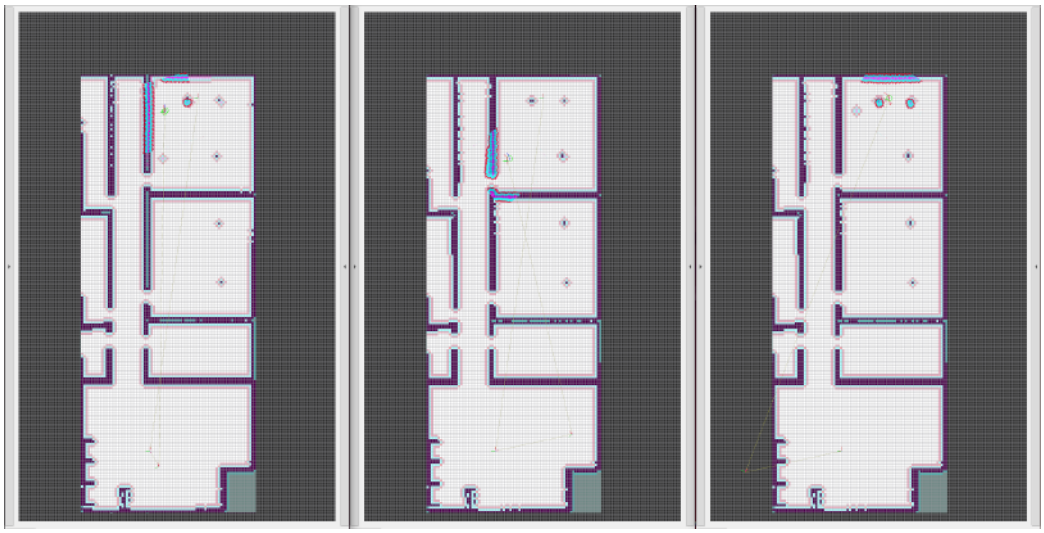


Figura 7.8: Gif de la construcción colaborativa de un mapa en el entorno simulado del InCo

Como se puede ver en la Figura 7.8, los robots son capaces de colaborar y lograr el objetivo de generar un mapa del entorno. Este es el resultado de las simulaciones de mayor importancia para el proyecto, ya que corrobora que la solución

implementada es capaz de conducir una flota de robots para que logre realizar un mapa del entorno de manera colaborativa.

7.2. En Rogel

Una vez realizadas las distintas pruebas utilizando el simulador, se procedió a validar los resultados en el mundo real utilizando a Rogel. Se llevaron a cabo una serie de experimentos de los que se desprenden características propias del trabajo desarrollado.

Las pruebas que se presentan para el entorno real son:

- En la Sección 7.2.1 se evalúa el resultado de la implementación de odometría descrita en la Sección 4.2.1.4.
- En la Sección 7.2.2 se presentan los resultados al utilizar el nodo scan_filter introducido en la Sección 5.2.2.1.2 teniendo en cuenta las consecuencias que tiene en los mapas.
- En la Sección 7.2.3 se exhiben los resultados que se obtienen al ejecutar el algoritmo de SLAM.
- En la Sección 7.2.4 se trabaja con el algoritmo de navegación y se documentan pruebas realizadas con el fin de validar su comportamiento.
- En la Sección 7.2.5 se describe el comportamiento del sistema al conformar una flota de robots.
- Finalmente en la Sección 7.2.6 se analiza la solución final al correr todo el sistema desarrollado sobre una planta del Instituto de Computación y se caracteriza el resultado obtenido utilizando distintas métricas.

7.2.1. Odometría

Para determinar el desempeño del algoritmo de odometría se llevó a cabo un experimento que consistió en realizar un cuadrado con el robot partiendo de un punto conocido donde se imponen velocidades durante cierto tiempo y se mide la ubicación de los vértices que quedan determinados. De esta forma se puede comparar lo que es esperado que retorne el algoritmo implementado y lo que en la realidad se obtiene. Con esta experiencia es posible evaluar tanto el control de velocidades del robot como el algoritmo que evalúa la odometría.

Se le enviaron al robot comandos de velocidad por cierto tiempo de forma tal que avance 1 m hacia adelante, luego 1 m hacia la izquierda, seguido de 1 m hacia atrás, finalizando con 1 m hacia la derecha. Realizando esto se tiene que el robot debería lograr formar un cuadrado en su trayectoria, sin embargo, se obtuvo la forma que se observa en la Figura 7.9.

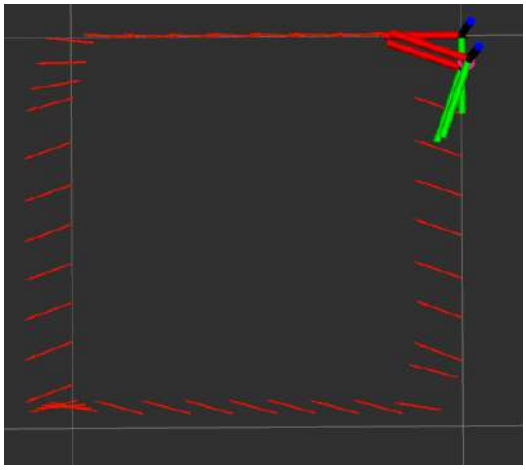


Figura 7.9: Trayectoria realizada por el robot cuando se quiere realizar un cuadrado.

Se puede apreciar en la Tabla 7.5 donde se comparan los valores esperados, los medidos y los resultantes del algoritmo de odometría, que el robot comete un RMSE de 35,24 % entre esperados y medidos, y un RMSE de 29,88 % entre medidos y odometría.

Esperado X	Esperado Y	Medido X	Medido Y	Odom X	Odom Y
1.00	0.00	0,92	0,01	0,9300	0,0010
1.00	1.00	0,69	0,92	1,0002	0,9186
0.00	1.00	-0,18	1,23	0,0676	0,9636
0.00	0.00	-0,58	0,47	-0,0018	0,0811

Tabla 7.5: Comparativa entre los vértices medidos y los resultados obtenidos por odometría.

Como se puede analizar de los resultados de la Tabla 7.5, el primer error refiere al manejo de las velocidades del robot, ya que cuando se impone una velocidad, este no logra alcanzar el objetivo. Por otro lado se tiene el error entre los valores medidos y la odometría. Este se debe a que la odometría no es un fiel reflejo del movimiento del robot, sino que es en general más cercano al valor ideal que al resultado obtenido.

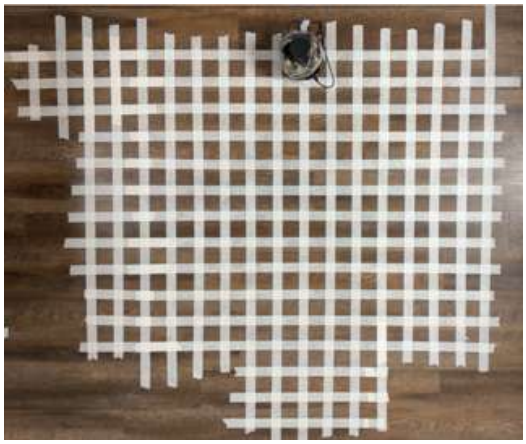


Figura 7.10: Gif de la prueba del cuadrado realizada.

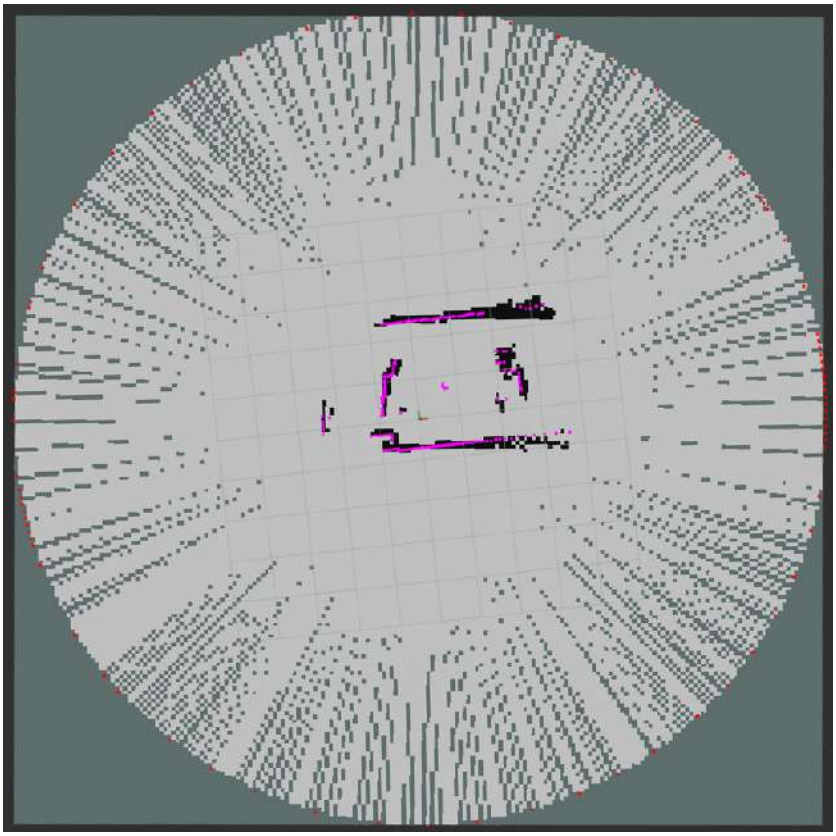
Es pertinente resaltar que si bien el resultado de la odometría no es bueno y este es utilizado en el algoritmo de SLAM, en la práctica esta información se complementa con los que se conoce como *scan matching*[1]. Es decir, se consideró adecuado tener en cuenta el *scan matching* para la estimación de la ubicación debido a los resultados que se obtuvieron de la odometria en esta experiencia.

