FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

CARRERA DE ESPECIALIZACIÓN EN
SISTEMAS EMBEBIDOS

MEMORIA DEL TRABAJO FINAL

Firmware para robot de navegación autónoma

Autor:
Ing. Alexis Martin Pojomovsky

Director:
Dr. Pablo De Cristóforis (FCEyN-UBA)

Jurados:
Esp. Ing. Diego Fernández (FI-UBA)
Esp. Ing. Edgardo Comas (CITEDEF/UTN-FRBA)
Esp. Ing. Gerardo Puga (UNLP)

Este trabajo fue realizado en la Ciudad Autónoma de Buenos Aires, entre enero de 2018 y agosto de 2020.

Resumen

La presente memoria describe el desarrollo del robot móvil “Lubobot“, destinado a entornos educativos de grado universitario a través de las distintas etapas de estudio de la materia “Robótica Móvil“, dictada como asignatura de la carrera de Ingeniería Mecatrónica en la Universidad Nacional de Asunción, Paraguay. Como parte del trabajo se realizó la implementación básica requerida para utilizar este robot con el Sistema Operativo Robótico (ROS).

Se aplicaron diversos conocimientos adquiridos durante la carrera, entre los que se destacan los referidos a sistemas operativos, control de versiones y protocolos de comunicación. Así también, se hizo uso de patrones de diseño y técnicas de modularización de software.

Agradecimientos

Agradezco el apoyo provisto por mi esposa Laura así como el de mis padres Evelyn y Ernesto, presentes durante todo el proceso de mi formación, siempre alentándome a ir un paso mas allá.
Al equipo docente y colaboradores de la Carrera de Especialización de Sistemas Embebidos, por la dedicación y entrega manifestados durante cada etapa de la cursada.

Índice general

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción general

En este capítulo se introduce al campo de estudio de la robótica móvil. Se aborda una comparación entre diferentes plataformas didácticas comerciales y se exponen el alcance y las motivaciones que llevaron al desarrollo del presente proyecto.

1.1. Robótica móvil

La robótica móvil se encarga del estudio de los robots móviles y hace especial hincapié en el desarrollo de capacidades les permitan decidir de manera autónoma cómo, cuándo y a dónde moverse.

En contraste con los robots manipuladores cuya base se encuentra fija con respecto a un sistema de referencia, los robots móviles son aquellos capaces de moverse de un lugar a otro. Esta particularidad los obliga a ser capaces de interactuar con entornos no determinísticos, es decir, propensos a situaciones impredecibles como por ejemplo una puerta entreabierta, un objeto o persona que obstaculiza el camino, etc.

1.1.1. Tipos de robots móviles

Dependiendo de cómo realizan su locomoción, es posible caracterizar a los robots móviles en los siguientes tipos:

- Robots con patas
- Robots aéreos
- Robots con ruedas

Cada uno de estos tipos plantea su propio conjunto de ventajas y desventajas, así como dificultades para su implementación. En el presente trabajo se hizo énfasis solo en el último tipo de la lista, es decir en los robots con ruedas.

1.2. Estado del arte

Existe una amplia gama de robots móviles con ruedas ofrecidos específicamente para el sector académico. A continuación se presenta un breve sumario de opciones que se encuentran actualmente en el mercado.

1.2.1. TurtleBot

TurtleBot constituye una familia de robots móviles para uso personal de bajo costo. Su uso se extiende tanto a la academia como a roboticistas aficionados en todo el mundo.

Aunque su primera iteración vio la luz en 2010 con un conjunto de características bastante modestas, el lanzamiento de nuevas versiones le aseguró su lugar como dispositivo de referencia dentro de la plataforma ROS.

Muchas de las consideraciones de diseño del robot propuesto en este trabajo fueron tomados de este modelo por lo que se espera que se encuentren similitudes entre ambos al observar la figura 1.1.

FIGURA 1.1. Vista frontal del Turtlebot 2 con un sensor Kinect acoplado.[1]

1.2.2. Clearpath Jackal

El Jackal es un robot móvil 4x4 apto para uso en exteriores. Su robustez lo hace la elección preferida de muchas universidades a la hora de implementar soluciones de campo, principalmente debido a su resistencia total al polvo y al agua de lluvia.

[1]Imagen tomada de https://static.generation-robots.com/6739-large_default/turtlebot-2-assembled-clearpathrobotics.jpg

Posee una capacidad de carga de hasta 20 kg, lo que lo hace apto para cargar una importante cantidad de sensores, actuadores y manipuladores, tal como el ejemplo mostrado en la figura 1.2.

FIGURA 1.2. Vista frontal de la plataforma robótica Jackal con una batería de sensores instalados por el usuario.[2]

1.2.3. Fetch Freight 100 Base

Fetch Robotics ofrece con el Freight 100 Base una plataforma robótica para uso en interiores [1]. Esta fue diseñada específicamente para moverse en edificios adaptados a personas en sillas de ruedas que cumplen con la normativa ADA o *Americans with Disabilities Act* [2].

El Freight 100 incluye un sensor del tipo LIDaR 2D, que se puede apreciar en la figura 1.3, lo que lo hace adecuado para tareas de navegación. Además, al estar basado en un robot industrial de carga, la plataforma Freight 100 está diseñada para soportar hasta 100 kg de peso, característica que no solo se impone por un amplio margen respecto a sus competidores del segmento, sino que le posibilita cargar con un manipulador industrial en su parte superior.

FIGURA 1.3. Vista frontal de la plataforma robótica Fetch Freight donde se puede apreciar su sensor integrado del tipo LIDaR.[3]

[2]Imagen tomada de https://img.directindustry.es/images_di/photo-g/177123-10073681.jpg

1.2.4. FESTO Robotino

Robotino es un robot móvil comercializado por la compañía alemana FESTO Didactic. Está diseñado para entornos educativos, de entrenamiento y de investigación. Esta plataforma dispone de un sistema de tracción omni que como su nombre sugiere, le otorga libertad de movimiento omnidireccional en dos dimensiones.

Asímismo, Robotino viene equipado de fábrica con una computadora tipo PC de grado industrial, lo que facilita su integración con elementos de hardware y software externos tales como cámaras, sensores y actuadores que utilicen el protocolo RS-232, RS-485 o USB. En la figura 1.4 se puede apreciar al robot con una *webcam* acoplada.

FIGURA 1.4. Vista frontal del Robot FESTO Robotino con un sensor externo acoplado.[4]

1.2.5. Pioneer 3-DX

El Pioneer 3-DX es un pequeño robot de tracción diferencial ideal para ser utilizado en entornos académicos ya sea en un laboratorio o en un salón de clases. Incorpora de fábrica cinco sensores del tipo SONAR al frente del robot como los que se pueden apreciar en la figura 1.5, encoders para las ruedas y un microcontrolador con firmware específico.

Su amplia superficie de carga permite transportar hasta 8 Kg, por lo que es un buen candidato para añadir sensores y actuadores extra, tales como cámaras o manipuladores de tamaño adecuado.

1.3. Motivación

Los robots móviles expuestos en la sección anterior representan propuestas comerciales listas para usar que permiten a profesores, investigadores y alumnos concentrarse en el estudio o desarrollo de aplicaciones de la robótica móvil sin la necesidad de diseñar un robot desde cero para cada caso de uso específico.

[3]Imagen tomada de https://www.dymesich.com/wp-content/uploads/2019/06/freight-100-low.jpg

[4]Imagen tomada de https://www.festo-didactic.com/ov3/media/customers/1100/robotinohome.png

[5]Imagen tomada de https://static.generation-robots.com/6645-large_default/robot-mobile-pioneer-3-at.jpg

FIGURA 1.5. Vista frontal del Robot Pioneer donde se aprecian sus cinco sensores SONAR.[5]

Gracias a esto, las instituciones o individuos que adquieren dichos equipos pueden ahorrar tiempo al concentrarse de inmediato en las tareas de interés para el estudio de la materia.

Es pertinente preguntar ¿por qué no todas las instituciones adquieren un robot comercial para el laboratorio de robótica?, la respuesta está en los costos asociados a la adquisición de estos equipos. Los robots destinados a investigación poseen precios prohibitivos para la mayoría de las instituciones educativas y por ende, son muy pocas las que se encuentran en condiciones de invertir en ellos.

1.4. Objetivos

En este trabajo se propone una plataforma de código abierto y con componentes disponibles en el mercado local paraguayo, destinada a la enseñanza de robótica móvil en instituciones educativas de nivel universitario.

1.5. Alcance

El alcance del presente trabajo involucra el desarrollo del firmware para el microcontrolador encargado de:

- intermediar la comunicación entre la plataforma robótica “Roomba 500” y el framework de robótica ROS (*Robot Operating System*).
- realizar la interfaz entre la unidad de medición inercial MPU6050 con ROS.
- implementar el *port* de la biblioteca rosserial para ser utilizada con FreeRTOS y STM32Cube HAL en el lenguaje de programación C++.

Se incluye el desarrollo de software requerido para:

- el paquete para ROS que brinda soporte básico para el robot propuesto "Lubobot", con sus sensores y actuadores.
- calcular la odometría del robot utilizando las lecturas de los encoders.
- la descripción del robot en formato URDF (*Unified Robot Description Format*), requerido para representar correctamente el robot en la herramienta RViz.

- una imagen de Docker con todas las dependencias necesarias para utilizar el robot de manera encapsulada y sin la necesidad de modificar la configuración del sistema operativo del usuario.
- una seccion extra en el repositorio oficial en GitHub con una Wiki con la documentación básica necesaria para iniciarse en el uso de la plataforma.

No se incluye con el presente trabajo:

- ningún algoritmo de navegación local ni global
- el código requerido para la conexión al microcontrolador mediante Ethernet
- el diseño de una placa electrónica dedicada

Capítulo 2

Introducción específica

En este capítulo se desglosan las diferentes herramientas tanto de hardware como de software elegidas para el desarrollo del robot propuesto.

2.1. Robot Operating System

Comúnmente denominado ROS, es un framework de robótica de código abierto que fue diseñado originalmente para robots de uso académico. Sin embargo, al día de hoy su uso se ha extendido tanto a la industria como al público aficionado.

ROS ofrece un variado *set* de herramientas que facilitan las tareas del roboticista en funciones como envío y recepción de mensajes, computación distribuida e implementación de algoritmos para aplicaciones robóticas.

2.1.1. Organización de archivos

Es adecuado considerar a ROS como algo más que un framework de desarrollo y referirnos a el como un "meta sistema operativo", ya que ofrece no solo herramientas y bibliotecas sino también funciones similares a las de un sistema operativo. Entre ellas, podemos citar su abstracción del hardware, el manejo de paquetes, además de un completo *toolchain* de compilación. Así también, tal como en un sistema operativo "real", los archivos que componen ROS se encuentran organizados en el disco duro de una manera particular, como se indica en la figura 2.1.

