UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Rosificando un robot para uso Agropecuario: Ikus

Informe de Proyecto de Grado presentado por

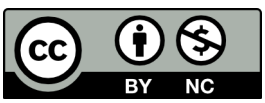
Christopher Alexis Friss de Kereki Manicera

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Gonzalo Tejera
Martin Llofriu

Montevideo, 28 de noviembre de 2025



Agradecimientos

Quiero agradecer a Gonzalo Tejera y Martín Llofriu por acompañarme y guiarme en este proyecto. A Mercedes Muñoz, por dejarme entrar al laboratorio una y otra vez y brindarme siempre su escucha. A mis padres, hermanos y a toda mi familia, por estar presentes en cada paso. A mis amigos y compañeros, por acompañarme en el camino. Y a Luna, Olivia y, sobre todo, a Agus.

Resumen

Este proyecto de grado presenta Ikus, un robot diseñado para el transporte de objetos en entornos agropecuarios. El objetivo principal fue desarrollar una plataforma robótica con una interfaz clara y documentada en ROS 2 (Robot Operating System), garantizando compatibilidad con versiones recientes y fomentando su reutilización en proyectos futuros.

El trabajo se estructuró en tres etapas: (1) implementación de un sistema de control diferencial en ROS 2 para la comunicación directa con los motores y la obtención de odometría basada en el movimiento de las ruedas, (2) integración de un sensor LiDAR, un módulo de odometría visual, mapeo y planificación de caminos, y (3) construcción de un entorno de simulación para su experimentación.

La validación se realizó tanto en un mundo simulado en Gazebo como en un entorno controlado de la Facultad de Ingeniería (Universidad de la República). Los resultados mostraron que la odometría basada en ruedas presentó limitaciones de precisión frente a la odometría visual apoyada en el LiDAR. La integración de planificación de caminos a Ikus fue realizada, y como parte del trabajo futuro se propone la experimentación exhaustiva de la herramienta, junto con mejoras en el hardware.

Palabras clave: Robótica, ROS 2, Mapeo, Simulación, Ikus, ros2_control

Índice general

Capítulo 1

Introducción

1.1. Motivación y Objetivos

La utilización de robots en el sector agropecuario permite aumentar la eficiencia y reducir riesgos y costos en diversas tareas. Sin embargo, los robots comerciales suelen ser costosos y presentan tecnologías desactualizadas y, en algunos casos, cerradas. Con el propósito de desarrollar un robot autónomo propio adaptado a entornos rurales, el Grupo de Investigación en Network Management / Artificial Intelligence (MINA) construyó Ikus utilizando componentes accesibles, intercambiables y tecnologías actualizadas y de código abierto.

Este proyecto tiene como objetivo adaptar Ikus para su integración con librerías de ROS 2, dado que ROS es el estándar en el desarrollo de software para robótica. En este contexto, surge el concepto de 'rosificación' de Ikus, lo que permitirá facilitar la integración con distintos sensores y actuadores en el futuro, reduciendo costos y complejidad.

Otro de los objetivos del proyecto es incorporar un entorno de simulación en Gazebo, con el fin de disponer de una plataforma virtual que permita probar y validar los distintos experimentos. La simulación proporciona un espacio seguro y flexible para experimentar con diferentes configuraciones de sensores, actuadores y escenarios propios del entorno agropecuario, reduciendo el riesgo de daños al robot y al entorno, y acelerando el proceso de desarrollo. De esta manera, se garantiza una validación temprana de las funcionalidades de Ikus, lo que facilita futuras extensiones y aplicaciones en campo.

1.2. Estructura del documento

El documento se organiza de la siguiente manera: en el Capítulo 2 se realiza una revisión del estado del arte, abordando proyectos relacionados y marcos de referencia relevantes. En el Capítulo 3 se presentan los fundamentos teóricos que sustentan el desarrollo del sistema, incluyendo descripciones de ROS, control de motores y simulación, entre otros.

En el Capítulo 4 se presenta la solución propuesta, detallando los requerimientos, la arquitectura de la solución, presentando las características de Ikus y los módulos de software construidos. El Capítulo 5 aborda las pruebas experimentales realizadas y los resultados obtenidos, mientras que el Capítulo 6 expone las conclusiones y las perspectivas de trabajo futuro.

Capítulo 2

Estado del arte

En este capítulo se presentan tres trabajos de investigación y desarrollo enfocados en la robótica móvil aplicada a entornos industriales y agrícolas. Cada uno se estructura en una sección que incluye un resumen del trabajo y las conclusiones obtenidas por los autores, destacando su relación con el desarrollo de Ikus. El primer trabajo aborda el diseño e integración de un manipulador móvil autónomo, el segundo ofrece una revisión del estado actual de la robótica agrícola a nivel global, y el tercero presenta la implementación del framework ros2_control en una plataforma de bajo costo.

2.1. Development of an Autonomous Mobile Manipulator for Industrial and Agricultural Environments

La tesis de maestría desarrollada por Giampà (2023) presenta el desarrollo de un robot móvil manipulador autónomo orientado a automatizar tareas en entornos industriales y agrícolas.

2.1.1. Resumen

El sistema combina una plataforma móvil con un brazo robótico, utilizando ROS 2 como marco de desarrollo, junto a Nav2 para navegación autónoma y MoveIt2 para planificación de movimientos. Un componente central es un actuador neumático blando, construido con piezas impresas en 3D, que permite manipular objetos delicados, enfrentando el reto de operar de forma autónoma en ambientes dinámicos y no estructurados mediante la fusión de sensores y algoritmos robustos.

Este robot, presentado en la figura 2.1, utiliza un robot AgileX Scout 2.0 como base móvil. Este robot móvil tiene una computadora Intel NUC 12 conectada mediante bus CAN, utilizada para recibir datos de los encoders del robot

y enviar datos de control de velocidad y dirección. El brazo robótico es un Igus Rebel 6-DoF **cobot**, refiriéndose por **cobot** a un robot colaborativo, diseñado para trabajar junto con humanos en un espacio compartido. Con el objetivo de realizar detección y reconocimiento de objetos, se utilizó una cámara estéreo Intel Realsense D435 RGB-D, colocada en una de las articulaciones del brazo robótico. Para crear mapas de sus entornos y localizar al robot, se utilizó un LiDAR 3D Ouster OS1-64, con un campo de visión de 360° y un rango de 120 metros. También está equipado con un router TP-Link Archer MR200 para establecer conexión remota desde una notebook via un punto de acceso Wi-Fi, permitiendo control y monitoreo remoto.

Figura 2.1: Vista lateral del robot, imagen tomada de 'Development of an Autonomous Mobile Manipulator for Industrial and Agricultural Environments' por Giampà

El proyecto logra integrar distintos componentes de hardware y software bajo una arquitectura modular que facilita futuras expansiones. Utiliza ros2_control para un manejo controlado del brazo robótico, mientras que la base móvil utiliza-

da ya presenta una interfaz integrable con las librerías de Nav2 para planificación de caminos.

Los resultados evidencian el potencial de este sistema para aumentar la productividad, eficiencia y seguridad en la industria y la agricultura, automatizando tareas repetitivas o peligrosas. Gracias a su diseño modular y al uso de tecnologías accesibles, el proyecto sienta las bases para futuras investigaciones y la creación de robots móviles aún más versátiles e inteligentes.

2.1.2. Conclusiones

El objetivo principal es demostrar la viabilidad de la navegación autónoma y la manipulación de objetos con diferentes objetivos finales, no para superar las capacidades humanas, sino para sentar una base sólida para futuras investigaciones y sistemas robóticos más complejos. Se plantea la necesidad de robots adaptativos e inteligentes que puedan ejecutar una variedad de tareas en escenarios reales, como la agricultura y la industria, utilizando hardware y software accesibles y de bajo costo.

El desarrollo de este sistema permitió comprobar la importancia clave de la percepción robusta y la localización precisa en entornos cambiantes, destacando el desafío de fusionar y procesar datos de sensores y de calibrar distintos componentes de hardware. Se remarca la necesidad de ambientes de simulación completos para probar y validar el sistema antes de desplegarlo en el mundo real. La principal lección obtenida es la eficacia de un enfoque iterativo en el desarrollo robótico, comenzando por tareas simples en simulación y aumentando progresivamente su complejidad para garantizar adaptabilidad y escalabilidad.

Aunque la integración con ros2_control está enfocada al control del brazo robótico, sirve como ejemplo de una correcta implementación de un sistema robótico controlado. Los experimentos realizados logran explicar cómo mejorar el mapeo, localización y planificación de caminos en entornos industriales y agrícolas. Para los experimentos realizados por Giampa, la localización realizada por 'SLAM Toolbox' tuvo mejor respuesta a aquella realizada por Nav2 mediante el algoritmo de 'Localización Adaptativa de Monte Carlo' (del inglés **A**daptive **M**onte **C**arlo **L**ocalization).

Por ejemplo, el algoritmo de 'SLAM Toolbox' tiene buena respuesta frente a problemas de localización en entornos con obstáculos dinámicos, y el algoritmo 'AMCL' tiene peor rendimiento en experimentos basados en rotaciones sin desplazamiento.

Este proyecto presenta un gran ejemplo a seguir, no solo por el alcance técnico similar al contexto de Ikus, el cual se integra ros2_control y librerías de mapeo, localización y planificación de caminos, sino también por el proceso iterativo incremental que sigue, trabajando en una parte del problema a la vez, para luego integrar con el panorama completo.

2.2. Advances in Agriculture Robotics: A State-of-the-Art Review and Challenges Ahead

El artículo elaborado por Oliveira, Moreira, y Silva (2021) realiza una investigación del estado del arte de los robots enfocados en el sector agropecuario.

2.2.1. Resumen

Los avances constantes en la robótica agrícola buscan responder a desafíos como el crecimiento poblacional, la urbanización acelerada, la alta competitividad en la producción de bienes de calidad, la preservación del medio ambiente y la escasez de mano de obra calificada. En este contexto, el presente artículo de revisión analiza las principales aplicaciones de los sistemas robóticos en la agricultura, abarcando tareas como la preparación del terreno, la siembra, el tratamiento de cultivos, la cosecha, la estimación de rendimiento y la fenotipificación.

Para cada robot analizado, se consideraron aspectos como su sistema de locomoción, aplicación final, presencia de sensores, brazo robótico, uso de algoritmos de visión por computadora, etapa de desarrollo y país de origen. A partir de estas características, se identificaron tendencias de investigación, errores comunes y factores que dificultan su comercialización. Además, se destaca la necesidad de profundizar en cuatro áreas clave para el avance de la agricultura inteligente: sistemas de locomoción, sensores, algoritmos de visión por computadora y tecnologías de comunicación. Los resultados indican que la inversión en sistemas robóticos agrícolas permite alcanzar objetivos tanto a corto plazo, como el monitoreo de cosechas, como a largo plazo, como la estimación del rendimiento.

2.2.2. Conclusiones

Para promover avances técnicos y científicos en el ámbito de la agricultura inteligente, es fundamental conocer los trabajos existentes, evaluando sus fortalezas, limitaciones y errores comunes, con el fin de identificar las verdaderas necesidades de mejora. Tras una revisión sistemática de 62 sistemas robóticos aplicados a tareas agrícolas como la preparación del suelo, la siembra, el tratamiento de cultivos, la cosecha, la estimación de rendimiento y la fenotipificación, se obtuvieron diversos hallazgos, de los cuales resaltamos aquellos que se relacionan con Ikus: el 6 % de los robots agrícolas emplean tracción en solo dos ruedas, el 64,52 % no cuenta con brazo robótico, el 22,06 % se utiliza en labores de deshierbe y el 8,82 % en tareas generales de agricultura, el 16,53 % incorpora LiDAR, el 35,48 % no menciona o no utiliza algoritmos de visión por computadora, el 80,65 % aún se encuentra en fase de investigación, el 16,13 % proviene de América, donde el 13,64 % ha sido desarrollado en los Estados Unidos.

Entre las características más destacadas se encuentran la escasa adopción de soluciones comerciales estandarizadas, el uso limitado de enfoques de robótica en enjambre o paralelismo, la baja utilización de algoritmos de visión por

computadora, así como de plataformas versátiles adaptables a distintos cultivos.

Para mejorar estos sistemas, se proponen cuatro áreas clave para futuras investigaciones: sistemas de locomoción, sensores, algoritmos de visión artificial y agricultura inteligente basada en IoT. Este estudio analizó 62 sistemas robóticos agrícolas y reveló un incremento del 22,98 % en la tasa promedio de éxito en la cosecha y una reducción del 42,78 % en el tiempo promedio del ciclo de recolección entre 2014 y 2021. Con avances en las áreas mencionadas, se espera que la eficiencia y robustez de estos sistemas continúe mejorando, consolidando su papel como herramientas clave en la transformación del entorno natural a través de la robótica móvil.

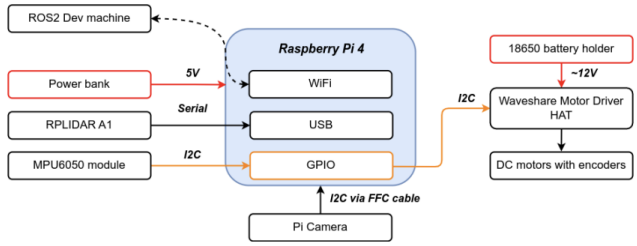
El relevamiento presentado sobre los 62 sistemas robóticos no hace referencia a las tecnologías utilizadas ni a si son de código abierto o cerrado.

2.3. An introduction to the ros2_control framework using a low cost differential drive robot

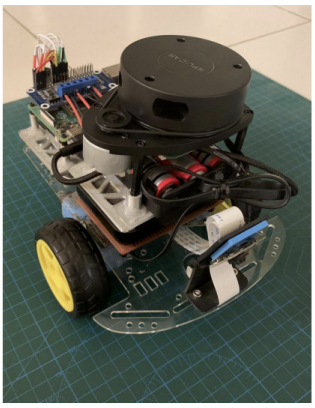
El artículo elaborado por Amadi, Mbanisi, y Smit (2024) presenta la construcción de un robot diferencial económico utilizando ros2_control.

2.3.1. Resumen

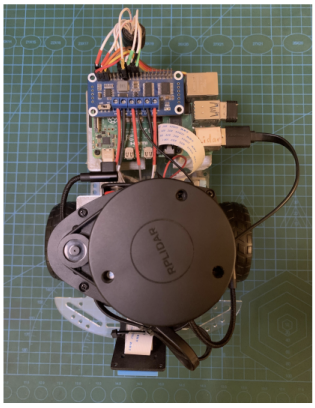
Con el creciente uso de la robótica en diversas aplicaciones a nivel mundial, se vuelve crucial el acceso a hardware para facilitar su desarrollo, especialmente en contextos educativos y de investigación. Esto es aún más relevante en regiones donde los robots comerciales son costosos y difíciles de adquirir, lo que puede limitar el aprendizaje y la innovación en robótica.



(a) *Lidarbot hardware architecture (Adapted from SMARTmBOT [5])*



(b) *Side view*



(c) *Top view*

Figura 2.2: Diagrama de Hardware de Lidarbot (a), fotografías de Lidarbot (b y c), imagenes tomadas de 'An introduction to the ros2_control framework using a low cost differential drive robot' por Amadi y cols.

En respuesta a esta necesidad, se presenta Lidarbot (observado en la figura 2.2), un robot diferencial de bajo costo y código abierto diseñado como una

plataforma de inicio para el aprendizaje de temas clave como ROS (Robot Operating System), SLAM, navegación autónoma y fusión de sensores. Con un costo aproximado de 250 dólares, el Lidarbot cuenta con los sensores y componentes necesarios para experimentar con estas tecnologías.

Este robot también se utiliza como ejemplo para implementar el framework de control agnóstico al hardware ros2_control, junto con la pila de navegación Nav2 para planificación de caminos. Su diseño y código fuente están disponibles públicamente en GitHub, lo que facilita su adopción y adaptación por parte de estudiantes, docentes e investigadores interesados en robótica.

2.3.2. Conclusiones

Lidarbot fue el robot presentado en el artículo como una alternativa económica ante plataformas robóticas comerciales para adentrarse en robótica con ROS 2, sensores y actuadores, teoría de control, SLAM y planificación de caminos. El robot sirvió como ejemplo práctico para ilustrar el uso del framework ros2_control, destacando cómo puede aplicarse junto con la pila de navegación Nav2 para lograr una navegación autónoma eficiente.

El uso de ros2_control permite centrar los esfuerzos en la configuración y ajuste de los parámetros del controlador para adaptarse al diseño específico del robot y en el desarrollo de aplicaciones. Esto es posible gracias a que el framework gestiona aspectos complejos como el sistema de control, el ciclo de vida del hardware, la comunicación y el acceso al mismo. La experiencia adquirida con esta plataforma puede trasladarse a sistemas más avanzados.

Se puede decir que Lidarbot es fruto de una 'rosificación', dado que el sistema basa la comunicación entre actuadores y sensores en paquetes y nodos de ROS 2. Estos nodos se ejecutan en la unidad de cómputo, una Raspberry Pi 4 con Ubuntu server 22.04, con ROS 2 Humble, permitiendo el control remoto vía Wi-Fi. Se decide incorporar una unidad de medición inercial (IMU) MPU6050 con el objetivo de complementar la odometría obtenida por ros2_control mediante el procesamiento de los encoders de las ruedas del Lidarbot.

En términos de mejoras a futuro, se plantean algunas actualizaciones tecnológicas. Reemplazar el LiDAR 'RPLIDAR A1' por 'RPLIDAR C1', el cual utiliza el 'tiempo de vuelo' (TOF) para la medición. Para facilitar el control de las ruedas con motores de corriente continua: substituir el 'Waveshare Motor Driver HAT' por una Raspberry Pi Pico y un módulo de controladora de motor 'TB6612FNG', con intenciones de implementar micro-ROS en la Raspberry Pi Pico y facilitar la integración con la controladora.

Capítulo 3

Marco Teórico

3.1. Introducción

En esta sección se busca definir y repasar conceptos que serán nombrados a lo largo del informe. El capítulo comienza con una breve introducción a la robótica, ahondando en los vehículos autónomos y robots diferenciales. Luego se presentan conceptos relacionados al hardware disponible para sistemas robóticos, como sensores LiDAR y motores BLDC. Por último, se presentan distintos conceptos y tecnologías relacionados al software utilizado en el área, introduciendo ROS, ros2_control, distintas herramientas que estos utilizan y conceptos más generales como simulación, mapeo y localización, y planificación de caminos, entre otros.

3.2. Robótica

Los robots son piezas de maquinaria programables para cumplir distintas tareas. Son utilizados en cientos de tipos de industrias, desde entretenimiento, como animatrónicos, hasta bélicos, como drones de guerra.

Los sensores son dispositivos que permiten obtener información del entorno mediante la medición de diferentes magnitudes físicas. Los actuadores, en cambio, son aquellos que posibilitan actuar sobre el entorno, pudiendo en algunos casos modificar las condiciones que luego serán percibidas por los sensores.

Un sistema robótico puede ser programado para utilizar la información sensada del medio, procesarla y generar directivas en base a las restricciones o modificaciones propuestas por el sistema de control, que son luego interpretadas por los actuadores. En la figura 3.1 se presenta un esquema de un sistema robótico simple.

3.2.1. Vehículos autónomos

> "An intelligent robot is a machine able to extract information from its environment and use knowledge about its world to move safely

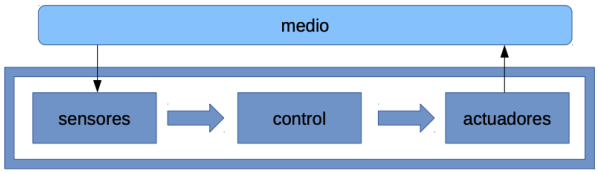


Figura 3.1: Interacción de sistema simple, imagen tomada de Fundamentos de Robótica Autónoma - Unidad 1.2

> in a meaningful and purposive manner."
> - Arkin (1998), p.3

Un vehículo robótico autónomo cumple con las características de robot inteligente de Arkin: 'una máquina capaz de extraer información del ambiente y utilizar conocimiento sobre el entorno para moverse con sentido y propósito'.

Los vehículos autónomos normalmente se componen de cámaras o sensores de distancia y ruedas o algún tipo de pierna mecánica con varias articulaciones. Son capaces de sensar el entorno y estimar sus posiciones en el espacio y distancias respecto a obstáculos que los rodean. A su vez, su unidad de cómputo y su programación indican una serie de velocidades, potencias o ángulos deseados a los motores en ruedas o pies, modificando la posición con sentido, intentando cumplir ese objetivo que Arkin nombra 'propósito'. Existen robots que son denominados de 'propósito general'; son aquellos a los que no se les asigna un propósito específico y tienen versatilidad para adaptarse (o ser adaptados) según sean necesarios.

El robot Jackal (Figura 3.2), un Vehículo Terrestre No Tripulado (UGV) fabricado por Clearpath Robotics, cumple con las características de 'propósito general', ya que es posible comprar o desarrollar distintos subsistemas de integración para diferentes escenarios.

3.2.2. Robot diferencial

Un robot es considerado un robot 'diferencial' si tiene dos ruedas motorizadas una opuesta de la otra, junto con alguna rueda giratoria de apoyo, como el que se observa en la figura 3.3. Para lograr mover el robot hacia adelante, ambas ruedas deben moverse a la misma velocidad en direcciones opuestas. Para girar a la derecha, la rueda izquierda debe moverse más rápido que la derecha, y viceversa para girar a la izquierda. También es posible realizar un giro en el lugar moviendo las ruedas a la misma velocidad en la misma dirección. Las ruedas giratorias de apoyo agregan estabilidad al sistema.

Figura 3.2: Robot Jackal de Clearpath Robotics, imagen tomada de Clearpath Robotics

3.3. Hardware

3.3.1. LiDAR

LiDAR (Light Detection and Ranging, o Detección y Medición por Luz) es un método de sensado que utiliza luz en forma de láser para medir distancias. Las utilidades de un sensor de este tipo en robótica varían desde el mapeo en dos o tres dimensiones de un área, hasta el reconocimiento y seguimiento de objetos.

Como ejemplo, el siguiente sistema de sensor LiDAR emite FMCW láseres (Frequency Modulated Continuous Wave laser, o láser de onda continua y frecuencia modulada) en distintas direcciones. Estos pulsos emitidos son proyectados sobre obstáculos en el entorno, y su luz es detectada por otro módulo del sensor. Basándose en el ToF (Time of Flight o tiempo de vuelo) asociado a cada frecuencia emitida, la unidad de procesamiento del sensor genera un grupo de datos conocido como nube de puntos, donde cada uno de esos puntos contiene la distancia sensada hacia el obstáculo más cercano en la dirección en la que fue emitido cada láser. Como se observa en la figura 3.4, utilizando herramientas de visualización, es posible observar una representación de esa nube de puntos sensada de la realidad.