7.2.2. Manejo del LiDAR

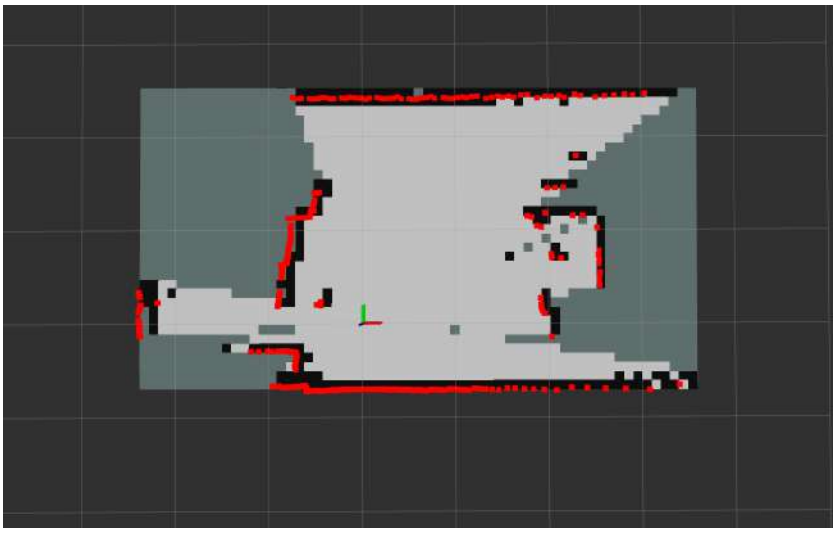
De manera análoga a como se procedió en el simulador, se busca evaluar los distintos resultados que se tienen al utilizar o no el filtro implementado en el nodo *scan_filter* y su repercusión a la hora de generar mapas. La única diferencia entre las pruebas realizadas en la simulación y las del mundo real es que no se cuenta con un ambiente lo suficientemente grande en el que se observe el comportamiento que se tiene en la simulación al utilizar el Open World. No obstante, se realizaron mapas de una planta del InCo utilizando el filtrado y sin utilizarlo obteniendo conclusiones a partir de estos.

Teniendo en cuenta lo visto en la Sección 7.1.1 resulta intuitivo pensar que utilizar el filtro puede mejorar los resultados de los mapas que se generan. Sin embargo, al trabajar con Rogel el mapa al utilizar el *scan_filter* resulta ser malo, en la Figura 7.11 se muestra una comparativa entre el mapa utilizando el filtro contra un mapa que no se utiliza.

[1] *Scan matching* es una forma de estimar la posición del robot a partir de las medidas del LiDAR.



(a) Con *scan_filter*



(b) Sin *scan_filter*

Figura 7.11: Comparativa de mapas utilizando o no el nodo *scan_filter*.

Como se puede apreciar, el resultado que se ve en la Figura 7.11a no es el esperado y esto se debe a la alta presencia de valores *'inf'* en las medidas del LiDAR que luego del filtrado son convertidos a medidas de valor el rango máximo

admisible por el dispositivo. El porcentaje de medidas con valores *'inf'*, como se menciono en la Sección 3.3.4, representa en una serie de medidas del LiDAR un mínimo de 30 %.

En base a los resultados anteriores, el uso del nodo *scan_filter* fue descartado para el caso del robot físico dado que como se observa en la Figura 7.11b, el mapa que se genera sin el nodo es una representación del entorno más fiel.

Como se puede observar, en la Figura 7.11b, no está presente el círculo que se tiene en la Figura 7.11a, sin embargo, en las próximas secciones se verá el comportamiento que se estudió durante la simulación cuando no se utiliza el nodo, es decir un mapa no convexo.

En conclusión, si bien conceptualmente resulta claro la necesidad de utilizar un filtrado en las medidas no numéricas devueltas por el sensor, se constató en la práctica que no es posible su uso debido al hardware que se utiliza. Por otro lado, se tiene una limitación en la solución implementada ya que no es posible realizar mapas si se parte de un ambiente muy amplio, es decir, que hacia todas las direcciones la distancia a un objeto sea mayor a lo que el LiDAR puede medir. Esto se debe a que no se logra solucionar el comportamiento descrito en la Sección 7.1.1. Cuando el robot se encuentra en un ambiente libre de objetos en un radio mayor al rango máximo de sensado del LiDAR, con la solución implementada, el robot no se movería.

7.2.3. SLAM

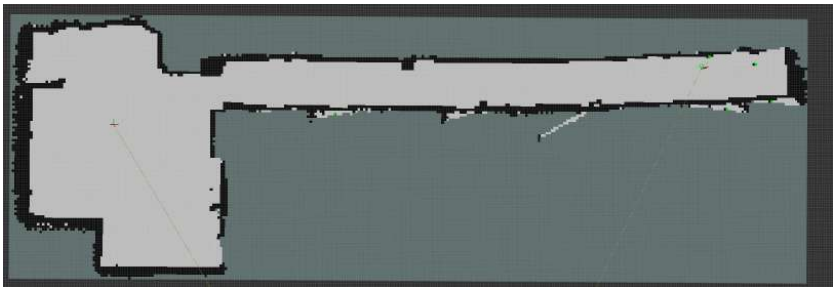
En esta sección se presentan dos experiencias distintas que tienen como objetivo caracterizar el algoritmo de SLAM utilizado. En la Sección 7.2.3.1 se presentan los resultados de las pruebas donde se caracteriza el mapa construido y en la Sección 7.2.3.2 los resultados de las pruebas donde se busca caracterizar la localización del robot en el mapa.

7.2.3.1. Mapping

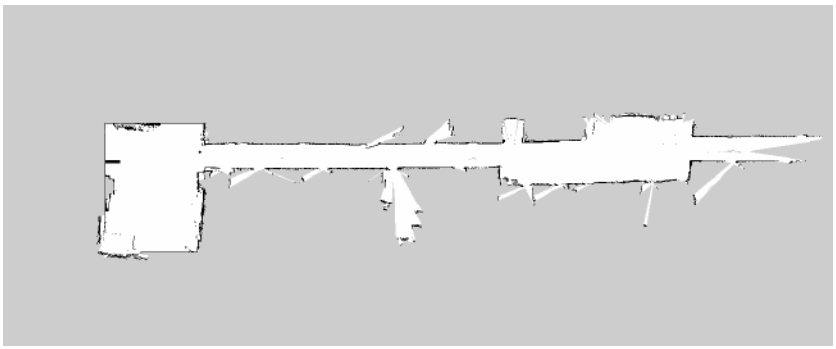
Debido a que el principal objetivo del robot es en último término construir un mapa del entorno que lo rodea, en esta sección se propone validar y cuantificar la precisión con la que se realiza la exploración.

Se construyó un mapa de la planta baja del Instituto de Computación, el cual ya contaba con un mapa de referencia generado por docentes de dicho instituto e incluso planos para poder comparar.

En la Figura 7.12 se puede comparar el mapa que se relevó utilizando los algoritmos y el robot implementado en este proyecto con el mapa que se tenía como referencia. En particular, la Figura 7.12a se presenta el mapa relevado con Rogel, mientras que en la Figura 7.12b se incluye el mapa de referencia brindado por los docentes del InCo.



(a) Generado por Rogel



(b) Generado por docentes del InCo

Figura 7.12: Mapas del InCo

Como se puede observar en la Figura 7.12 a simple vista el algoritmo implementado devuelve resultados similares a los obtenidos por grupos anteriores, se debe tener en cuenta que los mapas no son idénticos pues el de la Figura 7.12a se agregaron obstáculos para detener el recorrido. Igualmente, con el fin de cuantificar los errores del mapa generado, en la Figura 7.13 detallan puntos que se tuvieron en cuenta para cuantificar el error.

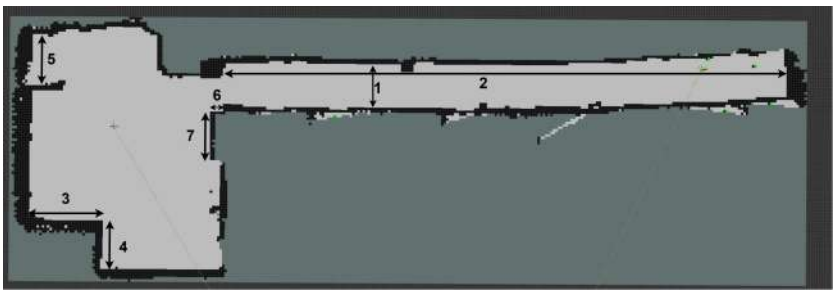


Figura 7.13: Mapa generado con referencia de medidas consideradas

En la Figura 7.13 se detallan diferentes medidas de referencia con el fin de comparar el mapa generado con las medidas. En la Tabla 7.6 se exponen los resultados obtenidos para el mapa generado y las medidas realizadas.

Número medida	Medición (cm)	Mapa (cm)	Diferencia porcentual (%)
1	151	160	6
2	2220	2220	0
3	282	290	3
4	179	190	6
5	177	190	7
6	47	50	6
7	181	190	5

Tabla 7.6: Comparación entre las medidas de referencia del plano y las obtenidas del mapa generado.

Teniendo en cuenta todas las medidas, se obtiene un error cuadrático medio de $RMSE = 7{,}22\ cm$. Este error representa un 14 % y 0.3 % de la menor distancia medida (47 cm) y la mayor distancia medida (22.2 m) respectivamente.

7.2.3.2. Localización

En esta sección se busca validar el otro componente del SLAM, la localización del robot en el mapa. Para ello, se propone un experimento donde se compara la supuesta ubicación del robot en el mapa contra determinados objetos del entorno.

En el experimento, consta de mapear de manera teleoperada al robot en un entorno de la empresa Focus. En esta experiencia, en determinados momentos se detuvo al robot y se realizaron medidas de distancia hacia objetos cercanos con el objetivo de referenciar su posición. Es importante aclarar que este experimento logra solamente comparar la posición del robot en coordenadas (x,y) pero no se tiene en cuenta el ángulo de rotación que posee el robot.

Los puntos en los cuales se tomaron las medidas se pueden observar en la Figura 7.14 mientras que los valores comparativos entre las medidas tomadas y las devueltas por el algoritmo de SLAM se presentan en la Tabla 7.7.

