BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA
FACULTAD DE CIENCIAS DE LA ELECTRÓNICA
MAESTRÍA EN INGENIERÍA ELECTRÓNICA,
OPCIÓN INSTRUMENTACIÓN ELECTRÓNICA

Tesis para obtener el título de:

MAESTRO EN INGENIERÍA ELECTRÓNICA

Sistema de localización y mapeo simultáneo para interiores basado en imágenes de profundidad (RGB-D)

Presenta:
Mauricio Longinos Garrido *

Directores de tesis:
M.C. Ricardo Álvarez González F.C.E.
Dr. José Fermi Guerrero Castellanos F.C.E
M.C. Alba Maribel Sánchez Gálvez F.C.C.

*Becario CONACYT

Puebla, Pue., Enero 2021

Agradecimientos

En primer lugar agradezco al Consejo Nacional de Ciencia y Tecnología (CONACYT) por el apoyo otorgado durante la realización del presente trabajo, a continuación, agradezco a la Benemérita Universidad Autónoma de Puebla, especialmente a la Facultad de Ciencias de la Electrónica por la formación proporcionada, así como el apoyo recibido durante las diferentes etapas de la realización de este trabajo.

Agradezco a mi director de tesis M. C.Ricardo Álvarez González por su apoyo, disposición y guiá para los asuntos que emergieron durante el desarrollo del presente trabajo.

Agradezco a mi co-asesor de tesis el Dr. José Fermi Guerrero Castellanos por su indiscutible guiá y apoyo tanto en el desarrollo como en la obtención de materiales, cursos y espacios de trabajo, sin lo cual el presente trabajo no hubiese sido posible.

Agradezco a mi co-asesora de tesis M.C. Alba Maribel Sánchez Gálvez por su incansable apoyo y búsqueda de materiales, recursos y bases de conocimientos que sirvieron como soporte a este trabajo.

Agradezco a mis sinodales, Dr. Aldrin Barreto Flores quien motivó y sentó bases para el presente trabajo, la M.C. Selene Edith Maya Rueda, la cual siempre mostró su apoyo y apertura durante el desarrollo tanto del trabajo como de forma académica, y a el Dr. Salvador Eugenio Ayala Raggi, quien motivó la búsqueda y comprensión de nuevas áreas de conocimiento.

Agradezco de una forma profunda y sincera el apoyo de mis familiares, a mi madre Luz María Martha Garrido quien siempre tuvo palabras de aliento y representa un gran apoyo de forma personal. Mi hermano Julio Longinos, mis tios Francisco Garrido y Jesus Garrido quienes apoyaron de diversas maneras, tanto al desarrollo profesional como personal.

Agradezco con un profundo afecto a mis amigos Sandra Huerta y Néstor Tolentino, por el mutuo apoyo durante esta etapa y por su preciada amistad. así como agradezco a Israel Sáenz, Rodrigo Cuevas, Saúl Villegas y Enrique Pérez por su amistad y compañerismo.

Agradezco sinceramente a todas las personas involucradas durante esta etapa de mi desarrollo, especialmente a aquellas que ofrecieron palabras de aliento en los momentos más desafiantes de esta etapa.

Agradezco de la manera más afectuosa posible a Zaira Angélica Hernández, quien, a pesar de las dificultades ha conformado una parte muy importante de mi desarrollo.

Resumen

En el presente trabajo, se realiza un estudio sobre la robótica móvil y la aplicación de algoritmos de SLAM en estructuras móviles, de forma más especifica sobre el robot omnidireccional (3,0), con la finalidad de integrar una base móvil capaz de desplazarse por entornos desconocidos, para que en trabajos futuros pueda servir como base de desarrollo para un asistente personal o sobre la investigación de algoritmos de control colaborativo.

Se adopta el modelo de robot (3,0), el cual, gracias a sus características, presenta diversas bondades tanto para el diseño del controlador aquí propuesto, como para el desplazamiento, ya que no cuenta con restricciones de movimiento.

Partiendo de ello, se presenta el modelo del robot (3,0), así como el desarrollo del controlador, el cual se desarrolla con la consideración de evasión de obstáculos por medio de campos atractivos/repulsivos.

El desarrollo aquí presente es creado con diferentes herramientas para lograr el objetivo, entre las cuales se pueden mencionar:

- MATLAB/Simulink: Herramienta sobre la cual se realiza el desarrollo tanto del modelo del sistema como del controlador.
- ROS: Este es un meta sistema operativo, el cual proporciona una plataforma para la ejecución de los algoritmos tanto de control como de SLAM, así como las facilidades para que los diversos componentes que conforman al sistema sean capaces de intercambiar información.
- Gazebo: Es un simulador de ambientes robóticos, el cual trabaja en conjunto con ROS para la simulación de ambientes realistas, ya que es posible simular variables físicas, así como sensores y actuadores.
- Arduino IDE: Es un entorno de desarrollo para la programación de micro controladores, el cual fue utilizado para la programación de una tarjeta Node MCU, la cual es la encargada del movimiento del robot, así como de el cálculo de su odometría.

Este sistema fue integrado dentro de una tarjeta de desarrollo Jetson NANO, la cual, se seleccionó por sus amplias prestaciones además de ser una herramienta de vanguardia y bajo costo, con la que es posible realizar el cómputo necesario para el correcto funcionamiento de todas las etapas del sistema aquí presentado.

Índice general

Índice de figuras

Siglas

GPS *"Global Positioning System"*.

ICP *"Iterative Closest Point"*.

IEEE *"Institute of Electrical and Electronics Engineers"*.

IMU *"Inertial Measurement Unit"*.

LIDAR *"Laser Imaging Detection and Ranging"*.

OMS Organización Mundial de la Salud.

RANSAC "*Random Sample Consensus"*.

ROS *"Robot Operating System"*.

SIFT *"Scale Invariant Feature Transform"*.

SLAM *"Simultaneous Localization and Mapping"*.

SURF *"Speed-Up Robust Feature"*.

UAVs *"Unmanned Aerial Vehicles"*.

URDF *"Unified Robot Description File"*.

Introducción

Desde la antigüedad, hemos sido cautivados por la forma en que los seres vivos interactúan con la naturaleza, siendo capaces de desplazarse con un mínimo esfuerzo, volar o navegar en entornos desconocidos, utilizando los sentidos, por ello, hemos tenido el deseo de emular estas capacidades en los robots que creamos, dotándolos de diferentes sensores por medio de los cuales pueden percibir condiciones del entorno, así mismo se implementan diversas metodologías para la toma de decisiones basándose dichas circunstancias para realizar alguna tarea [7].

En la robótica, de acuerdo con diversos autores, resulta complicado establecer una clasificación de los robots, ya que se reconocen diferentes formas de clasificarlos conforme a sus características, sin embargo, de acuerdo con la función que realizan se pueden dividir en: manipuladores, generadores de movimiento, móviles, acuáticos y aéreos [8]. En las diferentes categorías antes mencionadas es fundamental la habilidad de reconocer e interactuar con el entorno, por ejemplo, en la implementación de robots manipuladores, es necesario que reconozcan su entorno para evitar colisiones con las instalaciones, así como con otros robots y personal presente, la mayor parte del tiempo, debido a que estos robots están confinados a espacios bien delimitados y enfocados a tareas definidas, no es necesaria la implementación de sistemas que permitan a este conocer su entorno, el principal enfoque se centra en la detección de personas u obstáculos. Con respecto al resto de las categorías (robots móviles, aéreos y acuáticos) es crucial el reconocimiento del entorno, tanto para la localización como para el mapeo de ambientes desconocidos [7, 9].

El mayor reto para cualquier robot autónomo que tiene como principal tarea la navegación, está compuesta de dos problemas característicos; el mapeo del entorno y la localización. Para ser capaz de auto posicionarse, el robot requiere de un mapa, del mismo modo, para poder actualizar o extender el mapa se requiere de una localización precisa, este conjunto de problemas es conocido como SLAM, por sus siglas en inglés (*"Simultaneous Localization and Mapping"*) [7, 10].

En la robótica móvil, para la correcta implementación del SLAM es necesario hacer uso de la odometría, esta consiste en realizar una aproximación de la trayectoria del robot de acuerdo con su modelo matemático y a las señales que le dan movimiento, cabe destacar

que la odometría por sí sola permite obtener la trayectoria real del robot solo en casos ideales, debido a cambios en los coeficientes de rozamiento y otros factores físicos que pueden perturbar el desempeño del robot. Al combinarla con la información proveniente de otros sensores es posible reducir el error y lograr una mejor localización y cálculo de la trayectoria.

Existen muchos escenarios en los que se busca que un robot de estas características pueda operar de manera rápida y eficiente, tal es el caso de asistencia en hospitales o lugares de trabajo, del mismo modo, dispositivos domésticos de menores características como son los robots aspiradora Roomba que cuentan con una serie de sensores básicos, con los que pueden detectar colisiones, suciedad en pisos y sensores detectores de bordes con los que previenen caídas por escalones.

En la actualidad, los escenarios de asistencia resultan de gran interés a nivel mundial, ya que, de acuerdo con datos de la OMS, existe una tendencia, la cual marca el aumento de la edad media de la población, esto quiere decir, que el porcentaje de adultos mayores está en aumento con respecto al resto de la población [11, 12] es por ello que en trabajos como [13] se introducen sistemas robóticos para la asistencia de personas mayores.

Justificación

El SLAM es desde hace poco menos de una década, un problema de gran interés para diversas áreas, ya que las aplicaciones para este tipo de sistemas son diversas y puede llegar a ofrecer grandes beneficios en distintas ramas de la sociedad, como es en el ámbito industrial, en donde un sistema robotizado de este estilo podría transportar objetos de manera autónoma dentro de una bodega, en donde uno o varios robots se dedicarían a reubicar objetos, evitando así tareas repetitivas para el personal de almacén, y al mismo tiempo el desgaste por transportar objetos pesados o materiales peligrosos; asimismo este tipo de robots pudieran ser empleados para el monitoreo de áreas tanto terrestres como aéreas, en caso de ser implementados en vehículos aéreos no tripulados, UAVs, igualmente podrían ser implementados en la exploración de terrenos desconocido o en situaciones de búsqueda y rescate [7, 10].

Desafortunadamente muchos de los sistemas que existen en la actualidad trabajan bajo ambientes controlados y muchas veces dependen de infraestructura que encarece los sistemas, y debido a que en muchas ocasiones no es factible la ubicación de esta infraestructura, como en el caso de la exploración submarina o la conducción autónoma de vehículos motorizados (siendo esta última de gran interés para empresas internacionales), se busca la implementación de sistemas que no requieran de dichas infraestructuras [7, 9].

A manera de ejemplo, la empresa manufacturera de autos, Nissan, en un esfuerzo conjunto con otras empresas, se encuentran desarrollando una plataforma autónoma con láseres y tecnología de visión, con el propósito de crear mapas incorporando alertas para situaciones tanto estáticas como dinámicas, como lo pueden ser las condiciones del camino y la presencia de personas o animales en este [14].

La anterior es solo un ejemplo del trabajo de empresas de talla mundial como los son: Intel, Google, IBM, Nvidia, entre otras; que se encuentran compitiendo continuamente para mejorar el desempeño de sus sistemas y presentar la mejor solución de auto autónomo [15].

Objetivo General

Implementar un sistema de navegación para interiores a escala, capaz de desplazarse de manera autónoma dentro de un espacio cerrado, que puede representar las habitaciones de una casa de un nivel.

Objetivos Específicos

- Evaluar modelos de locomoción para la plataforma móvil.
- Implementar un modelo a escala de plataforma móvil.
- Evaluar técnicas para el mapeo de entorno, así como técnicas para la extracción de características.
- Implementar algoritmos para mapeo del entorno en lenguaje de programación Python, utilizando cámaras RGB-D.
- Evaluar algoritmos de cálculo de ruta.
- Implementar algoritmos de cálculo de ruta en lenguaje de programación Python.
- Integrar los algoritmos dentro de una tarjeta de desarrollo capaz de realizar el procesamiento de manera adecuada.

Organización de la tesis

En el capítulo 1 se presentan los antecedentes, los cuales abarcan la robótica móvil, repasando los tipos de rueda y las disposiciones de estas, para posteriormente hacer un

repaso de los sensores utilizados. Posteriormente se presentan los programas utilizados (ROS y Gazebo), finalmente se realiza una introducción a la problemática del SLAM y sus clasificaciones.
En el capítulo 2 se presenta el marco teórico para el diseño e implementación del control de movimiento del robot (3,0). Desde el modelado de los motores, hasta la implementación del controlador diseñado. Para concluir el capítulo se presentan los resultados experimentales de las implementaciones del controlador, tanto en simulación como en las variantes implementadas.
El capítulo 3 se dedica a el estudio del SLAM, donde se presenta una recopilación de los algoritmos más utilizados, así como una breve explicación de estos. Posteriormente se presentan las simulaciones así como la implementación realizada para cada variante de los algoritmos utilizados. Se concluye el capítulo con la presentación de los resultados obtenidos en dichas implementaciones.
Para concluir el trabajo se presenta un capítulo adicional, donde se expone una implementación alternativa.

Capítulo 1

Antecedentes

Con el crecimiento del campo de la robótica, se han desarrollado diversos algoritmos para los problemas de navegación, localización, mapeo entre otras tareas robóticas han crecido en complejidad a través de los años. Además, gracias a los desarrollos tecnológicos en el área de procesamiento, el área de robótica autónoma ha recibido mayor atención.

Basados en arquitecturas cada vez más robustas de hardware, los robots se vuelven capaces de incorporar sistemas de visión, como cámaras de tres dimensiones (RGB-D), escáneres láser, sensores de distancia láser entre otros, así mismo se integran algoritmos de inteligencia artificial para la identificación y navegación a través de los sensores[16]. Pero, debido a la creciente complejidad tanto de los componentes y algoritmos que se implementan en la robótica móvil, se busca, el reducir los problemas de compatibilidad y de integración, es por ello que surgen herramientas como ROS, el cual,es un meta-sistema operativo (esto quiere decir que requiere de un sistema operativo primario para funcionar, principalmente linux), el cual por medio de sus características de abstracción de hardware y su esquema modular, permite un mejor mantenimiento de códigos, además de que facilita la implementación de sistemas.

Dada la creciente popularidad de ROS, existen diversas implementaciones que pretenden acercar este sistema tanto a la academia como a la industria, tal es el caso del "TURTLEBOT" [1], el cual, es un robot de tipo uniciclo, que ejecuta nodos de ROS e incorpora diversos sensores de acuerdo a su costo. De la misma manera existen implementaciones realizadas cen el campo de la investigación, tal es el caso de [17], donde se propone una plataforma omnidireccional que incorpora un sensor LIDAR, así como cámara de profundidad Astra.

A lo largo de este capítulo se presentaran una serie de herramientas y conceptos utilizados para la integración de nuestro sistema.

[1] https://www.turtlebot.com/

1.1. Robótica móvil

El término "robot" aparece por primera vez en 1921, en la obra R.U.R. (Rossum's Universal Robots) del novelista y autor dramático checo Karel Capek, en cuyo idioma la palabra "robota" significa fuerza de trabajo o servidumbre, por aquellos años, la robótica se había introducido en númerosas fábricas, por lo que se discutía ya del poder de las máquinas y la dominación de los hombres por las máquinas, argumento de esta y otras obras teatrales, y películas de los años veinte en los que aparecen trabajadores robóticos [18].

Los robots industriales surgen de la convergencia de tecnologías del control automático y el control de máquinas y herramientas, de los manipuladores teleoperados, y de la aplicación de computadoras en tiempo real. En 1954 el ingeniero americano George Devol patentó el que se considera el primer robot industrial, un dispositivo que combinaba la articulación de un teleoperador con el eje servo controlado de una máquina de control numérico.

Tradicionalmente las aplicaciones de la robótica estaban centradas en los sectores manufactureros más desarrollados para la producción masiva, en donde se integraban dentro de ambientes controlados denominados celdas de manufactura, su tarea principal consistía en la reproducción de tareas repetitivas, como la alimentación de las distintas máquinas de componentes dentro de la celda [1].

El desarrollo de la robótica móvil responde a la necesidad de extender el campo de aplicación de la robótica, restringida inicialmente al alcance de una estructura mecánica anclada en uno de sus extremos, esto limita la operabilidad de los robots a medida que la celda de manufactura sufre modificaciones o ampliaciones, de esta forma aparecen en los años ochenta los primeros vehículos guiados autónomamente (Automatic Guided Vehicle), los cuales consistían en robots manipuladores montados sobre rieles. No es hasta la década de 1980 cuando surge una definición más concreta de la robótica móvil, en donde se plantea que un robot móvil tiene la capacidad de movimiento sobre entornos no estructurados de los que se posee un conocimiento incierto, mediante la interpretación la información obtenida a través de sus sensores y del estado actual del vehículo. [1, 18].

Los robots móviles poseen características que los hacen aptos para tareas particulares, son estas tareas las que determinan las características que deberá presentar el robot, entre las que se incluye: el tipo de rueda, el sistema de tracción y dirección y su forma física[1].

En general los robots distribuyen su sistema de tracción y dirección sobre los ejes de sus ruedas de acuerdo a sus necesidades y a las características del terreno, por ejemplo, la necesidad de movimientos rápidos y precisos implica la incorporación de un sistema de tracción confiable y un sistema de dirección que permita la maniobrabilidad adecuada, asimismo se determinan otros factores como como el número de ruedas necesarias y el tipo y disposición de éstas para una estructura mecánica estable [19, 20].

Los robots terrestres con ruedas se caracterizan por dos factores principales, el tipo de rueda empleada y su disposición en la estructura.

1.1.1. Tipos de rueda

Se distinguen dos tipos básicos de rueda: la rueda convencional y las ruedas "especiales", entre las que incluiremos la rueda sueca (swedish wheel), la rueda mecanum y la esférica, en ambos casos se asume que el contacto entre la rueda y el terreno se reduce a un único punto del plano. A su vez dentro de las ruedas convencionales se distinguen tres tipos (ver Figura 1.1)

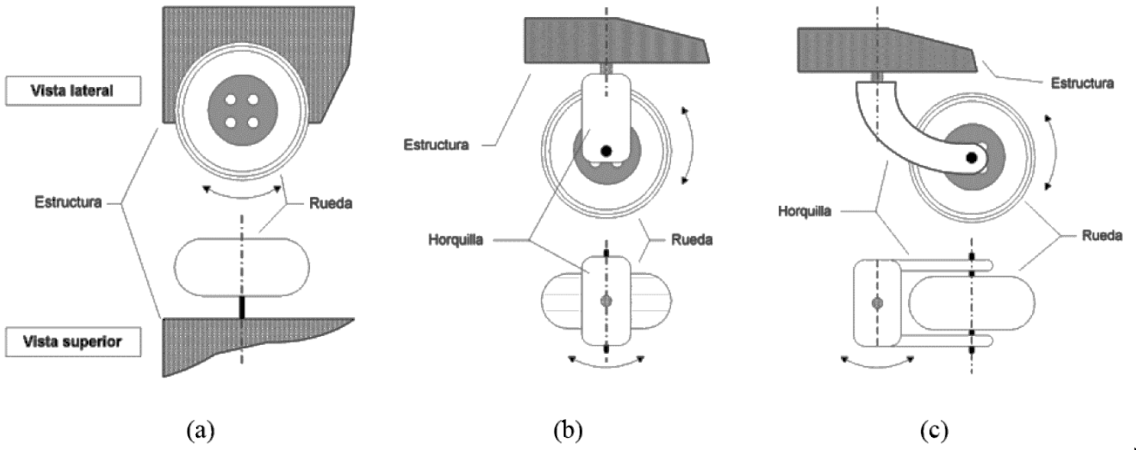

Figura 1.1: Tipos de ruedas: (a) Rueda fija. (b) Rueda orientable. (c)Rueda loca (reproducida de [1]).

- Rueda fija: el eje de la rueda se encuentra fijo a la estructura del robot, comúnmente asociado al sistema de tracción del robot [1].

- Rueda orientable centrada: Es aquella en la que el movimiento del plano de la rueda con respecto a la estructura es una rotación alrededor de un eje vertical que pasa a través del centro de la rueda. Suele cumplir funciones como rueda de dirección o como rueda de tracción-dirección.

- Rueda orientable no-centrada (rueda loca): También conocida como rueda castor (castor wheel) es una rueda orientable con respecto a la estructura, tal que la rotación del plano de la rueda es alrededor de un eje vertical el cual no pasa a través del centro de la rueda. Su principal función es la de dar estabilidad a la estructura mecánica del robot como rueda de dirección [1].

- Ruedas suecas: Estas permiten el movimiento omnidireccional aprovechando la idea de incluir en el diseño de la rueda una componente activa de movimiento, la cual es la aportación sobre el eje de la rueda, y una pasiva, que resulta de ubicar rodillos sobre la periferia de la rueda con su eje de rotación tangente a esta, cada una de estas componentes aportan tracción en diferentes direcciones.

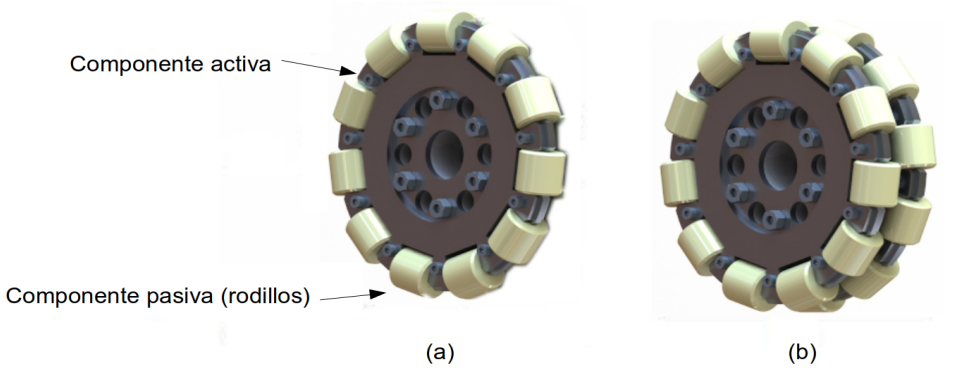

Figura 1.2: Ruedas universales: (a)Simple. (b)Doble.

Algunas de las características que podemos destacar de este tipo de ruedas son:

- Capacidad de carga limitada por el único punto de contacto entre la rueda (el punto de contacto pertenece a un rodillo) y la superficie sobre la que se desliza.
- Diseño simple.
- Radio de la rueda variable.
- Fricción baja.
- Mayor sensibilidad a la superficie respecto de otros tipos.

- Ruedas mecanum: al igual que las ruedas suecas incluyen una componente pasiva de movimiento, con la diferencia de que los rodillos poseen una rotación de cierto ángulo con respecto a la circunferencia de la rueda, usualmente 45 grados.

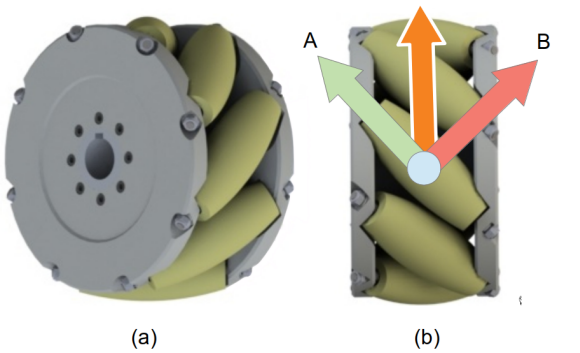

Figura 1.3: (a)Rueda omnidireccional de tipo Mecanum. (b)Descomposición de fuerzas ejercida por el motor sobre la rueda.

Debido a la disposición de los rodillos en la rueda principal, la fuerza emitida sobre la rueda, en la dirección de ésta, produce una fuerza sobre la superficie que se descompone en dos vectores de fuerza: uno perpendicular al eje del rodillo y el segundo paralelo al eje del rodillo como se ilustra en la Figura (1.3).

Por último, mencionaremos la existencia de otro tipo de rueda con forma esférica (véase Figura 1.4), Estas son impulsadas a través de motores los cuales mediante fricción aplican una fuerza sobre la esfera. La clara ventaja de este diseño es que permite movilidad en todas las direcciones, pero su principal desventaja es que se debe ejercer una gran potencia sobre los motores para lograr la fricción deseada en algunos casos.

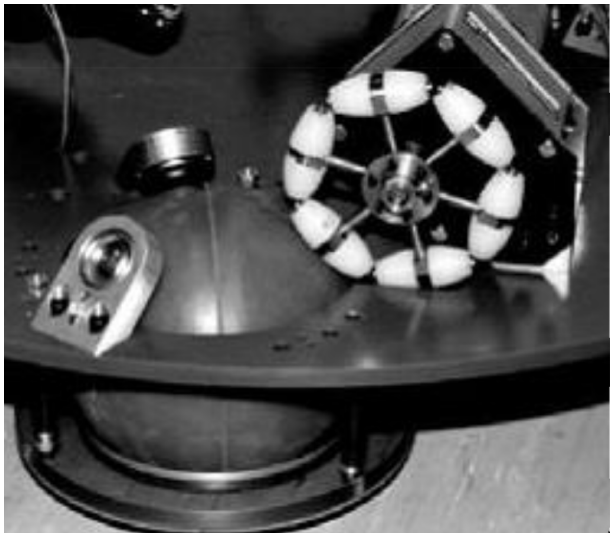

Figura 1.4: Rueda con forma esférica (reproducido de [2]).

1.1.2. Disposición de las ruedas

De acuerdo al tipo de rueda seleccionado y a las características del terreno en el que se desea que el robot se desempeñe existen diferentes configuraciones que dotarán al robot con diversas características de movilidad, de entre las cuales se hará mención de la configuración denominada uniciclo, la configuración Ackerman y dos configuraciones que emplean las ruedas especiales para lograr movimiento omnidireccional.

Configuración uniciclo

Es en general el elegido por investigadores a la hora de probar nuevas estrategias de control por tener una cinemática sencilla, está conformado por dos ruedas fijas convencionales sobre el mismo eje, controladas independientemente y una rueda loca que le proporciona estabilidad (véase la Figura 1.5).

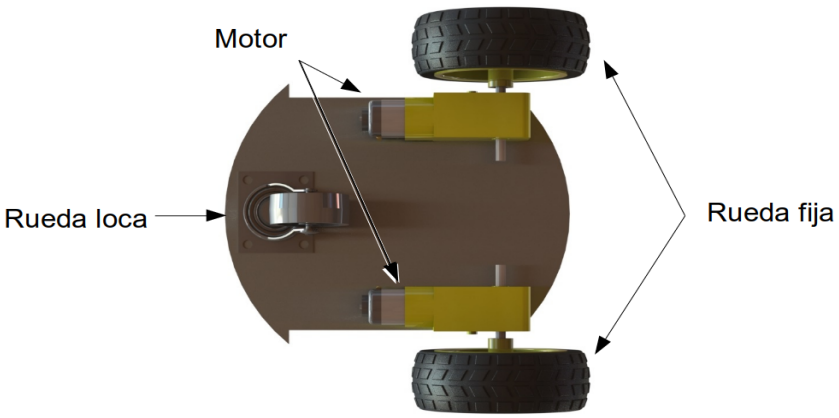

Figura 1.5: Estructura del robot uniciclo.

Configuracion Ackerman

En esta, se incorporan cuatro ruedas, de las cuales 2 son fijas y conforman el sistema de tracción en la parte posterior, y las restantes, en la parte frontal, conforman el sistema de dirección, los ejes de estas últimas se interceptan en un punto C que pertenece al eje común de las ruedas traseras, con esto, la dirección adquirida por el vehículo consta de arcos concéntricos a C donde todos los vectores de velocidad instantánea son tangentes a estos arcos [1] (véase Figura 1.6).

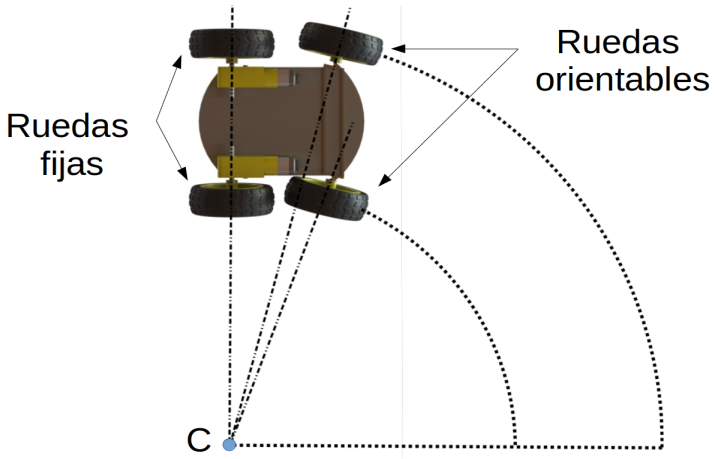

Figura 1.6: Configuración Ackerman.

Esta configuración, gracias a la disposición de sus ruedas permite al robot mayor estabilidad en comparación con la obtenida en la configuración uniciclo, ya que en esta última una mala colocación de cargas sobre la estructura puede causar una reducción de la tracción en alguna de las ruedas, lo que conduce a errores para el cálculo de trayectorias. Otra desventaja de esta configuración yace en la naturaleza de su modelo, ya que consta de dos grados de libertad y debido a la disposición de las ruedas no es posible realizar movimientos de reorientación o giros cerrados lo que complica el cálculo de trayectorias.

Configuraciones omnidireccionales

En esta sección se describirán algunas configuraciones especiales, las cuales permiten el movimiento omnidireccional aprovechando efectos mecánicos de ruedas especiales evitando la adición de actuadores extra, lo cual, desde el punto de vista de control representa una gran ventaja, ya que el modelo cinemático del robot se simplifica, y de igual manera es posible reducir la complejidad de los algoritmos de control de trayectorias, debido a que, como se verá más adelante, configuraciones como la Ackerman requieren de movimientos extra para la reorientación del robot [1].

Configuración con ruedas mecanum

Como se mencionó anteriormente, las ruedas especiales de tipo mecanum constan de dos componentes de movimiento, por medio de lo cual al realizar combinaciones de direcciones y velocidades en las cuatro ruedas dispuestas de forma independiente en el robot es posible obtener el movimiento omnidireccional deseado (véase en la Figura 1.7 [1]).

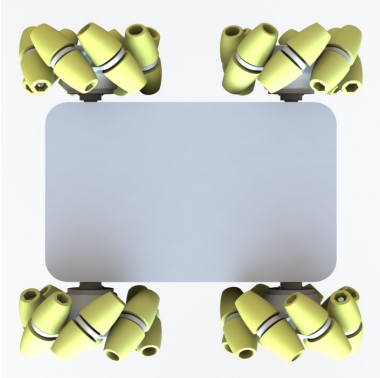

Figura 1.7: Configuración de ruedas mecanum.

Configuración (3,0)

Similar a la configuración omnidireccional con ruedas de tipo mecanum, esta configuración aprovecha el mismo tipo de componentes activas y pasivas presentes en el tipo de ruedas universales. Esta configuración consta de 3 ruedas universales dispuestas de manera uniforme al rededor de la circunferencia que conforma el perímetro de la base como se muestra en la Figura 1.8.

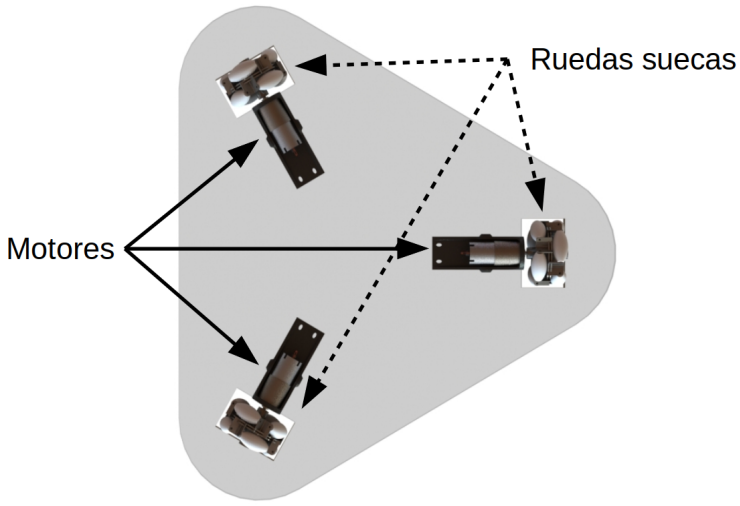

Figura 1.8: Distribución de las ruedas en un robot (3,0).

1.2. Sensores

Uno de los aspectos más importantes a la hora de implementar un algoritmo de SLAM, es el sensor a utilizar, ya que de este dependerá la calidad de la información que se adquiera para el procesamiento, uno de los sensores con mayor uso en este medio, son los denominados LIDAR, que en esencia se componen por un escáner basado en láser lo que

permite una gran resolución, llegando a tener un error inferior a 10 milímetros, del mismo modo constan con un gran campo de visión mayor al de las cámaras, pudiendo llegar a cubrir 360°, la mayor desventaja de estos equipos yace en su costo [21].