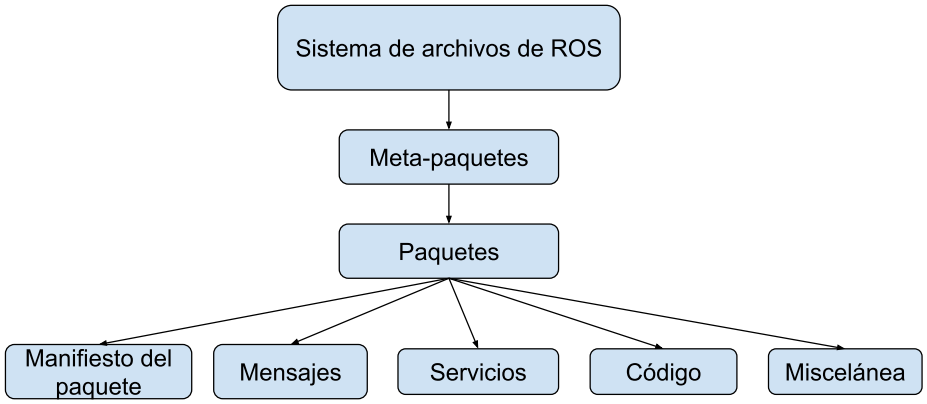

FIGURA 2.1. Organización de archivos en ROS

A continuación se detalla en qué consiste cada uno de los bloques que componen la organización de archivos:

- **Paquetes**: los paquetes de ROS representan la unidad básica de software en la plataforma. Contienen uno o más programas de ROS (nodos), bibliotecas, archivos de configuración, etc, que son organizados como una unidad coherente.
- **Manifiesto del paquete**: está representado por un archivo único dentro del paquete que contiene información sobre este tales como el nombre, autor, tipo de licencia, dependencias, banderas de compilación, etc. El archivo `package.xml` encontrado en la raíz del paquete es su propio manifiesto.
- **Metapaquete**: hace referencia a uno o más paquetes usualmente relacionados entre sí, pero sin estar necesariamente acoplados fuertemente unos con otros. El software provisto con este proyecto es un metapaquete.
- **Manifiesto del metapaquete**: similar al manifiesto del paquete, con la diferencia que este puede incluir dependencias a de tiempo de ejecución hacia otros paquetes.
- **Mensajes**: representados con la extensión `.msg`, son los tipos de datos utilizados por ROS internamente para comunicar los distintos procesos entre sí. El usuario puede definir tipos de mensajes personalizados con campos adaptados a sus necesidades en del directorio `msg` dentro del paquete.
- **Servicios**: representados con la extensión `.srv`, representan interacciones del tipo solicitud/respuesta entre distintos procesos. Los formatos tanto para solicitud como para respuesta pueden definirse en el directorio `srv` dentro del paquete.
- **Repositorios**: la gran mayoría de los paquetes de ROS son mantenidos utilizando un sistema de control de versiones (SCV) tales como Git, Mercurial o SVN. Es común encontrar metapaquetes definidos dentro de un mismo único repositorio. Este es el caso para el repositorio provisto en este trabajo.

2.1.2. Arquitectura interna

ROS está construido sobre una arquitectura basada en grafos, esto significa que las tareas de cómputo son realizadas a través de una red de procesos llamados nodos. Esta red es denominada *Computation Graph* o grafo de cómputo.

Los principales componentes de este grafo son los nodos, el *master* o maestro, el *parameter server* o servidor de parámetros, además de los mensajes, tópicos, servicios y *bags* o bolsas. Cada uno de estos elementos contribuye al funcionamiento del *Computation Graph* con una funcionalidad específica. Los elementos que lo componen, mostrados en la figura 2.2, se describen a continuación:

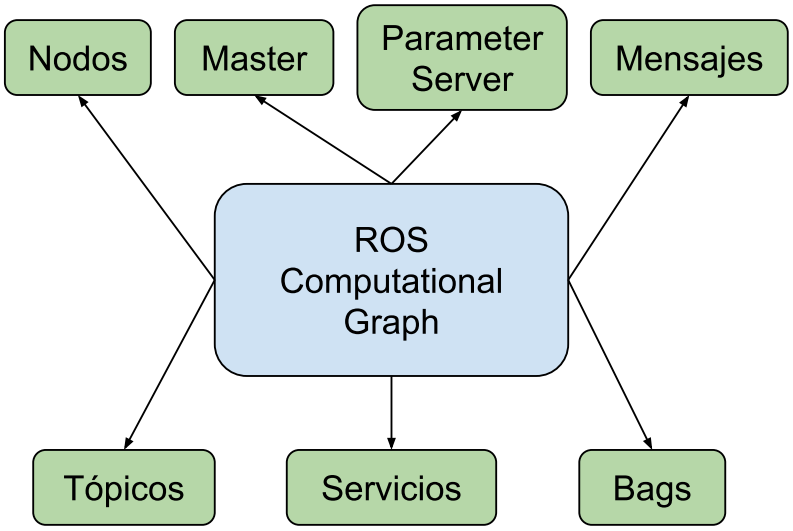

FIGURA 2.2. Estructura del *Computation Graph* de ROS.

- **Nodos**: son los procesos que realizan las tareas de cómputo dentro del robot, que pueden comunicarse unos con otros a través de la API de ROS. Esto resulta particularmente útil cuando distintos nodos necesitan compartir información entre sí. En ROS se fomenta el uso de múltiples nodos que realicen procesos sencillos por sobre procesos grandes y complejos que abarquen toda la funcionalidad.
- **Master**: el ROS Master se encarga de buscar y registrar los diferentes componentes que interactúan en el sistema. Esto posibilita que diferentes nodos sean capaces de "encontrarse" mutuamente, intercambiar mensajes o invocar servicios. En un sistema distribuido, el master debe ejecutarse solamente en una de las computadoras.
- **Servidor de parámetros**: el servidor de parámetros o *parameter server*, permite mantener la información utilizada en configuración de los nodos almacenada en una ubicación central. Esto permite a cada uno de los nodos a acceder y modificar dichos valores.
- **Mensajes**: los nodos se comunican entre sí mediante mensajes, estructuras de datos cuyos campos pueden editarse y permiten ser enviados entre sí. Existen tipos de mensajes estándares (enteros, flotantes, booleanos, etc.) así como también es posible definir mensajes propios, adaptados a las necesidades de la aplicación.
- **Tópicos**: cada mensaje en ROS es transportado utilizando buses llamados tópicos. Cuando un nodo envía un mensaje a través de un tópico, se puede decir que el nodo está "publicando un tópico". Asimismo cuando un nodo recibe un mensaje a través de un tópico, se puede decir que el nodo está "subscripto al tópico". El nodo publicante y el suscriptor no tienen información sobre su mutua existencia, por lo que es posible que existan nodos que publiquen en tópicos sin suscriptores o viceversa, es decir nodos suscriptos a tópicos sin publicantes.
- **Servicios**: en determinadas aplicaciones, el mecanismo de publicador/suscriptor definido en el ítem anterior podría no ser adecuado. Por ejemplo,

en ciertos casos es necesaria una interacción del tipo solicitud/respuesta. En dichas situaciones un nodo podría solicitar la ejecución de un procedimiento rápido por parte de otro nodo y el envío de una respuesta con el resultado de dicho cálculo.

- **Logging**: ROS provee un sistema de registro o *logging* denominado rosbag (bolsa), que se utiliza para almacenar información publicada en los tópicos activos, como por ejemplo la proveniente de un sensor que podría ser difícil de generar una y otra vez, pero que a su vez resulta necesaria para depurar determinados algoritmos. Un rosbag permitiría en este caso generar la información una única vez para luego reproducirla como si fuese una grabación las veces que resulte necesario.

2.1.3. Herramienta RViz

La herramienta RViz (o ROS Visualization tool) es la herramienta oficial de visualización ROS, que permite representar de manera gráfica la información transmitida a través de los distintos tópicos. Posee soporte nativo para la mayoría de los mensajes estándar y permite además, expandir su funcionamiento mediante *plugins* que posiblitan entre otras cosas, visualizar mensajes definidos por el usuario. La configuración estándar de RViz permite visualizar la configuración de articulaciones (o *joints*) y enlaces (o *links*) de un robot como se muestra en la figura 2.3.

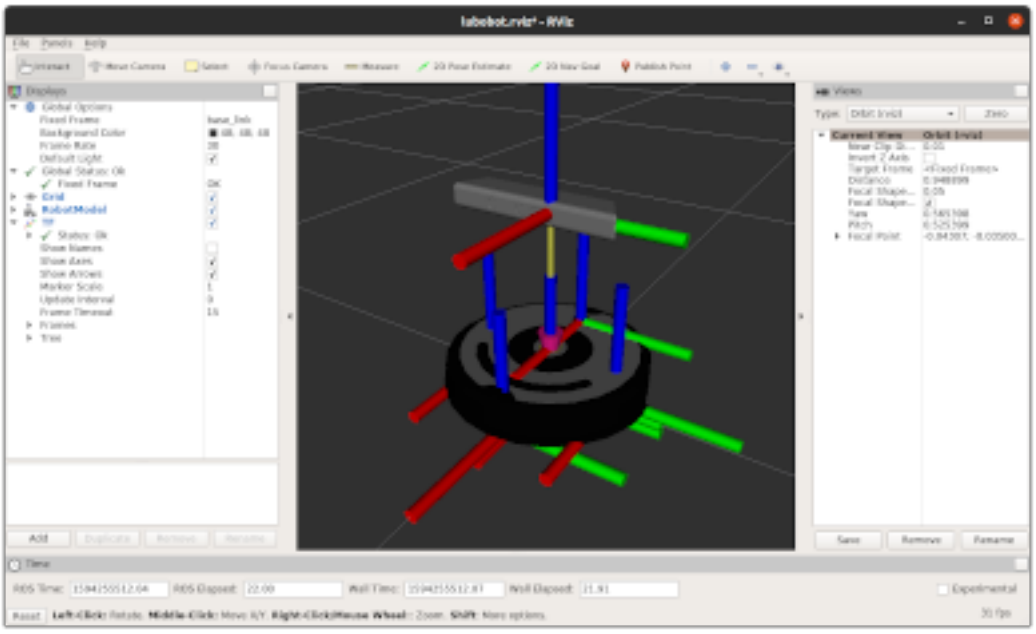

FIGURA 2.3. Interfaz gráfica de RViz donde se muestran los *links* y *joints* del robot Lubobot.

2.1.4. Formato universal de descripción de robots URDF

El formato universal de descripción de robots o *Universal Robot Description System*, comúnmente referido como URDF es un formato estándar para representación de modelos conformados por múltiples piezas conectadas entre sí, como es el caso de brazos robóticos o líneas de ensamblaje. Este es ampliamente utilizado dentro del ecosistema de ROS, plataforma donde vio su origen aunque al día de hoy su uso

se ha extendido a herramientas externas como MATLAB, que permite importar archivos URDF directamente a su *toolbox* de robótica.

2.1.5. Biblioteca rosserial

Es un protocolo para la transmisión serial de mensajes y multiplexación de múltiples tópicos y servicios de ROS sobre un *character device* tal como un puerto UART o un *socket* de red. Además de la definición del protocolo de serialización en si mismo, rosserial se compone de otros dos elementos principales:

- **bibliotecas cliente**: permiten la integración de nodos ROS en diferentes plataformas, apuntando principalmente a sistemas embebidos. Dichas librerias son especializaciones de una clase base, escrita en ANSI C++ para mayor compatiblidad y denominada rosserial_client. Para este trabajo se realizó un *port* de dicha biblioteca a la plataforma STM32CubeHAL.
- **Interfaz con ROS**: las bibliotecas cliente requieren de que haya un nodo corriendo en la computadora *host* que funcione como puente entre el serie y la red de ROS, que se encarga de des-serializar y serializar los mensajes que llegan y se despachan, respectivamente. La participación de los distintos actores involucrados en el uso de rosserial se muestran en la figura 2.4.

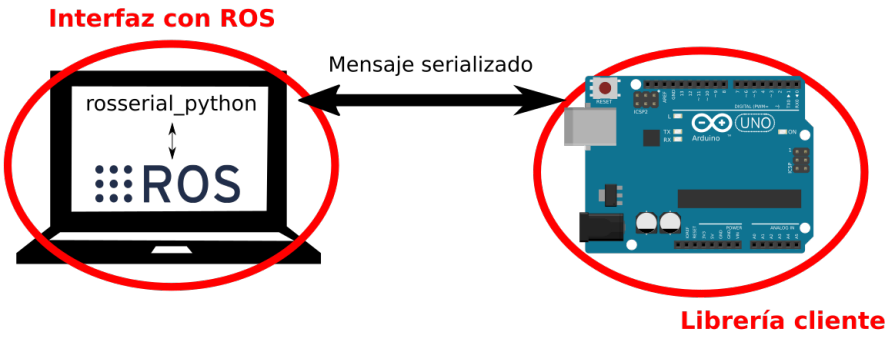

FIGURA 2.4. Interacción entre ROS y rosserial que se ejecutan en un microcontrolador.

2.1.6. Transformación de coordenadas con TF

Dentro del campo de la robótica resulta indispensable trabajar con múltiples sistemas de referencia tanto dentro del mismo robot (a la hora de representar el estado de sus articulaciones) así como para representar su estado en su universo de acción o mundo, como puede ser un mapa. Con la finalidad de y simplificar este tipo de cálculos ROS incorpora el paquete llamado TF.