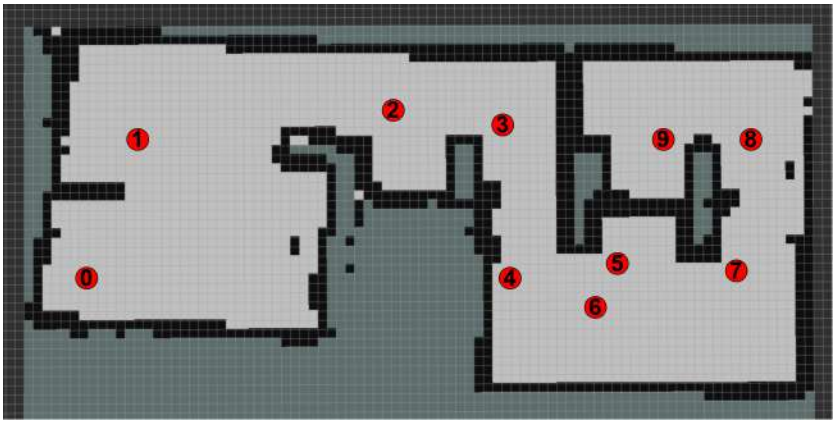


Figura 7.14: Mapa con el cual se evaluó la localización del robot marcando las ubicaciones medidas.

Ubicaion	Dirección	SLAM (cm)	Manual (cm)	Diferencia porcentual (%)
0	$-\hat{X}$	50	51	2
0	$-\hat{Y}$	50	48	4
1	$\hat{X}$	150	140	7
1	$\hat{Y}$	100	105	4.7
2	$\hat{X}$	170	171	0.6
2	$\hat{Y}$	60	55	9
3	$\hat{X}$	50	42	19
3	$\hat{Y}$	60	68	11.7
4	$-\hat{X}$	20	25	20
4	$-\hat{Y}$	110	115	4.3
5	$\hat{X}$	60	70	14.3
5	$-\hat{Y}$	130	140	7
6	$\hat{X}$	210	217	3.2
6	$-\hat{Y}$	80	86	7
7	$\hat{X}$	60	69	13
7	$-\hat{Y}$	120	126	4.7
8	$\hat{X}$	50	48	4
8	$\hat{Y}$	80	85	6
9	$\hat{X}$	30	27	11
9	$\hat{Y}$	80	87	8

Tabla 7.7: Comparación entre la localización del robot y las medidas manuales tomadas.

Nuevamente se puede obtener una medida de error de la localización utilizando

el error RMS. En este caso se obtiene que $RMSE = 6{,}4cm$

7.2.4. Navegación

En esta sección se evalúa el comportamiento del algoritmo de navegación en el robot. Para ello se realizó el siguiente experimento: Con un mapa realizado o parcialmente realizado se le enviaron objetivos al robot utilizando la interfaz gráfica rviz. En el experimento se evalúa principalmente que el robot logre alcanzar el objetivo y que lo alcance sin colisionar con el entorno.

Para ello, el algoritmo de navegación construye un plan para alcanzar cierto objetivo. Además, se construyen los mapas de costos para evitar obstáculos y alcanzar las diferentes metas intermedias.

En esta práctica se registró que para todas las posiciones objetivos enviadas al robot, se computaba un camino y se alcanzaba el objetivo si este se encontraba dentro del espacio libre y conocido. Pero en los casos donde se enviaban objetivos en los cuales el robot debería colisionar, por ejemplo un objetivo dentro de un armario o una pared, el algoritmo descartaba el objetivo.

Como conclusión de esta prueba se obtiene que el robot es capaz de navegar en el mapa generado por el algoritmo de SLAM de forma segura y eficiente sin colisionar con objetos y alcanzando las metas propuestas.

7.2.5. Colaboración

En esta sección se presentan los resultados obtenidos cuando se emplea la solución implementada en varios robots en simultáneo, conformando una flota de robots. Para llevar a cabo tal tarea, se deben tener todos los robots conectados a una misma red de forma que se puedan comunicar entre ellos para tomar decisiones y colaborar en la generación de un único mapa.

Al momento de realizar el mapeo colaborativo se colocaron los robots próximos entre ellos en un punto de un ambiente y se los fue configurando uno a uno. Para esto se conectó mediante una PC por SSH a cada uno de los robots y se fue levantando los distintos *launchfiles* necesarios para la ejecución de la tarea.

El primer robot se logró configurar de forma correcta sin problema alguno, pudiendo correr todos sus nodos. Sin embargo, a la hora de configurar el segundo robot, la conexión SSH establecida entre los robots y la PC comenzaba a fallar, respondiendo de forma lenta, a tal punto de perder completamente la conexión con los robots.

Al perderse la conexión remota la primera intuición fue considerar que el Odroid no logró procesar los paquetes que se encontraban corriendo y podría estar consumiendo memoria RAM en exceso o utilizando el 100 % de la CPU o incluso haberse apagado. Para descartar esta posibilidad se conectaron los robots a un monitor y se confirmó que ambas SBC continuaban encendidas y en condiciones normales, es decir, no estaban trabajando de forma exigida en torno a sus valores límites de memoria RAM y CPU.

Bajo la sospecha de que sea un problema de saturación de la red por la cantidad

de mensajes se quiso aislar los robots dentro la red WiFi a la que se estaban conectando. Aún así, habiendo hecho esto, el problema persistía y la conexión para configurar y poner a correr los robots se seguía perdiendo. Esto podría deberse a que la red aun siguiera saturada, ya que se tienen por cada robot corriendo una gran cantidad de tópicos y en ROS2 se inunda la red con los mensajes de todos ellos. Para mitigar este comportamiento se decidió buscar métodos que permitan publicar hacia la red únicamente los tópicos de interés, es decir, aquellos que sean necesarios de compartir entre los robots para las decisiones que deban tomarse y para lograr la correcta generación del mapa. No se encontró una forma de elegir qué tópicos se envían a la red visible para los otros equipos y cuales se envían sólo a nivel local.

Se encontró un resultado más próximo que se aplica a la totalidad de un nodo, esto es, que se envíen todos sus tópicos o ninguno a la red. Para esto se utiliza una variable de entorno al ejecutar el nodo que permite que el envío de mensajes se haga de forma local. Sin embargo, se encontró un obstáculo pues la posibilidad de utilizar esta variable solamente la permitía el vendor DDS FastDDS, lo cual genera un problema aún peor, ya que con dicho vendor no se puede realizar la exploración ni siquiera utilizando un solo robot debido a que lo mensajes no logran llegar a la PC remota. Fue por dicho motivo que se utilizó como vendor a CycloneDDS.

Con el fin de encontrar la causa del problema se intentó de nuevo realizar el mapeo pero esta vez levantando los nodos de a uno para poder identificar cuál de todos es el que genera la saturación de la red. No se logró determinar qué nodo era el causante del problema ya que diferentes combinaciones resultaban en la desconexión remota.

Teniendo en cuenta todo lo planteado en esta sección se optó por continuar el trabajo realizando las pruebas y mapas utilizando únicamente un robot. A pesar de esto, se siguió utilizando la solución implementada ya que contempla el caso de un solo robot y de esta forma poder poner en práctica las distintas etapas del algoritmo.

7.2.6. Solución Implementada

En esta sección se exponen los resultados que se obtienen al correr todo el sistema desarrollado utilizando la solución final implementada en el mundo real. Se estudia el caso del mapeo autónomo de una planta del Instituto de Computación.

Se recorrerán las diferentes etapas del algoritmo mostrando los detalles más relevantes en cada caso y representando de forma visual lo adquirido en el mapa. Partiendo de un punto en un ambiente del subsuelo del InCo, se ejecutan los *launchfiles* correspondientes y se deja en marcha la ejecución del sistema de mapeo y exploración. El nodo de SLAM se encuentra de forma periódica leyendo los mensajes provenientes de los nodos del *launchfile icre_robot*, generando a partir de estos un mapa. De esta forma, una vez inicializado el sistema, previo a realizar algún movimiento, ya se tiene un mapa como el que se observa en la Figura 7.15. Es con dicho mapa que se comienza a alimentar el algoritmo para determinar las fronteras, luego agruparlas, calcular los centroides a partir de ellas, asignarlos a

los robots y navegar hasta llegar al objetivo y comenzar el proceso de nuevo todo mientras se mapea el ambiente.

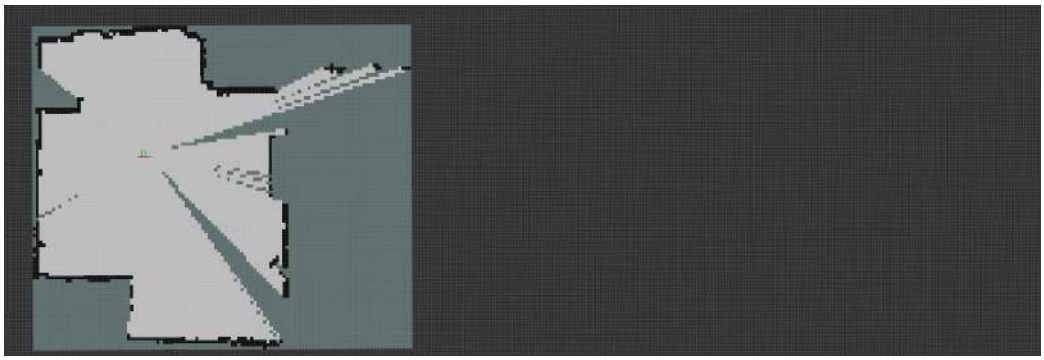


Figura 7.15: Mapa inicial previo a comenzar la exploración autónoma.

A continuación se describen en detalle los resultados obtenidos en los pasos mencionados anteriormente.

Siguiendo el procedimiento descrito en la Sección 5.2.5.1, se obtienen a partir de la grilla de ocupación que se tiene hasta el momento los puntos frontera como los que se aprecian en la Figura 7.16. Se puede observar en la imagen que las fronteras (puntos rojos) son aquellos puntos limítrofes entre lo conocido (color blanco) y lo desconocido (color gris).