En la figura 3.5 se observa la solución más popular actualmente para LiDAR en automóviles: el sistema de giro mecánico, el cual dirige los láseres mediante un espejo o prisma, controlado por un motor para realizar el giro, para generar un amplio campo de visión.

Figura 3.3: Robot diferencial, imagen tomada de 'Kinematics, localization and control of differential drive mobile robot' por Malu y cols. (2014)

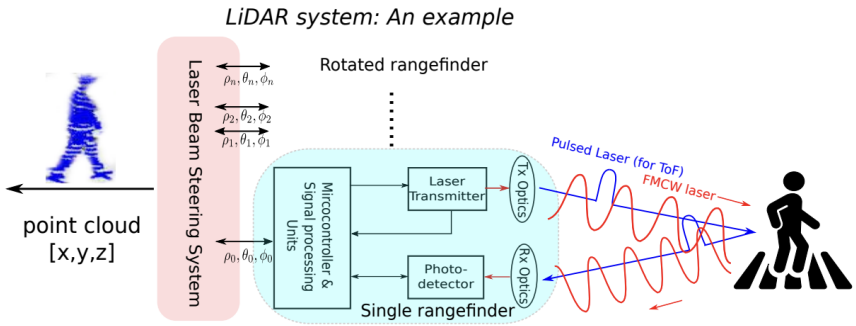


Figura 3.4: Ejemplo de sistema de LiDAR, tomada de 'Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems' por Li y Ibanez-Guzman (2020)

3.3.2. Encoder

Un encoder es un dispositivo electrónico que se encarga de medir posiciones, desplazamientos o velocidades de componentes giratorios. Existen distintos tipos de encoders que realizan sus mediciones en base a diferentes fenómenos físicos.

Los encoders pueden clasificarse según su principio de funcionamiento en:

- **Encoders ópticos**: Utilizan un disco perforado y un sistema de LED y fotodetector para detectar el movimiento mediante interrupciones de luz.
- **Encoders magnéticos**: Emplean sensores de efecto Hall o magnetorresistivos para detectar cambios en campos magnéticos generados por imanes en movimiento.
- **Encoders capacitivos**: Detectan variaciones de capacitancia causadas

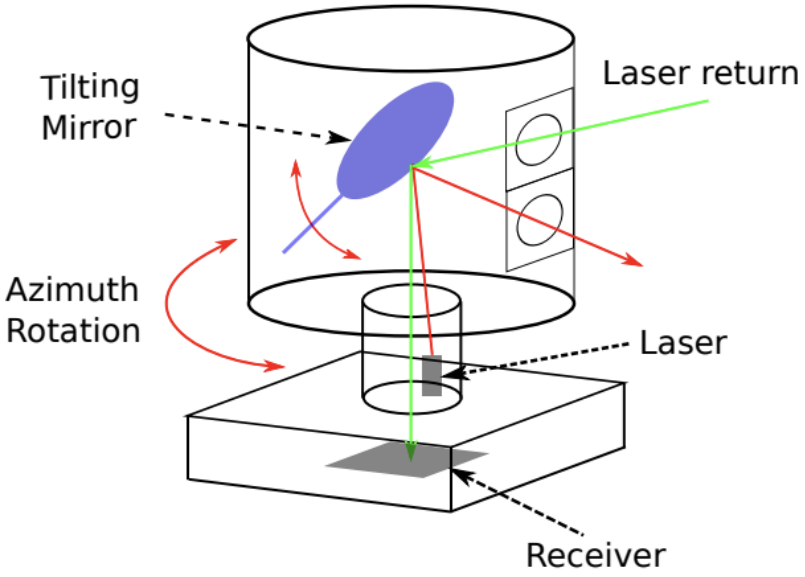


Figura 3.5: Ejemplo de sistema de LiDAR con giro mecánico, tomada de Li y Ibanez-Guzman (2020)

por el movimiento de un rotor.

Encoders Magnéticos - Efecto Hall

El efecto Hall ocurre cuando un semiconductor, al ser expuesto a un campo magnético, genera una diferencia de potencial en sus extremos. Este fenómeno se debe al desplazamiento interno de cargas eléctricas provocado por las líneas de fuerza del campo magnético, resultando en una diferencia de tensión proporcional a la intensidad del flujo magnético.

En los encoders magnéticos basados en efecto Hall, un sensor detecta el paso de los polos magnéticos de un imán montado en el eje del motor. Cada cambio de polaridad genera un pulso eléctrico que permite determinar la posición angular y la velocidad de rotación del eje.

3.3.3. Motores BLDC

Los motores de corriente continua sin escobillas o BLDCs (del inglés **B**rush**L**ess **DC**) son motores que utilizan imanes y bobinados que generan campos magnéticos para generar movimiento en base a un eje.

Usualmente, los BLDC suelen tener imanes fijos en el rotor y bobinados en

el estator. Uno de los principios fundamentales de estos motores es que el campo magnético generado por las bobinas del estator se sincroniza con el producido por los imanes del rotor, permitiendo un movimiento eficiente y controlado.

En los motores BLDC, los sensores de efecto Hall se utilizan para detectar la posición del rotor en tiempo real, permitiendo un control preciso de la conmutación de las bobinas del estator. Los motores requieren controladoras que logren manejar correctamente la intensidad de corriente que pasa por esas bobinas en cada momento, y que puedan interpretar la información medida por el sensor.

3.3.4. ESC

Una controladora electrónica de velocidad o ESC (del inglés **E**lectronic **S**peed **C**ontroller) es un dispositivo electrónico utilizado para controlar la velocidad y dirección de motores eléctricos, especialmente en aplicaciones de vehículos eléctricos, drones y robots. Los ESCs se encargan de regular el flujo de energía entre la fuente de alimentación y el motor, manteniendo un manejo preciso de aceleración y desaceleración. Para los BLDC, los ESCs son capaces de gestionar la conmutación de las fases del motor, y algunos permiten la lectura de encoders presentes en el motor, como por ejemplo de encoders basados en sensores del efecto Hall.

Dentro del universo de los ESCs, VESC (por **V**edder ESC) representa una solución avanzada y de código abierto creada por Benjamin Vedder. Vedder publicó todos sus diagramas de circuitos y software de forma gratuita, junto con varios posteos en su blog, con el objetivo de que hobbyistas puedan construir sus propios vehículos y puedan controlarlos fácilmente hasta con sus celulares.

El proyecto de Vedder incluye una ESC de motores BLDC junto a su firmware, y un programa de computadora que permite comunicarse con la ESC y el motor al que está conectada: VESC Tool. Mediante la interfaz gráfica de VESC Tool es posible controlar la velocidad, torque, potencia e intensidades máximas y mínimas del motor. También es posible acceder a la información obtenida del encoder, interpretada como cantidad de revoluciones realizadas

En su Foro, Vedder presenta los diagramas para la construcción del circuito de VESC, junto con puntos de venta donde es posible comprarla armada.

Utilizando la interfaz de la controladora proporcionada por Vedder, obtendremos control de los motores y lectura sobre la cantidad de revoluciones que realizan.

3.4. Software

3.4.1. ROS y ROS 2

ROS, o Robotic Operating System, es un conjunto de librerías y herramientas open source, utilizadas para construir software para robots. Utilizar librerías estándar ayuda a abstraer el contexto de bajo nivel de los actuadores y sensores utilizados, permitiendo ejecutar programas en C y Python sobre los robots. ROS

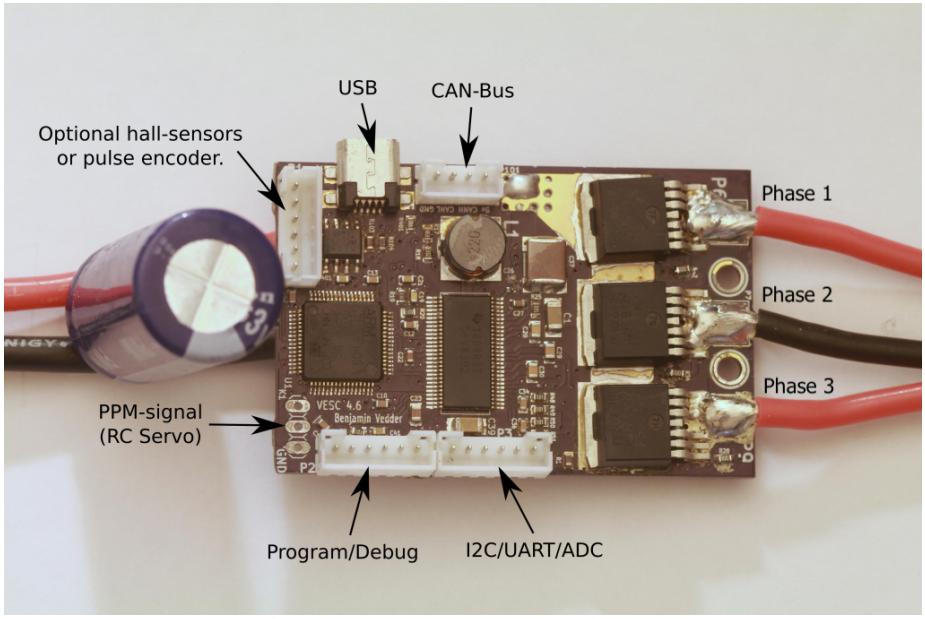


Figura 3.6: Circuito de Controladora VESC, indicando conexiones de: el sensor Hall arriba a la izquierda, USB para unidad de cómputo junto al del sensor hall, las 3 fases del motor a la derecha. A la izquierda rojo indica positivo y negro negativo, los cuales se conectan a la fuente de poder. Imagen tomada originalmente de Benjamin Vedder

maneja distintas distribuciones, las cuales buscan facilitar la compatibilidad con las distintas versiones de Ubuntu.

ROS también existe en su segunda versión, ROS 2, la cual tiene sus propias distribuciones orientadas a versiones más recientes de Ubuntu y otros sistemas operativos. ROS 2 aprende de los aciertos y errores cometidos en la construcción de ROS, y define un camino más estable para el desarrollo de la robótica en los próximos años.

Funcionamiento de ROS

A continuación se listarán algunos conceptos básicos de ROS junto a una especificación para su uso en ROS 1 y ROS 2.

Nodos

ROS ejecuta procesos denominados 'nodos' a los cuales se les adjudican tareas específicas. Los nodos son parte de un 'paquete', los cuales pueden ser propios o creados por la comunidad de ROS.

```
1
2
```

```
### ROS - Noetic
### Ejecutar nodo
$ rosrun [nombre_paquete] [nombre_nodo]

# Listar nodos en ejecución
$ rosnode list

######

### ROS 2 - Humble
### Ejecutar nodo
$ ros2 run [nombre_paquete] [nombre_nodo]

### Listar nodos en ejecución
$ ros2 node list
```

Launch

Para ejecutar distintos nodos en simultáneo o con relación entre sí, se utilizan archivos de tipo 'launch', que especifican los distintos nodos y algunos parámetros base.

```
### ROS - Noetic
### Ejecutar archivo de lanzamiento
$ roslaunch [nombre_paquete] [archivo.launch]

######

### ROS 2 - Humble
### Ejecutar archivo de lanzamiento
$ ros2 launch [nombre_paquete] [archivo.launch.py]
```

Tópicos

Los nodos pueden publicar datos a través de canales de comunicación llamados 'tópicos', o suscribirse a 'tópicos' expuestos por otros nodos para procesar dichos datos. En la figura 3.7 se observa una representación de la interacción entre nodos a través de tópicos.

```
### ROS - Noetic
### Listar tópicos
$ rostopic list

### Publicar en un tópico
$ rostopic pub /[nombre_topico] [tipo_mensaje] "{data: 'mensaje'}"

### Suscribirse a un tópico
$ rostopic echo /[nombre_topico]

######

### ROS 2 - Humble
### Listar tópicos
$ ros2 topic list

```

```
### Publicar en un tópico
$ ros2 topic pub /[nombre_topico] [tipo_mensaje] "{data: 'mensaje
    '}"

### Suscribirse a un tópico
$ ros2 topic echo /[nombre_topico]
```

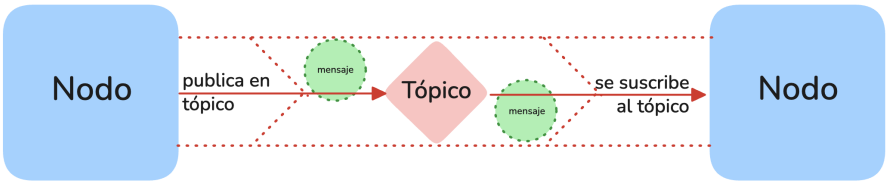


Figura 3.7: Representación de interacción entre nodos a través de tópicos

Los mensajes en ROS están organizados en paquetes y se definen en archivos .msg. Los paquetes más comunes con tipos de mensajes incluyen:

std_msgs: Contiene mensajes básicos como String, Int32, Float64, etc.

geometry_msgs: Contiene estructuras de datos geométricas como 'Twist', que expresa velocidades de un objeto separandolas entre partes lineales y angulares, o Pose, que representa la posición de un objeto en el espacio en base a su posición y orientación, entre otros.

sensor_msgs: Contiene mensajes relacionados con sensores como Image, obtenida de cámaras, donde el mensaje contiene la imagen sin procesar junto a información complementaria (tamaño de imagen i.e.), o LaserScan, obtenida de LiDARs por ejemplo, reprentando en metros la distancia del sensor a los objetos y paredes a su alrededor, entre otros.

nav_msgs: Contiene mensajes para navegación como Odometry, que representa la pose y velocidad estimada en el espacio, utilizando mensajes similares a Pose y Twist, u OccupancyGrid, que representa un mapa en una grilla 2D, donde cada celda contiene la probabilidad de que el espacio que representan esas coordenadas esté ocupado, entre otros.

Una estructura interesante es la de geometry_msgs/Twist, la cual es normalmente utilizada para enviar comandos de velocidad a robots a través del tópico 'cmd_vel'. Su agrupación en velocidad lineal indica velocidad en m/s en cada eje x,y,z, y para velocidad angular, con velocidad en rad/s en cada eje x,y,z. En el caso de los ejes x,y,z en velocidad angular, se hace referencia a los ángulos de navegación 'roll, pitch, yaw' o 'alabeo, cabeceo, dirección'.

A continuación se ejemplifica cómo se enviaría un comando para un movimiento hacia adelante con un giro antihorario en simultáneo:

```
geometry_msgs/Twist
-------------------
Vector3 linear
  float64 x
  float64 y
  float64 z
Vector3 angular
  float64 x
  float64 y
  float64 z

### ROS - Noetic
### Publicar mensaje de tipo Twist en el tópico cmd_vel
$ rostopic pub /cmd_vel geometry_msgs/Twist "
    {
        linear: {x: 0.5, y: 0.0, z: 0.0},
        angular: {x: 0.0, y: 0.0, z: 1.0}
    }
"

######

### ROS 2 - Humble
### Publicar mensaje de tipo Twist en el tópico cmd_vel
ros2 topic pub /cmd_vel geometry_msgs/Twist "
    {
        linear: {x: 0.5, y: 0.0, z: 0.0},
        angular: {x: 0.0, y: 0.0, z: 1.0}
    }
"
```

REP

Las propuestas de mejoras de ROS o REP (del inglés ROS Enhancement Proposals) son documentos de diseño que proveen información a la comunidad de ROS, describiendo normalmente nuevas funcionalidades, procesos, entornos y convenciones.

TF2

TF2 es la librería de ROS 2 encargada de gestionar transformaciones entre diferentes sistemas de coordenadas, permitiendo transformar puntos, vectores y posiciones entre distintos marcos de referencia, incluso en diferentes momentos del tiempo.

Este sistema organiza los marcos de referencia en una estructura de árbol, en la que cada nodo representa un marco, y las relaciones entre ellos (transformaciones) son almacenadas en un búfer temporal. En las figuras 3.8 y 3.9 se observan distintas representaciones de transformaciones entre marcos, la primera se enfoca en los tiempos en los que se publicaron, mientras que la segunda en las posiciones entre los marcos.

El REP 105 especifica convenciones de nomenclatura y semántica para marcos de plataformas móviles de ROS: base_link, odom y map.

base_link

El marco base_link está asociado a la base del robot móvil. Se utiliza como el punto de referencia para el robot

odom

El marco odom es un marco fijo globalmente, funciona como una fuente de 'verdad' absoluta frente al movimiento realizado. La pose de un robot en el marco odom es continua, es decir que la pose de la plataforma móvil se actualiza de manera uniforme sin realizar saltos discretos. Este marco se computa en base a una fuente de odometría, como basada en ruedas, en visión, o utilizando una Unidad de medición inercial (IMU). Odom suele acumular un error denominado 'deriva' o 'drift' en la pose de la plataforma móvil, por lo que suele ser preciso como referencia a corto plazo, pero a medida que avanza el tiempo su precisión se disminuye.

map

El marco map también es un marco fijo, con su eje Z apuntando hacia arriba. Este marco no es continuo, por lo que la pose de la plataforma móvil puede realizar saltos discretos en cualquier momento. En configuraciones típicas, un componente de localización está constantemente computando la pose del robot en base a los sensores visuales, realizando 'saltos' al actualizarla, eliminando así errores de drift.

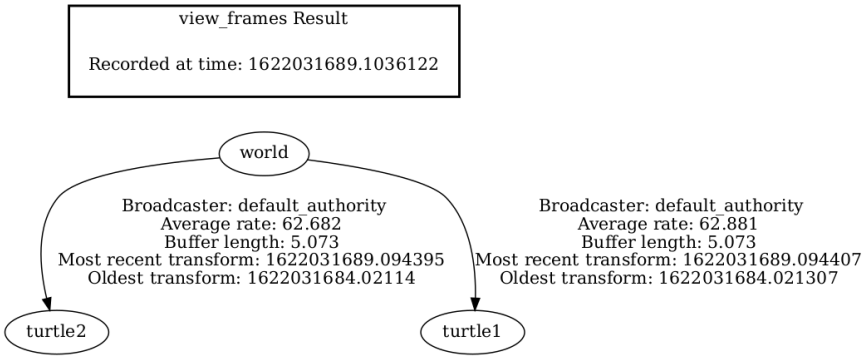


Figura 3.8: Ejemplo de árbol de transformaciones en ROS 2 con TF2, tomada de la documentación de TF2.

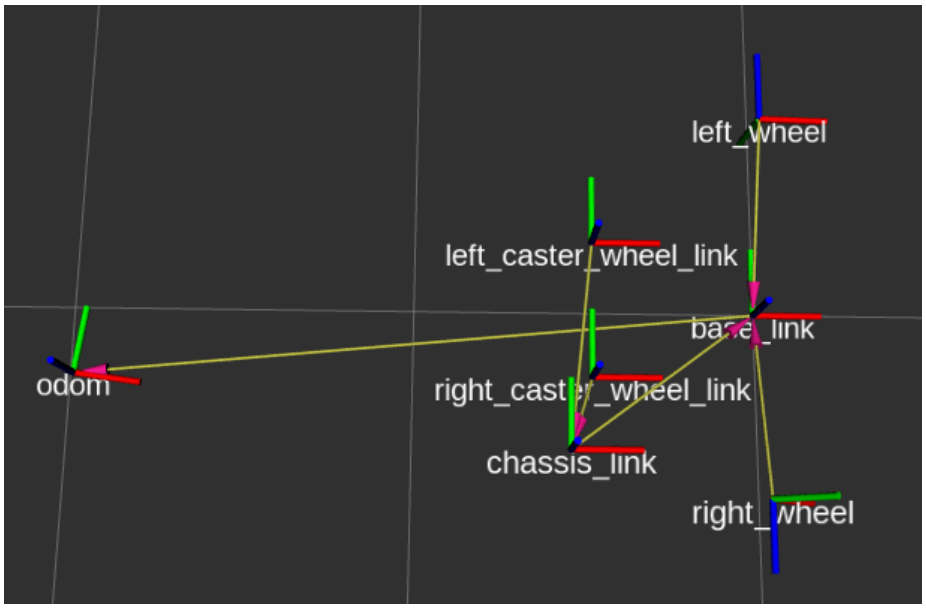


Figura 3.9: Representación visual en Rviz2 de transformaciones entre marcos.

Publicación de transformaciones

Para publicar transformaciones, se utiliza el tipo de mensaje 'geometry_msgs/msg/TransformStamped'. Cada mensaje define una transformación desde un marco padre a un marco hijo.

Por ejemplo:

```
geometry_msgs::msg::TransformStamped transform;
transform.header.stamp = rclcpp::Time::now();
transform.header.frame_id = "base_link";
transform.child_frame_id = "laser";
transform.transform.translation.x = 0.2;
transform.transform.translation.y = 0.0;
transform.transform.translation.z = 0.1;
// Se puede utilizar tf2::Quaternion para convertir de RPY (roll,
    pitch, yaw) a cuaternión
transform.transform.rotation = tf2::toMsg(quaternion);
```

Este fragmento indica que el marco 'laser' está ubicado a 20 cm hacia adelante y 10 cm hacia arriba del marco 'base_link'.

tf2_tools

Utilizando la herramienta 'tf2_tools' es posible obtener una visualización de las transformadas y sus relaciones como se observa en la figura 3.8. Este diagrama se genera y exporta en un archivo en formato .pdf una vez ejecutado este comando:

```
ros2 run tf2_tools view_frames
```

Teleop Twist Keyboard

El paquete teleop_twist_keyboard se utiliza para enviar mensajes de tipo 'geometry_msgs/Twist' en un tópico, normalmente 'cmd_vel', a partir de las teclas presionadas.

El paquete puede utilizarse en una terminal de la siguiente manera:

```

$ ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args
    --remap
      cmd_vel:=/ikus_base_controller/cmd_vel_unstamped

### --remap también puede utilizarse como -r
```

Twist MUX

El paquete twist_mux se utiliza para multiplexar distintos comandos de velocidad (en tópicos que aceptan mensajes Twist) permitiendo priorizarlos o deshabilitarlos.

Este paquete se ejecuta de la siguiente manera:

```
ros2 run twist_mux twist_mux.launch --params-file twist_mux.yaml
```

El archivo de configuración 'twist_mux.yaml' puede contener los siguientes parámetros:

name: Nombre de la configuración, utilizado para depuración.

topic: Nombre del tópico de ROS, el nodo twist_mux se suscribirá al tópico. Debe ser de tipo 'geometry_msgs/Twist'

timeout: Tiempo de vida del mensaje. En caso de que no llegue otro mensaje antes de que pase este tiempo, se selecciona otro tópico.

priority: Prioridad del tópico desde 0 a 255. Cuanto más alto, más prioridad frente a otros tópicos.