También existen implementaciones de SLAM basadas en sensores ultrasónicos, son de un costo mucho menor a un LIDAR, pero a cambio se sacrifica la precisión de las muestras.

Por otro lado, los sensores ópticos han sido ampliamente usados en esta rama, debido a su capacidad de extraer grandes cantidades de información, de la que es posible capturar características específicas. En los recientes años, gracias al crecimiento de la industria y el aumento del poder computacional estos sensores se han vuelto los más utilizados [21].

A continuación, mencionamos algunas de las configuraciones que se emplean.

- **Cámaras monoculares** Son sensores ópticos que constan de arreglos matriciales de receptores CMOS, con los que se captan las componentes rojo, verde y azul, estos tipos de sensores los podemos encontrar en todas partes ya que son los que se utilizan para las cámaras digitales.

  Para la implementación de SLAM con este tipo de sensores, se emplean algoritmos de extracción de características como lo son la transformación de características invariantes a la escala, SIFT, extracción acelerada de características robustas SURF, entre otros; su mayor ventaja es que no cuenta con un rango límite de profundidad, por otro lado su mayor desventaja consta en el error de la estimación en los mapas generados, ya que, típicamente se estima la escala, lo cual puede ser logrado a través de caracterización antes o después del mapeo, también resulta complicado mantener la consistencia de la escala y los barridos de imagen muchas veces son inevitables.

- **Cámaras estereoscópicas** Las cámaras estereoscópicas, son dispositivos, los cuales consisten de dos lentes separados por una distancia conocida, de modo que es posible calcular la distancia a los objetos y generar una nube de puntos, esta topología presenta desventajas similares a las de la cámara monocular.

- **Cámaras RGB-D** Estas últimas, son cámaras compuestas de una cámara monocular a color adicionada con un emisor y receptor de láser infrarrojo, similares al conocido sensor de Microsoft *"KINECT"*, con lo que se calcula el tiempo que tarda en volver el láser y se obtiene una medición fiel de los valores de distancias, una de las desventajas de estos sistemas radica en su rango de detección, ya que oscila en el rango de 11 cm a 10 m (dependiendo del fabricante del sensor), otra desventaja consiste en su ángulo de visión, que suele ser menor a los 85°, lo cual, comparado con escáneres láser (LIDAR), con capacidad de 360° hace que la creación de cuadros sea más desafiante. Entre sus ventajas se encuentra el hecho de que la presencia o no

de luz no afecta las mediciones realizadas por el láser, por lo que no requiere ajustes para contrarrestar el efecto de la luz en el entorno, por lo que la interpretación de texturas es mejor que con una cámara monocular.

También se considera la implementación de sistemas de captura dispuestos en la estructura del área de trabajo como lo es OPTITRACK.

Optitrack

Optitrack es un sistema óptico, en el espectro infrarrojo, que se compone de una serie de cámaras las cuales se disponen en el perímetro del área deseada1.2, enfocando al centro de esta. Este sistema, es capaz de detectar la presencia en tiempo real de marcadores esféricos reflejantes y de este modo, puede calcular la posición de objetos dentro del área, representándolo posteriormente de forma grafica en la interfaz Motive.

Figura 1.9: Ilustración de la distribución de las cámaras Optitrack y el agente.

1.3. ROS (*"Robot Operating System"*)

La robótica ha experimentado un cambio transformador en la última década. El surgimiento de nuevos marcos de código abierto como ROS y *"MoveIt!"* ha hecho que la robótica sea más accesible a nuevos usuarios, tanto en la investigación como en las aplicaciones para el consumidor. En particular, ROS ha revolucionado la comunidad de desarrolladores, proporcionándole un conjunto de herramientas, infraestructura y las mejores prácticas para construir nuevas aplicaciones y robots (como el Baxter robot de investigación). Un pilar clave del esfuerzo de ROS es la noción de no reinventar el rueda proporcionando bibliotecas fáciles de usar para diferentes capacidades como la navegación, manipulación, control (y más) [22].

ROS es un *"framework"* de software de robótica libre y de código abierto que se utiliza en casos tanto comerciales como en aplicaciones de investigación. El *"framework"* de ROS

provee las siguientes capacidades para la programación de robots:

- Interfaz de paso de mensajes entre procesos: ROS proporciona una interfaz de paso de mensajes para comunicarse entre dos programas o procesos. Por ejemplo, una cámara procesa una imagen y encuentra coordenadas en ella, y luego estas coordenadas son enviadas a un proceso de seguimiento. El proceso del rastreador hace el seguimiento de la imagen mediante el uso de motores. Como se ha mencionado, esta es una de las características necesarias para programar un robot. También se llama comunicación entre procesos porque estos pueden comunicarse entre sí.

- Características similares a las de los sistemas operativos. Como su nombre indica, ROS no es un sistema operativo real. Es un meta-sistema operativo que proporciona algunas funcionalidades de sistema operativo. Estas funcionalidades incluyen multihilo, control de dispositivos de bajo nivel, administración de paquetes y abstracción de hardware. La capa de abstracción de hardware permite a los programadores programar un dispositivo. La ventaja es que se puede escribir código para un sensor que funciona de la misma manera con diferentes proveedores. Por lo tanto, no necesitamos reescribir el código cuando usamos un nuevo sensor. La gestión de paquetes ayuda a los usuarios a organizar el software en unidades llamadas paquetes. Cada paquete tiene código fuente, archivos de configuración o archivos de datos para una tarea específica. Estos paquetes pueden ser distribuidos e instalados en otros ordenadores.

- Soporte y herramientas de lenguaje de programación de alto nivel. La ventaja del ROS es que soporta los lenguajes de programación más populares utilizados en la programación de robots, incluyendo C++, Python y Lisp. Hay soporte experimental para lenguajes como C #, Java, Node.js, entre otros.[2]. ROS proporciona bibliotecas de cliente para estos lenguajes, lo que significa que el programador puede obtener las funcionalidades ROS en los lenguajes mencionados. Por ejemplo, si un usuario quiere implementar una aplicación de Android que utilice la funcionalidad de ROS, Rosjava puede utilizar la biblioteca del cliente. ROS también proporciona herramientas para construir aplicaciones de robótica. Con estas herramientas, nosotros podemos construir muchos paquetes con un solo comando. Esta flexibilidad ayuda a los programadores a pasar menos tiempo creando sistemas de construcción para sus aplicaciones.

- Disponibilidad de bibliotecas de terceros. El marco ROS se integra con las bibliotecas de terceros más populares; por ejemplo, OpenCV [3] está integrado para la visión robótica, y el PCL [4] está integrado para la percepción de los robots en 3D. Estas

[2]http://wiki.ros.org/Client%20Libraries
[3]https://opencv.org
[4]http://pointclouds.org

bibliotecas hacen de ROS una herramienta más fuerte, y el programador puede construir aplicaciones poderosas a partir de ellas.

- Algoritmos listos para usar. Esta es una característica útil. ROS ha implementado populares algoritmos de robótica como PID [5]; SLAM (Simultáneo Localización y cartografía) [6]; y planificadores de caminos como A*, Dijkstra [7], y AMCL (*Adaptive Monte Carlo Localization*) [8]. La lista de implementaciones de algoritmos en ROS continúa. Los algoritmos de la plataforma reducen el tiempo de desarrollo para la creación de un prototipo de robot.

- Facilidad en la creación de prototipos. Una de las ventajas de ROS son los algoritmos "*off-the-shelf*". Junto con eso, ROS tiene paquetes que pueden ser fácilmente reutilizados con cualquier robot; por ejemplo, se podría fácilmente hacer un prototipo de un robot móvil, personalizando un paquete de robot móvil existente disponible en el depósito del ROS. Podemos reutilizar fácilmente el repositorio ROS porque la mayoría de los paquetes son de código abierto y re utilizables para fines comerciales y con fines de investigación. Así que, esto puede reducir el tiempo de desarrollo en el software de los robots.

- Apoyo al ecosistema/comunidad. La principal razón de la popularidad y el desarrollo de ROS es el apoyo de la comunidad. Los desarrolladores de ROS están en todo el mundo. Ellos desarrollan y mantienen activamente los paquetes ROS. El gran apoyo de la comunidad incluye a los desarrolladores que hacen preguntas relacionadas con ROS. "*ROS Answers*" es una plataforma para consultas relacionadas con el ROS [9]. "*ROS Discourse*" es un foro en línea en el que los usuarios de ROS discuten diversos temas y publican noticias relacionadas con ROS [10].

- Amplias herramientas y simuladores. ROS está construido con muchas herramientas de línea de comandos y GUI para depurar, visualizar y simular aplicaciones de robótica. Estas herramientas son muy útiles para trabajar con un robot. Por ejemplo, la herramienta Rviz [11] se utiliza para la visualización con cámaras, escáneres láser, inercia unidades de medida, y así sucesivamente. Para trabajar con simulaciones de robots, existen simuladores como Gazebo [12].

[5] http://wiki.ros.org/pid
[6] http://wiki.ros.org/gmapping
[7] http://wiki.ros.org/global_planner
[8] http://wiki.ros.org/amcl
[9] https://answers.ros.org/questions/
[10] https://discourse.ros.org
[11] http://wiki.ros.org/rviz
[12] http://gazebosim.org

Arquitectura y conceptos de ROS

ROS es una buena opción para comunicación entre procesos, esto resulta bastante útil, ya que un robot puede tener múltiples sensores y actuadores, que en la realidad no resulta práctica la programación conjunta de estos. Este es el panorama en donde ROS resulta de mayor utilidad.

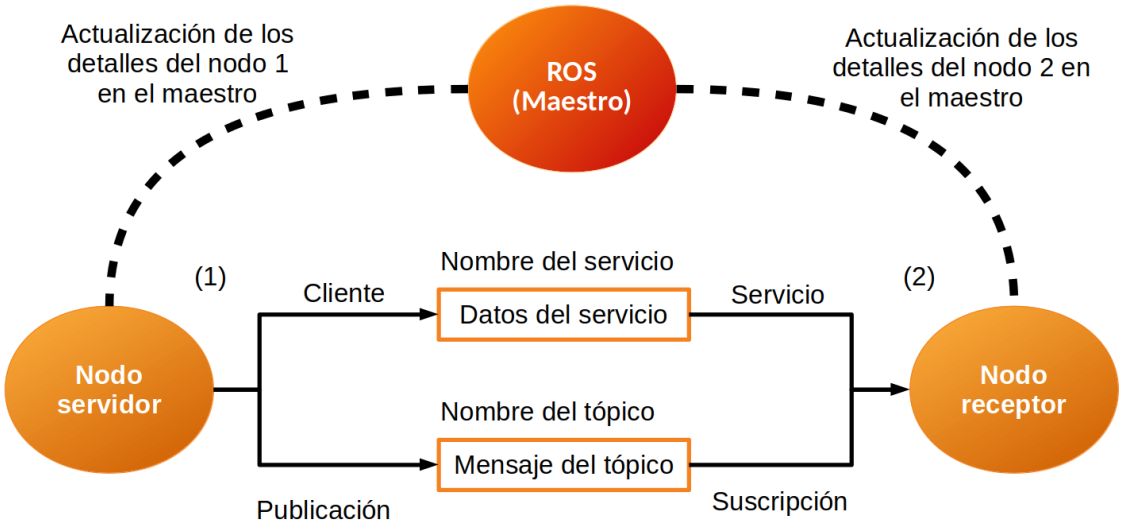

Figura 1.10: Diagrama de bloques de la comunicación de ROS.

A continuación se listan los términos asociados a los conceptos de computación de ROS.

- Nodos ROS: Proceso que utiliza las API de ROS para realizar cálculos.
- Maestro ROS: Un programa intermedio que conecta Nodos ROS.
- Servidor de parámetros ROS: Un programa que normalmente se ejecuta junto con el maestro de ROS. El usuario puede almacenar varios parámetros o valores en este servidor y todos los nodos pueden acceder a ellos. El usuario también puede establecer la privacidad de los parámetros. Si es un parámetro público, todos los nodos tienen acceso; si es privado, sólo un nodo específico puede acceder el parámetro.
- Tópico ROS: Buses nombrados en los que los nodos ROS pueden enviar un mensaje. Un nodo puede publicar o suscribir cualquier número de temas.
- Mensaje ROS: Los mensajes son básicamente enviados a través de los tópicos. Existen mensajes basados en tipos de datos primitivos, y los usuarios pueden escribir sus propios mensajes.

- Servicio ROS:Ya hemos visto los tópicos de ROS, que tiene un mecanismo de publicación y suscripción. El Servicio ROS tiene un mecanismo de Solicitud/Respuesta. Una llamada de servicio es una función, que puede llamar cada vez que un nodo de cliente envía una solicitud. El nodo que crea una llamada de servicio se llama nodo servidor y el que llama al servicio se llama nodo cliente.
- Bolsas ROS: Un método útil para guardar y reproducir tópicos de ROS. También es útil para registrar los datos de un robot para procesarlo más tarde.

1.4. Gazebo

Gazebo es un simulador de ambientes robóticos que ofrece la posibilidad de simular sistemas complejos de robots con sensores de una manera precisa y eficiente, emulando las variables físicas del mundo real, permitiendo probar algoritmos de control y diseños propios de robots en escenarios realistas [23]. Tiene una arquitectura cliente/servidor que se basa en las herramientas de ROS de suscripción y publicación de mensajes en los tópicos para generar los procesos de comunicación entre los diferentes componentes en el sistema. Algunas de las características principales de Gazebo son:

- Simulación dinámica: Permite acceder a motores de físicas de alto rendimiento.
- Gráficos 3D avanzados: Considera aspectos de iluminación, sombras y texturas de alta calidad.
- Sensores y ruido: Genera sensores de datos con la posibilidad de agregar y ajustar diferentes patrones de ruido.
- Plugins: Desarrolla plugins personalizados para robots, sensores y controles.
- Modelos de robots: Ofrece una variedad predeterminada de diferentes modelos de robots existentes.
- Herramienta de línea de comandos: Facilita la inspección de simulación y control.

1.5. SLAM

En el año de 1986, durante un congreso del IEEE [24], se concluyó que la generación de mapas consistentes, era un problema fundamental que necesitaba ser estudiado, en los años subsecuentes se publicaron diversos artículos en donde se estableció la relación entre los objetos de referencia, llamados "*landmarks*", y éstas a su vez requerían de una correlación entre las estimaciones de localización en las distintas imágenes, del mismo modo, debían crecer con sucesivas correlaciones.

En el año de 1995 en el *International Symposium on Robotics Research* se presentó por primera vez la estructura del problema, y se acuñó el término SLAM.

Formulación y estructura del problema del SLAM

El SLAM es un proceso por el cual un robot puede construir un mapa del entorno y al mismo tiempo utilizar este mapa para deducir su localización. En SLAM, tanto la trayectoria de la plataforma como la ubicación estimada de las "*landmarks*" son estimadas en linea sin necesidad de conocimiento a priori de la ubicación.

Preliminares

Considere un robot desplazándose en un ambiente, tomando observaciones relativas de un número de "*landmarks*" desconocidas utilizando sensores ubicados en el robot. En el instante k las siguientes variables son definidas:

- $\mathbf{x}_k$: Vector de estados que describe la ubicación y orientación del vehículo.
- $\mathbf{u}_k$: Vector de control, aplicado en el tiempo $k-1$ para posicionar el vehículo en el estado x_k en el tiempo k.
- $\mathbf{m}_i$: Vector que describe la ubicación de la i-ésima "*landmark*", de la cual, su ubicación real se asume como invariante en el tiempo.
- $\mathbf{z}_{i,k}$: Observación tomada desde el vehículo de la i-ésima "*landmark*" en el tiempo k. Cuando hay múltiples "*landmarks*" observadas en cualquier instante de tiempo o cuando la "*landmark*" especificada no es relevante para la discusión, la observación sera escrita simplemente como $\mathbf{z}_k$.

Adicionalmente, los siguientes conjuntos son definidos:

- $\mathbf{X}_{0:k} = \{\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_k\} = \{\mathbf{X}_{0:k-1}, \mathbf{x}_k\}$: El historial de las ubicaciones del vehículo.
- $\mathbf{U}_{0:k} = \{\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_k\} = \{\mathbf{U}_{0:k-1}, \mathbf{u}_k\}$: El historial de las entradas de control.
- $\mathbf{m} = \{\mathbf{m}_1, \mathbf{m}_2, \cdots, \mathbf{m}_n\}$: El conjunto de todas las "*landmarks*".
- $\mathbf{Z}_{0:k} = \{\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_k\} = \{\mathbf{Z}_{0:k-1}, \mathbf{z}_k\}$: El conjunto de todas las "*landmarks*" observadas.

SLAM probabilístico

En términos probabilistas, el problema de la localización y mapeo simultaneo (SLAM) requiere que la distribución de probabilidad

$$P(\mathbf{x}_k, \mathbf{m}|\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0) \tag{1.1}$$

Sea calculada para cada instante k, esta distribución de probabilidad describe la densidad de la unión de la ubicación del las "*landmarks*" y el estado del vehículo (en el tiempo k) dadas las observaciones guardadas y las entradas de control hasta el instante k, incluyendo el estado inicial del vehículo. En general, una solución general del problema del SLAM es deseable. Comenzando con una distribución $P(\mathbf{x}_k, \mathbf{m}|\mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k-1}, \mathbf{x}_0)$ en el tiempo $k-1$, siguiendo un control $\mathbf{u}_k$ y una observación $\mathbf{z}_k$, se calcula en conjunto posterior utilizando el teorema de Bayes. Este cálculo requiere que un modelo de cambio de estados y un modelo de observación sean definidos describiendo el efecto de la señal de control y de la observación respectivamente.

El modelo de observación describe la probabilidad de hacer una observación $\mathbf{z}_k$ cuando la ubicación del vehículo y de la "*landmark*" son conocidas y se describe de manera general en la forma

$$P(\mathbf{z}_k|\mathbf{x}_k, m) \tag{1.2}$$

Es razonable asumir que una vez sean definidos la ubicación y el mapa, las observaciones son condicionalmente independientes dado el mapa y el estado actual del vehículo.

El *modelo de movimiento* para el vehículo puede ser descrito en términos de distribuciones de probabilidad para la transición de estados de la forma

$$P(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{u}_k) \tag{1.3}$$

Esta es la transición de estados, que se asume como un proceso de Markov, en el que el siguiente estado $\mathbf{x}_k$ depende solamente del estado inmediato anterior $\mathbf{x}_{k-1}$ y de la señal de control aplicada $\mathbf{u}_k$ y es independiente de las observaciones y del mapa.

Clasificaciones de SLAM

A continuación se incluyen algunas posibles clasificaciones del problema del SLAM.

- **Offline vs. Online.** En el caso del SLAM online se procesa la información en el mismo robot mientras este navega en el entorno. Por otro lado, en el offline se realiza SLAM sobre un conjunto de datos que previamente fueron recuperados con algún robot, tanto de la medida de sus sensores como las medidas de odometría (movimiento) [4, 25].

- **Topológico vs. Métrico.** Algunas técnicas de armado de mapas solamente mantienen la descripción de algunas características del entorno, que caracteriza la relación entre lugares. Estos métodos son conocidos como topológicos. Un mapa topológico se define por un conjunto de lugares diferentes y de otro conjunto que caracterizan las relaciones entre estos. Por otro lado, los métodos métricos proveen información métrica entre los lugares. En los últimos años los métodos topológicos han pasado de moda a pesar de la amplia evidencia de que los humanos utilizan a menudo información topológica[4, 25].

- **Activo vs. Pasivo.** En los algoritmos de SLAM pasivos, es otra entidad quien se encarga de controlar el robot, mientras que el algoritmo de SLAM es puramente observador. La gran mayoría de los algoritmos de SLAM son de este tipo. En el caso de los algoritmos de SLAM activo, el robot explora de forma activa el entorno en busca de conseguir un mapa más preciso en el menor tiempo posible. Existen técnicas híbridas donde el algoritmo de SLAM solo controla la dirección de los sensores y otra entidad se encarga de la dirección del movimiento del robot[4, 25].

- **Estático vs. Dinámico.** En el caso del SLAM estático se asume que el entorno no cambia con el tiempo, a diferencia de los métodos dinámicos que sí lo hacen. La gran mayoría de la literatura asume entornos estáticos[4, 25].

- **Volumétrico vs. Basado en marcas.** En SLAM volumétrico, el mapa es muestreado a una resolución que permite una reconstrucción fotográfica del entorno. En este caso el costo computacional es alto. Por otro lado, en el SLAM basado en marcas se extraen características de las medidas de los sensores a forma de armar el mapa en base a marcas dispersas. Las técnicas que utilizan SLAM basado en marcas suelen ser más eficientes ya que se descarta información de los sensores[4, 25].

- **SLAM con un solo robot vs. Multi-robot.** La gran parte de los problemas están definidos para un solo robot, sin embargo, recientemente el problema de trabajar con más de un robot ha ganado mucha popularidad[4, 25]. existe una amplia gama de problemas que se relacionan con la implementación de SLAM con múltiples robots. En algunos casos los robots son capaces de observarse entre ellos. También se distinguen por el tipo de comunicación que utilizan. Los más realistas permiten que solamente los robots que están más cerca se puedan comunicar[4, 25].

- **Manejo de mucha incertidumbre vs. poca incertidumbre.** Estos algoritmos se distinguen por la cantidad de incertidumbre que pueden manejar. Los más simples solo pueden manejar poca incertidumbre en la localización. Son útiles para los casos en los cuales un camino no se intersecta a sí mismo. En cambio para los mapas los cuales tienen lugares que se pueden alcanzar de distintas maneras, es necesario

que el robot pueda manejar una amplia incertidumbre en su posición. La incertidumbre puede ser disminuida si el robot puede sensar información sobre su posición de forma absoluta. Un ejemplo puede ser mediante el uso de GPS[4, 25].

De acuerdo al enfoque utilizado para la solución del problema del SLAM se reconocen 3 vertientes principales, la basada en filtros de Kalman, basada en filtros de partículas y basada en grafos, a continuación mencionaremos los aspectos principales de estas.

Filtros de Kalman

Los filtros de Kalman [26] son una solución al problema de Online SLAM que propone actualizar el estado del sistema conforme se obtiene información de odometría y sensado.

La principal característica de este método es el uso de distribuciones gaussianas como estimador. Esta solución se basa en el hecho que al multiplicar una variable de distribución gaussiana por un número (o matriz en el caso multivariable) se obtiene otra variable de distribución gaussiana. Luego, el algoritmo propone que la evolución del sistema puede modelarse mediante dos ecuaciones lineales[25].

Modelado

El modelado de la evolución del sistema se realiza mediante dos ecuaciones, la ecuación de transición y la ecuación de observación. A continuación se describen ambas ecuaciones.

Ecuación de transición: Esta actualiza el estado del sistema estimado, esto en función de la evolución independiente de factores externos y de la información de odometría:

$$x_t = F \cdot x_{t-1} + B \cdot u_t + r_t \tag{1.4}$$

donde:

- x_t representa al estado del sistema a estimar.
- F Es una matriz que representa el cambio inherente al sistema, ajeno a factores externos. Este cambio podría representar, por ejemplo, la evolución de la posición de un objeto que viaja a velocidad constante.
- u_t representa la información de odometría.

- B consta de una matriz que representa la transformación de la información de odometría en los cambios producidos en el sistema.
- r_t es un factor de ruido que representa la naturaleza estocástica de los otros términos de la ecuación. Este ruido es también gaussiano.

Ecuación de observación: Propone una relación entre el estado estimado del sistema y la observación a realizar por el robot en un tiempo t. La ecuación puede expresarse como:

$$z_t = H \cdot x_t + \omega_t \tag{1.5}$$

Donde:

- z_t Corresponde a la información de sensado.
- H Consta de una matriz que transforma el estado del sistema en la observación a realizar. Por ejemplo esta matriz podría transformar las coordenadas de una marca en las coordenadas del campo visual del robot (proyección).
- w_t Representa el ruido en el otro término de la ecuación. Este término podría corresponderse con el ruido introducido por los sensores del robot, que no son totalmente precisos [25].

Cálculo del estado del sistema: Bajo las condiciones mencionadas, la solución que minimiza el error esperado (mínimos cuadrados) del estado del sistema puede calcularse de forma cerrada utilizando un filtro de Kalman [27]. Esta actualización se realiza mediante un algoritmo que realiza solo operaciones de álgebra lineal para actualizar los estimadores de la posición del sistema (media μ y varianza σ^2) [25].

Linealidad: El algoritmo de Kalman propone la evolución de un sistema, modelado por distribuciones gaussianas, regido por dos ecuaciones lineales. La linealidad de estas ecuaciones representa una restricción importante. Supongamos que nuestras observaciones constan del sensado de la distancia a una determinada marca. La función que toma como entrada las coordenadas del robot y la marca y retorna la distancia no es una función lineal (norma euclidiana). Por lo tanto, al utilizar este sistema se cometerían sistemáticamente errores por utilizar una función lineal (la más aproximada posible) para la representación de la función de distancia [25]. Además, la utilización de una distribución gaussiana como estimador del estado del sistema implica que solo es posible mantener un único máximo local en la creencia del estado de este sistema.

Filtro de Kalman extendido

Para eliminar la restricción de modelos de sensado y transición lineales, se diseñaron extensiones donde se reemplazan los modelos de sensado y odometría para contemplar funciones no lineales. Las ecuaciones 1.5 y 1.4 quedan:

$$\begin{aligned} x_t &= f(x_{t-1}, u_t) r_t \\ z_t &= h(x_t) + \omega_t \end{aligned} \tag{1.6}$$

Donde f y h son los modelos de transición y sensado. Esta modificación permite la utilización de filtros de Kalman con modelos no lineales. Se logra utilizando un algoritmo similar al original, pero linealizando los modelos de datos utilizando series de Taylor en cada iteración. Sin embargo, el nuevo algoritmo no corresponde a la solución cerrada del problema, es decir que la solución no es exacta y el algoritmo puede diverger por problemas vinculados a la linealización de los modelos.

Filtros de partículas

Los filtros de partículas intentan aproximar la distribución de probabilidad del estado del sistema x_t utilizando métodos de Montecarlo. Para esto, mantienen un conjunto de partículas que son N muestras (*samples*) de la distribución a actualizar. Al igual que el filtro de Kalman, actualizan esta distribución a medida que se encuentra disponible nueva información de sensado u odometría [25].

Actualización de la distribución de probabilidad

La dinámica de actualización de la función de densidad del filtro de partículas es similar al del filtro de Kalman. Esta actualización consta de dos pasos, el de predicción y el de actualización propiamente dicho.

Paso de predicción: El paso de predicción busca modificar la función de densidad para reflejar un movimiento realizado por el robot. Para esto, se modifica la posición de cada una de las partículas según:

$$x_t = f(x_{t-1}, u_t) \tag{1.7}$$

Donde f representa el modelo de movimiento a utilizar. En la Figura 1.11 se puede ver un conjunto de partículas inicial (abajo a la izquierda) al que se le aplican sucesivamente los pasos de predicción. Esta imagen ilustra como este paso del algoritmo aumenta la dispersión de las partículas, y por consiguiente la varianza de la

distribución. Esto se debe a que el modelo de sensado incluye ruido para modelar la naturaleza estocástica del movimiento del robot. Este ruido se va acumulando en cada paso de predicción y aumenta la incertidumbre de la estimación [25].

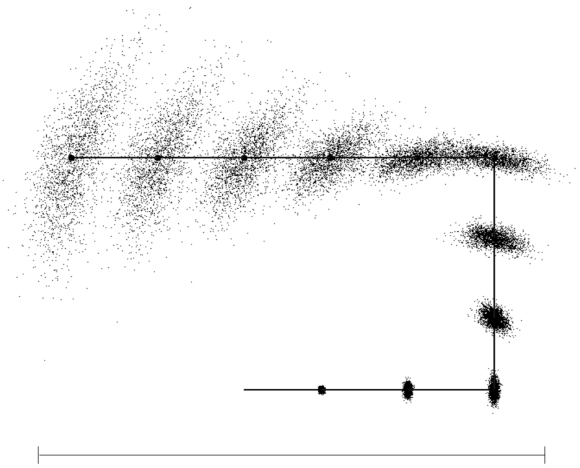

Figura 1.11: Aplicación sucesiva del paso de predicción(reproducido de [3]).

Paso de actualización: Este paso busca ajustar la función de densidad a la última información de sensado recibida. Para esto se le asigna a cada partícula un peso w_i proporcional a la verosimilitud de la información de sensado acorde al estado actual del sistema. Es decir:

$$w_i \approx p(z_t|x_t) \tag{1.8}$$

Este peso indica, en resumen, cuán verosímil es el modelo representado por la partícula en función de la última información de sensado. Luego, se realiza un proceso denominado remuestreo (*resampling*) en el que se extraen con reposición N partículas del conjunto de partículas actuales. La probabilidad de seleccionar una partícula es proporcional a su peso w_i. En general, el paso de actualización disminuye la incertidumbre de la estimación. En otras palabras, disminuye la dispersión de las partículas.

Grafos

A diferencia de las soluciones de SLAM basados en filtros, el SLAM de Grafos (*Graph SLAM*) resuelve el problema de Full SLAM.

Graph SLAM es utilizado para la resolución del SLAM basado en marcas. Para esto, la solución modela el problema de SLAM como un grafo, donde los nodos

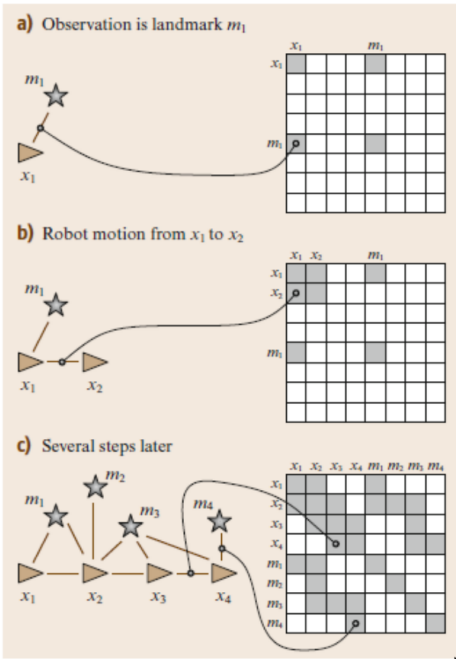

Figura 1.12: Modelo del grafo de Graph SLAM. La matriz de la izquierda corresponde a la matriz de conectividad del grafo. Figura extraída de [4].

representan posiciones del robot x^i y marcas m^i (el superíndice se utiliza para no confundir con el subíndice que indica instante en el tiempo). Los datos obtenidos de odometría y sensado se procesan para transformarlos en aristas que relacionan estos nodos. En la Figura 1.12 se ilustra este concepto [25].

Esta formulación puede verse como un problema de optimización, donde se debe optimizar la posición de las marcas (mapa) y las posiciones sucesivas del robot (trayectoria) respetando lo mejor posible un conjunto de restricciones suaves 3 representadas por las aristas. Estas restricciones se expresan en distancias, inferidas de las observaciones de las marcas y de los datos de odometría.

Este modelo se compara con el modelo de masas y resortes (*spring-mass model*), donde un conjunto de objetos con masa se encuentran interconectados entré si por medio de resortes (ver Figura 1.12). Este sistema converge a la distribución de posiciones de los objetos para minimizar la energía total contenida en cada resorte.

Es importante notar que *Graph SLAM* pospone los cálculos (*lazy algorithm*) a modo de procesar una gran cantidad de datos (o todos), a la vez. Para esto, se divide la solución en dos partes o módulos, el *front-end* y el *back-end*. El *front-end* se encarga de construir el grafo de restricciones a partir de la información de sensado y odometría. El *back-end* realiza la optimización de este grafo de modo de obtener la solución más verosímil a los datos disponibles [25].

La resolución del problema por parte del *back-end* resulta computacionalmente costosa, cuando se lo compara con un algoritmo de Online SLAM. Esto se debe a que

el algoritmo debe contemplar un mayor volumen de información, debido a que todas las conexiones derivadas de las observaciones deben ser tomadas en cuenta. En Online SLAM, en cambio, la información pasada es marginalizada en el estimado de la posición actual x_i.

Capítulo 2

Robot omnidireccional

En el presente capítulo se describe el modelo de motores así como del robot implementado, además se plantea el uso de una metodología basada en control colaborativo, lo que posteriormente facilitará el seguimiento de trayectorias por medio del uso de un agente virtual, también se introducen las metodologías para evitar colisiones con obstáculos conocidos, así como entre otros agentes.

2.1. Modelado de los motores

El modelo para hacer girar la llanta se determina por medio del sistema eléctrico del motor como se muestra en la Figura 2.1 en donde V es el voltaje aplicado, R es la resistencia de armadura, L la inductancia de armadura, i es la corriente de armadura, V_c es la fuerza electromotriz, J_m es la masa del motor, θ_m es el ángulo de giro antes de los engranes, N es la relación entre los engranes y θ es el giro de las llantas.