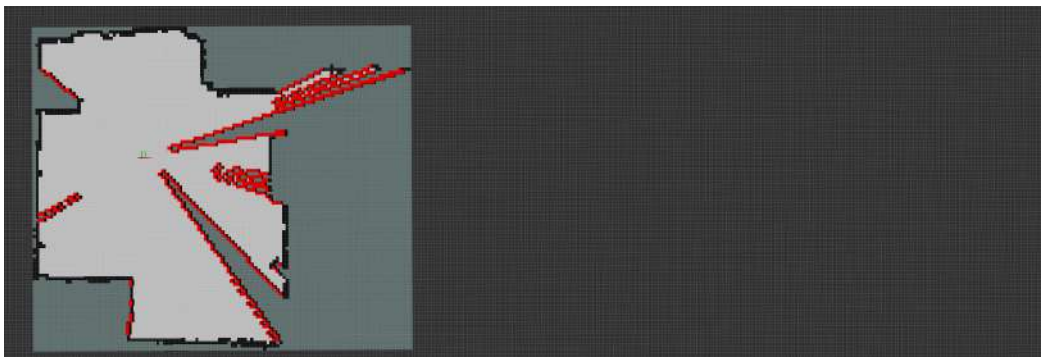


Figura 7.16: Fronteras calculadas a partir de la grilla de ocupación junto con el mapa generado.

Se quieren agrupar las fronteras calculadas en diferentes *clusters* para poder sectorizar los espacios a donde debe dirigirse el robot para explorar y que no se tengan objetivos muy cercanos. El método utilizado, también presentado en la Sección 5.2.5.1, es el algoritmo *'custom'* creado por el equipo. El resultado que devuelve dicho método se puede observar en la Figura 7.17 donde con distintos colores se marcan los grupos de fronteras.

Una vez obtenidos los grupos de fronteras, se calculan los centroides de los mismos. Los centroides resultantes del algoritmo *'custom'* a partir de la configuración de fronteras dada se encuentran marcados como puntos verdes en la Figura

7.17. Como se puede observar, existen casos donde el centroide no coincide con un punto frontera debido a la implementación realizada. Otro detalle es que no se tienen centroides próximos entre sí. Esto último es bueno para que, en caso de utilizar más de un robot, estos no tengan la posibilidad de ir hacia sectores cercanos aprovechando así el contar con más de un robot para cubrir más espacio explorado.

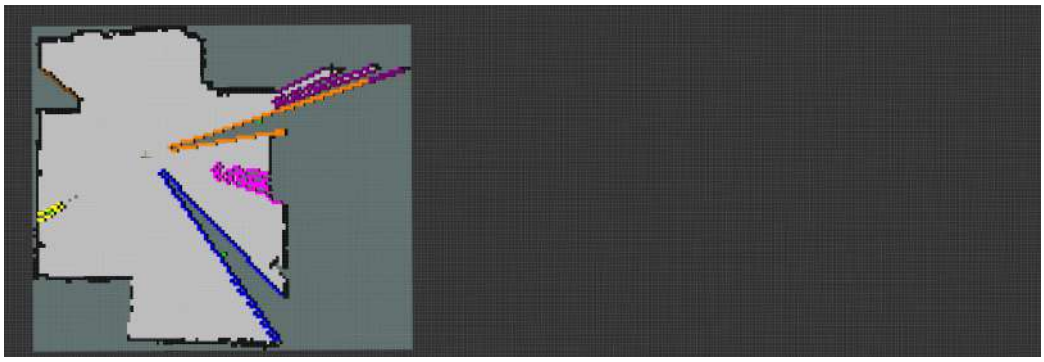


Figura 7.17: Agrupación de fronteras obtenidas a partir del algoritmo *'custom'*.

De los centroides calculados, se le asigna al robot como meta a donde dirigirse aquel punto centroide que se encuentre más cerca de la posición que tiene el mismo en ese momento. Para ello se efectúa el método de subasta introducido en la Sección 5.2.5.2.

Ya con el objetivo marcado, el robot procede a navegar hacia este destino siguiendo cierto *path* trazado por el paquete Nav2 como se muestra en la Figura 7.18. Mientras el robot avanza y se realiza la exploración, se descubren nuevos sectores del entorno lo que luego resulta en nuevas fronteras. Una vez que el nodo de navegación determina que el robot cumplió su tarea y llegó al objetivo, se procede a realizar nuevamente todos los pasos descritos.

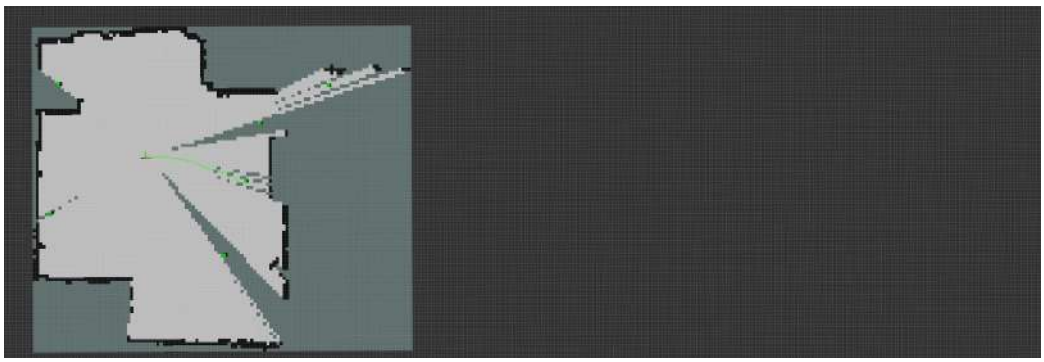


Figura 7.18: Recorrido trazado por el nodo de navegación para que el robot llegue al objetivo.

Finalmente, se adjunta un recorrido realizado por Rogel donde se muestra la construcción ilustrando cada uno de los pasos de la solución para generar el mapa total del subsuelo del InCo, Figura 7.19.

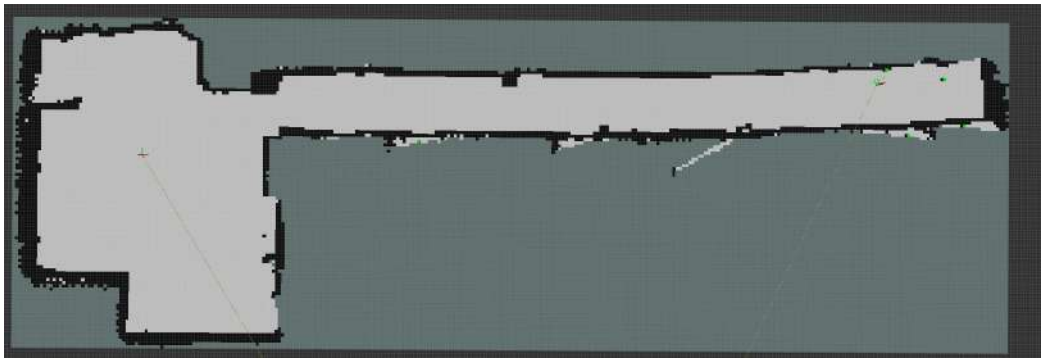


Figura 7.19: Mapa del subsuelo de InCo generado utilizando la solución desarrollada por el equipo.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 8

Conclusiones

En este capítulo se presentan las conclusiones más relevantes de la realización del proyecto. En la Sección 8.1 se presentan conclusiones generales que refieren más al manejo de un proyecto, formas de trabajo e incluso también con métodos para mitigar riesgos y gestión de proyecto. Por otra parte en la Sección 8.2 se presentan las conclusiones técnicas del proyecto. Aquí se reflexiona sobre los aspectos técnicos, se analizan los objetivos iniciales y en particular en la Sección 8.2.1 se proponen mejoras y sugerencias para futuras iteraciones en la temática.

8.1. Conclusiones Generales

Como conclusión general más importante se destaca la gestión del proyecto. Con esto se quiere decir que el equipo logró gestionar un proyecto de larga duración, siguiendo las metodologías de trabajo introducidas en el curso introductorio, se mantuvo el buen relacionamiento entre las diferentes partes interesadas en el proyecto y también se cumplieron en general los objetivos planteados.

Con respecto a la forma de trabajar se siguió una metodología ágil con reuniones diarias entre los integrantes del equipo y reuniones cada dos semanas con los tutores del proyecto a los que llamaremos *sprints*. Esta forma de trabajo por *sprints* generó un lazo de realimentación recurrente entre el avance del proyecto y los objetivos parciales planteados por los docentes a cargo. Una práctica habitual que aportó grandes resultados fue la de generar un reporte con los resultados del *sprint*, donde se evaluaban los objetivos planteados, los resultados obtenidos y se analizaba el progreso del *sprint*. Estas prácticas resultaron en un trabajo que si se compara con lo planificado originalmente difiere en menos de un mes de trabajo aún cuando el alcance del proyecto incrementó con respecto al original.

Además, debido a la periodicidad de las reuniones se logró atacar en tiempo y forma la gran mayoría de inconvenientes o problemas detectados. Por ejemplo, al detectarse que el control de bajo nivel de la plataforma motora no estaba completamente desarrollado para esta aplicación, se decidió trabajar en ello logrando incluso contribuir con los responsables del proyecto Robotito, si bien nada de esto se encontraba incluido dentro del alcance inicial. Otros ejemplos pueden ser la im-

portación de materiales como los PCB o el diseño de elementos estructurales, que en un principio no se tuvieron en cuenta, pero al detectarse se resolvieron gracias a ese lazo de realimentación corto del trabajo.

En cuanto al trabajo del equipo, dado los diferentes perfiles humanos y técnicos de los integrantes del equipo, resultó vital el trabajo en equipo y la correcta división de tareas para llegar a buen puerto. Teniendo en cuenta que este proyecto nucleaba diferentes áreas de la ingeniería, la división de trabajo acorde a los conocimientos de cada integrante fue esencial. Resulta importante aclarar que esto no quiere decir que se haya dividido el trabajo y se haya evitado la interacción entre las partes, sino que al contrario. Al asignar como responsable de cada tarea al integrante con mayor conocimientos de la tarea, se generaron discusiones constructivas y ámbitos de interacción en las reuniones diarias. Allí el responsable reportaba avances, y entre todos los integrantes se trataban dificultades y diseñaban soluciones.