Esta biblioteca fue diseñada para proveer una manera estándar para mantener el registro de los *coordinate frames* o marcos de coordenadas y de realizar las transformaciones entre si mismas en todo el robot, de modo a que los usuarios puedan consumir la información de manera ordenada y sin la necesidad de ocuparse de generar las transformaciones en forma manual [3].

TF puede operar de forma transparente en sistemas distribuidos, muy típicos en el entorno ROS. Esto significa que toda la información sobre los marcos de coordenadas de un robot se encuentran disponibles para todos los componentes de un sistema ROS en cualquiera de las computadoras que componen el sistema.

En la figura 2.5 se visualiza un robot con múltiples articulaciones. Dado que la pose de estas se encuentra sujeta a cambios en el tiempo, también así lo hacen sus transformaciones. Por este motivo TF se encarga de monitorearlas en todo momento, lo que provee al robot la capacidad de responderse preguntas como las siguientes:

- ¿a dónde se encontraba el *frame* "cabeza" con respecto al *frame* "mundo" hace cinco segundos?
- ¿cuál es la pose del objeto sostenido en mi *gripper* con respecto a mi base?
- ¿cuál es la pose actual del *frame* "base" en el *frame* "mapa"?

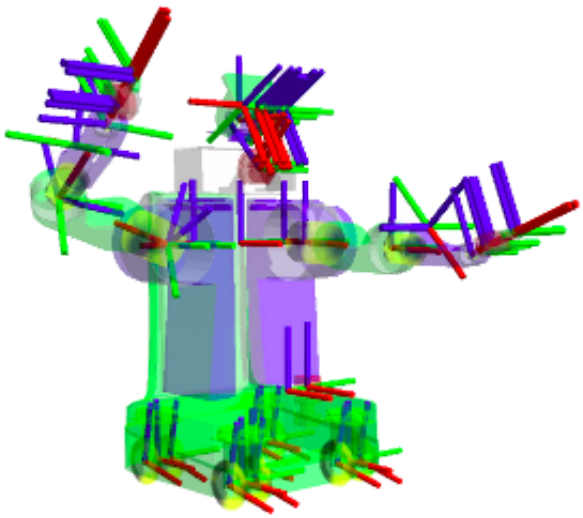

FIGURA 2.5. Ejemplo de los *frames* registrados con TF en el robot PR2, de la compañía Willow Garage.[1]

2.1.7. Paquete de localización robot_localization

El paquete robot_localization es una colección de estimadores no lineales para robots en espacios 2D y 3D, aunque para el aclance de este trabajo solo se hace hincapié en entornos en dos dimensiones.

Cada uno de estos estimadores permite fusionar un número arbitrario de sensores (IMUs, fuentes de odometría, sistemas de localización interna, receptores GPS, etc.) a su salida genera un vector de 15 dimensiones con el estado del robot (x, y, z, $roll$, $pitch$, yaw, $\dot{x}$, $\dot{y}$, $\dot{z}$, $\dot{roll}$, $\dot{pitch}$, $\dot{yaw}$, $\ddot{x}$, $\ddot{y}$, $\ddot{z}$).

Todos los estimadores de estados controlados por este paquete aceptan mediciones provenientes de cualquier sensor de pose siempre y cuando los mismos publiquen en alguno de los tres tipos de mensaje compatibles que se citan a continuación:

[1] Imagen tomada de http://wiki.ros.org/tf?action=AttachFile&do=get&target=frames2.png

- **nav_msgs/Odometry**: posición y orientación, y velocidades lineal y angular.
- **sensor_msgs/Imu**: orientación, velocidad angular y aceleración lineal.
- **geometry_msgs/PoseWithCovarianceStamped**: posición y orientación.
- **geometry_msgs/TwistWithCovarianceStamped**: velocidad lineal y angular.

Para este trabajo se hace uso de los dos primeros tipos de mensajes citados en la lista.

En base a estas mediciones, los estimadores publican los valores de estado filtrados de posición, orientación y velocidades lineal y angular mediante un mensaje del tipo **`nav_msgs/Odometry`** y opcionalmente, valores de aceleración (no utilizadas en el presente trabajo).

El sotware permite a los usuarios decidir cuál de los campos de información proveidos por los sensores van a ser tomados en cuenta para la fusión en el estimador. Por este motivo, como parte de la configuración del paquete es necesario definir una matriz de booleanos para cada una de las fuentes de información o sensores para informar al estimador cuáles son los campos a tener en cuenta. A continuación se muestra la matrix completa de variables requerida para la configuración de cada una de las fuentes:

$$\begin{bmatrix} x & y & z \\ roll & pitch & yaw \\ \dot{x} & \dot{y} & \dot{z} \\ \dot{roll} & \dot{pitch} & \dot{yaw} \\ \ddot{x} & \ddot{y} & \ddot{z} \end{bmatrix}$$

2.1.8. Paquete de navegación ros_navigation_stack

El paquete de navegación de ROS, usualmente referido como *navigation stack*, es un set de algoritmos que hace uso de los sensores y las fuentes de odometría disponibles para controlar al robot es decir, le permiten moverse de un punto conocido a otro del mapa a la vez que mantiene una estimación de su posición en el espacio a cada instante.

Para que un robot pueda hacer uso de este paquete correctamente, es necesario que se satisfaga una serie de requisitos:

- Solo puede utilizarse en robots con ruedas en configuración de tracción diferencial u holonómica. Cabe mencionar que el robot presentado en este trabajo es de configuración diferencial.
- Es necesario que el robot publique la información sobre las relaciones entre las posiciones de todas las articulaciones y sensores que lo componen.
- El robot deberá reportar sus velocidades lineal y angular utilizando un mensaje estándar de ROS.
- Un sensor del tipo LIDaR 2D deberá estar presente en el robot para generar el mapa y para el proceso de localización. Alternativamente, es posible utilizar otros tipos de sensores como cámaras de profundidad o *depth cameras*

o SONAR, siempre y cuando la información recolectada se publique con el tipo de mensaje adecuado.

En la figura 2.6 se puede apreciar cómo se encuentra organizado el paquete de navegación, el que utiliza dos mapas de obstáculos, uno local y otro global, que son construidos a partir de la información generada por el escáner láser en conjunto con el sistema de navegación. El mapa global es más extenso en dimiensiones que el local y está diseñado para el cálculo de trayectorias globales. Por otro lado, el mapa local es usualmente más acotado pero con mayor detalle y su objetivo es el de evadir obstáculos cercanos.

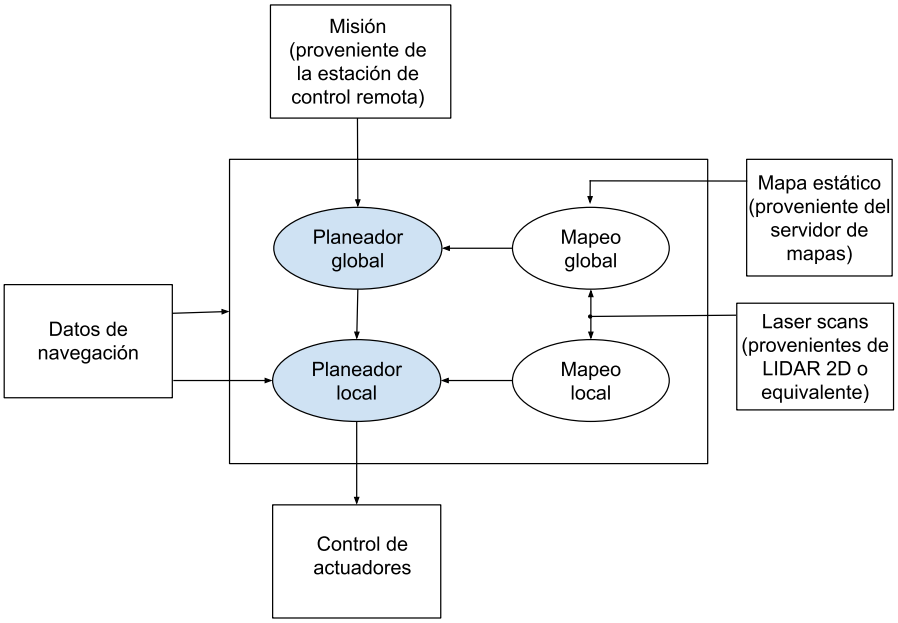

FIGURA 2.6. Configuración típica del paquete de navegación en ROS.

Los mapas global y local son utilizados por los planeadores global y local, respectivamente. El planeador global se encarga de generar un plan para ir de un punto a otro del mapa homónimo. Por otro lado, el local se encarga de controlar los distintos actuadores del robot, por ejemplo las ruedas, para llevar a cabo el plan global pero a la vez que esquiva los obstáculos cercanos que puedan aparecer, incluso si los mismos no se encontrasen registrados en el mapa global. Esto resulta especialmente útil a la hora de esquivar obstáculos nuevos, hasta el momento desconocidos.

2.2. Conceptos de robótica móvil

En esta sección se introduce a algunos conceptos de robótica móvil que resultan de especial importancia para la comprensión en detalle del trabajo expuesto en los siguientes capítulos.

2.2.1. Cinemática de un robot de tracción diferencial

Este tipo de tracción se caracteriza por disponer de dos ruedas situadas sobre el mismo eje horizontal con la particularidad de poder ser comandadas individualmente, es decir, el sentido y velocidad de giro de cada una es independiente de la otra.

Las ventajas de este tipo de movimiento que lo hacen especialmente útil para aplicaciones como las del presente trabajo son las siguientes:

- Los cálculos matemáticos requeridos para describirlo son sencillos de calcular y en consecuencia, de aplicar.
- Hace posible realizar giros de 360 grados alrededor del eje vertical.
- Es la opción elegida por los fabricantes para bases móviles comerciales de bajo costo, como el Roomba 500 descripto en la sección 2.3.

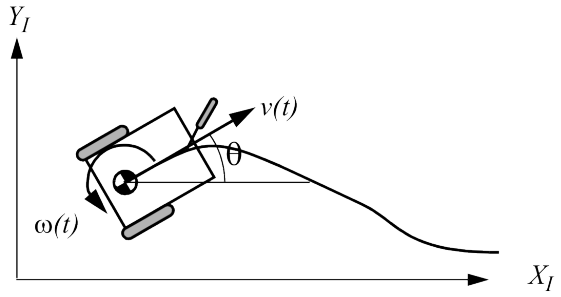

FIGURA 2.7. Descripción de la trayectoria de un robot de tracción diferencial con rueda "loca" o *caster*.

Como se muestra en la figura 2.7, es posible controlar la trayectoria descripta por el robot en un plano (X, Y) mediante la manipulación de las velocidad lineal v y angular w del conjunto [4]. En un sistema de tracción diferencial, las fórmulas que describen la velocidad lineal v y angular w se muestran en la ecuacion 2.1 y 2.2, respectivamente.

$$v = \frac{1}{2}\frac{V_i + V_d}{V_i - V_d} \tag{2.1}$$

$$w = \frac{V_i - V_d}{l} \tag{2.2}$$

donde V_i y V_d representan las velocidades de desplazamiento lineal de las ruedas izquierda y derecha, respectivamente y l es la distancia entre ambas ruedas sobre el eje de giro.

Se generan tres casos particulares dignos de mención:

- Si $V_i = V_d$, se tendrá un movimiento lineal en línea recta por lo que la velocidad angular w será nula.
- Si $V_i = -V_d$, entonces solo habrá velocidad angular w y la velocidad lineal v del conjunto será nula.

- Si $V_i = 0$, entonces se tendrá una rotación en sentido anti-horario alrededor de la rueda izquierda con un radio de giro l. Se verá el mismo efecto para $V_d = 0$, esta vez con rotación en sentido horario.

2.2.2. Odometría basada en encoders

La odometría es el estudio de la estimación de la posición de robots con ruedas durante la navegación, para la que se utiliza información sobre la rotación de las ruedas que permiten estimar cambios en la posición de la base móvil en el tiempo, como se muestra en la figura 2.8.

