

**UNIVERSIDAD DE COSTA RICA**  
**Facultad de Ingeniería**  
**Escuela de Ingeniería Eléctrica**

**IE0499 – Proyecto Eléctrico**

**Implementación de mapeo y localización simultánea  
para un robot móvil con ruedas mecanum.**

por

**Stuart Leal Quesada**

**Ciudad Universitaria Rodrigo Facio**

**Agosto de 2018**



# **Implementación de mapeo y localización simultánea para un robot móvil con ruedas mecanum.**

por

**Stuart Leal Quesada**

B53777

**IE0499 – Proyecto Eléctrico**

Aprobado por

---

Ing. Federico Ruiz Ugalde, Dr.rer.nat

*Profesor guía*

---

Ing. Israel Arbaiza, Lic.

*Profesor lector*

---

Ing. Fabián Abarca Calderón, M.Sc.

*Profesor lector*

Agosto de 2018



# Índice general

Índice general	v
Índice de figuras	vii
Índice de cuadros	x
<b>1 Introducción</b>	<b>1</b>
1.1. Alcance . . . . .	2
1.1.1. Equipo . . . . .	2
1.1.2. Herramientas . . . . .	3
1.1.3. Resultados . . . . .	3
1.2. Justificación . . . . .	3
1.3. Objetivos . . . . .	3
1.3.1. Objetivo general . . . . .	3
1.3.2. Objetivos específicos . . . . .	4
1.4. Metodología . . . . .	4
<b>2 Nota teórica</b>	<b>5</b>
2.1. Movilidad . . . . .	6
2.1.1. Ruedas . . . . .	6
2.1.2. Motores . . . . .	8
2.1.3. Encoder . . . . .	8
2.2. Visión . . . . .	9
2.2.1. Profundidad . . . . .	9
2.2.2. Textura . . . . .	10
2.3. Autonomía . . . . .	10
2.3.1. Mapeo . . . . .	11
2.3.2. Localización . . . . .	12
2.3.3. Navegación . . . . .	14
<b>3 Infraestructura Existente</b>	<b>17</b>
3.1. Carrito omnidireccional . . . . .	18

3.2. Motores . . . . .	18
3.3. Controladores . . . . .	19
3.4. Sistema de desarrollo . . . . .	21
3.5. Hokuyo . . . . .	22
3.6. Control de PS4 . . . . .	23
<b>4 Diseño</b>	<b>25</b>
4.1. Justificación de diseño . . . . .	26
4.1.1. Hardware . . . . .	26
4.1.2. Software . . . . .	26
4.2. Diagrama eléctrico . . . . .	27
4.3. Código del stm32f411 . . . . .	28
4.3.1. Biblioteca de comunicación . . . . .	28
4.3.2. Código del Encoder . . . . .	31
4.3.3. Control PID . . . . .	32
4.3.4. Protocolo de comunicación entre Discovery y PC . . . . .	33
4.3.5. Odometría global del robot . . . . .	34
4.3.6. Diagrama completo del código . . . . .	36
4.4. Código de la computadora principal . . . . .	37
4.4.1. Sensores láser . . . . .	37
4.4.2. Odometría . . . . .	37
4.4.3. Árbol de Tf . . . . .	38
4.4.4. Comandos de velocidad de la base . . . . .	39
<b>5 Implementación</b>	<b>41</b>
5.1. Kinect . . . . .	42
5.1.1. Instalación del software . . . . .	42
5.1.2. Calibración del Kinect . . . . .	43
5.1.3. Pruebas de funcionamiento . . . . .	43
5.1.4. Retos . . . . .	46
5.2. Hokuyo . . . . .	46
5.2.1. Instalación del software . . . . .	46
5.2.2. Pruebas de funcionamiento . . . . .	47
5.2.3. Retos . . . . .	47
5.3. Conexión de un STM32F4 y un roboclaw . . . . .	47
5.3.1. Conexión . . . . .	48
5.3.2. Retos . . . . .	50
5.4. Encodificador de cuadratura . . . . .	51
5.4.1. Conexión . . . . .	51
5.4.2. Programación . . . . .	52
5.4.3. Retos . . . . .	52
5.5. Controlador PID . . . . .	54

5.5.1.	Implementación . . . . .	55
5.5.2.	Sintonización . . . . .	55
5.5.3.	Retos . . . . .	55
5.5.4.	Gráficas . . . . .	55
5.6.	Ensamble de la base . . . . .	58
5.6.1.	Retos . . . . .	60
5.6.2.	Pruebas de movimiento . . . . .	60
5.7.	Protocolo de comunicación hacia la PC . . . . .	60
5.7.1.	Prototipos y retos . . . . .	60
5.7.2.	Libopencm3-plus . . . . .	61
5.7.3.	Retos . . . . .	61
5.8.	Implementación del paquete en ROS . . . . .	61
5.9.	Cálculo de la odometría . . . . .	63
5.9.1.	Retos . . . . .	63
5.10.	Pruebas y corrección de errores . . . . .	64
5.11.	DualShock 4 en ROS . . . . .	64
5.12.	Implementación de Navegación . . . . .	66
5.12.1.	Configuración . . . . .	67
5.12.2.	Resultados del mapeo . . . . .	68
5.12.3.	Retos . . . . .	70
<b>6</b>	<b>Conclusiones y recomendaciones</b>	<b>75</b>
6.1.	Conclusiones . . . . .	75
6.2.	Recomendaciones . . . . .	76
<b>A</b>	<b>Actobotics 638276</b>	<b>77</b>
<b>B</b>	<b>STM32F4 Encoder</b>	<b>79</b>
<b>C</b>	<b>Roboclaw Instrucciones</b>	<b>83</b>
<b>Bibliografía</b>		<b>87</b>

# Índice de figuras

2.1.	Diagrama de componentes de un robot omnidireccional. Autoría propia. . . . .	6
------	--	---

2.2.	Ruedas omnidireccionales Mecanum, con $\alpha = 45^\circ$ . Tomado de [2]. . . . .	7
2.3.	Diagram simplificado de un motor DC. Tomado de [7] . . . . .	8
2.4.	Ejemplo de un encoder utilizando un Hall Sensor, y la señal producida por un encoder de cuadratura. Tomado de [8]. . . . .	9
2.5.	Imagen de una captura de un sensor de profundidad. Tomado de [15]. . . . .	10
2.6.	Mapa del edificio ACES en la universidad de Texas, cuarto piso. Tomado de [9]. . . .	11
2.7.	Proceso de mapeo en tiempo real. Tomado de [9]. . . . .	12
2.8.	Diagrama del proceso de localización utilizando únicamente la odometría de las ruedas. Tomado de [6]. . . . .	13
2.9.	Diagrama del proceso de estimación de la posición realizado por ACML. Tomado de [6].	13
2.10.	Ejemplo de un uso del global planner en ROS, para obtener una posible ruta. Tomado de [4]. . . . .	15
3.1.	Diagrama básico de la implementación de la parte eléctrica del carrito omnidireccional. Autoría propia. . . . .	18
3.2.	Muestra de los pines de un Roboclaw 2x15A fabricado por IonMotion. Tomado del manual del fabricante. . . . .	20
3.3.	Diagrama de las funciones que debe realizar el microcontrolador STM32F411. Autoría propia. . . . .	21
3.4.	Imagen de un ejemplo del sensor Hokuyo, tomado de la página de ventas robots-hop.com. . . . .	22
3.5.	Imagen de un control de PS4 dualshock. Tomado de la página de ventas BestBuy.com	23
4.1.	Revisión de configuración para el stack de navegación. Tomado de ROS. . . . .	27
4.2.	Diagrama de las conexiones eléctricas entre los dispositivos. Autoría propia. . . . .	29
4.3.	Ejemplo de configuración de un puerto USART utilizando la biblioteca libopencm3. Autoría propia. . . . .	31
4.4.	Ejemplo de inicialización del systick utilizando la biblioteca libopencm3. Autoría propia. . . . .	31
4.5.	Diagrama del PID a programar utilizado para controlar la velocidad de cada motor. Autoría propia. . . . .	32
4.6.	Diagrama de la actualización de la posición global. Autoría propia. . . . .	35
4.7.	Esquema de lo que contiene el archivo principal del programa a correr en el microcontrolador. Autoría propia. . . . .	36
4.8.	Ejemplo de la utilidad de Tf. Tomado de [5]. . . . .	38
4.9.	Ejemplo de un launch file en ROS que habilita un árbol tf. Autoría propia. . . . .	39
4.10.	Diagrama de las interacciones entre el nodo de ROS y el driver de la conexión USB. Autoría propia. . . . .	39
5.1.	Kinect One (V2) utilizado para realizar pruebas. Autoría propia. . . . .	42
5.2.	Imagen de calibración de prueba. Tomado de [18]. . . . .	43
5.3.	Resultado del comando <code>kinect2_bridge</code> . Autoría propia. . . . .	44

5.4.	Prueba de la salida del kinect, utilizando el visor por defecto. Autoría propia. . . . .	45
5.5.	Esquema generado por RQT_GRAPH que muestra las topicos disponibles por Iai Kinect. Autoría propia. . . . .	46
5.6.	Inicialización correcta del módulo hokuyo_node en ROS. Autoria propia. . . . .	47
5.7.	Diagrama generado por <i>RQT_GRAPH</i> , demostrando la salida de hokuyo_node. Autoría propia. . . . .	48
5.8.	Visualización de la salida del sensor láser Hokuyo utilizando Rviz. Autoría propia. .	48
5.9.	Conexión serial en los puertos USART de ambos dispositivos. Autoría propia. . . . .	49
5.10.	Batería de 12V utilizada para alimentar a los controladores Roboclaw. Autoría propia.	50
5.11.	Encodificador de cuadratura. Autoría propia. . . . .	51
5.12.	Cables del motor DC Actobotics. Autoría propia. . . . .	52
5.13.	Cables de motor Actobotics. Tomado del manual de usuario. . . . .	53
5.14.	Reparación de cableado en un motor, debido a daños en las conexiones. Autoría propria. . . . .	54
5.15.	Tiempo versus la posición del motor en ticks del encoder, ante un cambio en la referencia en 0.5s. Autoría propia. . . . .	56
5.16.	Tiempo versus la velocidad del motor en rev/s, ante un cambio en la referencia en 0.5s. Autoría propia. . . . .	57
5.17.	Acople y rueda a utilizar durante el ensamble. Autoría propia. . . . .	58
5.18.	Proceso incial del ensamble. Autoría propia. . . . .	58
5.19.	Soporte de las ruedas. Autoría propia. . . . .	59
5.20.	Etiquetado de las conexiones. Autoría propia. . . . .	59
5.21.	Plataforma ensamblada totalmente. Autoría propia. . . . .	60
5.22.	Instrucciones de instalación e instalación de dependencias. Autoría propia. . . . .	62
5.23.	Visualización de las relaciones entre tópicos y nodos levantados en ROS. Autoría propia.	63
5.24.	Muestra de la salida del nodo <i>ps4-ros</i> . Autoría propia. . . . .	65
5.25.	Uso del control de PS4 para enviar comandos hacia el robot de velocidad. Autoría propia. . . . .	66
5.26.	Algunos de los requisitos necesarios para la navegación con ROS. Tomado de [14]. .	68
5.27.	Mapeo realizado en el cuarto de la cocina del INII. Autoría propia. . . . .	69
5.28.	Fotografía real de la cocineta en la que realizó el mapeo anterior. . . . .	70
5.29.	Mapeo realizado en el ala Sur del INII. Autoría propia. . . . .	71
5.30.	Fotografía real del pasillo en el que se realizó el mapeo anterior. . . . .	72
5.31.	Mapeo realizado en la esquina a las afueras del ARCOS-LAB. Autoría propia. . . . .	73
5.32.	Fotografía real del pasillo en el que se realizó el mapeo anterior. . . . .	74

# Índice de cuadros

3.1.	Información más relevante de los motores Actobotics, a ser utilizados. Autoría propia.	19
3.2.	Información más relevante de los controladores Roboclaw 2x15A. Autoría propia.	19
3.3.	Tabla con las funciones de los pines en un Roboclaw 2x15A fabricado por IonMotion. Tomado del manual del fabricante.	20
3.4.	Especificaciones del sensor Hokuyo. Autoría propia.	22
4.1.	Contenido de un mensaje del tipo “Odometry” en el entorno ROS. Autoría propia.	38
5.1.	Tabla de dos mediciones realizadas para comprobar el cálculo de odometría en la base. Autoría propia.	64
5.2.	Medidas comparativas entre el mapa generado por la navegación y la realidad	69

# CAPÍTULO 1

## INTRODUCCIÓN

La robótica es la ciencia de percibir y manipular el mundo físico através de dispositivos mecánicos controlados por computadoras [17]. Hasta hoy la robótica ha logrado perfeccionar los movimientos precisos milimétricamente, y las tareas repetitivas. Sin embargo, fuera de una línea de producción, estas ventajas no son útiles, por ejemplo en un hogar los ambientes son dinámicos y el robot por lo tanto debe poseer características cognitivas [10].

El laboratorio de investigación ARCOS-Lab (Autonomous Robots and Cognitive Systems) se enfoca primariamente en desarrollar e implementar nuevas técnicas en dispositivos mecánicos con el fin de satisfacer algunas de las necesidades mencionadas anteriormente.

Actualmente (2018) el ARCOS-Lab está trabajando en la construcción de un robot humanoide, el cuál se desea que pueda reconocer su ambiente, y realizar tareas dinámicas como cocinar. La navegación es uno de los aspectos más importantes que se debe desarrollar para cumplir el fin propuesto. Por lo tanto, el presente proyecto plantea implementar un algoritmo llamado SLAM (Simultaneous localization and mapping) en la navegación dos-dimensional de un robot omnidireccional con ruedas mecanum.

Actualmente, el ARCOS-Lab ha desarrollado varios prototipos de robots omnidireccionales. Sin embargo, a través de las iteraciones se han ido perfeccionando aspectos, como el diseño de la base, y las piezas necesarias para crear una base omnidireccional estable, robusta y diseñada con precisión.

De los primeros diseños que se realizó, el ensamble de la base se realizó en una base de madera en la cuál los montajes para los motores no se diseñaron con computadora. Esto causa problemas en los movimientos omnidireccionales del robot, por lo que el diseño de la base se debió realizar con una computadora, y cortar con precisión en una cortadora láser.

Este proyecto pretende ensamblar eléctricamente la base del carro omnidireccional, utilizando las piezas ya diseñadas, impresas, y cortadas, y dotarlo de la capacidad de navegación dos dimencional. Por lo tanto, viene a completar la posibilidad de que con este robot se realice navegación autónoma.

## 1.1. Alcance

Se pretende implementar SLAM en robots pequeños denominados “*mobile robots*” dentro del contexto del ARCOS-Lab. Los mismos son plataformas de aproximadamente  $50\text{cm} \times 20\text{cm}$ . Fueron construidos para implementar los algoritmos de navegación, y realizar investigación en esta área, con el fin de perfeccionar el sistema que se utilizará para el robot humanoide. Las plataformas poseen en esencia el mismo sistema de ruedas omnidireccionales mecanum del robot humanoide.

En principio, es difícil experimentar con la base del robot humanoide desde un inicio. Primero está la limitación de que la misma plataforma está en un constante proceso de desarrollo, lo que quiere decir que se le hacen mejoras constantemente. Esto dificulta la disponibilidad de la base. Además, en los casos en que la plataforma se controle incorrectamente, los daños que pueda ocasionar a una persona o los objetos dentro del laboratorio son superiores a los que podrían ocasionar las plataformas más pequeñas.

El producto final de este proyecto eléctrico es proporcionar los diseños de código y ensamble, así como de conexiones eléctricas para los componentes existentes, para ensamblar un robot omnidireccional que listo para realizar navegación autónoma, por medio del sistema ROS (Robot Operating System).

### 1.1.1. Equipo

A continuación se hace una breve descripción del equipo principal utilizado para la construcción de las plataformas omnidireccionales utilizadas.

- *Motores DC*: Los motores DC utilizados para cada una de las cuatro ruedas de la plataforma omnidireccional. Son motores de 12V, con un consumo de aproximado 0,53A sin carga, y un codificador de cuadratura que produce 3415 pulsos por revolución. El motor logra alcanzar hasta 118rpm según la hoja de fabricante.
- *Roboclaw 2x15A*: Este controlador está diseñado para manejar dos motores DC de máximo 15A cada uno, por lo que se utilizarán dos de estos controladores para manejar los cuatro motores.
- *STM32F4*: Se necesita un controlador con capacidad de mandar los comandos de control hacia el Roboclaw para poder comandar la plataforma correctamente. Se utilizará un microcontrolador STM32F411-disco para esta tarea. La odometría será calculada en este microcontrolador, en vez del Roboclaw para mayor precisión.
- *Raspberry PI*: Se utiliza este microcontrolador, como la unidad de procesamiento principal en donde se controlarán los periféricos por medio del sistema de ROS. A esta tarjeta se conectarán el Kinect y el STM32F4.

- *Kinect 2*: Este será el sensor utilizado para la odometría visual. Será utilizado principalmente para generar la información del mapa, y posteriormente la ubicación del robot dentro del mismo mapa.

### 1.1.2. **Herramientas**

Se describen las herramientas que se utilizarán en el desarrollo del proyecto.

- *ROS*: El desarrollo e implementación de SLAM se realizará en ROS, puesto que es un sistema que permite la modularización de los diferentes componentes del robot. Facilita la conexión entre los diferentes periféricos. Posee librerías que contienen el código necesario para implementar SLAM.
- *Libopencm3*: Biblioteca para desarrollar el código en C del microcontrolador STM32F411.
- *Libopencm3-plus*: Biblioteca desarrollada por el Arcos-LAB que provee funcionalidades extra a *Libopencm3*.
- *Roboclaw serial communication library*: Esta Biblioteca es provista por la empresa Roboclaw y contiene una implementación del protocolo de comunicación de los controladores de los motores. Este código se usó como base para desarrollar la biblioteca utilizada en el microcontrolador.

### 1.1.3. **Resultados**

El resultado final esperado es la implementación física del algoritmo a las plataformas ya existentes del laboratorio.

## 1.2. **Justificación**

Este proyecto pretende brindar la base sobre la cuál nuevos proyectos pueden trabajar en nuevos algoritmos de navegación. Además, pretende fijar un primer paso para la implementación de navegación en el robot humanoide.

## 1.3. **Objetivos**

### 1.3.1. **Objetivo general**

Dotar a un robot omnidireccional con ruedas mecanum la capacidad de mapeo y localización simultánea.