Ejemplo de archivo de configuración:

```
twist_mux:
  ros__parameters:
    publish_rate: 10.0
    topics:
      navigation:
        topic   : cmd_vel
        timeout : 0.5
        priority: 10
      keyboard:
        topic   : cmd_vel_key
        timeout : 0.5
        priority: 100
      joystick:
        topic   : cmd_vel_joy
        timeout : 0.5
```

```
16          priority: 110
```

En el ejemplo, el tópico asociado al nombre 'joystick' ('cmd_vel_joy') tiene más prioridad que el asociado al nombre 'keyboard' ('cmd_vel_key'), y ambos tienen mucha más prioridad que el asociado al nombre navigation ('cmd_vel')

Rviz2

Rviz2 es una herramienta de visualización 3D para ROS.

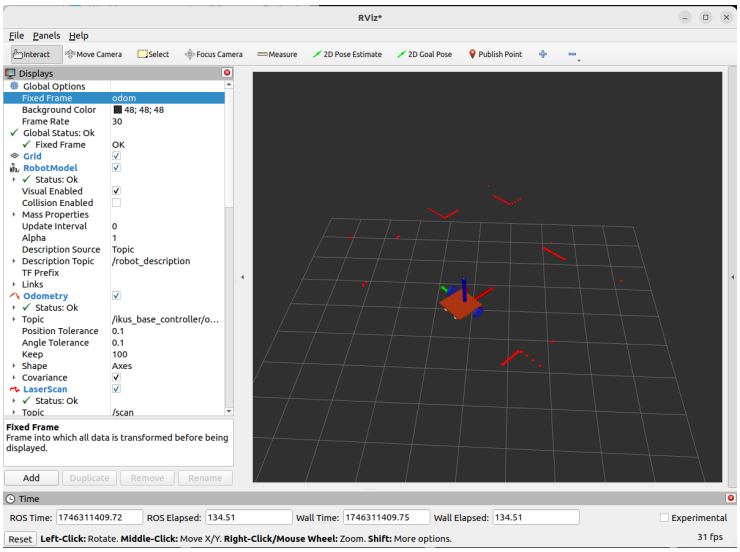


Figura 3.10: Representación en la interfaz gráfica de Rviz

Esta herramienta permite visualizar información obtenida a través de tópicos de distintas formas. Es posible observar transformaciones realizadas por un robot, actualizando la posición y forma en tiempo real a través del tópic 'tf2', como muestra la figura 3.10. También es posible visualizar representaciones de información obtenida de sensores. Con un LiDAR por ejemplo, que puede exponerse en tópicos de tipo 'sensor_msgs/LaserScan', puede verse representada con un punto por cada distancia sensada respecto al origen, como muestra la figura 3.10 con puntos rojos.

Rviz2 también cuenta con distintos atajos para simplificar el uso de herramientas de paquetes de Ros 2, evitando utilizar la línea de comandos. Existe un atajo para guardar mapas obtenidos a partir de tópicos, para luego ser cargados y utilizados en localización. Otro, como se observa en la figura 3.11 permite indicar una posición objetivo en el espacio 2D, para ser aplicados por el paquete de planificación de caminos.

Es posible guardar configuraciones de Rviz2 para simplificar el uso las próximas veces. Para ejecutar Rviz2, simplemente:

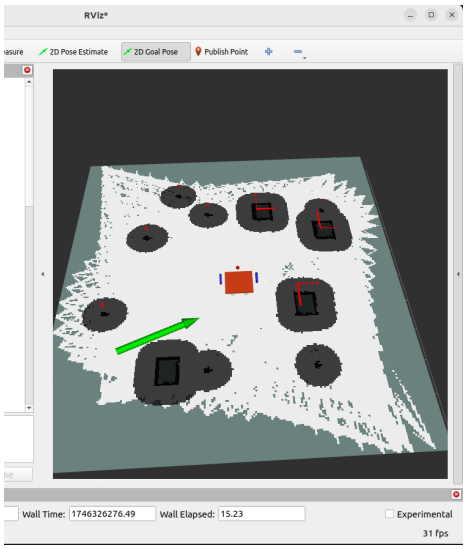


Figura 3.11: Atajo para indicar posición objetivo en Rviz2

```
$ rviz2
```

ros2_control

ros2_control es un framework para control en tiempo real de robots usando ROS 2. El objetivo es simplificar la integración de piezas de hardware y utilizar el trabajo ya construido sobre el control de robots. Se basó en reconstruir los paquetes de ros_control, la versión utilizada para el mismo propósito en ROS 1.

El paquete de ros2_control actúa como intermediario entre el sistema de software y las controladoras físicas de los dispositivos. Esta herramienta cuenta con distintos módulos que interactúan entre sí, abstrayendo la vinculación entre el software y el hardware.

Arquitectura

ros2_control presenta distintos módulos y conceptos: Controller Manager (administrador de controladoras), Resource Manager (administrador de recursos), Controllers (controladoras), Hardware Components (represetanción abstracta del componente de hardware), State Interface (Interfaz de estado), Command Interface (Interfaz de Comandos) y Hardware Description (Descripción de hardware).

En la figura 3.12 se muestran los componentes que serán descritos en esta sección.

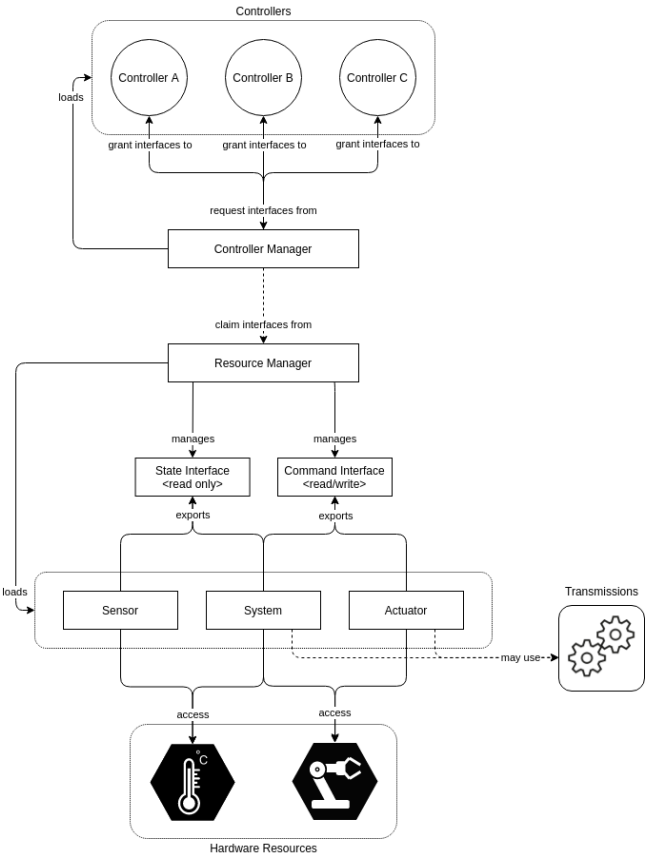


Figura 3.12: Arquitectura de ros2_control, tomada de la documentación de ros2_control

Controller Manager

El Controller Manager (CM) se encarga de conectar las controladoras (controllers) y las abstracciones de hardware de ros2_control (Hardware Components). El CM es capaz de cargar, activar, desactivar y descargar controladoras y las interfaces que necesiten. A su vez, tiene acceso a los componentes de hardware a través del Resource Manager. De esta manera es capaz de actuar como intermediario leyendo y escribiendo entre las interfaces de los componentes de hardware y las controladoras. También se encarga de reportar errores en caso de que la carga de una controladora falle.

En la ejecución del ciclo de control, la lectura de la información de los componentes de hardware, la actualización de los controladores activos y la escritura de los resultados en los controladores son otras de las responsabilidades del CM a través del método 'update()'.

Resource Manager

El Resource Manager (RM) abstrae la interacción entre el hardware físico (una ESC por ejemplo) y el software que lo controla, llamado Hardware Components (componente de hardware). Carga los componentes y administra su ciclo de vida junto con las interfaces de estado y comandos que éstos exponen.

Esta abstracción permite la reutilización de componentes de hardware ya implementados y flexibilidad a la hora de utilizar las interfaces de estado y comandos, por ejemplo, aislando la implementación a bajo nivel del resto del sistema.

En el ciclo de control, el RM se encarga de los métodos de lectura ('read()') y escritura ('write()') que se encargan de la comunicación con los componentes de hardware.

Controllers

Las controladoras (controllers) de ros2_control son programas basados base de la teoría de control. Comparan los valores de referencia con los valores medidos y, basados en la diferencia, calculan una nueva entrada al sistema. Las controladoras se implementan extendiendo la clase 'ControllerInterface', la cual forma parte del paquete ros2_control. También existe una librería de controladoras típicas creadas por la comunidad, por ejemplo 'DiffDriveController', utilizada para control en robots diferenciales. El ciclo de vida de estas controladoras está inspirado en el ciclo de vida definido para los nodos de ROS 2. En la figura 3.13 se ilustra la máquina de estados que describe este proceso, mostrando las distintas fases y transiciones que experimentan durante su operación.

El método 'update()' del ciclo de control permite que las controladoras accedan a las interfaces de estado más recientes y escriban sobre las interfaces de comando.

Hardware Components

Los componentes de hardware, implementados como plugins, se encargan de la comunicación entre el dispositivo físico y la abstracción que realiza ros2_control. El RM carga dinámicamente los componentes de hardware y administra sus ciclos de vida, también basados en el ciclo de vida de nodos de ROS 2 de la figura 3.13.

Existen tres tipos básicos de componentes:

Sistema (System):
Hardware complejo, de múltiples grados de libertad. Este tipo de componente tiene capacidades de lectura y escritura. La principal diferencia entre este y el actuador es la posibilidad de utilizar transmisiones complejas, por ejemplo, en el caso de manos robot humanoides.

Sensor:
Hardware utilizado para sensar el entorno. Este tipo de componente solamente puede usarse como lectura.

Actuador (Actuator):
Hardware robótico simple, de un grado de libertad (1 DOF, del inglés Degree of

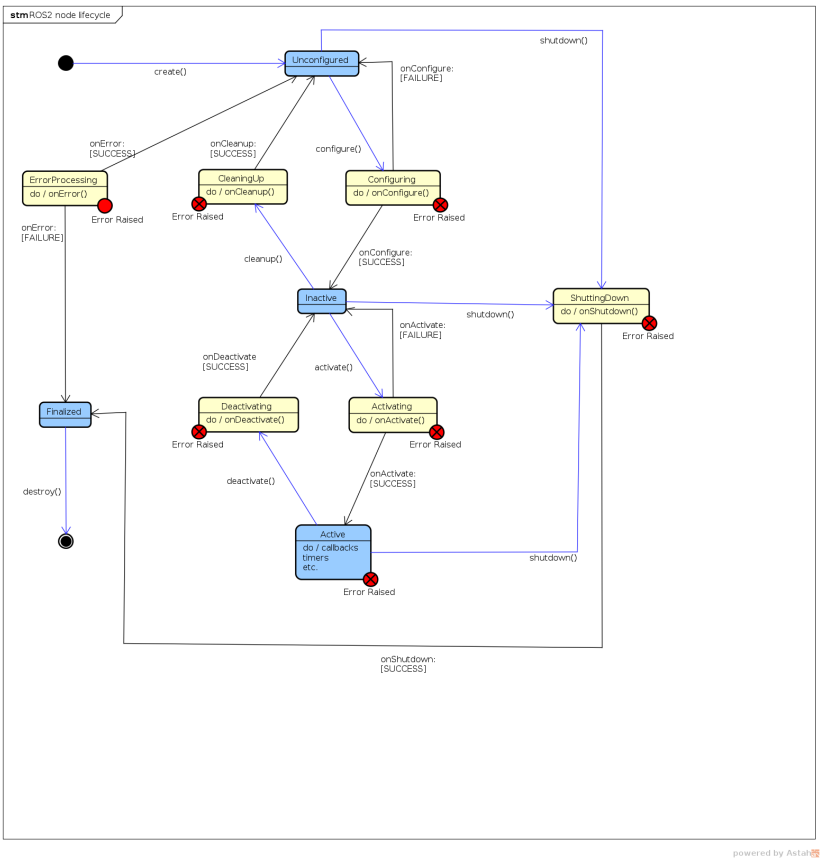


Figura 3.13: Ciclo de vida de nodos en ROS2, tomada de design.ros2.org

Freedom), como motores, válvulas, etc. La implementación de un actuador se relaciona a una única articulación. Este tipo de componentes tiene capacidades de lectura y escritura, aunque las de lectura son opcionales.

La definición de los componentes de hardware se realiza a través del 'Hardware Interface', en el que se implementa un conjunto de métodos definidos por la clase que hereda: 'SystemInterface', 'SensorInterface' o 'ActuatorInterface'

Interfaces

Las interfaces de estado y comandos son utilizadas por las controladoras (controllers) para comunicarse con el resto del sistema que utiliza ROS. La interfaz de estado es solamente de lectura y permite al sistema acceder a la información obtenida a partir del procesamiento de las controladoras, y el estado en el que el componente de hardware se encuentra.

La interfaz de comandos se utiliza para que los mensajes recibidos por la controladora de parte del resto del sistema sean escritos como comandos para

ser interpretados por los componentes de hardware.

Existen distintos tipos de interfaces disponibles en ros2_control, los cuales son utilizados para distintos tipos de piezas de hardware y sus controladoras (controllers). Estos pueden ser: velocidad, posición, potencia, esfuerzo, temperatura, entre otros.

Hardware Description

Dentro de los archivos en formato URDF (Unified Robot Description Format, un formato que se explicará en detalle en la Sección 3.4.2) utilizados en la descripción física del robot, ros2_control utiliza el tag '$< ros2\_control >$' para describir los componentes de hardware con los que interactúa. Por ejemplo, el siguiente fragmento ilustra la configuración para dos articulaciones de un brazo robótico, donde se incluyen los límites en radianes permitidos (en este caso entre -1 y 1) para la posición angular de los actuadores asociados a esas articulaciones:

```
1
2  <!-- Nombre y tipo del componente de hardware -->
3  <ros2_control name="RRBotSystemPositionOnly" type="system">
4   <hardware>
5      <!-- Directorio del plugin del componente hardware -->
6      <plugin>ros2_control_demo_hardware/
       RRBotSystemPositionOnlyHardware</plugin>
7   </hardware>
8   <!-- Primera Articulación -->
9   <joint name="joint1">
10     <!-- Interfaz de comandos de posición, definiendo valores má
       ximos y mínimos -->
11    <command_interface name="position">
12      <param name="min">-1</param>
13      <param name="max">1</param>
14    </command_interface>
15    <!-- Interfaz de estado de posición-->
16    <state_interface name="position"/>
17  </joint>
18   <!-- Segunda Articulación -->
19  <joint name="joint2">
20    <command_interface name="position">
21      <param name="min">-1</param>
22      <param name="max">1</param>
23    </command_interface>
24    <state_interface name="position"/>
25  </joint>
26 </ros2_control>
```

Interfaz de usuario

A su vez, el CM expone una interfaz de usuario, la cual se integra con la interfaz de línea de comandos de ROS 2, permitiendo así administrar y supervisar las controladoras y componentes de hardware.

```
1 # Ejemplos de comandos de ros2_control disponibles en la interfaz
      de linea de comandos de ROS 2:
2
```

```
# Lista controladoras, indicando su estado
$ ros2 control list_controllers
diffbot_base_controller[diff_drive_controller/DiffDriveController]
    active
joint_state_broadcaster[joint_state_broadcaster/
    JointStateBroadcaster] active


# Listar interfaces de hardware
$ ros2 control list_hardware_interfaces
command interfaces
      left_wheel_joint/velocity [available] [claimed]
      right_wheel_joint/velocity [available] [claimed]
state interfaces
      left_wheel_joint/position
      left_wheel_joint/velocity
      right_wheel_joint/position
      right_wheel_joint/velocity
```

3.4.2. URDF

URDF (Unified Robot Description Format) es un formato de archivo basado en XML utilizado para definir modelos de robots. Su estructura sigue un esquema de árbol, donde los elementos principales son los links (enlaces) y joints (articulaciones). Dentro de estos, se pueden especificar propiedades clave como la representación visual, las colisiones y los parámetros de inercia, así como la incorporación de sensores, lo que permite simular datos del entorno. A partir de estos elementos y sus respectivos valores, es posible construir modelos robóticos detallados. Estos modelos pueden visualizarse en herramientas como Gazebo o Rviz, facilitando su simulación y análisis en entornos de ROS. En la figura 3.14 se observa el robot TurtleBot 4 junto a su modelado 3D a partir de un archivo URDF.

A continuación se presenta un fragmento de un archivo URDF de la definición de hardware de una rueda:

```
    <!-- Asignación de variables globales -->

    <xacro:property name="wheel_radius" value="0.16"/>
    <xacro:property name="wheel_thickness" value="0.06"/>
    <xacro:property name="wheel_mass" value="0.5"/>
    <xacro:property name="wheel_offset_y" value="0.55"/>

    (...)

    <!-- Definición de la articulación de la rueda izquierda -->
    <joint name="left_wheel_joint" type="continuous">
        <parent link="base_link"/>
        <child link="left_wheel"/>
        <origin xyz="0 ${wheel_offset_y} ${wheel_radius}" rpy="-${
    pi/2} 0 0" />
        <axis xyz="0 0 1"/>
    </joint>

```

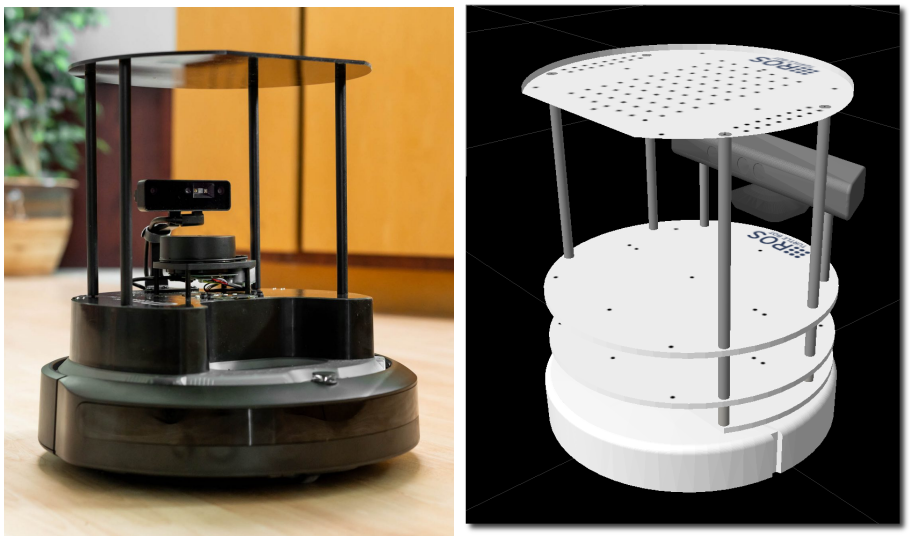


Figura 3.14: Izquierda: TurtleBot 4, tomada de Clearpath Robotics. Derecha: Modelado 3D de un TurtleBot a partir de archivo URDF tomado de la documentación de turtlebot_description.

```
18      <!-- Definición del enlace de la rueda izquierda, especificando
         geometría, material, área de colisiones e inercia -->
19      <link name="left_wheel">
20          <visual>
21              <geometry>
22                  <cylinder radius="${wheel_radius}" length="${
    wheel_thickness}"/>
23              </geometry>
24              <material name="blue"/>
25          </visual>
26          <collision>
27              <geometry>
28                  <sphere radius="${wheel_radius}"/>
29              </geometry>
30          </collision>
31          <xacro:inertial_cylinder mass="${wheel_mass}" length="${
    wheel_thickness}" radius="${wheel_radius}">
32              <origin xyz="0 0 0" rpy="0 0 0"/>
33          </xacro:inertial_cylinder>
34      </link>
```

3.4.3. Simulación y Gazebo

Al trabajar en proyectos de robótica, la simulación suele ser una muy buena inversión de tiempo y recursos. Esta es capaz de reducir costos y posibles accidentes, y aumentar el tiempo de vida de los componentes.

Gazebo es la herramienta más popular en simulación de robots de que utilizan ROS y ROS 2. Se encarga de simular en un mundo virtual al robot con sus piezas de hardware, y permitir que la ejecución del resto del sistema funcione lo más cercano a la realidad posible.

Al igual que ROS, Gazebo también cuenta con varias versiones, y algunas son más compatibles que otras frente a distintas versiones de ROS.

Utilizando distintos archivos, es posible simular detalladamente robots y entornos en los que estos robots se encuentran en el mundo real. En la figura 3.15, a la izquierda, se puede observar un ejemplo de un robot simulado, y a la derecha, se puede observar un robot y su entorno simulado, representando el caso de uso real del mismo.

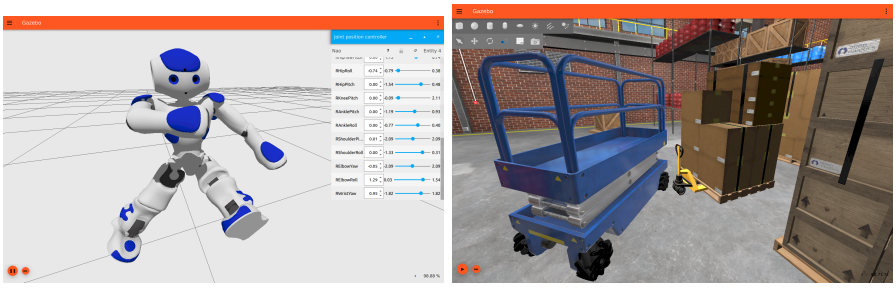


Figura 3.15: Robot modelado en Gazebo, y Robot y entorno modelados en Gazebo

Para manejar la interacción entre ROS 2 y Gazebo, se utiliza un paquete llamado ros_gz_bridge. Utilizando este paquete, los tópicos de ROS 2 especificados son escuchados y comunicados a Gazebo, e igualmente, la información sobre sensores en Gazebo es publicada en tópicos. Un archivo de configuración para este paquete sigue el siguiente formato:

```
- ros_topic_name: "cmd_vel_unstamped"
  gz_topic_name: "cmd_vel"
  ros_type_name: "geometry_msgs/msg/Twist"
  gz_type_name: "gz.msgs.Twist"
  direction: ROS_TO_GZ
```

En este caso, Gazebo sería capaz de interpretar los mensajes de tipo Twist enviados al tópico 'cmd_vel_unstamped' de ROS 2. La dirección de los mensajes puede ser: 'BIDIRECTIONAL', 'GZ_TO_ROS' o 'ROS_TO_GZ' como indica el ejemplo.