8.2. Conclusiones Técnicas

Con respecto a las conclusiones técnicas se puede afirmar que se logró ensamblar y desarrollar una flota de 3 robots con capacidades de mapeo y navegación autónoma. Estos robots se construyeron adaptando una plataforma robótica desarrollada por el instituto de computación de bajo costo.

Para estos robots se pudo implementar un algoritmo de control basados en controladores PID para imponer movimientos controlados a los robots.

El modelado físico matemático considerado para el sistema se considera adecuado, ya que permitió que el robot se desplace de forma controlada gracias a la implementación del controlador PI y además se logró obtener información de odometría confiable a partir de la integración del movimiento angular de las ruedas.

La utilización de los paquetes de SLAM y navegación fue exitosa. El primero, slam_toolbox permite al robot generar mapas con un RMSE de 7.22 cm, mientras que se localiza en ellos con un RMSE de 6.4 cm. Por otro lado, el algoritmo de navegación Nav2, demostró que lograba dirigir al robot desde una posición dada hacia un objetivo satisfactoriamente. Esto se puede concluir ya que el robot fue capaz de generar trayectorias y recorrerlas de manera autónoma sin registrar colisiones.

Respecto a los algoritmos de asignación de tareas se puede concluir que se generó un algoritmo y mecanismo de subastas que le otorga a una flota robótica la característica de autónoma a la hora de realizar el mapeo colaborativo. Es decir, se diseñó y desarrolló un algoritmo de asignación de tareas descentralizado capaz de distribuir y asignar tareas a los diferentes robots de una flota para que obtengan objetivos concretos de mapeo. Para ello fue necesario también contar con métodos de agrupación de fronteras, entre los cuales se destaca uno desarrollado e implementado también por el equipo.

Con respecto al aspecto simulado, se concluye que es una herramienta muy valiosa a la hora de desarrollar soluciones robóticas ya que permite implementar y probar soluciones en un ambiente controlado. Sin embargo, no se pueden tomar como pruebas reales ya que a la hora de trabajar con robots físicos se observan

otros problemas como los reportados para la comunicación, la odometría deja de ser exacta, entre otros efectos de la realidad que no se consideran en la simulación.

Como conclusión final se puede decir que el robot construido y la solución de software implementada en él, cumplen con los objetivos propuestos y resuelven problemas no planteados inicialmente. Esta solución tiene un verdadero potencial para ser usado en aplicaciones relacionadas a la arquitectura, para mapear plantas en tiempos reducidos o cualquier actividad que requiera de un mapa de un entorno cerrado.

8.2.1. Trabajo futuro

Con respecto al trabajo futuro o posibles mejoras al proyecto se considera como primer punto la resolución de los problemas detectados en la comunicación. Se recomienda especialmente trabajar en estos temas ya que le agrega un salto de calidad a la solución, como se ve en las comparaciones de tiempos en las simulaciones de la Sección 7.3.

Si bien se exploró la utilización de QoS, diferentes configuraciones de los vendors del DDS de ROS2 e incluso el equipo se puso en contacto con desarrolladores reconocidos de ROS2, se considera de gran importancia lograr la comunicación entre los robots físicos. Vale la pena analizar si el problema no se encuentra por el lado del hardware, ya que en las pruebas simuladas no se detectaron problemas en la asignación ni comunicación.

Otra mejora para el manejo del robot sería configurar el controlador del *stack* de navegación para que aproveche las capacidades holonómicas del robot. Este problema se intentó atacar en la ejecución del proyecto, pero se descartó ya que no era prioritario.

Con respecto a la asignación de tareas se pueden trabajar varios puntos para mejorar el algoritmo. A la hora de elegir las fronteras, se recomienda considerar la distancia como camino al objetivo y no la distancia euclidiana como se considera en la implementación actual. Se recomienda mejorar el algoritmo que genera los objetivos para que no tenga en cuenta objetivos cercanos a menos del rango máximo del láser entre ellos. También se propone utilizar los objetivos del resto de los robots a la hora de determinar el de uno en particular, nuevamente con el objetivo de no enviar a dos robots hacia el mismo objetivo. Finalmente, se sugiere modificar el algoritmo *custom* para que el objetivo resultante pertenezca al conjunto de fronteras.

Con respecto al estudio del algoritmo *custom* se propone además, realizar un estudio para evaluar a fondo sus características y particularidades. Por ejemplo, siguiendo el trabajo realizado en [8], considerar el tiempo de procesamiento del algoritmo en función de: la cantidad de fronteras de entrada, la cantidad de tareas, el tiempo total de exploración y también en distintos entornos.

En cuanto a las simulaciones, se concluye que puede ser importante generar un modelo de gazebo para la plataforma motora. A la cual se le pueden agregar sensores luego, pero dado que Robotito es una plataforma educativa y se puede utilizar como plataforma de aprendizaje se cree que tener la herramienta para

simular sería de gran ayuda para los trabajos futuros.

Como se mencionó en la Sección 4.2.1.5, la odometría que se calcula en la ESP no posee realimentación del algoritmo de localización que se computa en la Odroid. Se considera que una potencial mejora es enviar hacia el ESP además de los comandos de velocidad, información de localización para reducir los errores cometidos por la odometría. Esta mejora se basa en que el algoritmo de localización, que se realiza en la Odroid, utiliza *scan matching* para determinar la ubicación del robot y con ello se reducen los errores acumulativos en la odometría.

Finalmente, con respecto al hardware utilizado se puede concluir que si bien en grandes rasgos la solución es robusta y cumple con los requerimientos mínimos. Dado los componentes que se le agregaron a robotito y por ende el peso que aumentó, se puede considerar trabajar con motores con un mayor torque y menor velocidad ya que los motores en todo momento trabajan exigidos y por ende se desgastan y consumen más de lo que se obtendría con motores mejor dimensionados. Siguiendo con esta línea, motores con mayor torque además posibilitaron al robot a sortear mayores obstáculos como rendijas o piedras y cables que se encuentren en el ambiente. Con respecto a las plataformas de cómputo y sensado vale la pena estudiar el comportamiento del lidar en detalle, pues resulta sorprendente el comportamiento detectado en el proyecto. Mientras que para nuevas versiones se recomienda trabajar con una SBC de al menos 4 GB de memoria RAM, ya que por momentos se debió asignar memoria swap por problemas de memoria.

Apéndice A

freeRTOS y ESP-IDF

En este anexo se introduce al lector en mas detalle sobre los conceptos e implementación de freeRTOS y de como ESPRESSIF modificó la implementación de freeRTOS para utilizarlo en el microcontrolador ESP32.

A.1. freeRTOS

FreeRTOS es un sistema operativo en tiempo real (RTOS) diseñado específicamente para funcionar de manera eficiente en microcontroladores, aunque su uso no se limita a aplicaciones de microcontroladores.

Los microcontroladores se emplean en aplicaciones que generalmente tienen tareas específicas y dedicadas. Debido a las restricciones de tamaño y la naturaleza de estas aplicaciones, el uso de un RTOS completo no suele ser justificado o incluso posible. FreeRTOS proporciona funcionalidades básicas de programación en tiempo real como comunicación entre tareas, primitivas de sincronización y temporización.

En el contexto de un RTOS, una aplicación se estructura como un conjunto de tareas independientes. Cada tarea se ejecuta en su propio contexto sin depender de otras tareas en el sistema. La mayoría de los puertos de FreeRTOS están diseñados para trabajar con un solo núcleo de hardware, lo que implica que solo una tarea se ejecuta a la vez. Para lograr la ejecución de múltiples tareas, se implementa un *scheduler* o planificador.

El *scheduler* es responsable de determinar qué tarea se ejecuta en cada momento. Una tarea puede ejecutarse y suspenderse varias veces a lo largo de su ciclo de vida. El *scheduler* utiliza una política para decidir qué tarea se ejecuta en un momento dado. En sistemas de tiempo real, se ajusta la política para cumplir con las restricciones de las distintas tareas. Dado que una tarea no tiene conocimiento de las actividades del *scheduler*, es responsabilidad del *scheduler* asegurarse de que el contexto del procesador (registros, contenido de la pila, etc.) se mantenga exactamente igual cuando se cambia de una tarea a otra. Cada tarea tiene su propia pila, y cuando se produce un cambio de tarea, el contexto de ejecución se guarda en la pila correspondiente para poder restaurarlo de manera precisa cuando la tarea se reanuda posteriormente.

Además de ser suspendida voluntariamente por el kernel, una tarea puede optar por suspenderse. Una tarea bloqueada o inactiva no puede ejecutarse y no se le asigna tiempo de procesamiento. Los estados en los que puede encontrarse una tarea y las condiciones para que una tarea cambie de estado se muestran en la Figura A.1.

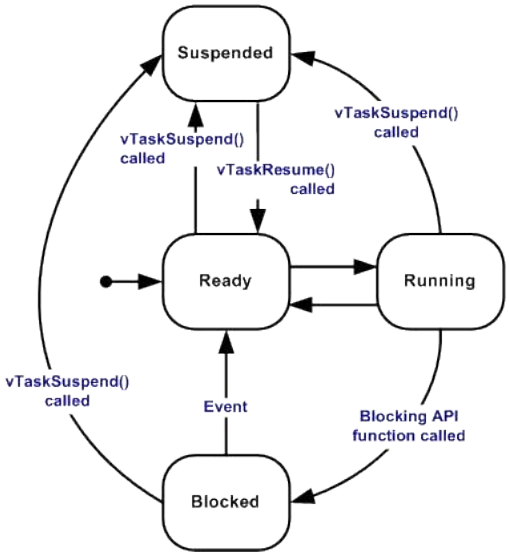


Figura A.1: Diagrama de estados de una tarea en freeRTOS. Extraído de [41]

El estado "*Ready*" indica que las tareas están listas para ejecutarse, pero están esperando porque una tarea de igual o mayor prioridad está siendo ejecutada. El estado"*Running*" indica que una tarea se encuentra en ejecución actualmente, es decir, está utilizando el procesador. En el estado "*Blocked*", se encuentran las tareas que están esperando un cierto periodo de tiempo o un evento externo. Las tareas también pueden estar en este estado si están esperando una cola, un semáforo, un evento de un grupo o una notificación. Normalmente, las tareas en estado "*Blocked*" tienen un tiempo de espera después del cual se desbloquean, incluso si el evento que estaban esperando no se ha producido. En este estado, las tareas no utilizan tiempo de procesamiento y no pueden ser seleccionadas para pasar al estado "*Running*".