### 1.3.2. Objetivos específicos

Para el desarrollo de este proyecto se establecieron los siguientes objetivos:

- Desarrollar el software de control para los motores de las ruedas y los sensores de odometría.
- Desarrollar la infraestructura de comunicación entre el Raspberry Pi y el cuerpo del robot (motores y odometría).
- Implementar el software de comunicación y visualización del sensor de profundidad.
- Integrar todos los módulos del sistema (software y hardware) y realizar pruebas de conectividad y funcionamiento para cada módulo del sistema.
- Implementar el sistema de SLAM en la computadora principal.
- Determinar y ejecutar experimentos que permitan validar el funcionamiento completo de todo el sistema para realizar pruebas de mapeo y localización simultánea.

## 1.4. Metodología

1. Conectar el Kinect con una computadora corriendo Linux e utilizar ROS (Robot Operating System) para ver la imagen producida por la cámara.
2. Conectar el sensor láser Hokuyo a una computadora corriendo Linux e utilizar ROS para interpretar la imagen en formato LaserScan.
3. Conectar el controlador de motores RoboClaw con un STM32F4 y hacer pruebas de comunicación.
4. Diseñar el código necesario para leer el codificador de cuadratura que poseen los motores.
5. Diseñar un controlador PID para la velocidad del motor.
6. Realizar el ensamblaje de la base.
7. Realizar pruebas de movimiento, e implementar el protocolo de comunicación en el STM32f4.
8. Confeccionar el módulo que conecta el STM32F4 con ROS.
9. Implementar el código necesario para la odometría de los motores.
10. Realizar pruebas de funcionamiento, y corregir problemas.
11. Utilizar un Joystick DualShock 4 en ROS y utilizarlo para mover el robot.
12. Implementar el algoritmo de gmapping.
13. Implementar el algoritmo de localización y navegación amcl.

## CAPÍTULO 2

### NOTA TEÓRICA

Un robot omnidireccional es una plataforma que se puede mover autónomamente. Son utilizadas en industrias de manufactura flexibles y en ambientes de servicio [2]. Hay tres aspectos importantes que posee un robot omnidireccinal. La figura 2.1 ayuda a ilustrar las tres características básicas de un rotot omnidireccional.

A continuación, se procede en las siguientes secciones a clarificar algunos conecepts necesarios para entender cómo se consiguen estas características en un robot móvil.

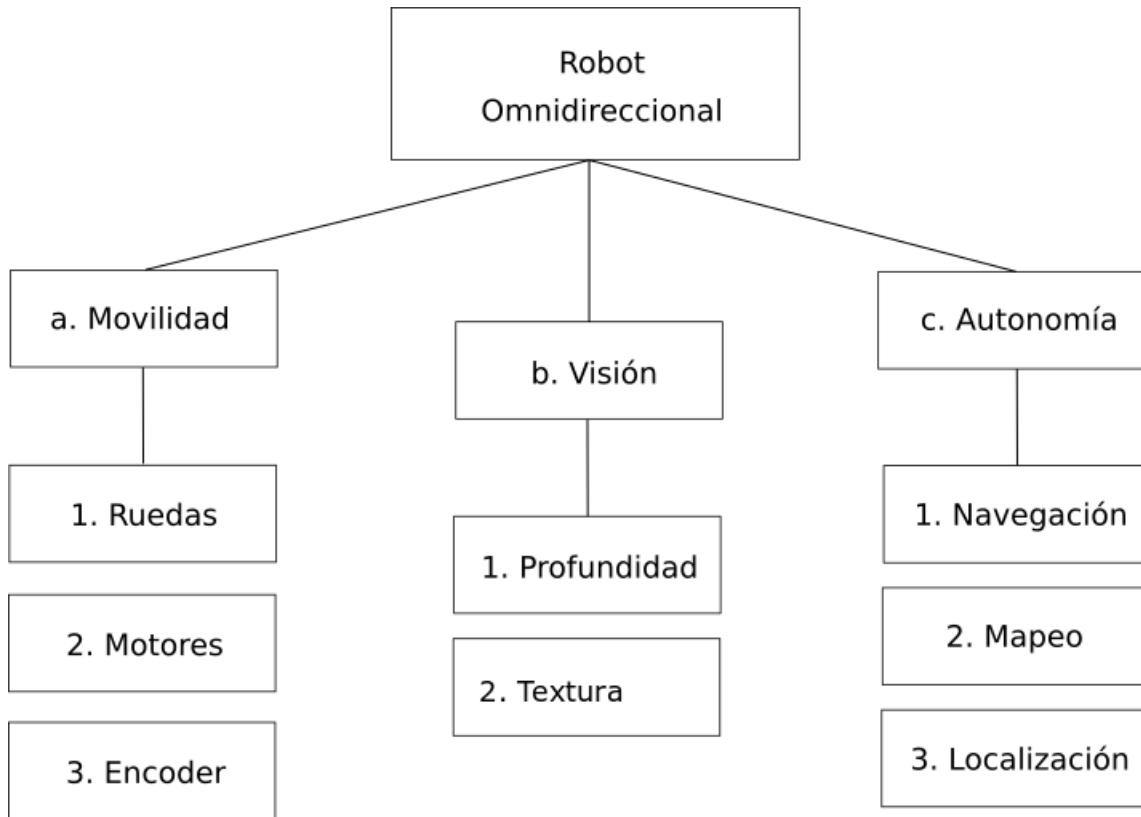


Figura 2.1: Diagrama de componentes de un robot omnidireccional. Autoría propia.

## 2.1. Movilidad

### 2.1.1. Ruedas

Existen varios aspectos a considerar cuando se diseña un robot: movilidad, control y posicionamiento. La primera hace referencia a la cantidad de movimientos que un robot puede realizar para llegar a una configuración final. Deben ser capaces de alcanzar cualquier posición y cualquier orientación en su plano de movimiento. Esto quiere decir que el marco del robot debe poseer tres coordenadas independientes del plano general de movimiento [2].

El movimiento se puede realizar con un robot que tiene dos grados de libertad (se puede mover hacia adelante y atrás, con un ángulo de dirección), pero hay que maniobrar. Es más fácil realizar esta tarea si un robot tiene tres grados de libertad. Esto quiere decir, se puede mover hacia adelante y hacia atrás, izquierda derecha, y rotar. De esta manera, la movilidad aumenta, puesto que la cantidad de movimientos posibles que un robot puede realizar para llegar a una configuración final aumenta [2].

Existe un tipo de ruedas, conocidas como ruedas omnidireccionales, que están compuestas

de pequeños rodillos que habilitan el deslizamiento de la rueda en cierta dirección. En particular, las ruedas mecanum, como las que se muestran en la figura 2.2, son un caso especial de las ruedas omnidireccionales. Los rodillos están en cierto ángulo  $\alpha$  de desfase, que permite realizar los movimientos izquierda-derecha con una combinación de fuerzas específica para cada ruedas.

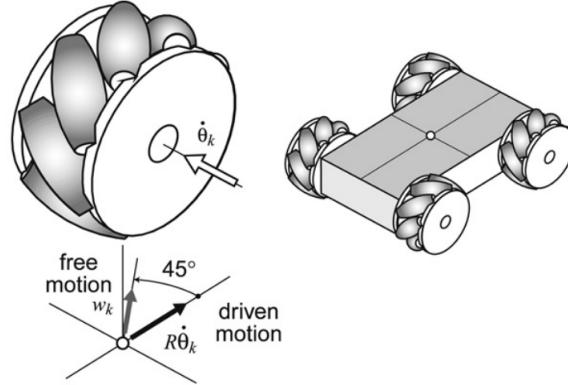


Figura 2.2: Ruedas omnidireccionales Mecanum, con  $\alpha = 45^\circ$ . Tomado de [2].

La combinación de las diferentes velocidades en cada uno de los motores, producen los tres grados de libertad en el marco del robot. Específicamente, la kinemática del robot la define el autor [16]. A continuación, se muestran las ecuaciones que definen el movimiento del robot:

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} 1 & -1 & -(l_x + l_y) \\ 1 & 1 & (l_x + l_y) \\ 1 & 1 & -(l_x + l_y) \\ 1 & -1 & (l_x + l_y) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ w_z \end{bmatrix} \quad (2.1)$$

$$\begin{bmatrix} v_x \\ v_y \\ w_z \end{bmatrix} = \frac{r}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ \frac{-1}{l_x+l_y} & \frac{1}{l_x+l_y} & \frac{-1}{l_x+l_y} & \frac{1}{l_x+l_y} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} \quad (2.2)$$

$$\begin{bmatrix} \frac{dv_{xabs}}{dt} \\ \frac{dv_{yabs}}{dt} \end{bmatrix} = \begin{bmatrix} \cos(\frac{dw_z}{dt}) & -\sin(\frac{dw_z}{dt}) \\ \sin(\frac{dw_z}{dt}) & \cos(\frac{dw_z}{dt}) \end{bmatrix} \begin{bmatrix} \frac{dv_x}{dt} \\ \frac{dv_y}{dt} \end{bmatrix} \quad (2.3)$$

La ecuación 2.1 se utiliza para averiguar la velocidad necesaria en cada una de las ruedas, dado un comando de velocidad relativa al robot. Esto quiere decir que  $v_x$  siempre es la velocidad del robot hacia adelante y hacia atrás, sin importar su posición con el mundo.  $v_y$  siempre es la velocidad del robot hacia los lados, y lo mismo con  $w_z$ .

La ecuación 2.2 en conjunto con la ecuación 2.3, se utilizan para saber la posición real del robot en un plano x,y cartesiano. Es decir, se puede conocer la posición del robot en  $x, y$ , con respecto al mundo.

### 2.1.2. Motores

Para el robot omnidireccional en construcción, se utilizarán motores DC. El diagrama básico de un motor DC se muestra en la figura 2.3.

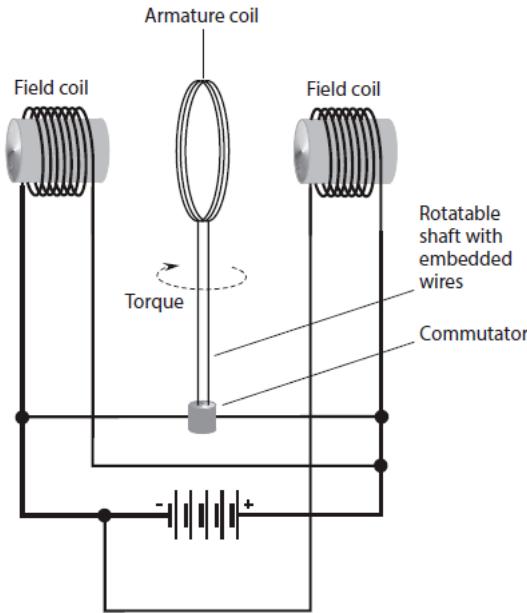


Figura 2.3: Diagrama simplificado de un motor DC. Tomado de [7]

Cómo se observa en la figura, en un nivel básico, un motor DC posee dos o más embobinados, los cuales ejercen una fuerza a un rotor el cuál posee un imán. Cuando pasa una corriente por el embobinado, la bobina produce un campo magnético, que produce una rotación en el rotor. Un comutador causa que se entregue corriente en diferentes momentos hacia los diferentes embobinados, causando que el rotor gire constantemente [7].

Cabe destacar que esta comutación se debe a una característica mecánica de la construcción del motor. Por lo tanto, lo único que se requiere para mover más rápido o más lento el motor, es más corriente o menos. Esto causará que el campo magnético producido sea mayor, y gire el rotor más velozmente.

### 2.1.3. Encoder

Un encoder es un dispositivo que se encuentra comúnmente en sistemas de control modernos, y son utilizados para convertir desplazamiento lineal o rotacional en señales codificadas o pulsos de señal. Los encoders que publican información digital son conocidos como encoders absolutos [8].

Por otra parte, los encoders incrementales, producen un pulso en cada incremento, sin embargo no hacen distinción entre los incrementos [8].

En específico, un encoder de quadratura, posee dos sensores a un cierto desfase entre ellos, generalmente de  $90^\circ$ . La figura 2.4 muestra un ejemplo de un encoder utilizando un Hall Sensor. Este sensor es magnético, y produce la señal por una inducción que provoca el imán en dos sensores al girar.

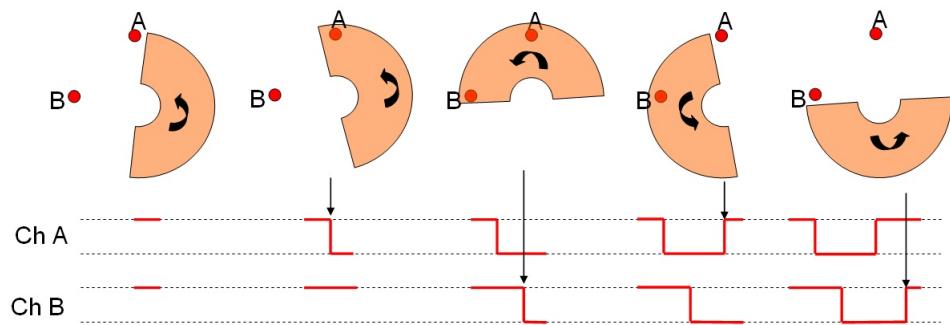


Figura 2.4: Ejemplo de un encoder utilizando un Hall Sensor, y la señal producida por un encoder de cuadratura. Tomado de [8].

La señal que se produce, ayuda a identificar la dirección de rotación del motor, ya que el desfase entre las señales proporciona esta información. La frecuencia de los clicks, indican la velocidad del encoder, y finalmente, la cantidad de clicks ayudan a identificar la posición.

## 2.2. Visión

Para que un robot omnidireccional realice navegación, existe cierta información que resulta ser útil. Los sensores utilizados, generalmente permiten medir dos propiedades del ambiente: la profundidad de lo que se ve (La más importante), y el color de lo que se está viendo. La primera es esencial para lograr la navegación.

### 2.2.1. Profundidad

Existen muchos tipos de sensores diseñados para dar información de profundidad, como los sensores de ultrasonido, sensores laser de rango, entre otros. Sin embargo, la mayoría posee ciertas limitaciones que impiden el uso de los mismos en el campo de los robots autónomos, como los problemas exactitud [3].

La llegada de los sensores de profundidad en 3D, como el Kinect o el Asus Xtion, que operan con patrones infrarrojos han superado estas limitaciones. Estas cámaras son relativamente exactas, y proveen una información densa y tridimensional directamente del hardware [3]. Por esta razón, los mismos sensores son muy utilizados en la navegación de robots autónomos.

Además de sensores como el Kinect, existen sensores de mayor precisión en un ámbito 2D, como lo son los sensores denominados en inglés *Lidar*. Un ejemplo de estos sensores son producidos por la marca Hokuyo.

A continuación se muestra en la figura 2.5 una muestra de la salida de un sensor de profundidad, utilizando una librería llamada PCL para visualizar la información.

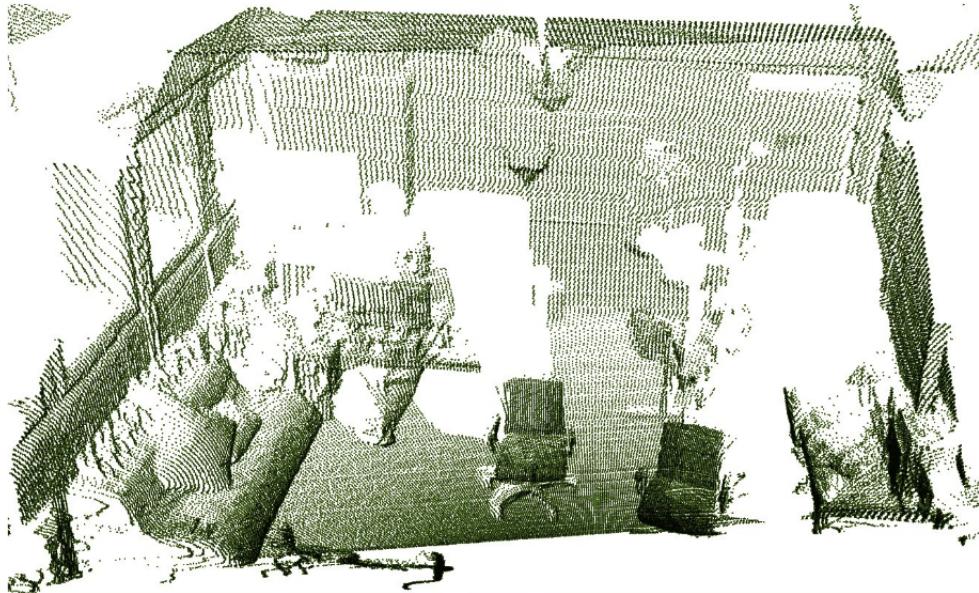


Figura 2.5: Imagen de una captura de un sensor de profundidad. Tomado de [15].

### 2.2.2. Textura

La textura en la parte de visión no es fundamental para realizar navegación. Esta información se puede utilizar para realizar detección de objetos, o encontrar cierto tipo de marcadores en el mundo para ubicarse mejor, sin embargo no es indispensable en el uso de navegación. Ejemplos de casos en donde se utilizan los sensores de color se pueden observar en [12] y [11].

## 2.3. Autonomía

La autonomía en un robot, hace referencia a la capacidad del robot de comportarse de manera “inteligente” en algún aspecto [17]. Por ejemplo un vehículo que tenga la capacidad de manejarse sólo, y no provoque choques. Otro ejemplo sería robots que puedan limpiar un desastre nuclear, y no sean los humanos los que se vean expuestos a la radiación [17].

Un robot que realice navegación autónomamente, se espera que pueda maniobrar en un espacio, ya sea nuevo o conocido, y pueda crear un mapa del lugar. La navegación por lo tanto, se compone de dos aspectos importantes. El primero, sería el mapeo y el segundo, la localización.

### 2.3.1. Mapeo

El objetivo del mapeo en un robot autónomo es aprender del entorno que rodea el robot, con el fin de realizar navegación en un futuro, o enviar este mapa a seres humanos. Es primordial tener un sistema robusto, para que los resultados del mapeo en diferentes ambientes sean consistentes.

Existen librerías que contienen la programación necesaria para realizar mapeo en un robot. Un ejemplo, es la librería denominada Navigation Stack, en el entorno de Ros. ROS (Robot Operating System), es un set de librerías, herramientas y convenciones que tratan de simplificar la tarea de crear comportamientos robóticos complejos y robustos a través de una amplia variedad de plataformas robóticas [13].