3.4.4. Odometría

La odometría es un método utilizado en robótica para estimar la posición y orientación de un robot en el espacio a partir de datos de movimiento. Se basa en la integración de información proveniente de sensores como encoders o sistemas de visión. A través del cálculo de desplazamientos y rotaciones, la

odometría permite obtener una estimación continua de la pose del robot en base a una referencia. Sin embargo, debido a la acumulación de errores (drift), suele complementarse con otras técnicas, como la localización basada en mapas y sensores externos. En ROS, la odometría se representa mediante mensajes estándar como nav_msgs/Odometry, que contienen información de posición, velocidad y covarianza del sistema.

rtabmap_odom

RTAB-Map o Real-Time Appearance-Based Mapping (Mapeo en tiempo real basado en apariencia) es un proyecto de ROS que presenta un conjunto de paquetes cuyo objetivo es mapear entornos. Uno de los paquetes que RTAB-MAP ofrece es rtabmap_odom, el cual permite estimar odometría a partir de información visual, como pueden ser imágenes de cámaras estéreo o un LiDAR.

Este paquete publica en el tópico 'odom', mensajes de tipo 'nav_msgs/Odometry', junto con otros tópicos de información complementaria, y según el tipo de sensores disponibles, espera diferentes tópicos de entrada. En el caso de un sistema con LiDAR el nodo 'icp_odometry', por Punto Iterativo más cercano (del inglés **I**terative **C**losest **P**oint), se esperan mensajes de tipo 'sensor_msgs/LaserScan' en el tópico 'scan', o 'sensor_msgs/PointCloud2' en 'scan_cloud', y en el tópico '/tf' espera una transformada entre el marco 'base_link' al marco de referencia del sensor utilizado. En caso de un LiDAR por ejemplo, esta transformada va de 'base_link' a 'laser_frame'. Se ejecuta de la siguiente manera:

```
ros2 rtabmap_odom icp_odometry
```

3.4.5. SLAM y Navegación

SLAM que significa Localización y Mapeo Simultáneos (del inglés **S**imultaneous **L**ocalization **A**nd **M**apping) es un área fundamental que abarca múltiples disciplinas, como la robótica.

El concepto de localización se refiere a determinar la pose del objeto en el espacio, mientras que el de mapeo consiste en representar o visualizar el entorno que rodea al objeto.

Las técnicas de SLAM pueden clasificarse generalmente en dos categorías: basadas en visión (normalmente utilizando cámaras estéreo) y basadas en LiDAR.

Estas técnicas se utilizan para alimentar algoritmos de planificación de caminos, buscando alcanzar los objetivos evitando obstáculos en el camino.

En la figura 3.17 se muestra un ejemplo de mapeo y localización en Rviz2, utilizando un sensor LiDAR. El robot se representa mediante una base de color naranja, con ruedas en azul y un sensor LiDAR en rojo. Los puntos rojos corresponden a las mediciones obtenidas por el sensor, mientras que los obstáculos se identifican por la acumulación de puntos negros. Las áreas libres de obstáculos se visualizan en blanco, mientras que las regiones en color turquesa corresponden a zonas sin información sensada. En la imagen se aprecian áreas en color

gris alrededor de los obstáculos. Estas corresponden a las zonas de inflación del mapa de costos, las cuales representan un aumento en el "costo" de desplazamiento al transitar cerca de un obstáculo. Dichas zonas permiten al planificador de trayectorias evitar no solo colisiones directas, sino también trayectorias demasiado cercanas a los objetos, incrementando la seguridad del movimiento del robot.

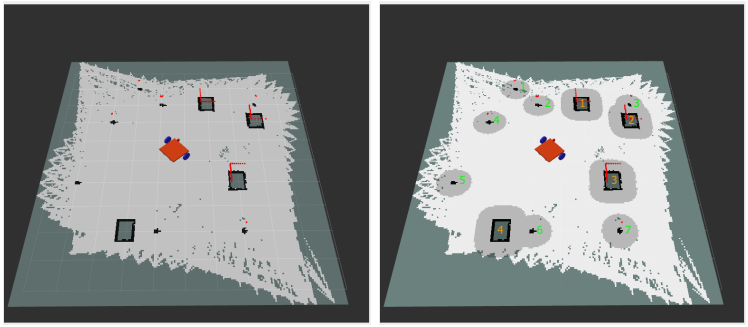


Figura 3.16: Representación en Rviz2 de mapas obtenidos con herramientas SLAM y Nav2

Figura 3.17: Mapeo y localización (izquierda), y costos según zonas asignados por planificación de caminos (derecha)

Capítulo 4

Solución Propuesta

4.1. Requerimientos

Llamaremos rosificación al proceso que tiene como objetivo ampliar las capacidades y mejorar el potencial de un robot mediante la integración con ROS. Este proceso implica: instalar ROS en el sistema del robot, configurar los controladores para que sean ejecutados y mantenidos por nodos y paquetes ROS (como ros2_control), utilizar los sistemas de comunicación entre nodos de ROS (tópicos y servicios), y habilitar el uso de herramientas de simulación y visualización (Gazebo y Rviz, utilizando URDF). Además, permite la integración con paquetes ROS de código abierto que facilitan el desarrollo y la interoperabilidad con distintas herramientas.

En nuestro caso particular, la rosificación se aplica a un robot diferencial orientado a tareas autónomas en el ámbito agropecuario. Por lo tanto, los esfuerzos se centran en resolver aspectos clave como el control de la velocidad de las ruedas, la determinación de la posición del robot, la navegación en el espacio y el reconocimiento del entorno.

Bajo el título 'Rosificando un robot para uso agropecuario' se identifican los siguientes requerimientos específicos para alcanzar una integración completa y funcional:

- Investigación sobre controladoras VESC y sus interfaces para indicar velocidades y obtener información sensada de las ruedas.
- Implementación de un sistema diferencial controlado utilizando ros2_control.
- Integración con sensor LiDAR.
- Integración con herramienta de odometría visual.
- Integración con herramientas SLAM (slam_toolbox) para mapeo y localización basado en referencias.
- Integración con herramientas de planificación de caminos (Nav2).

- Modelado 3D de Ikus basado en medidas reales.
- Soporte para simular ambientes en Gazebo, compatible con ROS 2 y ros2_control, incluyendo la creación de escenarios que simulen el caso de uso real de Ikus.
- Archivos de ejecución aptos para distintos casos de uso, parametrizados adecuadamente.
- Documentación clara sobre la ejecución y uso de las herramientas desarrolladas.

4.2. Especificaciones de Ikus y otras herramientas

Ikus es un robot autónomo de propósito general diseñado específicamente para entornos agropecuarios. La construcción del hardware fue previamente realizada por el grupo MINA. Este prototipo fue construido con componentes accesibles y tecnologías de código abierto, lo que facilita su mantenimiento y reproducibilidad. A continuación, se describirán los componentes de hardware que forman parte de Ikus y algunos componentes que resultaron útiles en el desarrollo de la solución.

4.2.1. Dimensiones generales

Las dimensiones de Ikus, presentado en la figura 4.1, en su totalidad son: 80 cm de largo, 101 cm de ancho y 47.5 cm de alto.

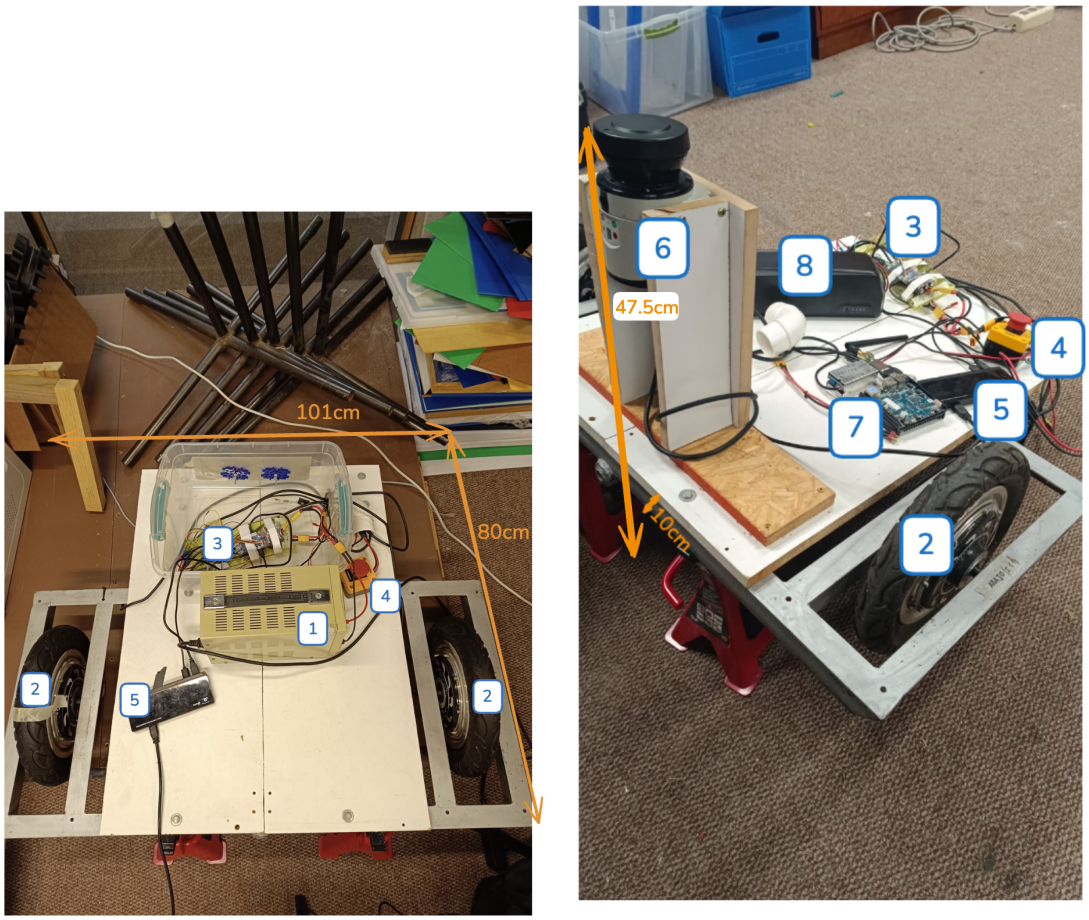


Figura 4.1: Ikus, con referencias

Chasis

El chasis se compone de un soporte de metal de 101 cm de largo y 49 cm de ancho, junto a dos tablas conjuntas de madera, que alargan a Ikus. Estas tablas unidas miden 80 cm de largo y 60 cm de ancho. El chasis en su completitud no supera los 10 cm de altura. La distancia del chasis al piso es de aproximadamente 16 cm, la mitad del diámetro de las ruedas actuantes, situadas en los laterales. Ikus también tiene dos ruedas giratorias de 12,7 cm de diámetro y 3 cm de grosor, situadas en la parte trasera.

Número	Referencia
1	Fuente regulada y regulable
2	Ruedas con motor, de kit de bicicleta eléctrica Golden Motor
3	Controladoras VESC
4	Botón de parada de emergencia
5	HUB USB
6	LiDAR LMS101-10000
7	Odroid N2+
8	Batería genérica de bicicleta eléctrica, de 36v

Tabla 4.1: Referencias para la figura 4.1

4.2.2. Actuadores y sensores

Motores y controladoras

Las ruedas utilizadas para el movimiento controlado de Ikus eran originalmente parte de un kit de conversión de bicicleta eléctrica llamado Magic Pie (SMP(E)-12F THUMB 36V KIT), de la empresa Golden Motor. Cada rueda, de 16 pulgadas de rodado, tiene un motor BLDC en su eje para realizar el giro, el cual también cuenta con encoders de efecto Hall para realizar mediciones de la posición de rueda en cada momento.

Figura 4.2: Rueda de 16 pulgadas de rodado del kit 'Magic Pie' de Golden Motor

Cada rueda está conectada a una controladora VESC, conectando un cable

para cada una de las fases necesarias para saltar los pasos del motor, y un cable para la información sensada por el efecto Hall.

La controladora se conecta a la unidad de cómputo mediante un cable USB. A través de los métodos del proyecto sbgisen/vesc, la unidad de cómputo establece y gestiona los canales de comunicación con la controladora.

Cada rueda tiene 32 cm de diámetro y 6 cm de grosor.

LiDAR LMS101-10000

El LiDAR del modelo LIDAR LMS101-10000 tiene un rango angular de 270° y un alcance de 0.5 m hasta 20 m, y puede ser utilizado con luz solar directa. Gracias a que este sensor está pensado para uso exterior, cumple con las necesidades de Ikus por estar orientado al uso en el agro. De esta forma, el sistema podrá obtener información sobre la distancia a los diferentes obstáculos.

Figura 4.3: LiDAR LMS101-10000

Se comunica con la unidad de cómputo central a través de un cable Ethernet categoría 5. Existe un paquete de ROS 2 llamado sick_scan_xd utilizado como controladora de software de LiDAR SICK. A través de este paquete se publica en un tópico, normalmente llamado '/scan', las distancias percibidas por el LiDAR.

4.2.3. Elementos de cómputo

Odroid N2

La placa Odroid N2 fue elegida inicialmente como unidad de cómputo de Ikus, pero fue descartada al haber experimentado interrupciones en la ejecución, las cuales aparentemente se originaron por falta de memoria.

ThinkPad T470s

Por comodidad, gran parte del desarrollo, configuración y simulación se realizó sobre una ThinkPad T470s con Ubuntu 22.04.

Sirio 2021

Los experimentos finales fueron realizados con una computadora de Ceibal Sirio 2021 , ilustrada en la figura 4.4, la cual se montó sobre el chasis de Ikus. Esta computadora cuenta con un CPU: Intel® Celeron® N4000 @ 1.10 GHz Dual Core, y 4GB de RAM.

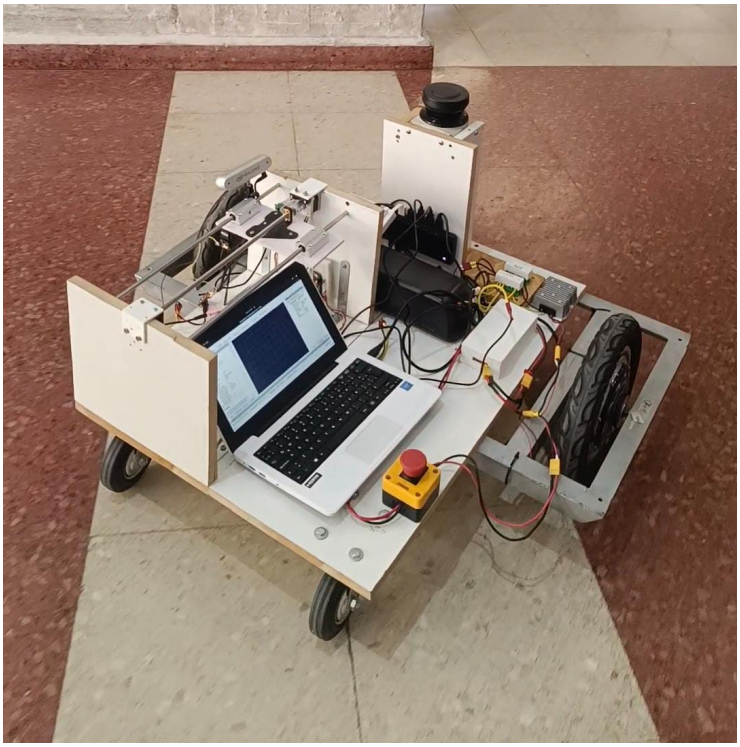


Figura 4.4: Computadora Sirio 2021 montada sobre Ikus

4.2.4. Fuentes de poder

Baterías

Se utilizó una batería genérica de bicicleta eléctrica, de 36v y aproximadamente 10Ah para alimentar el hardware del sistema: los motores de las ruedas de Ikus a través de sus controladoras VESC, el LiDAR LMS101-10000 y el HUB USB. Las unidades de cómputo utilizadas fueron alimentadas por sus propias baterías o fuentes de alimentación externas.

Fuente Regulada y Regulable

Se utilizó una fuente regulada y regulable genérica como fuente de poder en pruebas con Ikus en el laboratorio.

4.2.5. Otros Elementos

Botón de parada de emergencia

El boton de parada de emergencia se utiliza en caso de emergencia, está conectado entre la fuente de poder y la alimentación de las controladoras. Si se presiona, el sistema motriz del robot deja de ser alimentado inmediatamente.

HUB USB

Se utiliza para ampliar la cantidad de puertos USB disponibles en la unidad de cómputo y centralizar las conexiones USB.

4.3. Solución de software

Esta sección comienza explicando la arquitectura de la solución, para luego ahondar en los distintos módulos utilizados. En el anexo A se presenta la estructura de los directorios de la solución.

4.3.1. Arquitectura de la Solución

En la figura 4.5 se presenta un diagrama de la arquitectura de software de Ikus. Se muestran con diferentes figuras los distintos tipos de componentes disponibles: Hardware, Software configurado (es decir, librerías o paquetes de los que solamente se modificaron parámetros) y Software propio (haciendo referencia a la implementación llevada a cabo en el proyecto). A su vez, se pueden observar distintas formas de comunicación, las cuales pueden ser tópicos, nombres de métodos o protocolos como USB o Ethernet.

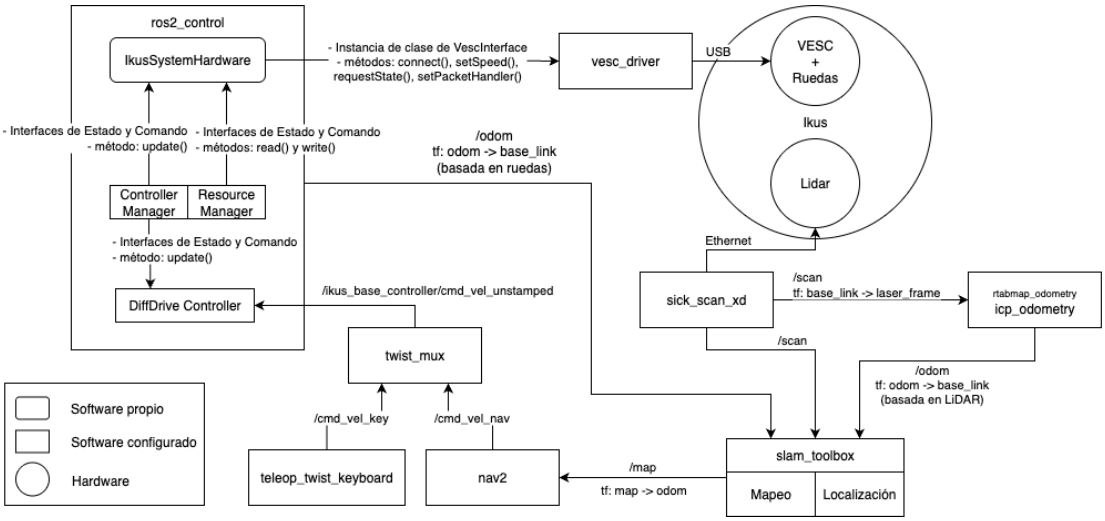


Figura 4.5: Diagrama de la arquitectura de Ikus

En las figuras 4.6, 4.7 y 4.8 se presenta un diagrama de secuencia de inicio y ejecución del sistema dividido en 3 partes.