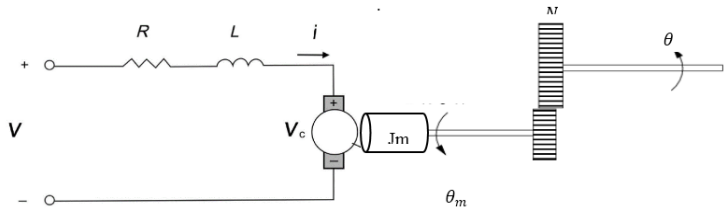

Figura 2.1: Modelo eléctrico y mecánico de los motores de las ruedas.

Donde las ecuaciones resultantes del modelo son:

$$V = R * i + L * \frac{di}{dt} + V_c \tag{2.1}$$

$$J_m * \ddot{\theta}_m = ka * i - f(\dot{\theta}_m) - \frac{\tau}{N} \tag{2.2}$$

Donde ka es una constante, τ es el par generado después de los engranes y $f(\dot{\theta}_m)$ es la fricción en función de la velocidad angular antes de los engranes. Que, sustituyendo en la ecuación $V_c = k_b * \dot{\theta}_m \quad y \quad \theta = n * \theta_m$ resulta el modelo:

$$J_m * \ddot{\theta} = \frac{k_a * V}{R * N} - \frac{k_a k_b \dot{\theta}}{R} - \frac{f(\dot{\theta}_m)}{N} - \frac{\tau}{N^2} \tag{2.3}$$

Para simplificar el análisis fundamental del sistema se igualan las fricciones $-\frac{f(\dot{\theta}_m)}{N} - \frac{\tau}{N^2} = 0$ de tal forma que resulta la expresión reducida al hacer estos términos cero y al dividir la expresión por J_m se tiene:

$$\ddot{\theta} = \frac{k_a * V}{R * N * J_m} - \frac{k_a k_b \dot{\theta}}{R * J_m} \tag{2.4}$$

Ahora, se engloban las constantes de forma que $b_m = \frac{k_a}{R*N*J_m}$ y $a_m = \frac{k_a k_b}{R*J_m}$ además si $\dot{\theta}$ es la velocidad angular entonces se puede considerar que $\omega = \dot{\theta}$ por lo que $\dot{\omega} = \ddot{\theta}$ entonces la ecuación resultante es:

$$\dot{\omega} = -a_m * w + b_m * v \tag{2.5}$$

Donde al obtener la transformada de Laplace inversa se obtiene:

$$\Omega(S) * S = -a_m * \Omega(S) + b_m V(S) \tag{2.6}$$

De tal manera que la función de transferencia resultante es:

$$H(S) = \frac{\Omega(s)}{V(S)} = \frac{b_m}{S + a_m} \tag{2.7}$$

Para determinar los valores b_m y a_m se procede a aplicar un voltaje escalón al motor de tal manera que se pueda observar una curva de respuesta en donde los valores observados estarán relacionados directamente con b_m y a_m como se muestra en la Figura 2.2 donde:

$$\frac{b_m A}{a_m} = 0.63 \tag{2.8}$$

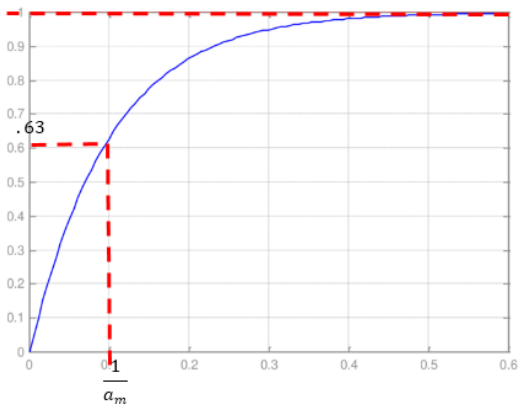

Figura 2.2: Relación curva de respuesta con parámetros de función de transferencia.

2.2. Sintonización de los controladores PI

Para realizar la sintonización de los controladores PI implementados, se siguió la metodología de curva de respuesta al escalón, el cual se describe a continuación:

- Primero con la planta a lazo abierto, llevar a la planta a un punto de operación normal y_0 para una entrada constante U_0.
- Segundo. En el instante inicial t_0, aplicar un cambio en la entrada escalón, desde U_0 a U_1 (esto en un rango de 10 al 20 % de rango completo).
- Registrar la salida hasta que se estabilice en el nuevo punto de operación y_1.

La curva obtenida se denomina curva de reacción, la cual se muestra en la Figura 2.3.

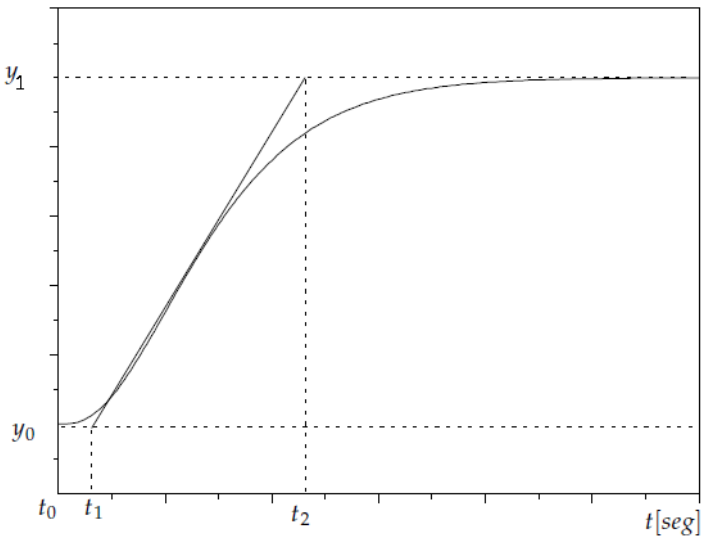

Figura 2.3: Curva de reacción típica.

Una vez obtenida la curva de reacción, se traza una recta tangente al punto donde la señal alcanza el 63.3 % de la amplitud, con *esto* se obtienen dos puntos; el primero, el cruce de esta recta con el valor inicial y_0, el segundo, representado con cruce del valor de estabilización y_1.

Con los valores antes obtenidos se pueden calcular:

$$k_0 = \frac{y_1 - y_0}{U_1 - U_0} \quad \tau_0 = t_1 - t_0 \quad v_0 = t_2 - t_1 \tag{2.9}$$

Finalmente aplicamos los valores de la Figura 2.4 para obtener los parámetros, tanto proporcional como integral.

K_p	T_i
$\frac{0.9v_0}{K_0\tau_0}$	$3\tau_0$

Figura 2.4: Parámetros proporcional e integral.

Aplicando la metodología anterior se obtuvo la siguiente curva de reacción, a partir de la cual se obtienen los valores $y_0, y_1, U_0, U_1, t_0, \quad t_1 \quad y \quad t_2$.

2.3. Modelo cinemático del robot (3,0)

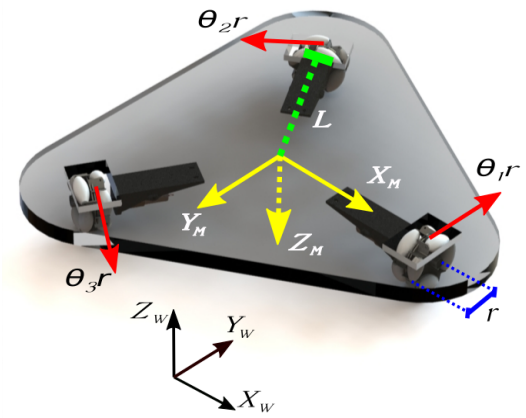

Figura 2.5: Ilustración de marcos de referencia.

El vector de velocidad lineal V_{ri} para una de las ruedas, está dado por:

$$\begin{aligned} V_{ri} &= \tfrac{d_{si}}{dt} \\ d_{si} &= r d\theta_i \end{aligned} \tag{2.10}$$

Donde r es el radio de la rueda y θ el radio de giro, además:

$$V_i = rd\dot{\theta}_i = r\omega_i \tag{2.11}$$

Con respecto al marco de referencia [X_m, Y_m] la velocidad lineal en la tercera rueda está dada por:

$$r\dot{\theta}_3 = L\dot{\phi} + \dot{x_m} \tag{2.12}$$

Donde L es la distancia del marco de referencia [X_m, Y_m] al centro de una de las ruedas. Por otra parte, para la primera rueda se tiene que:

$$r\dot{\theta}_1 = L\dot{\phi} + V_{L1} \tag{2.13}$$

Donde V_{L1} es la contribución de la suma de la magnitud de las proyecciones de las velocidades en x_m y y_m en esa dirección. Sea δ el ángulo entre los vectores $r\dot{\theta_L}i = 1, 2, 3$ y el marco de referencia [X_m, Y_m], se obtiene:

$$\sin(\delta) = \frac{-\dot{x_m}}{V_{L1}} \quad y \quad \cos(\delta) = \frac{\dot{y_m}}{V_{L1}} \tag{2.14}$$

Esto es $-\dot{x_m} = V_{L1}\sin(\delta)$ y $\dot{y_m} = V_{L1}\cos(\delta)$

$$\begin{aligned} -\sin(\delta)\dot{x_m} + \dot{y_m}\cos(\delta) &= V_{L1}\sin^2(\delta) + V_{L1}\cos^2(\delta) \\ V_{L1} &= -\sin(\delta)\dot{x_m} + \cos(\delta)\dot{y_m} \end{aligned} \tag{2.15}$$

Entonces la ecuación (2.13) puede reescribirse como,

$$r\dot{\theta}_1 = L\dot{\phi} - \sin(\delta)\dot{x_m} + \cos(\delta)\dot{y_m} \tag{2.16}$$

para la rueda dos se tiene

$$\begin{aligned} r\dot{\theta}_2 &= L\dot{\phi} + V_{L2} \\ V_{L2} &= -\sin(\delta)\dot{x_m} + \cos(\delta)\dot{y_m} \end{aligned} \tag{2.17}$$

Por lo tanto la ecuación se reescribe como

$$r_r\dot{\theta}_2 = L\dot{\phi} - \sin(\delta)\dot{x_m} - \cos(\delta)\dot{y_m} \tag{2.18}$$

De los desarrollos anteriores el modelo de la cinemática inversa para el marco de referencia interno resulta ser:

$$\begin{aligned} r\dot{\theta}_2 &= L\dot{\phi} - \sin(\delta)\dot{x_m} + \cos(\delta)\dot{y_m} \\ r\dot{\theta}_2 &= L\dot{\phi} - \sin(\delta)\dot{x_m} - \cos(\delta)\dot{y_m} \\ r\dot{\theta}_3 &= L\dot{\phi} + \dot{x_m} \end{aligned} \tag{2.19}$$

Nótese que el marco de referencia fijo para el espacio de trabajo está dado por $[X_\omega, Y_\omega]$ por lo tanto, considerando la transformación

$$\begin{pmatrix} \dot{x_\omega} \\ \dot{y_\omega} \end{pmatrix} = \begin{pmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} \dot{x_m} \\ \dot{y_m} \end{pmatrix} \tag{2.20}$$

Donde

$$\begin{pmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{pmatrix} \tag{2.21}$$

Es la matriz de rotación alrededor del eje ortogonal al plano. La transformación inversa de la ecuación (2.20) es

$$\begin{pmatrix} \dot{x_m} \\ \dot{y_m} \end{pmatrix} = \begin{pmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} \dot{x_\omega} \\ \dot{y_\omega} \end{pmatrix} \tag{2.22}$$

Sustituyendo en la ecuación (2.19) se tiene:

$$\begin{aligned} r\dot{\theta_1} &= L\dot{\phi} - \dot{x_\omega}\cos(\phi)\sin(\delta) - \dot{x_\omega}\sin(\phi)\cos(\delta) - \dot{y_m}\sin(\phi)\sin(\delta) + \dot{y_m}\cos(\phi)\cos(\delta) \\ r\dot{\theta_2} &= L\dot{\phi} - \dot{x_\omega}\cos(\phi)\sin(\delta) + \dot{x_\omega}\sin(\phi)\cos(\delta) - \dot{y_m}\sin(\phi)\sin(\delta) - \dot{y_m}\cos(\phi)\cos(\delta) \\ r\dot{\theta_3} &= L\dot{\phi} + \dot{x_\omega}\cos(\phi) + \dot{y_\omega}\sin(\phi) \end{aligned} \tag{2.23}$$

La utilización de identidades trigonométricas adecuadas permite la simplificación de la ecuación (2.23) dada por:

$$\begin{aligned} r\dot{\theta_1} &= -\dot{x_\omega}\sin(\delta+\phi) + \dot{y_\omega}\cos(\delta+\phi) + L\dot{\phi} \\ r\dot{\theta_2} &= -\dot{x_\omega}\sin(\delta+\phi) - \dot{y_\omega}\cos(\delta+\phi) + L\dot{\phi} \\ r\dot{\theta_3} &= \dot{x_\omega}\cos(\phi) + \dot{y_\omega}\sin(\phi) + L\dot{\phi} \end{aligned} \tag{2.24}$$

Por lo tanto, las ecuaciones que describen la cinemática inversa del robot móvil omnidireccional están descritas verticalmente por:

$$\begin{pmatrix} \dot{\theta_1} \\ \dot{\theta_2} \\ \dot{\theta_3} \end{pmatrix} = \frac{1}{r} \begin{pmatrix} -\sin(\delta+\phi) & cos(\delta+\phi) & L \\ -\sin(\delta+\phi) & -cos(\delta+\phi) & L \\ \cos(\phi) & \sin(\phi) & L \end{pmatrix} \begin{pmatrix} \dot{x_\omega} \\ \dot{y_\omega} \\ \dot{\phi_\omega} \end{pmatrix} \tag{2.25}$$

El mapeo inverso de la ecuación (2.25) produce

$$\begin{pmatrix} \dot{x_\omega} \\ \dot{y_\omega} \\ \dot{\phi_\omega} \end{pmatrix} = \begin{pmatrix} \frac{-\sin(\phi)L - L\cos(\delta+\phi)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{\sin(\phi)L - L\cos(\delta+\phi)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{L\cos(\delta+\phi) + L\cos(\delta-\phi)}{2\cos(\delta)L + L\sin(2\delta)} \\ \frac{\cos(\phi)L + L\sin(\delta+\phi)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{-\cos(\phi)L - L\sin(\delta+\phi)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{L\sin(\delta+\phi) - L\sin(\delta-\phi)}{2\cos(\delta)L + L\sin(2\delta)} \\ \frac{\cos(\delta)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{\cos(\delta)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{\sin(2\delta)}{2\cos(\delta)L + L\sin(2\delta)} \end{pmatrix} \begin{pmatrix} r_r\dot{\theta_1} \\ r\dot{\theta_2} \\ r\dot{\theta_3} \end{pmatrix} \tag{2.26}$$

Considerando ahora la relación $2\sin(\delta) = \frac{\sin(2\delta)}{\cos(\delta)}$ y las propiedades de la suma de los ángulos es posible simplificar algunos de los términos de la ecuación (2.26)

$$\begin{pmatrix} \dot{x_\omega} \\ \dot{y_\omega} \\ \dot{\phi_\omega} \end{pmatrix} = \begin{pmatrix} \frac{-\sin(\phi)L - L\cos(\delta+\phi)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{\sin(\phi)L - L\cos(\delta+\phi)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{L\cos(\phi)}{L + L\sin(\delta)} \\ \frac{\cos(\phi)L + L\sin(\delta+\phi)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{-L\cos(\phi)L - L\sin(\delta+\phi)}{2\cos(\delta)L + L\sin(2\delta)} & \frac{L\sin(\phi)}{L + L\sin(\delta)} \\ \frac{1}{2L + 2L\sin(2\delta)} & \frac{1}{2L + 2L\sin(2\delta)} & \frac{\sin(\delta)}{L + L\sin(\delta)} \end{pmatrix} \begin{pmatrix} r_r\dot{\theta}_1 \\ r\dot{\theta}_2 \\ r\dot{\theta}_3 \end{pmatrix} \tag{2.27}$$

Finalmente esto se puede representar como:

$$\begin{pmatrix} r\dot{\theta}_1 \\ r\dot{\theta}_2 \\ r\dot{\theta}_3 \end{pmatrix} = \begin{pmatrix} \frac{1}{2}\sqrt{3} & \frac{1}{2} & L \\ 0 & -1 & L \\ -\frac{1}{2}\sqrt{3} & \frac{1}{2} & L \end{pmatrix} \begin{pmatrix} \dot{x}_m \\ \dot{y}_m \\ \dot{\phi}_m \end{pmatrix} \tag{2.28}$$

2.4. Control colaborativo

Para implementar el control colaborativo se considera un conjunto de N robots móviles. Se define entonces por medio de la representación de espacio de estados el vector $x_{i,s} = \begin{pmatrix} x_{i,1} & x_{i,2} & x_{i,3} \end{pmatrix}^T \in \mathbb{R}^3$ el cual se conforma por los elementos de las posiciones $\dot{x_m}$ y $\dot{y_m}$ así como de la orientación $\dot{\phi_m}$ respectivamente, las cuales fueron presentadas en la ecuación (2.28). También, se define $m_{i,s} = \begin{pmatrix} m_{i,1} & m_{i,2} & m_{i,3} \end{pmatrix}^T$ como el conjunto de los estados $x_{i,s}$ la última vez que ocurrió un evento, donde $i \in V$ y $s \in \{1, 2, 3\}$. De igual manera, se introduce $u_{i,s} = \begin{pmatrix} u_{i,1} & u_{i,2} & u_{i,3} \end{pmatrix}^T \in \mathbb{R}^3$, el cual es el vector que conforma un control por linealización exacta, permitiendo así que el modelo cinemático del vehículo omnidireccional resulte en:

$$\begin{aligned} \dot{x}_{i,1} &= u_{i,1}cos(x_{i,3}) - u_{i,2}sin(x_{i,3}) \\ \dot{x}_{i,2} &= u_{i,1}sin(x_{i,3}) + u_{i,2}cos(x_{i,3}) \\ \dot{x}_{i,3} &= u_{i,3} \end{aligned} \tag{2.29}$$

En donde el control por linealización exacta $u_{i,s}$ que se desea para este sistema se define como:

$$\begin{aligned} u_{i,1} &= cos(x_{i,3}) * r_{i,1} + sin(x_{i,3}) * r_{i,2} \\ u_{i,2} &= -sin(x_{i,3}) * r_{i,1} + cos(x_{i,3}) * r_{i,2} \\ u_{i,3} &= r_{i,3} \end{aligned} \tag{2.30}$$

De tal manera que el modelo del i-ésimo robot resultará en un sistema delimitado por

simples integradores de la forma:

$$\begin{aligned} \dot{x}_{i,1} &= r_{i,1} \\ \dot{x}_{i,2} &= r_{i,2} \\ \dot{x}_{i,3} &= r_{i,3} \end{aligned} \tag{2.31}$$

Donde $r_{i,s} = \begin{pmatrix} r_{i,1} & r_{i,2} & r_{i,3} \end{pmatrix}^T$ representa la estrategia de control colaborativa, basada en el consenso para llegar a un punto común la cual se definirá como:

$$r_{i,s} = r^{\alpha}_{i,s} + r^{\gamma}_{i,s} \tag{2.32}$$

Donde $r^{\gamma}_{i,s}$, corresponde al control colaborativo por consenso y $r^{\alpha}_{i,s}$ corresponde a un control para evadir colisiones entre agentes.

El control basado en consenso se define como:

$$r^{\gamma}_{i,s} = \kappa_1 \left[\sum_{j \in \mathbb{N}} (m_{j,s} - m_{i,s}) + g_i(\xi_{0,s} - m_{i,s}) \right] \tag{2.33}$$

Donde $s \in \{1, 2, 3\}$, $\kappa > 0$, y g_i, siendo la ganancia de fijación que determina si el agente i estará siguiendo a un líder virtual (con $g_i = 1$ en caso de ser así y con $g_i = 0$ en caso contrario) y $\xi_{0,s}$ denota la referencia o trayectoria a seguir por el líder virtual. Para generar formaciones y que los agentes no lleguen al mismo punto, se agrega un elemento para generar la agrupación deseada mediante un conjunto F de locaciones asociadas, dado por:

$$F = \{\zeta_1, \zeta_2, ...\zeta_N\}, \ \zeta_i \in \mathbb{R}^3, \ i = 1, .., N \tag{2.34}$$

Donde

$$\|\zeta_i - \zeta_j\| = \varrho_{ij} \tag{2.35}$$

Convirtiendo así la ecuación (2.33) en:

$$r^{\gamma}_{i,s} = \kappa \left[\sum_{j \in \mathbb{N}} (m_{j,s} - m_{i,s}) - (\zeta_{j,s} - \zeta_{i,s}) + g_i(\xi_{0,s} - m_{i,s}) \right] \tag{2.36}$$

Donde $d_i = (\zeta_{j,s} - \zeta_{i,s})$ representa la distancia entre el i-ésimo agente y el j-ésimo vecino.

2.5. Evasión de colisiones entre agentes

En [28] se introduce un método para implementar un algoritmo de evasión de colisiones. Este algoritmo considera un comportamiento en "flocking" de los agentes que conforman

el grupo. Cada nodo es considerado como un agente de segundo orden, usando los valores de sus posiciones para el término que previene las colisiones, mientras que las velocidades son utilizadas para el término que permite el consenso. Sin embargo, en este trabajo los agentes son definidos como un sistema de primer orden y, por tanto, el consenso es calculado usando también las posiciones y orientaciones de los vehículos. El término para la evasión de colisiones entre agentes, según [28], es u_i^α. Esta función está basada en una función de campos potenciales atractiva/repulsiva que permite calcular un término adicional cuando se puede dar una colisión. Los vehículos cercanos se denominan α-agentes los cuales, en consecuencia del enfoque basado en eventos, tienen un conjunto de posiciones previas $\hat{m}_j = \begin{pmatrix} m_{j,1} & m_{j,2} \end{pmatrix}^T \in \mathbb{R}^2$ mientras que el conjunto de posiciones actuales del nodo son $\hat{m}_i = \begin{pmatrix} m_{i,1} & m_{i,2} \end{pmatrix}^T \in \mathbb{R}^2$. Los nodos considerados como α-agentes son:

$$N_i = j \in V : \|\hat{m}_j - \hat{m}_i\| < r \tag{2.37}$$

donde r es el rango de interacción entre agentes. La ley de control introducida para la evasión de colisiones con α-agentes es:

$$r_{i,s}^\alpha = c_\alpha \sum_{j \in N_i} \phi_\alpha(\|\hat{m}_j - \hat{m}_i\|_\sigma)\hat{n}_{ij} \tag{2.38}$$

Donde la norma sigma $\|z\|$ de la matriz z es calculada por:

$$\|z\|_\sigma = \frac{1}{\epsilon}[\sqrt{1 + \epsilon \|z\|^2} - 1] \tag{2.39}$$

Además $c_\alpha > 0$ y $\hat{n_{ij}}$ es un vector definido como:

$$\hat{n}_{ij} = \frac{\hat{m}_j - \hat{m}_i}{\sqrt{1+\epsilon\|\hat{m}_j - \hat{m}_i\|}} \tag{2.40}$$

Con $\epsilon > 0$. El término ϕ_α es una función de acción para $z = \|\hat{m}_j - \hat{m}_i\|$ que se desvanece siempre que los α-agentes se encuentran lo suficientemente lejos el uno del otro. La función de acción se define como:

$$\phi_\alpha = \rho_h(z/r_\alpha)\phi(z - d_\alpha) \tag{2.41}$$

donde $\rho_h(z)$ es una función "bump" que consiste en una función escalar que varia suavemente desde 0 a 1 y se define por:

$$\rho_h(z) = \begin{cases} 1, & z \in [0, h) \\ \frac{1}{2}[1 + cos(\pi \frac{(z-h)}{1-h})] & z \in [h, 1] \\ 0 & de\, otra\, manera \end{cases} \tag{2.42}$$

y $\phi(z)$ es una función sigmoidal desigual definida por:

$$\phi(z) = \frac{1}{2}[(a+b)\sigma_1(z+c) + (a-b)] \tag{2.43}$$

Donde:

$$\sigma_1 = z/\sqrt{1+z^2} \tag{2.44}$$

Con $0 < a \leq b, c = a - b/\sqrt{4ab}$ y donde $r_\alpha = \|r\|_\sigma$ y $d_\alpha = \|d\|_\sigma$ son la σ-norma del radio r y de la distancia d entre los agentes respectivamente. El valor d se obtiene con:

$$d = \|\hat{m}_j - \hat{m}_i\| \tag{2.45}$$

Finalmente, la ecuación (2.32) con base en la ecuación (2.33) y en la ecuación (2.38) resulta:

$$r_{i,s} = \kappa \left[\sum_{j \in \mathbb{N}} (m_{j,s} - m_{i,s}) - (\zeta_{j,s} - \zeta_{i,s}) + g_i(\xi_{0,s} - m_{i,s}) \right] + c_\alpha \sum_{j \in N_i} \phi_\alpha(\|\hat{m}_j - \hat{m}_i\|_\sigma)\hat{n}_{ij} \tag{2.46}$$

2.6. Evasión de obstáculos

Esta técnica se desglosa de la siguiente manera:

- Determinar el conjunto de obstáculos O_k que están cercanos al vehículo.
- Crear un agente virtual en el límite del área del obstáculo.
- Agregar el término de potencial colectivo de repulsión $\phi_\beta(\|\hat{m}_{i,k} - m_i\|)$

Este algoritmo se representa el obstáculo como una circunferencia, representado como el agente virtual β, el cual tendrá una posición $\hat{m}_{i,k}$ mientras que la posición del vehículo será m_i.

Para conseguir que el vehículo logre evitar el obstáculo es necesario agregar en la señal de referencia el término de potencial colectivo, este término consiste en:

$$u_\beta = c_\beta \phi_\beta(\|\hat{m}_{i,k} - m_i\|_\sigma)\hat{n}_{i,k} \tag{2.47}$$

Aquí c_β es una ganancia mayor a 0. además:

$$\hat{n}_{i,k} = \frac{\hat{m}_{i,k} - m_i}{\sqrt{1 + \epsilon \|\hat{m}_{i,k} - m_i\|^2}} \tag{2.48}$$

A su vez

$$\phi_\beta(\|\hat{m}_{i,k} - m_i\|_\sigma) = \rho_h(\|\hat{m}_{i,k} - m_i\|_\sigma / r_\beta)(\sigma_1(\|\hat{m}_{i,k} - m_i\|_\sigma - d\beta) - 1) \tag{2.49}$$

ϵ es una constante mayor a 0. En donde ρ_β es una *"bump function"* que permitirá variar suavemente entre 0 y 1, definiéndose por:

$$\rho_h((\|\hat{m}_{i,k} - m_i\|_\sigma / r_\beta) = \begin{cases} 1, & \|\hat{m}_{i,k} - m_i\|_\sigma / r_\beta \ \epsilon[0, h) \\ \frac{1}{2}[1 + cos(\pi \frac{(\|\hat{m}_{i,k} - m_i\|_\sigma / r_\beta - h)}{1-h})] & \|\hat{m}_{i,k} - m_i\|_\sigma / r_\beta \ \epsilon[h, 1] \\ 0 & otherwise \end{cases} \tag{2.50}$$

En donde h permitirá determinar el grado de variación de la función y será un valor entre 0 y 1. También:

$$\sigma_1(\|\hat{m}_{i,k} - m_i\|_\sigma - d\beta) = \frac{\|\hat{m}i, k - m_i\|_\sigma - d\beta}{\sqrt{1 + (\|\hat{m}i, k - m_i\|_\sigma - d\beta)^2}} \tag{2.51}$$

Finalmente:

$$\|z\|_\sigma = \frac{1}{\epsilon}[\sqrt{1 + \epsilon \|z\|^2} - 1] \tag{2.52}$$

En donde:

$$\|x - y\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} d_\beta = \|d\|_\sigma \tag{2.53}$$

Donde:

$$d = \|\hat{m}_{i,k} - m_i\| \tag{2.54}$$

A su vez

$$r_\beta = \|r\|_\sigma \tag{2.55}$$

Donde r es el radio que el vehículo respetará para evitar la colisión con el obstáculo.

Con el algoritmo anterior, cada vez que el agente se encuentre cerca del área del obstáculo en función al radio r de cobertura, la *bump function* generará un valor diferente de 0 para agregar el término a la señal de control y evitar así que el vehículo choque con el obstáculo.

Finalmente la ley de control se compone agregando el término u_β a la ecuación 2.46. Dicha ecuación de control se implemento en MATLAB/Simulink como se muestra en la Figura 2.6, esta implementación incluye a los extremos (Figuras 2.7 y 2.8) los bloques necesarios para la comunicación con la red de ROS. En la parte superior de la Figura 2.7 se obtiene la posición del robot (3,0), en la parte inferior de la misma se adquiere la posición del obstáculo. En la parte inferior de ña Figura 2.9 se implementa la ley de control, mientras que en el recuadro superior, se implementa el generador de trayectoria, el cual puede ser sustituido por una suscripción a un tópico de tipo *"pose"* en donde se publique una posición deseada desde la red de ROS, por último, Figura 2.8, se encuentra el

publicador, por donde se transmiten las velocidades requeridas por el robot para realizar el seguimiento de la trayectoria deseada.

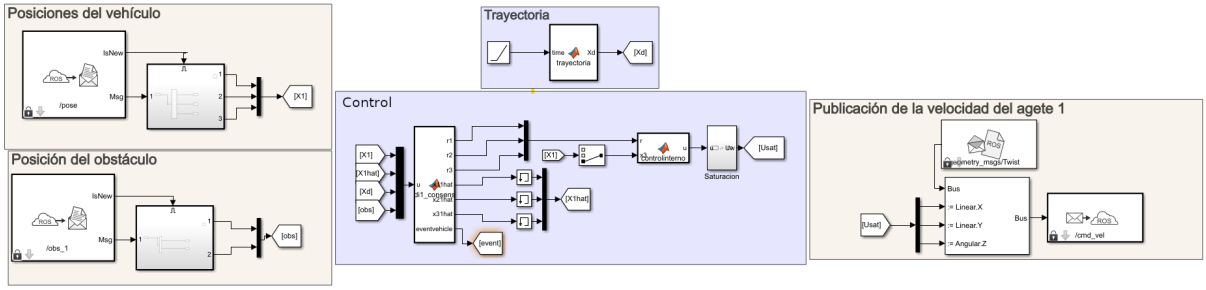

Figura 2.6: Diagrama de bloques de sistema de control.

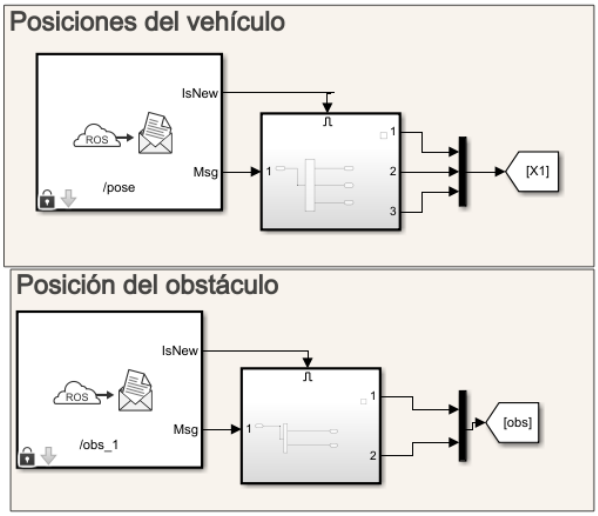

Figura 2.7: Suscriptores para la obtención de la posición del vehículo y el obstáculo.

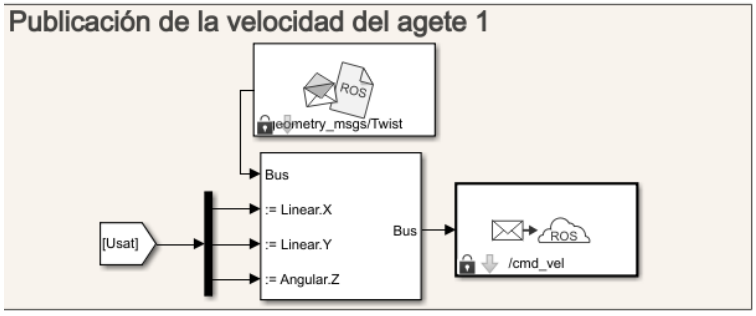

Figura 2.8: Publicador de las velocidades.