La librería de ROS, posee un módulo llamado *gmapping*, como parte del “Navigation Stack”. De acuerdo con lo mencionado en [1], es un algoritmo que utiliza *particle filter* (filtrado de partículas) de Rao-Blackwellized para realizar un mapeo a partir de sensores del tipo *laser range*.

Esta solución ha probado ser muy eficiente, para ser utilizada en resolver el problema de mapeo y localización simultánea. El algoritmo utiliza un filtro de partículas, en el cuál cada partícula contiene un mapa individual del ambiente. Correspondientemente, la pregunta clave es cómo reducir la cantidad de partículas. El algoritmo en *gmapping* propone un acercamiento al problema que utiliza una distribución precisa que toma en cuenta no sólo el movimiento del robot, pero también la observación más reciente. Esto decrementa drásticamente la incertidumbre de la posición del robot, en el paso de la predicción del filtro [9]. La figura 2.6 muestra el ejemplo de un mapa generado por este módulo de ROS en el laboratorio de investigaciones de la Universidad de Texas.

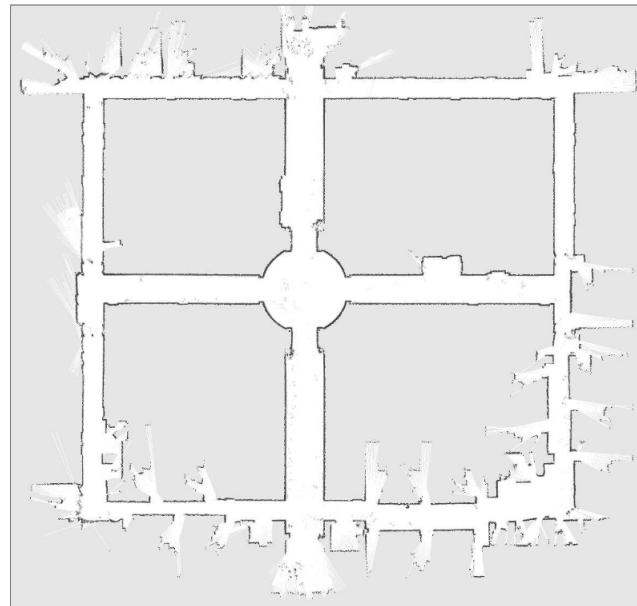


Figura 2.6: Mapa del edificio ACES en la universidad de Texas, cuarto piso. Tomado de [9].

De los mapas generados por Gmapping, las secciones grises son secciones desconocidas, o inexploradas. Las secciones en blanco son secciones bien conocidas que no posee obstáculos. Las secciones negras, son secciones que poseen obstáculos. La figura 2.7 muestra una visualización del proceso de mapeo en tiempo real, utilizando Rviz.

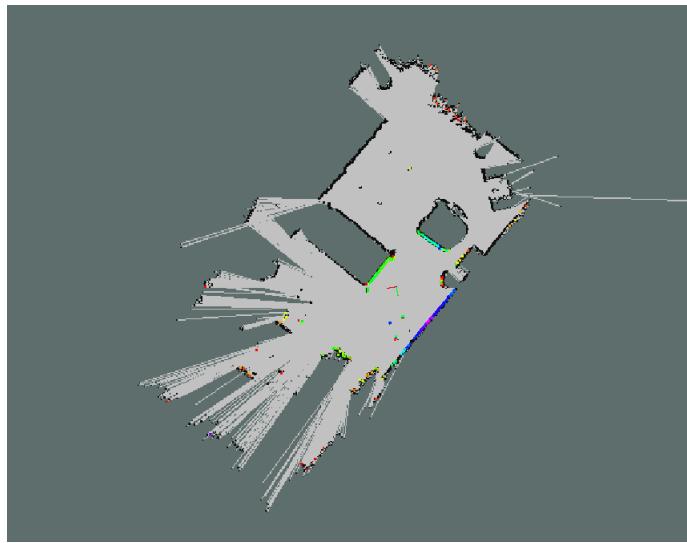


Figura 2.7: Proceso de mapeo en tiempo real. Tomado de [9].

### 2.3.2. Localización

El proceso de localización hace referencia a ubicar la posición del robot, relativa al mundo. Por lo tanto, se requiere de tener un mundo en el cuál localizarse. Este proceso se basa primariamente en la información de la odometría de las ruedas, tomado de lo que serían los encoders de los motores. La figura 2.8 muestra un diagrama de este proceso. Sin embargo, siempre existe un pequeño nivel de error en la odometría, que puede ser causado por el deslizamiento, y otros problemas similares.

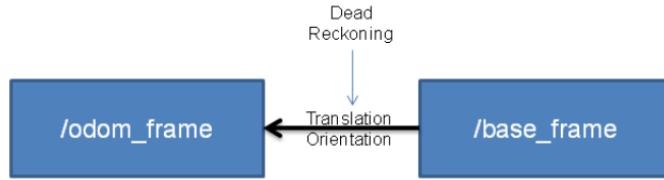


Figura 2.8: Diagrama del proceso de localización utilizando únicamente la odometría de las ruedas. Tomado de [6].

Se debe por lo tanto utilizar alguna otra fuente de información para corregir los posibles deslizamientos que puede sufrir el robot, y tener una posición real del robot en el mundo. Existen varias formas de hacer esto, sin embargo el “Navigation Stack” utiliza un módulo llamado *amcl*, para la localización.

*Amcl* es un sistema de localización probabilística para un robot en movimiento de 2-D. Implementa el modelo adaptativo de localización Monte Carlo, como fué descrito por Dieter Fox, el cual utiliza un filtro de partículas para llevar un record de la posición del robot contra un mapa conocido. La figura 2.9 muestra el proceso de localización, utilizando la información de los nodos activos en ROS.

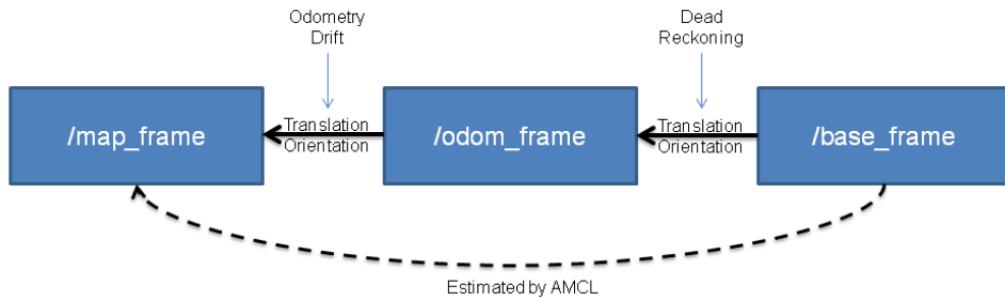


Figura 2.9: Diagrama del proceso de estimación de la posición realizado por ACML. Tomado de [6].

En otras palabras, AMCL trata de relacionar los *laser scans* con el mapa, y así detecta si ha ocurrido alguna desviación en la estimación de la pose basado en la odometría. Este “deslizamiento” es compensado publicando una transformación entre el marco del mapa, y el marco de la odometría, tal que el final la transformación del mapa hacia la base corresponda con la posición real del robot en el mundo.

### 2.3.3. Navegación

Para empezar a diseñar el aspecto de la navegación en un robot, se deben tener principalmente dos cosas. La primera que el robot se pueda mover inteligentemente, y maniobrar lo menos posible. Por lo tanto, un robot que posea tres grados de libertad es deseable, pues hace que la tarea de desplazarse en un ambiente sea bastante sencilla [2].

Además se necesita saber la distancia que ha recorrido el robot en su plano, y cuánto ha rotado. Esto para que el robot sepa que el cambio en la información del sensor de profundidad se debe a un cambio en su posición, y no a un cambio en el mundo. Esto se logra a través de los encoders que se encuentran en los motores del robot.

Finalmente, se necesita un sensor de profundidad, como se habló anteriormente. Esto le permitirá al robot descubrir su entorno, y crear un mapa del mismo.

Igual que para el caso del mapeo, la librería de Navigation Stack en ROS contienen la programación necesaria para realizar navegación, siempre y cuando el robot cumpla con las características anteriormente descritas.

El Stack de Navegación de ROS requiere dos mapas de costos, uno local y otro global, los cuales contienen la información que representa la proyección de obstáculos in un plano 2D, así como un radio de seguridad, donde se garantiza que el robot no colisionará con ningún otro objeto, sin importar su orientación [14].

La navegación se realiza utilizando un módulo llamado *global\_planner*, el cual utiliza un mapa actual, y el radio del robot para planear una ruta del punto A al punto B.

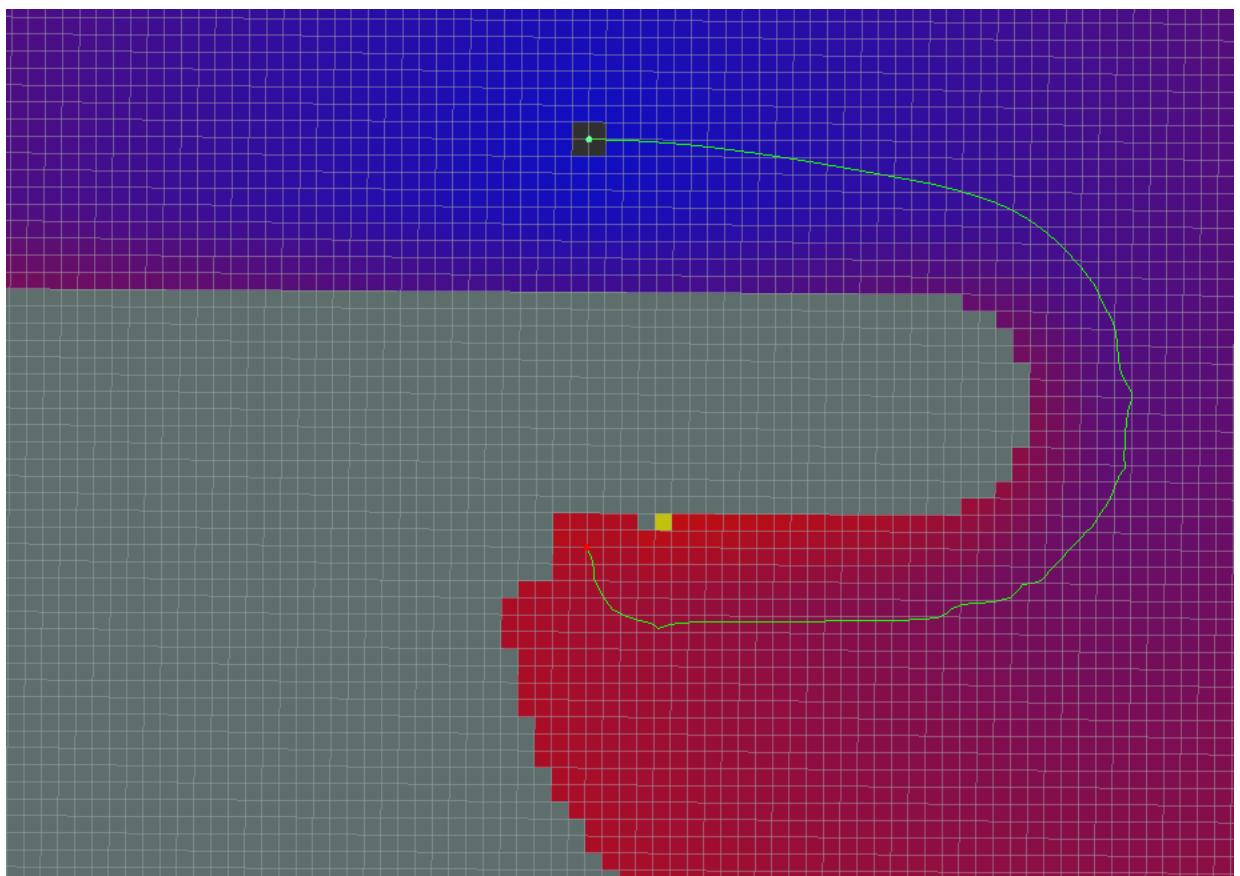


Figura 2.10: Ejemplo de un uso del global planner en ROS, para obtener una posible ruta. Tomado de [4].

Este módulo utiliza diferentes algoritmos, que pueden ser activados o desactivados para realizar la planeación de las rutas, como el algoritmo Dijkstra's.



## CAPÍTULO 3

# INFRAESTRUCTURA EXISTENTE

Conocer la infraestructura disponible es fundamental cuando se debe realizar un diseño, puesto que el diseño debe cumplir con las especificaciones y debe ser compatible con el Hardware. Este proyecto no pretende diseñar piezas mecánicas de construcción del robot, ni confeccionarlas, por lo que el diseño no incluirá esa sección.

En este capítulo se mencionará algunos de los componentes existentes del robot. Por ejemplo, información de los motores, encoders que poseen, la relaciones de engranajes. Se pretende realizar un recuento de los componentes que se utilizarán durante este proyecto, pero que no fueron diseñados propiamente por el proyecto. Esto servirá de guía en los próximos capítulos para el diseño, y la implementación, puesto que los mismos van dirigidos a incorporar todos estos dispositivos en un sólo robot.

En la primera sección, titulada **Carrito omnidireccional** se discutirá un poco del diagrama de comunicación básico del proyecto, el cuál justifica el uso de los controladores Roboclaw, y el microcontrolador Stm32f411.

En la sección **Motores**, se hablará de las especificaciones eléctricas y mecánicas de los motores a utilizar, las cuáles serán utilizadas posteriormente en el diseño.

La sección **Controladores** menciona los controladores de los motores a utilizar, y las funciones relevantes de los mismos que permitirán la conexión con el Stm32f411.

La sección **Microcontrolador** explica algunas de las funciones del STM32F411 utilizado en este proyecto, y las funciones para leer encoders de cuadratura.

**Hokuyo** habla de ciertas especificaciones importantes del escaner láser, y finalmente **Control de PS4** menciona el joystick utilizado posteriormente para mover la base del robot.

### 3.1. Carrito omnidireccional

El ARCOS-Lab tiene varios prototipos de carritos omnidireccionales utilizando ruedas mecanum, los cuales fueron desarrollados por varios miembros del laboratorio y liderados por Nicole Orosco. Todos los diseños mecánicos de la plataforma, como los soportes de las ruedas, soportes para la batería, para el Kinect, STM32F411 y Raspberry Pi fueron diseñados como parte de otros proyectos de investigación.

Al momento de iniciar el proyecto, todas las piezas se encontraban impresas y cortadas, además de ensambladas, por lo que sólo fué necesario agregar los componentes que se mostrarán a continuación en la figura 3.1.

Hay que recordar que se requiere de un sistema que pueda controlar los motores, y además llevar control de la odometría de las ruedas. Además se deberá incorporar el sensor de profundidad en el carrito omnidireccional.

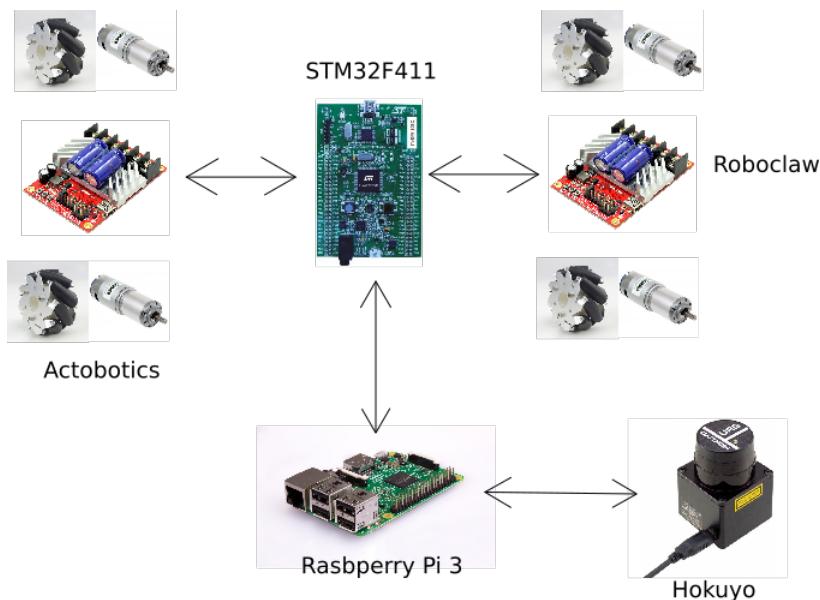


Figura 3.1: Diagrama básico de la implementación de la parte eléctrica del carrito omnidireccional.  
Autoría propia.

### 3.2. Motores

Empezando por los **motores**, se utilizarán motores DC Actobotics 638276. La hoja de fabricante con los datos en la sección de los apéndices. Además, en la tabla 3.1 se muestra la información más relevante de los motores a utilizar, como por ejemplo el consumo máximo, la tensión de trabajo, y la relación de los engranes.

Cuadro 3.1: Información más relevante de los motores Actobotics, a ser utilizados. Autoría propia.

Especificación	Valor
Voltage de operación	6-12 VDC
Corriente máxima de operación	0.53 A
Velocidad máxima sin carga	118 ± 12 rpm
Corriente máxima detenido	20 A
Razón de los engranes	1/71
Clicks del encoder por revolución	3408

Se utilizarán cuatro motores, uno para cada rueda, conectados a ruedas mecanum genericas, de 6mm de diámetro. Estas son ruedas omnidireccionales con rodillos a un ángulo de  $\alpha = 45^\circ$ .

### 3.3. Controladores

En lo que concierne a los **controladores** de los motores, no se diseñarán, puesto que se utilizarán controladores comerciales de marca IonMotion Roboclaw. En específico, se utilizará el modelo Roboclaw 2x15A. En el cuadro 3.2 se resumen los datos más importantes de este controlador.

Cuadro 3.2: Información más relevante de los controladores Roboclaw 2x15A. Autoría propia.

Especificación	Valor
Tensión de la batería principal	6-34 VDC
Tensión de la batería lógica	6-34 VDC
Bits de los contadores para encoder	32 bits
Velocidad máxima sin carga	118 +- 12 rpm
R232 Baud Rate	460,800 Bits/s
Tensión de I/O	3.3 VDC

Es importante mencionar, que estos controladores se utilizan para manejar los motores. Pueden manejar hasta dos motores, y son muy versátiles en cuánto a cómo se pueden utilizar. En particular, para este proyecto se usará la comunicación serial que posee el controlador. En la figura 3.2 se muestran los pines que posee el roboclaw. En específico, los pines S1, y S2 se pueden utilizar como puertos USART para comunicación serial. La tabla 3.3 muestra las funcionalidades que pueden cumplir cada pin.