Este modelo supone que el desplazamiento de las ruedas es traducible al desplazamiento real del robot. Sin embargo, en la práctica esto no es necesariamente cierto debido a la presencia de distintos tipos de errores tales como ruedas desiguales, medidas inexactas o el resbalamiento de las ruedas en la superficie [5]. Estos motivos son los causantes de un error acumulativo no acotado que hacen que este método solo resulte apto para estimaciones en intérvalos de distancia pequeños.

Resulta necesario enriquecer esta estimación mediante su fusión con otras fuentes de información diferentes, tales como las que proveen las unidades de medición inercial, odometría visual mediante cámaras, sensores de profundidad, etc a modo de obtener una estimación final mas precisa. En este trabajo se aborda el proceso de fusión de varios de los sensores mencionados, y permite además, agregar otras fuentes de información con relativa facilidad.

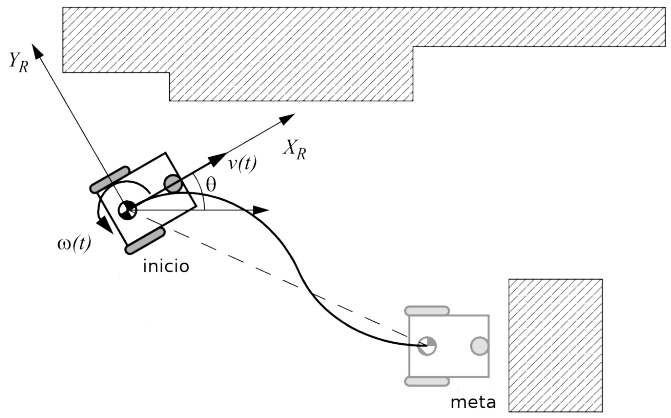

FIGURA 2.8. Trayectoria real descripta por un robot de tracción diferencial.

2.3. iRobot Roomba 500

El Roomba es un robot de limpieza fabricado y comercializado por la compañía iRobot. El primer modelo salió al mercado en el año 2002 y ha recibido siete actualizaciones desde entonces. En cada iteración, se han mejorado aspectos tanto de diseño como funcionalidad y esto le ha permitido mantenerse como el robot de limpieza con más unidades vendidas en el mundo.

Todos los Roomba incluyen una serie de sensores táctiles, ópticos y acústicos que le permiten detectar obstáculos, residuos, así como escalones o desniveles en el piso.

A nivel de locomoción, se catalogan como robots móviles con ruedas y utilizan el tipo de tracción diferencial, que consiste en dos ruedas motrices independientes que le permiten ejecutar giros de 360 grados. Esto es posible sin que el robot deba incurrir en desplazamiento lineal alguno como en el caso de los automóviles, por ejemplo.

La base móvil utilizada para este trabajo consiste en un Roomba de la serie 500 como el mostrado en la figura 2.9, que fue introducido al mercado en el año 2007 y se comercializó hasta el 2017. Actualmente es posible conseguir estos equipos en condicion de usado o remanufacturado a precios muy accesibles comparado al precio de un equipo más actual.

FIGURA 2.9. iRobot Roomba 500, utilizado como base móvil para este trabajo.[2]

2.3.1. Roomba Open Interface

Todos los Roomba lanzados a partir del año 2005 son compatibles con una interfaz de comunicación serial denominada Roomba Open Interface, a la que es posible acceder mediante la conexión a un puerto físico disponible en la placa madre del robot. Esto habilita al usuario a todo tipo de interacción con el hardware del robot, como consultar la lectura de cada uno de sus sensores o comandar sus actuadores. Este protocolo ha sido actualizado a la par de las sucesivas actualizaciones del Roomba para ofrecer nuevas funcionalidades o mejoras y en cada caso, se encuentra acompañado de un documento oficial por parte de iRobot en formato PDF[3].

2.3.2. Conexionado del robot por puerto serie

En la figura 2.10 y la tabla 2.1 se exponen respectivamente, la distribución de pines en el conector y su conexión correspondiente para entablar comunicación con el robot mediante el puerto Mini-DIN.

[2]Imagen tomada de https://uncrate.com/assets_c/2009/04/roomba-560-stretched-thumb-960x640-3177.jpg

[3]Imagen tomada de https://cdn-shop.adafruit.com/datasheets/create_2_Open_Interface_Spec.pdf

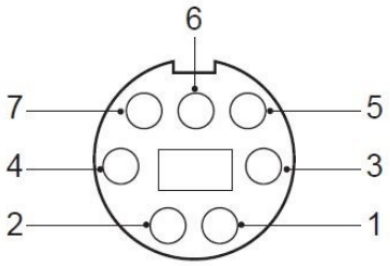

FIGURA 2.10. Distribución de pines en el conector hembra Mini-DIN.

TABLA 2.1. Referencia de pines en conector Mini-DIN hembra.

Pin	Nombre	Descripción
1	Vpwr	Positivo directo de la batería (no regulado)
2	Vpwr	Positivo directo de la batería (no regulado)
3	RXD	0 - 5 VCC Entrada serie
4	TXD	0 - 5 VCC Salida serie
5	BRC	Cambio de *baud-rate*
6	GND	Tierra del robot
7	GND	Tierra del robot

2.4. Placa de desarrollo STM32-NUCLEO

Es una familia de placas de desarrollo elaboradas por la compañía ST. Entre sus características principales se destacan:

- Bajo costo
- Compatibilidad con *shields* de Arduino
- Programador/debugger ST-Link integrado
- biblioteca del tipo HAL con variados ejemplos de código

Con el fin de garantizar la escalabilidad del sistema, para este proyecto se optó por una de las placas mas avanzadas de esta familia, denominada NUCLEO-F746ZG y mostrada en la figura 2.11. Esta provee un microcontrolador ARM Cortex-M7 con FPU de doble precisión, 320 kB de RAM, así como 1 MB de Flash. Ofrece además conexión Ethernet y una amplia cantidad de GPIOs.

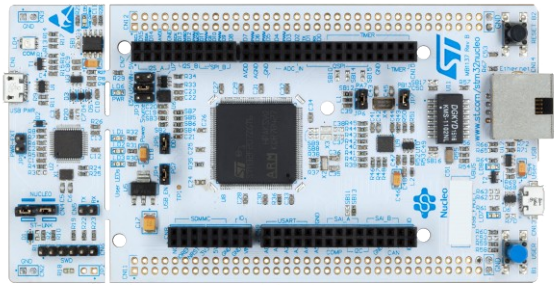

FIGURA 2.11. Placa de desarrollo STM32 NUCLEO-F746ZG elegida para comandar el robot.[4]

2.5. Sensor Kinect 360

El Kinect para Xbox 360 o simplemente Kinect, es un dispositivo de captura de movimiento diseñado por la empresa Microsoft que utiliza tecnología desarrollada por la empresa israelita PrimeSense.

Cuenta con una cámara que captura imágenes en formato RGB, un *array* de micrófonos, así como un sensor de profundidad. Si bien existen otras características extra que podrían mencionarse, vale la pena detenerse en esta última y explicar en qué consiste, puesto que se la utilizó de manera extensiva en el trabajo propuesto para la generación de mapas del robot.

2.5.1. Imagen de profundidad

El sensor de profundidad del Kinect consiste en un proyector de nube de puntos infrarrojos combinado con un sensor CMOS monocromático, similar al de una cámara fotográfica. Funcionando en combinación, estos dos elementos son capaces de captar la información necesaria para que un ASIC genere con ella una imagen de profundidad o *Depth map*, que consiste en un mapa de bits con información sobre la distancia relativa entre el sensor y los objetos detectados. En la figura 2.12 se puede apreciar una imagen de profundidad generada mediante el sensor Kinect en el que se distingue claramente la silueta de una persona en un entorno con diferentes objetos. Las distintas tonalidades de grises visualizadas son una representación de la distancia del objeto al sensor, por lo que para objetos cercanos veremos píxeles de color gris mas claro mientras que para objetos lejanos se verán mas oscuros.

FIGURA 2.12. *Depth map* de la silueta de una persona.

2.6. Unidad de medición inercial

Una unidad de medición inercial o IMU por sus siglas en inglés, es un dispositivo electrónico capaz de medir y reportar la velocidad angular y en algunos casos, el campo magnético que rodea al cuerpo.

Las IMU funcionan detectando la aceleración lineal mediante uno o más acelerómetros y la tasa de rotación usando uno o más giroscopios. Algunos de estos dispositivos incluyen también un magnetómetro que se utiliza comúnmente como una referencia de rumbo.

[4]Imagen tomada de https://www.carminenoviello.com/wp-content/uploads/2015/12/nucleo_144_large-2-660x330.jpg

Las configuraciones típicas de una IMU continenen un acelerómetro, un giroscopio y un magnetómetro para cada uno de los tres ejes de giros y fuerzas del vehículo: cabeceo, alabeo y guiñada, que se muestran en la figura reffig:imu. Para el caso particular del robot móvil propuesto en este trabajo, solo se hace uso de las aceleraciones: angular en el eje Z y lineal en el eje X, respectivamente.

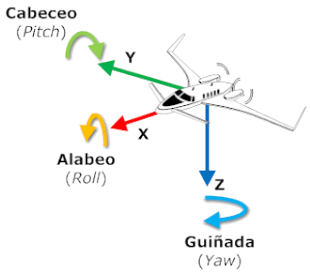

FIGURA 2.13. Ejes de giros y fuerzas de un vehículo.[5]

2.6.1. Sensor MPU6050

El dispositivo MPU-6050 mostrado en la figura 2.14 combina en el mismo chip un giroscopio de 3 ejes con un acelerómetro también de 3 ejes. Mediante lo que el fabricante denomina *Digital Motion Processor* o DMP, esta IMU es capaz de procesar algoritmos de fusión de datos de 6 ejes, usualmente ejecutados en un microcontrolador o DSP mediante un filtro de Kalman o similar. Si bien el chip no incluye un magnetómetro, sí ofrece una interfaz I2C que posibilita conectarla a un magnetómetro externo, de este modo el DMP puede aprovechar la información de campo magnético en en sus algoritmos de fusión para generar resultados más precisos.

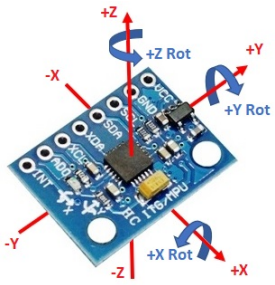

FIGURA 2.14. Sensor MPU-6050 con sus ejes de giros y fuerzas.

[5]Imagen tomada de http://skiras.blogspot.com/2012/10/4copter-conceptos-i-ejes-giros-y-fuerzas.html

Capítulo 3

Diseño e implementación

En este capítulo se exponen las decisiones de diseño tomadas para la elaboración del robot propuesto. Se detallan las dimensiones de las piezas que constituyen el chasis, su esquema de conexiones eléctricas y finalmente la implementación del software y el firmware embebido.

3.1. Estructura mecánica

En esta sección se describe la estructura física del robot y se explican los criterios utilizados tanto para el dimensionamiento de piezas como para su disposición.

3.1.1. Disposición en niveles

Tal como se menciona en la sección 1.2.1, el diseño propuesto toma como referencia a la base móvil *open source* Turtlebot 2. Por este motivo, Lubobot también se compone de una base móvil cilindrica de tracción diferencial la cual funciona como soporte para una estructura escalable en "niveles", gracias a lo que el usuario puede agregar nuevos componentes sin dificultad, según los requisitos particulares de su aplicación. En la figura 3.1 y 3.2 muestran respectivamente, una representación en 3D de la estructura física del robot y una fotografía del montaje real del robot.

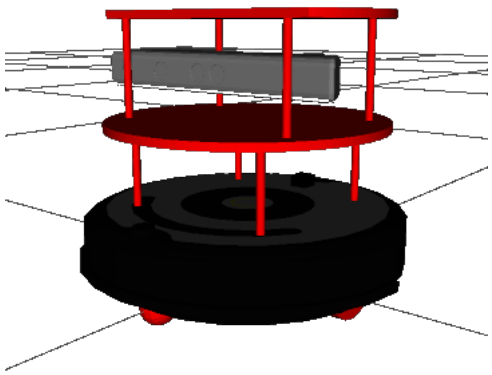

FIGURA 3.1. Vista frontal del modelo URDF de Lubobot representado en RViz.