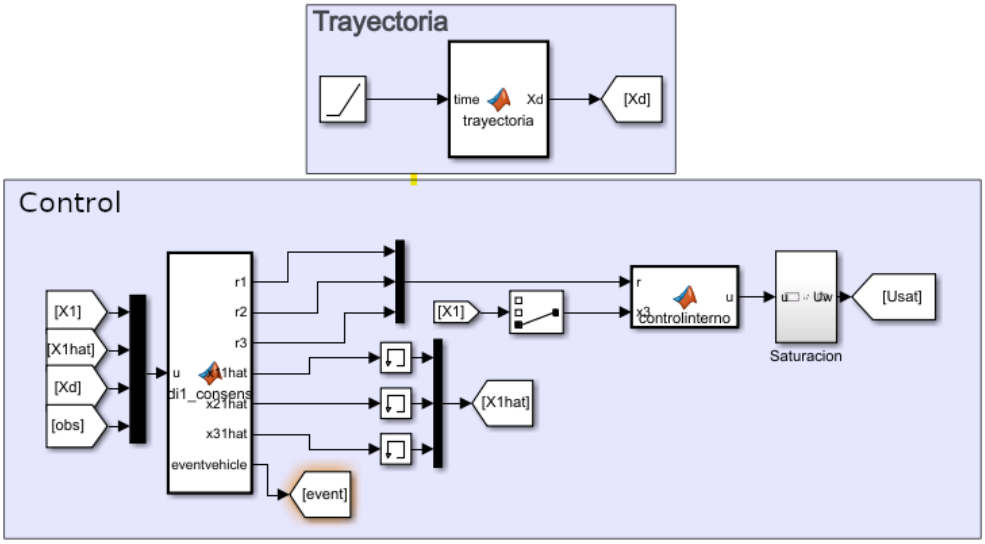

Figura 2.9: Etapa de control y generador ed trayectoria.

2.7. Simulación del modelo del robot (3,0)

Partiendo del grupo de ecuaciones (2.27) descritas en la Sección 2.3 Modelo cinemático del robot (3,0), se implementó en MATLAB/Simulink el siguiente diagrama de bloques (ver Figura 2.10).

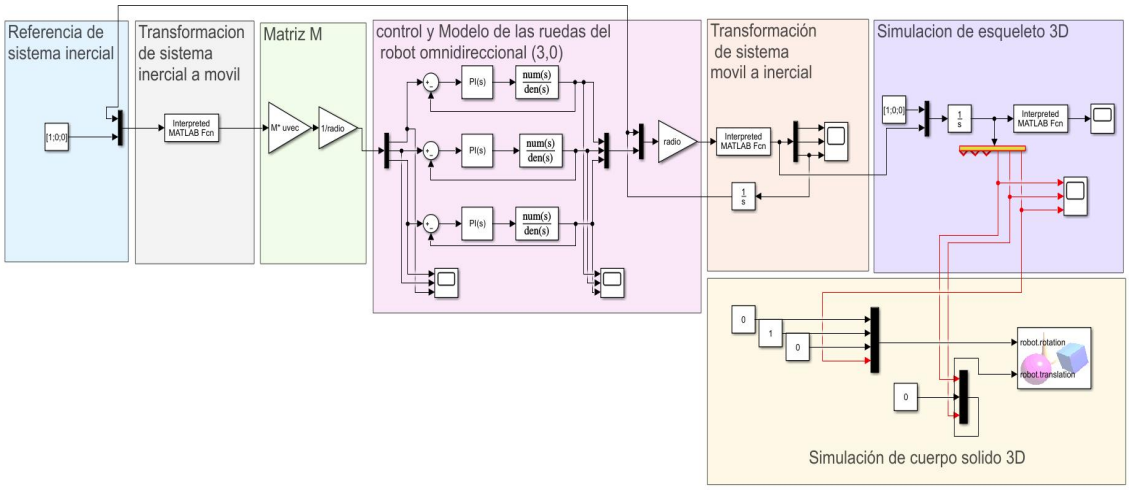

Figura 2.10: Simulación de Robot omnidireccional (3,0).

La primera sección del diagrama de bloques representa la referencia del sistema inercial, la cual está conformada por un vector de tres elementos, los cuales describen las

velocidades deseadas en dirección a **X**, **Y** y la velocidad angular deseada (sección azul de la Figura 2.11).

En la segunda sección se realiza una transformación de coordenadas del sistema inercial al móvil (sección gris de la Figura 2.11).

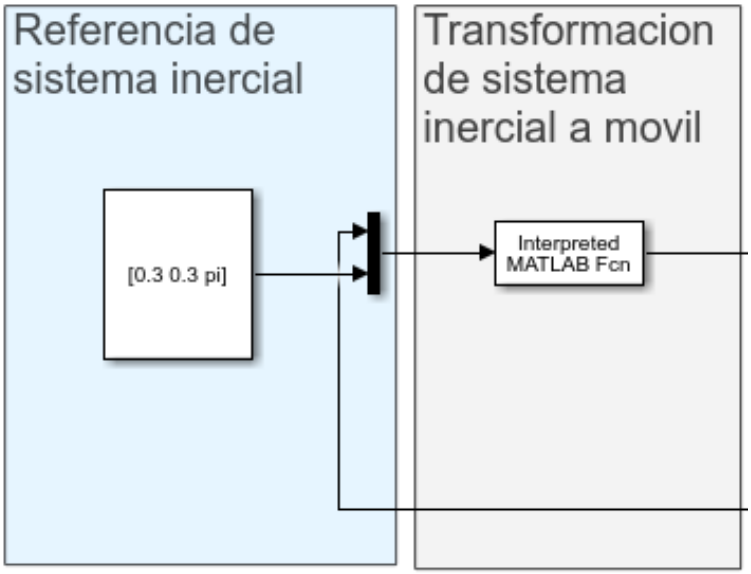

Figura 2.11: Primera y segunda sección.

Una vez hecho esto en la sección "*matrizM*" se multiplica los vectores obtenidos por la matriz de inercias y se divide este valor entre el radio de las ruedas, para de este modo obtener las velocidades angulares correspondientes a cada rueda del robot omnidireccional 2.12.

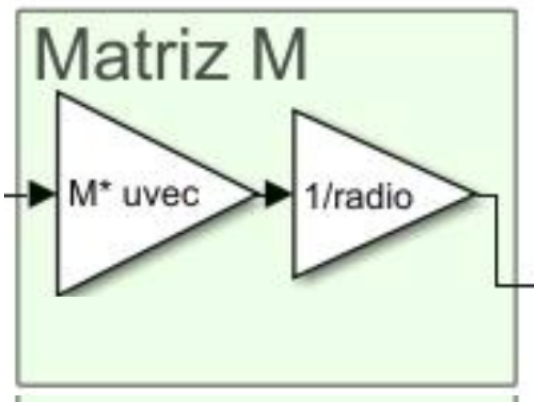

Figura 2.12: Tercera sección, matriz de inercias.

Posteriormente, las velocidades de cada motor son introducidas en el modelo del motor correspondiente, a los cuales se les implementó un controlador de tipo PI 2.13.

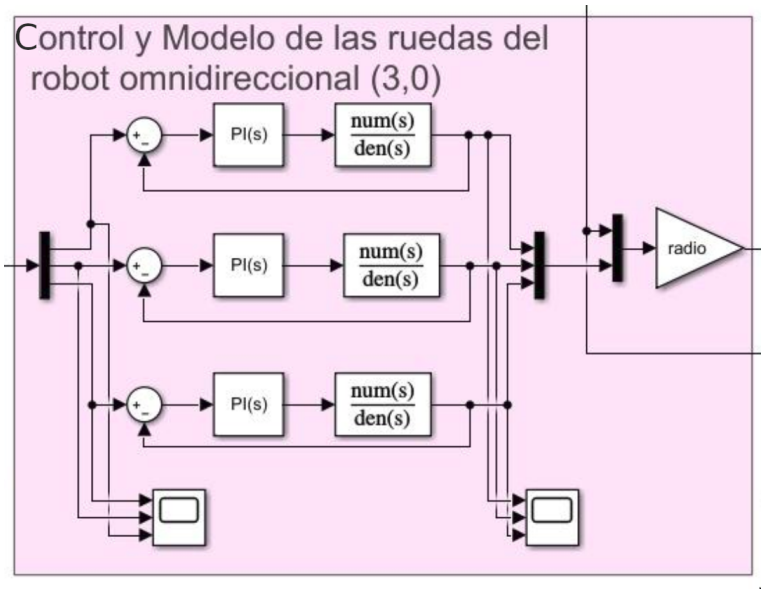

Figura 2.13: Modelo de bloques del robot omnidireccional (3,0).

Una vez obtenidas las velocidades de cada rueda, estas se introducen en el sistema de ecuaciones para poder obtener la posición y la orientación con respecto al eje de referencia inercial.

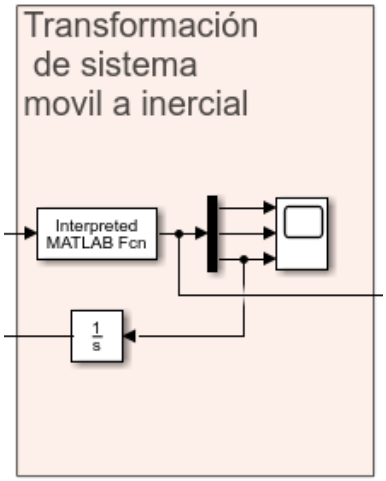

Figura 2.14: Transformación del sistema móvil a inercial.

Una vez obtenidas las componentes de posición en **X** y **Y**, así como la orientación, se introducen a 2 bloques de simulación, el primero realiza una simulación dibujando los bordes del cuerpo del robot como se muestra en la Figura 2.15, el segundo realiza la simulación utilizando el bloque $VRSINK$, como se puede apreciar en la Figura 2.16.

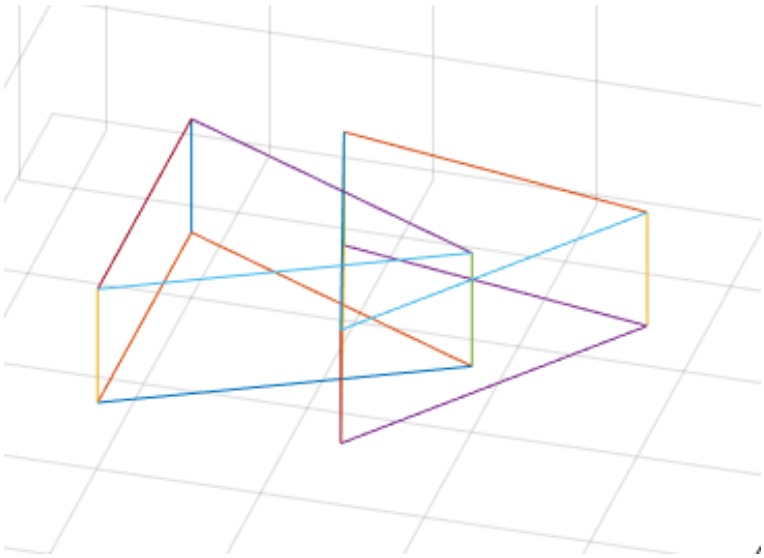

Figura 2.15: Simulación de bordes del robot omnidireccional (3,0).

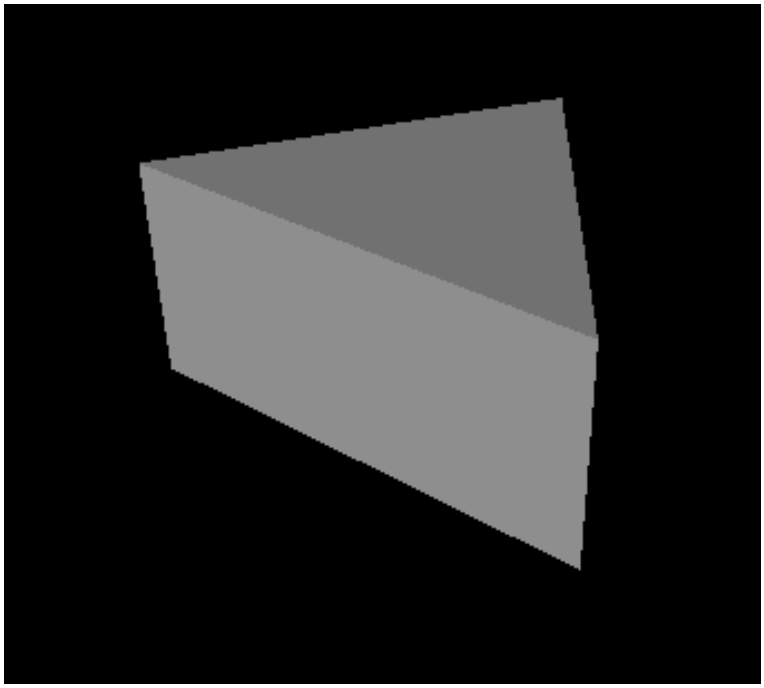

Figura 2.16: Simulación utilizando VR SINK.

Finalmente podemos observar de manera gráfica el desplazamiento del robot en la Figura 2.17 en donde las velocidades de referencia fueron $[0.3,\ 0.3,\ pi]$, por lo que el robot se mueve en linea recta en una dirección a 45 grados del eje **X** ya que las velocidades tanto en **X** como en **Y** son iguales, pero, realiza este movimiento al mismo tiempo que gira sobre su eje a una velocidad de pi radianes sobre segundo.

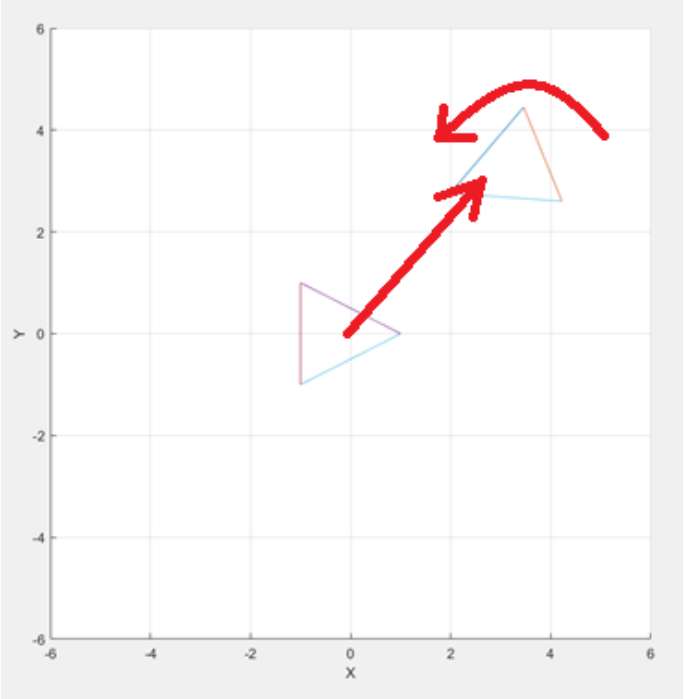

Figura 2.17: Desplazamiento del robot con velocidades $[0.3, 0.3, pi]$.

Con lo anterior se logró alcanzar las velocidades angulares deseadas, por ejemplo, para el desplazamiento descrito anteriormente se obtienen las siguientes velocidades mostradas en la Figura 2.18.

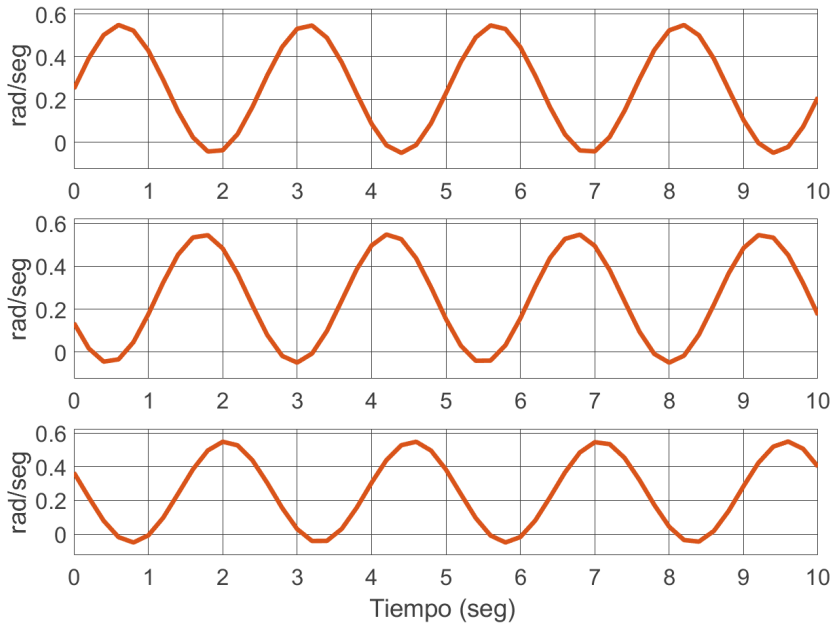

Figura 2.18: Velocidades de las ruedas del robot omnidireccional (3,0) para alcanzar las velocidades deseadas $[0.3,\ 0.3,\ \pi]$.

2.8. Simulación de robot (3,0) en Gazebo

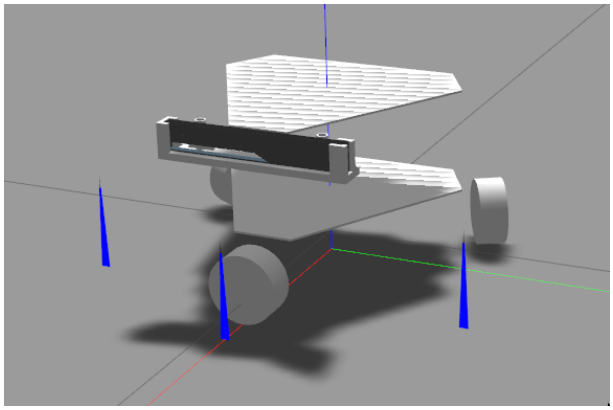

Figura 2.19: Robot (3,0) en la plataforma *"Rviz"*.

En la Figura 2.19 se muestra la representación del robot implementado en la plataforma de Gazebo, para realizar la simulación en este entorno, el robot se crea a partir de archivos descriptores, los cuales son conocidos como URDF, estos archivos son de estructura XML y describen los diferentes aspectos del robot, como son las eslabones de la cadena cinemática, denominados *"LINKS"*, así como las articulaciones o *"JOINTS"*, por medio de las cuales nuestro robot logrará desplazarse. Para nuestro propósito, únicamente utilizaremos un tipo de articulaciones, las cuales se denominan "continuas", éstas permiten una rotación continua, de este modo se emulará el funcionamiento de los motores.

Para que el robot funcione dentro del simulador, es necesario que cuente con *"plugins"*, los cuales son encargados de conectar los nodos y transmitir la información de modo que se puedan emular las circunstancias de movimiento reales.

Para la simulación de este robot en especifico, se utilizaron diversos *"plugins"*, como: *"Joint state publisher"*, el cual es el encargado de publicar el estado de las articulaciones del robot, de este modo, se pueden actualizar los cálculos de las ecuaciones físicas, según sea necesario. Así mismo se implementa el "*Plugins*" de "*planar move*" el cual emulará el movimiento omnidireccional de la base utilizada.

Con el fin de probar los algoritmos a implementar en el robot que se desarrolló, se implementó una simulación, la cual, consta de la base del robot con sus respectivas ruedas y la cámara *"Realsense r200"*, con lo que se simulará el funcionamiento del sistema real.

Con el fin de ilustrar el archivo URDF implementado, es posible utilizar una herramienta grafica, denominada RQT, la cual nos otorga una representación de las relaciones que conforman al robot. en la Figura 2.20 se muestran las relaciones de la base con sus ruedas a demás de la relación con el *bracket*, el cual es la base que sostiene a la cámara *Realsense r200*, también, en la Figura 2.21 se muestran las relaciones de los componentes que representan a la cámara.

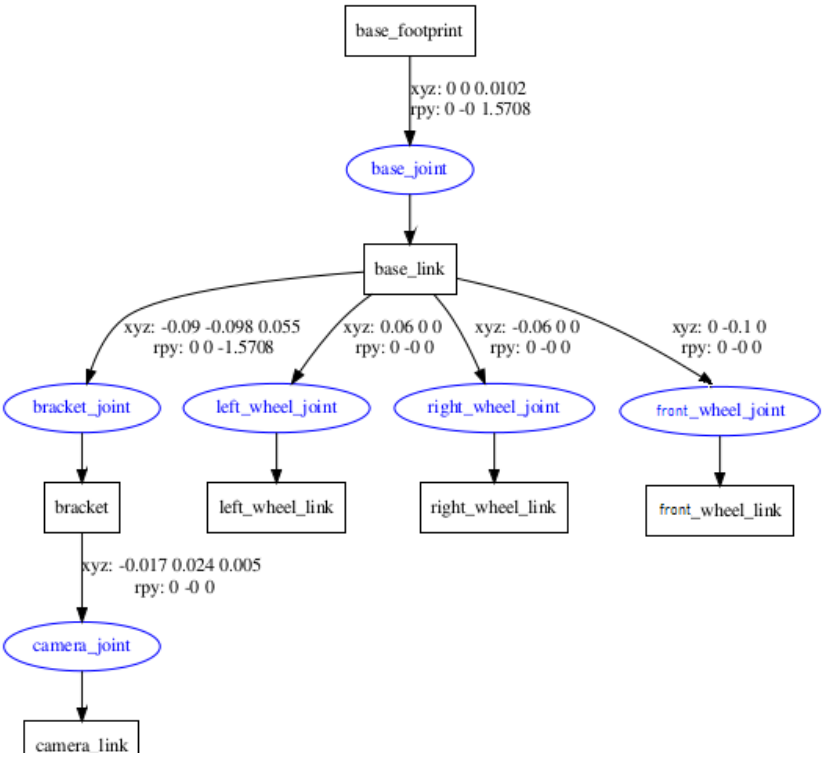

Figura 2.20: Representación grafica de los componentes del robot y sus relaciones.

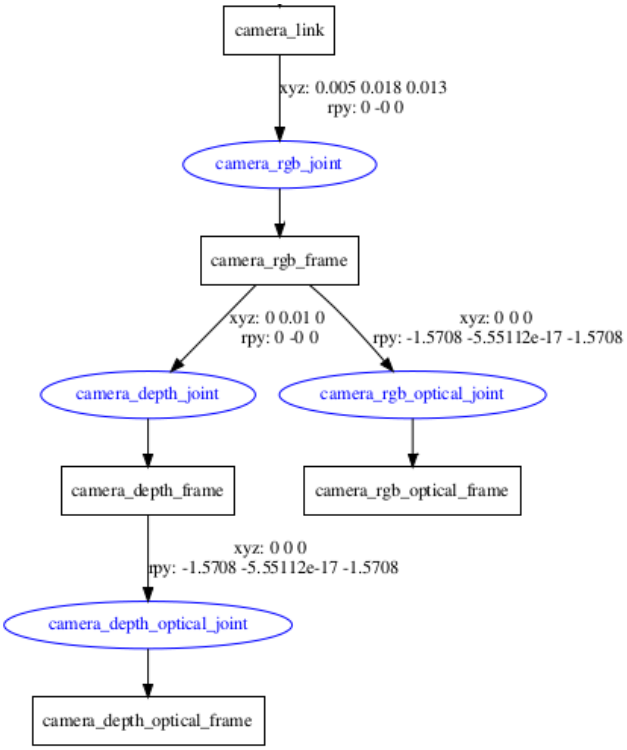

Figura 2.21: Representación grafica de los componentes de la cámara y sus relaciones.

2.9. Implementación del robot (3,0)

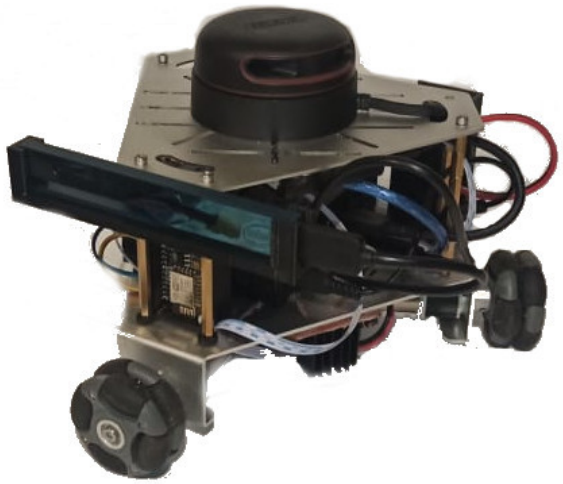

Figura 2.22: Robot (3,0) implementado.

Para el ensamblaje de este robot (véase la Figura 2.22) se utilizó una plataforma delta, como se muestra en la Figura 2.23, la cual cuenta con motores y moto-reductores con una relación 80:1, y encoders de cuadratura, lo que nos proporciona una resolución de 160:1 o de 2.25° por cada señal del encoder.

Figura 2.23: Plataforma delta.

En esta plataforma se integraron diversos componentes para lograr su funcionamiento y su autonomía, de los cuales podemos mencionar:

- Tarjeta de cómputo empotrado Jetson NANO: Es la encargada de realizar el procesamiento de los algoritmos implementados.

- Tarjeta de desarrollo Node MCU: Es la encargada de interpretar las ordenes de movimiento para cada motor, así como del cálculo de la odometría, la cual se devuelve a la tarjeta de cómputo empotrado Jetson NANO por medio de la red de ROS.
- Puentes H L298N: los cuales funcionan como interfaces de potencia entre la tarjeta de desarrollo Node MCU y los motores, estos puentes H son capaces de manejar voltajes desde los 3v a los 35v a una corriente de 2 amperes.
- *Power bank* insignia: esta batería sera la encargada de suministrar la energía a la tarjeta de cómputo empotrado, ya que cuenta con una capacidad de 1200 mAh y dos salidas de 5 volts a 2.1 amperes.
- Batería LiPo de 7.4 volts a 2 amperes: esta batería será la encargada de proporcionar la energía a los motores.
- Cámara Intel *Realsense R200*: Esta cámara conforma nuestro sistema de visión, de ella se obtendrán los datos de imágenes, así como datos de profundidad.
- LIDAR RPLIDAR A2: Es un sensor LIDAR con un rango de visión de 360°.

La tarjeta de cómputo empotrado Jetson Nano no cuenta con salidas PWM suficientes para realizar el control de los motores de robot, es por ello que se implemento la tarjeta Node MCU (véase Figura 2.24) la cual cuenta con un módulo "*Wi-Fi*" por medio del cual se estableció la comunicación con la tarjeta Jetson NANO a través de la red de ROS.

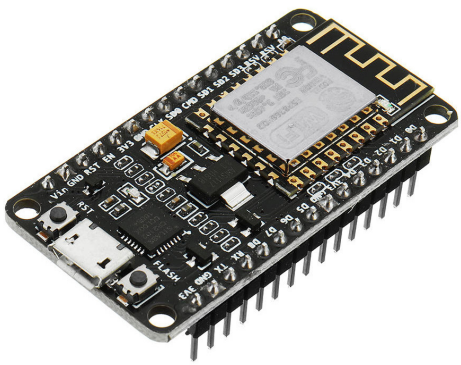

Figura 2.24: Ilustración de la tarjeta de desarrollo Node MCU.

En ella se programaron las ecuaciones descritas en la sección 2.3. Tanto cinemática directa, para la generación de movimiento, como la cinemática inversa, para el cálculo de la odometría a partir de las señales de los encoders. se incluye el código de esta tarjeta en el apéndice A.

En la Figura 2.25 se presenta el diagrama de conexión de los puentes H L298N con la tarjeta Node-MCU.

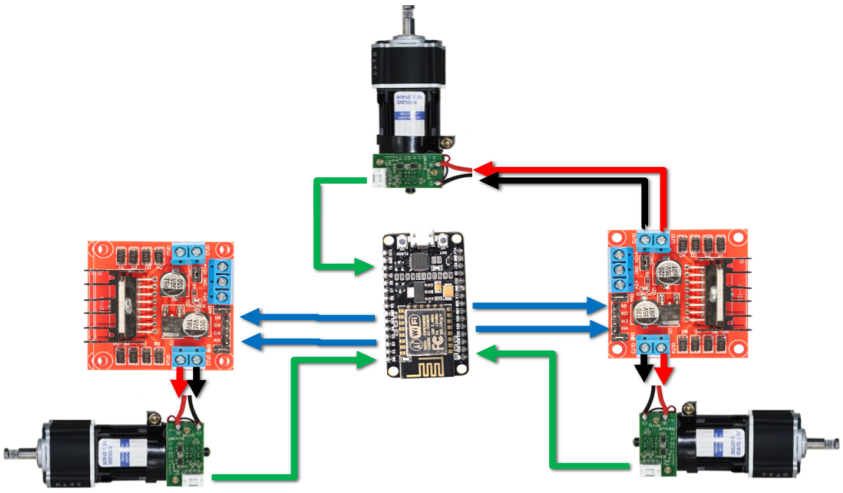

Figura 2.25: Diagrama de conexión de los motores.

Finalmente para la comunicación con la red de ROS se ejecuta un nodo, denominado ROS_SERIAL, el cual utiliza la dirección IP de la computadora que ejecuta este nodo, en este caso, la tarjeta Jetson NANO, esto se logra por medio del comando:

Ejecución de ROS_SERIAL

```
rosrun rosserial_python serial_node.py tcp
```

Para probar el funcionamiento de la correcta comunicación entre los nodos de la red se cuenta con una aplicación de *"ANDROID"* o con herramientas en la consola de comandos, como lo es *TELEOP_TWIST_KEYBOARD*, por medio de los cuales es posible enviar instrucciones dentro de la red de ROS, por medio de mensajes de tipo "*pose*", en donde se define una componente de velocidad lineal, a lo largo de los ejes X y Y del robot y al mismo tiempo una componente angular que describe el giro sobre el eje Z.

El diagrama de conexión en la red de ROS se muestra en la Figura 2.26.

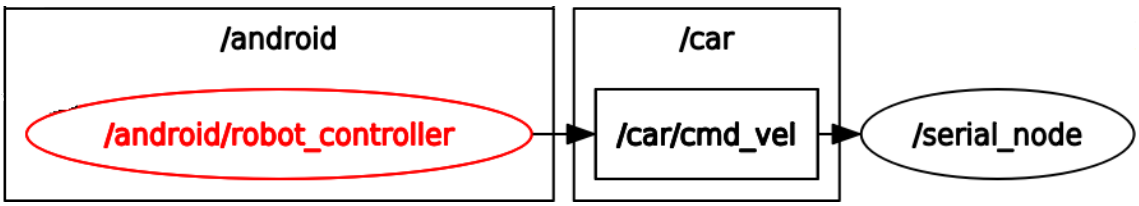

Figura 2.26: Diagrama de conexiones de la red de ROS por medio de la cual se controla el carro de tipo (3,0).

Dentro de la tarjeta de desarrollo *"Node MCU"* se ejecuta un nodo de ROS,en el cual se realiza tanto el control de velocidad de los motores, gracias a los encoders de cada

motor. Las señales que se adquieren de estos son procesadas en un controlador *"PID"*, al mismo tiempo, se calcula la odometría a partir del conteo y de los pulsos provenientes de los encoders y de la aplicación de las ecuaciones de la cinemática inversa, una vez calculada, es enviada devuelta a la red de ROS, esto con el fin de que el nodo maestro pueda adquirir dicha información e interpretarla como un cambio de posición y así poder cerrar el lazo de control.

2.9.1. Implementación utilizando OPTITRACK

Para realizar la integración del sistema OPTITRACK, se colocó en la plataforma una estructura de marcadores (véase la Figura 2.27), la cual por medio del software *"MOTIVE"*, se agrupan y se identifica como un agente, del cual se puede conocer su ubicación y orientación en las 3 dimensiones.

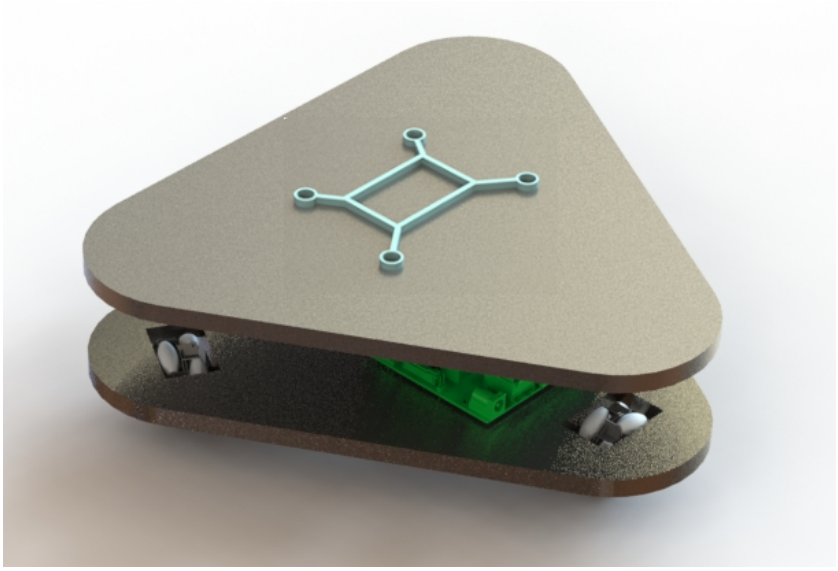

Figura 2.27: Ilustración del robot omnidireccional (3,0) con estructura de marcadores para seguimiento con *"OPTITRACK"*.

Una vez adquirida la posición del robot, ésta debe ser comunicada al robot por medio de la red de ROS, esto se logra por medio de un programa denominado **VRPN** (Virtual-Reality Peripherical Network), el cual levantará un nodo en la red de ROS y publicará las posiciones bajo el tópico de **/agent11/pose/**. los mensajes de tipo pose tiene una estructura definida por ROS, que está conformado por un punto de posición espacial, seguido de un cuaternión para definir la orientación.