Por último, las tareas pueden encontrarse en el estado "*Suspended*". Al igual que las tareas en estado "*Blocked*", las tareas en estado "*Suspended*" no pueden pasar al estado "*Running*". Sin embargo, las tareas en estado "*Suspended*" no tienen tiempo de espera. Las tareas solo entran o salen del estado "*Suspended*" cuando se les ordena explícitamente a través de llamadas a la API.

La comunicación entre tareas se realiza principalmente a través de colas y notificaciones. Las colas se utilizan para enviar mensajes entre tareas o entre tareas e interrupciones. En la mayoría de los casos, se utilizan como colas FIFO (primero en entrar, primero en salir), pero también es posible colocar mensajes al frente de la cola. Las tareas pueden bloquearse cuando intentan leer desde una cola vacía o escribir en una cola llena. Permanecerán en el estado "*Blocked*" hasta que los

datos estén disponibles en la cola o hasta que expire el tiempo de bloqueo. Si varias tareas están bloqueadas en la misma cola, la tarea con mayor prioridad será desbloqueada primero.

Otra forma de comunicación entre tareas son las notificaciones directas a las tareas. Cada tarea tiene una matriz de notificaciones que consta de un estado (pendiente o no pendiente) y un valor de 32 bits. Este tipo de notificaciones evitan intermediarios como colas o semáforos y se envían directamente a la tarea destinataria. El envío de una notificación de este tipo establece el estado de la notificación de la tarea destinataria como "pendiente". Al igual que una tarea puede bloquearse esperando la disponibilidad de un semáforo, una tarea puede bloquearse esperando que el estado de una notificación se vuelva pendiente. El envío de una notificación a una tarea también puede opcionalmente actualizar el valor de la notificación de destino.

A.2. ESP-IDF

ESP-IDF es una versión de FreeRTOS desarrollada sobre la base del vanilla FreeRTOS que admite multiprocesamiento. La principal diferencia entre ESP-IDF y Vanilla FreeRTOS es que Vanilla FreeRTOS está diseñado para ejecutarse en hardware que presenta un solo núcleo, sin embargo como el ESP 32 tiene 2 núcleos, por lo que admite la ejecución de 2 tareas en simultáneo ESP-IDF esta diseñado para poder realizar procesamiento en paralelo. Estos núcleos son idénticos en la práctica y comparten la misma memoria. Esto permite que los dos núcleos ejecuten tareas de manera intercambiable entre ellos.

En las tareas y la creación de tareas, el gran cambio viene al momento de crear una tarea y el contexto que es necesario guardar en el trabajo del scheduler. Cuando se crea una tarea, es necesario que se especifique en que CPU se va a ejecutar dicha tarea, este valor depende de la CPU, y existe la posibilidad de no asignar una afinidad y permitir que la tarea se ejecute en cualquier CPU. A su vez, este valor se debe guardar con el contexto de la tarea y por tanto el cambio de contexto requiere de mayor cantidad de memoria.

La otra principal diferencia se encuentra en la ejecución de las tareas en estado "Ready". En FreeRTOS existe una función que es responsable de seleccionar la tarea de mayor prioridad para ejecutar de una lista de tareas en el estado "Ready". En ESP-IDF FreeRTOS, cada núcleo llamará a esa función de forma independiente para seleccionar una tarea desde la Lista de tareas preparadas que se comparte entre ambos núcleos. Si bien ambos planificadores utilizan Round Robin, ESP-IDF FreeRTOS tiene la posibilidad de saltarse algunas tareas sin ejecutarlas. Esto se debe a que la lista de tareas se implementa como un arreglo donde cada entrada representa una prioridad y contiene una lista con el bloque de control de las tareas que tienen esa prioridad. Al ser una única lista, en cada bloque de control, se encuentra la CPU en la cual se tiene que ejecutar cada tarea, por lo que al momento de tomar una tarea, una CPU se fija si le corresponde o no ejecutar dicha tarea. Esto presenta una dificultad, que hace que una tarea asignada a una CPU pueda ser salteada por la otra CPU mientras busca qué tarea ejecutar. Esto se debe a

que la lista de tareas presenta en cada entrada un solo índice que dice cuál fue la última tarea ejecutada. Una solución al problema de saltear tareas es asegurarse de que todas las tareas entren en estado "Blocked" para que se eliminen de la Lista de tareas listas. Otra solución es distribuir las tareas en múltiples prioridades de modo que a una determinada prioridad no se le asignen múltiples tareas que estén dirigidas a diferentes núcleos. Existen otras diferencias que se pueden encontrar en [77].

A.2.1. Estructura de un proyecto en ESP-IDF

En ESP-IDF un proyecto puede verse como una combinación de varios componentes. Un "proyecto" es un directorio que contiene todos los archivos y la configuración para crear una sola "aplicación".

La "configuración del proyecto" se guarda en un archivo llamado sdkconfig en el directorio raíz del proyecto. Este archivo de configuración se modifica para personalizar la configuración del proyecto. Un solo proyecto contiene exactamente una configuración de proyecto.

Una "aplicación" es un ejecutable creado por ESP-IDF. Un solo proyecto generalmente creará dos aplicaciones: una "aplicación de proyecto" (el ejecutable principal, es decir, su firmware personalizado) y un *bootloader* (el programa que inicia la aplicación del proyecto).

Los 11componentes" son piezas modulares de código independiente que se compilan en bibliotecas estáticas y se vinculan a una aplicación. Estos componentes son los que le dan las funcionalidades a la aplicación principal para que pueda cumplir y realizar su objetivo.

Apéndice B

Validación de medidas con RPLidar

En este anexo se introduce al lector en más detalle sobre cómo recrear la experiencia descrita en la Sección 3.3.4. Se procede a explicar cómo utilizar el paquete controlador del dispositivo de medidas y el posterior trabajo a realizar con las medidas obtenidas para generar información de interés.

B.1. Set-up y procesamiento de datos

Previo a comenzar se debe situar el LiDAR con su ángulo 0^{o} apuntando hacia una pared y a la distancia que se quiera verificar el desempeño de medición.

Para trabajar con el sensor utilizando ROS2 primero se debe compilar el paquete que lo controla que se encuentra dentro de la carpeta /src dentro del *workspace* en el que se esté trabajando. Para ello, estando en directorio raíz del *workspace*, se debe ejecutar el comando `colcon build --packages-select rplidar_ros` siendo "rplidar_ros" el paquete que maneja los *drivers* del sensor.

Luego de compilado, para poder utilizar los ejecutables y librerías instaladas, se debe hacer un *source* del archivo generado en la compilación que ayuda a realizar la configuración del entorno ejecutando `. install/setup.bash`. Una vez preparado el ambiente, se debe correr: `ros2 launch rplidar_ros rplidar.launch.py`. Este comando levanta los *drivers* para trabajar con el LiDAR y publica en cierto tópico las medidas tomadas, generalmente en */scan*.

Para obtener una primera impresión de las medidas, en una nueva ventana de terminal se debe utilizar el comando: `ros2 topic echo /scan` y se desplegará en la pantalla de la terminal una serie de mensajes del tipo *LaserScan*. Estos mensajes se componen de varias partes, entre ellas se encuentran las correspondientes medidas del LiDAR y características del sensor como lo son los rangos máximos y mínimos de medidas, el período entre medidas tomadas, etc. En este caso, es de interés en principio obtener solamente la medida de una dirección en particular, habiendo elegido el ángulo cero por la disposición utilizada para la medida.

En ROS2 para obtener la medida de distancia únicamente para esta dirección se debe realizar un procesamiento de los datos[1], es así que se utiliza el comando

[1] A diferencia de ROS1 que permite acceder a un elemento en particular dentro del

`ros2 topic echo /scan --field ranges > nombre_archivo.csv` para obtener en un archivo de llamado "nombre_archivo.csv"todos los valores de medidas que se encuentran en el campo 'ranges' del tipo de dato *LaserScan*. El archivo en formato .csv generado contiene una serie de medidas en todas las direcciones (360 grados) y texto indicando que se trata de una estructura del tipo *array*. Se lo modificó eliminando la palabra "array" y los caracteres no numéricos que aparecían, luego utilizando un *script* de python se consideró únicamente el primer valor de cada lista de medidas ya que este corresponde a la medición en el ángulo 0^{o} y se realiza el histograma de las medidas. Se calculan también el valor medio y la desviación estándar de la serie de medidas.

array de medidas.

Apéndice C

Algoritmo de agrupación de fronteras

En este apéndice se detalla el algoritmo implementado para la agrupación de fronteras y se realiza un breve análisis de sus tiempos de ejecución y de su correctitud.

C.1. Algoritmo

La idea detrás del algoritmo es hallar todas las componentes conexas dentro del conjunto de fronteras.

Algorithm 3: AgruparFronteras

Input: fronteras

```
1 frontiers_copy = frontiers.copy();
2 groups = [];
3 while forntiers_copy is not empty do
4 |   frontera ← copia_fronteras;
5 |   grupo = ObtenerGrupo(frontera, copia_frontera);
6 |   grupos = grupos + grupo;
7 end
```

Result: *grupos*

El algoritmo 3 recibe como entrada las fronteras y realiza una copia de estas. Luego, itera hasta que no queden mas fronteras en el conjunto. Por cada iteración, toma una frontera, y calcula el grupo al que pertenece (es decir la componente conexa a la que pertenece), usando la función 4.