FIGURA 3.2. Fotografía frontal del montaje del robot Lubobot.

Sobre la base móvil se montaron cuatro varillas roscadas de 5 mm de diámetro y 80 mm de largo, de las cuales las dos frontales fueron fijadas a la agarradera o *handler* del Roomba, mientras que las dos traseras se fijaron directamente al chasis. Dicha disposición fué calculada en base a las áreas del robot que el autor determinó como seguras para su modificación, es decir que no representaban ningún riesgo al correcto funcionamiento de la base móvil.

3.1.2. Disposición de componentes

Los diferentes componentes funcionales del robot fueron dispuestos en la configuración que se aprecia en la figura 3.3, a excepción de la *laptop*, presente solo para fines de ejemplo ya que no forma parte del robot. Para la disposición propuesta se tuvieron en cuenta los siguientes criterios:

- Mantener bajo el centro de gravedad del conjunto.
- Maximizar el espacio libre en el nivel superior para futuros *upgrades*.
- Posicionar la IMU lo mas cerca posible del centro de rotación de la base y lo mas abajo posible.
- Posicionar el sensor Kinect a alrededor de 30 cm del suelo para incrementar su línea de visión.
- Mantener libre de obstrucciones el acceso al botón central del Roomba.

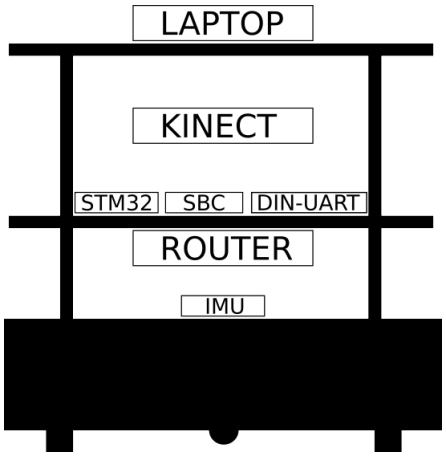

FIGURA 3.3. Distribución de componentes del robot.

3.2. Diagrama en bloques de conexiones

En la figura 3.4 se muestra el diagrama de conexiones entre los distintos componentes electronicos del sistema y se indica además, el protocolo utilizado para cada interacción. Se decidió dotar al robot de un *router* wifi a bordo de modo a facilitar su conexión a una computadora externa, destinada como estación de control para el envío de misiones.

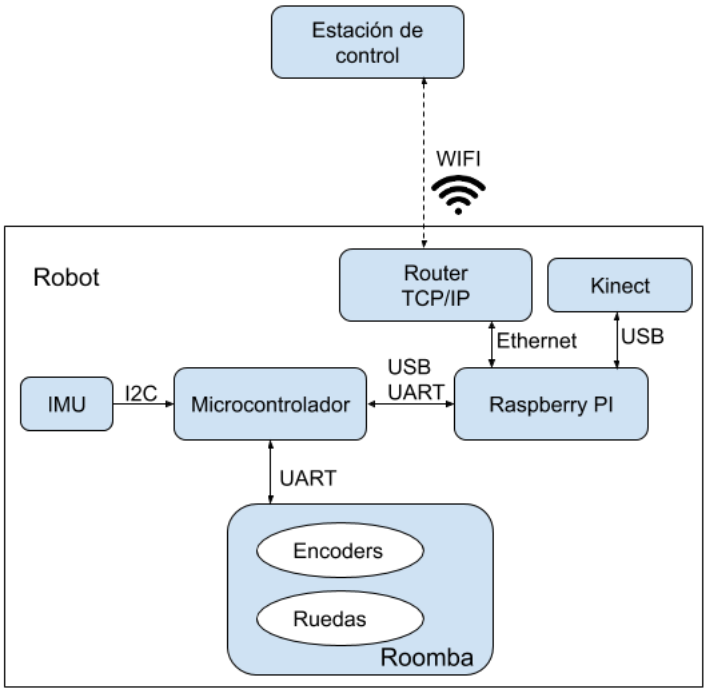

FIGURA 3.4. Diagrama de conexiones de los componentes constitutivos del robot.

3.3. Implementación de firmware

En esta sección se detallan las decisiones de diseño e implementación del firmware, encargado de la interconexión del robot Roomba y la Raspberry PI así como también de la lectura de sensores ajenos a la base móvil original como lo es la IMU.

3.3.1. Distribución de tareas en FreeRTOS

Debido a la necesidad de atender rutinas naturaleza tanto síncrona como asíncrona en el firmware, se decidió utilizar un RTOS para su implementación. Las tareas involucradas se describen a continuación, agrupadas en base a su responsabilidad dentro del sistema.

3.3.2. Interfaz Roomba-microcontrolador

Está formada dos tareas que en conjunto funcionan como *proxy* entre la base móvil Roomba y el resto del sistema. El microcontrolador se ocupa de la serialición y des-serialización de los paquetes intercambiados con el robot mediante el protocolo Open Interface Las tareas responsables del manejo de esta función se describen en el apartado siguiente.

- **Solicitud de lectura de sensores**: realiza la lectura de cada uno de los dos encoders disponibles de manera periódica cada 100 ms, actualizando las variables internas de estado en cada iteración con el valor actualizado.
- **Envío de comandos a actuadores**: realiza el despacho de comandos a los distintos actuadores disponibles en base a las órdenes recibidas desde ROS. Si bien esta tarea puede considerarse de carácter asíncrono, se implementó sobre esta un mecanismo de *watchdog* para evitar que un posible fallo en la comunicación con ROS ponga en peligro la integridad del usuario y del robot. Para esto, la tarea verifica cada 100 ms la existencia de un comando de velocidad nuevo. En su ausencia, envía una orden para detener el robot.

Se escribió una biblioteca en lenguaje C++ bajo el nombre de "Roomba600" que ofrece una interfaz simplificada para la interacción del microcontrolador con el Roomba. Está implementada sobre la capa de abstracción STM32CubeHAL ofrecida por el fabricante ST, que permite el manejo de puertos serie UART con mecanismos de control asíncronos aptos para RTOS.

El protocolo Open Interface permite la interacción con todos los sensores y actuadores disponibles en el robot Roomba. Sin embargo, los requerimientos del presente proyecto se limitan al control de velocidad de ruedas y lectura de encoders, por lo que en la librería presentada se implementan solamente las interfaces para dichas funciones, dejando los demás métodos pendientes de implementación para un trabajo futuro. En el código 3.1 se muestra la interfaz de la clase Roomba600 con los métodos actualmente implementados.

```
1 class Roomba600 {
2 public:
3   /// @brief Constructs Roomba600 object with given parameters
4   /// @param _uart pointer to the STM32CubeHAL UART handler
5   /// @param _brcPort pointer to the STM32CubeHAL BRC port
6   /// @param _brcPin pointer to the STM32CubeHAL BRC pin
7   Roomba600(UART_HandleTypeDef *_uart, GPIO_TypeDef *_brcPort,
```

```
8         uint16_t _brcPin);
9     /// @brief Initializes data stream.
10    void start();
11    /// @brief Resets communication.
12    void reset();
13    /// @brief Stops robot and data stream.
14    void stop();
15    /// @brief Switches to safe mode.
16    void goSafeMode();
17    /// @brief Switches to full mode (be careful).
18    void goFullMode();
19    /// @brief Switches to passive mode.
20    void goPassiveMode();
21    /// @brief Sets independent velocity set-points for each wheel in m/s.
22    /// Relies on internal PID loop from the Roomba.
23    void driveVelocity(int16_t leftVel, int16_t rightVel);
24    /// @brief Sets independent PWM values to each wheel.
25    void drivePWM(int16_t leftPWM, int16_t rightPWM);
26    /// @brief Pauses the stream of sensor data reads.
27    void pauseStream();
28    /// @brief Resumes the stream of sensor data reads.
29    void resumeSteam();
30    /// @brief Retrieves left encoder value.
31    uint16_t readLeftEncoder();
32    /// @brief Retrieves right encoder value.
33    uint16_t readRightEncoder();
34  }
```

CÓDIGO 3.1. Interfaz de la clase Roomba600 que implementa el protocolo Open Interface.[1]

3.3.3. Interfaz microcontrolador-ROS

Esta interfaz involucró la migración de la biblioteca rosserial, mencionada en la sección 2.1.5, que hace posible entablar una comunicación serial asíncrona entre el microcontrolador y el una computadora corriendo ROS mediante su protocolo de mensajes estándar.

- **Spin de ROS** esta tarea se encarga de llamar al método **`spin`** de la biblioteca rosserial manera periódica. Esta cumple la función de *heartbeat* para la comunicación con la computadora, donde ambas partes se indican mutuamente si existen mensajes o servicios que transmitir.
- **Suscriptor de ROS** recibe los mensajes y solicitudes de servicios que fueron enviados desde ROS al microcontrolador y llama a las funciones de *callback* requeridas para cada uno de ellos.
- **Publicador de ROS** despacha los mensajes y solicitudes de servicios que salen del microcontrolador.

Para la migración de la biblioteca rosserial fué necesario implementar una clase cuyas interfaces sean compatibles con la definición mostrada en el código 3.2.

```
1
2  class STM32Hardware {
3    public:
4      /// @brief Constructs STM32Hardware object with given parameter
```

[1]Definición de la clase en el repositorio del proyecto https://github.com/apojomovsky/lubobot/blob/master/firmware/proyecto_unificado/create_2/Inc/create2.h

```
5       /// @param huart_ pointer to the STM32CubeHAL UART handler
6       STM32Hardware(UART_HandleTypeDef *huart_);
7       /// @brief Initializes comunication.
8       void init();
9       /// @brief Reads from input buffer.
10      int read();
11      /// @brief Writes data of a given length.
12      /// @param data pointer to bytes array.
13      /// @param length number of bytes to transmit.
14      void write(uint8_t* data, int length);
15      /// @brief Returns time elapsed since startup in milliseconds.
16      unsigned long time();
17  };
```

CÓDIGO 3.2. Interfaz de la clase STM32Hardware requerida por la biblioteca rosserial.[2]

3.4. Implementación de software

En esta sección se entra en detalle sobre la organización del código y archivos que acompañan al robot propuesto en este trabajo.

3.4.1. Metapaquete Lubobot para ROS

En esta sección se explica la estructura del meta-paquete Lubobot, que contiene las herramientas mínimas necesarias para comandar, modificar e inspeccionar el robot con ROS.

Como se describió en la sección 2.1.1, se considera una buena práctica el utilizar un metapaquete para contener varios paquetes que estan relacionados entre sí. Siguiendo estas recomendaciones, el software que acompaña al robot propuesto se organiza de la siguiente manera:

- **lubobot_description** incluye los archivos de descripción del robot en formato URDF y los archivos de visualización para RViz.
- **lubobot_msgs** incluye los mensajes utilizados para intercambiar información entre el firmware y ros, a través de rosserial.
- **lubobot_drivers** incluye el código de fuente para los nodos o programas que monitorean los sensores del robot, así como los que envían comandos a los actuadores.

3.4.2. Nodo re-publicador de IMU "lubo_imu_relay"

Se encuentra suscripto a los mensajes del tipo **`lubo_imu`**, compuesto por las lecturas crudas de orientación y aceleración angular provenientes del microcontrolador. El programa se encarga de procesarlos a medida que llegan, agregandoles información de tiempo y secuencia y luego los republica en el formato de mensaje estándar de ROS para unidades de medición inercial[3].

[2]Clase STM32Hardware implementada en el proyecto https://github.com/apojomovsky/lubobot/blob/master/firmware/proyecto_unificado/ros_lib/STM32Hardware.h

Cabe preguntarse por qué de no evitar este nodo y publicar mensajes con el tipo estándar directamente desde el microcontrolador. Hay un par de motivos específicos que son válidos no solo para este mensaje en particular, sino también para la gran mayoría de los mensajes intercambiados con el microcontrolador:

- los mensajes estandar de ROS poseen un *footprint* en memoria poco adecuado para dispositivos con poca memoria RAM.
- el bus UART utilizado para la comunicación con el microcontrolador ofrece un ancho de banda ajustado que se satura rápidamente con mensajes del tipo estándar.