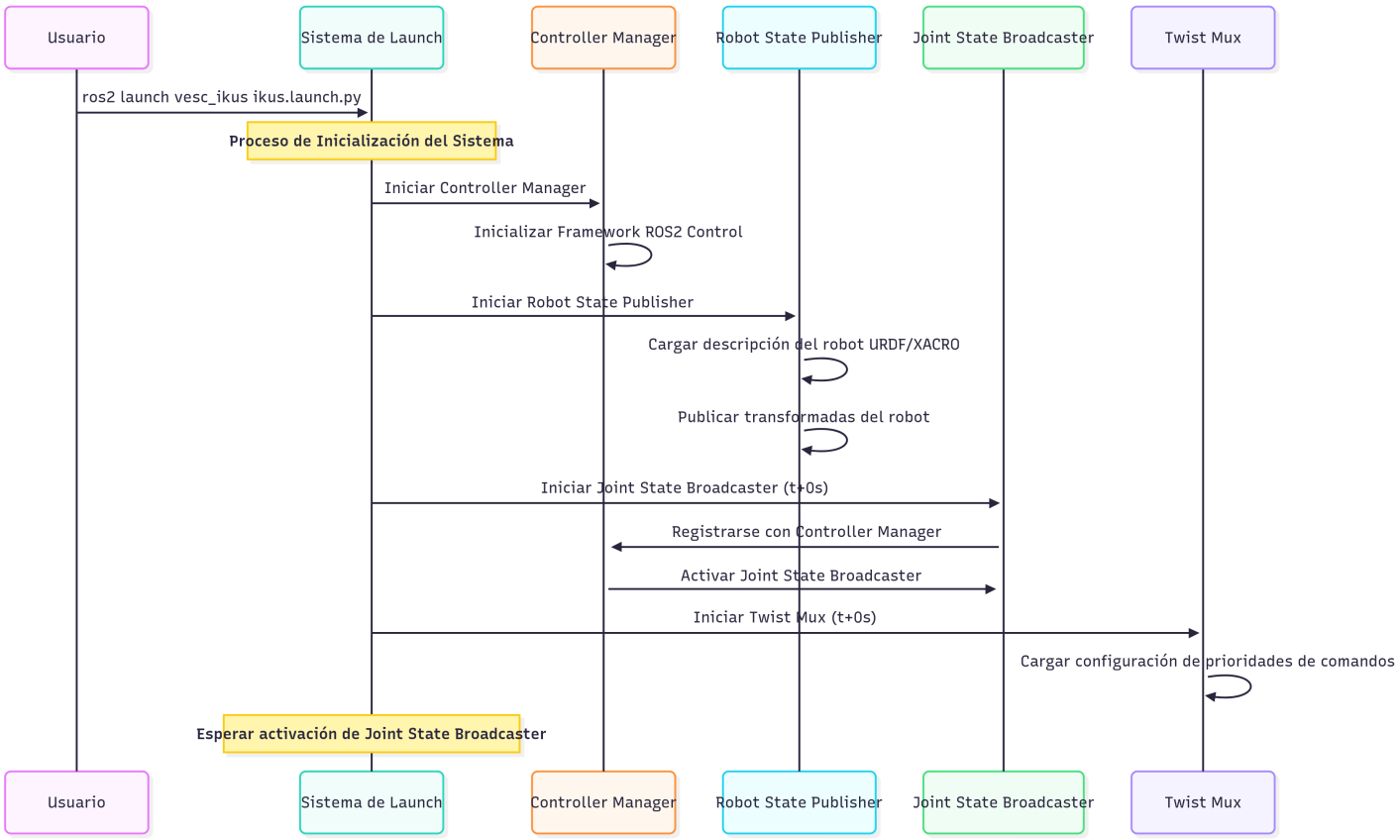


Figura 4.6: Diagrama de secuencia de inicio del sistema de Ikus: Inicio de ros2_control

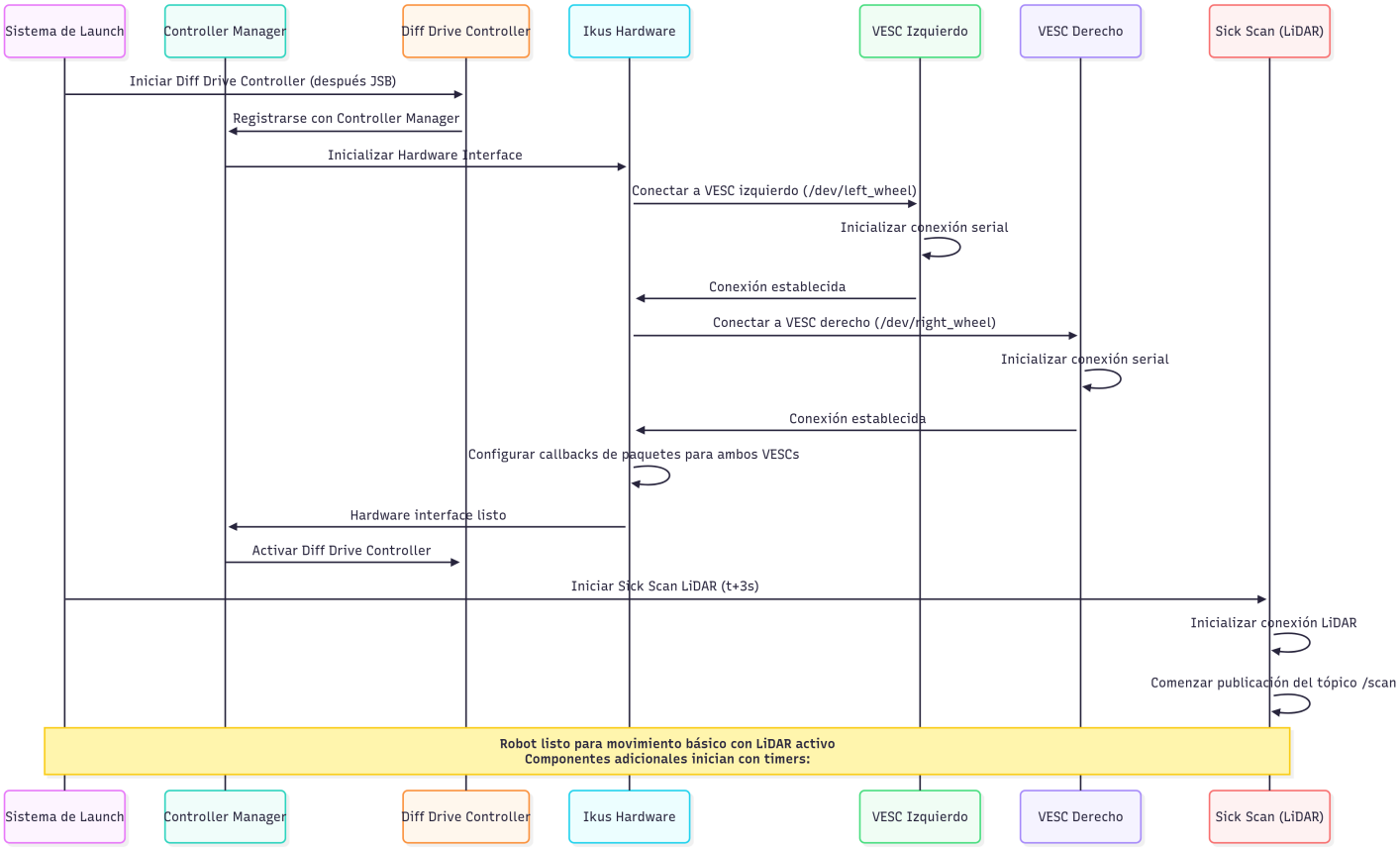


Figura 4.7: Diagrama de secuencia de inicio del sistema de Ikus: Comienzo de comunicación con controladoras VESC y LiDAR

A continuación se explicará la necesidad de los distintos módulos de la solución junto a su implementación, configuración y formas de ejecución.

4.3.2. SLAM y Navegación

Para lograr navegación autónoma, el sistema requiere capacidades de SLAM (Localización y Mapeo Simultáneos) y planificación de trayectorias. Estas funcionalidades de alto nivel dependen fundamentalmente de dos tipos de información: datos del sensor LiDAR publicados en el tópico 'scan' y transformaciones espaciales en el tópico 'tf' que indican la relación entre el marco de referencia 'odom' y 'base_link'.

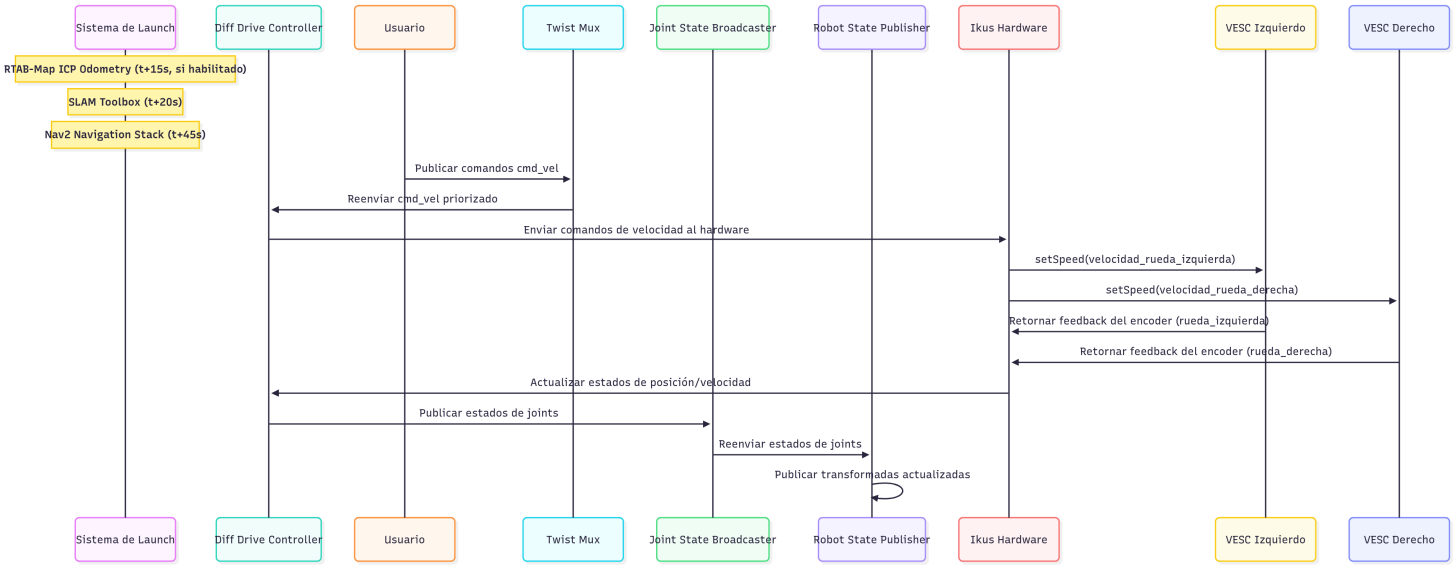


Figura 4.8: Diagrama de secuencia de inicio del sistema de Ikus: Inicio de icp_odometry, slam_toolbox y nav2, envío de comandos de velocidad a controladoras VESC

LiDAR y sick_scan_xd

Para obtener los datos del tópico 'scan' requeridos por SLAM, se utiliza el sensor LiDAR LMS101-10000 junto con el paquete sick_scan_xd. En el ambiente simulado, el LiDAR y su comunicación con ROS 2 forman parte de la simulación de Ikus en Gazebo. Para sensar las distancias desde Ikus físico, se utiliza el siguiente comando:

```
1
2 $ ros2 launch sick_scan_xd sick_lms_1xx.launch.py hostname
    :=192.168.0.1  frame_id:=laser_frame tf_base_frame_id:=
    base_link
```

Se ejecuta el archivo específico para LiDARs de la línea LMS 1XX: 'sick_lms_1xx.launch.py' con los siguientes parámetros:
hostname: indicando la dirección IP del LiDAR.
frame_id: indicando el identificador del marco de referencia del láser.
tf_base_frame_id: indicando el identificador del marco de referencia de la pieza padre del láser, especificado en la descripción de hardware.

Este paquete publica en el tópico 'scan' mensajes de tipo 'sensor_msgs/msg/LaserScan', los cuales contienen arreglos de números, donde cada número hace referencia a la distancia en metros al obstáculo más cercano según su ángulo.

Para resolver el mapeo, localización y la planificación de caminos de Ikus, se utilizaron las soluciones de 'slam_toolbox' y 'nav2'.

SLAM

La herramienta slam_toolbox se suscribe a dos tópicos: 'tf' y 'scan', para publicar otros dos tópicos: 'map' y 'pose'.

tf

'slam_toolbox' requiere que exista una transformada entre los marcos asignados a 'odom_frame' y 'base_frame'. También se encarga de proveer una transformada desde 'map_frame' a 'odom_frame', indicando que existe un marco mapa que contiene una odometría en base a ese mapa. Los nombres de los marcos son configurados en el archivo de parámetros de tipo '.yaml'.

map

En el tópico 'map', slam_toolbox envía mensajes de tipo 'nav_msgs/OccupancyGrid', calculada en base a lo obtenido en los tópicos 'scan' y 'tf'.

pose

El tipo de mensaje 'geometry_msgs/PoseWithCovarianceStamped' utilizado en el tópico 'pose' contiene la posición estimada en coordenadas con covarianza y registro temporal, en base al 'map' calculado.

Representación

En la figura 4.9 se observa la representación en rviz2 de un mapa obtenido a partir del ambiente simulado en Gazebo; los obstáculos visibles son aquellos presentados en la sección 4.3.6, presentados en la figura 4.13.

Ejecución con Ikus

Se construyó un archivo de tipo 'launch' específico para la ejecución de slam_toolbox en el contexto de Ikus para aumentar su versatilidad mediante el uso de parámetros personalizados. Se utiliza 'slam_params_file' para indicar el archivo de parámetros del paquete 'slam_toolbox', y 'use_sim_true' para indicar si se trata de la ejecución en ambiente simulado o físico.

Para la ejecución de este paquete se utiliza el siguiente comando:

```
1
2 $ ros2 launch vesc_ikus online_async_launch.py slam_params_file:=./
      src/vesc_ikus/config/mapper_params_online_async.yaml
3
4 # En caso de trabajar con ambiente simulado:
5 $ ros2 launch vesc_ikus online_async_launch.py slam_params_file:=./
      src/vesc_ikus/config/mapper_params_online_async.yaml
      use_sim_time:=true
6
7 # Para localización: se cambia el archivo de configuración
8 $ ros2 launch vesc_ikus online_async_launch.py slam_params_file:=./
      src/vesc_ikus/config/localization_params_online_async.yaml
```

Parámetros

A partir de los archivos por defecto presentados en la documentación de 'slam_toolbox', se crearon dos archivos de configuración: 'mapper_params_online_async.yaml'

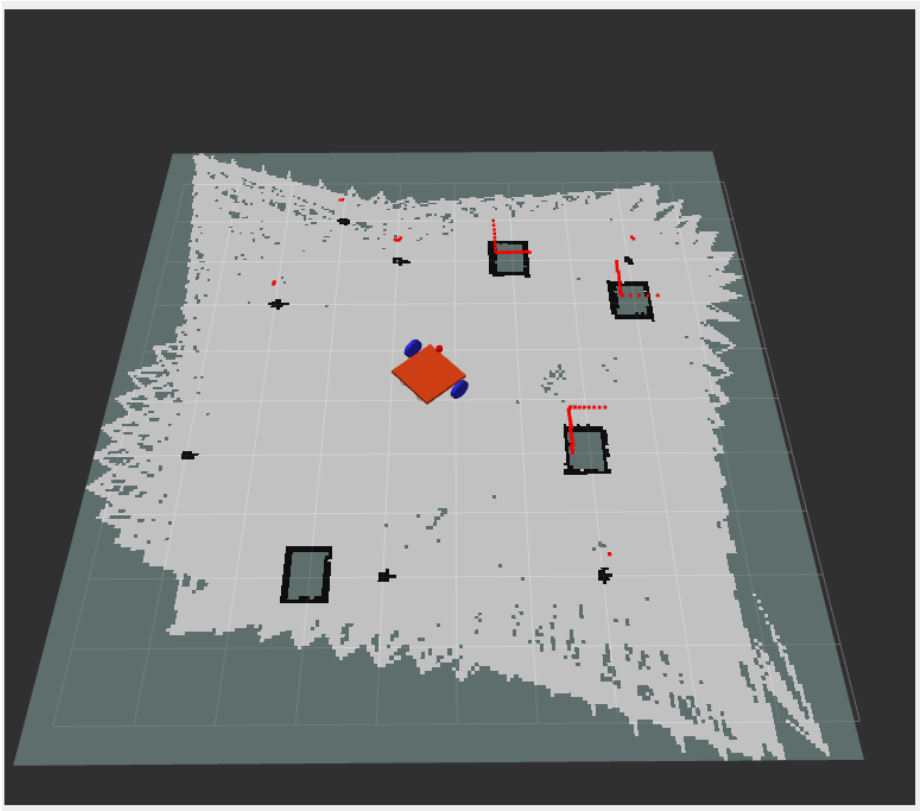


Figura 4.9: Representación en rviz2 de un mapa obtenido a partir del ambiente simulado

y 'localization_params_online_async.yaml', diferenciándose en el tipo de ejecución. En el caso de 'mapper', este se configura con el modo 'mapping', haciendo referencia al mapeo, mientras que el segundo con el modo 'localization' por localización. El segundo archivo también utiliza un parámetro llamado 'map_file_name', que debe contener la ruta al archivo de mapa guardado anteriormente.

En estos archivos también se especifican los nombres de los marcos para el caso de Ikus.

```
# Sección de archivo 'mapper_params_online_async.yaml
(...)
    odom_frame: odom
    map_frame: map
    base_frame: base_link
    scan_topic: /scan
    mode: mapping
(...)
```

Navegación

Se utilizó el entorno de trabajo conocido como nav2, el cual provee un gran nivel de abstracción a la hora de trabajar en la planificación de caminos en base a obstáculos.

map y tf Este paquete se suscribe a los tópicos 'map' (de tipo 'nav_msgs/OccupancyGrid') y 'tf' (más precisamente a la transformada entre los marcos asignados a 'map_frame' y 'odom_frame')

goal_pose

El paquete se suscribe al tópico 'goal_pose' de tipo de mensaje 'geometry_msgs/PoseStamped', en el que lee posiciones en el espacio que son interpretadas como 'objetivo' o 'destino' para el sistema robótico. Una vez interpretado, el entorno de nav2 se encargará de publicar comandos de tipo 'geometry_msgs/Twist' necesarios para intentar cumplir el objetivo.

Es posible publicar un mensaje en el tópico '/goal_pose' utilizando la interfaz de rviz2 como se comentó en la sección 3.4.1, o mediante la terminal de comandos:

```
# Ejemplo de mensaje publicado en goal_pose indicando posición
    objetivo o destino
$ ros2 topic pub /goal_pose geometry_msgs/PoseStamped "
    { header:
        { stamp: { sec: 0 },
          frame_id: 'map'
        },
      pose:
        { position: { x: 0.2, y: 0.0, z: 0.0 },
          orientation: { w: 1.0 }
        }
    }"
```

cmd_vel y cmd_vel_nav

Se publican mensajes de tipo 'geometry_msgs/Twist' en el tópico cmd_vel, el cual renombramos por cmd_vel_nav para facilitar su identificación a la hora de realizar experimentos.

'costmap' y su representación

Se denomina 'costmap' a un mapa que asigna distintos costos a distintas coordenadas, basándose en la cercanía a obstáculos. Subscribiendo rviz2 al tópico '/global_costmap/costmap' publicado por nav2, se puede ver una representación del mapa con sus costos, como se observa en la figura 4.10.

Ejecución con Ikus

Al igual que en el caso anterior, con el objetivo de personalizar la experiencia, se construyó un archivo de tipo 'launch' específico para la ejecución de la pila de paquetes de nav2 en el contexto de Ikus. Se utiliza 'params_file' para indicar el archivo de parámetros del paquete 'nav2', y 'use_sim_true' para indicar si se trata de la ejecución en ambiente simulado o físico.

Para la ejecución de este paquete se utiliza el siguiente comando:

```

```

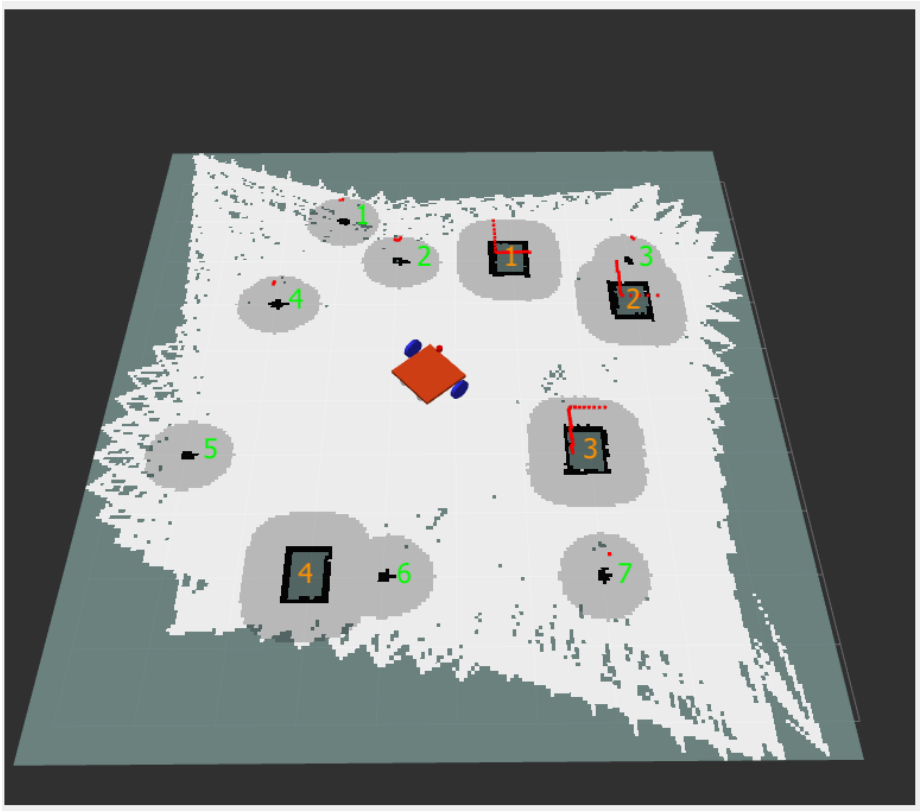


Figura 4.10: Representación en rviz2 de un mapa con costos obtenido a partir del ambiente simulado

```
$ ros2 launch vesc_ikus navigation_launch.py params_file:=src/
    vesc_ikus/config/nav2_params.yaml

# En caso de trabajar con ambiente simulado:
$ ros2 launch vesc_ikus navigation_launch.py use_sim_time:=true
    params_file:=src/vesc_ikus/config/nav2_params.yaml
```

4.3.3. ros2_control

Cuando se trabaja con varios actores y se desea que estos trabajen en conjunto, es posible establecer un sistema de control que logre interpretar los comandos recibidos y asegurarse de enviar los comandos correspondientes a cada una de las controladoras. A su vez, es posible utilizar la información obtenida a través de sensores para ajustar adecuadamente el movimiento.

Utilizando la herramienta ros2_control junto con su implementación del DiffDriveController, o Controladora de Robot Diferencial, es posible solucionar am-

bos problemas mencionados anteriormente. DiffDriveController se suscribe a un tópico '/cmd_vel' de tipo 'geometry_msgs/msg/Twist', a través del cual se indica una velocidad lineal o angular deseada del robot. A su vez, utiliza la información obtenida a través de la retroalimentación de la controladora de hardware para estimar la odometría del robot, publicándola en el tópico '/odom', de tipo 'nav_msgs::msg::Odometry', junto con las transformaciones de 'odom' a 'base_link', en el tópico 'tf', de tipo 'tf2_msgs::msg::TFMessage'.

DiffDriveController

El DiffDriveController es la pieza central que permite el control coordinado de un robot diferencial. Este controlador:

- Se suscribe al tópico 'ikus_base_controller/cmd_vel' de tipo 'geometry_msgs/msg/Twist' para recibir comandos de velocidad
- Convierte comandos de velocidad lineal y angular en velocidades específicas para cada rueda
- Utiliza la retroalimentación de los encoders para calcular la odometría
- Publica la odometría en 'ikus_base_controller/odom' (tipo 'nav_msgs::msg::Odometry')
- Publica la transformada odom→base_link en '/tf' (tipo 'tf2_msgs::msg::TFMessage')

Esta transformada odom→base_link es fundamental para que SLAM y los algoritmos de planificación de caminos funcionen correctamente, ya que proporciona la estimación de la posición del robot basada en los movimientos realizados por las ruedas.

La integración del sistema con ros2_control se realiza mediante: la implementación de una interfaz de hardware, la descripción del hardware utilizado y la configuración de los parámetros utilizados por DiffDriveController.

Interfaz del componente de Hardware

La interfaz del componente de hardware requerida por ros2_control define un conjunto de funciones que son luego invocadas por el controller manager y el resource manager en sus ciclos de vida. En el directorio: 'vesc_ikus/hardware' se encuentra 'ikus_system.cpp', junto a 'include/ikus_system.hpp', donde se implementa y presenta la interfaz: 'IkusSystemHardware'.

IkusSystemHardware es el nombre de la clase del sistema de control del componente de hardware. Este nombre, aparte de describir que se trata de la interfaz del componente de hardware de un sistema con actuadores y sensores, sirve para identificar y diferenciar a este sistema de otros sistemas de ros2_control que podrían estar funcionando en simultáneo. Al no ser este el caso, el nombre tendrá solo la función de identificación y referencia.

En la figura 4.11 se presenta el diagrama de secuencia correspondiente al proceso de inicio del componente de hardware. Por otro lado, la figura 4.12 ilustra el diagrama de secuencia que describe la interacción entre ROS 2, ros2_control y la interfaz VESC.