Algorithm 4: ObtenerGrupo

Input: frontera, fronteras

```
1 grupo = [frontera];
2 fronteras = fronteras − frontera;
3 vecinos = ObtenerVecinos(frontera);
4 for vecino in vecinos do
5 |   if vecino in fronteras then
6 |   |   aux ← ObtenerGrupo(vecino, fronteras);
7 |   |   grupo = grupo + aux;
8 |   end
9 end
```

Result: *grupo*

La función ObtenerGrupos recibe una frontera y una conjunto de fronteras. Crea un nuevo grupo para dicha frontera y la remueve del conjunto. Obtiene los vecinos (ejemplificados en la Tabla C.1) utilizando la función ObtenerVecinos e itera sobre ellos comprobando si pertenecen o no a la lista de fronteras. Para cada vecino que pertenece al conjunto, llama recursivamente a la función ObtenerGrupo, concatenando el resultado en el grupo original. Finalmente retorna dicho grupo.

(x-1,y+1)	(x,y+1)	(x+1,y+1)
(x-1,y)	**(x,y)**	(x+1,y)
(x-1,y-1)	(x,y-1)	(x+1,y-1)

Tabla C.1: Vecinos para una celda (x,y)

C.2. Orden

Para analizar el orden del algoritmo primero se analiza el de la función ObtenerGrupo. Primero, se remueve a la frontera del conjunto de fronteras, lo cual tiene orden lineal. Luego, se tiene que la función para obtener los vecinos es de orden constante, ya que retorna los 8 valores contiguos a la celda. De esta forma, el *for* se ejecuta siempre 8 veces. Para verificar si un vecino esta en la lista de fronteras, se recorre la lista, lo que tiene orden lineal. Por ultimo, para los casos en que el vecino pertenece, se llama a la función recursiva.

Por lo tanto, dado que al principio se elimina la frontera del conjunto, disminuyendo el conjunto para las siguientes recursiones, y que los ordenes en recursion son lineales en el tamaño del conjunto de fronteras, se tiene que el algoritmo es de orden n^2.

El algoritmo AgruparFronteras también presenta orden n^2 ya que llama a la funcion ObtenerGrupo, una vez por componente conexa.

C.3. Correctitud

Para mostrar la correctitud del algoritmo se deciden mostrar los siguientes puntos:

- Toda fronteras esta en un grupo
- No hay frontera que este en dos grupos
- Los grupos obtenidos son conexos
- Hay un grupo por cada componente conexo de las fronteras

C.3.1. Toda Frontera esta en un Grupo

El algoritmo de *AgruparFronteras* finaliza al salir del *while*, cuya condición es que no queden mas fronteras en el conjunto. En el único momento que se remueve una frontera del conjunto es en el algoritmo *ObtenerGrupos*, en el que previamente se incorpora dicha frontera al grupo que sera retornado por el algoritmo. Ese grupo es posteriormente incorporado al conjunto de grupos en el algoritmo *AgruparFronteras*. Por lo tanto, si el algoritmo finaliza es porque ya no hay mas fronteras en el conjunto, y por lo tanto las fronteras pertenecen a algún grupo.

C.3.2. No hay frontera que este en dos grupos

Cada vez que se agrega una frontera a un grupo, esta se remueve del conjunto de fronteras por lo que no puede volver a seleccionarse para otro grupo.

C.3.3. Los grupos obtenidos son conexos

Cada grupo se obtiene llamando a ObtenerGrupo. Dicha función inicializa un grupo con la frontera que recibe y luego lo completa concatenándole los grupos resultantes de llamar recursivamente a la función ObtenerGrupo. Dicho llamado se realiza únicamente con los vecinos de la frontera (en particular con vecinos que son fronteras), por lo que en cada llamado se crea un grupo con dicho vecino, se remueven de la lista de fronteras, se vuelve a llamar recursivamente a la función con los vecinos del vecino en cuestión y se concatenan los grupos resultantes. De esta forma, los grupos resultantes de las llamadas son conexos, y a su vez al concatenarlos son conexos entre si porque todos tienen un vecino en común.

C.3.4. Hay un grupo por cada componente conexo de las fronteras

Esta prueba se realiza por absurdo. Supongo que tengo una componente conexa en el conjunto de fronteras que el algoritmo retorna en dos grupos. Sin embargo, como el algoritmo itera sobre los 8 vecinos de cada frontera que va a incorporar a un grupo antes de finalizar, esto implicaría que si se tienen dos grupos, ningún elemento de un grupo es vecino del otro, lo cuál es absurdo porque sino no seria

conexo. Por lo tanto, se tiene que por cada componente conexa de fronteras, el algoritmo retorna un grupo.

Glosario

MINA grupo dentro del Instituto de Computación de la Universidad de la República enfocado en temáticas de robótica móvil y gestión de redes de computadoras. MINA proviene de: Network Management / Artificial Intelligence.

SDK *Software Development Kit* en inglés, es un conjunto de herramientas, bibliotecas, documentación y recursos que permite crear aplicaciones de software para plataformas específicas simplificando y agilizando el proceso. .

Esta página ha sido intencionalmente dejada en blanco.

Siglas

rcl ROS2 *Client Library*.

BT Behavior Trees.

DC Direct Current.

DDS Data Distribution Service.

ESP-IDF Espressif IoT Development Framework.

GPIO General Purpose Input/Output.

HAL Hardware Abstraction Layer.

ICRE Indoor Collaborative Robot Exploration.

InCo Instituto de Computación.

IoT Internet of Things.

LED Light Emitting Diode.

LiDAR Light Detection and Ranging.

PCB Printed Circuit Board.

PID Proporcional Integral Derivativo.

PWM Pulse Width Modulation.

QoS Quality of Service.

RAM Random Access Memory.

ROS Robot Operating System.

ROS2 Robot Operating System 2.

RTOS Real Time Operating System.

Siglas

SBC Single Board Computer.

SLAM Simultaneous Localization And Mapping.

UART Universal Asynchronous Receiver/Transmitter.

USB Universal Serial Bus.

Bibliografía

[1] Yulun Tian et al. «Kimera-Multi: Robust, Distributed, Dense Metric-Semantic SLAM for Multi-Robot Systems». En: *IEEE Transactions on Robotics* 38.4 (2022), págs. 2022-2038. DOI: 10.1109/TRO.2021.3137751.

[2] Steven Macenski et al. «Robot Operating System 2: Design, architecture, and uses in the wild». En: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[3] Josh Bongard. «Probabilistic Robotics. Sebastian Thrun, Wolfram Burgard, and Dieter Fox. (2005, MIT Press.) 647 pages». En: *Artificial Life* 14.2 (2008), págs. 227-229. DOI: 10.1162/artl.2008.14.2.227.

[4] MINA. *Robotito*. URL: https://www.robotito.cicea.ei.udelar.edu.uy. (accessed: 07.03.2023).

[5] MINA. *MINA Group FING*. URL: https://www.fing.edu.uy/inco/grupos/mina/. (accessed: 07.03.2023).

[6] Tully Foote y Melonee Wise. *Turtlebot*. URL: https://www.turtlebot.com. (accessed: 07.03.2023).

[7] S. Bernheim, A. Costa y A. De Luca. *Robot autónomo móvil terrestre con énfasis en SLAM y fusión sensorial*. URL: https://www.colibri.udelar.edu.uy/jspui/bitstream/20.500.12008/22512/1/BCD19a.pdf. (accessed: 07.03.2023).

[8] María Victoria Díaz et al. «Cooperative indoor exploration on affordable robots». En: *2022 Latin American Robotics Symposium (LARS), 2022 Brazilian Symposium on Robotics (SBR), and 2022 Workshop on Robotics in Education (WRE)*. 2022, págs. 235-240. DOI: 10.1109/LARS/SBR/WRE56824.2022.9996045.

[9] Tully Foote y Melonee Wise. *Gazebo*. URL: https://gazebosim.org/home. (accessed: 07.03.2023).

[10] Tully Foote y Melonee Wise. *Learn TurtleBot and ROS, Autonomous Navigation*. URL: https://learn.turtlebot.com/2015/02/03/9/. (accessed: 07.03.2023).

[11] Nived Chebrolu, David Marquez-Gamez y Philippe Martinet. *Collaborative Visual SLAM Framework for a Multi-Robot System*. URL: https://inria.hal.science/hal-02459361/document. (accessed: 07.03.2023).

[12] MINA Group. *Hardware - Robotito*. URL: https://gitlab.fing.edu.uy/robotito/hardware. (accessed: 07.03.2023).

[13] Pololu. *Rueda*. URL: https://www.robotshop.com/products/38mm-aluminum-omni-wheel. (accessed: 07.03.2023).

[14] Jean-Claude Latombe. «Robot Motion Planning». En: *Springer New York, NY* 1 (1991). DOI: 10.1007/978-1-4615-4022-9.

[15] Pololu. *Hubs*. URL: https://www.pololu.com/product/1996. (accessed: 07.03.2023).

[16] Pololu. *Micro Metal Gearmotor*. URL: https://www.pololu.com/product/2213/specs. (accessed: 07.03.2023).

[17] Pololu. *Anclajes de plastico*. URL: https://www.pololu.com/product/989. (accessed: 07.03.2023).

[18] Félix Monasterio-Huelin, Álvaro Gutiérrez y Blanca Larraga. *Modelado de un motorDC*. URL: http://www.robolabo.etsit.upm.es/asignaturas/seco/apuntes/modelado.pdf. (accessed: 07.03.2023).

[19] Pololu. *Encoder*. URL: https://www.pololu.com/product/3081. (accessed: 07.03.2023).

[20] Dejan. *How Rotary Encoder Works and How To Use It with Arduino*. URL: https://howtomechatronics.com/tutorials/arduino/rotary-encoder-works-use-arduino/. (accessed: 17.04.2023).

[21] adaFruit. *Driver Motor*. URL: https://www.adafruit.com/product/3297. (accessed: 07.03.2023).

[22] Wikipedia. *Puente H (electrónica)*. URL: https://es.wikipedia.org/wiki/Puente_H_(electr%C3%B3nica). (accessed: 07.03.2023).

[23] SparkFun. *ESP32 Thing*. URL: https://www.sparkfun.com/products/13907. (accessed: 07.03.2023).

[24] adaFruit. *ESP32 Datasheet*. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. (accessed: 07.03.2023).