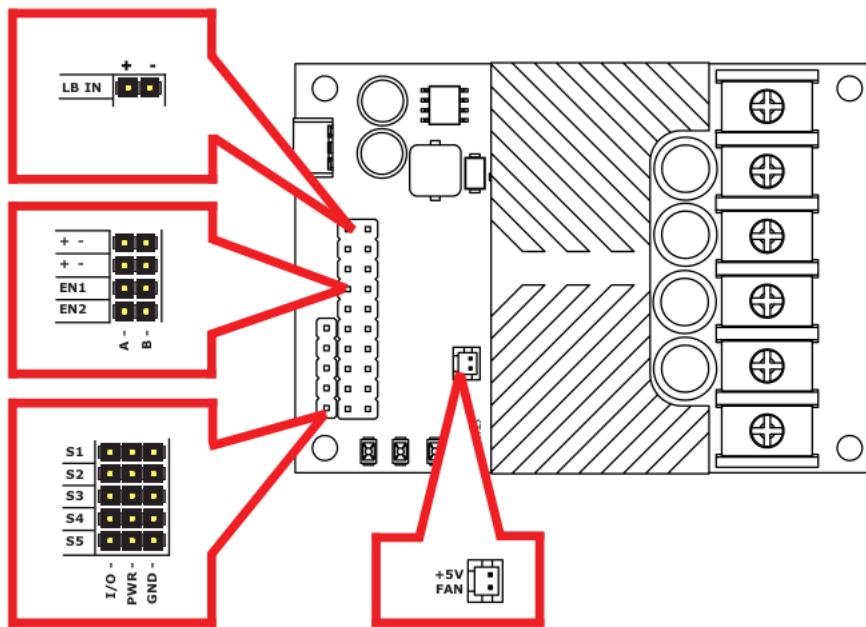


Figura 3.2: Muestra de los pines de un Roboclaw 2x15A fabricado por IonMotion. Tomado del manual del fabricante.

Cuadro 3.3: Tabla con las funciones de los pines en un Roboclaw 2x15A fabricado por IonMotion. Tomado del manual del fabricante.

NAME	UART TTL	ANALOG	R/C PULSE	FLIP SWITCH	E-STOP	HOME	LIMIT	V-CLAMP	Encoder
S1	RX	Motor 1	Motor 1						
S2	TX	Motor 2	Motor 2						
S3				X	X			X	
S4					X	Motor 1	Motor 1	X	
S5					X	Motor 2	Motor 2	X	
EN1									Motor 1
EN2									Motor 2
+5V									
FAN									

Se utilizará una frecuencia de 115200 en la transmisión serial, ya que es la más alta soportada por .

### 3.4. Sistema de desarrollo

Siguiendo el esquema de la figura 3.1, los controladores Roboclaw van conectados al **sistema de desarrollo** STM32F411DISCOVERY, el cuál es manejado por un **microcontrolador** STM32F411. La principal función de este microcontrolador será llevar cuenta de la odometría de cada una de las ruedas, y del robot como tal. Además, este aceptará instrucciones por USB del Raspberry Pi, de movimiento en el formato  $(V_X, V_Y, W_Z)$  y enviará información de vuelta de la odometría del robot en el formato  $(X_X, X_Y, \alpha_Z)$ . También enviará instrucciones de movimiento a los dos controladores Roboclaw, y controlará la velocidad de cada motor por medio de un PID, en lazo cerrado con la información del encoder como entrada.

El esquema mostrado en la figura 3.3 muestra en resumen lo que se explicó anteriormente.

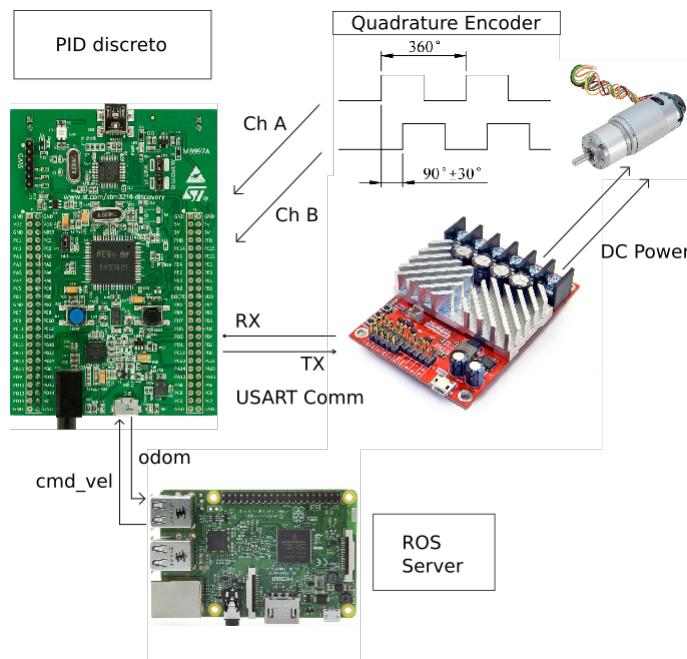


Figura 3.3: Diagrama de las funciones que debe realizar el microcontrolador STM32F411. Autoría propia.

Es importante notar, que la programación del microcontrolador se realizará mediante las librerías libopencm3 y libopencm3-plus, las cuales permiten escribir el código en el lenguaje de programación C. Además, el microcontrolador posee las funciones para leer los encoders directamente, puesto que posee timers que se pueden configurar en un modo para leer codificadores de cuadratura.

Información sobre el uso de encoders de cuadratura en el stm se puede encontrar en apéndice correspondiente.

### 3.5. Hokuyo

El Hokuyo URG-04LX-UG01 RangeFinder es un sensor de alta calidad y precisión diseñado para realizar escaneos dos dimensionales con un sensor láser. Algunas de las especificaciones del sensor se encuentran en la tabla 3.4. Adicionalmente, la figura 3.4 muestra la fotografía de un sensor Hokuyo.

Cuadro 3.4: Especificaciones del sensor Hokuyo. Autoría propia.

Especificación	Valor
Distancia de escaneo	20mm a 5600mm
Región de escaneo	240°
Duración de escaneo	100ms/scan
Resolución angular	0,36°
Peso aproximado	160g



Figura 3.4: Imagen de un ejemplo del sensor Hokuyo, tomado de la página de ventas robots-hop.com.

### 3.6. Control de PS4

Se utilizará un control de Play Station 4 para mandar comandos de movimiento hacia la base, y realizar pruebas. Los controles de PS4, también conocidos como DualShock4. En la figura 3.5 se muestra una figura del control.



Figura 3.5: Imagen de un control de PS4 dualshock. Tomado de la página de ventas BestBuy.com



## CAPÍTULO 4

# DISEÑO

El presente capítulo pretende informar al lector de los respectivos diseños tanto en Hardware como en Software necesarios para la implementación de navegación y localización simultánea. Es indispensable tener en cuenta los requerimientos del diseño, lo por que en la primera sección titulada **Justificación de diseño** se mencionan algunas de las características que debe cumplir cada etapa.

A partir de esto, se inicia con el proceso de diseño, donde el la sección **Diagrama eléctrico** se habla de las conexiones eléctricas necesarias entre los diferentes módulos, así como de las características de cada módulo por las cuáles las conexiones se realizan de esa manera. Además se realiza un análisis rápido del grosor de los cables, debido al consumo de corriente máximo de los motores.

Posteriormente se habla del diseño del código necesario en el STM32F411 en la sección **Código del stm32f411**, el cuál debe cumplir con las características mencionadas en la sección de justificación, como poseer una biblioteca de comunicación, realizar la lectura de los timers relacionados a cada encoder, realizar un controlador PID para la velocidad de los motores, entre otros.

Finalmente, la sección **Código de la computadora principal** menciona el diseño de lo necesario del lado de la computadora principal, de acuerdo con la justificación, para publicar la información de la odometría y sensores del robot en el ambiente de ROS.

- La primera, será el diseño eléctrico del carrito omnidireccional, con los componentes necesarios para funcionar.
- La segunda es el diseño del código para el carrito omnidireccional, encargado del control de los motores y la odometría de las ruedas.
- Por último, se hablará del diseño necesario del lado de la computadora principal, para lograr la navegación y el mapeo.

## 4.1. Justificación de diseño

Antes de iniciar con el diseño de cada etapa, se procederá a enlistar las necesidades existentes, las cuáles justificarán el diseño posterior. Primero se hablará de las necesidades en Hardware, y posteriormente de las necesidades en Software.

### 4.1.1. Hardware

En lo que respecta a Hardware, en diseño hay muy poco trabajo por hacer. Los diseños de las piezas del carrito omnidireccional, y componentes electrónicos **no** son necesarios para este proyecto, pues son etapas ya concluidas. Sin embargo, será importante realizar un esquema de conexiones eléctricas que permita interconectar los diferentes dispositivos. Por lo tanto se deberá:

1. Realizar el análisis de los pines tanto del lado de los controladores Roboclaw, como del lado del STM32F411 para analizar la compatibilidad entre ambos dispositivos.
2. Definir las conexiones a realizar entre los dispositivos.
3. Definir las conexiones necesarias para conectar los sensores magnéticos de los encoders de cuadratura al microcontrolador STM32F411.
4. Definir en un diagrama la nomenclatura para los motores del diseño físico.
5. Sintetizarlo todo en un sólo diagrama.

### 4.1.2. Software

Para el diseño del código, se debe considerar el hardware que se tiene y las funciones que este permite. A continuación se listarán las funciones que debe cumplir el código por escribir:

1. Biblioteca de comunicación hacia el Roboclaw. Comandos básicos de prueba, como leer firmware, leer información de la batería, y enviar comandos de velocidad.
2. Escribir una función que sea llamada por medio de interrupciones de hardware, cada cierto tiempo constante, para llevar control de la velocidad de cada encoder, y la posición del encoder.
3. Diseñar una función que controle la velocidad de cada rueda, por medio de un control PID discreto.
4. Diseñar un protocolo de comunicación que permita recibir comandos de velocidad del Raspberry Pi, y enviar información de la odometría del robot al mismo.
5. Escribir la función que realice el cálculo de la odometría global del robot en cada interrupción por hardware.

En la computadora principal se correrá el servidor de ROS que interconectará el robot omnidireccional con el stack de navegación. Hay varios requisitos que se deben cumplir para que el stack de navegación funcione correctamente. La figura 4.1 muestra un diagrama de la interacción entre Navigation Stack y el robot deseado. Los cuadros que se encuentran de color azulado, son específicas de la plataforma.

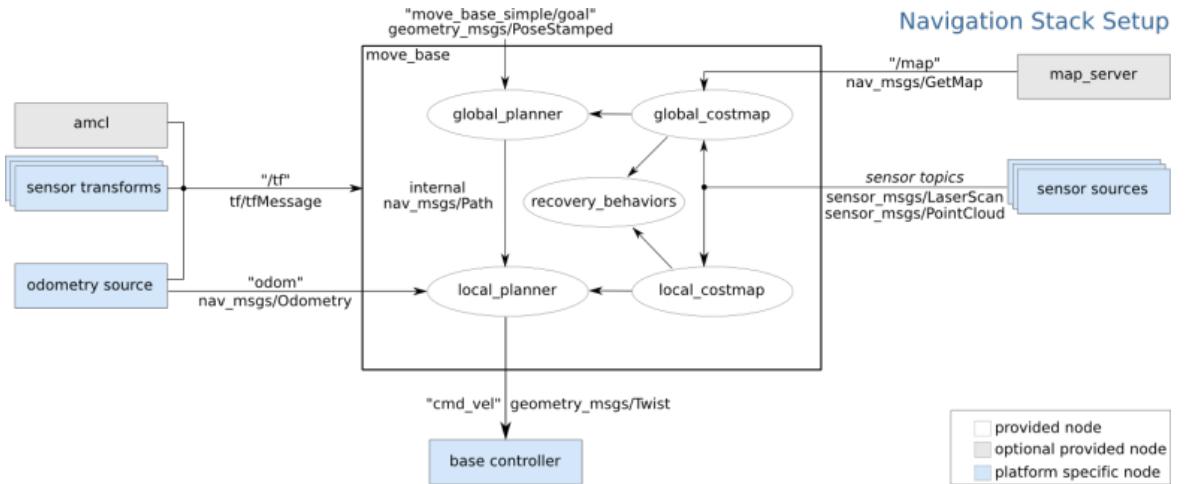


Figura 4.1: Revisión de configuración para el stack de navegación. Tomado de ROS.

Es necesario por lo tanto, crear un nodo en ros que posea las siguientes características:

1. Publicación de la información de los sensores láser en el tópico LaserScan para sensores de profundidad 2D, o PointCloud para sensores en 3D.
2. Publicación de la odometría actual del robot en el tópico llamado “odom”. Esta es la información de la posición de la base con respecto al mundo.
3. Publicación del árbol de transformación en el tópico “tf”.
4. Subscripción al nodo “cmd\_vel”, para recibir instrucciones de movimiento de la base.

## 4.2. Diagrama eléctrico

De acuerdo a lo mencionado en la justificación del diseño, se debe tener un sistema que interconecte el Roboclaw con el STM32F411 para enviarle comandos desde el último. Como se mencionó en el capítulo de infraestructura existente, tanto el Roboclaw como el Stm32f4 poseen soporte para pines de comunicación serial USART. En específico, el manual de usuario del Roboclaw menciona que los pines S1 y S2 cumplen la función de RX y TX respectivamente.

Del lado del STM32F411, el mismo posee varios canales de comunicación USART, sin embargo los únicos dos canales disponibles se encuentran en el puerto de USART 1, con los pines PB6 y

PB7 para TX y RX respectivamente, y el puerto de USART 2, con los pines de PA2 y PA3 para TX y RX respectivamente.

Cada Roboclaw puede soportar 2 motores. Por lo tanto, para soportar cuatro motores se necesitarán dos Roboclaws. Cada uno será controlado en cada uno de los puertos USART mencionados anteriormente.

Es importante definir el grosor de los cables de alimentación para los motores, y de la batería hacia el controlador Roboclaw. Según la tabla de diseño para grosor de cable AWG, para un amperaje máximo de 20A a 60°, se necesitaría un cable de AWG 12 para la alimentación de los motores. Sin embargo, se utilizará cable de grosor mayor puesto que es el que se tiene en el laboratorio.

Para los cables de comunicación, el amperaje no subirá de 0.18A, por lo que un cable grosor 34 AWG o mayor será más que suficiente.

En el diagrama 4.2 se puede apreciar las conexiones entre los puertos USART del Stm32f4, y los respectivos controladores Roboclaw.

Para la conexión de los sensores magnéticos, se necesitan timers que puedan leer encoders de cuadratura. La manera en la que funciona un timer en esta función se explica en los apéndices. Los timers 2,3,4,5 del STM32F4 soportan la opción de leer encoders de cuadratura, por lo tanto serán utilizados para esta tarea.

Finalmente, la nomenclatura de cada motores, se encuentra en el mismo diagrama 4.2, en la esquinas superior izquierda.

### 4.3. Código del stm32f411

#### 4.3.1. Biblioteca de comunicación

La biblioteca de comunicación que comunica al microcontrolador STM32F411 con los controladores de motores Roboclaw se desarrollará basándose en la biblioteca ya existente de Roboclaw. La biblioteca contiene una serie de instrucciones que le indican a los microcontroladores realizar ciertas acciones, como por ejemplo mover los motores en una dirección u otra, y además indicarles la velocidad. La biblioteca se encuentra escrita en Python, y de acá se tomarán los ejemplos necesarios para realizar una biblioteca en C con sólo las instrucciones necesarias. La biblioteca original contiene más de 100 instrucciones, sin embargo no se utilizarán todas las instrucciones de momento.

Es importante destacar que el microcontrolador posee soporte para puertos USART. Inclusive, la biblioteca libopencm3 posee un paquete para transmisión a través de estos puertos. Sólo se utilizan dos comandos importantes, los cuales son *uart\_send\_blocking* y *uart\_recv\_blocking*.

Se implementarán un total de 6 instrucciones para proporcionar el funcionamiento básico del robot. En el apéndice se puede encontrar más información acerca de la estructura de las instrucciones. En particular nos interesan las instrucciones 0, 1, 4, 5, 21 y 24. A continuación se muestra la estructura de las instrucciones a implementar:

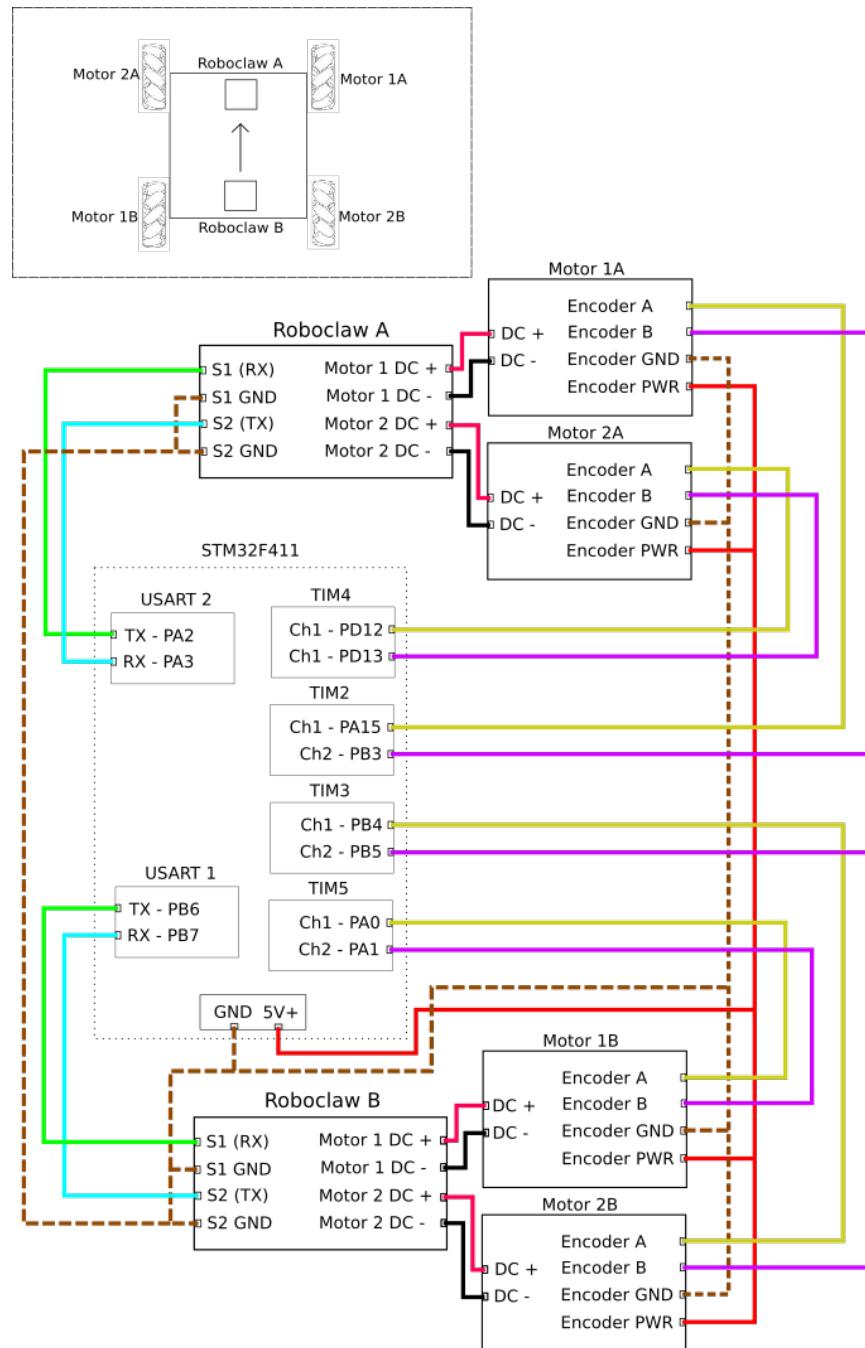


Figura 4.2: Diagrama de las conexiones eléctricas entre los dispositivos. Autoría propia.