3.4.3. Nodo publicador de odometría "lubo_odom_node"

Está suscripto a los mensajes del tipo **`lubo_encoders`** que provienen del microcontrolador. El mensaje se compone de las lecturas de los encoders izquierdo y derecho de la base Roomba. Cabe mencionar que las lecturas de ambos encoders deben tomarse en simultaneo para el correcto cálculo de la odometría, es por esto que el mensaje utilizado incluye ambas lecturas.

El microcontrolador publica los mensajes a una frecuencia por defecto de 20 hz, pero este y otros parámetros son también configurables desde la estación de control en *runtime*. Esto se logra gracias al mecanismo de configuración **`rosparam`**, que ahorra la necesidad de recompilar el código cada vez que es necesario actualizar parámetros.

3.4.4. Configuración del paquete ros_localization

Así como se describió en la subsección 2.1.7, es necesario definir una matriz de booleanos para cada uno de los sensores o fuentes de información que consume el estimador. Para el presente trabajo, se utilizaron las mediciones provistas por dos fuentes distintas:

- unidad de medición inercial.
- odometría generada a partir encoders.

Para la configuración del paquete se utilizó un archivo del tipo YAML que se carga de manerara automática al servidor de parámetros de ROS al inicio de cada ejecución. Las partes mas importantes de este archivo residen en la configuración de campos a ser tomados en cuenta por el estimador para cada una de las fuentes de información. Una versión reducida de esta configuración se muestra en el bloque de código 3.3.

[3]Definición de mensaje estándar de ROS para Imu http://docs.ros.org/kinetic/api/sensor_msgs/html/msg/Imu.html

```
1
2  # Odometry configuration
3  odom_frame: odom
4  base_link_frame: base_link
5
6  # Input topic name for odometry
7  odom0: /odom
8
9  # For the odometry source, we only consider vx, vy and vyaw
10 odom0_config: [false, false, false,
11                 false, false, false,
12                 true,  true,  false,
13                 false, false, true,
14                 false, false, false]
15
16 # IMU configuration
17
18 # Input topic name for IMU
19 imu0: /imu/data_raw
20
21 # For the IMU source we consider only yaw, vyaw and ax
22 imu0_config: [false, false, false,
23                false, false, true,
24                false, false, false,
25                false, false, true,
26                true, false, false]
```

CÓDIGO 3.3. Configuraciones del filtro proveído por el paquete ros_localization .[4]

3.4.5. Configuración del paquete move_base

3.4.6. Configuración del paquete amcl

3.5. Herramientas ofrecidas al usuario

De acuerdo a los objetivos del presente trabajo, el desarrollo del robot móvil Lubobot está acompañado de una serie de herramientas y documentos con los que se pretende facilitar el proceso de adopción de la plataforma por nuevos usuarios.

3.5.1. Entorno de desarrollo encapsulado

La instalación de un entorno de desarrollo compatible con un proyecto puede tornarse una tarea muy frustrante para el usuario si no se tomen los los necesarios. Casos como el del uso de un sistema operativo total o parcialmente incompatible, diferencias de versiones de las bibliotecas requeridas e instaladas, etc. son solo algunos de problemas con los que el usuario promedio podría encontrarse.

Con la intención de simplificar al máximo el proceso de instalación de dependencias, se provee de un entorno de desarrollo encapsulado utilizando la tecnología

[4] Archivo de configuración YAML completo https://github.com/apojomovsky/lubobot/blob/master/lubobot/config/ekf.yaml

de *containers* mediante la plataforma Docker[5]. Esta utilizad permite la generación de entornos de trabajo aislados del sistema operativo *host* mediante el uso de un archivo especial denominado Dockerfile, en el que se detalla una lista de acciones a seguir para generar el entorno de trabajo. En el presente trabajo, se incluyen los siguientes archivos:

- **Dockerfile**: continene las definiciones para generar el entorno de trabajo con todas las dependencias necesarias para la ejecución del software del robot en ROS.
- **build_docker.sh**: *script* encargado de generar el entorno de desarrollo, solo es necesario ejecutarlo una única vez.
- **run_docker.sh**: *script* encargado de lanzar el entorno de desarrollo. El usuario deberá llamarlo cada vez que desee arrancar el sistema.

Se proveen instrucciones específicas sobre como utilizar y expandir la configuración inicial propuesta de esta herramienta en la Wiki del proyecto.

3.5.2. Documentación en formato Wiki

Se ofrece en conjunto con el repositorio Git[6], una sección aparte dedicada exclusivamente a la documentación del robot. Para este fin se utilizó la funcionalidad de *Wiki* ofrecida por la plataforma Github.

En la figura 3.5 se muestra la página principal de la *Wiki* en su primera iteración, donde se realiza una breve presentación del proyecto, seguido de *links* de interés con tutoriales y especificaciones técnicas.

Se decidió utilizar este formato de documentación en favor de incluir todo en el archivo **`README.md`** por una suma de motivos entre los que se destacan:

- organización de archivos en un sub-repositorio proveído de manera automática por Github, con historial de cambios separado del repositorio principal.
- estructura de archivos múltiples organizados en forma de árbol, que no solo facilitan la búsqueda de información por parte del usuario sino que también facilita las tareas de mantenimiento.
- generación automática de índice en forma de *sidebar* en el sitio que facilita la navegación entre las páginas que componen el sitio.

[5]Página web del proyecto Docker https://www.docker.com

[6]Repositorio principal del proyecto Lubobot https://github.com/apojomovsky/lubobot

[7]Wiki del proyecto Lubobot https://github.com/apojomovsky/lubobot/wiki

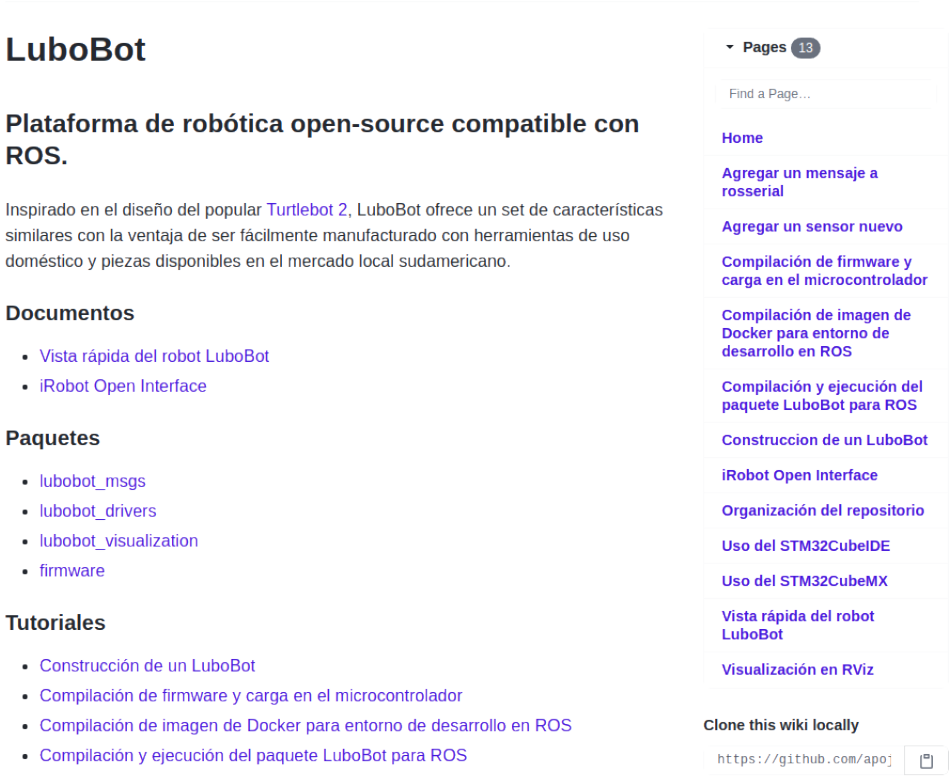

FIGURA 3.5. Página principal de la *wiki* de LuboBot.[7]

Capítulo 4

Ensayos y resultados

En este capítulo se detallan las pruebas que fueron realizadas sobre los distintos módulos de hardware y software que componen al robot así como los resultados de las pruebas de campo que se llevaron a cabo.

4.1. Pruebas unitarias en el robot

En esta sección se describen las pruebas funcionales ejecutadas sobre cada uno de los módulos que componen el robot de manera aislada. Esto permitió encontrar problemas puntuales y reducir las fuentes de error para las pruebas de integración que le siguieron.

4.1.1. Validación de conexión bidireccional Roomba-microcontrolador

Descripción del setup del microcontrolador usando 2 conexiones UART simultaneas.

Envío de comandos de: conexión, desconexión.

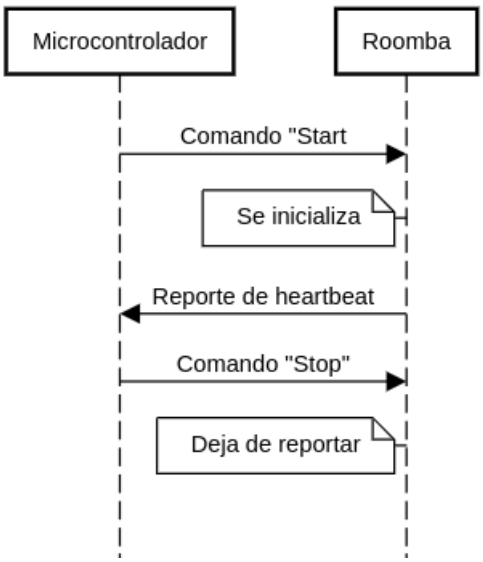

FIGURA 4.1. Diagrama de secuencias ejecutadas durante el test de comunicación entre el microcontrolador y el Roomba.

4.1.2. Validación de conexión bidireccional microcontrolador-ROS

Descripción de la conexión con ROS usando un mensaje de "ping" con una doble confirmación. Pequeño diagrama explicando el método.

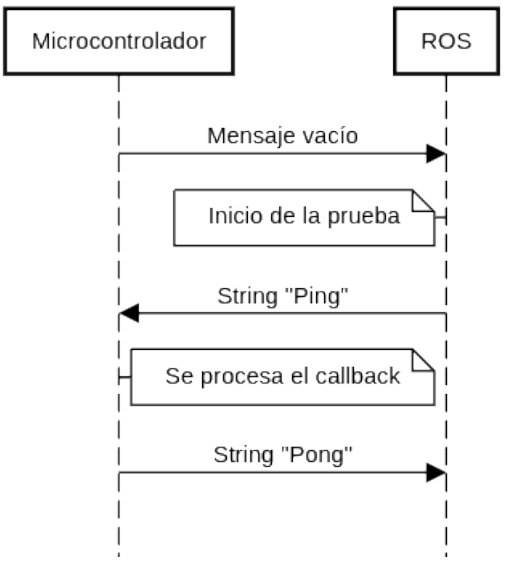

FIGURA 4.2. Diagrama de secuencias ejecutadas durante el test entre el microcontrolador y una PC con ROS.

4.1.3. Validación de frenado de emergencia

El Roomba funciona de una manera tal que si este recibe un comando de velocidad, el mismo será aplicado al robot por un tiempo indefinido. De modo a prevenir riesgos, se implementó en el controlador un sistema de frenado de emergencia que responde a los siguientes pasos:

- Se activa en cuanto se recibe un comando de velocidad distinto de cero.
- Se activa una tarea periódica, ejecutada cada 500 ms en donde se comprueba la existencia de al menos un mensaje nuevo en el tópico **`/cmd_vel`**.
- En caso que no haya ningún mensaje, la tarea envía la orden de detener el robot.