[25] Wikipedia. *Pulse-width modulation*. URL: https://en.wikipedia.org/wiki/Pulse-width_modulation. (accessed: 07.03.2023).

[26] Robotis. *TurtleBot 3 Burger*. URL: https://www.robotis.us/turtlebot-3-burger-us/. (accessed: 07.03.2023).

[27] Robotis. *TurtleBot 3 Waffle*. URL: https://www.robotis.us/turtlebot-3-waffle-pi/. (accessed: 07.03.2023).

[28] Hard Kernel. *Odroid N2/N2+*. URL: https://wiki.odroid.com/odroid-n2/odroid-n2. (accessed: 28.02.2023).

[29] Curtis Manore, Pratheek Manjunath y Dominic Larkin. *Performance of Single Board Computers for Vision Processing*. URL: https://digitalcommons.usmalibrary.org/cgi/viewcontent.cgi?article=1472&context=usma_research_papers. (accessed: 07.03.2023).

[30] SlamTec. *RPLidar A1*. URL: https://www.slamtec.ai/home/rplidar_a1/. (accessed: 07.03.2023).

[31] Forza. *Forza UPS*. URL: https://www.forzaups.com/es/productos/interna/DC-140USB-esp/. (accessed: 07.03.2023).

[32] Xuan Bao Nguyen y Hoa Thi Truong. «Dynamics and Robust Adaptive Controller of Three Wheel Omnidirectional Mobile Robot». En: *2020 Applying New Technology in Green Buildings (ATiGB)*. 2021, págs. 110-114. DOI: 10.1109/ATiGB50996.2021.9423160.

[33] S. Boyd. *The Laplace transform*. URL: https://web.stanford.edu/~boyd/ee102/laplace.pdf. (accessed: 31.07.2023).

[34] Katsuhiko Ogata. *Modern Control Engineering*. 4th. USA: Prentice Hall PTR, 2001. ISBN: 0130609072.

[35] Universidad de Guanajuato. *Análisis de respuesta de sistemas de segundo orden*. URL: https://blogs.ugto.mx/rea/clase-digital-5-analisis-de-respuesta-de-sistemas-de-segundo-orden/. (accessed: 31.07.2023).

[36] Manuel Gräber. *Practical PID tuning guide*. URL: https://tlk-energy.de/blog-en/practical-pid-tuning-guide. (accessed: 07.03.2023).

[37] MATTHIEU BOUAT. *Understanding the effect of PWM when controlling a brushless dc motor*. URL: https://www.controleng.com/articles/understanding-the-effect-of-pwm-when-controlling-a-brushless-dc-motor/. (accessed: 07.03.2023).

[38] Tech Web. *Driving Brushed DC Motors Using PWM Output: Losses and Points to be Noted*. URL: https://techweb.rohm.com/product/motor/brushed-motor/brushed-motor-basic/346/. (accessed: 07.03.2023).

[39] SlamTec. *RPLidar A1 Datasheet*. URL: https://bucket-download.slamtec.com/d1e428e7efbdcd65a8ea111061794fb8d4ccd3a0/LD108_SLAMTEC_rplidar_datasheet_A1M8_v3.0_en.pdf. (accessed: 31.07.2023).

[40] Qoitech. *Otii Arc Pro*. URL: https://www.qoitech.com/otii-arc-pro/. (accessed: 31.07.2023).

[41] Richard Barry. *FreeRTOS*. URL: https://www.freertos.org. (accessed: 07.03.2023).

[42] Scott Campbell. *BASICS OF UART COMMUNICATION*. URL: https://www.circuitbasics.com/basics-uart-communication/. (accessed: 07.03.2023).

[43] Mimax y ngrennan. *What is ROS exactly? Middleware, Framework, Operating System?* URL: https://answers.ros.org/question/12230/what-is-ros-exactly-middleware-framework-operating-system/. (accessed: 08.08.2023).

[44] Addison Sears-Collinss. *ROS2 Architecture Overview*. URL: https://automaticaddison.com/ros-2-architecture-overview/. (accessed: 08.08.2023).

[45] William Woodall. *ROS on DDS*. URL: https://design.ros2.org/articles/ros_on_dds.html. (accessed: 01.08.2023).

[46] Open Robotics. *Different ROS 2 middleware vendors*. URL: https://docs.ros.org/en/humble/Concepts/Intermediate/About-Different-Middleware-Vendors.html. (accessed: 01.08.2023).

[47] Federico Andrade Martin Llofriu. «Estudio del estado del arte del SLAM e implementación de una plataforma flexible». En: *Proyecto de grado, Facultad de Ingeniería, Universidad de la República (Uruguay)*.

[48] Australian National University. *Airborne Inertial-SLAM*. URL: http://users.cecs.anu.edu.au/~Jonghyuk.Kim/PastResearch.html. (accessed: 08.08.2023).

[49] Aleksa Lukovic. *Grid Mapping in ROS*. URL: https://github.com/lukovicaleksa/grid-mapping-in-ROS. (accessed: 08.08.2023).

[50] Grupo de Investigación MINA. «Navegacion». En: Fundamentos de la Robótica Autónoma, Instituto de Computación, Facultad de Ingeniería, Universidad de la Republica (Uruguay), 2020.

[51] Felipe Espinosa et al. «Odometry and Laser Scanner Fusion Based on a Discrete Extended Kalman Filter for Robotic Platooning Guidance». En: *Sensors* 11.9 (ago. de 2011), págs. 8339-8357. ISSN: 1424-8220. DOI: 10.3390/s110908339. URL: http://dx.doi.org/10.3390/s110908339.

[52] Grupo de Investigación MINA. «Locomoción y cinemática». En: Fundamentos de la Robótica Autónoma, Instituto de Computación, Facultad de Ingeniería, Universidad de la Republica (Uruguay), 2017.

[53] Wolfram Burgard et al. «Robot Motion Planning». En: Universidad de Friburgo (Alemania), 2011.

[54] Open Robotics. *costmap 2d*. URL: http://wiki.ros.org/costmap_2d#Map_type_parameters. (accessed: 08.08.2023).

[55] Open Robotics. *move base*. URL: http://wiki.ros.org/move_base. (accessed: 08.08.2023).

[56] David Edwards. *The big AMR fleet computing debate: Distributed vs centralized*. URL: https://roboticsandautomationnews.com/2022/02/24/the-big-amr-fleet-computing-debate-distributed-vs-centralized/49537/. (accessed: 08.08.2023).

[57] Slamtec. *Slamtec LiDAR ROS2 Package*. URL: https://github.com/Slamtec/rplidar_ros/tree/ros2. (accessed: 31.07.2023).

[58] Slamtec. *Slamtec - Global Network*. URL: https://www.slamtec.ai/. (accessed: 31.07.2023).

[59] Steve Macenski. *Slam Toolbox ROS2*. URL: https://github.com/SteveMacenski/slam_toolbox. (accessed: 31.07.2023).

[60] Steve Macenski. *Steve Macenski*. URL: https://www.steve.macenski.com/. (accessed: 31.07.2023).

[61] Open Robotics. *Gmapping*. URL: http://wiki.ros.org/gmapping. (accessed: 08.08.2023).

[62] Open Robotics. *Slam Karto*. URL: http://wiki.ros.org/slam_karto. (accessed: 08.08.2023).

[63] Open Robotics. *Hector Slam*. URL: http://wiki.ros.org/hector_slam. (accessed: 08.08.2023).

[64] Aditya Kamath. *Comparing different SLAM methods*. URL: https://adityakamath.github.io/2021-09-05-comparing-slam-methods/. (accessed: 08.08.2023).

[65] Achala Athukorala. *Multirobot SLAM ros2*. URL: https://github.com/SteveMacenski/slam_toolbox/pull/592. (accessed: 08.08.2023).

[66] Open Navigation LLC. *Nav2 - ROS2 Navigation Stack*. URL: https://navigation.ros.org/. (accessed: 31.07.2023).

[67] B. Yamauchi. «A frontier-based approach for autonomous exploration». En: *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*. 1997, págs. 146-151. DOI: 10.1109/CIRA.1997.613851.

[68] Brian Yamauchi. «Frontier-Based Exploration Using Multiple Robots». En: *Proceedings of the Second International Conference on Autonomous Agents*. AGENTS '98. Minneapolis, Minnesota, USA: Association for Computing Machinery, 1998, págs. 47-53. ISBN: 0897919831. DOI: 10.1145/280765.280773. URL: https://doi.org/10.1145/280765.280773.

[69] scikit-learn developers. *K-Means*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html. (accessed: 05.09.2023).

[70] F. Pedregosa et al. «Scikit-learn: Machine Learning in Python». En: *Journal of Machine Learning Research* 12 (2011), págs. 2825-2830.

[71] Sergio Robaudo María Victoria Díaz. «Exploración colaborativa con Butiá». En: *Proyecto de grado, Facultad de Ingeniería, Universidad de la República (Uruguay)*. 2018, págs. 35-39.

[72] scikit-learn developers. *Affinity-Propagation*. URL: https://scikit-learn.org/stable/modules/clustering.html#affinity-propagation. (accessed: 08.08.2023).

[73] Open Robotics. *Gazebo Sensors*. URL: https://github.com/gazebosim/gz-sensors. (accessed: 01.04.2023).

[74] Guilherme Alan Ritter. *OpenBase: An omnidirectional mobile platform with a 3 omnidirectional wheels layout*. URL: https://github.com/GuiRitter/OpenBase. (accessed: 27.02.2023).

[75] *Linorobot2*. URL: https://github.com/linorobot/linorobot2. (accessed: 01.04.2023).

[76] Dhendra Marutho et al. «The Determination of Cluster Number at k-Mean Using Elbow Method and Purity Evaluation on Headline News». En: *2018 International Seminar on Application for Technology of Information and Communication*. 2018, págs. 533-538. DOI: 10.1109/ISEMANTIC.2018.8549751.

[77] ESPRESSIF. *FreeRTOS (ESP-IDF)*. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos_idf.html. (accessed: 07.03.2023).

Índice de tablas

Índice de figuras

Esta es la última página.
Compilado el martes 31 octubre, 2023.
http://iie.fing.edu.uy/