- **Instrucción 0:** Drive Forward M1. Se utiliza para mover el motor 1 hacia adelante. Tiene valores válidos desde el rango de 0 a 127, donde 127 es velocidad máxima, y 0 completamente detenido. Se manda de la siguiente manera:

Dirección, 0, Valor, CRC(2 bytes)

- **Instrucción 1:** Drive Backwards M1. Se utiliza para mover el motor 1 hacia atrás. Tiene valores válidos desde el rango de 0 a 127, donde 127 es velocidad máxima, y 0 completamente detenido. Se manda de la siguiente manera:

Dirección, 1, Valor, CRC(2 bytes)

Recibe lo siguiente:

0xFF

- **Instrucción 4:** Drive Forward M2. Se utiliza para mover el motor 2 hacia adelante. Tiene valores válidos desde el rango de 0 a 127, donde 127 es velocidad máxima, y 0 completamente detenido. Se manda de la siguiente manera:

Dirección, 4, Valor, CRC(2 bytes)

Recibe lo siguiente:

0xFF

- **Instrucción 5:** Drive Backwards M2. Se utiliza para mover el motor 2 hacia atrás. Tiene valores válidos desde el rango de 0 a 127, donde 127 es velocidad máxima, y 0 completamente detenido. Se manda de la siguiente manera:

Dirección, 5, Valor, CRC(2 bytes)

Recibe lo siguiente:

0xFF

- **Instrucción 21:** Read Firmware Version. Se utiliza para leer la información de la versión de software que actualmente se encuentra corriente en el controlador. Lo que se envía es bastante sencillo:

Dirección, 21

Recibe algo similar a lo siguiente:

“RoboClaw 10.2A v4.1.11”,10,0, CRC(2 bytes)

- **Instrucción 24:** Read Logic Battery Voltage Level. Se utiliza para medir la tensión entre las terminales B+ y B-. La información se encuentra en decenas de voltage, ejemplo: 300 = 30V. A continuación se muestra lo que se debe enviar:

Dirección, 24

Recibe lo siguiente

Valor (2 bytes), CRC (2 bytes)

Adicionalmente, en cada instrucción se manda y recibe un byte de información a la vez, y los bits de inicio y fin son configurables, al igual que la velocidad de transferencia, la cuál debe coincidir con la velocidad del controlador Roboclaw. Además de estos, existen otros parámetros que se deben configurar para el funcionamiento correcto del puerto de comunicación. La figura 4.3 muestra un ejemplo de cómo configurar un puerto usart utilizando la biblioteca libopencm3.

```

/* Setup usart_port_x->usart parameters. */
usart_set_baudrate(usart_port_x->usart, usart_port_x->baudrate);
usart_set_databits(usart_port_x->usart, 8);
usart_set_stopbits(usart_port_x->usart, USART_STOPBITS_1);
usart_set_mode(usart_port_x->usart, USART_MODE_TX_RX);
usart_set_parity(usart_port_x->usart, USART_PARITY_NONE);
usart_set_flow_control(usart_port_x->usart, USART_FLOWCONTROL_NONE);

/* Finally enable the USART. */
usart_enable(usart_port_x->usart);

```

Figura 4.3: Ejemplo de configuración de un puerto USART utilizando la biblioteca libopencm3. Autoría propia.

### 4.3.2. Código del Encoder

Para llevar el conteo de la cantidad de ticks que ha ocasionado un encoder en un timer del stm32f411, se debe leer constantemente el estado de cada timer, y transferir la información a una variable donde se lleve la cuenta de los ticks. Esta lectura debe de ser en intervalos de tiempo constante, para a su vez poder calcular la velocidad del motor.

El microcontrolador stm32f411 posee una función llamada systick, el cuál es un contador que produce una bandera cuando llega al valor de recarga, el cuál puede ser configurado. Esta bandera es utilizada para llamar una subrutina, que ha de ser ejecutada en el momento en que se produce la bandera.

Por lo tanto, se obtienen interrupciones en intervalos contantes, que son producidas cada cierta cantidad de ciclos de reloj. La figura 4.4 muestra un ejemplo de cómo inicializar el contador. La variable llamada SYS TICK\_AUTORELOAD tiene unidades en ciclos.

```

systick_set_reload(SYS_TICK_AUTORELOAD); // clo
systick_set_clocksource(STK_CSR_CLKSOURCE_AHB);
systick_counter_enable();
systick_interrupt_enable();

```

Figura 4.4: Ejemplo de inicialización del systick utilizando la biblioteca libopencm3. Autoría propia.

En la interrupción, se debería de actualizar una variable que lleve control de los ticks que han sucedido desde el inicio del programa. Además, se deberá revisar una bandera que se activa cuando el timer respectivo al encoder llega a 0.

La bandera relacionada a este evento se llama TIM\_SR\_UIF en el contexto de la biblioteca libopencm3. En particular, de la mencionada biblioteca se deberán utilizar tres comandos importantes, los cuáles son:

- **timer\_get\_counter(TIMER\_PERIPHERAL):** Esta función retorna el valor del contador, el cuál es un entero sin signo de máximo 32 bits.
- **timer\_get\_flag(TIMER\_PERIPHERAL, FLAG):** Esta función se utiliza para obtener el valor de una bandera específica del timer. Retorna un valor booleano.
- **timer\_clear\_flag(TIMER\_PERIPHERAL, FLAG):** Limpia el valor de la bandera, por lo tanto la pone en Falso. No retorna nada.

#### 4.3.3. Control PID

Al igual que en el caso anterior, esta función de control del pid debe ser llamada en intervalos de tiempo constante, para que la acción del controlador siempre tenga la misma rapidez. Por lo tanto es prudente incorporar esta función como parte de la subrutina systick.

La figura 4.5 muestra el PID en cuestión a programar. La función debe comprobar si existe una diferencia entre la velocidad actual y la deseada, y dado el error producido por la resta de ambas, multiplicar este valor por una constante Kp. Utilizando la suma del error además, multiplicar eso por una constante Ki. Finalmente, el cambio del error, será multiplicado por una constante Kd. La suma de los tres componentes, da como resultado la acción a mandar al controlador.

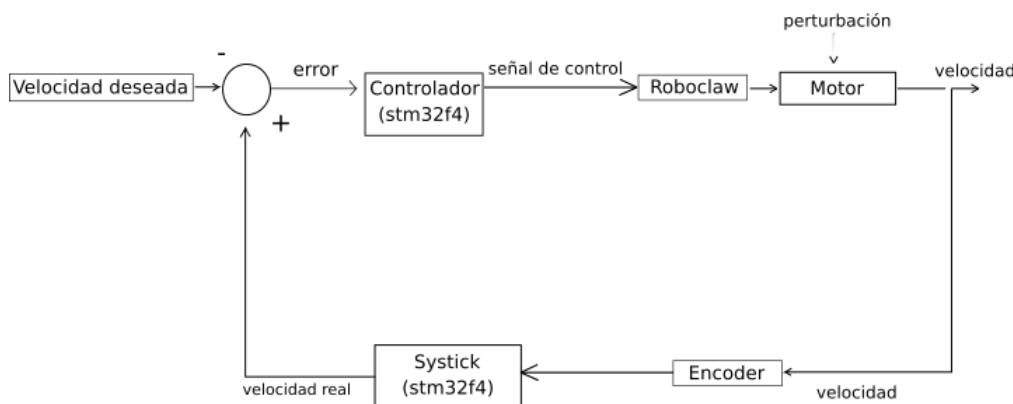


Figura 4.5: Diagrama del PID a programar utilizado para controlar la velocidad de cada motor. Autoría propia.

Los valores de Kp, Ki, y Kd deben ser fácilmente ajustables desde el ciclo principal, para su ajuste correspondiente. Algo a notar, diferente de un PID convencional, es que la acción del

controlador tiene un límite, puesto que el controlador de los motores no acepta señales análogas. Esto quiere decir que se debe manejar la condición en donde la señal de control sea mayor al número 127 (que es la máxima señal de control que acepta el Roboclaw).

#### 4.3.4. Protocolo de comunicación entre Discovery y PC

Para esta función, será necesario utilizar la biblioteca llamada libopencm3-plus, la cuál contiene funciones que permiten utilizar comandos básicos como putc y getc en el buffer del puerto USB que trae el microcontrolador STM32F411. La conexión se realiza entonces entre el puerto de comunicación USB del microcontrolador, y un puerto USB de la PC.

El protocolo de comunicación deberá ser binario, para mayor eficiencia en la transmisión. Al igual que en el caso del protocolo de comunicación hacia el roboclaw, se utilizará una verificación crc16 para todas las instrucciones. Se diseñarán varias instrucciones con el fin de enviar comandos de velocidad desde la pc principal, y además poder recibir información de la odometría actual del robot.

A continuación se listan las instrucciones por implementar, y la estructura de cada una:

- **Move Robot:** Esta instrucción recibe comandos de velocidad en  $x$ ,  $y$ , y rotación en  $z$ . Se recibe lo siguiente:

$m$  (1 byte ascii),  $V_X$  (float 32 bits),  $V_Y$  (float 32 bits),  $\omega_Z$  (float 32 bits), 10, 0, CRC (2 bytes)

Se envía de vuelta el checksum calculado localmente:

CRC (2 bytes)

- **Read Odometry:** Esta instrucción envía de vuelta la información de la odometría actual del robot. Se recibe lo siguiente:

$o$  (1 byte ascii), 10, 0, CRC (2 bytes)

Se envía de vuelta lo siguiente:

$X_X$  (float 32 bits),  $X_Y$  (float 32 bits),  $\theta_Z$  (float 32 bits), CRC (2 bytes)

- **Read Velocity:** Esta instrucción envía de vuelta la velocidad actual del robot. Se recibe lo siguiente:

$v$  (1 byte ascii), 10, 0, CRC (2 bytes)

Se envía de vuelta lo siguiente:

$V_X$  (float 32 bits),  $V_Y$  (float 32 bits),  $\omega_Z$  (float 32 bits), CRC (2 bytes)

- **Reset Robot:** Esta instrucción devolverá el robot a estado estacionario, y regresará la odometría del robot a 0,0,0. Se recibe lo siguiente:

$r$  (1 byte ascii), 10, 0, CRC (2 bytes)

Se envía de vuelta lo siguiente:

CRC (2 bytes)

Las instrucciones anteriores son las únicas necesarias para lograr el correcto funcionamiento de la navegación. Sin embargo, se escribirán las siguientes instrucciones adicionales para revisar que no existan errores.

- **Read Wheel Velocities:** Envía de vuelta las velocidades de cada una de las ruedas. Se recibe lo siguiente:

e (1 byte ascii), 10, 0, CRC (2 bytes)

Se envía de vuelta lo siguiente:

$V_{1A}$  (float 32 bits),  $V_{2A}$  (float 32 bits),  $V_{1B}$  (float 32 bits),  $V_{2B}$  (float 32 bits), CRC (2 bytes)

- **Read Serial Comm failures:** Envía la cantidad de fallos de la comunicación entre el stm32f411 y el roboclaw desde el último reset robot. Se recibe lo siguiente:

t (1 byte ascii), 10, 0, CRC (2 bytes)

Se envía de vuelta lo siguiente:

Fallos (float 32 bits), CRC (2 bytes)

- **Read Wheel Position:** Envía la posición de las ruedas desde el último reset robot en cantidad de ticks. Se recibe lo siguiente:

h (1 byte ascii), 10, 0, CRC (2 bytes)

Se envía de vuelta lo siguiente:

$X_{1A}$  (float 32 bits),  $X_{2A}$  (float 32 bits),  $X_{1B}$  (float 32 bits),  $X_{2B}$  (float 32 bits), CRC (2 bytes)

#### 4.3.5. Odometría global del robot

Existen dos tipos de odometría en un robot. La odometría local, la cuál se mide con respecto al marco de referencia del robot (Se mueve el robot hacia la izquierda, el robot rota sobre su propio eje 90 grados, etc.). El segundo tipo es la odometría global, la cuál se mide con respecto al marco de referencia del mundo. Se mueve hacia el este, al oeste, rota, etc, sin importar la orientación del robot dentro de este marco de referencia. Para efectos de navegación, nos interesa el segundo tipo de odometría.

Para realizar el cálculo de la odometría global del robot, se utilizarán las ecuaciones descritas en la nota teórica. Para esto, se utiliza la velocidad angular de cada una de las ruedas en un instante dado, se calcula el cambio en la posición de cada rueda en un instante dado, y con esto obtenemos el cambio en la posición del robot en ese instante. Finalmente, estos cambios se suman constantemente en variables globales, para mantener la posición global del robot. A continuación, el diagrama mostrado en 4.6 muestra el diagrama de flujo de lo anteriormente mencionado.

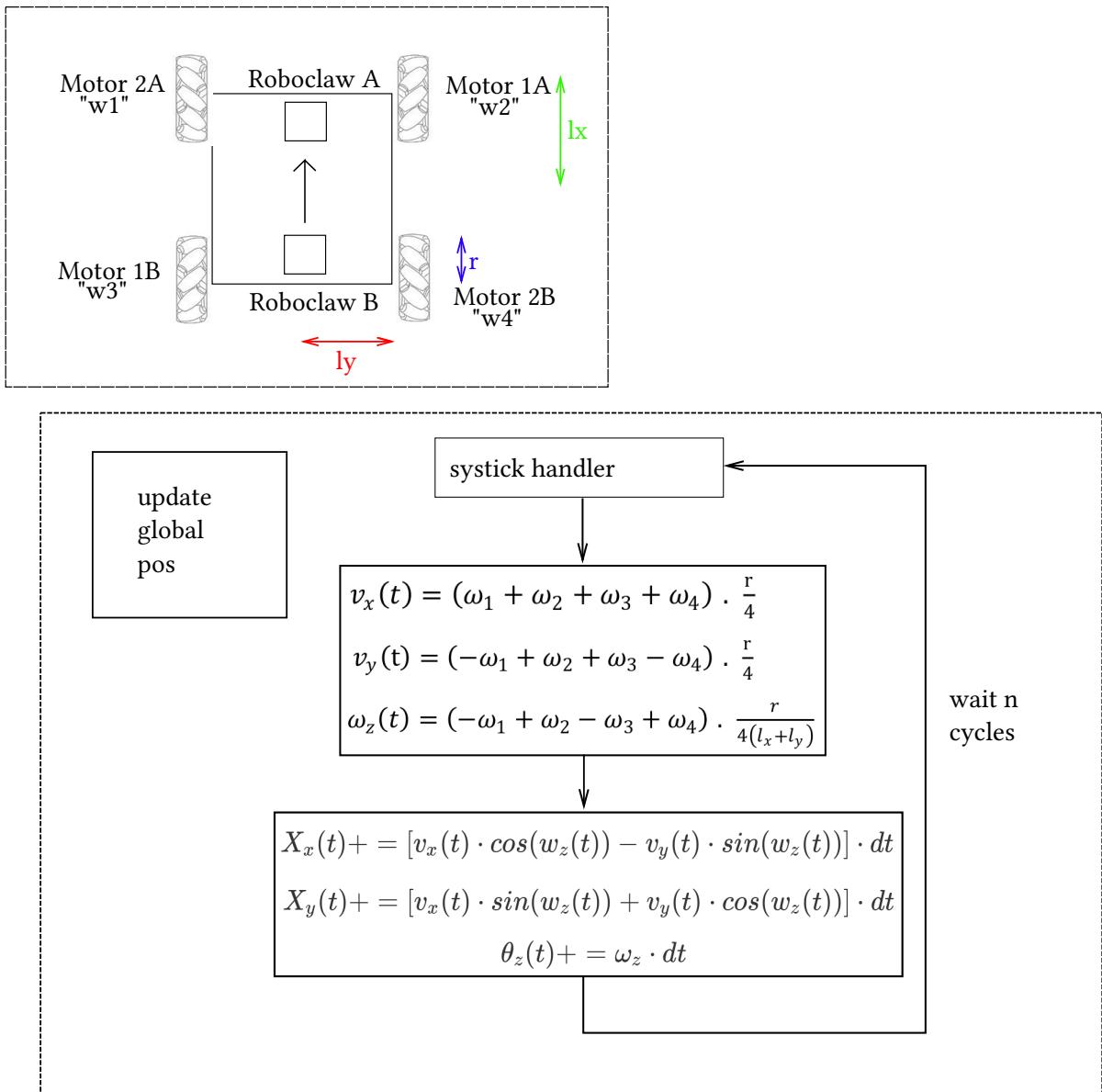


Figura 4.6: Diagrama de la actualización de la posición global. Autoría propia.

Es importante mencionar que para la escritura del código, se utilizará la plataforma GitHub. El código mencionado anteriormente se puede encontrar en el repositorio **arcoslab/stm32-roboclaw**.

#### 4.3.6. Diagrama completo del código

A continuación en la figura 4.7 se muestra un diagrama completo que muestra un poco de pseudo-código.