4.1.4. Validación de desplazamiento efectivo del robot

El Roomba implementa su propio sistema de control interno para llevar sus ruedas hasta el *set point* requerido por los comandos de velocdad. En visto que esta funcionalidad es crítica para el sistema, se desarrolló un caso de prueba especial para velocidades tanto positivas como negativas el cual se describe a continuación:

- mediante un script de Python conectado a ROS se envió un comando, el cual establece la velocidad de ambas ruedas a 0.1 m/s por una duración de 10 s es decir, el equivalente a un desplazamiento lineal de 1 m.

TABLA 4.1. Resultados de validación de desplazamiento efectivo del robot

Movimiento	Desplazamiento esperado	Desplazamiento obtenido	Error obtenido
Lineal positivo 1 m	1,0 m	0,98 m	2 %
Lineal negativo 1 m	-1,0 m	-0,96 m	4 %

TABLA 4.2. Resultados de validación de lectura de encoders

Movimiento	Ticks esperados		Ticks obtenidos		Error obtenido	
	Izq.	Der.	Izq.	Der.	Izq.	Der.
Lineal positivo 1 m	2250	2250	2201	2137	2,22 %	5,28 %
Lineal negativo 1 m	2250	2250	2194	2103	2,55 %	6,99 %

- mediante una cinta métrica se compararon los valores esperados con el desplazamiento lineal real.

Un análisis cualitativo de los resultados expuestos en la tabla 4.1 evidencia que el sistema de control de velocidades interno del Roomba asume que la masa del robot es constante y no toma en cuenta el peso del hardware agregado. Como consecuencia los comandos para mover el robot muestran un porcentaje de error importante.

Como medida paliativa se decidió no utilizar el robot en modo reversa para las tareas de mapeo de modo a reducir el error acumulativo.

4.1.5. Validación de lectura de encoders

Dado que es posible obtener las lecturas crudas de los encoders del robot, se realizó la comprobación siguiente de manera similar a la prueba anterior:

- mediante un script de Python conectado a ROS se envió un comando de velocidad, el cual establece la velocidad de ambas ruedas a 0.1 m/s por una duración de 10 s es decir, el equivalente a un desplazamiento de 1 m.
- mediante los encoders se obtuvo la cantidad de *ticks* o cuentas ocurridas durante dicho periodo.
- se calculó el desplazamiento estimado mediante la siguiente fórmula obtenida a partir del documento de especificaciones del fabricante [6]:

$$Disancia = nTicks/4498,9125 \tag{4.1}$$

Los resultados expuestos en la tabla 4.2 muestran ciertas similitudes con los resultados de la prueba anterior en el sentido que el robot reporta un movimiento mas "corto" que el que le fue comandado. Dado que el número de *ticks* de la rueda izquierda es mayor que el de la derecha, es correcto asumir que el robot describió un movimiento curvilíneo con radio de giro hacia la derecha, lo que pudo comprobarse por el autor al observar la posición del robot al final de la prueba.

4.1.6. Validación de lecturas crudas de la IMU

Esta prueba consistió en graficar las lecturas de las seis variables registradas por el sensor durante a una secuencia de tres movimientos a los que fue sometido.

Para esta prueba se partió de un estado inicial con la IMU quieta, con el plano definido por sus ejes (X, Y) en posición normal al vector de aceleración de la gravedad (que apunta al centro de la tierra).Para todas las pruebas se utilizó la convención "North-East-Down" [1]que el sensor adopta por defecto.

Basado en las lecturas del acelerómetro de la figura 4.3 se describen las observaciones realizadas:

- en la primera columna se observa el vector de aceleración de la gravedad de aproximadamente 9,8 m/s^2 apuntando en la dirección negativa del eje Z.
- en la segunda columna se observa el mismo vector de aceleración, esta vez transladado en dirección del eje X negativo.
- finalemente en la tercer columna se observa al vector transladado en dirección al eje Y negativo.

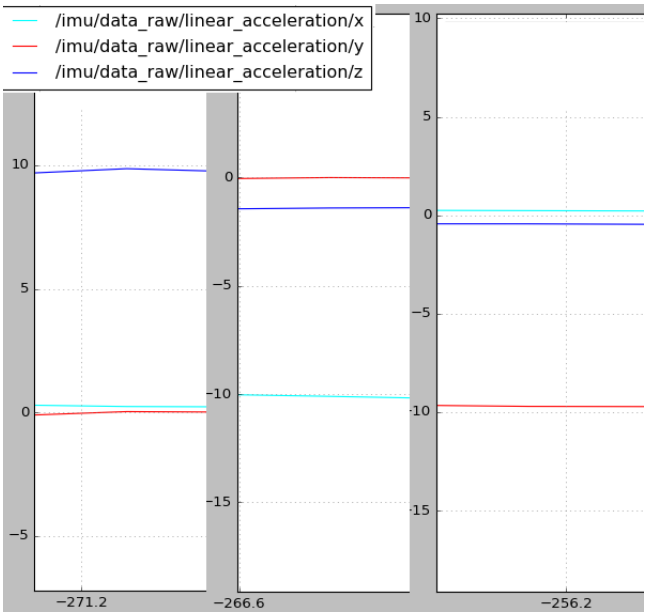

FIGURA 4.3. Gráfico de aceleraciones lineales sobre cada uno de los ejes registrados durante las tres posiciones de prueba.

Para las gráficas del el giroscopio se procedió de manera similar que con el acelerómetro. La diferencia en estas pruebas radica en que los valores medidos por el giroscopio son instantáneos, ya que los mismos corresponden a las velocidades angulares alrededor que aparecen alrededor de los ejes **durante** los movimientos efectuados, y desaparecen al dejar de mover la IMU. Por este motivo se presentan seis gráficas que fueron distribuidas en las figuras 4.4 y 4.5 de izquierda a derecha, donde se muestran los valores adoptados por los tres ejes del giroscopio durante los movimientos descriptos a continuación:

[1]Coordenadas locales del plano tangente NED: https://en.wikipedia.org/wiki/Local_tangent_plane_coordinates

1. giro alrededor del eje Y en sentido antihorario.
2. giro alrededor del eje Y en sentido horario.
3. giro alrededor del eje X en sentido antihorario.
4. giro alrededor del eje X en sentido horario.
5. giro alrededor del eje Z en sentido antihorario.
6. giro alrededor del eje Z en sentido horario.

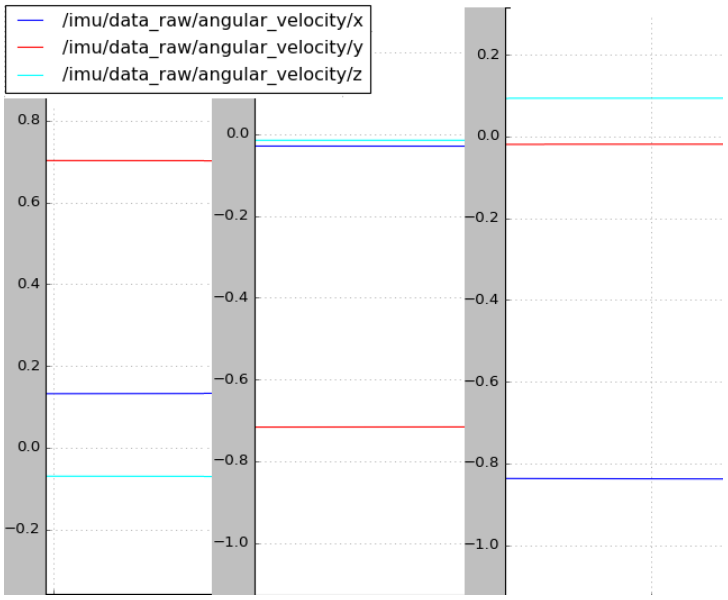

FIGURA 4.4. Gráfico de velocidades angulares durante los movimientos 1, 2 y 3.

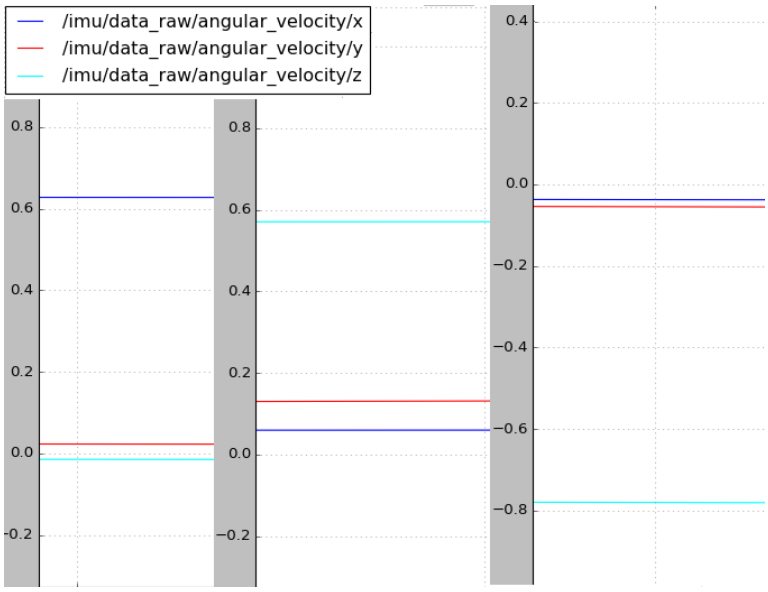

FIGURA 4.5. Gráfico de velocidades angulares durante los movimientos 4, 5 y 6.

TABLA 4.3. Resultados de validación de odometría de encoders

Variable	Medición esperada	Medición reportada	Error obtenido
Posición en X	1 m	1,04 m	4 %
Posición en Y	1 m	0,8 m	20 %
Orientación en Z	-0,785 rad	-0,729 rad	7,73 %

4.2. Pruebas unitarias en la PC

En esta sección se describen las pruebas realizadas sobre los resultados arrojados por los componentes de software ejecutados en la PC. Estos se utilizan para procesar los datos crudos arrojados por los sensores y generar información en un formato aprovechable por ROS.

4.2.1. Validación de cálculo de odometría con encoders

Esta prueba tuvo como objetivo validar el correcto funcionamiento del programa **`lubo_odom_node`**, el cual consume las lecturas de encoders y realiza con estas una estimación de la posición y orientación del robot en un plano (X, Y).

Para la validación de esta característica, la prueba realizó siguiendo los siguientes pasos:

1. se marcó la pose inicial del robot en el suelo.
2. se comandaó el robot con un *joystick* a una velocidad aproximada de 0,1 m/s.
3. se movió el robot aproximadamente 1 m hacia adelante.
4. se hizo girar el robot aproximadamente 90 grados.
5. se movió nuevamente el robot aproximadamente 1 m hacia adelante.
6. se midió la distancia lineal entre el punto de inicio y el punto final del robot.
7. se obtuvieron las componentes en X e Y del vector de distancia trazado en el punto anterior.
8. se compararon estas medidas con los valores calculados por la odometría, los cuales se exponen en la tabla 4.3.

4.2.2. Validación de estimador de odometría con encoders + IMU

Gracias a la aplicación de un Filtro de Kalman Extendido por parte del paquete **`robot_localization`**, fue posible combinar la información provista por el cálculo de odometría de encoders junto con el estimador de pose de la IMU.

Esta pruebe consistió en comparar los resultados de la prueba de odometría de encoders pura del punto anterior con la estimación de odometria resultante de combinar este valor con la pose de la IMU. El procedimiento ejecutado para esta prueba es el mismo que se utilizó en el punto anterior, y los resultados se exponen en la tabla 4.4.

Gracias a que ambos tests se encargan de reproducir el mismo escenario fue posible comparar ambos resultados y obtener el porcentaje de mejora ofrecido por

TABLA 4.4. Resultados de validación del estimador de odometría mediante la fusión de datos de encoders e IMU.

Variable	Medición esperada	Medición reportada	Error obtenido
Posición en X	1 m	1,01 m	1 %
Posición en Y	1 m	0,98 m	2 %
Orientación en Z	-0,785 rad	-0,786 rad	0,13 %

TABLA 4.5. Resultados obtenidos al comparar las lecturas arrojadas por la odometría de encoders "pura" y la odometría resultante de la fusión de encoders e IMU.