2.10. Resultados experimentales

En esta sección del documento se describirán los resultados de los experimentos conducidos tanto de simulación como en las implementaciones realizadas, las cuales se dividen en dos, la primera, por medio del uso del sistema de captura *OPTITRACK*, el cual, es el encargado de la adquisición de la posición del robot en un área delimitada denominada arena. la segunda implementación se realizó utilizando los datos provenientes del la odometría calculada a bordo.

2.10.1. Simulación en Gazebo

Las simulaciones se dirigieron dentro del simulador *Gazebo* en conjunto con la plataforma ROS, por medio de los cuales se obtuvieron las posiciones tanto del robot como de los obstáculos, como se puede ver en a Figura 2.28.

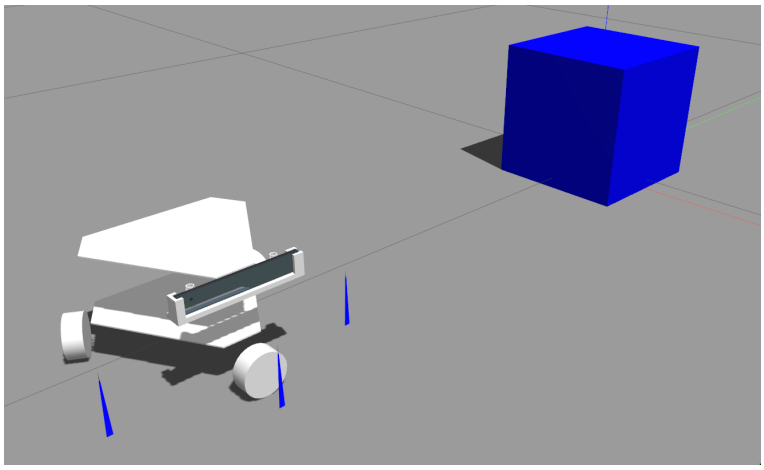

Figura 2.28: Agente y obstáculo, dentro del entorno de simulación Gazebo.

A continuación se describirán algunas de las pruebas dirigidas durante la elaboración del sistema de control implementado.

En la Figura 2.29 se muestra el seguimiento de trayectoria realizado por el robot, partiendo este de la posición (0,-1) y realiza una trayectoria circular, de 2 metros de radio, centrado en el origen. Como se puede apreciar en la Figura 2.30 se presenta un pico en el error de posición, esto se debe a que el robot parte de un punto que no coincide con el inicio de la trayectoria, para posteriormente realizar el seguimiento de ésta. A partir de los datos recabados en esta gráfica, fue posible calcular el Error Cuadrático Integral (ISE), el cual tuvo un valor de 3.69.

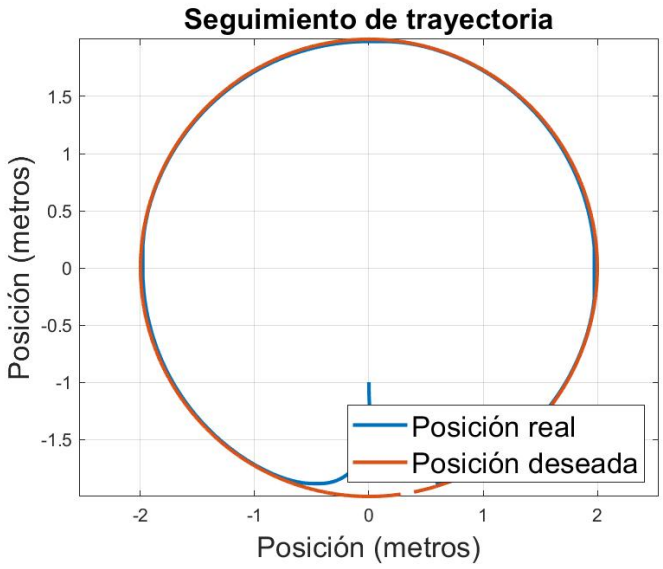

Figura 2.29: Muestra del seguimiento de trayectoria circular.

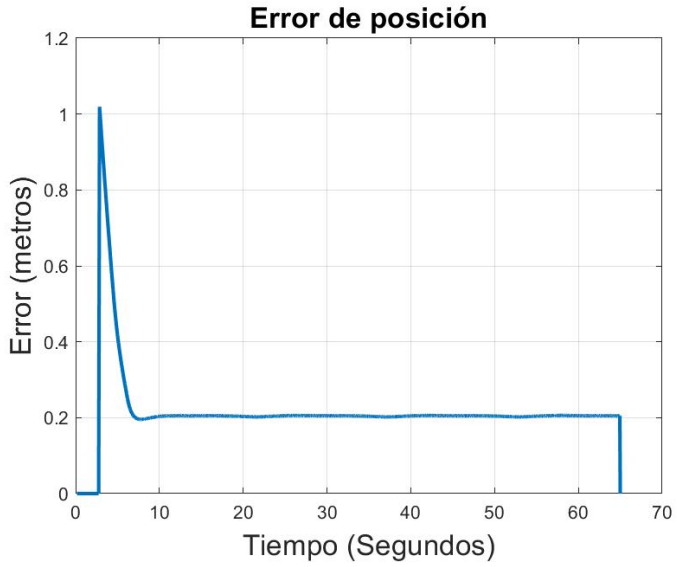

Figura 2.30: Gráfica del error de seguimiento de trayectoria circular.

Del mismo modo,en la Figura 2.31 se presenta una prueba en la que se involucra la presencia de un obstáculo para la misma trayectoria circular. Como se puede apreciar, el robot es capaz de evitar la colisión, y retomar la trayectoria. En la Figura 2.32 se presenta el error de posición correspondiente a esta prueba, en ella se puede apreciar la presencia de dos picos de error, el primero se debe, al igual que en el caso anterior a que el robot no se encuentra en la posición de inicio de la trayectoria, mientras, el segundo, es debido a la presencia del obstáculo, con estos datos se obtuvo un valor ISE de 39.8.

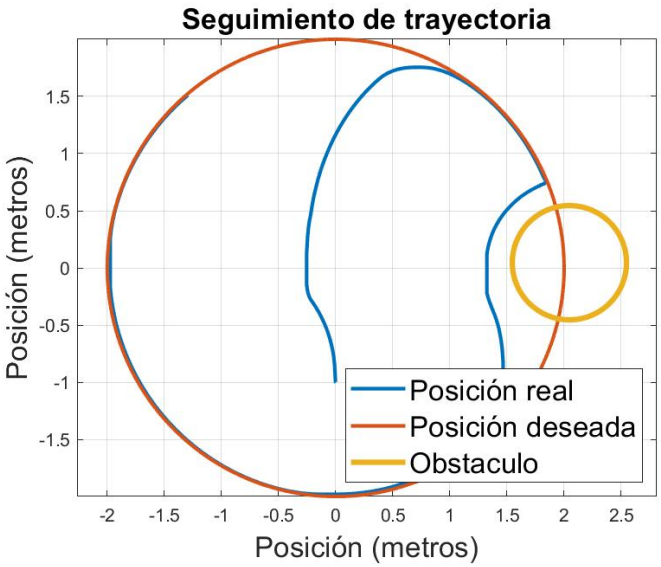

Figura 2.31: Muestra del seguimiento de trayectoria circular ante la presencia de obstáculos.

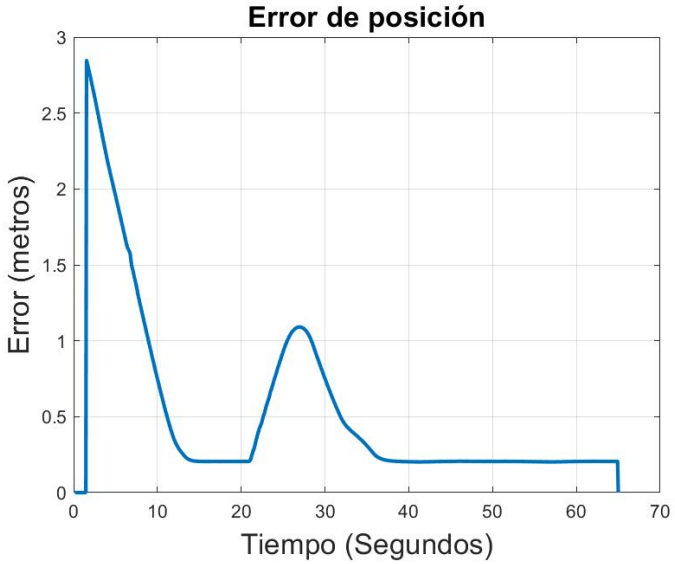

Figura 2.32: Gráfica del error de seguimiento de trayectoria circular ante la presencia de un obstáculo.

Para continuar, se realizaron pruebas con una trayectoria diferente, una lemniscata. En la Figura 2.33 se muestran las trayectorias real y deseada, tal como en el ejercicio anterior, del mismo modo presentamos su gráfica de error en la Figura 2.34 de la cual se obtuvo un valor ISE de 1.63.

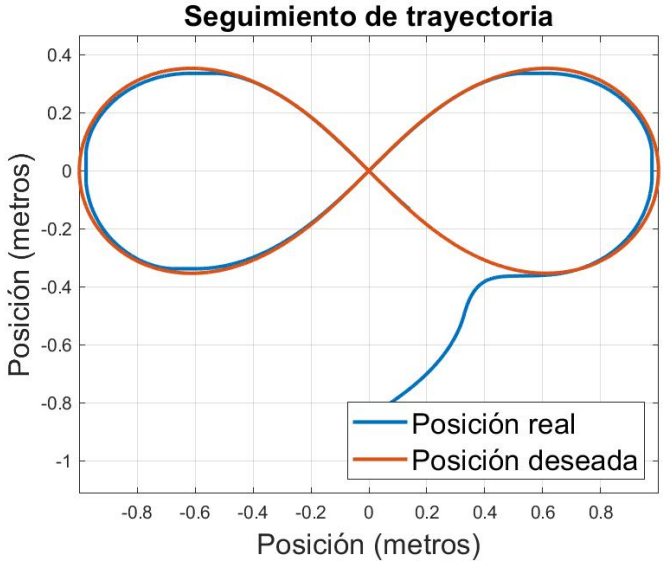

Figura 2.33: Muestra del seguimiento de trayectoria lemniscata sin obstáculos.

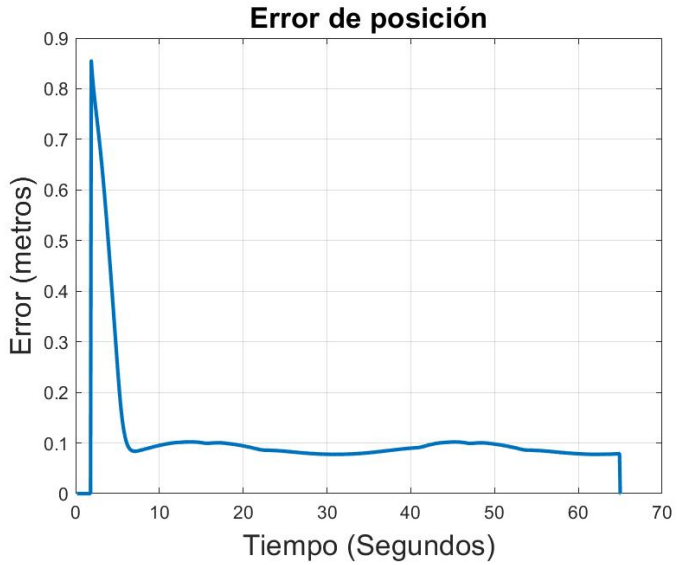

Figura 2.34: Gráfica de error de posición para trayectoria lemniscata.

Finalmente presentamos la prueba realizando la trayectoria anterior, con un obstáculo presente en las coordenadas (1.3,0.2), tal como se ilustra en la Figura 2.35 y la gráfica de error correspondiente en la Figura 2.36, de donde se obtuvo un valor ISE de 6.57.

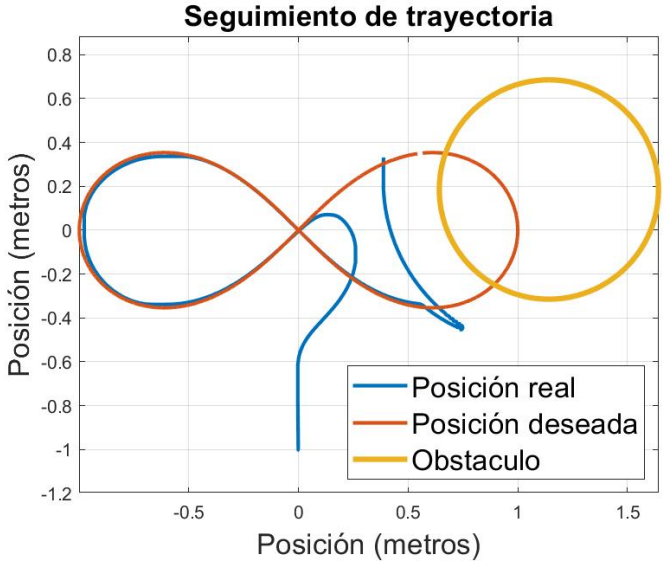

Figura 2.35: Muestra del seguimiento de trayectoria lemniscata con un obstáculo.

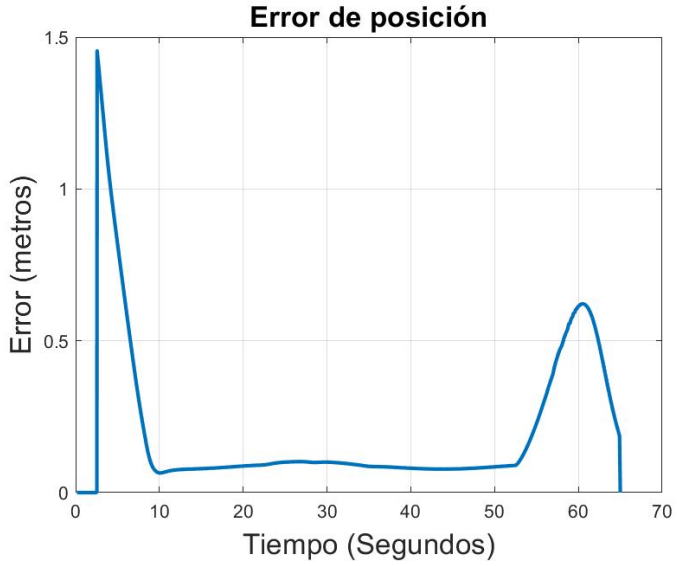

Figura 2.36: Gráfica de error de posición para trayectoria lemniscata con obstáculo.

2.10.2. Implementación con cálculos de odometría

Para las pruebas realizadas con el robot basadas en odometría, se designo un área de dos metros por dos metros, en la cual se colocaron marcas que señalan las distancias a medio metro y a un metro del origen como se muestra en la Figura 2.37.

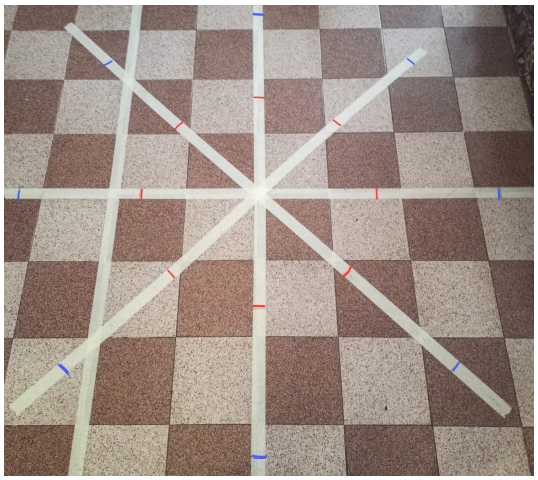

Figura 2.37: Área de trabajo; marcas de 1 metro en azul; marcas de medio metro en rojo.

En el área de trabajo antes mostrada se condujeron pruebas para los algoritmos de seguimiento de trayectorias, los cuales se presentan a continuación.

En la Figura 2.38 se muestra los resultados del seguimiento de la trayectoria circular que tiene medio metro de radio, en esta imagen se muestran tanto la trayectoria deseada (en rojo) y la trayectoria real (en azul) con lo que se puede corroborar que el seguimiento es satisfactorio.

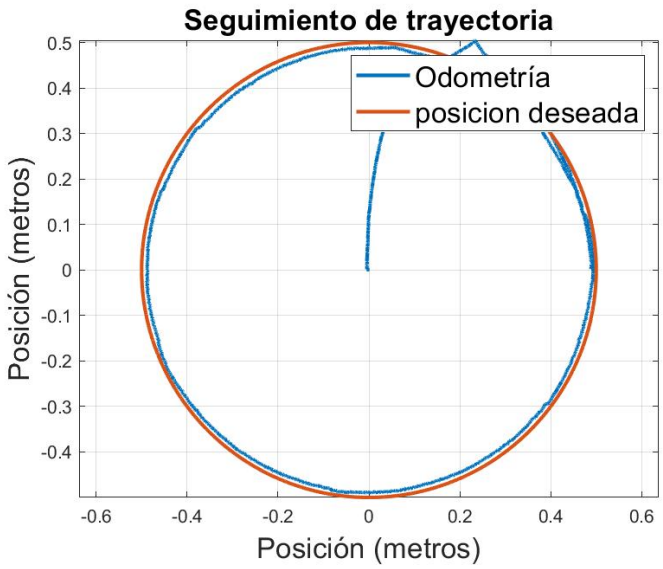

Figura 2.38: Seguimiento de trayectoria circular con medio metro de radio.

En la Figura 2.39 se muestra el valor del error calculado durante el seguimiento de dicha trayectoria, y a partir de estos datos se obtuvo un valor de ISE de 7.4724.

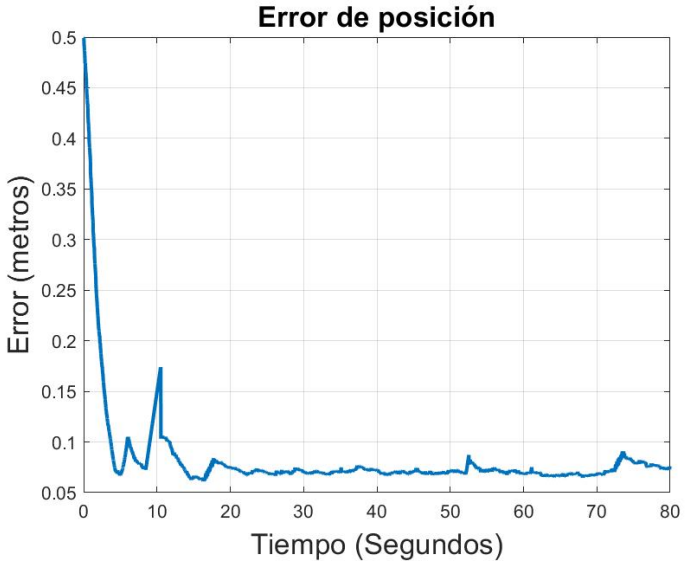

Figura 2.39: Error de posición durante el seguimiento de la trayectoria circular.

Del mismo modo, se realizo el seguimiento de una lemniscata, como se puede apreciar en la Figura 2.40.

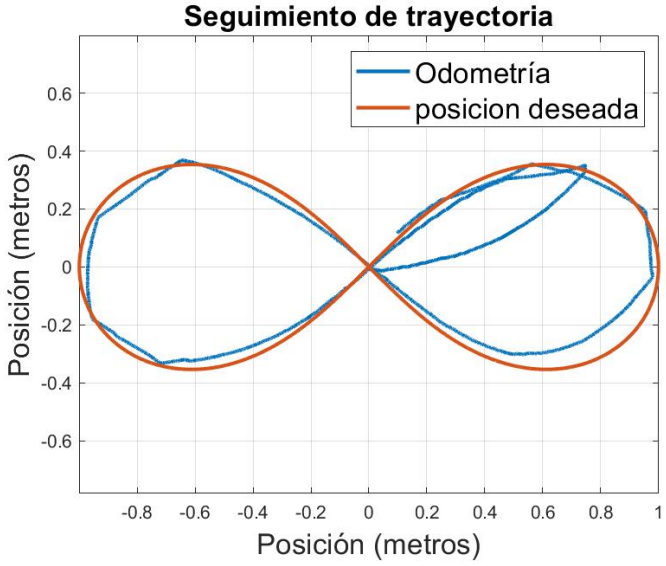

Figura 2.40: Seguimiento de trayectoria lemniscata.

En la Figura 2.41 se puede apreciar el error de posición durante el seguimiento de la trayectoria lemniscata, con lo que se obtuvo un valor de ISE de 70.7802.

Figura 2.41: Error de posición durante el seguimiento de la trayectoria lemniscata.

2.10.3. Implementación con *OPTITRACK*

Por medio del uso de las cámaras *OPTITRACK* se obtiene la posición real del robot dentro del área de trabajo.

En las siguientes pruebas, se programó directamente la tarjeta de cómputo empotrado con el controlador, de modo que esta adquiere los datos de posición por medio de la red de ROS desde la cámara *OPTITRACK*, cerrando así el bucle de control. Al mismo tiempo, en una instancia de *MATLAB*, se obtienen tanto la posición deseada como la real, por medio de la red de *ROS*.

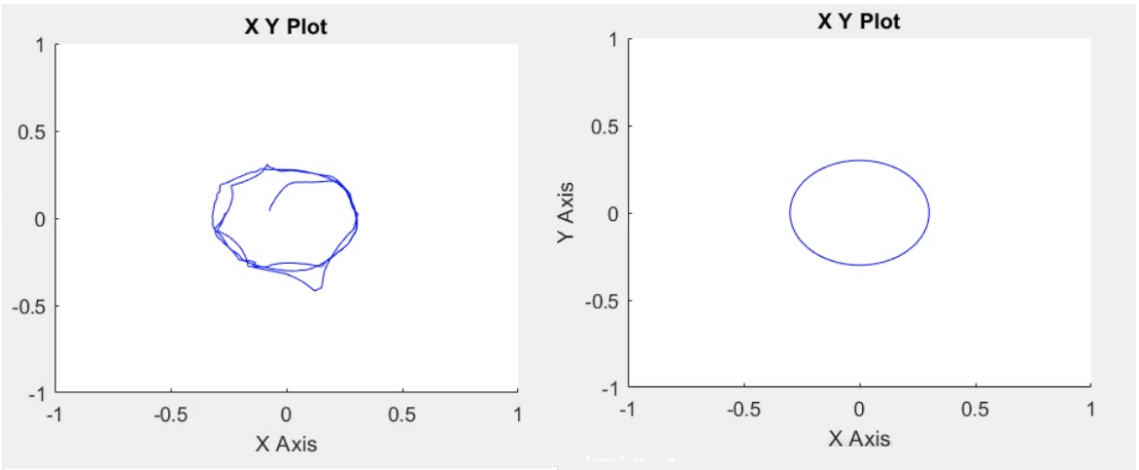

Figura 2.42: Muestra del seguimiento de trayectorias,robot real (lado izquierdo) y trayectoria calculada (lado derecho).

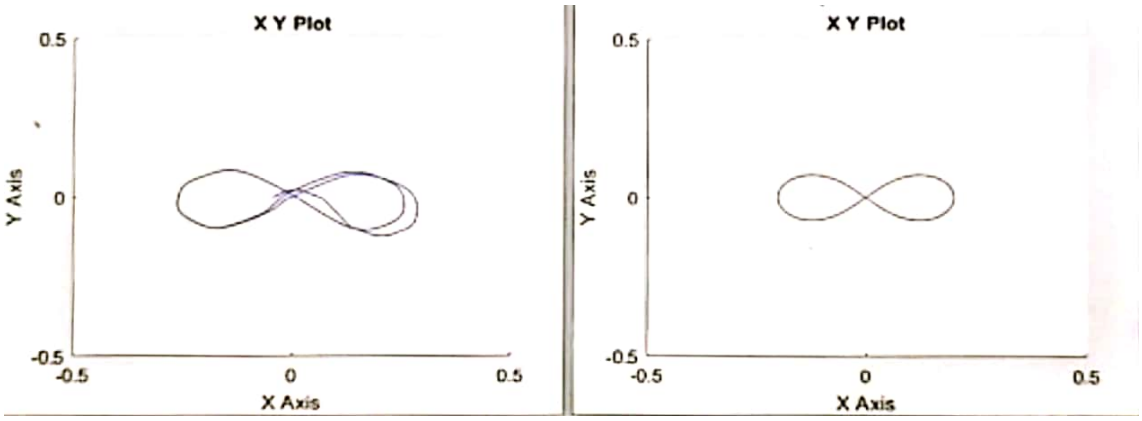

Figura 2.43: Muestra del seguimiento de trayectorias, robot real (lado izquierdo) y trayectoria calculada (lado derecho).

Capítulo 3

SLAM (*"Simultaneous Localization and Mapping"*)

En el presente capítulo se abordara la problemática del SLAM, describiendo algunos de los algoritmos más utilizados hoy en día, posteriormente se presentan argumentos para la elección de los algoritmos implementados así como las implementaciones tanto en simulación como en el sistema real. Finalmente se presentan los resultados experimentales de los algoritmos implementados.

3.1. Algoritmos en la literatura

En la actualidad existe una infinidad de algoritmos desarrollados para la solución del problema SLAM, desafortunadamente no todos ellos cuentan con una implementación en la plataforma ROS, por lo que, en la presente sección se realiza un estudio de los principales algoritmos disponibles. continuación se presentan dos algoritmos para la creación de mapas bidimensionales al igual que dos algoritmos para la creación de mapas tridimensionales.

3.1.1. HectorSLAM

Este algoritmo de SLAM combina un sistema de SLAM 2D basado en el emparejamiento robusto de escaneos (scan matching) y una técnica de navegación 3D que utiliza un sistema de detección inercial [29].

Los autores se han centrado en la estimación del movimiento del robot en tiempo real, aprovechando la alta tasa de actualización y el bajo ruido de las mediciones de distancia de los modernos LIDAR. No se utiliza la información odométrica, lo que da la posibilidad de implementar este enfoque en robots aéreos como UAVs o en robots terrestres que operan en terrenos irregulares. Por otro lado, podría tener problemas cuando sólo se dispone de

escaneos de baja velocidad y no se aprovecha cuando las estimaciones odométricas son bastante precisas.

La estimación de la pose en 2D se basa en la optimización de la alineación de los puntos finales del haz con el mapa obtenido hasta el momento. Los puntos finales se proyectan en el mapa real y se estiman las probabilidades de ocupación. El emparejamiento de los escaneos se resuelve mediante una ecuación de Gauss-Newton, que encuentra la transformación que mejor ajusta los rayos láser con el mapa. Además, se utiliza una representación cartográfica multirresolución, para evitar quedarse atascado en mínimos locales. Por último, la estimación de estado 3D para el filtro de navegación se basa en el EKF. Sin embargo, esto sólo es necesario cuando hay una unidad de medición inercial (IMU), como en el caso de los robots aéreos. Por lo tanto, no se utilizará en este trabajo.

3.1.2. Gmapping

Gmapping es un algoritmo de SLAM basado en sensores láser, tal y como se describe en [30]. Además, es el paquete de SLAM más utilizado en los robots de todo el mundo. Este algoritmo ha sido propuesto por Grisetti et al. y es un enfoque SLAM basado en filtro de partículas Rao-Blackwellized. La familia de algoritmos de filtros de partículas suele requerir un elevado número de partículas para obtener buenos resultados, lo que aumenta su complejidad computacional. Además, el problema de agotamiento asociado al proceso de remuestreo de los filtros de partículas disminuye la precisión del algoritmo. Esto ocurre porque los pesos asociados a las partículas pueden llegar a ser insignificantes. Por lo tanto, esto significa que hay una pequeña probabilidad de que la hipótesis correcta pueda ser eliminada.

En [30] se ha desarrollado una técnica de remuestreo adaptativo que minimiza el problema de agotamiento de partículas, ya que este proceso sólo se realiza cuando es necesario. Los autores también propusieron una forma de calcular una distribución precisa teniendo en cuenta no sólo el movimiento de la plataforma robótica, sino también las observaciones más recientes. Esto disminuye la incertidumbre sobre la postura del robot en el paso de predicción del filtro de partículas. Como consecuencia, el número de partículas necesarias disminuye, ya que la incertidumbre es menor, debido al proceso de emparejamiento de escaneos (scan matching).

En la Figura 3.1 se presenta un diagrama de flujo del algoritmo de Gmapping, dicho proceso es ejecutado para cada escaneo de entrada.

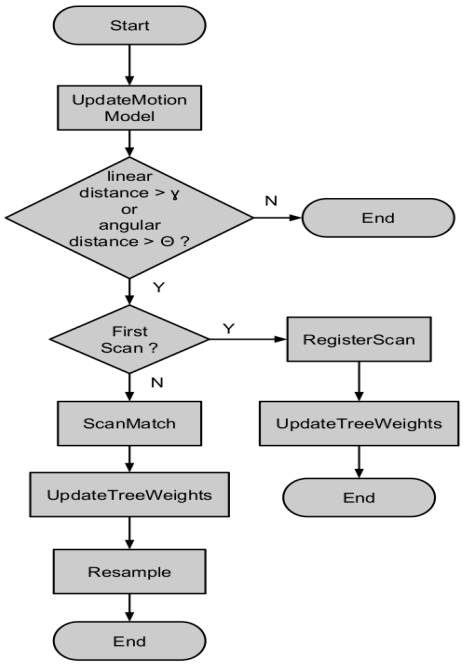

Figura 3.1: Diagrama de flujo de la función de procesamiento de escaneo del algoritmo Gmapping [5].

3.1.3. RGB-D SLAM

Este es uno de los primeros algoritmos de SLAM presentados en los que se emplean las imágenes en color y de profundidad de las cámaras RGB-D; comparado con otros métodos es uno de los más fiables y robustos. En general, el algoritmo desarrollado en este capítulo consta de cuatro pasos:

- Extracción de las características y landmarks de las imágenes obtenidas por la cámara RGB-D.
- Comparación de las características extraídas con las obtenidas en anteriormente.
- Una vez obtenido el conjunto de correspondencias, se utiliza un algoritmo RANSAC para estimar la transformación relativa entre los pares de imágenes obtenidas.
- Optimización del grafo global.

El algoritmo RANSAC, consiste en un método iterativo que puede ser utilizado para extraer líneas rectas de las lecturas de un sensor de distancia. De este modo se consigue distinguir líneas rectas dentro de un entorno en tres dimensiones, como las que conforman las paredes, techos, suelos y muebles que pueden encontrarse en un ambiente cerrado. Del mismo modo, se emplea el algoritmo ICP en el método RGB-D SLAM. Es utilizado para minimizar las diferencias entre dos nubes de puntos. Consiste en un proceso iterativo basado en la asociación de puntos utilizando el criterio del vecino más cercano.

3.1.4. RTAB-map

Uno de los aspectos mas importantes dentro de las técnicas de SLAM es la capacidad de reconocer localizaciones conocidas. El proceso aquí descrito es capaz de determinar si la observación actual es nueva o pertenece a una localización que ya fue visitada anteriormente, a esta función se le conoce como "Detección de cierre de bucle" (*Loop closure detection*).

Generalmente, la detección de cierre de bucle se realiza comparando la observación actual con todas las almacenadas en memoria, independientemente de la posición estimada del robot. Si al buscar entre las localizaciones conocidas no encontramos ninguna que coincida con la actual, ésta última posición se añade a la base de datos como nueva localización. Sin embargo, si el robot realiza un recorrido muy largo y añade demasiadas observaciones nuevas, el tiempo de cálculo aumenta dramáticamente.conforme el tiempo de cálculo supere al de adquisición, las imágenes comenzarán a presentar retrasos que dificultarán la construcción del mapa en tiempo real.

La técnica RTAB-map (Real-Time Appearance-Based Mapping) pretende de realizar SLAM en tiempo real, sin importar si el recorrido del robot es demasiado extenso o que el tiempo de la misión sea elevado.

En esta aproximación al SLAM se plantea una estrategia de particionado de memoria, donde las principales particiones de memoria son la memoria de trabajo del robot (*Working Memory*), la memoria a largo plazo (*Long Term Memory*), la memoria a corto plazo (Short-term memory) y la memoria sensorial (*Sensory Memory*). De este modo, se mantendrán en la memoria de trabajo del robot aquellas localizaciones que se han visitado recientemente y con más frecuencia, mientras que el resto pasarán a la memoria de largo plazo, como se ilustra en la Figura 3.2.

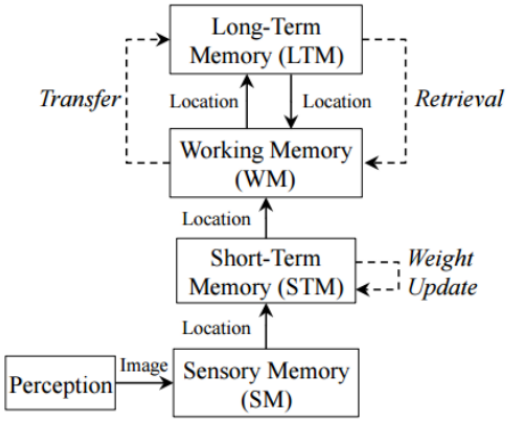

Figura 3.2: Tratamiento de la memoria en el algoritmo RTAB-map(reproducido de [6]).