A continuación se presentan los distintos métodos implementados en la interfaz, necesarios para la ejecución correcta del sistema en ros2_control.

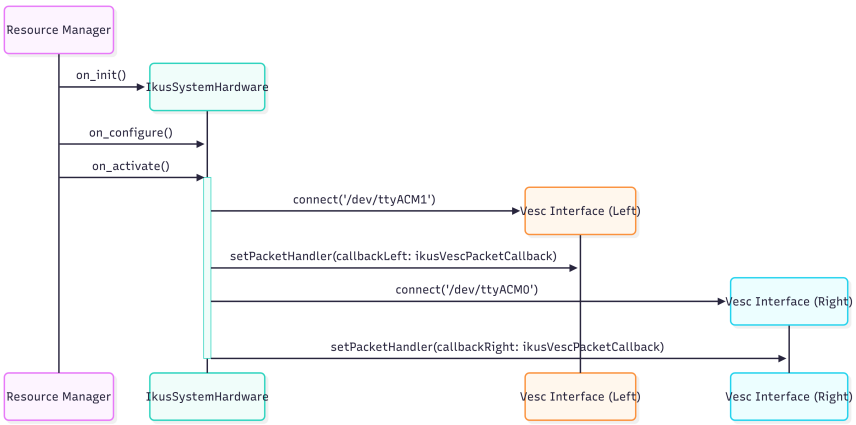


Figura 4.11: Diagrama de Secuencia del proceso de inicio del componente de hardware

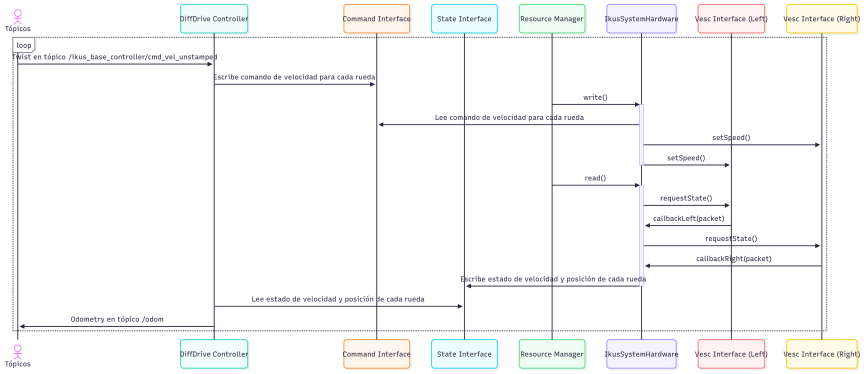


Figura 4.12: Diagrama de secuencia de interacción entre ROS 2, ros2_control y la interfaz VESC

on_init Esta función es invocada por el controller manager en su inicialización. Se encarga de asignar memoria a variables globales de la interfaz y verificar que

la cantidad de interfaces de control y de estado establecidas sea la correcta para cada tipo.

export_state_interfaces Retorna un arreglo cuyos elementos son las direcciones en memoria de cada interfaz de estado. Existen dos por rueda, una de posición y otra de velocidad, cuatro en total. De esta forma, el controller manager puede acceder directamente a los valores en memoria sin tener que invocar llamadas.

export_command_interfaces Al igual que export_state_interfaces, se retorna un arreglo cuyos elementos son las direcciones en memoria de cada una de las interfaces de control. En este caso, solamente existen interfaces de control de velocidad, una por cada rueda, dos en total. Los beneficios de rendimiento por acceso directo a memoria son iguales a aquellos que en export_state_interfaces.

on_activate Para cada instancia de vesc_interface, utilizadas para comunicación con las controladoras VESC, se establece conexión con una controladora VESC. A cada una de estas interfaces se le asigna una función, ikus_vesc_packet_callback, para ejecutar cada vez que la controladora envía un paquete de tipo vesc_packet. Este paquete proporciona información de la controladora VESC, incluyendo datos devueltos por sus sensores. Entre estos, se encuentra la diferencia de posición respecto al punto de inicio, determinada a través de los sensores de efecto Hall integrados en el motor a través del método 'getDisplacement()'. La función ikus_vesc_packet_callback, procesa la posición y guarda los resultados en un arreglo global llamado 'temp_hw_positions'.

on_deactivate Se encarga de desconectar las instancias de vesc_interface de las controladoras VESC.

write Invoca el método setSpeed($velocidad_{erpm}$) para cada instancia de vesc_interface con los valores encontrados en el arreglo de velocidades de la interfaz de comandos, los cuales fueron escritos previamente por el controller manager.

read La función read es invocada por el controller manager tantas veces por segundo como indique el update_rate en el archivo de configuración del controlador. Llama al método requestState() para cada instancia de vesc_interface, lo que implica la invocación de ikus_vesc_packet_callback, resultando en la actualización de los valores de posición y velocidad. Estos valores actualizados son asignados al arreglo de estado de velocidad y posición para que el controller manager acceda posteriormente.

Descripción de hardware - URDF

Para describir la forma en la que el robot utiliza ros2_control, se utilizan tags de <ros2_control> en la descripción de hardware del robot.

```
<ros2_control name="IkusSystemHardware" type="system">
    <hardware>
        <plugin>vesc_ikus/IkusSystemHardware</plugin>
    </hardware>
    <joint name="right_wheel_joint">
          <command_interface name="velocity"/>
          <state_interface name="position"/>
          <state_interface name="velocity"/>
    </joint>
    <joint name="left_wheel_joint">
          <command_interface name="velocity"/>
          <state_interface name="position"/>
          <state_interface name="velocity"/>
    </joint>
</ros2_control>
```

El tag <ros2_control> utiliza los parámetros de 'name' y 'type' para describir el nombre y el tipo del sistema. Bajo el tag <hardware> y <plugin> (al tratarse de una extensión de lo ya proporcionado por ros2_control), indicamos la interfaz de hardware definida en nuestro código C++. Luego, los tags <joint> definen las articulaciones del sistema, junto a las interfaces de comando y de estado que posee cada una.

Configuración y Parametrización de control

Para que ros2_control comience a ejecutarse correctamente, debe tener acceso a un archivo que contenga distintos parámetros y configuraciones del sistema. Realizando modificaciones en el ejemplo de archivo de configuración de robots diferenciales proporcionado por ros2_control en su documentación, obtenemos 'ikus_controllers.yaml' e 'ikus_controllers_without_tf.yaml'. Las modificaciones más relevantes fueron las siguientes:

Características de las ruedas

Parámetros que indican tamaño y distancia entre las ruedas fueron modificados con los valores correspondientes a Ikus:
'wheel_separation: 0.84'
'wheel_radius: 0.16'

Límites de velocidad

Los parámetros que indican los límites de velocidad fueron alterados de ±1,0 a ±1,5 en velocidad lineal, y de ±1,0 a ±0,8:
'linear.x.max_velocity = 1.5'
'linear.x.min_velocity = -1.5'
'angular.z.max_velocity = 0.8'
'angular.z.min_velocity = -0.8'

Otros parámetros

'open_loop: false' Deshabilitando 'open_loop', nos aseguramos de que la odometría del robot sea calculada en base a la retroalimentación y no según los comandos Twist de entrada.

'enable_odom_tf' La única diferencia entre 'ikus_controllers.yaml' e 'ikus_controllers_without_tf.yaml' es que el segundo tiene el parámetro 'enable_odom_tf' en false, deshabilitando la publicación de la transformada de 'odom' a 'base_link' de parte de DiffDrive-Controller. El objetivo de este cambio es evitar redundancia de transformadas entre ros2_control y rtabmap_odom.

4.3.4. Comunicación con Controladora VESC

Una vez que ros2_control calcula los comandos de velocidad necesarios para cada rueda, estos deben ser transmitidos a las controladoras físicas VESC a través de la interfaz de hardware. Para establecer esta comunicación, se utilizó una interfaz de un paquete de ROS 2 llamado 'f1tenth/vesc', originalmente creado por el equipo F1Tenth (ahora RoboRacer.AI).

Este paquete sirve como interfaz entre ROS 2 y la controladora VESC, permitiendo comunicación bidireccional: envío de comandos de velocidad y lectura del estado de los motores, incluyendo cantidad de revoluciones realizadas por cada rueda (información necesaria para el cálculo de odometría).

VescInterface y comunicación de bajo nivel

Para su integración con ros2_control, el paquete 'vesc_driver' presenta una interfaz llamada 'VescInterface' que implementa los métodos utilizados en la comunicación a bajo nivel con la controladora VESC. Esta interfaz proporciona comunicación directa con menor latencia comparada con la comunicación a través de tópicos. El equipo de SoftBank Corp. partió de la base de 'f1tenth/vesc' y agregaron actualizaciones sbgisen/vesc, las cuales forman parte de nuestra solución de software. Este directorio se encuentra en el mismo directorio que 'vesc_ikus', el cual implementa la integración con ros2_control.

A continuación se describen los métodos principales de 'VescInterface':

connect('/dev/ttyACM'): Se encarga de inicializar la conexión entre la controladora VESC, los motores BLDC y la interfaz VescInterface. Recibe como parámetro la dirección al puerto USB en el que está conectado la controladora VESC. También existe su contraparte, disconnect().

setSpeed($velocidad_{erpm}$): Se invoca con un parámetro '$velocidad_{erpm}$' de tipo 'double', que indica la velocidad en revoluciones 'eléctricas' por minuto (eRPM) a la que la rueda debe llegar. Para que este método funcione correctamente, se transforma la velocidad lineal que ros2_control calcula para esa rueda en una magnitud comprensible para la controladora VESC:

$$velocidad_{erpm} = \frac{velocidad_{lineal}}{2 \cdot \pi} \cdot 60 \cdot pares\_de\_polos\_del\_motor \cdot 0{,}7 \qquad (4.1)$$

Este cálculo se realizó en base a un comentario en el código fuente de 'vesc_driver'. A su vez, se aplica un factor de calibración experimental (0.7) debido a que la relación indicada no coincidía con la respuesta real del robot,

produciendo una velocidad mayor a la esperada. De esta manera, se asegura que el valor enviado a la controladora VESC represente la velocidad esperada.

requestState(): VescInterface cuenta con una función llamada 'packet_handler', la cual se ejecuta de forma asíncrona cada vez que se invoca requestState().

setPacketHandler(callback): Asigna una función de retorno 'callback' al 'packet_handler' de VescInterface. Este 'packet_handler' recibe como parámetro un objeto de tipo VescPacket, que contiene datos sobre el estado de la controladora. Una vez invocado requestState(), se invoca el 'packet_handler' con una captura del estado actual de la información que contiene la controladora VESC. A través del objeto 'VescPacket', accedemos al método 'getDisplacement()', el cual proporciona un valor relacionado a la cantidad de revoluciones 'eléctricas' que ha realizado el motor desde su encendido. A diferencia del caso de setSpeed, no se encontró documentación sobre la unidad del valor que retorna la función 'getDisplacement()', por lo que se le aplicó un factor experimental que se denominó 'gear_ratio'. El valor obtenido de 'getDisplacement()', $desplazamiento\_absoluto_{vesc}$ se transforma de la siguiente manera:

$$posicion\_absoluta_{rueda} = \frac{2 \cdot desplazamiento\_absoluto_{vesc}}{gear\_ratio \cdot \pi} \tag{4.2}$$

El valor de gear_ratio es configurable en el archivo 'ros2_control.xacro', y su valor por defecto es: '789432'.

Aproximadamente cada 0,1 segundos, se calcula la diferencia entre dos valores de $posicion\_absoluta_{rueda}$ consecutivos, logrando así calcular la velocidad lineal, utilizando:

$$\vec{v} = \frac{\Delta \vec{d}}{\Delta t} \tag{4.3}$$

Tipos de datos VESC

El paquete 'vesc_driver' también depende de un paquete llamado 'vesc_msgs', el cual define el tipo de datos de ROS 2 'VescState':

```
// Definición de 'VescState' en vesc_msgs

# Vedder VESC open source motor controller state (telemetry)

# fault codes
int32 FAULT_CODE_NONE=0
int32 FAULT_CODE_OVER_VOLTAGE=1
int32 FAULT_CODE_UNDER_VOLTAGE=2
int32 FAULT_CODE_DRV8302=3
int32 FAULT_CODE_ABS_OVER_CURRENT=4
int32 FAULT_CODE_OVER_TEMP_FET=5
int32 FAULT_CODE_OVER_TEMP_MOTOR=6

float64 voltage_input        # input voltage (volt)
```

```
float64 temperature_pcb      # temperature of printed circuit board
    (degrees Celsius)
float64 current_motor        # motor current (ampere)
float64 current_input        # input current (ampere)
float64 speed                # motor velocity (rad/s)
float64 duty_cycle           # duty cycle (0 to 1)
float64 charge_drawn         # electric charge drawn from input (
    ampere-hour)
float64 charge_regen         # electric charge regenerated to input
    (ampere-hour)
float64 energy_drawn         # energy drawn from input (watt-hour)
float64 energy_regen         # energy regenerated to input (watt-
    hour)
float64 displacement         # net tachometer (counts)
float64 distance_traveled    # total tachnometer (counts)
int32   fault_code
```

4.3.5. Odometría ICP con rtabmap_odom

Como se mencionó anteriormente, tanto SLAM como navegación requieren de la transformada entre 'odom' y 'base_link' publicada en el tópico 'tf' para funcionar correctamente. Esta transformada representa la odometría del robot, es decir, su estimación de posición y orientación basada en el movimiento relativo desde un punto de referencia.

En Ikus, esta odometría puede obtenerse a partir de los encoders de las ruedas (gracias a ros2_control) o, de forma alternativa, a través del algoritmo ICP (Iterative Closest Point) implementado en 'rtabmap_odom'. Este enfoque utiliza los datos del LiDAR para estimar el movimiento del robot comparando escaneos consecutivos y encuentra la transformada que mejor alinea las nubes de puntos.

Configuración de ICP Odometry

El nodo 'icp_odometry' de rtabmap_odom se configura a través del archivo 'rtabmap_icp_odometry.yaml' y se ejecuta mediante el launch file 'rtabmap_icp_odometry.launch.py'. La configuración principal incluye:

```
icp_odometry:
  ros__parameters:
    frame_id: "base_link"
    odom_frame_id: "odom"
    publish_tf: true
```

Este nodo:

- Se suscribe al tópico '/scan' para recibir datos del LiDAR
- Se suscribe al tópico '/tf' para encontrar la transformada asociada al los datos visuales, en nuestro caso: base_link→laser_frame
- Calcula la odometría comparando escaneos consecutivos usando ICP

- Publica la transformada odom→base_link en '/tf'
- Publica mensajes de odometría en '/odom'

Integración con el sistema

Para que la odometría ICP funcione correctamente, se debe desactivar la publicación de tf por parte de ros2_control, ya que el nodo 'icp_odometry' se encarga de publicar la transformación odom→base_link. Esto se logra agregando la línea 'enable_odom_tf: false' en el archivo de configuración de ros2_control.

4.3.6. Simulación

La simulación de Ikus fue realizada a través de Gazebo. En la figura 4.13 se observa el modelado 3D de Ikus en un bosque con cajas simulado. Esta sección se puede dividir en tres partes: Descripción del hardware, Simulación del mundo e Integración de Gazebo con ROS 2.

Descripción del Hardware

La descripción del hardware de Ikus fue modularizada según responsabilidades. La descripción física de Ikus es responsabilidad del archivo 'vesc_ikus/description/ikus_core.xacro'.

Por otro lado, se utilizó el plugin 'gz_ros2_control/GazeboSimSystem' para simular la interacción bajo nivel que ros2_control suele tener con las controladoras de hardware y, en nuestro caso, los motores BLDC. Una curiosidad de este plugin es que toma la responsabilidad de ejecutar algunos nodos necesarios en el ciclo de vida de ros2_control, como es el caso de 'controller_manager'. Este plugin es importado y ejecutado por Gazebo una vez definido en el archivo 'vesc_ikus/description/ros2_control.xacro', el cual también utiliza los archivos de configuración 'vesc_ikus/config/ikus_controllers.yaml' y 'vesc_ikus/config/gazebo_controllers.yaml':

```
    <ros2_control name="GazeboSimSystem" type="system">
        <hardware>
          <param name="calculate_dynamics">true</param>
          <plugin>gz_ros2_control/GazeboSimSystem</plugin>
        </hardware>
        <joint name="right_wheel_joint">
          <command_interface name="velocity"/>
          <state_interface name="position"/>
          <state_interface name="velocity"/>
        </joint>
        <joint name="left_wheel_joint">
          <command_interface name="velocity"/>
          <state_interface name="position"/>
          <state_interface name="velocity"/>
        </joint>
      </ros2_control>

      <gazebo>
```

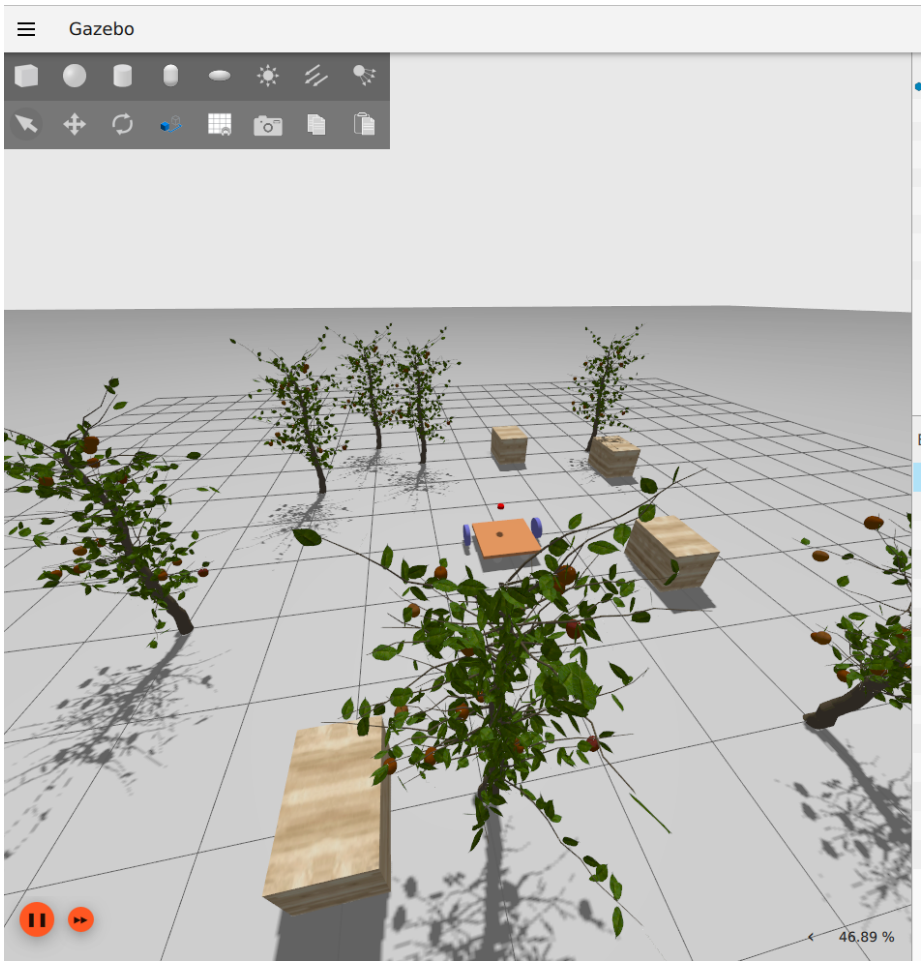


Figura 4.13: Simulación de Ikus en bosque con cajas en Gazebo

```
19          <plugin name="gz_ros2_control::GazeboSimROS2ControlPlugin"
    filename="libgz_ros2_control-system">
20            <parameters>$(find vesc_ikus)/config/ikus_controllers.
    yaml</parameters>
21            <parameters>$(find vesc_ikus)/config/gazebo_controller.
    yaml</parameters>
22          </plugin>
23        </gazebo>
```

Simulación del mundo

La construcción del mundo simulado se fundamentó en dos premisas principales. En primer lugar, se consideró que el robot sería empleado en un entorno agropecuario, siendo la asistencia en el proceso de recolección de manzanas un

ejemplo representativo de los posibles casos de uso. En segundo lugar, se buscó crear un entorno estático que permitiera realizar diferentes experimentos de manera controlada, facilitando la modificación aislada de parámetros o configuraciones para comparar resultados de forma clara y sin ambigüedades. Como resultado, se desarrolló una simulación que representa un mundo con árboles de manzanas y cajones.

La descripción de este mundo se realiza mediante archivos de tipo 'world', dentro de los cuales se definen las distintas entidades presentes en el mundo, especificando la forma o las direcciones a archivos que definen esas formas y la posición. Para los árboles, como el de la figura 4.14, se utilizó un modelo de árbol de manzanas de 'Reconocimiento y conteo de manzanas', un proyecto de grado de la Facultad de Ingeniería de la Universidad de la República realizado por Garderes (2023). El modelo de cajón de madera presente en la figura 4.15 fue construido a partir de dos modelos de la biblioteca de modelos de Open Robotics, la cual tiene entidades de uso público. Se utilizó la textura de un modelo de un 'pallet', y la geometría de un modelo de una caja de cartón.

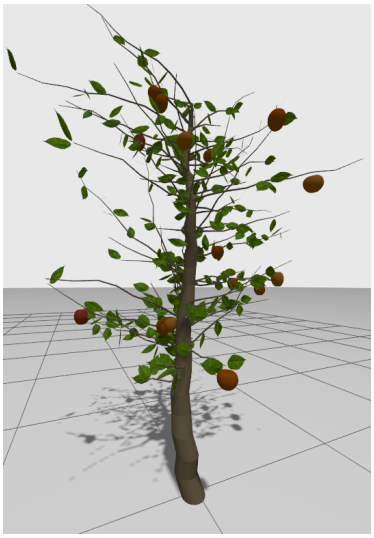


Figura 4.14: Modelo 3D de un árbol de manzanas

Además, gracias a la funcionalidad de exportación de mapas de la herramienta slam_toolbox, se incluyen en la solución los archivos ubicados en el directorio maps. Estos mapas pueden ser utilizados directamente en el proceso de localización, evitando así la necesidad de mapear el entorno desde cero.

Integración de Gazebo con ROS 2

Para realizar la integración entre Gazebo y ROS 2, se utilizan archivos tipo yaml llamados 'bridge', los cuales indican mapeos entre mensajes de tipo Gazebo y tópicos ROS 2.