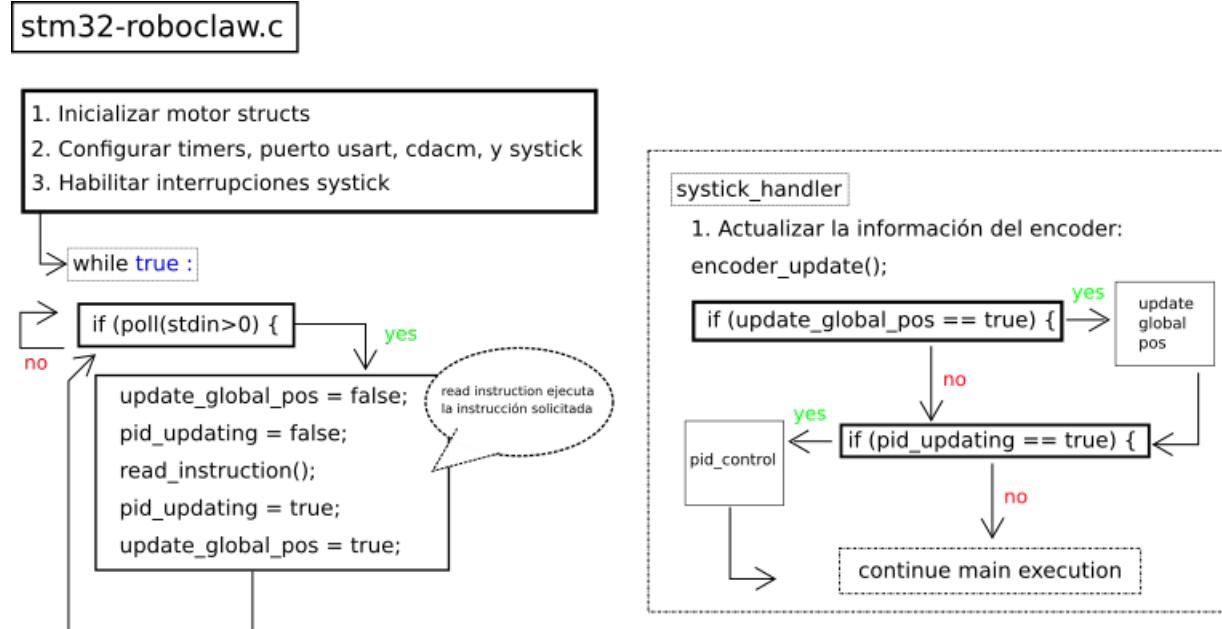


Figura 4.7: Esquema de lo que contiene el archivo principal del programa a correr en el microcontrolador. Autoría propia.

## 4.4. Código de la computadora principal

Este es el código que debe correr en el Raspberry PI. Para efectos prácticos, se iniciará el desarrollo en una laptop, esto pues la Raspberry PI podría tener una limitante computacional que se sale de los objetivos del proyecto.

### 4.4.1. Sensores láser

Se tienen dos tipos de sensores que se utilizarán a lo largo del desarrollo de este proyecto, los cuáles son un Kinect V2 y un sensor Hokuyo. Ambos tienen soporte en ROS, por lo que en esta sección, el trabajo a realizar es instalar el software que ya existe.

- Para el caso del sensor Kinect V2 se utilizó el software que se encuentra en el repositorio [Iai\\_Kinect2](#). En este repositorio se encuentran una serie de herramientas para calibrar, realizar el registro de la información de profundidad, y utilizar libfreenect2 por debajo. Esta biblioteca utiliza aceleración gráfica con OpenCL, por lo que es importante tener los mejores drivers de la tarjeta de video disponible.
- Para el uso del sensor Hokuyo, también existe un paquete llamado [Hokuyo\\_node](#) que puede ser utilizado para la lectura del sensor directamente al tópico LaserScan.

La salida de ambos sensores puede ser visualizada en [Rviz](#), un visualizador del ambiente de ROS.

### 4.4.2. Odometría

La información de la odometría del robot se puede leer directamente del mismo, sin embargo la biblioteca del lado de la computadora principal debe ser implementada primero. Los comandos son los mismos que los explicados en la sección del Protocolo de comunicación USB, sin embargo lo que se envía y recibe está inverso. Se implementará esta “driver” en Python, por su facilidad y modularidad.

La figura 4.10 explica la relación necesaria entre el nodo principal, y este driver para obtener la información de odometría.

ROS posee una biblioteca llamada **rospy** la cuál es útil para realizar nodos de ros que puedan publicar mensajes. En particular, los mensajes de odometría se pueden publicar utilizando un tipo de mensaje llamado **Odometry**, el cuál se encuentra en la biblioteca **nav\_msgs.msg**.

Los mensajes de este tipo poseen las características que se encuentran resumidas en el cuadro 4.1.

Cuadro 4.1: Contenido de un mensaje del tipo “Odometry” en el entorno ROS. Autoría propia.

Tipo de mensaje en el contenido	Nombre
std_msgs/Header	header
string	child_frame_id
geometry_msgs/PoseWithCovariance	pose
geometry_msgs/TwistWithCovariance	twist

El mensaje del tipo PoseWithCovariance contiene la información de la posición actual, mientras que el TwistWithCovariance contiene la información de la velocidad actual del robot.

#### 4.4.3. Árbol de Tf

La mayoría de paquetes en Ros requieren que un árbol de transformaciones sea publicado utilizando la biblioteca Tf. Esta biblioteca se utiliza para definir las compensaciones necesarias para cada objeto del mundo cuando se da un movimiento. Es decir, indica cuánto se mueve un objeto dada su relación estática o dinámica contra otro objeto. En este caso por ejemplo, el sensor Hokuyo tiene una relación estática contra la base del robot omnidireccional. Dado un movimiento de la base del carrito omnidireccional en el mundo, hay un movimiento equivalente del sensor en el mundo.

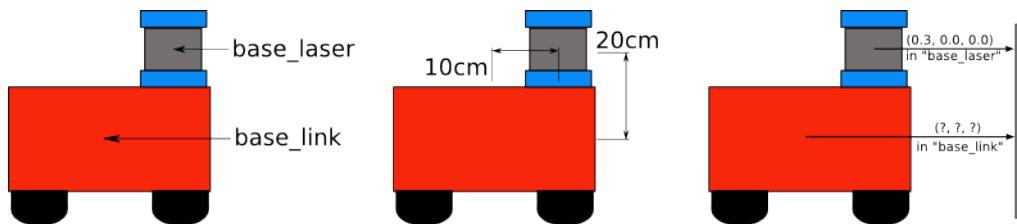


Figura 4.8: Ejemplo de la utilidad de Tf. Tomado de [5].

La figura 4.8 muestra un robot omnidireccional, que tiene un láser encima de él. Cada vez que el robot se mueve, deberíamos de ajustar la posición actual del láser, puesto que ahora el láser no está en la misma posición que antes. Este ajuste se podría hacer manualmente cada vez que se actualiza la posición del robot, sin embargo cuando se tienen muchos objetos en el mundo que dependen de la posición actual del robot, estas transformaciones se vuelven engorrosas. Por lo tanto, es más sencillo utilizar tf, y establecer la relación que existe entre la base y el sensor láser [5].

Hay varias formas de establecer esta relación. Sin embargo, la más sencilla cuando se tiene una relación estática (no cambia con el tiempo) es definir la misma en el launch file utilizado para levantar el nodo de ROS. La figura 4.9 muestra un ejemplo de cómo realizar esta transformación.

```

<launch>

<node pkg="tf" type="static_transform_publisher" name="laser_to_base"
      args="-0.1016 0 0.0889 0 0 0 base_link laser 10" />

<node name="omnidirectional" pkg="omnidirectional_node" type="omnidirectional.py" output="screen">
    <param name="port" value="/dev/ttyACM0" />
</node>

</launch>

```

Figura 4.9: Ejemplo de un launch file en ROS que habilita un árbol tf. Autoría propia.

#### 4.4.4. Comandos de velocidad de la base

Al igual que con la sección de Odometría, en este nodo es necesario un intermediario que pueda comunicarse con el stm32f411 y mande la instrucción correspondiente para comandar la velocidad. Un diagrama que explica la relación se puede encontrar en la figura 4.10.

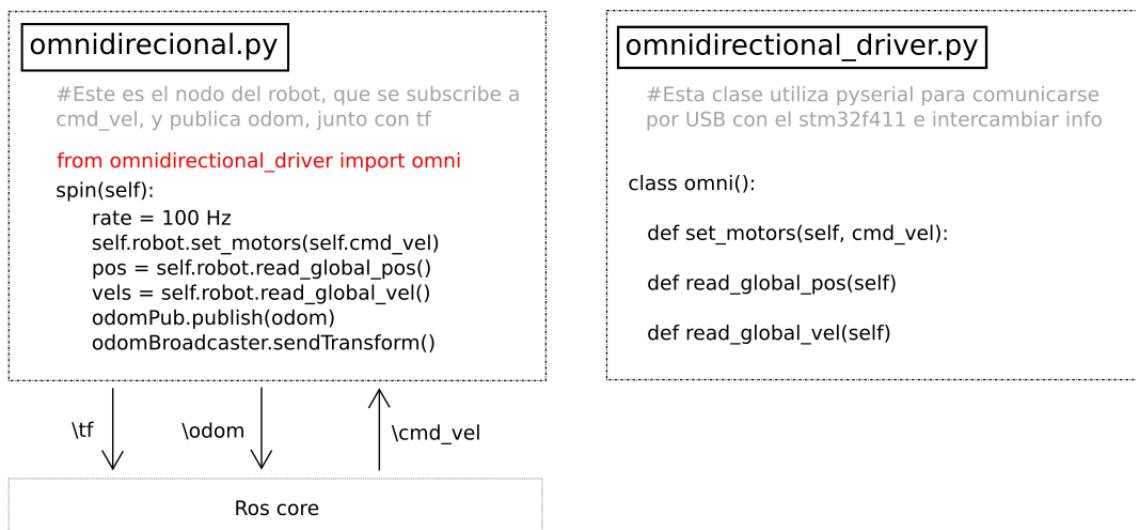


Figura 4.10: Diagrama de las interacciones entre el nodo de ROS y el driver de la conexión USB. Autoría propia.



## CAPÍTULO 5

# IMPLEMENTACIÓN

En este capítulo se revisará la implementación del diseño anteriormente descrito, con los resultados que se obtuvieron para cada etapa. El orden de la metodología descrita en el capítulo introductorio coincide con el orden en el que se implementó cada etapa, por lo que se aconseja tener presente la metodología.

La sección **Kinect** explica el proceso para realizar la instalación del software, calibración y pruebas de funcionamiento utilizando el sistema ROS en Linux. Se agrega una sección de retos, en los que se identifica los mayores retos encontrados en esta etapa.

Continuando con la sección **Hokuyo**, se proceseade a mostrar el proceso de instalación del software, y se muestran algunas capturas que de nuevo demuestran el funcionamiento correcto de este sensor en ROS.

Posteriormente se realizan las pruebas de conexión entre un **STM32F4** y un **roboclaw**, donde se realiza además el desarrollo de la librería de comunicación explicada en el capítulo de diseño.

La sección de **Encodificador de cuadratura** realiza un recuento del proceso de conexión entre el motor y el stm, además de la programación necesaria para realizar la lectura, y detalla algunos retos encontrados para esta etapa.

La sección **Controlador PID** realiza un recuento de la implementación, sintonización y muestra los resultados de un cambio en la referencia a 0.5s.

El **Ensamblado de la base** se muestra en la sección correspondiente, donde además se proporcionan algunos videos de las primeras pruebas de funcionamiento del robot.

Finalmente, la sección **Protocolo de comunicación**, establece el proceso que se llevó a cabo para desarrollar esta librería, y muestra algunos de los retos más importantes presentados durante el desarrollo de este proyecto.

El cálculo de la odometría, y la corrección de errores son secciones que llegan a culminar el proceso de diseño. Las secciones siguientes, denominadas **Implementación de gmapping** y **amcl**, realizan la implementación de los módulos necesarios para realizar la localización y navegación.

## 5.1. Kinect

Lo primero a realizar, es instalar el entorno de ROS en una computadora con el fin de realizar pruebas con los módulos de ROS. Se realizó la instalación en un Chroot, dentro de una laptop personal corriendo ARCH Linux 4.17.10-1 utilizando la guía oficial de ROS Indigo que se puede encontrar [aquí](#).

El sistema resultante es una instalación de ROS Indigo dentro de un chroot de Ubuntu Trusty. El siguiente paso fué utilizar la librería de Iai Kinect [18] para conectar un Kinect One (V2) a la computadora corriendo ROS. La figura 5.1 muestra una imagen del Kinect que se utilizó durante las pruebas.



Figura 5.1: Kinect One (V2) utilizado para realizar pruebas. Autoría propia.

### 5.1.1. Instalación del software

La instalación del software necesario se realizó siguiendo la guía del repositorio Iai Kinect 2, el cuál se puede encontrar [aquí](#). La instalación depende del repositorio libfreenect2, el cuál se encuentra [aquí](#).

### 5.1.2. Calibración del Kinect

Siguiendo las instrucciones que se encuentran en el repositorio, lo primero a realizar cuando se instala el software adecuadamente, es proceder a generar los archivos de calibración para la cámara RGB e infrarroja. Esto se logra realizando tomas a varios ángulos utilizando una imagen cuadriculada que proporciona el repositorio. Un ejemplo de esta imagen se muestra en la figura 5.2.

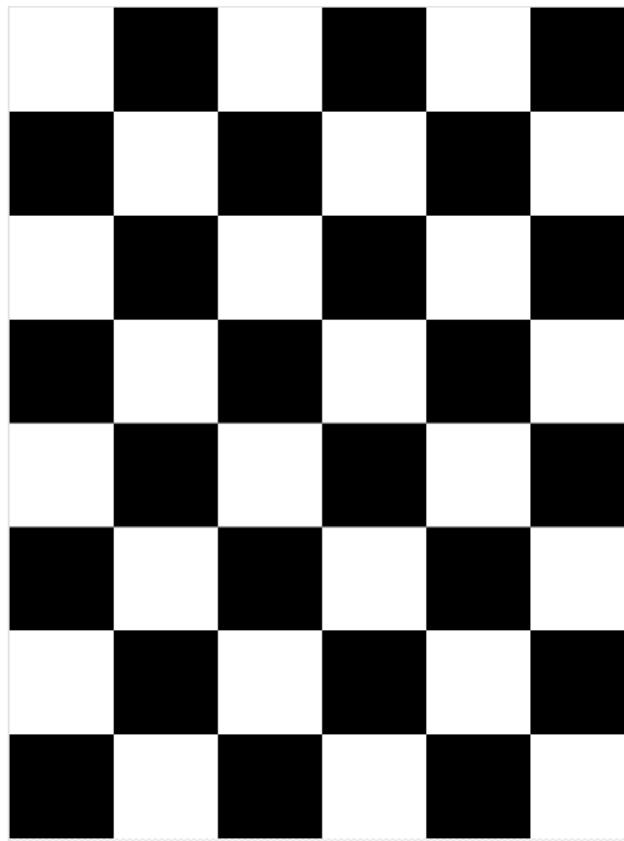


Figura 5.2: Imagen de calibración de prueba. Tomado de [18].

### 5.1.3. Pruebas de funcionamiento

A continuación, para lograr el funcionamiento correcto de la cámara, se debe correr roscore:

```
1 roscore
```

Posteriormente, con la cámara conectada a la computadora, se procede a correr el nodo de ROS proporcionado por la librería Iai\_Kinect.

```
1 roslaunch kinect2_bridge kinect2_bridge.launch
```

El resultado de este comando, se puede observar en la figura 5.3.

```
[Info] [DepthPacketStreamParser] 30 packets were lost
[Info] [DepthPacketStreamParser] 7 packets were lost
[ INFO] [1533689319.328948275]: [Kinect2Bridge::main] depth processing: ~65.6814ms (~15.225
Hz) publishing rate: ~11.9621Hz
[ INFO] [1533689319.329285177]: [Kinect2Bridge::main] color processing: ~14.0247ms (~71.305
Hz) publishing rate: ~5.64878Hz
[Info] [DepthPacketStreamParser] 31 packets were lost
[Info] [DepthPacketStreamParser] 30 packets were lost
[Info] [DepthPacketStreamParser] 9 packets were lost
[ INFO] [1533689322.336125451]: [Kinect2Bridge::main] depth processing: ~65.6853ms (~15.224
Hz) publishing rate: ~12.2933Hz
[ INFO] [1533689322.336293586]: [Kinect2Bridge::main] color processing: ~13.1232ms (~76.200
Hz) publishing rate: ~5.64829Hz
[Info] [OpenGLDepthPacketProcessor] avg. time: 29.6608ms -> ~33.7146Hz
[Info] [DepthPacketStreamParser] 19 packets were lost
[Info] [DepthPacketStreamParser] 34 packets were lost
[ INFO] [1533689325.342012071]: [Kinect2Bridge::main] depth processing: ~65.0601ms (~15.370
Hz) publishing rate: ~12.9746Hz
[ INFO] [1533689325.342169225]: [Kinect2Bridge::main] color processing: ~9.72222ms (~102.85
Hz) publishing rate: ~5.65561Hz
[Info] [DepthPacketStreamParser] 16 packets were lost
[Info] [DepthPacketStreamParser] 17 packets were lost
[ INFO] [1533689328.345229779]: [Kinect2Bridge::main] depth processing: ~60.3559ms (~16.568
Hz) publishing rate: ~12.9864Hz
[ INFO] [1533689328.346245105]: [Kinect2Bridge::main] color processing: ~11.6745ms (~85.656
Hz) publishing rate: ~5.66076Hz
[Info] [DepthPacketStreamParser] 31 packets were lost
[Info] [DepthPacketStreamParser] 3 packets were lost
```

Figura 5.3: Resultado del comando `kinect2_bridge`. Autoría propia.

Una vez que este comando ha sido exitoso, se puede utilizar el siguiente comando para visualizar la salida:

```
1 rosrun kinect2_viewer kinect2_viewer kinect2 sd cloud
```

Lo cuál produce la salida mostrada en las figura 5.4.