3.2. Selecciona de algoritmos a implementar

Debido al uso de una cámara de profundidad *Realsense r200* y un LIDAR RPLIDAR A2 se estableció el uso de 2 algoritmos diferentes para probar, estos se seleccionaron debido a su desempeño reportado en [31] y al uso que se le proporciona, además del soporte presente en comunidad de ROS.

Como algoritmo de SLAM con la cámara *Realsense R200* se seleccionó el algoritmo RTAB-map, debido a que este implementa imágenes de profundidad para la creación de un mapa tridimensional. Asimismo, gracias a una paquetería de presente en ROS es posible convertir dichas imágenes en una emulación de escáner láser, con lo que es posible aplicar el algoritmo Gmapping, el cual genera un mapa bidimensional.

Del mismo modo, dada la presencia de un LIDAR, se planteo el uso de Gmapping con este.

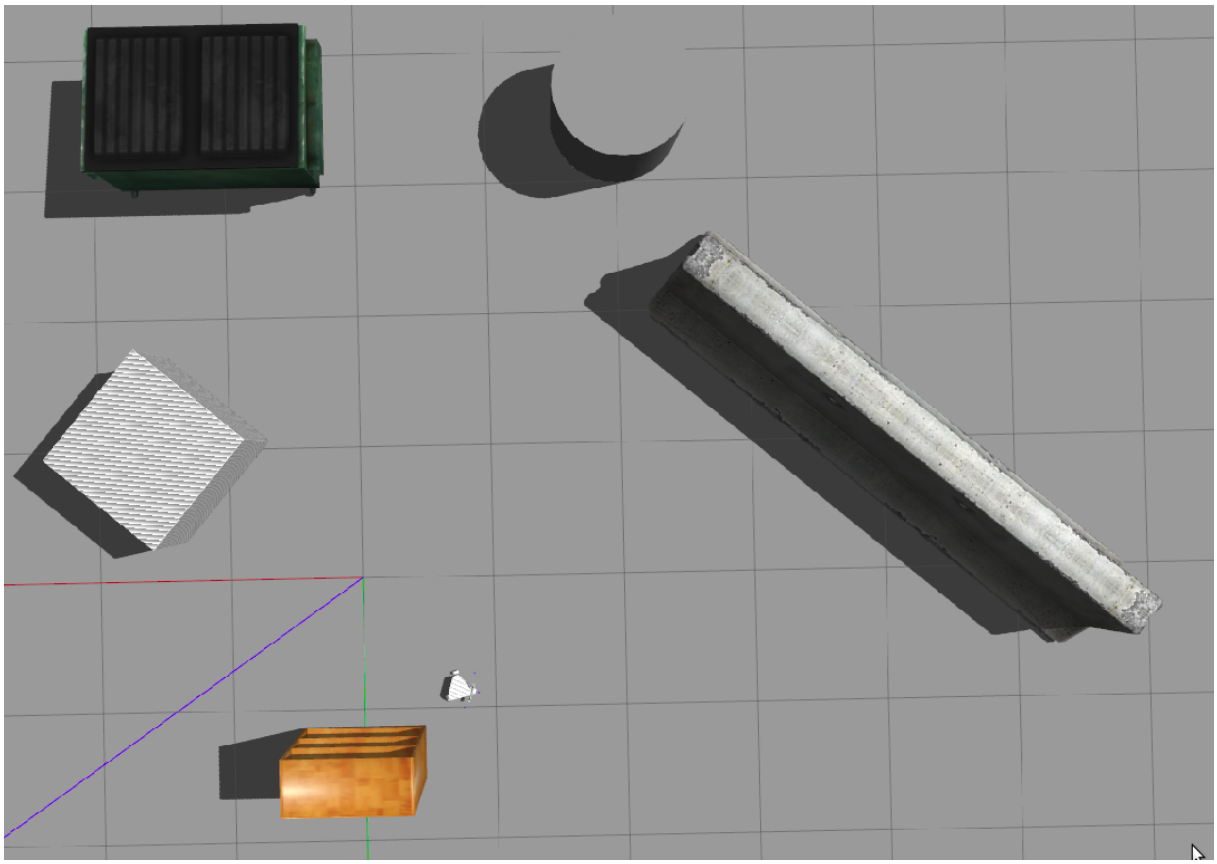

Figura 3.3: Ilustración del entorno de Gazebo.

3.3. Simulaciones

A continuación se presentan las simulaciones de SLAM realizadas, comenzando por la implementación de *Gmapping*, la cual, es una variación de SLAM, que tiene la bondad de

crear mapas bidimensionales, los cuales pueden ser usados para la navegación autónoma. Cabe destacar que este tipo de mapeo, a pesar de poder ser utilizado por medio de cámaras de profundidad, como se presenta aquí, es mayormente usado con sensores LIDAR, ya que cuentan con un mayor ángulo de visión.

Posteriormente abordaremos una implementación de RTAB-map, el cual, por medio de las imágenes de profundidad, genera una nube de puntos, a modo de realizar una reconstrucción tridimensional del entorno.

3.3.1. SLAM Gmapping

La simulación del robot se realizó de modo que fuese lo más parecido al robot real, incluso, se implementa una simulación de la cámara RGB-D *"Realsense R200"*.

Para ejecutar la simulación, debe ejecutarse en una terminal el siguiente comando:

Lanzamiento del entorno virtual

```
roslaunch delta_robot_description delta_robot_world.launch
```

Una vez hecho esto, podremos ver la interfaz de Gazebo, como se muestra en la 3.3. En ella podremos observar diversos elementos con los que interactuará el robot, también podemos modificar el entorno y guardarlo para diferentes experimentos.

Una vez hecho esto para poder mover al robot en el entorno, podemos ejecutar el comando:

Nodo de operación remota

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Con esto, nos es posible desplazarnos utilizando las teclas de la siguiente manera:

Instrucciones para el manejo del robot

```
Reading from the keyboard  and Publishing to Twist!
---------------------------
Moving around:
u    i    o
j    k    l
m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
U    I    O
J    K    L
M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
```

Ahora que nos es posible desplazarnos por el entorno, podemos saber como cambian las entradas y salidas del robot, para lo que utilizaremos *"Rviz"*, este se ejecutará de la siguiente manera.

Inicio de Rviz

```
roslaunch delta_robot_description view_robot_rviz.launch
```

Cómo se puede apreciar, la imagen de profundidad, en Rviz, es interpretada como un mapa de calor, por lo que los objetos más cercanos tendrán color rojo, y a medida que se alejen tendrán un color azul.

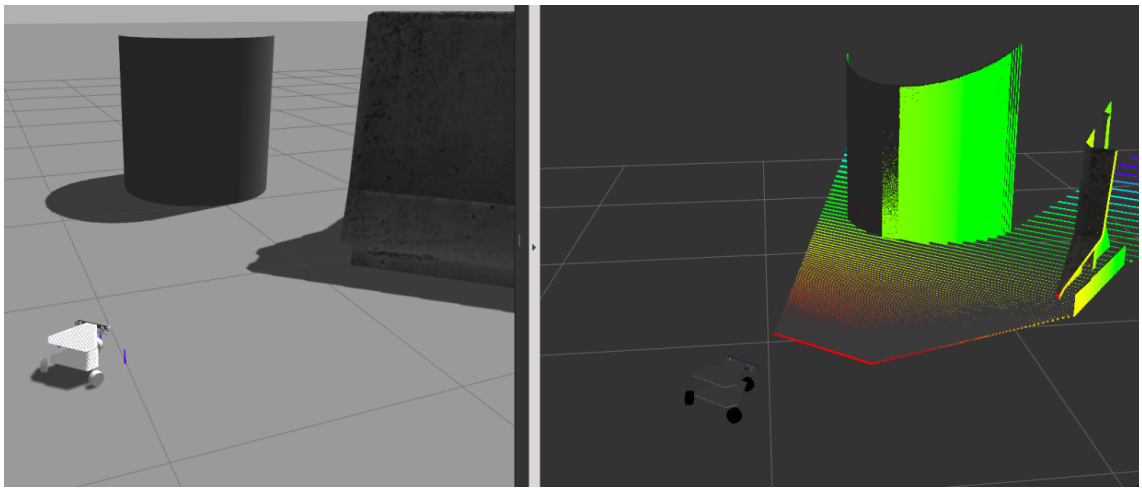

Figura 3.4: Visualización del entorno Gazebo (izquierda) y Rviz(derecha)

Debido a que, para la generación de mapas de ocupación es necesario el uso de sensores LIDAR, que obtengan la información de distancias en una misma altura, pero dada la ausencia de este en nuestro sistema, haremos uso de un nodo, el cual se denomina *"depth_image_to_laser_scan"*, este nodo se encargara de generar una simulación de sensor LIDAR, como se muestra en la Figura 3.5

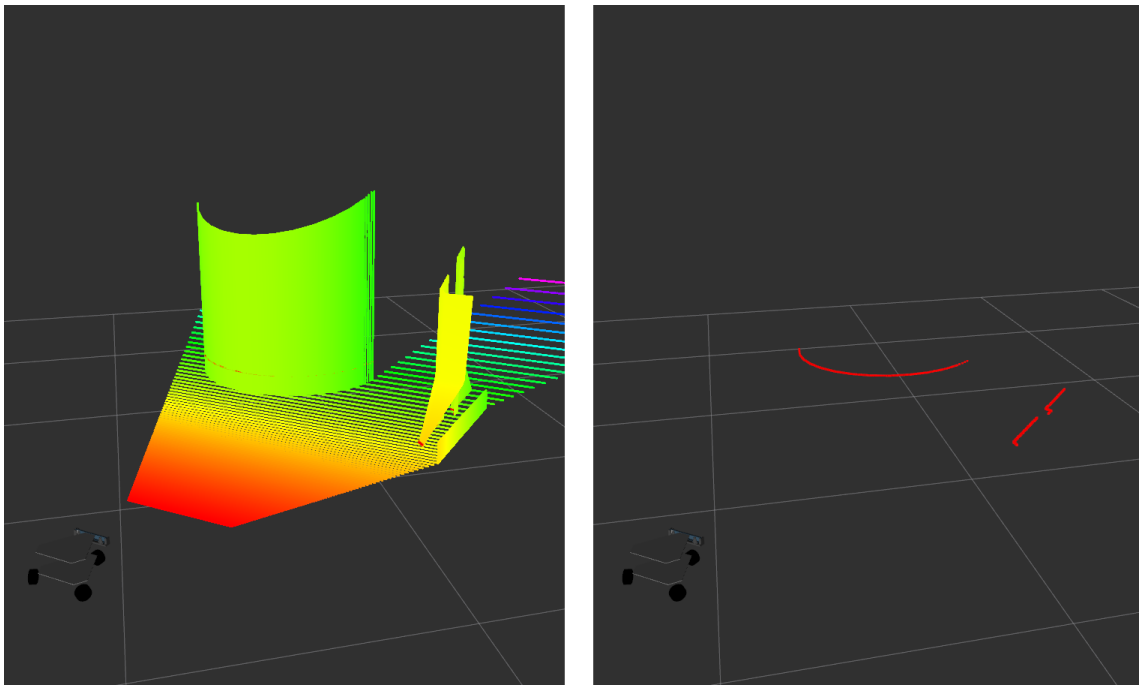

Figura 3.5: Visualización de la nube de puntos (izquierda), Visualización de el escaneo láser(derecha).

Como se puede apreciar, esta implementación tiene la desventaja de que, a diferencia de un sensor LIDAR real, este tiene un rango de visión mucho más corto, lo que limita su funcionamiento.

Una vez que podemos visualizar esta información de forma adecuada, podemos proceder a realizar un mapeo del área, lo cual se inicia por medio del comando:

Lanzamiento de nodos de mapeo

```
roslaunch delta_robot_description gmapping.launch
```

Posteriormente, podemos comenzar a realizar el mapeo del área, lo cual, se logrará, desplazando el robot por las diferentes áreas de interés, una vez que se obtenga un mapa satisfactorio como en la 3.6 del área deseada, procedemos a guardar dicho mapa, de la siguiente manera:

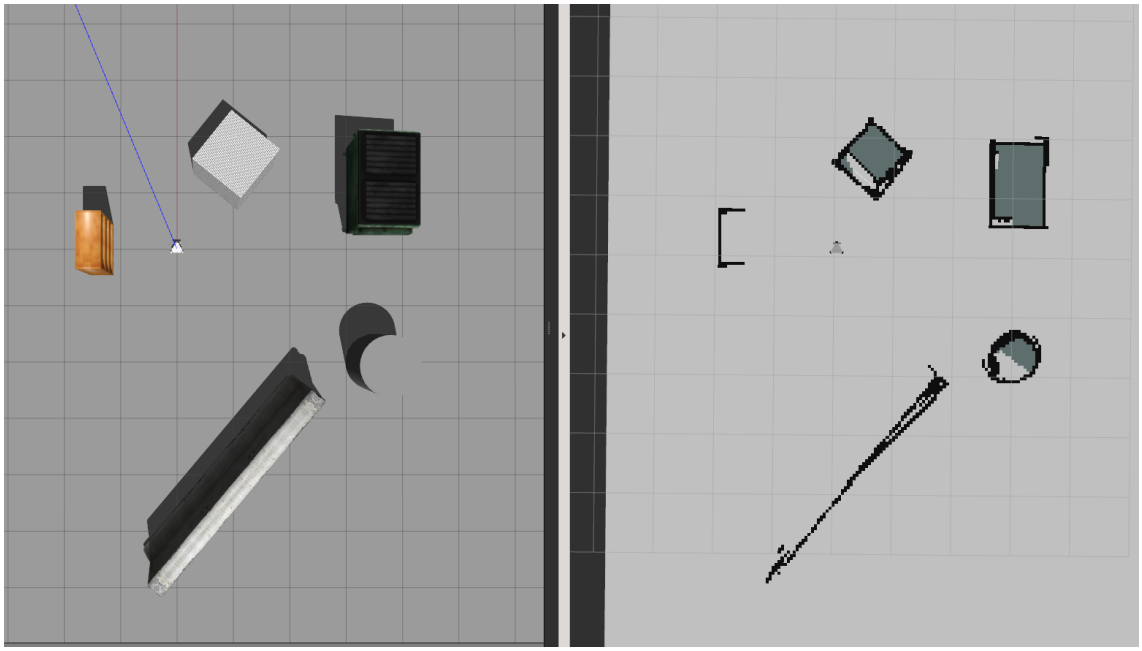

Figura 3.6: Visualización de mapa terminado.

Comando para guardar el mapa

```
rosrun map_server map_saver -f <nombre del archivo>
```

Por medio de este comando serán generados 2 archivos, el primero correspondiente al mapa, que se guarda en formato "pgm", y el segundo, donde se guardan los valores de resolución del mapa, origen y valores de umbral para la rejilla de ocupación.

Una vez hecho esto, podemos reiniciar el entorno, para posteriormente ejecutar la localización, la cual se llevará a cabo por medio del nodo AMCL, el cual es una implementación de "localización de Montecarlo".

Localización de Montecarlo

```
roslaunch turtlebot_gazebo amcl.launch map:=<ubicacción del
    mapa>
```

Finalmente, es posible realizar localización, de modo que la interfaz de Rviz lucirá de la siguiente manera.

Visualización del entorno con variables de localización y mapeo

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

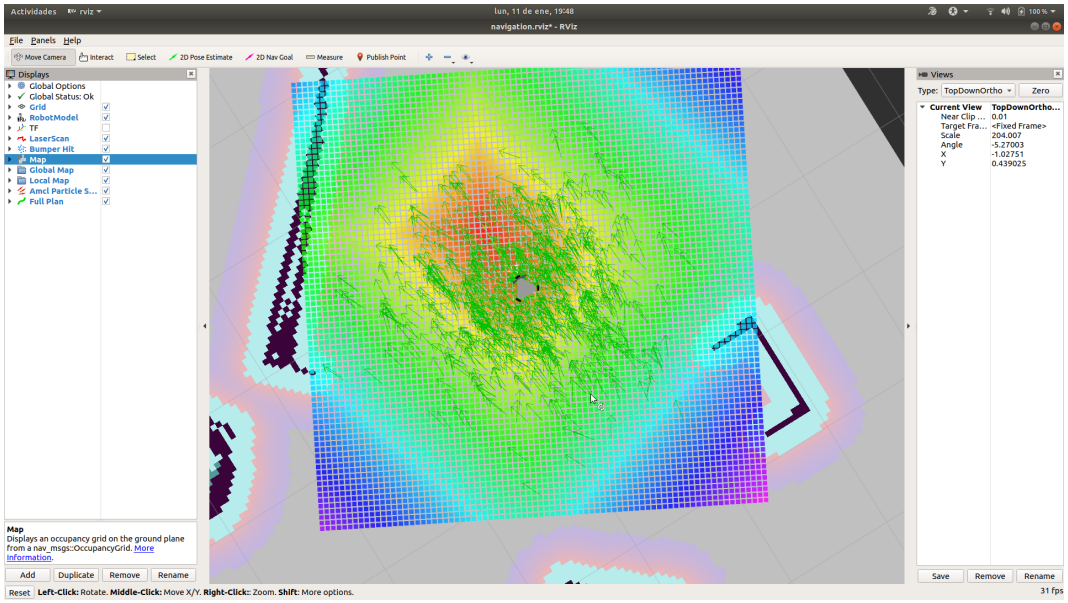

Figura 3.7: Rviz, mostrando la información del modulo de localización.

Cómo se puede apreciar, en la imagen 3.7, el modelo del robot está rodeado de flechas de color verde, éstas representan una posible ubicación, de acuerdo a las observaciones, por lo que, cuando se desplace el robot obtendrá más observaciones y podrá refinar el cálculo de esa aproximación, como se ilustra en la figura 3.8.

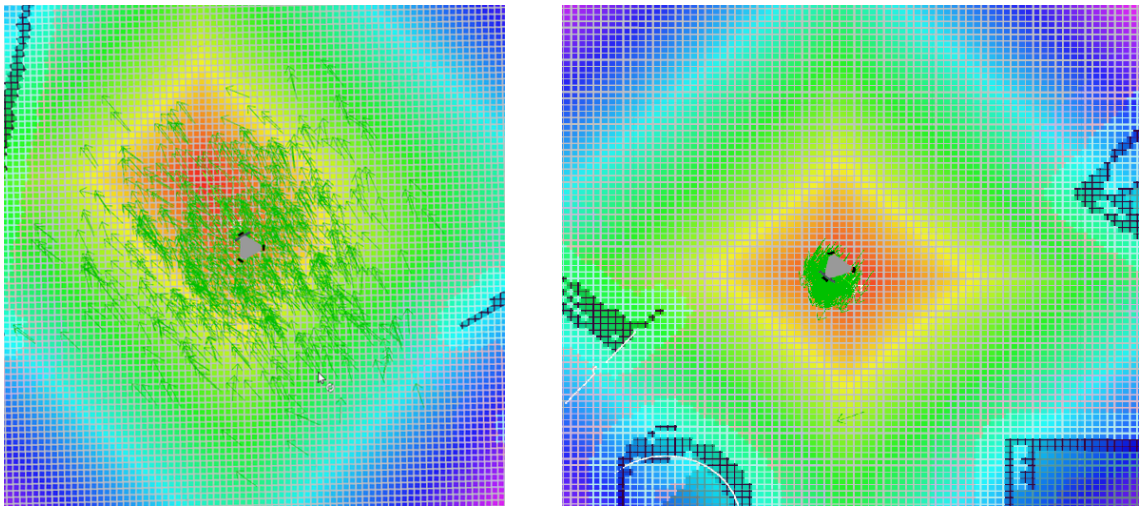

Figura 3.8: Refinación de la localización.

También, nos es posible especificar objetivos de navegación por medio de vectores en la interfaz de Rviz, una vez especificado el objetivo, se utiliza el cálculo de los mapas de ocupación, local y global, al evaluar la vecindad del robot, es posible asignar fuerzas atractivas, hacia los objetivos, y repulsivas, desde los obstáculos, de este modo se realiza la evasión de colisiones, todo esto se ilustra en la Figura 3.9.

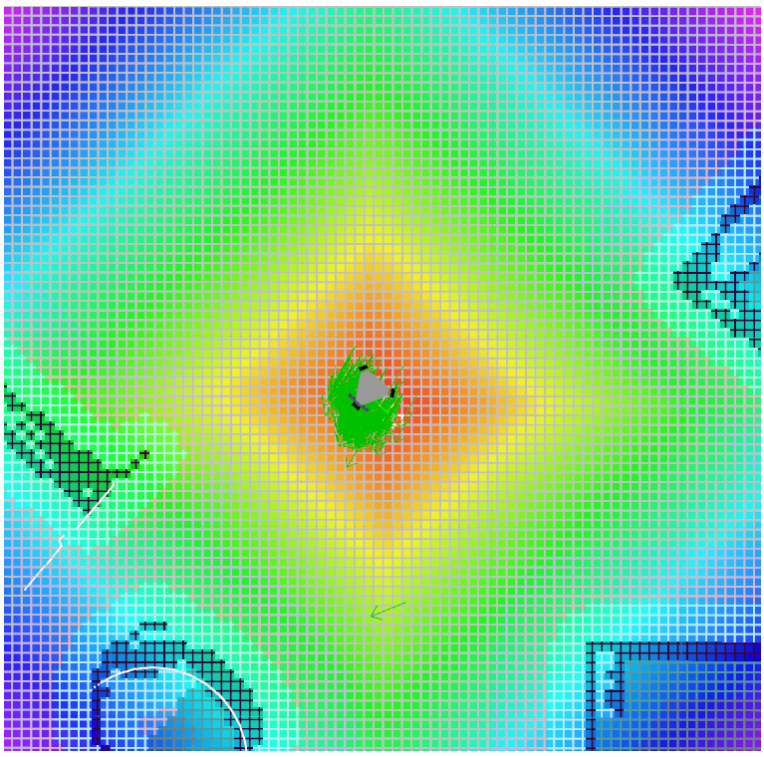

Figura 3.9: Ilustración del mapa de ocupación local superpuesto al global.

3.3.2. SLAM RTAB-map

La implementación de RTAB-map también se llevó a cabo de ambas formas, en simulación e implementación real, cabe destacar que debido a las limitaciones tanto de infraestructura, como del robot en si mismo, debido a una pobre odometría, se probaron dos aproximaciones, la primera, utilizando *"encoders"* integrados en el robot, para así calcular la odometría, y la segunda, haciendo uso de una odometría visual, la cual, utiliza puntos de interés en las imágenes RGB para calcular la ubicación y el desplazamiento del robot.

3.3.2.1. Simulación de RTAB-map por medio de odometría

Para esta implementación utilizaremos el mismo modelo del robot uniciclo presentado en la sección anterior, por lo que utilizaremos los siguientes comandos para iniciar el entorno.

Lanzamiento del entorno virtual

```
roslaunch delta_robot_description delta_robot_world.launch
```

Inicio de Rviz

```
roslaunch delta_robot_description view_robot_rviz.launch
```

Nodo de operación remota

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Una vez que los ambientes se estén ejecutando de manera adecuada, podemos ejecutar el siguiente comando, el cual, ejecuta los nodos del algoritmo RTAB-map.

Nodo RTAB-map

```
roslaunch rtabmap_ros rtabmap.launch \
        rtabmap_args:="--delete_db_on_start" \
        depth_topic:=/camera/depth/image_raw \
        rgb_topic:=/camera/rgb/image_raw rviz:=false \
        rtabmapviz:=false visual_odometry:=false odom_topic
            :=/odom
```

De este modo, a l igual que con la implementación de G-Mapping, es necesario que un usuario, realice el mapeo, haciendo circular el robot por el entorno, de este modo el algoritmo sera capaz de construir una nube de puntos, que represente el entorno como se muestra en la Figura 3.10.

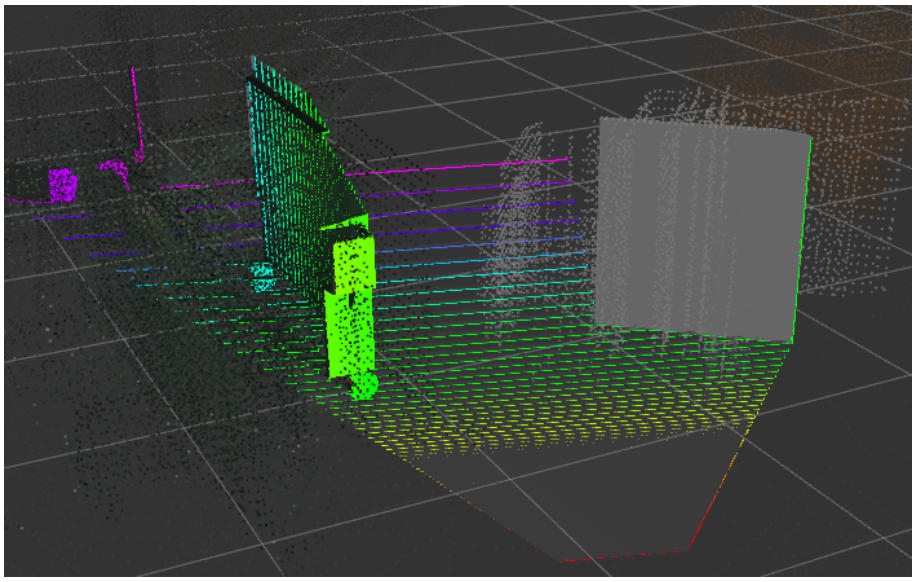

Figura 3.10: Visualización del procedimiento de mapeo en el entorno RVIZ.

Después de realizar un recorrido de el entorno, utilizando este método se obtuvo la reconstrucción como se muestra en la Figura 3.11, en esta misma imagen se puede apreciar también el recorrido realizado por el robot en color cían.

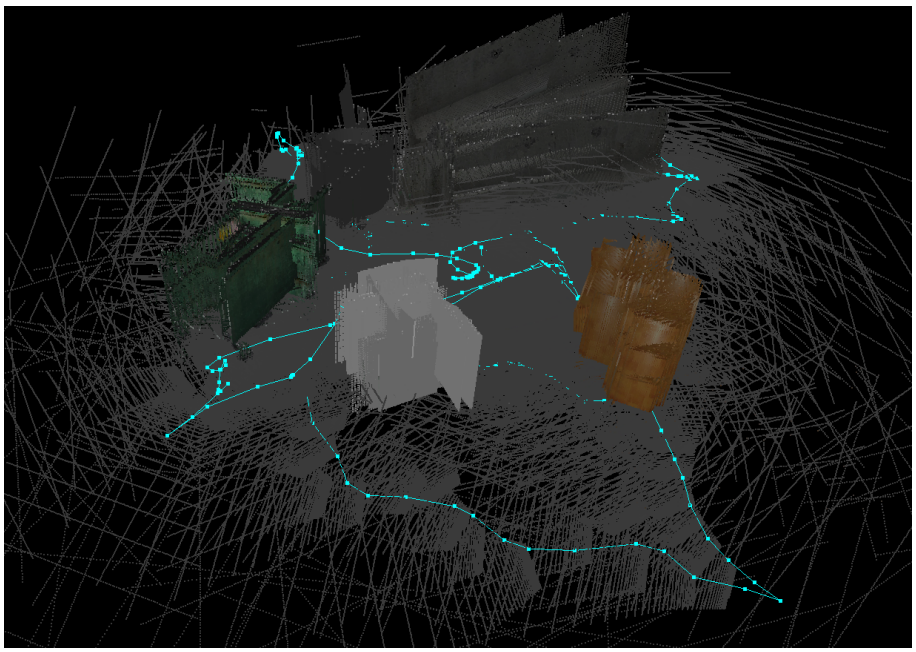

Figura 3.11: Visualización de la reconstrucción utilizando odometría.

3.3.2.2. Simulación de RTAB-map por medio de odometría visual

Al igual que en el caso anterior ejecutaremos los tres primeros comandos, para tener listo el entorno, posteriormente, ejecutaremos el siguiente comando.

Nodo RTAB-Map

```
roslaunch rtabmap_ros rtabmap.launch rtabmap_args:="--
    delete_db_on_start" depth_topic:=/camera/depth/image_raw
    rgb_topic:=/camera/rgb/image_raw \
rviz:=false rtabmapviz:=false
```

De igual forma, se realiza el mapeo, haciendo navegar al robot por el escenario, para obtener finalmente la representación del entorno como se muestra en la Figura 3.12.

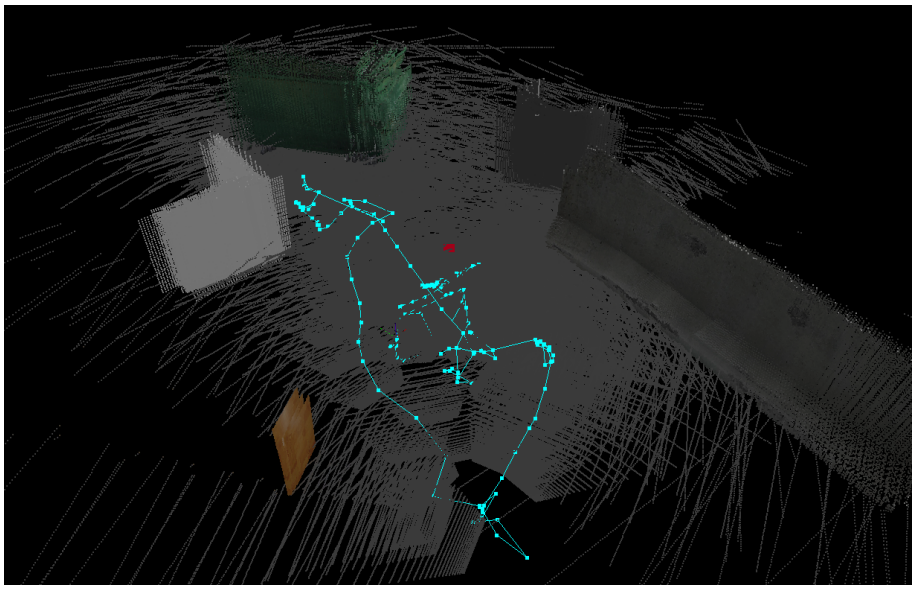

Figura 3.12: Visualización de la reconstrucción utilizando odometría visual.

3.4. Implementación

En esta sección se presentan los comandos necesarios para ejecutar los diferentes algoritmos de SLAM de acuerdo al sensor que se desee utilizar.

3.4.1. Implementación de Gmapping utilizando la cámara *Intel R200*

De forma similar a lo realizado en la sección anterior, la ejecución del algoritmo Gmapping requiere transformar las imágenes de profundidad a una emulación de escáner láser

o LIDAR, esto se logra por medio del comando:

Inicialización de la cámara Intel R200, emulación de escáner láser y control de robot (3,0)

```
roslaunch robot_delta delta_gmapping.launch
```

El comando anterior ejecuta el controlador de la cámara *Realsense R200* además la transformación de imágenes de profundidad a la emulación de escaneo láser, a su vez ejecuta también el controlador del robot en el nodo ros_serial.

una vez que este comando ha sido ejecutado podemos iniciar el proceso de mapeo, por medio del comando:

Ejecución del nodo de Gmapping

```
roslaunch robot_delta gmapping.launch
```

3.4.2. Implementación de Gmapping utilizando LIDAR RPLIDAR A2

Para la realización del mapeo por medio de Gmapping con LIDAR, es necesario ejecutar el comando:

Ejecución del nodo LIDAR RPLIDAR A2, control del robot (3,0) y ejecución del nodo de Gmapping

```
roslaunch robot_delta delta-gmapping_rplidar.launch
```

3.4.3. Implementación de RTAB-map por medio de odometría visual

Para la implementación en el hardware propuesto, resulta un poco diferente, ya que la tarjeta de cómputo empotrado es una computadora independiente con sistema linux Ubuntu, es necesario ejecutar los comandos por medio de una terminal remota (SSH), para esto es necesario que la computadora remota, desde donde estaremos ejecutando los comandos, y la tarjeta de cómputo empotrado, integrada en el robot, se encuentren en la misma red de área local. una vez hecho esto, la sesión se puede iniciar por medio del siguiente comando.

Inicio de sesión remota

```
ssh <nombre del usuario>@<direccion IP de la tarjeta>
```

Posterior a esto, para poder visualizar de forma remota los datos en la red de ROS, es necesario que en la computadora que se ejecute el núcleo de ROS, se exporten las variable ROS_IP y ROS_MASTER_URI, estas deberán tomar el valor de la IP de la computadora que ejecutará el núcleo, en nuestro caso, el robot. una vez exportadas las variables es necesario ejecutar el núcleo de ROS. esto se logra, en la sesión remota por medio de las siguientes instrucciones.