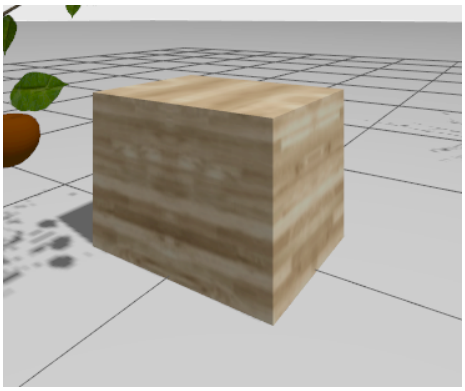


Figura 4.15: Modelo 3D de un cajón de madera

Los mapeos necesarios, especificados en los archivos 'vesc_ikus/config/gz_bridge.yaml' y 'vesc_ikus/config/gz_bridge_rtabmap.yaml' fueron los siguientes:
clock: Utilizado internamente por ROS 2 para mantenerse sincronizado con Gazebo

```
- ros_topic_name: "clock"
  gz_topic_name: "clock"
  ros_type_name: "rosgraph_msgs/msg/Clock"
  gz_type_name: "gz.msgs.Clock"
  direction: GZ_TO_ROS
```

scan: Mapeo del canal de comunicación de datos del sensor LiDAR simulado, desde Gazebo hacia ROS 2.

```
# gz topic published by Sensors plugin
- ros_topic_name: "scan"
  gz_topic_name: "scan"
  ros_type_name: "sensor_msgs/msg/LaserScan"
  gz_type_name: "gz.msgs.LaserScan"
  direction: GZ_TO_ROS
```

odom: Mapeo del canal de comunicación de los datos de Odometría, desde Gazebo hacia ROS 2. Este mapeo es omitido en 'gz_bridge_rtabmap.yaml', ya que en lugar de ser el plugin de ROS2 de Gazebo que realiza el cálculo de odometría, con esta configuración pasa a ser el nodo de rtabmap_odom.

```
- ros_topic_name: "odom"
  gz_topic_name: "odom"
  ros_type_name: "nav_msgs/msg/Odometry"
  gz_type_name: "gz.msgs.Odometry"
  direction: GZ_TO_ROS
```

tf: Mapeo del canal de comunicación de los datos de tf (Tree Frame), desde Gazebo hacia ROS 2.

```
- ros_topic_name: "tf"
```

```
  gz_topic_name: "tf"
  ros_type_name: "tf2_msgs/msg/TFMessage"
  gz_type_name: "gz.msgs.Pose_V"
  direction: GZ_TO_ROS
```

ikus_base_controller/cmd_vel_unstamped a cmd_vel: Mapeo del canal de comunicación mensajes de tipo 'Twist' para indicar movimiento al plugin de ros2_control dentro de Gazebo, desde ROS 2 hacia Gazebo.

```
- ros_topic_name: "ikus_base_controller/cmd_vel_unstamped"
  gz_topic_name: "cmd_vel"
  ros_type_name: "geometry_msgs/msg/Twist"
  gz_type_name: "gz.msgs.Twist"
  direction: ROS_TO_GZ
```

joint_states Mapeo del canal de comunicación mensajes de tipo 'JointState' para indicar el estado de las articulaciones de Ikus, desde Gazebo hacia ROS 2.

```
- ros_topic_name: "joint_states"
  gz_topic_name: "joint_states"
  ros_type_name: "sensor_msgs/msg/JointState"
  gz_type_name: "gz.msgs.Model"
  direction: GZ_TO_ROS
```

4.4. Integración del sistema completo

La solución de software implementada permite diferentes modos de ejecución según las necesidades del usuario. A continuación, se presentan los distintos launch files disponibles con sus comandos de ejecución:

Robot Físico (Ikus):

- **Movimiento básico sin SLAM/Navegación:**

```
# Con odometría de diff_drive_controller (por defecto)
$ ros2 launch vesc_ikus ikus_mapless.launch.py

# Con odometría de rtabmap\_odom (ICP)
$ ros2 launch vesc_ikus ikus_mapless.launch.py
    use_rtabmap_odometry:=true


# Este archivo launch ejecuta:

# - controller_manager ros2_control_node
#   > indicando si utiliza ikus_controller.yaml o
    ikus_controller_without_tf.yaml
#
# - robot_state_publisher robot_state_publisher
#   > con 'ikus.urdf.xacro' como parámetro
#
# - controller_manager spawner
#   > con 'joint_state_broadcaster' como argumento
#
```

```
# - controller_manager spawner
#   > con 'ikus_base_controller' como argumento
#
# - twist_mux twist_mux
#   > multiplexa comandos Twist, utiliza twist_mux.yaml como archivo de configuración
#   > mapea el tópico '/cmd_vel_out' a '/ikus_base_controller/cmd_vel_unstamped'
#
# - sick_scan_xd sick_generic_caller
#   > con 'sick_scan_xd.launch' como parámetro
#   > utiliza la información en su archivo .launch para configurar el LiDAR
#
# - rtabmap_odom icp_odometry
#   > ejecuta solamente si use_rtabmap_odometry:=true
#   > con 'rtabmap_icp_odometry.yaml' como parámetro
#
```

- **Solo SLAM y navegación:**

```
$ ros2 launch vesc_ikus slam_and_nav.launch.py use_sim_time:=false

# Este archivo launch ejecuta:

# - online_async_launch.py
#   > ejecuta el nodo slam_toolbox async_slam_toolbox_node
#   > por defecto, utiliza el archivo de configuración 'mapper_params_online_async.yaml'
#
# - navigation_launch.py
#   > ejecuta los nodos de nav2
#   > por defecto, utiliza el archivo de configuración 'nav2_params.yaml'
```

- **Lanzamiento completo:**

```
# Con odometría por defecto
$ ros2 launch vesc_ikus ikus.launch.py

# Con odometría de rtabmap\_odom
$ ros2 launch vesc_ikus ikus.launch.py use_rtabmap_odometry:=true

# Este archivo une las ejecuciones de ikus_mapless.launch.py y slam_and_nav.launch.py
```

Simulación:

- **Simulación básica sin SLAM/Navegación:**

```
# Con odometría de diff_drive_controller (por defecto)
$ ros2 launch vesc_ikus simulator_mapless.launch.py

```

```
# Con odometría de rtabmap_odom (ICP)
$ ros2 launch vesc_ikus simulator_mapless.launch.py
    use_rtabmap_odometry:=true

# Especificando un mundo diferente
$ ros2 launch vesc_ikus simulator_mapless.launch.py world_path
    :=/ruta/a/tu/mundo.world

# Este archivo launch ejecuta:


# - robot_state_publisher robot_state_publisher
#   > con 'ikus.urdf.xacro' como parámetro
#
# - controller_manager spawner
#   > con 'joint_state_broadcaster' como argumento
#
# - controller_manager spawner
#   > con 'ikus_base_controller' como argumento
#
# - twist_mux twist_mux
#   > multiplexa comandos Twist, utiliza twist_mux.yaml como
    archivo de configuración
#   > mapea el tópico '/cmd_vel_out' a '/ikus_base_controller/
    cmd_vel_unstamped'
#
# - rtabmap_odom icp_odometry
#   > ejecuta solamente si use_rtabmap_odometry:=true
#   > con 'rtabmap_icp_odometry.yaml' como parámetro
#
# - ros_gz_sim gz_sim.launch.py
#   > ejecuta Gazebo
#   > utiliza ruta al directorio del mapa como parámetro
    opcional
#
# - ros_gz_sim create
#   > crea objeto 'ikus' en la simulación, basado en tópico '/
    robot_description'
#
# - ros_gz_bridge parameter_bridge
#   > realiza mapeo entre Gazebo y ROS 2
#   > recibe un archivo como parámetro: gz_bridge.yaml o
    gz_bridge_rtabmap.yaml
```

- **Simulación completa:**

```
# Con odometría por defecto
$ ros2 launch vesc_ikus simulator.launch.py

# Con odometría de rtabmap_odom
$ ros2 launch vesc_ikus simulator.launch.py
    use_rtabmap_odometry:=true

# Este archivo une las ejecuciones de simulator_mapless.launch
    .py y slam_and_nav.launch.py
```

Herramientas adicionales:

- **Teleoperación:**

```
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-
    args -r cmd_vel:=cmd_vel_key
```

Configuración previa para simulación:

Para que Gazebo encuentre los modelos 3D, se debe exportar la variable de entorno:

```
$ export SDF_PATH="ruta/a/su/workspace/src/vesc_ikus/worlds"
```

Capítulo 5

Experimentación

En este capítulo se evalúa el rendimiento del sistema robótico desarrollado, con el objetivo principal de determinar si es capaz de estimar su posición y generar mapas de sus entornos de manera adecuada. Para ello, las pruebas se dividieron en dos contextos: entorno simulado y entorno real. Esta separación permite experimentar sobre el sistema implementado sin depender inicialmente del hardware, para luego validar el funcionamiento del sistema completo en condiciones reales.

Además, se decidió comparar los resultados de los experimentos utilizando distintas fuentes de odometría. Por un lado, se empleó la odometría basada en ruedas, provista por la solución ros2_control, y por otro, la odometría visual generada mediante icp_odometry de rtabmap_odom. Esta comparación se consideró relevante ya que la odometría es un parámetro fundamental en el proceso de mapeo.

En cuanto a la experimentación relacionada con el paquete de planificación de caminos nav2, el alcance se centró en la construcción del mapa de costos a partir de los mapas obtenidos mediante la integración con el paquete slam_toolbox.

En todos los experimentos se utilizó teleop_twist_keyboard para enviar comandos Twist indicando la velocidad deseada del robot en cada momento.

```
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args
    --remap
      cmd_vel:=/ikus_base_controller/cmd_vel_unstamped
```

5.1. Entorno Simulado

Con el objetivo de evaluar la integración del sistema robótico de manera independiente del hardware físico, se llevaron a cabo pruebas en un entorno simulado, replicando tanto la información proveniente de los sensores como parte de la integración con ros2_control (utilizando el plugin 'gz_ros2_control/GazeboSimSystem'). Para estas pruebas, se empleó el entorno descrito en la sección 4.3.6, que consiste en un mundo simulado de aproximadamente 9x9 metros, conformado por

7 árboles y 4 cajones. La posición y orientación inicial es idéntica para ambos casos.

5.1.1. Odometría basada en Ruedas

Para realizar este experimento se ejecutó el siguiente comando:

```
$ ros2 launch vesc_ikus simulator.launch.py
```

Haciendo uso de teleop_twist_keyboard, se gestionó el desplazamiento del robot simulado en el entorno virtual, lo que permitió efectuar con éxito el mapeo del mundo simulado. Este proceso se observa en la figura 5.1, donde también se muestra el mapa de costos generado a partir del mismo. Cabe destacar que para esta tarea se utilizó la información de odometría proporcionada por el sistema 'gz_ros2_control/GazeboSimSystem', la cual simula el proceso de transformar la posición y la velocidad angular de cada rueda en datos de odometría, como parte de la solución diffdrive_controller de ros2_control.

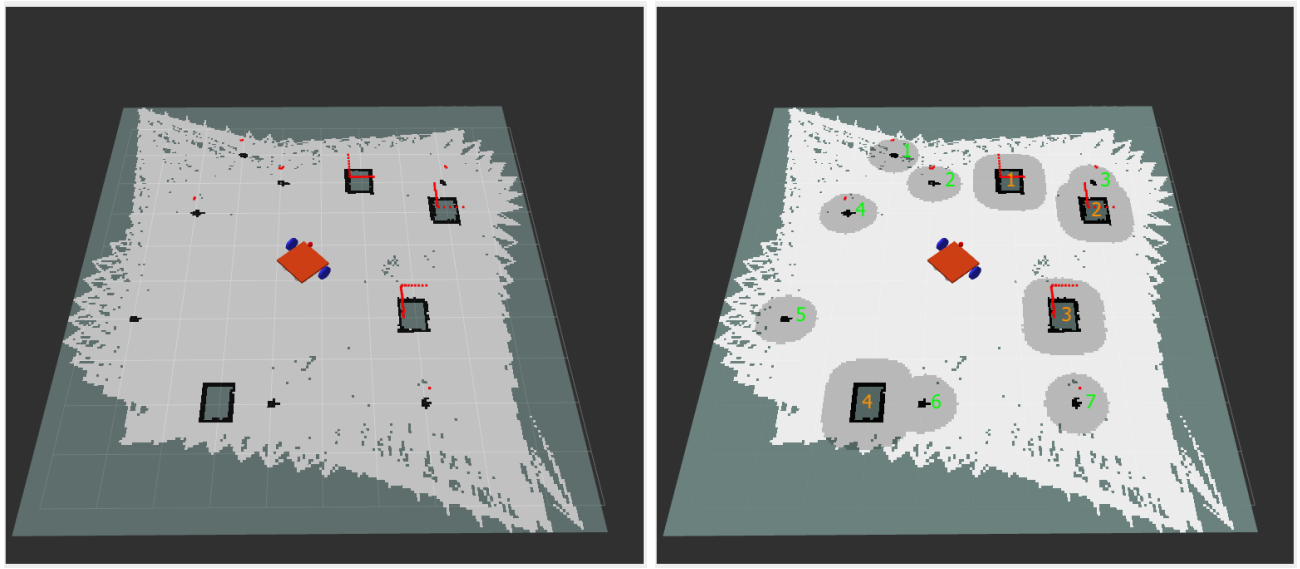


Figura 5.1: Visualización del mapeo del mundo simulado y su mapa de costos, utilizando odometría basada en ruedas

5.1.2. Odometría basada en LiDAR

Para experimentar el proceso de mapeo en base a la odometría visual se ejecutó el siguiente comando:

```
$ ros2 launch vesc_ikus simulator.launch.py use_rtabmap_odometry:=
    true
```

Al igual que en el caso anterior, se construyó el mapa y el mapa de costos, los cuales se muestran en la figura 5.2, a partir del desplazamiento del robot simulado en Gazebo. En esta ocasión, la localización se realizó utilizando la odometría visual proporcionada por icp_odometry.

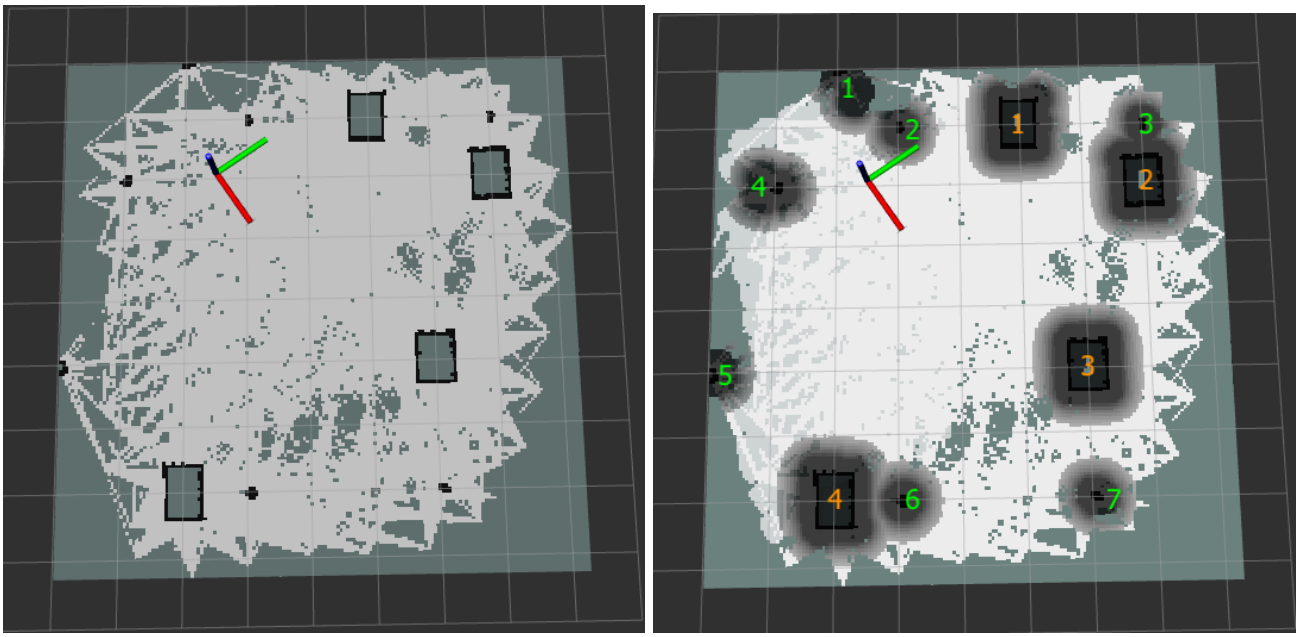


Figura 5.2: Visualización del mapeo del mundo simulado y su mapa de costos, utilizando odometría basada en LiDAR

5.1.3. Análisis

No se aprecian diferencias significativas entre los experimentos. En ambos casos, los resultados permiten distinguir claramente siete obstáculos cilíndricos pequeños y cuatro obstáculos rectangulares de casi un metro de lado. Cabe destacar que estos últimos son más notorios en los mapas de costos. En las figuras 5.1 y 5.2, los mapas de costos numeran en verde los árboles y en naranja los cajones de madera.

5.2. Entorno Real

Los experimentos principales con el robot físico fueron realizados en pasillos del primer piso de la Facultad de Ingeniería de la Universidad de la República. Como referencia, la figura 5.3 contiene una representación del recorrido realizado en el primer piso. Para ambos experimentos realizados, el recorrido fue el mismo, representado con una línea verde en la figura. Ambos experimentos comenzaron y terminaron en la misma posición, marcada con una cruz roja. La cruz azul indica la meta parcial. Para habilitar el acceso remoto a Ikus, se implementó un canal SSH entre una computadora externa y la computadora Ceibal Sirio instalada en el robot. Este canal permitió gestionar el desplazamiento del robot de manera remota utilizando la herramienta teleop_twist_keyboard durante ambos experimentos.

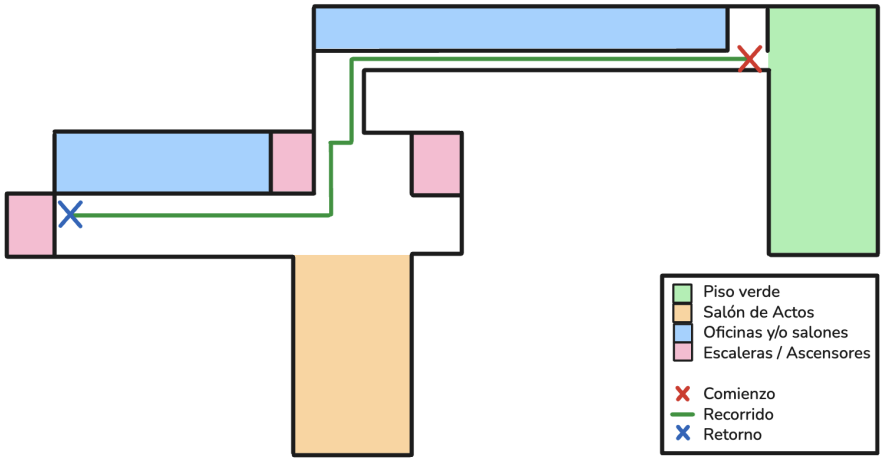


Figura 5.3: Representación del recorrido utilizado en la experimentación en el primer piso de la Facultad de Ingeniería de la Universidad de la República

Con el objetivo de optimizar la calidad del mapeo en los experimentos presentados a continuación, se optó por ejecutar el sistema de manera parcial, registrando un archivo bag con toda la información sensada para cada caso. Esta decisión se tomó para evitar la saturación del sistema durante la adquisición de datos. Posteriormente, dichos archivos fueron reproducidos y, para realizar el proceso de mapeo de manera precisa, se utilizó el siguiente comando:

```
$ ros2 launch vesc_ikus slam_and_nav.launch.py use_sim_time:=true
```

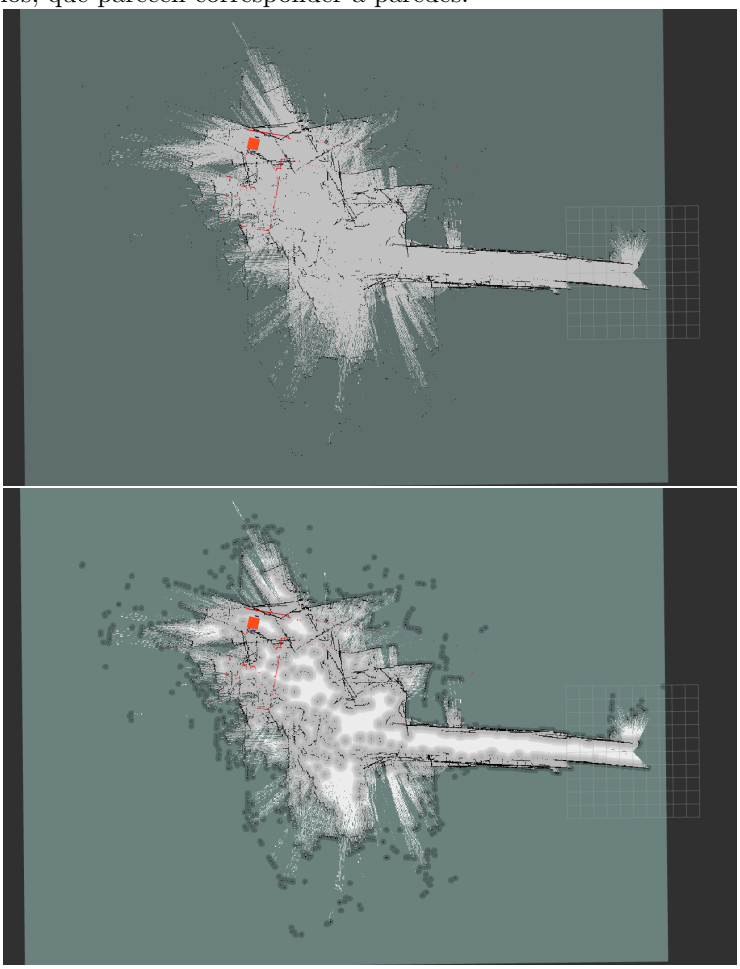
Para entender mejor los resultados obtenidos, se presenta un experimento adicional realizado en el laboratorio de robótica de la Facultad de Ingeniería de la Universidad de la República.

5.2.1. Odometría basada en Ruedas

Para ejecutar el sistema utilizando odometría basada en ruedas sin mapeo se utilizó el comando:

```
$ ros2 launch vesc_ikus ikus_mapless.launch.py
```

En la figura 5.2.1 se muestra el resultado de los mapas generados a partir de la odometría basada en ruedas, obtenida mediante ros2_control, tras realizar el recorrido ilustrado en la figura 5.3. Las diferencias entre el recorrido realizado y el mapa obtenido son notables: la posición final difiere significativamente de la inicial (ubicada cerca de la grilla de Rviz2), no se aprecia la característica forma de 'S' del pasillo recorrido y se observan múltiples superposiciones de obstáculos, que parecen corresponder a paredes.