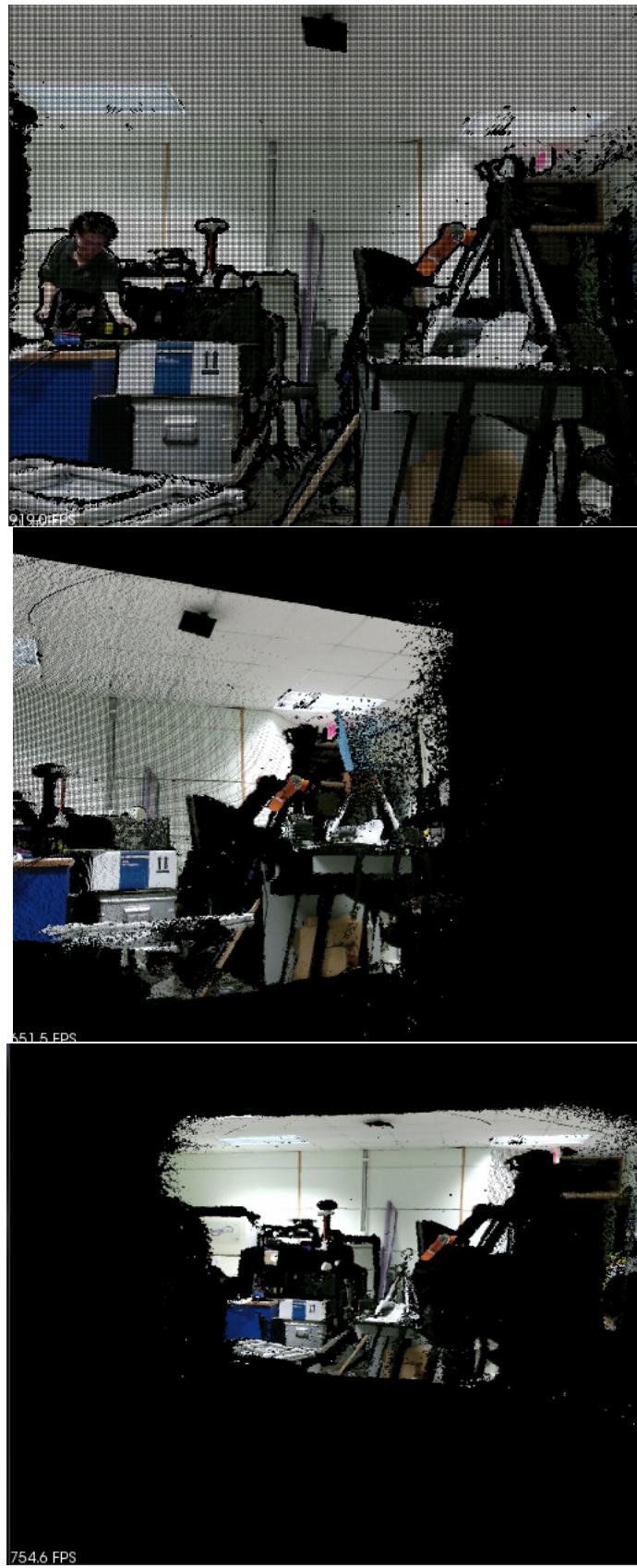


Figura 5.4: Prueba de la salida del kinect, utilizando el visor por defecto. Autoría propia.

Finalmente, ROS posee una herramienta llamada *RQT\_GRAPH*, la cuál ayuda a visualizar las conexiones entre los nodos y topicos siendo utilizados en el momento. Usando Rviz, y esta herramienta, vemos los topicos disponibles cuando se utiliza el nodo. La figura 5.5 muestra el resultado de *RQT\_GRAPH*.

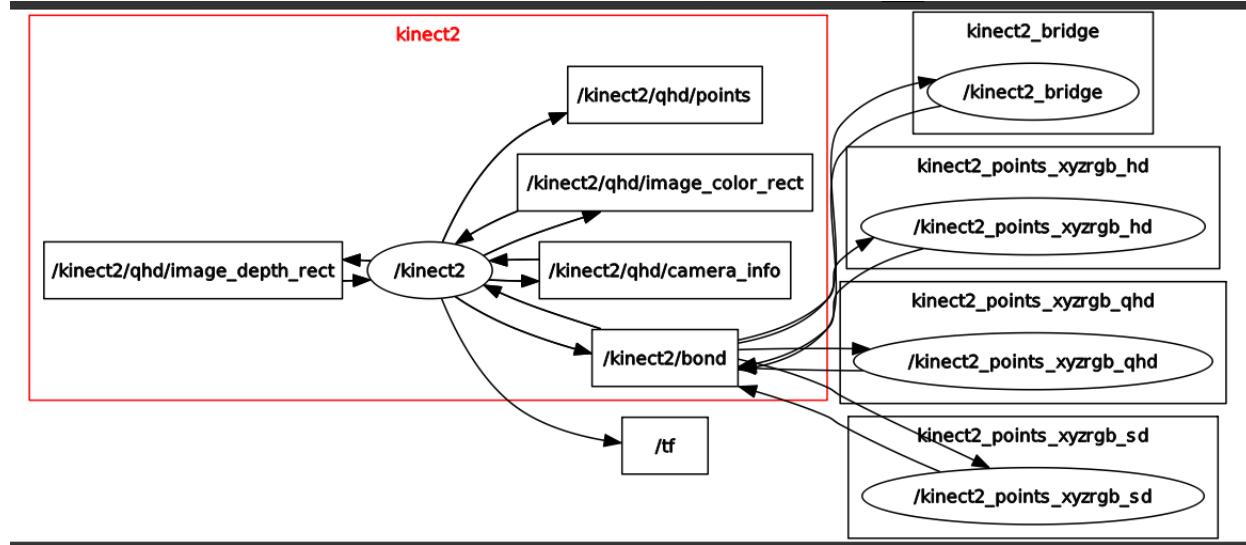


Figura 5.5: Esquema generado por *RQT\_GRAPH* que muestra las topicos disponibles por Iai Kinect. Autoría propia.

#### 5.1.4. Retos

En esta etapa, el único reto que se encontró fué que el nodo de ROS por defecto no posee permisos para utilizar el USB, por lo que se le deben otorgar los mismos. La solución se menciona el repositorio libfreenect2.

### 5.2. Hokuyo

El uso del Hokuyo como parte del proyecto, es utilizar simplemente por razones de prueba. Se desea utilizar el Kinect en vez de este sensor pues es considerablemente más barato, y proporciona información en 3D. Sin embargo, es posible que la navegación sea más sencilla utilizando un sensor 2D.

#### 5.2.1. Instalación del software

La instalación es relativamente sencilla, pues consiste meramente en instalar un paquete de ROS llamando *Hokuyo\_node*. La página oficial del paquete se encuentra [aquí](#). Una vez instalado el paquete, iniciararlo es bastante sencillo.

```
1 rosrun hokuyo_node hokuyo_node
```

### 5.2.2. Pruebas de funcionamiento

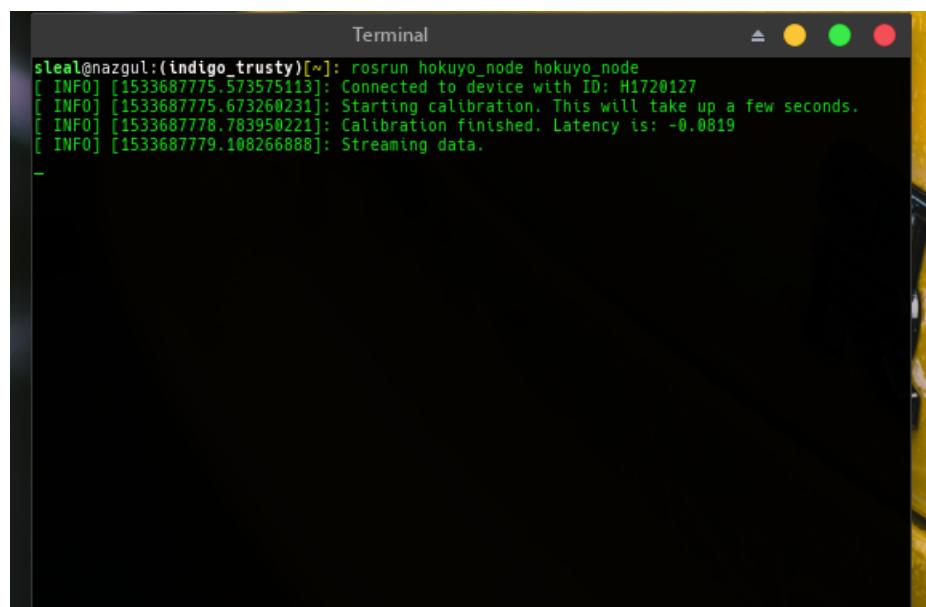


Figura 5.6: Inicialización correcta del módulo hokuyo\_node en ROS. Autoria propia.

La figura 5.6 muestra la salida de la terminal que confirma que el módulo ha sido inicializado correctamente.

Utilizando *RQT\_GRAPH* como se hizo anteriormente, se confirma que el nodo publica la información del laserscan correctamente en un nodo llamado `{scan}`, como era de esperarse. La figura 5.7 muestra la salida del programa.

Además, la figura 5.8 muestra la visualización del sensor laser, utilizando la herramienta *Rviz*.

### 5.2.3. Retos

Con las instalación del software *Hokuyo\_node* no se encontró ningún reto.

## 5.3. Conexión de un STM32F4 y un roboclaw

Como se ha mencionado en capítulos anteriores, el STM32F4 será quién se conecta con la PC principal, recibe mensajes, y controla la velocidad de los motores enviando comandos de movimiento al Roboclaw. A continuación se muestra el proceso de conexión.

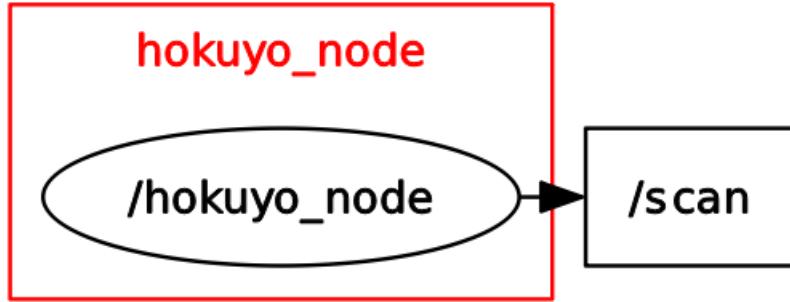


Figura 5.7: Diagrama generado por *RQT\_GRAPH*, demostrando la salida de hokuyo\_node. Autoría propia.

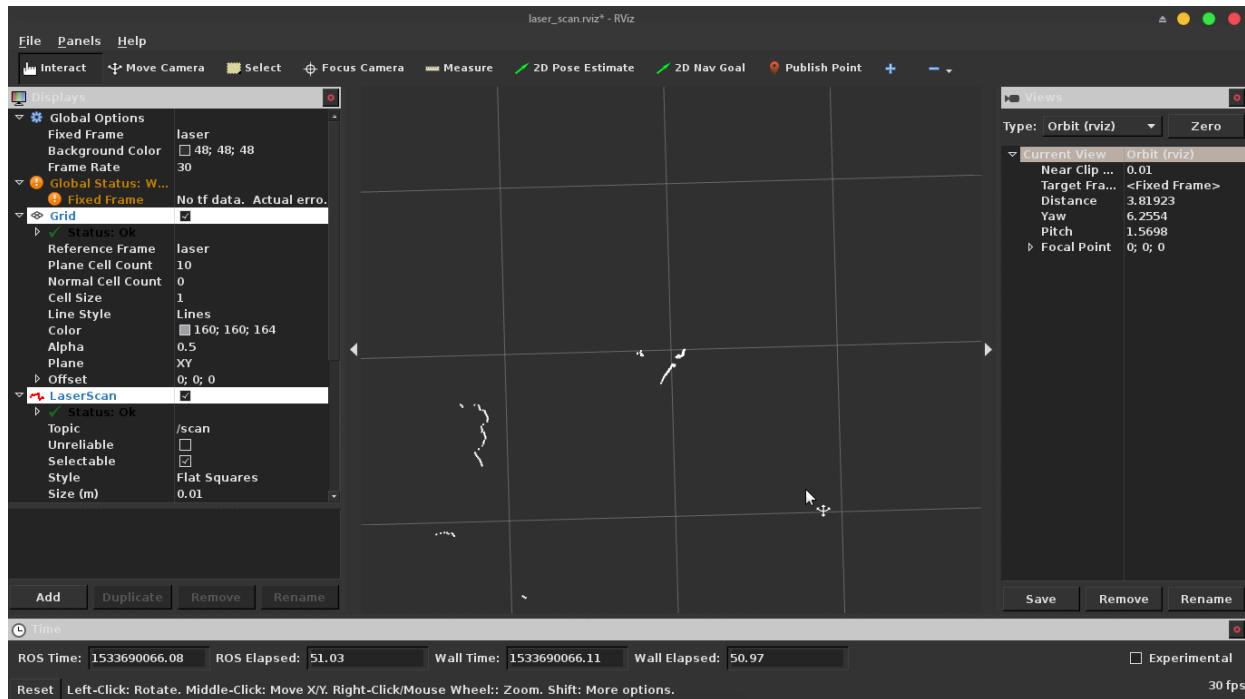


Figura 5.8: Visualización de la salida del sensor láser Hokuyo utilizando Rviz. Autoría propia.

### 5.3.1. Conexión

Antes de iniciar con la conexión completa mostrada en la figura 4.2, se realizaron pequeñas pruebas de comunicación, para diseñar la librería de comunicación, y probar la configuración

correcta del equipo. La figura 5.9 muestra la conexión que se realizó.

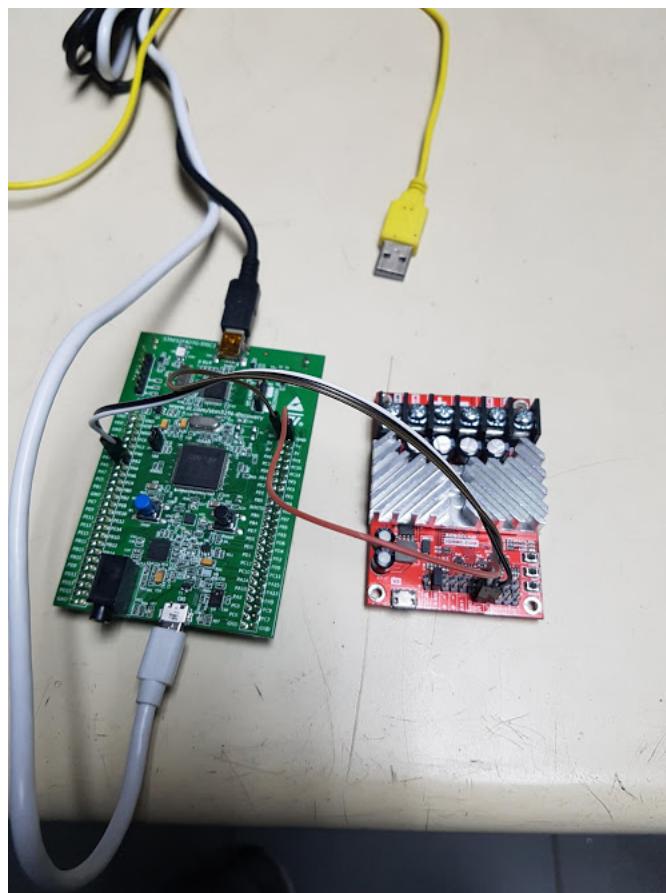


Figura 5.9: Conexión serial en los puertos USART de ambos dispositivos. Autoría propia.

Para alimentar la parte lógica del controlador Roboclaw, existen dos opciones. Se puede configurar para alimentarse de la batería principal (de donde se toma energía para mover los motores) o se puede utilizar una alimentación lógica. Debido a que el STM32F4 no está diseñado para proporcionar suficiente corriente para una alimentación lógica de dos controladores, se optó por utilizar la opción de alimentación de la batería principal. La figura 5.10 muestra la batería que se utilizó para realizar todas las pruebas.

El código que controla el microcontrolador stm32f411 se puede encontrar en el repositorio [stm32-roboclaw](#). En específico el commit donde se realizan las primeras pruebas de conexión con el roboclaw se puede encontrar [aquí](#).



Figura 5.10: Batería de 12V utilizada para alimentar a los controladores Roboclaw. Autoría propia.

### 5.3.2. Retos

En esta etapa del proyecto, las conexiones eléctricas eran bastante sencillas, pues realmente sólo se necesitan cuatro cables (RX, TX y dos tierras) para probar la conexión.

En el aspecto de la programación, fué fundamental la lectura del código escrito por IonMotion para realizar la misma tarea desde una computadora. El código mencionado se puede encontrar [aquí](#). El manual de Roboclaw también jugó un papel importante proporcionando la función que calcula el crc16 bits.

Posteriormente se realizaron cambios importantes a la librería, conforme algunos otros retos fueron surgiendo. Por ejemplo se implementó una función “padre” que llamara a las instrucciones 0,1,4,5 según correspondiera.

Además, se reimplementaron las instrucciones de libopencm3 *usart\_send\_blocking* y *usart\_recv\_blocking* con timeout. El commit donde se realizó dicho cambio se puede encontrar [aquí](#).

## 5.4. Encodificador de cuadratura

### 5.4.1. Conexión

El sensor codificador de cuadratura, en el caso de los motores que se van a utilizar, utilizan un imán y dos sensores magnéticos con un desfase mecánico de  $90^\circ$  para producir las señales. La imagen 5.11 muestra el sensor mencionado.

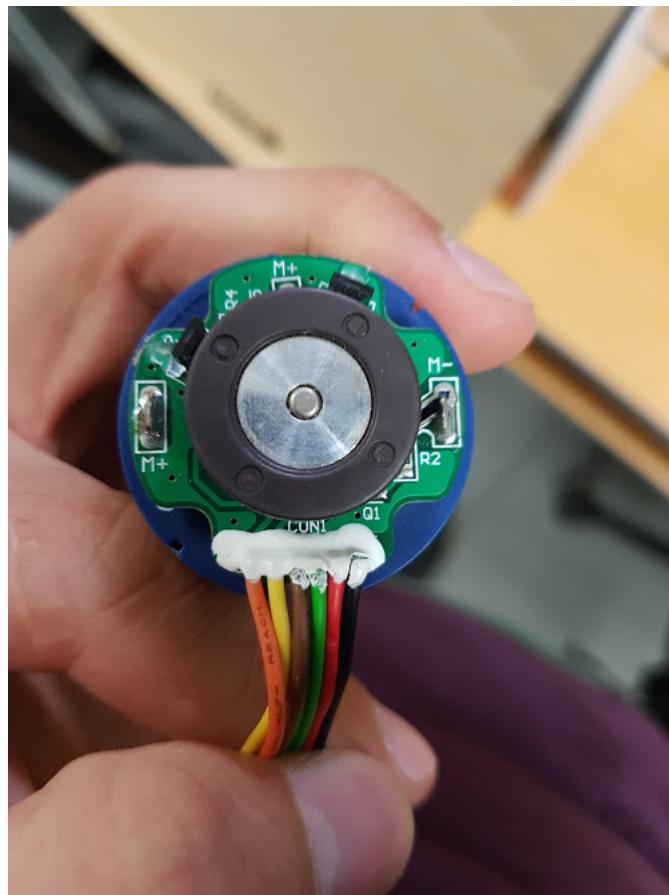


Figura 5.11: Encodificador de cuadratura. Autoría propia.