Exportación de variables e inicio del núcleo de ROS

```
export ROS_MASTER_URI='http://<direccion IP>:11311'
export ROS_IP=<direccion IP>
roscore
```

Ya que la tarjeta utilizada no cuenta con salidas PWM accesibles para el control de los motores, esto se implementó en una tarjeta Node MCU, la cual cuenta con interfaz WIFI, la cual también debe estar conectada a la misma red de área local para poder ser accesible a la red de ROS. Para lograr la comunicación entre la tarjeta Node MCU y la tarjeta de cómputo empotrado se ejecuta un nodo esta última, el cual recibirá los datos y los publicará a la red. Para ejecutar este nodo se hace uso del siguiente comando.

Inicio de ROS serial (comunicación con tarjeta Node-MCU)

```
rosrun rosserial_python serial_node.py tcp
```

El siguiente paso consiste en ejecutar el nodo que realizara la comunicación entre el hardware de la cámara "Intel Realsense R-200" y su la red de ROS.

Inicio de Intel realsense r200

```
roslaunch realsense_camera r200_nodelet_rgbd.launch
```

Una vez que todo se encuentre en ejecución podemos hacer uso del siguiente comando, el cual ejecutará el algoritmo de RTAB-map.

Inicio de RTAB-map

```
roslaunch rtabmap_ros rtabmap.launch \
        rtabmap_args:="--delete_db_on_start" \
        depth_topic:=/camera/depth/points/image_raw \
        rviz:=false rtabmapviz:=false
```

Finalmente, con todas las dependencias corriendo en la computadora remota, podemos dirigir el robot en el entorno, haciendo uso del mismo comando para manejar el robot por medio del teclado.

3.4.4. Implementación de RTAB-map por medio de odometría

Para esta última sección se utilizan los mismos comandos que la anterior, a excepciona del encargado de ejecutar el nodo de RTAB-map, el cual se sustituye por el siguiente.

Inicio de RTAB-map

```
roslaunch rtabmap_ros rtabmap.launch \
        rtabmap_args:="--delete_db_on_start" \
        depth_topic:=/camera/depth/points/image_raw \
        rviz:=false rtabmapviz:=false \
        visual_odometry:=false odom_topic:=/car/odom
```

Una vez hecho esto, se puede navegar con el robot por el área deseada para obtener la reconstrucción de esta.

3.5. Resultados experimentales

En esta sección se realiza la recopilación de los resultados obtenidos de las diferentes implementaciones de SLAM realizadas.

3.5.1. Implementación de Gmapping con cámara *Intel Realsense r200*

En esta implementación de Gmapping, como se puede apreciar el mapeo se complica debido al limitado ángulo de visión de la cámara *Realsense R200*, es por ello que requiere de un mayor tiempo de mapeo.

Como se puede apreciar en la Figura 3.13 se logra una buena reconstrucción del área de trabajo, la cual, conforme se continua con el proceso de mapeo, debido al error incremental de la odometría puede ocasionar deformaciones en el mapa.

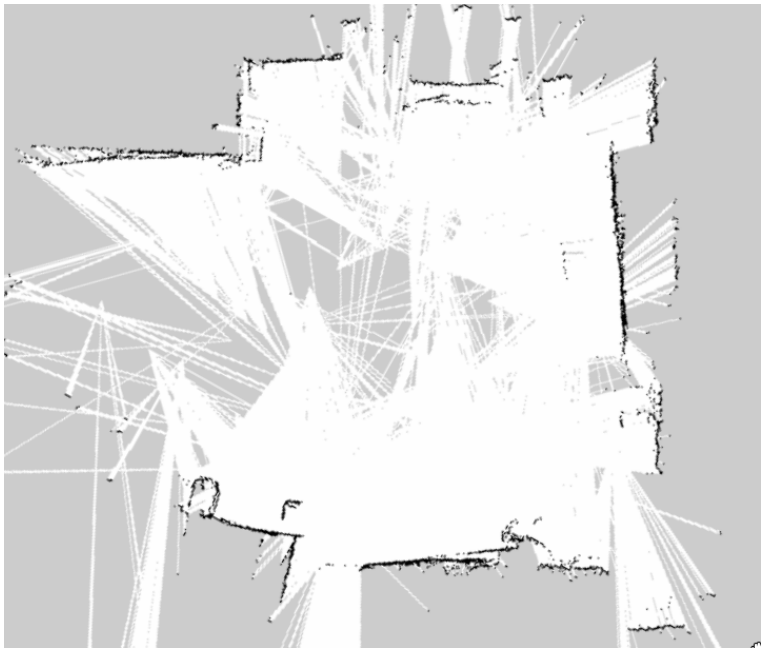

Figura 3.13: Resultados obtenidos con la cámara *Intel Realsense r200*.

3.5.2. Implementación de Gmapping con LIDAR RPLIDAR A2

En la presente implementación de Gmapping se implementó un LIDAR, de la marca RPLIDAR modelo A2, el cual cuenta con un rango de visión de 360°, con lo que se puede obtener una mejor representación del área de trabajo sin la necesidad de realizar recorrer repetidas veces el área de trabajo.

Como se puede apreciar en la Figura 3.14, se obtiene un mapa mejor definido en comparación al caso anterior.

Figura 3.14: Resultados obtenidos con LIDAR RPLIDAR A2.

3.5.3. Implementación de RTAB-map con odometría visual

En el caso de la odometría visual, se desprecia la odometría calculada abordo, y se estima la posición y el desplazamiento por medio de las imágenes adquiridas en la cámara RGB. en la Figura 3.15 se puede apreciar el mapa generado por medio de la odometría visual, en donde se aprecia en color cían la trayectoria calculada por esta,

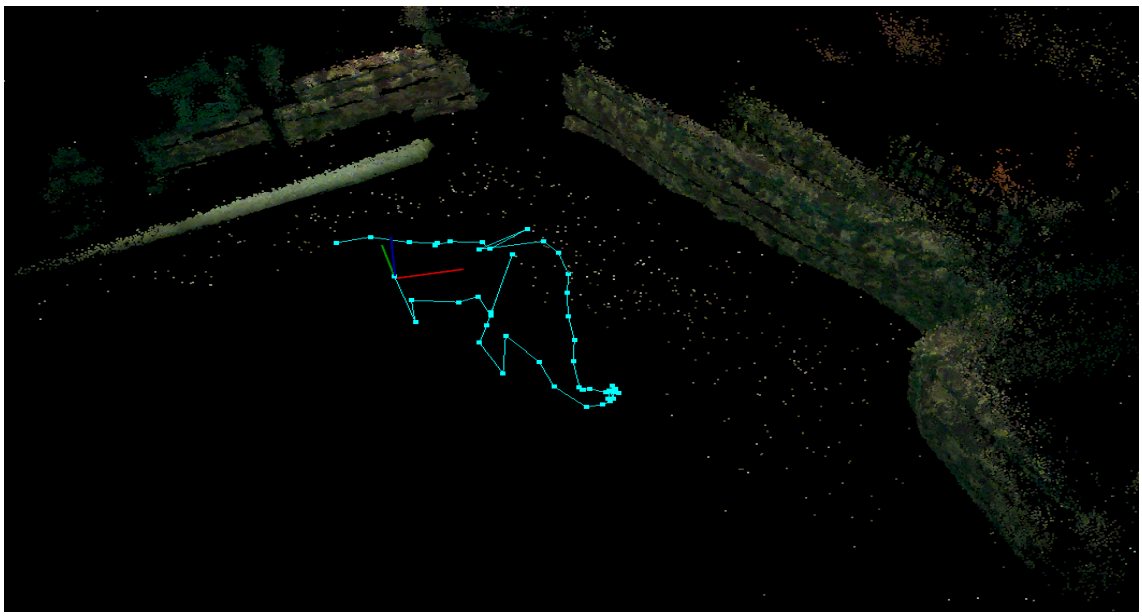

Figura 3.15: Resultados obtenidos por medio de odometría visual.

3.5.4. Implementación de RTAB-map con odometría

En el caso del mapeo con odometría, se aprovecha la odometría calculada abordo, con lo que se reducen errores presentes en el cálculo de la odometría visual, además de se evita la pérdida de sincronización debido a errores en el cálculo de la odometría visual los cuales pueden deberse a ligeros cambios en la iluminación de la escena.

En la Figura 3.16 se aprecia el mapa generado por medio de odometría visual, así como la trayectoria recorrida por el robot en el área de trabajo en color cían.

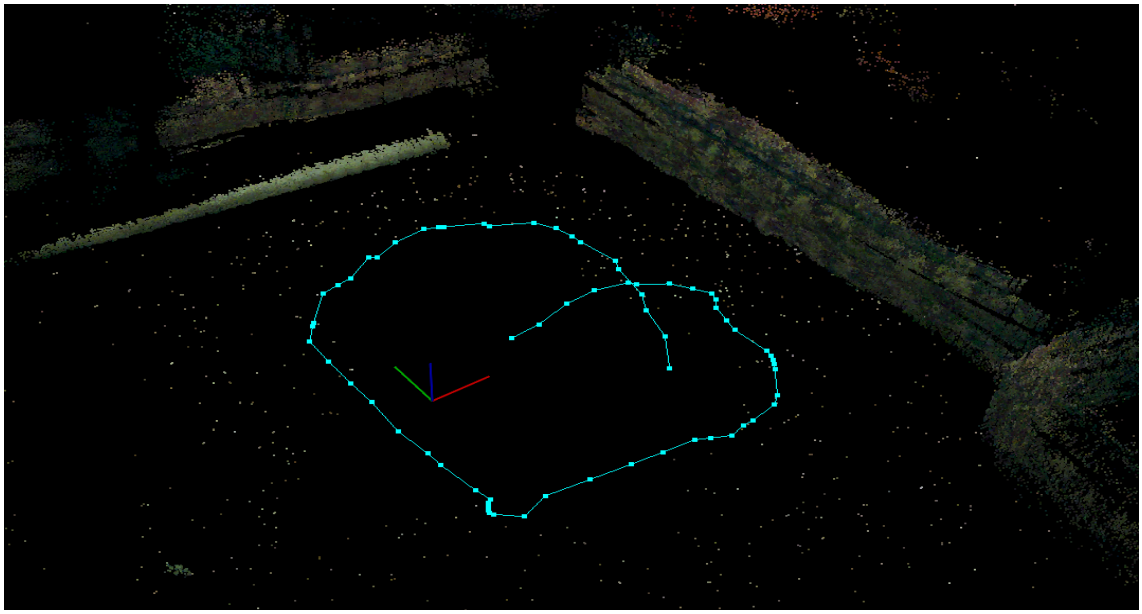

Figura 3.16: Resultados obtenidos por medio de odometría.

Capítulo 4

Trabajo paralelo

Debido a los diferentes inconvenientes a lo largo de la elaboración de esta tesis se optó por la implementación de un robot uniciclo para continuar con el desarrollo de los objetivos. Esto, en parte se logro gracias a las características de ROS, las cuales permitieron una alta compatibilidad del trabajo realizado con el objetivo.

4.1. Implementación de robot uniciclo

La implementación de este modelo se llevó acabo sobre una base de acrílico como la mostrada en la Figura 4.1.

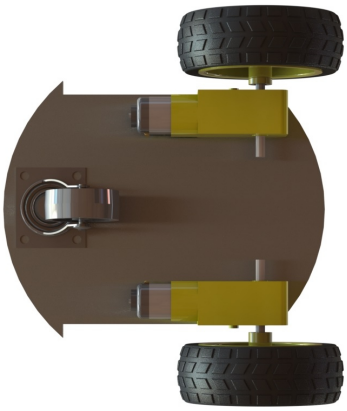

Figura 4.1: Base uniciclo.

La implementación del robot uniciclo se dividió en dos secciones,comenzando por el robot real y continuando con el modelo de simulación.

Para el desarrollo, tanto de la simulación, como del robot real, se hará uso de la herramienta *"Rviz"*, la cual, es una interfaz gráfica, por medio de la cual, se pueden presentar datos de forma visual, como lo es el modelo del robot, las transformaciones, las cadenas cinemáticas de las articulaciones, datos de sensores, como: radares láser, imágenes

de profundidad y RGB, entre otras. Por todo esto, se puede decir, que nos permite tener una perspectiva de las entradas y salidas de nuestro robot.

Implementación física del robot uniciclo

Modelado de motores para robot uniciclo

Para este caso, debido a el material disponible, se optó por realizar la caracterización por medio de un sensor de herradura, para poder realizar la lectura de velocidad se creó un disco de encoder de 16 muescas, el cual, debido al modelo del motor, fue anclado sobre el eje de la rueda véase Figura 4.2.

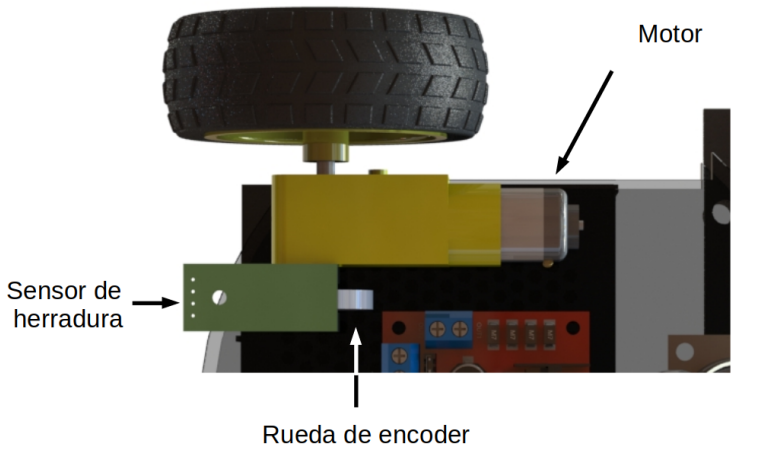

Figura 4.2: Ilustración del montaje del encoder.

Una vez hecho esto, por medio de una tarjeta node MCU esp8266 se adquirieron las señales para calcular la frecuencia de giro de los motores a $5V$, para así obtener la respuesta al escalón y calcular los parámetros del motor. Para lograr esto, se implementó el siguiente código a través de la interfaz de programación Arduino IDE.

Código de interrupción

```
void ICACHE_RAM_ATTR iEncoder() {
        iTiempoActual = millis();
        if (iTiempoActual != iTiempoAnterior)
        {
                iFrecc = 1000. / (iTiempoActual -
                   iTiempoAnterior);
                iTiempoAnterior = iTiempoActual;
        }
}
```

Con esto se obtuvo la respuesta al escalón para ese motor, como se muestra en la Figura 4.3, en ella se muestra en rojo la entrada escalón, en azul la respuesta obtenida a partir de la tarjeta Node MCU y finalmente la linea amarilla que es una filtración de la señal obtenida por medio de un promediador móvil.

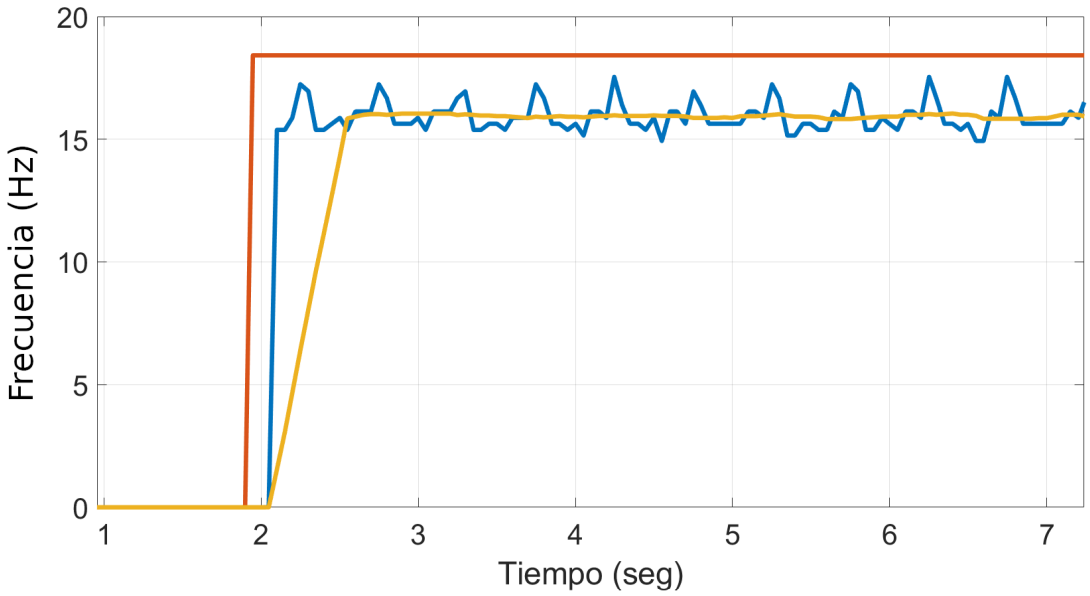

Figura 4.3: Ilustración de las señales obtenidas en el encoder. señal de control(rojo), lectura del encoder (azul), señal del encoder filtrada(amarillo).

Posteriormente se realizó la sintonización de los controladores PID, tal como en la sección 2.2.

Para este robot, se montó una estructura de acrílico, la cual, alberga todos los elementos necesarios para permitir al robot funcionar de forma autónoma, tal como se muestra en la Figura 4.4.

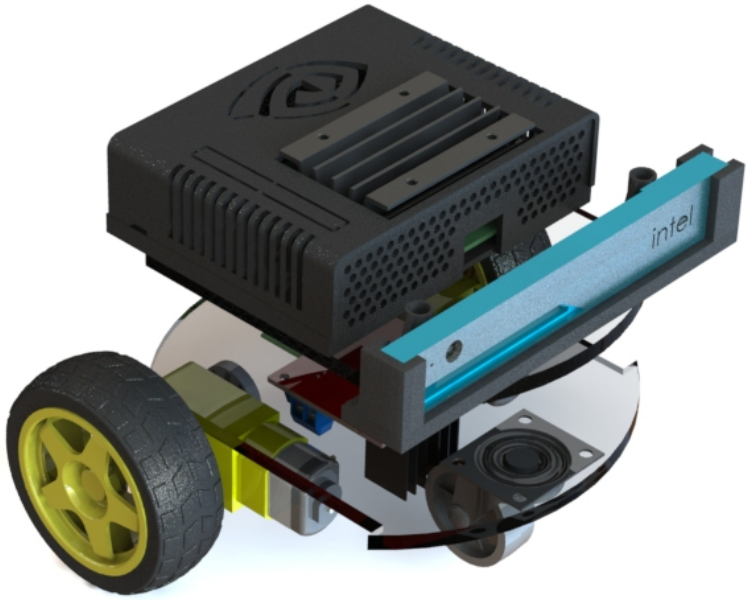

Figura 4.4: Ilustración del robot uniciclo con tarjeta Jetson NANO y cámara Realsense r200.

Al igual que en el caso del robot delta, fue necesaria la integración de una tarjeta Node-MCU,la cual, se conectó de acuerdo al diagrama que se presenta en la Figura 4.5. Además, se presenta el código utilizado en esta implementación en el Apéndice B.

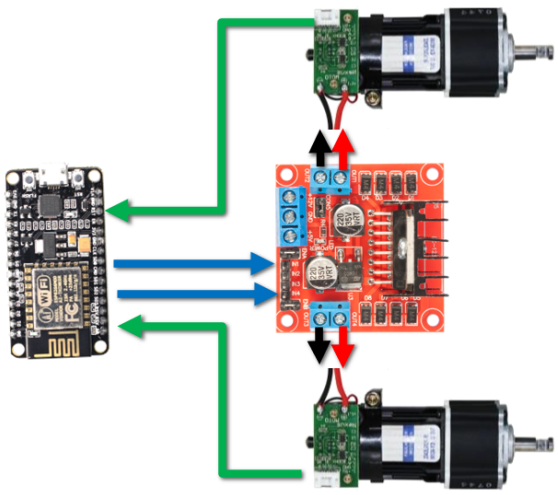

Figura 4.5: Diagrama de conexiones para robot uniciclo.

Para comprobar el correcto funcionamiento de la comunicación entre los nodos de la red se utiliza una aplicación de *"ANDROID"*. El diagrama de conexión en la red de ROS se conforma de la forma descrita en la imagen 4.6.

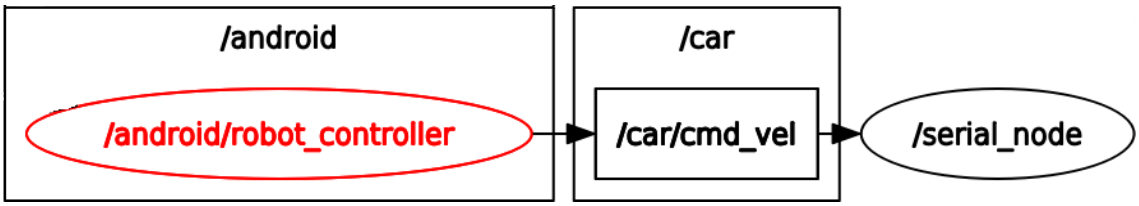

Figura 4.6: diagrama de conexiones de la red de ROS por medio de la cual se controla el carro de tipo uniciclo.

Dentro del nodo de ROS que se ejecuta en la tarjeta de desarrollo *"Node MCU"*, se realiza tanto el control de velocidad de los motores, gracias a los sensores de herradura, montados en la base, las señales que se adquieren son procesadas en un controlador *"PID"*, al mismo tiempo, se calcula la odometría a partir de las señales provenientes de los encoders, una vez calculada, es enviada devuelta a la red de ROS, esto con el fin de que el nodo maestro pueda adquirir la información de cambio de posición y así poder actualizar las visualizaciones y generar el mapa.

Simulación de robot uniciclo

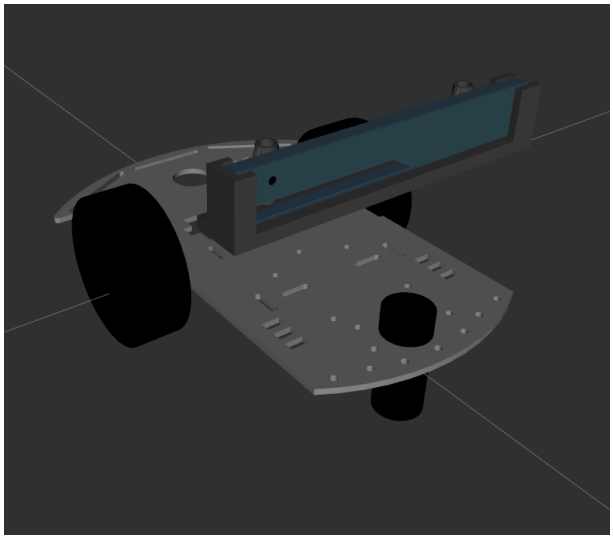

Figura 4.7: Visualización del robot uniciclo en la plataforma *"Rviz"*.

Similar a el caso del robot (3,0) ,en capítulos anteriores, este robot, fue descrito por medio de los archivos XML, con la diferencia de que su tercera rueda no es actuada, por lo que se le denomina como "*caster wheel*", la cual unicamente proporciona un punto de apoyo al robot.

De forma más especifica podemos dividir el árbol en dos partes, la primera 4.8, correspondiente a la base móvil y sus respectivas transformaciones, y la segunda 4.9, representa la cámara en el simulador.

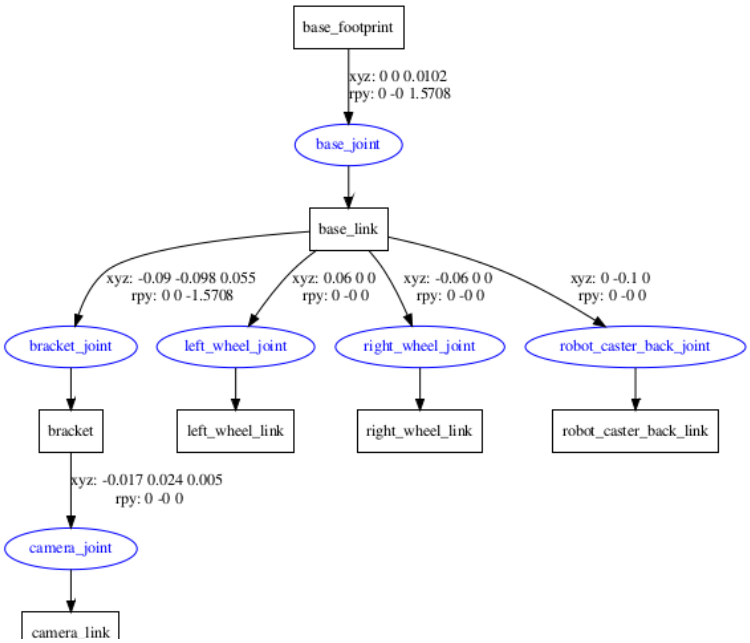

Figura 4.8: Vista general del árbol de transformaciones y relaciones mecánicas.

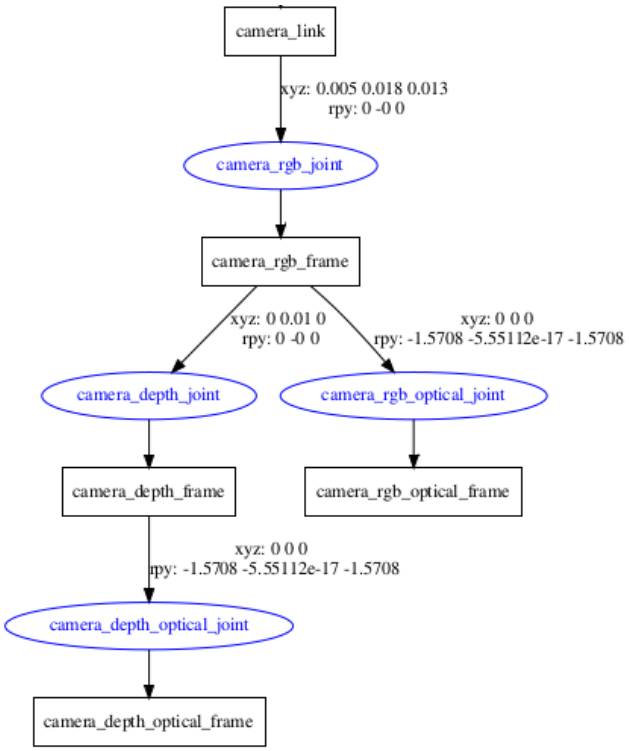

Figura 4.9: Vista general del árbol de transformaciones y relaciones mecánicas.

El procedimiento para ejecutar la simulación para este robot es similar al descrito en las secciones 3.3.1 y 3.3.2, con la diferencia que se requiere reemplazar la parte del comando “mobile_robot_description”por “delta_robot_description”.

También, debido a el cambio de la locomoción del robot, es necesario ejecutar un control diferente, el cual, restringe los movimientos posibles, esto se logra con el siguiente comando:

Visualización del entorno con variables de localización y mapeo

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Con esto, nos es posible desplazarnos utilizando las teclas de la siguiente manera:

Instrucciones para el manejo del robot

```
Moving around:
u    i    o
j    k    l
m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly
```

4.1.1. Resultados

Como es de esperarse, los resultados en las simulaciones no difieren. Mientras el desplazamiento se haga de una forma suave y controlada, es posible obtener resultados similares en ambos casos, Por otro lado, la implementación de robot físico, mostró un desempeño bajo, esto se debe a la odometría con la que se contaba, como se puede apreciar en la figura 4.10, la trayectoria parece irregular, y las imágenes no llegan a solaparse para generar un mapa.

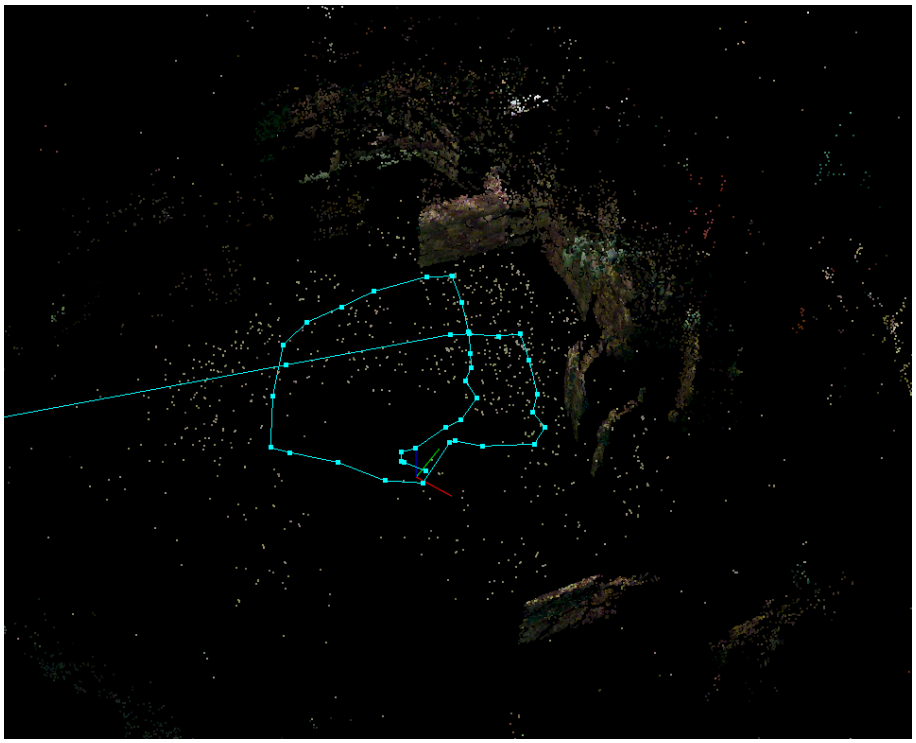

Figura 4.10: Ilustración del mapa obtenido con odometría.

Conclusiones

A lo largo del desarrollo de este proyecto se realizaron tres implementaciones distintas, en las cuales hubo diversos retos, desde la integración de los sensores, hasta la implementación de las leyes de control. Los cuales, se solucionaron de una manera más ágil gracias a las bondades de ROS, en comparación con la programación de un solo elemento que ejecute todas las tareas en un mismo programa. También, gracias a la estructura de ROS, es posible probar los mismos algoritmos en el simulador sin necesidad de realizar cambios.

El uso del simulador Gazebo presenta diferentes ventajas para la implementación de sistemas robóticos, pero la curva de aprendizaje resulta bastante prolongada debido a las distintas herramientas que se integran dentro de una implementación de este estilo.

El desempeño de las diferentes plataformas durante el seguimiento de trayectorias se considera satisfactorio, si se comparara con los datos de odometría, pero, ya que esta puede tener diferencias debido a el deslizamiento y otros factores físicos, tal es el caso del robot uniciclo presentado en el capítulo 4, el cual cuenta con una resolución de dieciséis señales por vuelta, en comparación a las siento sesenta obtenidas en el robot (3,0).

El robot aquí integrado representa, gracias a los componentes utilizados, una alternativa de bajo costo, ya que puede ser fácilmente replicado, en comparación con KITs de desarrollo para sistemas robóticos móviles, los cuales representa una gran inversión para su adquisición.

La implementación de sensores de profundidad o distancia, para la solución del problema del SLAM, otorgan buenos resultados, pero, debido a la creciente presencia de diversos sensores ópticos en nuestro entorno, se busca la aplicación de algoritmos que aprovechen este tipo de datos para obtener resultado similares a las cámaras de profundidad, con la ventaja de un menor costo.

Trabajo futuro

Como trabajo futuro se pretende implementar un algoritmo de mapeo automático, de modo que al iniciar el robot no sea necesaria la intervención de una personas, ya que en los programas presentados es necesaria la intervención para evitar las colisiones con el entorno y poder crear así el mapa, posteriormente es posible utilizar puntos guiá para que el robot se desplace de forma autónoma.

Así mismo se buscará la integración de un mayor número y mejor calidad de sensores, esto debido a que es posible integrar una IMU para un mejor cálculo de odometría.

Finalmente se pretende buscar la implementación de la base aquí desarrollada para la integración de un robot de asistencia doméstica.