Visualización de mapa y mapa de costos con odometría basada en ruedas, del primer piso

5.2.2. Odometría basada en LiDAR

Para ejecutar el sistema utilizando odometría visual sin mapeo, se utilizó el comando:

```
$ ros2 launch vesc_ikus ikus_mapless.launch.py use_rtabmap_odometry
    :=true
```

En la figura 5.2.2 se muestra el resultado de los mapas generados a partir de la odometría visual, obtenida mediante icp_odometry, tras realizar el recorrido ilustrado en la figura 5.3. En este caso, la posición final es muy próxima a la inicial, la forma en 'S' del recorrido se aprecia claramente en el mapa y no se observa superposición de obstáculos.

Visualización de Mapa y mapa de costos con odometría basada en LiDAR, del primer piso

5.2.3. Análisis y Experimentos en Laboratorio

Las diferencias entre ambos experimentos fueron notables, siendo el experimento de la odometría visual considerablemente más preciso. El primer experimento parece presentar algún tipo de error que genera mapas inconsistentes, los cuales resultarían inseguros y caóticos si se utilizaran para desplazamiento autónomo. Si bien anteriormente se mencionó la deriva o drift como un posible error acumulativo durante el uso prolongado de la odometría, en caso de tratarse únicamente de deriva, el error debería ser menor y el proceso de mapeo debería ser capaz de mitigarlo o corregirlo, lo cual no ocurre en este caso.

Dado que este fenómeno se presentó en la odometría basada en ruedas, se propone registrar los valores utilizados para calcular dicha odometría, los cuales se obtienen a través del VESC y de los encoders de efecto Hall de las ruedas.

Utilizando el método 'getDisplacement()' de 'vesc_driver' obtenemos valores de $desplazamiento\_absoluto_{vesc}$ provenientes de dichos encoders. Repitiendo el experimento de diferentes fuentes de odometría, esta vez en el laboratorio de robótica de la Facultad de Ingeniería de la Universidad de la República, obtenemos los mapas de la figura 5.4

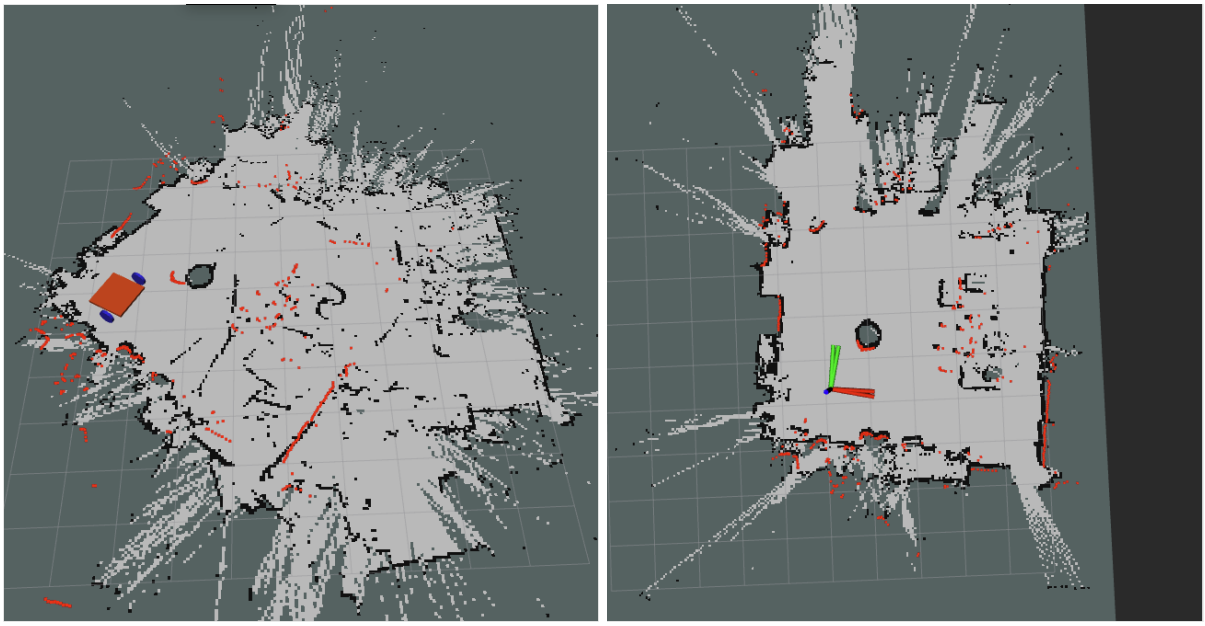


Figura 5.4: Mapas obtenidos en el laboratorio utilizando odometría basada en ruedas (izquierda) y odometría visual (derecha)

En la figura 5.5 se muestra la gráfica del desplazamiento absoluto de cada rueda en función del tiempo. En la rueda izquierda (arriba), el recorrido presenta un comportamiento mayormente continuo, mientras que en la rueda derecha (abajo) se observan varios saltos abruptos a lo largo del trayecto. Estos 'saltos'

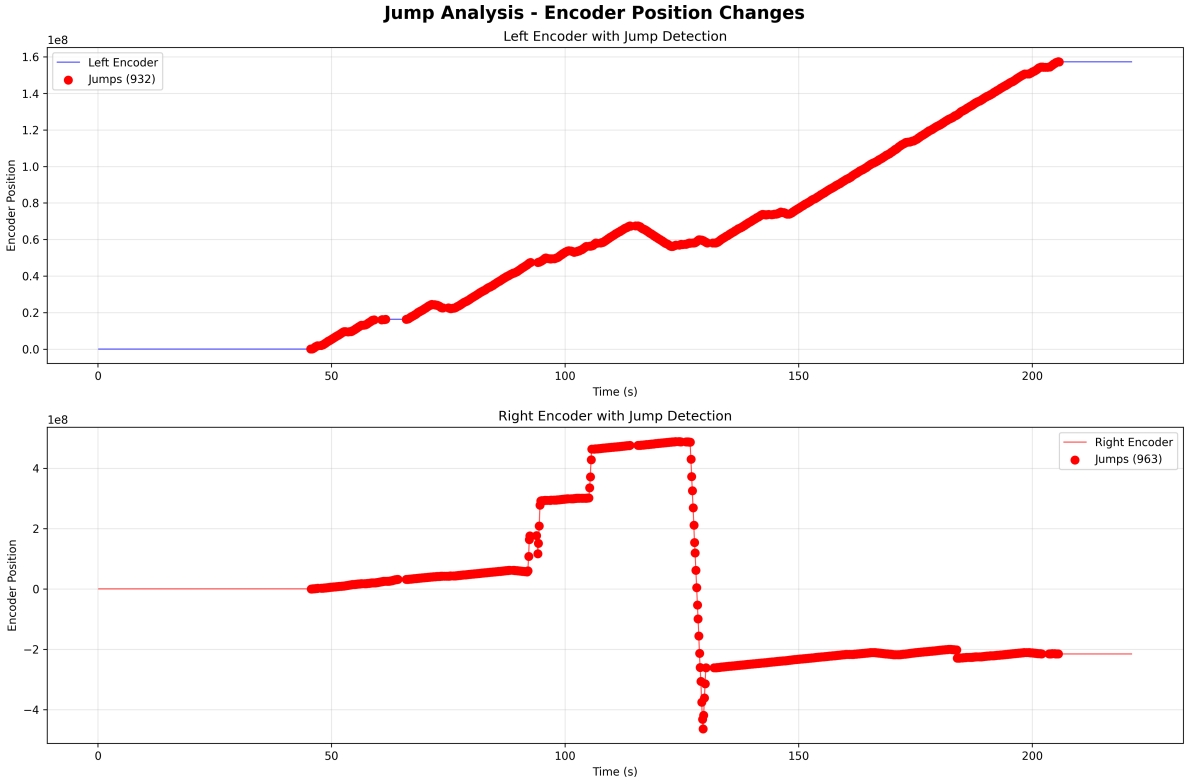


Figura 5.5: Gráfica de posiciones obtenidas para cada rueda por las codificadoras y las controladoras VESC, para rueda izquierda (arriba) y derecha (abajo)

parecen originarse por lecturas erróneas o ruidosas provenientes del controlador VESC, o bien por algún fallo en el sensor Hall instalado en la rueda. Dichas anomalías introducen ruido en el cálculo de la odometría realizado por el DiffDriveController de ros2_control, lo que a su vez afecta negativamente el mapeo generado por slam_toolbox, como se aprecia en las figuras 5.2.1 y 5.4.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

Se realizó una integración de distintas herramientas en el contexto de ROS 2 para un robot diferencial orientado al sector agropecuario. Ikus es capaz de interpretar comandos de tipo Twist en tiempo real, sensar el entorno a su alrededor y realizar un mapa y mapa de costos de ese mismo entorno. La calidad de estos mapas obtenidos varía según la fuente sensorial utilizada para el cálculo de odometría.

Aunque se tratase de una plataforma de software y hardware abierto, la integración con la controladora VESC no fue trivial. La información encontrada sobre las ruedas y la comunicación y transformación de datos con la controladora VESC fue escasa y llevó a la necesidad de ajustar parámetros experimentalmente.

Las herramientas de ros2_control fueron integradas correctamente. Una vez comprendidos los conceptos, la integración del framework fue fácil y muy valiosa para transformar las diferencias de posición de las ruedas en la posición en el espacio del robot.

La odometría obtenida de la utilización del paquete rtabmap-odom fue una alternativa de gran valor frente a los problemas encontrados con la odometría basada en ruedas.

La integración con las herramientas de slam-toolbox y nav2 supo llevar las capacidades de Ikus al siguiente nivel con poca configuración.

Debido al tamaño y peso del robot, la experimentación con el robot real fue ocasional. Gracias a la integración con Gazebo es posible realizar experimentos sin necesidad de utilizar el robot real.

Durante el proceso de experimentación, no se obtuvieron conclusiones respecto a los saltos producidos por la rueda derecha, que impactaban ampliamente en la odometría y luego en el proceso de mapeo. Antes de que suceda el primer

'salto' que contamina la odometría basada en ruedas, el mapeo parece coherente con la realidad y similar a su contraparte basada en odometría visual.

El proceso de experimentación también se vio afectado por un alto nivel de cómputo detectado una vez se ejecutaban los paquetes de 'slam_toolbox' y 'nav2', los cuales requieren buena sincronización con el resto del sistema para funcionar correctamente. Estos fenómenos ocasionalmente generaban latencia, pérdida de información, detención de módulos y reinicios del sistema de la unidad de cómputo.

El objetivo del proyecto fue cumplido parcialmente. Los experimentos realizados se enfocaron en obtener mapas claros antes de permitir que Ikus recorriera caminos planificados de forma autónoma.

6.2. Trabajo Futuro

Frente a los resultados de odometría en base a ruedas, se propone realizar un cambio en la derecha y/o en la controladora VESC del lado derecho frente a la sospecha de que los saltos sean provocados por problemas en el hardware.

La integración de una IMU o Unidad de Movimiento Inercial podría agregarle valor a la solución presente, ya que sería otra fuente de información sobre la posición en el espacio.

Las frecuencias en las que se envían las transformadas pueden ser optimizadas, mejorando el procesamiento de los paquetes de mapeo, localización y planificación de caminos.

Los experimentos fueron realizados en un entorno controlado, en pasillos rectos con paredes claras. Al tratarse de un robot agropecuario, se deben realizar experimentos de mapeo en entornos más adecuados.

El mundo simulado utilizado también puede mejorarse para adecuarse más a la realidad, y a su vez agregar mundos para pruebas controladas, como puede ser un pasillo o una habitación cerrada con algunos obstáculos.

Una actualización de unidad de cómputo también podría impactar positivamente en el rendimiento a la hora de realizar el mapeo y, más adelante, la planificación de caminos.

La actualización del sistema en su completitud de ROS 2 Humble (la cual dejará de ser soportada en mayo de 2027) a ROS 2 Jazzy (soportada hasta mayo de 2029) mantendría a Ikus en la vanguardia tecnológica.

Referencias

Amadi, C. A., Mbanisi, K., y Smit, W. J. (2024). An introduction to the ros2_control framework using a low cost differential drive robot.

Arkin, R. C. (1998). *Behavior-based robotics* (3.ª ed.). MIT Press Academic, ISBN: 9780262529204.

Benjamin's robotics - vesc – open source esc. (s.f.). https://vedder.se/2015/01/vesc-open-source-esc/. (Accessed: 2025-02-27)

Ciclo de vida de nodos en ros. (s.f.). https://design.ros2.org/articles/node_lifecycle.html. (Accessed: 2025-09-30)

Clearpath robotics. (s.f.). https://clearpathrobotics.com/. (Accessed: 2025-09-27)

Computadora sirio de ceibal. (s.f.). https://ceibal.edu.uy/institucional/articulos/hardware-sirio-2021/. (Accessed: 2025-09-05)

Conjunto de herramientas de ros de mapeo en tiempo real basados en sensores visuales. (s.f.). https://introlab.github.io/rtabmap/. (Accessed: 2025-08-23)

Código fuente de vesc tool. (2017). https://github.com/vedderb/vesc_tool. (Accessed: 2024-09-03)

Documentación de ros2_control. (s.f.). https://control.ros.org/humble/index.html. (Accessed: 2025-09-04)

Documentación de ros_control. (s.f.). http://wiki.ros.org/ros_control. (Accessed: 2024-09-03)

Documentación nav2. (s.f.). https://docs.nav2.org/. (Accessed: 2025-04-20)

Fundamentos de robótica autónoma - unidad 1.2. (s.f.). https://eva.fing.edu.uy/course/view.php?id=869. Facultad de Ingeniería, Universidad de la República. (Accessed: 2025-04-20)

Garderes, F., R. y Gutiérrez. (2023). Reconocimiento y conteo de manzanas [en línea] tesis de grado.

Gazebo. (s.f.). https://gazebosim.org/home. (Accessed: 2024-09-03)

Giampà, S. (2023). Development of an autonomous mobile manipulator for industrial and agricultural environments.

Golden motor. (s.f.). https://goldenmotor.bike/. (Accessed: 2025-02-23)

Ikus código fuente. (s.f.). https://gitlab.fing.edu.uy/christopher.friss/rosemary. (Accessed: 2025-05-04)

Li, Y., y Ibanez-Guzman, J. (2020). Lidar for autonomous driving: The princi-

ples, challenges, and trends for automotive lidar and perception systems. *IEEE Signal Processing Magazine*, *37*(4), 50–61.

Lidar lms101-10000. (s.f.). https://www.sick.com/br/es/catalog/productos/sensores-lidar-y-de-radar/sensores-lidar/lms1xx/lms101-10000/p/p346868. (Accessed: 2025-02-23)

Magic pie 3 e-bike conversion kit. (s.f.). https://www.goldenmotor.com/magicpie/magicpie.html. (Accessed: 2025-02-23)

Malu, S. K., Majumdar, J., y cols. (2014). Kinematics, localization and control of differential drive mobile robot. *Global Journal of Research In Engineering*, *14*(1), 1–9.

Odroid n2. (s.f.). https://wiki.odroid.com/odroid-n2/odroid-n2. (Accessed: 2024-09-03)

Oliveira, L. F., Moreira, A. P., y Silva, M. F. (2021). Advances in agriculture robotics: A state-of-the-art review and challenges ahead. *Robotics*, *10*(2), 52.

Paquete de rtabmap enfocado en odometría visual. (s.f.). http://wiki.ros.org/rtabmap_odom. (Accessed: 2025-08-23)

Paquete turtlebot description. (s.f.). https://wiki.ros.org/turtlebot_description. (Accessed: 2025-03-31)

Rep 1. (s.f.). https://ros.org/reps/rep-0001.html. (Accessed: 2025-09-27)

Rep 105. (s.f.). https://ros.org/reps/rep-0105.html. (Accessed: 2025-09-27)

Roboracer. (s.f.). https://roboracer.ai/. (Accessed: 2025-04-16)

Robotics, O. (s.f.). *Fuel latest models.* https://app.gazebosim.org/fuel/models. (Accessed: 2025-03-31)

Ros. (s.f.). https://www.ros.org/. (Accessed: 2024-09-03)

Ros gz bridge. (s.f.). https://index.ros.org/p/ros_gz_bridge/. (Accessed: 2025-03-31)

Ros humble. (s.f.). https://docs.ros.org/en/humble/index.html. (Accessed: 2025-09-04)

Rviz2. (s.f.). https://docs.ros.org/en/humble/Tutorials/Intermediate/RViz/RViz-Main.html. (Accessed: 2025-09-04)

Slam toolbox, steve macenski. (s.f.). https://github.com/SteveMacenski/slam_toolbox. (Accessed: 2025-04-20)

Softbank robotics group corp. (s.f.). https://www.softbankrobotics.com. (Accessed: 2025-04-16)

Teleop twist keyboard. (s.f.). https://index.ros.org/r/teleop_twist_keyboard/. (Accessed: 2025-04-06)

Tf2. (s.f.). https://docs.ros.org/en/humble/Concepts/Intermediate/About-Tf2.html. (Accessed: 2025-07-06)

Twist mux. (s.f.). https://wiki.ros.org/twist_mux. (Accessed: 2025-04-13)

Urdf. (s.f.). http://wiki.ros.org/urdf. (Accessed: 2024-09-03)

Vesc project. (s.f.). https://vesc-project.com/. (Accessed: 2024-09-03)

vesc, repositorio de f1tenth (ahora roboracer). (s.f.). https://github.com/f1tenth/vesc/tree/humble. (Accessed: 2025-04-16)

vesc, repositorio de softbank corp. (s.f.). https://github.com/sbgisen/vesc/tree/humble-devel. (Accessed: 2025-04-16)

Anexo A

Directorios de la solución

La solución de software, presente en el código fuente de Ikus, se divide en los siguientes directorios y archivos.

```
~/
|-- vesc_driver/
|-- vesc_msgs/
|-- vesc_ikus/
|    |-- config/
|    |-- description/
|    |-- hardware/
|    |-- launch/
|    |-- maps/
|    |-- worlds/
|    |-- CMakeLists.txt
|    |-- ikus_control.xml
|    |-- package.xml
|    |-- setup.cfg
|    `-- setup.py
`-- README.md
```

A.1. vesc_ikus

La mayor parte del desarrollo y configuraciones fue volcada sobre este directorio.

config
Dentro del directorio 'config', se encuentran archivos '.yaml' que contienen información de configuración para los distintos nodos y paquetes utilizados. Entre ellos:

```
# Configuración de ambiente de gazebo
gazebo_controller.yaml
gz_bridge.yaml
gz_bridge_rtabmap.yaml
# Configuración de controladora para ros2_control
```

```
ikus_controllers.yaml
ikus_controllers_without_tf.yaml
# Configuracion de rtabmap
rtabmap_icp_odometry.yaml
# Configuración para SLAM y planificación de caminos
mapper_params_online_async.yaml
localization_params_online_async.yaml
nav2_params.yaml
# Configuración de prioridad de comandos tipo Twist
twist_mux.yaml
```

description
En 'description' se incluyen los archivos de descripción de hardware del sistema.

```
# Descripción de hardware general
ikus.urdf.xacro
# Descripción de hardware, incluyendo distancias, posiciones, articulaciones
ikus_core.xacro
# Apartado de definición inercial, utilizada en ikus_code.xacro
inertial_macros.xacro
# Definición del hardware actuador bajo ros2_control, definición de plugin de interfaz de hardware, interfaces de estado y comando y límites
ros2_control.xacro
# Definición del hardware del sensor, indicando distancias y límites para simulación en Gazebo
lidar.xacro
```

hardware
El directorio 'hardware' contiene el archivo 'ikus_hardware.cpp' en el que se implementa el componente de hardware utilizado 'IkusSystemHardware', que a su vez utiliza el archivo 'hardware/include/vesc_ikus/ikus_system.hpp' para su definición.

launch
Los archivos de inicio de distintas partes del sistema se ubican en el directorio 'launch'

```
# Comienza la ejecución del paquete de odometría visual
rtabmap_icp_odometry.launch.py

# Archivo de configuración de sick_scan_xd
sick_scan_xd.launch

# Comienza la ejecución del sistema Ikus, ejecuta paquetes de ros2_control, twist_mux, entre otros
ikus_mapless.launch.py

# Comienza la ejecución de Ikus simulado, ejecuta Gazebo y paquetes de ros2_control, twist_mux, entre otros
simulator_mapless.launch.py
```

```
12
13 # Comienza la ejecución del paquete de Mapeo o Localización
14 online_async_launch.py
15
16 # Comienza la ejecución del paquete de planificación de caminos
17 navigation_launch.py
18
19 # Comienza online_async_launch.py y navigation_launch.py
20 slam_and_navigation.launch.py
21
22 # Comienza lo mismo a ikus_mapless.launch.py sumado a
       slam_and_navigation.launch.py
23 ikus.launch.py
24
25 # Comienza lo mismo a simulator_mapless.launch.py sumado a
       slam_and_navigation.launch.p
26 simulator.launch.py
```

maps
En 'maps' se almacenan registros de mapeo realizados en el ambiente simulado.

```
1 forest_slam_map.pgm
2 forest_slam_map.yaml
3 forest_slam_map_serialized.data
4 forest_slam_map_serialized.posegraph
```

worlds
El directorio 'worlds' contiene el archivo 'forest.world' que es utilizado por la simulación en Gazebo para representar un bosque en el ambiente virtual. En el directorio también se encuentran subdirectorios 'cardboard_box' y 'apple_10' que contienen recursos que simulan un cajón de madera y un árbol de manzanas.

CMakeLists.txt
Utilizado por ROS 2 para compilar en C++, este describe cómo se construye el código dentro del paquete. Gracias a la presencia de la herramienta de compilación 'ament_cmake_python', es posible compilar en C++ y Python de forma conjunta.

ikus_control.xml
En este archivo se realiza una descripción básica del componente de hardware, utilizado por ros2_control.

package.xml
En 'package.xml' se listan los paquetes de ROS 2 a utilizar, junto con la descripción básica del propio paquete.

A.2. Otros directorios y archivos

README.md

El archivo 'README.md' contiene información útil sobre la instalación y ejecución de distintos módulos y del sistema en general.

vesc_driver

En este directorio se encuentra una copia del paquete vesc_driver, el cual contiene definiciones de nodos e interfaces para realizar la comunicación a bajo nivel con la controladora Vesc.

vesc_msgs

Una dependencia del paquete vesc_driver es vesc_msgs. Esta dependencia surge de la definición de un tipo de mensaje: 'VescState.msg', utilizado para la comunicación a bajo nivel con la controladora. En este directorio se encuentra una copia de vesc_msgs.