Variable	Técnica 1	Técnica 2	Porcentaje de mejora
Posición en X	1,04 m	1,01 m	~300 %
Posición en Y	0,8 m	0,98 m	~1000 %
Orientación en Z	-0,729 rad	-0,786 rad	~6000 %

la fusión de datos sobre el error absoluto. Los resultados de esta comparación se pueden apreciar en la tabla 4.5.

4.3. Pruebas de integración en campo

En esta sección se describen las pruebas funcionales ejecutadas sobre cada uno de los módulos que componen el robot de manera aislada. Esto permitió encontrar problemas puntuales y reducir las fuentes de error para las pruebas de integración que le siguieron.

4.3.1. Generación de mapa con gmapping

Esta prueba consistió en la generación de un mapa del tipo *occupancy grid* o mapa de ocupación, el cual consiste en una representación similar a la de un plano en dos dimensiones como los utilizados en arquitectura. Los puntos en negro representan una sección "ocupada" mientras que los píxeles blancos representan un área libre.

Para que el resultado obtenido a partir de este procedimiento sea lo suficientemente preciso como para utilizarse en la tarea de localizar al robot fue necesario que la fuente de odometría sea muy precisa, motivo por el cual resultó indispensable el uso de la fuente de odometría filtrada con información de la IMU.

En las figuras 4.6 se puede apreciar el mapa obtenido a partir de la odometría de encoders "pura". Este primer mapa resulta totalmente inutilizable para tareas de navegacion puesto que no refleja la morfología real del recinto. Por el contrario, el mapa de la figura 4.7 se generó utilizando la fuente de odometría filtrada con IMU. Aún con sus detalles, este último mapa resultó suficiente para las tareas de localización y navegación del robot.

FIGURA 4.6. Mapa obtenido utilizando odometría de encoders "pura".

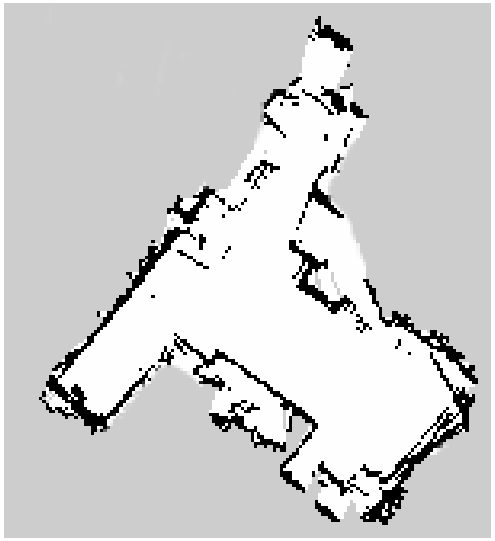

FIGURA 4.7. Mapa obtenido utilizando odometría filtrada.

4.3.2. Localización en mapa con amcl

Tal como se describió en la sección 3.4.6, amcl es un paquete para ROS que provee al robot la capacidad de encontrar su posición y orientación de un mapa previamente provisto. Esta prueba consistió en la utilización del mapa generado de la sala de estar del departamento del autor. Este fue provisto a la herramienta amcl mediante el servidor de parámetros de ROS y luego se procedió a teleoperar el robot mediante un *joystick* inalámbrico. El procedimiento utilizado fue el siguiente:

1. se colocó el robot en una pose aleatoria en el mapa como se muestra en la figura 4.8.
2. por medio del *joystick* se hizo girar al robot en círculos en sentido horario hasta completar dos vueltas completas de 360 grados a una velocidad angular aproximada de 0,5 rad/s.

3. se aplicó el mismo procedimiento para un giro en sentido anti-horario.

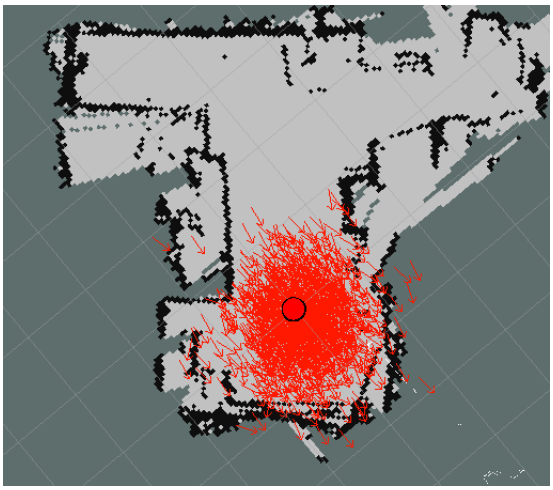

FIGURA 4.8. Captura de pantalla de RViz con el robot aún sin localizarse. Las flechas rojas muestran las posibles poses del robot en el mapa.

El resultado final se puede apreciar en la figura 4.9. En esta imagen, tanto la posición en X e Y del robot con respecto al mapa han sido estimados y así también su orientación respecto al eje Z.

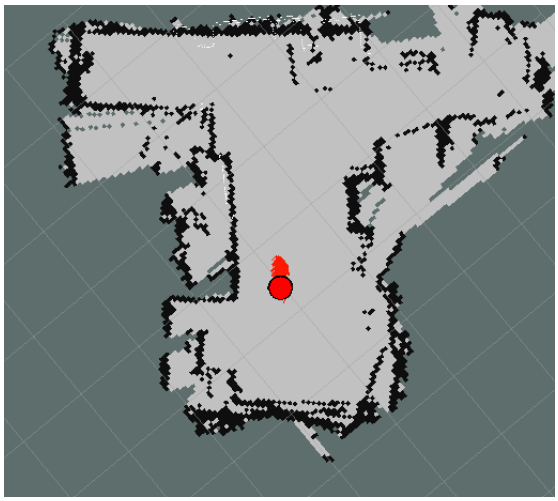

FIGURA 4.9. Captura de pantalla de RViz con el robot localizado. En este momento las flechas rojas han convergido hacia una misma región donde se estima que se encuentra el robot.

4.3.3. Navegación en mapa con move_base

Así como se describe en la sección 3.4.5, move_base provee la funcionalidad necesaria para mover al robot dentro del mapa provisto, para lo cual la herramienta se encarga de calcular una ruta de navegación desde el punto en que el robot se encuentra al momento de enviarse la orden y otro punto cualquiera del mapa.

Esta prueba consistió en indicar a la herramienta que deseabamos mover el robot de un punto a otro en el mapa. Para esto fue necesario cumplir con ciertos requisitos previos:

1. un mapa estático debió ser provisto al servidor de mapas de ROS.
2. el robot debió encontrarse previamente localizado en el mapa.

El envío de consigna se realiza mediante la herramienta RViz, en la cual se representa mediante un vector de color verde en el mapa. Su punto de origen representa la posición deseada mientras que su dirección representa la orientación final del robot. En las figuras 4.10 y 4.11 se puede apreciar al robot en su punto de inicio al momento de recibir la orden de navegación y al robot ya en su punto de destino, respectivamente.

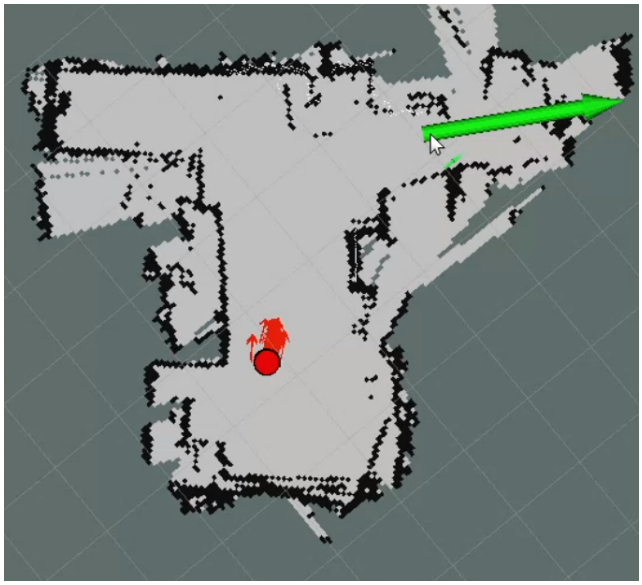

FIGURA 4.10. Captura de pantalla de RViz con el robot localizado al momento de recibir una orden de navegación.

FIGURA 4.11. Captura de pantalla de RViz con el robot en su punto de consigna.

Capítulo 5

Conclusiones

En este capítulo se mencionan los aspectos más releventes del trabajo realizado, se analizan los resultados obtenidos y se identifican los pasos a seguir para una siguiente iteración del proyecto.

5.1. Conclusiones generales

En este trabajo se completó el desarrollo e implementación de un prototipo de robot autónomo así como la documentación requerida para su adopción por parte de estudiantes de grado universitario.

Se implementó de manera satisfactoria el prototipo mecánico basado en un robot de limpieza Roomba 500, así como el firmware requerido para comunicarse con el mismo mediante protocolo Open Interface. Así también, se provee una capa de comunicación con el framework de robótica ROS y un conjunto de paquetes listos para conectarse al robot desde una computadora.

El trabajo ha cumplido satisfactoriamente los puntos principales planteados en la planificación:

- migración de la biblioteca rosserial e i2clib al HAL del microcontrolador.
- implementación de un algoritmo para generar la odometría del robot en base a las lecturas de los encoders.
- configuración de los paquetes necesarios para permitir la navegacion del robot con ROS dentro de la residencia del autor.
- implementación funcional del robot propuesto.
- redacción de una Wiki hosteada en Github con instrucciones para terceros sobre como fabricar un nuevo robot así como para utilizar el software provisto.

Asimismo, ciertos objetivos planteados inicialmente debieron ser modificados por diversos motivos tales como complejidad excesiva o problemas de rendimiento. Se describen a continuación algunos de ellos:

- la utilización del sistema operativo NuttX para la implementación del firmware. Este se terminó reemplazando con FreeRTOS debido a que este último se encuentra oficialmente soportado por el fabricante ST, motivo que facilitó de manera considerable la tarea de la implementación del firmware.

- la utilización de mensajes estándares de ROS a nivel del microcontrolador. Estos debieron ser reemplazados por unos diseñados específicamente para la tarea.

5.2. Próximos pasos

5.2.1. Hardware

La gran mayoría de los objetivos propuestos pudieron cumplirse, aún así, se identificaron algunos espacios de mejora donde podría generarse un beneficio para el usuario final tales como:

- utilización el protocolo TDP para la comunicación de rosserial con ROS en vez del actual UART sobre USB.
- reeplazar el sensor Kinect por un LIDaR 2D económico.
- reemplazar la IMU con una que ofrezca magnetómetro integrado.

5.2.2. Software

El cambio de UART a TCP permitiría, entre otros beneficios, el de contar con un ancho de banda mayor y latencias menores a los actuales. Esto posibilitaría la utilización de mensajes nativos de ROS en el microcontrolador, que originalmente debieron ser reemplazados por la necesidad de limitar el ancho de banda.

5.2.3. Documentación

Como se mencionó anteriormente, a este trabajo lo acompaña una wiki, la cual se encuentra hosteada en Github junto al código del robot. Con respecto a la documentación se plantea la posiblidad de agregar más tutoriales a la Wiki donde se aborden cómo utilizar otros paquetes populares del ecosistema ROS en el robot.

Bibliografía

[1] Melonee Wise y col. *Fetch & Freight : Standard Platforms for Service Robot Applications*. 2016.

[2] 101st United States Congress. *Americans with Disabilities Act of 1990*. 1.ª ed. United States Congress, 1990. URL: https://www.ada.gov/pubs/adastatute08.pdf (visitado 09-04-2020).

[3] T. Foote. *tf: The transform library*. 2013.

[4] Roland Siegwart e Illah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. USA: Bradford Company, 2004. ISBN: 026219502X.

[5] Sol Pedre Thomas Fischer Matías A. Nitsche. *Fusión de encoders de cuadratura, sensores inerciales y magnéticos para la localización de robots móviles*. 2014. URL: https://robotica.dc.uba.ar/public/papers/fischer2014.pdf (visitado 18-07-2020).

[6] iRobot Corporation. *iRobot Create 2 Open Interface (OI) Specification based on the iRobot Roomba 600*. 2015. URL: https://cdn-shop.adafruit.com/datasheets/create_2_Open_Interface_Spec.pdf.