Apéndice A

Código de control para robot delta

```
#include <ESP8266WiFi.h>
#include <ros.h>
#include <std_msgs/String.h>
#include <std_msgs/Int16.h>
#include <std_msgs/Float64.h>
#include <ros/time.h>
#include <tf/tf.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/Range.h>
#include <Servo.h>
#include <PID_v1.h>
extern "C" {
        #include "user_interface.h"
}
#include <rosserial_arduino/Test.h>
using rosserial_arduino::Test;

// Init constants and global variables

//#define DEBUG

#define m1pos    D0
#define m1neg    D1
#define m2pos    D2
```

```
#define m2neg    D3
#define m3pos    D4
#define m3neg    D5

#define m1enc    D6
#define m2enc    D7
#define m3enc    D8

// Differential drive working variables and constants
double m1pwm = 0, m1v = 0, m1vt = 0;           // motor pwm
   output and calculated velocities (m1v as tics/sec, m1vt
   as tic/(time interval))
double m2pwm = 0, m2v = 0, m2vt = 0;           // motor pwm
   output and calculated velocities (m2v as tics/sec, m2vt
   as tic/(time interval))
double m3pwm = 0, m3v = 0, m3vt = 0;           // motor pwm
   output and calculated velocities (m3v as tics/sec, m3vt
   as tic/(time interval))

int m1mc = 0, m1mc0 = 0, m1dir = 1, m1level = 1; //
int m2mc = 0, m2mc0 = 0, m2dir = 1, m2level = 1; //
int m3mc = 0, m3mc0 = 0, m3dir = 1, m3level = 1; //


double m1In, m1Out, m1Set = 0;
double m2In, m2Out, m2Set = 0;
double m3In, m3Out, m3Set = 0;


double m1kp = 1  , m1ki = 1, m1kd = 0;
                                    // PID constants.
double m2kp = 1  , m2ki = 1, m2kd = 0;
double m3kp = 1  , m3ki = 1, m3kd = 0;


PID m1PID(&m1In, &m1Out, &m1Set, m1kp, m1ki, m1kd, DIRECT);
PID m2PID(&m2In, &m2Out, &m2Set, m2kp, m2ki, m2kd, DIRECT);
PID m3PID(&m3In, &m3Out, &m3Set, m3kp, m3ki, m3kd, DIRECT);
```

```
int period = 5;                                          // PID
   period in milliseconds
double kt = 1000 / period;                               // number
    of periods/sec
int CPR = 160;                                           //
   Encoder Count per Revolutions (can be duplicated changing
    RAISING interrupt by CHANGE)
double l = 0.12;
double rw = 0.055966 / 2;


// Timing using timer for PID and time interval between
   encoders ticks

int m1currenttime, m1lasttime;
int m2currenttime, m2lasttime;
int m3currenttime, m3lasttime;

os_timer_t myTimer;

// WiFi configuration. Replace '***' with your data
const char* ssid     = "Totalplay-1F9D";
const char* password = "1F9D9858BH6PUm4h";


IPAddress server(192, 168, 100, 10); // Set the rosserial
   socket ROSCORE SERVER IP address
const uint16_t serverPort = 11411;    // Set the rosserial
   socket server port


// Functions definitions //
void setupWiFi() {                   // connect to ROS
   server as as a client
      #ifdef DEBUG
      {
            Serial.print("Connecting to ");
```

```
                Serial.println(ssid);
                WiFi.begin(ssid, password);
        }
        #endif
        while (WiFi.status() != WL_CONNECTED) {
                delay(500);
        }
        #ifdef DEBUG
        {
                Serial.println("");
                Serial.println("WiFi connected");
                Serial.println("IP address: ");
                Serial.println(WiFi.localIP());
        }
        #endif
}

static inline int8_t sgn(int val) {   // Better sgn function
    definition
        if (val < 0) return -1;
        //  if (val == 0) return 0;
        return 1;
}

void motion(double m1pwm, double m2pwm, double m3pwm) {  //
   move motor at pwm power and change directions flags only
   when motor cross stop
        int m1f = 0, m1b = 0;
        int m2f = 0, m2b = 0;
        int m3f = 0, m3b = 0;

        if (abs(m1Set) > 0) {
                if (m1pwm >= 0) {
                        m1dir = 1;
                        m1f = 1;
                        m1b = 0;
                        m1level = HIGH;
                }
                else {
```

```
                m1dir = -1;
                m1f = 0;
                m1b = 1;
                m1level = LOW;
        }
}

if (abs(m2Set) > 0) {
        if (m2pwm >= 0) {
                m2dir = 1;
                m2f = 1;
                m2b = 0;
                m2level = HIGH;
        }
        else {
                m2dir = -1;
                m2f = 0;
                m2b = 1;
                m2level = LOW;
        }
}
if (abs(m3Set) > 0) {
        if (m3pwm >= 0) {
                m3dir = 1;
                m3f = 1;
                m3b = 0;
                m3level = HIGH;
        }
        else {
                m3dir = -1;
                m3f = 0;
                m3b = 1;
                m3level = LOW;
        }
}


analogWrite(m1pos, (400 + abs(m1pwm)) * m1f);
analogWrite(m1neg, (400 + abs(m1pwm)) * m1b);
```

```
        analogWrite(m2pos, (400 + abs(m2pwm)) * m2f);
        analogWrite(m2neg, (400 + abs(m2pwm)) * m2b);

        analogWrite(m3pos, (400 + abs(m3pwm)) * m3f);
        analogWrite(m3neg, (400 + abs(m3pwm)) * m3b);
}


void ICACHE_RAM_ATTR m1encode() {           // GPIO ISR
   Interrupt service routines for encoder changes
        m1currenttime = millis();
        if (m1currenttime != m1lasttime)
        {
                m1vt = m1dir * 1000. / (m1currenttime -
                   m1lasttime);
                m1lasttime = m1currenttime;
                m1mc = m1mc + m1dir;
        }
}

void ICACHE_RAM_ATTR m2encode() {
        m2currenttime = millis();
        if (m2currenttime != m2lasttime)
        {
                m2vt = m2dir * 1000. / (m2currenttime -
                   m2lasttime);
                m2lasttime = m2currenttime;
                m2mc = m2mc + m2dir;
        }
}

void ICACHE_RAM_ATTR m3encode() {
        m3currenttime = millis();
        if (m3currenttime != m3lasttime)
        {
                m3vt = m3dir * 1000. / (m3currenttime -
                   m3lasttime);
                m3lasttime = m3currenttime;
```

```
                m3mc = m3mc + m3dir;
        }
}


void tic(void *pArg) {     // timerCallback, repeat every "
   period"
        m1v = (m1mc - m1mc0);
        m1mc0 = m1mc;
        m2v = (m2mc - m2mc0);
        m2mc0 = m2mc;
        m3v = (m3mc - m3mc0);
        m3mc0 = m3mc;

        m1In = m1v * kt;
        m2In = m2v * kt;
        m3In = m3v * kt;

        if (abs(m1v) >= 1) m1In = abs(m1vt) * sgn(m1v); //
           use timing to calculate velocity only if ticks
           are greater than one
        if (abs(m2v) >= 1) m2In = abs(m2vt) * sgn(m2v); //
           to avoid problems with sign and inertia for a
           simple encoder (no quadrature)
        if (abs(m3v) >= 1) m3In = abs(m3vt) * sgn(m3v); //
           to avoid problems with sign and inertia for a
           simple encoder (no quadrature)

        m1PID.Compute();
        m2PID.Compute();
        m3PID.Compute();

        #ifdef DEBUG
        {
                Serial.print("l time \t");
                Serial.print(m1currenttime);
                Serial.print("\t");
                Serial.print(m1lasttime);
                Serial.print("\t");
```

```
                Serial.print(m1currenttime - m1lasttime);
                Serial.print("\t cmd_vel \t");
                Serial.print(m1In);
                Serial.print("\t");
                Serial.print(m2In);
                Serial.print("\t pwm \t");
                Serial.print(int(m1Out));
                Serial.print("\t");
                Serial.print(int(m2Out));
                Serial.print("\t pwm set \t");
                Serial.print(m1Set);
                Serial.print("\t");
                Serial.println(m2Set);
        }
        #endif

        motion(m1Out, m2Out, m3Out);
}

void cmd_velCallback( const geometry_msgs::Twist& CVel) {

        //geometry_msgs::Twist twist = twist_msg;
        double vel_x = CVel.linear.x;
        double vel_y = -CVel.linear.y;
        double vel_th = -CVel.angular.z;

        if (vel_x > 1)
        {
                vel_x = 1;
        }
        if (vel_y > 1)
        {
                vel_y = 1;
        }

        if (vel_th > 1)
        {
                vel_th = 1;
        }
```

```
        double m1_vel = 0.0;
        double m2_vel = 0.0;
        double m3_vel = 0.0;

        m1_vel = vel_y / (2 * rw) + (sqrt(3) * vel_x) / (2 *
            rw) + (l * vel_th) / rw;
        m2_vel = (l * vel_th) / rw - vel_y / rw;
        m3_vel = vel_y / (2 * rw) - (sqrt(3) * vel_x) / (2 *
            rw) + (l * vel_th) / rw;

        //write new command speeds to global vars
        // para mapear la velocidad la multiplicamos por 200
            ya que es maximo q otorga el motor
        m1Set = (m1_vel/54) * 200;
        m2Set = (m2_vel/54) * 200;
        m3Set = (m3_vel/54) * 200;
}

// ROS nodes //
ros::NodeHandle nh;
geometry_msgs::TransformStamped t;      // transformation
    frame for base
geometry_msgs::TransformStamped t2;     // transformation
    frame for camera to base
geometry_msgs::TransformStamped t3;     // transformation
    frame for lidar to base

tf::TransformBroadcaster broadcaster;
nav_msgs::Odometry odom;                // Odometry message
nav_msgs::Odometry enccountmsg;                 // Odometry
    message

geometry_msgs::Twist odom_msg;          //

//tf::TransformBroadcaster odom_broadcaster;

// ROS topics object definitions PUBLISHERS
std_msgs::String str_msg;
```

```
std_msgs::Int16 int_msg;

ros::Publisher odom_pub("/car/odom", &odom);

ros::Publisher enccount_pub("/car/enccount", &odom);

// ROS SUBSCRIBERS
ros::Subscriber<geometry_msgs::Twist> Sub("/car/cmd_vel", &
    cmd_velCallback );

// ros variables
double x = 0.0;
double y = 0.0;
double th = 0;

char base_link[] = "/base_footprint";
char odomid[] = "/odom";
char enc_countid[] = "/enccount";
char cameraid[] = "/camera_link";
char lidarid[] = "/laser";

void reset_odom_callback(const Test::Request & req, Test::
    Response & res) {
        x = 0.0;
        y = 0.0;
        th = 0;
        m1mc = 0;
        m2mc = 0;
        m3mc = 0;
        res.output = "reseted";
}

ros::ServiceServer<Test::Request, Test::Response>
    reset_server("reset_srv", &reset_odom_callback);


void setup() {
        // configure GPIO's and Servo
        pinMode(m1pos, OUTPUT); //motor 1
```

```
pinMode(m1neg, OUTPUT); //motor 1
pinMode(m2pos, OUTPUT); //motor 2
pinMode(m2neg, OUTPUT); //motor 2
pinMode(m3pos, OUTPUT); //motor 3
pinMode(m3neg, OUTPUT); //motor 3
pinMode(m1enc, INPUT); //encoder
pinMode(m2enc, INPUT); //encoder
pinMode(m3enc, INPUT); //encoder

#ifdef DEBUG
Serial.begin(115200);
#endif
setupWiFi();
delay(2000);
// Ros objects constructors
nh.getHardware()->setConnection(server, serverPort);
nh.initNode();
broadcaster.init(nh);
nh.advertise(odom_pub);
nh.advertise(enccount_pub);
nh.advertiseService(reset_server);
nh.subscribe(Sub);

// configure interrupts to their ISR's
attachInterrupt(m1enc, m1encode, RISING); // Setup
   Interrupt
attachInterrupt(m2enc, m2encode, RISING); // Setup
   Interrupt
attachInterrupt(m3enc, m3encode, RISING); // Setup
   Interrupt

sei();                                    // Enable
   interrupts

// configure timer
os_timer_setfn(&myTimer, tic, NULL);
os_timer_arm(&myTimer, period, true);    // timer in
   ms
```

```
        // Configure and start PID's
        m1PID.SetSampleTime(period);
        m2PID.SetSampleTime(period);
        m3PID.SetSampleTime(period);
        m1PID.SetOutputLimits(-623, 623);
        m2PID.SetOutputLimits(-623, 623);
        m3PID.SetOutputLimits(-623, 623);
        m1PID.SetMode(AUTOMATIC);
        m2PID.SetMode(AUTOMATIC);
        m3PID.SetMode(AUTOMATIC);
}


// odometry configuration
ros::Time current_time = nh.now();
ros::Time last_time = current_time;


double Radpercount = (360 / CPR) * DEG_TO_RAD ;
int last_m1mc = m1mc;
int last_m2mc = m2mc;
int last_m3mc = m3mc;
int current_m1mc = m1mc;
int current_m2mc = m2mc;
int current_m3mc = m3mc;


void loop() {

        ///-------
        current_time = nh.now();
        current_m1mc = m1mc;
        current_m2mc = m2mc;
        current_m3mc = m3mc;

        double dt = current_time.toSec() - last_time.toSec()
           ;
```

```
double m1d = (current_m1mc - last_m1mc) *
   Radpercount; // m1 wheel angular distance
double m2d = (current_m2mc - last_m2mc) *
   Radpercount; // m2 wheel angular distance
double m3d = (current_m3mc - last_m3mc) *
   Radpercount; // m3 wheel angular distance

double vm1 = m1d / dt;
                                    // m1 wheel angular
    velocity
double vm2 = m2d / dt;
                                    // m2 wheel angular
    velocity
double vm3 = m3d / dt;
                                    // m3 wheel angular
    velocity

double vx =  (sqrt(3) * rw * vm1) / 3 - (sqrt(3) *
   rw * vm3) / 3;                    // base center x
   velocity
double vy =  (rw * vm1) / 3 - (2 * rw * vm2) / 3 + (
   rw * vm3) / 3;                   // base center y
   velocity
double vth = -((rw * vm1) / (3 * l) + (rw * vm2) /
   (3 * l) + (rw * vm3) / (3 * l));  // base center
   angular velocity

double delta_th = (vth) * dt;
th += delta_th;

last_m1mc = current_m1mc;
last_m2mc = current_m2mc;
last_m3mc = current_m3mc;

last_time = current_time;

double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
```

```
x += delta_x;
y += delta_y;


//----------

if (nh.connected()) {


        t.header.frame_id = odomid;
        t.child_frame_id = base_link;
        t.transform.translation.x = x;
        t.transform.translation.y = y;
        t.transform.rotation = tf::
           createQuaternionFromYaw(th);
        t.header.stamp = current_time;
        broadcaster.sendTransform(t);

        t2.header.frame_id = base_link;
        t2.child_frame_id = cameraid;
        t2.transform.translation.x = 0.1;
        t2.transform.translation.y = 0.0;
        t2.transform.translation.z = 0.1;
        t2.transform.rotation = tf::
           createQuaternionFromYaw(0);
        t2.header.stamp = current_time;
        broadcaster.sendTransform(t2);

        t3.header.frame_id = base_link;
        t3.child_frame_id = lidarid;
        t3.transform.translation.x = 0.0;
        t3.transform.translation.y = 0.0;
        t3.transform.translation.z = 0.12;
        t3.transform.rotation = tf::
           createQuaternionFromYaw(120*DEG_TO_RAD);
        t3.header.stamp = current_time;
        broadcaster.sendTransform(t3);

        // odometry
```

```
            geometry_msgs::Quaternion odom_quat = tf::
                createQuaternionFromYaw(th);
            odom.header.stamp = current_time;
            odom.header.frame_id = odomid;

            //set the position
            odom.pose.pose.position.x = x;
            odom.pose.pose.position.y = y;
            odom.pose.pose.position.z = 0.0;
            odom.pose.pose.orientation = odom_quat;

            //set the velocity
            odom.child_frame_id = "base_footprint";
            odom.twist.twist.linear.x = vx;
            odom.twist.twist.linear.y = vy;
            odom.twist.twist.angular.z = vth;

            //publish the message
            odom_pub.publish(&odom);


            enccountmsg.header.stamp = current_time;
            enccountmsg.header.frame_id = enc_countid;

            //set the position
            enccountmsg.pose.pose.position.x = m1mc;
            enccountmsg.pose.pose.position.y = m2mc;
            enccountmsg.pose.pose.position.z = m3mc;

            enccount_pub.publish(&enccountmsg);

    }
    #ifdef DEBUG
    else {
            Serial.println("Not Connected");
    }
    #endif

    nh.spinOnce();
```

```
        // Loop aprox. every
        delay(100);   // milliseconds
}
```

Apéndice B

Código de control para robot uniciclo

```
#include <ESP8266WiFi.h>
#include <ros.h>
#include <std_msgs/String.h>
#include <std_msgs/Int16.h>
#include <std_msgs/Float64.h>
#include <ros/time.h>
#include <tf/tf.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/Range.h>
#include <Servo.h>
#include <PID_v1.h>
extern "C" {
        #include "user_interface.h"
}

// Init constants and global variables

#define DEBUG 0
#define TRIGGER D8  // ultrasonic trigger pin
#define ECHO    D0  // ultrasonic echo pin  (!!Note: use 5v
   Vcc on ultrasonic board & a 2k,1k divider for ECHO GPIO
   protection)
```

```
// Differential drive working variables and constants

double lpwm = 0, rpwm = 0, lv = 0, rv = 0, lvt = 0, rvt = 0;
              // motor pwm output and calculated velocities
   (lv as tics/sec, lvt as tic/(time interval))
int lmc = 0, rmc = 0, lmc0 = 0, rmc0 = 0, ldir = 1, rdir =
   1, llevel = 1, rlevel = 1; // l=left, r=right motors
   encoder params

double lIn, rIn, lOut, rOut, lSet = 0, rSet = 0;
                          // PID Input Signal, Output command
    and Setting speed for each wheel
//double lkp = 50, lki = 30, lkd = 10;
                                    // Left/right wheel PID
   constants.
//double rkp = 50, rki = 30, rkd = 10;

double lkp = 4.6934 , lki = 51.4037, lkd = 0.0027;
                                    // Left/right wheel PID
   constants.
double rkp = 4.6934 , rki = 51.4037, rkd = 0.0027;
                                    // Left/right wheel PID
   constants.

PID lPID(&lIn, &lOut, &lSet, lkp, lki, lkd, DIRECT);
PID rPID(&rIn, &rOut, &rSet, rkp, rki, rkd, DIRECT);

int period = 50;                                        // PID
   period in milliseconds
double kt = 1000 / period;                              // number
    of periods/sec
double WheelSeparation = 0.125;
   // wheel separation in meters
double whedia = 0.065;                                   // wheel
    diameter in meters
int CPR = 16;                                           //
   Encoder Count per Revolutions (can be duplicated changing
    RAISING interrupt by CHANGE)
```

```
// Timing using timer for PID and time interval between
   encoders ticks

int rcurrenttime, rlasttime, lcurrenttime, llasttime;
os_timer_t myTimer;

// WiFi configuration. Replace '***' with your data

const char* ssid     = "Totalplay-1F9D";
const char* password = "1F9D9858BH6PUm4h";
IPAddress server(192, 168, 100, 10); // Set the rosserial
   socket ROSCORE SERVER IP address
const uint16_t serverPort = 11411;    // Set the rosserial
   socket server port


// Functions definitions //

void setupWiFi() {                    // connect to ROS
   server as as a client
        if (DEBUG) {
                Serial.print("Connecting to ");
                Serial.println(ssid);
                WiFi.begin(ssid, password);
        }
        while (WiFi.status() != WL_CONNECTED) {
                delay(500);
        }
        if (DEBUG) {
                Serial.println("");
                Serial.println("WiFi connected");
                Serial.println("IP address: ");
                Serial.println(WiFi.localIP());
        }
}

static inline int8_t sgn(int val) {   // Better sgn function
    definition
```

```
        if (val < 0) return -1;
        //  if (val == 0) return 0;
        return 1;

}

void motion(double lpwm, double rpwm) {  // move motor at
   pwm power and change directions flags only when motor
   cross stop
        int rf = 0, rb = 0, lf = 0, lb = 0; //rigth front,
           rigth back, left front, left back

        if (abs(lSet) > 0) {
                if (lpwm >= 0) {
                        ldir = 1;
                        lf = 1;
                        lb = 0;
                        llevel = HIGH;
                }
                else {
                        ldir = -1;
                        lf = 0;
                        lb = 1;
                        llevel = LOW;
                }
        }

        if (abs(rSet) > 0) {
                if (rpwm >= 0) {
                        rdir = 1;
                        rf = 1;
                        rb = 0;
                        rlevel = HIGH;
                }
                else {
                        rdir = -1;
                        rf = 0;
                        rb = 1;
                        rlevel = LOW;
```

```
                }
        }


        analogWrite(D1, (400 + abs(lpwm)) * lf);
        analogWrite(D3, (400 + abs(lpwm)) * lb);

        analogWrite(D2, (400 + abs(rpwm)) * rf);
        analogWrite(D4, (400 + abs(rpwm)) * rb);
}


void ICACHE_RAM_ATTR lencode() {          // GPIO ISR
   Interrupt service routines for encoder changes
        lcurrenttime = millis();
        if (lcurrenttime != llasttime)
        {
                lvt = ldir * 1000. / (lcurrenttime -
                   llasttime);
                llasttime = lcurrenttime;
                lmc = lmc + ldir;
        }
}

void ICACHE_RAM_ATTR rencode() {
        rcurrenttime = millis();
        if (rcurrenttime != rlasttime)
        {
                rvt = rdir * 1000. / (rcurrenttime -
                   rlasttime);
                rlasttime = rcurrenttime;
                rmc = rmc + rdir;
        }
}

void tic(void *pArg) {    // timerCallback, repeat every "
   period"
        lv = (lmc - lmc0);
        lmc0 = lmc;
```

```
        rv = (rmc - rmc0);
        rmc0 = rmc;

        lIn = abs(lvt)* ldir;//sgn(lv); // use timing to
           calculate velocity only if ticks are greater than
            one
        rIn = abs(rvt)* rdir;//sgn(rv); // to avoid problems
            with sign and inertia for a simple encoder (no
           quadrature)

        lPID.Compute();
        rPID.Compute();

        if (DEBUG) {
                Serial.print("l time \t");
                Serial.print(lcurrenttime);
                Serial.print("\t");
                Serial.print(llasttime);
                Serial.print("\t");
                Serial.print(lcurrenttime -llasttime);
                Serial.print("\t cmd_vel \t");
                Serial.print(lIn);
                Serial.print("\t");
                Serial.print(rIn);
                Serial.print("\t pwm \t");
                Serial.print(int(lOut));
                Serial.print("\t");
                Serial.print(int(rOut));
                Serial.print("\t pwm set \t");
                Serial.print(lSet);
                Serial.print("\t");
                Serial.println(rSet);
        }
        motion(lOut, rOut);
}

void cmd_velCallback( const geometry_msgs::Twist& CVel) {

        //geometry_msgs::Twist twist = twist_msg;
```

```
        double vel_x = CVel.linear.x;
        double vel_th = CVel.angular.z;
        double right_vel = 0.0;
        double left_vel = 0.0;

        left_vel =  (vel_x - vel_th ) / 2.0;
        right_vel = (vel_x + vel_th)  / 2.0;

        //write new command speeds to global vars

        // para mapear la velocidad la multiplicamos por 16
           ya que es maximo q otorga el motor
        lSet = left_vel * 25;
        rSet = right_vel * 25;
}


// ROS nodes //
ros::NodeHandle nh;
geometry_msgs::TransformStamped t;     // transformation
   frame for base
geometry_msgs::TransformStamped t2;    // transformation
   frame for camera to base

tf::TransformBroadcaster broadcaster;
nav_msgs::Odometry odom;               // Odometry message
geometry_msgs::Twist odom_msg;         //

//tf::TransformBroadcaster odom_broadcaster;

// ROS topics object definitions PUBLISHERS
std_msgs::String str_msg;
std_msgs::Int16 int_msg;

ros::Publisher odom_pub("/car/odom", &odom);

// ROS SUBSCRIBERS
ros::Subscriber<geometry_msgs::Twist> Sub("/car/cmd_vel", &
   cmd_velCallback );
```

```
// ros variables
double x = 0.0;
double y = 0.0;
double th = 0;

char base_link[] = "/base_footprint";
char odomid[] = "/odom";
char cameraid[] = "/camera_link";

void setup() {
        if (DEBUG) Serial.begin(115200);
        setupWiFi();
        delay(2000);
        // Ros objects constructors
        nh.getHardware()->setConnection(server, serverPort);
        nh.initNode();
        broadcaster.init(nh);
        nh.advertise(odom_pub);
        nh.subscribe(Sub);

        // configure GPIO's and Servo
        pinMode(D0, OUTPUT); // Ultrasonic Trigger
        pinMode(D1, OUTPUT); // 1,2EN aka D1 pwm left
        pinMode(D2, OUTPUT); // 3,4EN aka D2 pwm right
        pinMode(D3, OUTPUT); // 1A,2A aka D3
        pinMode(D4, OUTPUT); // 3A,4A aka D4
        pinMode(D5, INPUT); //  Left encoder
        pinMode(D6, INPUT); //  Right encoder

        // configure interrupts to their ISR's
        attachInterrupt(D5, lencode, RISING); // Setup
           Interrupt
        attachInterrupt(D6, rencode, RISING); // Setup
           Interrupt
        sei();                                // Enable
           interrupts

        // configure timer
```

```
        os_timer_setfn(&myTimer, tic, NULL);
        os_timer_arm(&myTimer, period, true);    // timer in
           ms

        // Configure and start PID's
        lPID.SetSampleTime(period);
        rPID.SetSampleTime(period);
        lPID.SetOutputLimits(-623, 623);
        rPID.SetOutputLimits(-623, 623);
        lPID.SetMode(AUTOMATIC);
        rPID.SetMode(AUTOMATIC);
}


// odometry configuration
ros::Time current_time = nh.now();
ros::Time last_time = current_time;

double DistancePerCount = (TWO_PI * 0.0325) / 16;    // 2*PI*
   R/CPR  WHEEL ENCODER 8 CPR
double lengthBetweenTwoWheels = 0.13;

int last_lmc = lmc;
int last_rmc = rmc;
int current_lmc = lmc;
int current_rmc = rmc;


void loop() {

        ///-------
        current_time = nh.now();
        current_lmc = lmc;
        current_rmc = rmc;

        double dt = current_time.toSec() - last_time.toSec()
           ;
        double ld = (current_lmc - last_lmc) *
           DistancePerCount; // left wheel linear distance
```

```
double rd = (current_rmc - last_rmc) *
   DistancePerCount; // right wheel linear distance
double vlm = ld / dt;
                                    // left wheel
   linear velocity
double vrm = rd / dt;
   // right wheel linear velocity
double vx = (ld + rd) / 2.;
   // base center forward velocity
double vy = 0;
double th = (current_rmc - current_lmc) / 45. * PI;
double vth = th / dt;

last_lmc = current_lmc;
last_rmc = current_rmc;
last_time = current_time;

double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;

x += delta_x;
y += delta_y;

//----------

if (nh.connected()) {


        t.header.frame_id = odomid;
        t.child_frame_id = base_link;
        t.transform.translation.x = x;
        t.transform.translation.y = y;
        t.transform.rotation = tf::
           createQuaternionFromYaw(th);
        t.header.stamp = current_time;
        broadcaster.sendTransform(t);

        t2.header.frame_id = base_link;
        t2.child_frame_id = cameraid;
```

```
                t2.transform.translation.x = 0.05;
                t2.transform.translation.y = 0.0;
                t2.transform.translation.z = 0.1;
                t2.transform.rotation = tf::
                    createQuaternionFromYaw(0);
                t2.header.stamp = current_time;
                broadcaster.sendTransform(t2);


                // odometry
                geometry_msgs::Quaternion odom_quat = tf::
                    createQuaternionFromYaw(th);
                odom.header.stamp = current_time;
                odom.header.frame_id = odomid;

                //set the position
                odom.pose.pose.position.x = x;
                odom.pose.pose.position.y = y;
                odom.pose.pose.position.z = 0.0;
                odom.pose.pose.orientation = odom_quat;

                //set the velocity
                odom.child_frame_id = "base_footprint";
                odom.twist.twist.linear.x = vx;
                odom.twist.twist.linear.y = vy;
                odom.twist.twist.angular.z = vth;

                //publish the message
                odom_pub.publish(&odom);

        } else {
                if (DEBUG) Serial.println("Not Connected");
        }
        nh.spinOnce();
        // Loop aprox. every
        delay(100);   // milliseconds
}
```

Apéndice C

Repositorios de simulaciones

Figura C.1: Repositorios de paquetes de simulación.

En la Figura C.1, se presenta una imagen de los repositorios en donde han sido almacenados los paquetes de simulación desarrollados en esta tesis, y se encuentran disponibles en los siguientes enlaces:

https://github.com/longinos/ros-gazebo_uniciclo.git

https://github.com/longinos/ros-gazebo_delta.git

Bibliografía

[1] Á. Araya del Río. Una introducción a los robots móviles. Accedido el 28 de mayo del 2019 de http://docplayer.es/7826797-Una-introduccion-a-los-robotsmoviles-il-bambino-piu-avanti-hs-gmail-com.html.

[2] Ioan Doroftei, Victor Grosu, and V. Spinu. *Omnidirectional Mobile Robot - Design and Implementation*. 09 2007.

[3] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. Intelligent Robotics and Autonomous Agents series. MIT Press, 2005.

[4] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. springer, 2016.

[5] Bruno Duarte Gouveia, David Portugal, Daniel C Silva, and Lino Marques. Computation sharing in distributed robotic systems: A case study on slam. *IEEE Transactions on Automation Science and Engineering*, 12(2):410–422, 2014.

[6] Mathieu Labbe and Francois Michaud. Appearance-based loop closure detection for online large-scale and long-term operation. *IEEE Transactions on Robotics*, 29(3):734–745, 2013.

[7] Tawsif Gokhool, Renato Martins, Patrick Rives, and Noela Despre. A compact spherical rgbd keyframe-based representation, 2015.

[8] Jorge Angeles and J Angeles. Fundamentals of robotic mechanical systems: theory. *Methods, and Algorithms (Mechanical Engineering Series, Vol. 2)*, page 98, 2007.

[9] K. Yousif, Y. Taguchi, and S. Ramalingam. Monorgbd-SLAM: Simultaneous localization and mapping using both monocular and rgbd cameras. In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, pages 4495–4502, May 2017.

[10] F. Endres, J. Hess, J. Sturm, D. Cremers, and W. Burgard. 3-D mapping with an rgb-d camera. *IEEE Transactions on Robotics*, 30(1):177–187, February 2014.

[11] Organización Mundial de la Salud. Informe mundial sobre la discapacidad,2011. Accedido el 28 de mayo del 2019 de https://www.who.int/disabilities/world_report/2011/summary_es.pdf.

[12] Organización Mundial de la Salud. Informe mundial sobre el envejecimiento y la salud,2015. Accedido el 28 de mayo del 2019 de https://www.who.int/ageing/publications/world-report-2015/es/.

[13] C. Park, S. Kang, J. Kim, and J. Oh. A study on service robot system for elder care. In *2012 9th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 546–547, 2012.

[14] P. E. Ross. Robot, you can drive my car. *IEEE Spectrum*, 51(6):60–90, June 2014.

[15] K. Bimbraw. Autonomous cars: Past, present and future a review of the developments in the last century, the present scenario and the expected future of autonomous vehicle technology. In *Proc. Automation and Robotics (ICINCO) 2015 12th Int. Conf. Informatics in Control*, volume 01, pages 191–198, July 2015.

[16] Eniko T Enikov and Juan-Antonio Escareno. Application of sensory body schemas to path planning for micro air vehicles (mavs). In *2015 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, volume 1, pages 25–31. IEEE, 2015.

[17] Hiep Do Quang, Tien Ngo Manh, Cuong Nguyen Manh, Dung Pham Tien, Manh Tran Van, Duyen Ha Thi Kim, Dam Hong Duan, et al. Mapping and navigation with four-wheeled omnidirectional mobile robot based on robot operating system. In *2019 International Conference on Mechatronics, Robotics and Systems Engineering (MoRSE)*, pages 54–59. IEEE, 2019.

[18] Aníbal Ollero Baturone. *Robótica: manipuladores y robots móviles*. Marcombo, 2005.

[19] G. Campion, G. Bastin, and B. Dandrea-Novel. Structural properties and classification of kinematic and dynamic models of wheeled mobile robots. *IEEE Transactions on Robotics and Automation*, 12(1):47–62, February 1996.

[20] Anita M Flynn, Joseph L Jones, and Bruce A Seiger. Mobile robots: Inspiration to implementation. *AK Peters, Wellesley, Mass*, 1993.

[21] Patricia López Torres. Análisis de algoritmos para localización y mapeado simultáneo de objetos. 2016.

[22] Lentin Joseph. *Robot Operating System (ROS) for Absolute Beginners*. Springer, 2018.

[23] Gazebosim. http://gazebosim.org/. Accessed: 2020-04-07.

[24] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. 13:99–110, 2006.

[25] Federico Andrade, Martin Llofriu, Gonzalo Tejera, and Facundo Benavides. Slam estado del arte, 2007.

[26] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. 1960.

[27] Greg Welch, Gary Bishop, et al. An introduction to the kalman filter. 1995.

[28] Reza Olfati-Saber. Flocking for multi-agent dynamic systems: Algorithms and theory. *IEEE Transactions on automatic control*, 51(3):401–420, 2006.

[29] Stefan Kohlbrecher, Oskar Von Stryk, Johannes Meyer, and Uwe Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *2011 IEEE international symposium on safety, security, and rescue robotics*, pages 155–160. IEEE, 2011.

[30] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE transactions on Robotics*, 23(1):34–46, 2007.

[31] Joao Machado Santos, David Portugal, and Rui P Rocha. An evaluation of 2d slam techniques available in robot operating system. In *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 1–6. IEEE, 2013.