Del motor, salen 6 cables que se pueden observar en la figura 5.12.

Siguiendo el diagrama que se encuentra en la figura 4.2, junto con el manual del fabricante de los motores, encontramos que la señal DC+ y DC- del motor son los cables rojo y negro respectivamente. La alimentación DC+ y DC- del sensor de cuadratura son los cables anaranjado y verde respectivamente. Finalmente el cable amarillo corresponde al canal A, mientras que el cable café corresponde al canal B. La figura 5.13 resume lo anteriormente descrito.

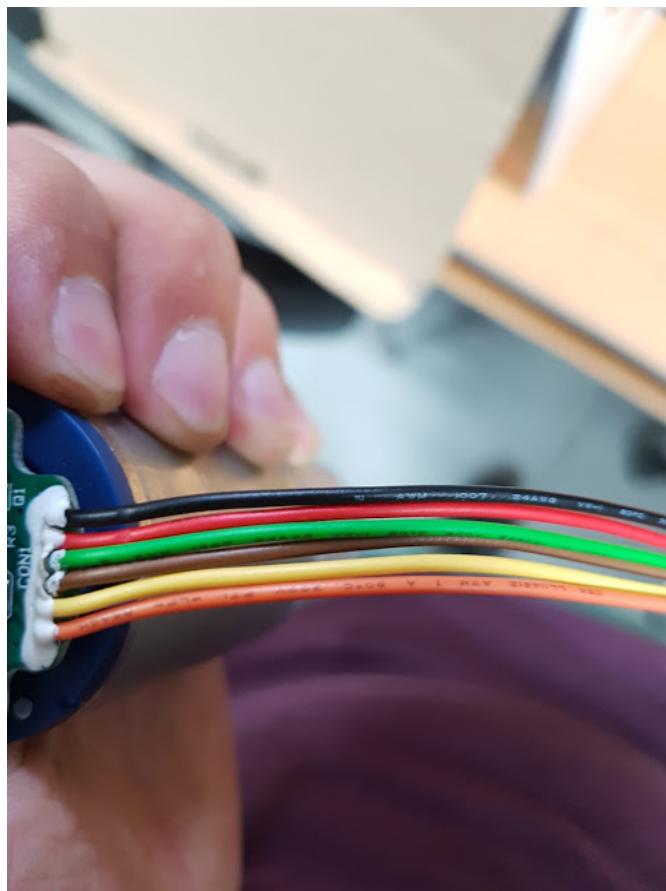


Figura 5.12: Cables del motor DC Actobotics. Autoría propia.

#### 5.4.2. Programación

La programación de los cálculos del encoder de cuadratura conllevó varias iteraciones, puesto que hay varios factores a tomar en consideración. Por ejemplo, tomar en cuenta el caso en donde se da un rebase del contador del timer, y la acción a tomar una vez que esto sucede. La primera iteración funcional se puede ver en este [commit](#). Sin embargo, si se observa la historia, la librería ha sufrido varios cambios posterior a estos.

Actualmente, existe un error con esta sección específica, por lo cual debe ser revisada.

#### 5.4.3. Retos

Algunos de los retos que se encontraron durante el desarrollo de esta sección son:

- En de las conexiones eléctricas debido a “jumpers” defectuosos.



Figura 5.13: Cables de motor Actobotics. Tomado del manual de usuario.

- En la PCB del sensor. Daño en los cables del motor, como se observa en la figura 5.14.
- Con la estructura del código, es específico la manera en la que se calcula la velocidad actual del motor.



Figura 5.14: Reparación de cableado en un motor, debido a daños en las conexiones. Autoría propia.

## 5.5. Controlador PID

El controlador PID se diseñó con el propósito de ser una función completamente configurable. Por ejemplo, la frecuencia en la que se actualiza la velocidad actual, en ms es un parámetro, ya que se debe controlar la cantidad instrucciones que se mandan al roboclaw, para que las mismas no fallen. Además, los parámetros del límite superior, y las constantes de ganancia son configurables, como es de esperar.

### 5.5.1. Implementación

La primera versión del código completo, junto con K<sub>p</sub>, K<sub>i</sub>, y K<sub>d</sub> fué [este](#) commit.

Anteriormente existieron otras iteraciones funcionales, pero únicamente tenían activados K<sub>p</sub> y K<sub>i</sub>.

### 5.5.2. Sintonización

La primera sintonización se realizó imprimiendo la información en pantalla, y usando un algoritmo empírico propuesto por Federico Ruiz. Esta primera sintonización funcionó bastante bien para la mayoría de pruebas en el robot que se realizaron posteriormente. El algoritmo para sintonizar consiste en aumentar K<sub>p</sub> hasta tener oscilaciones, regresar al punto donde no hay oscilaciones y aumentar K<sub>i</sub>. K<sub>d</sub> se utilizó en algunas pruebas, pero se determinó que no era necesario para la etapa en donde se encontraba el proyecto.

Es factible que posteriormente se haga un trabajo más profundo en lo que es la sintonización de este PID, sin embargo de momento no es un problema.

### 5.5.3. Retos

Primeramente, se encontró que el algoritmo que estaba siendo utilizado para el control PID era incorrecto. Esto puesto que inicialmente el cálculo que se estaba realizando para el error no contemplaba que la velocidad podía ser negativa o positiva, y el cálculo varía de acuerdo a esto. Esto se solucionó utilizando un programa para graficar las velocidades y los valores del PID en cada instante, justamente para encontrar errores.

Segundo, se encontró un fallo intermitente en las conexiones del encoder, lo cuál causaba errores difíciles de detectar. Finalmente se pudo solventar moviendo las conexiones mientras se realizaban pruebas con el robot en el aire.

Se encontró que se realizaba el cálculo incorrecto de las velocidades de cada motor, lo cuál afecta el control cerrado. Esto puesto que las velocidades se calculaban utilizando el tiempo transcurrido como una constante, sin embargo esto no era cierto para todos los casos.

Finalmente, se encontró que había un mal ajuste de la frecuencia de comandos hacia el Roboclaw. Esto ocasionaba que el controlador de los motores no respondiera correctamente a los comandos. Bajando la frecuencia de los comandos se solucionó el problema.

### 5.5.4. Gráficas

Debido a los últimos retos inesperados en la odmetría del robot, se procedió a diseñar funciones de depuración en todos los aspectos anteriormente diseñados. Por lo tanto, se decidió realizar gráficas de la velocidad de los motores, y la posición. Indirectamente estas gráficas muestran el funcionamiento del PID, donde se aplica un cambio en la referencia a los 0.5s. Las figuras 5.15 y 5.16 muestran lo anteriormente mencionado.

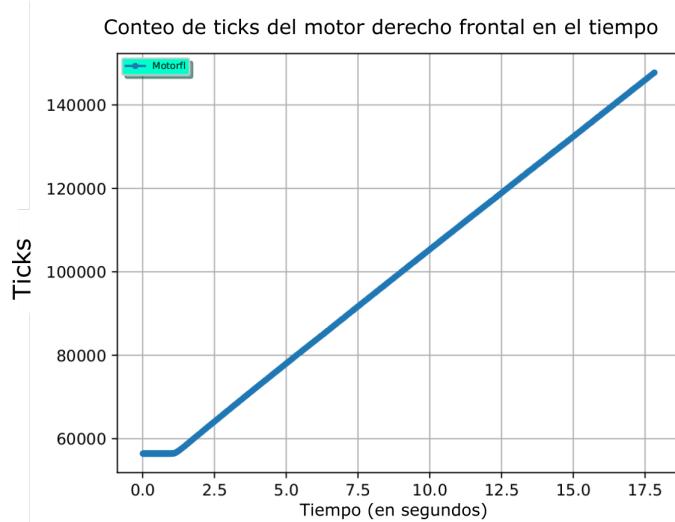


Figura 5.15: Tiempo versus la posición del motor en ticks del encoder, ante un cambio en la referencia en 0.5s. Autoría propia.

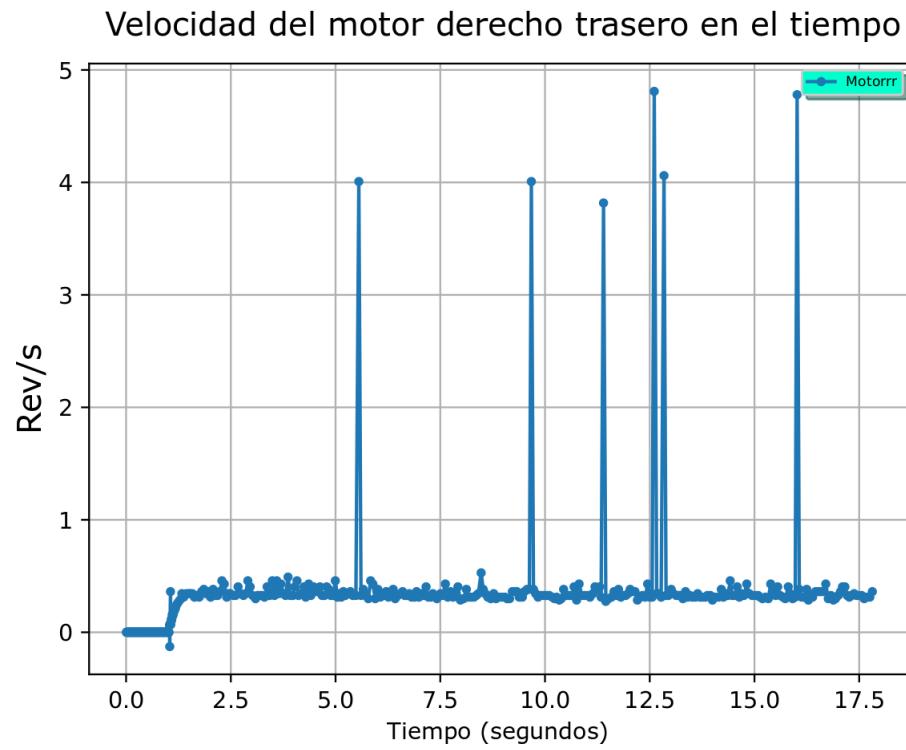


Figura 5.16: Tiempo versus la velocidad del motor en rev/s, ante un cambio en la referencia en 0.5s. Autoría propia.

## 5.6. Ensamble de la base

A continuación se muestran las capturas del ensamblaje. El proceso requirió armar ruedas junto con los acoplos, y realizar todas las conexiones eléctricas. Las figuras 5.17, 5.18, 5.19, 5.20, 5.21 muestran el proceso de ensamblado.



Figura 5.17: Acople y rueda a utilizar durante el ensamblaje. Autoría propia.



Figura 5.18: Proceso inicial del ensamblaje. Autoría propia.

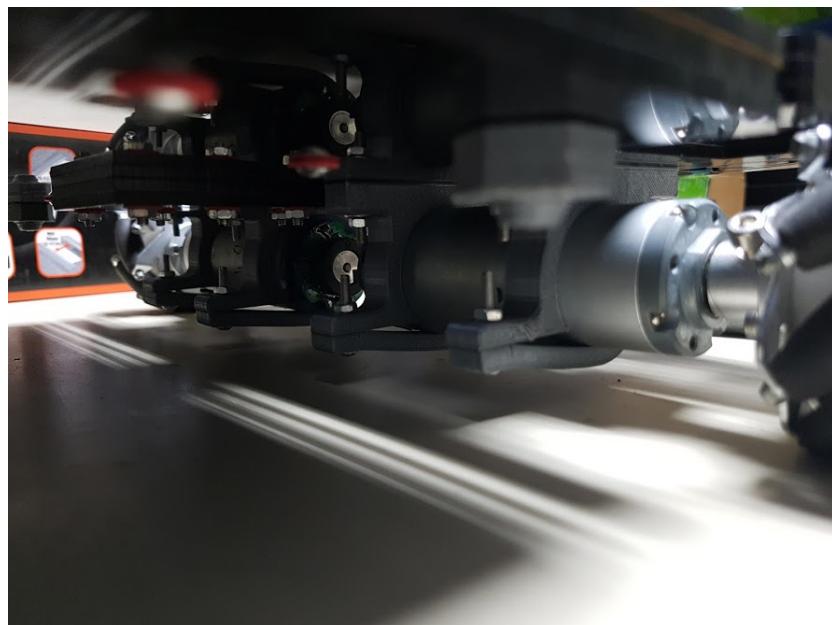


Figura 5.19: Soporte de las ruedas. Autoría propia.

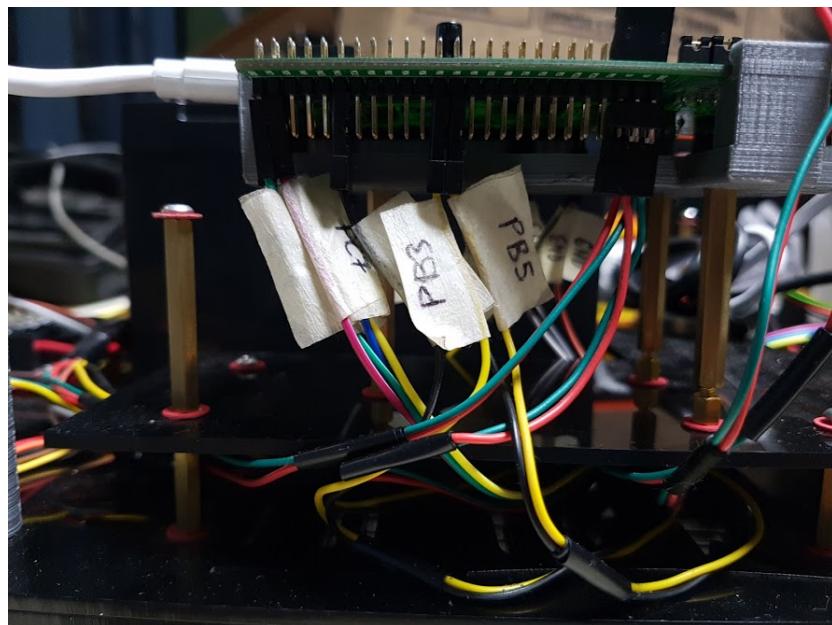


Figura 5.20: Etiquetado de las conexiones. Autoría propia.

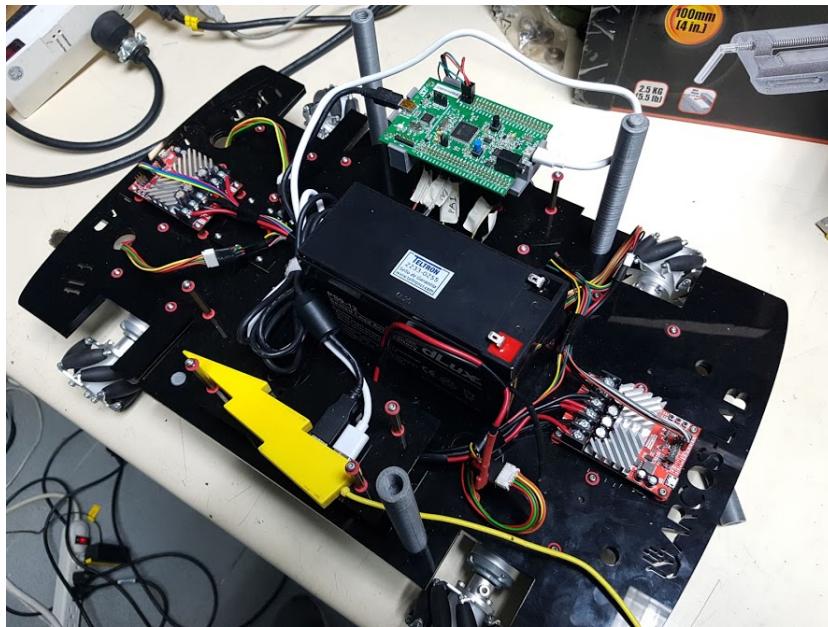


Figura 5.21: Plataforma ensamblada totalmente. Autoría propia.

### 5.6.1. Retos

- La mayor parte de los retos relacionados con esta sección involucran el cableado eléctrico. Muchos de los cables utilizados no hicieron buen contacto, por lo que algunas conexiones no eran estables.
- Falta de piezas impresas en 3D, y ruptura de algunas.

### 5.6.2. Pruebas de movimiento

A continuación en [este](#) link se puede encontrar un video de la primera prueba de movimiento de los cuatro motores, junto con [este](#) video, el cuál muestra los contadores de cada encoder incrementando durante la prueba.

## 5.7. Protocolo de comunicación hacia la PC

Para esta sección se realizaron varias iteraciones, y fué de los aspectos más problemáticos del proyecto.

### 5.7.1. Prototipos y retos

Se inció con un protocolo que realizaba la transmisión de información, enviando la información como un string, y no byte por byte. Es decir, 4.123 lo enviaría como 4 (1 byte), . (1 byte), 1

(1 byte), 2 (1 byte), 3 (1 byte). Esto hace que la transmisión sea muy ineficiente, pues para un número flotante, que necesita 4 bytes para guardarse en memoria, se necesitan muchos bytes para mandarlo como string.

El mayor reto surgió dado que al ser variable la cantidad de datos que transmite (cuando se transmite 1.23 son cuatro bytes, pero un número como 1.299999999999999 consume más bytes) es variable, se llenaba el espacio de memoria dinámica en la recepción, y el programa fallaba.

Por esta razón se implementó el segundo prototipo, el cuál manda toda la información en bytes. Esto provoca que la cantidad de datos transmitidos sean constantes, y el protocolo sea más eficiente. Además se agregó el checksum crc16 bits para no ejecutar instrucciones corruptas, pues esto en algún momento causó problemas también.

### 5.7.2. Libopencm3-plus

Existieron ciertos retos con esta librería, puesto que una sección de código insertaba un “\r” después de cada “\n” en la transmisión serial. Este bug fué el reto más difícil de encontrar durante el desarrollo del proyecto.

### 5.7.3. Retos

- Consumo de memoria dinámico por mal diseño de protocolo de comunicación.
- Caractéres inesperados enviados por parte de la librería libopencm3-plus.

## 5.8. Implementación del paquete en ROS

El diseño del paquete en ROS se realizó utilizando como referencia código escrito para controlar el Robot llamado Neato. Esta librería se puede encontrar [aquí](#). Nos interesa en particular, el código que se encuentra dentro de neato\_robot/nodes/neato.py puesto que este script se encarga de suscribirse al tópico *cmd\_vel* y publicar información de *tf* y de *odom*.

Se creó un repositorio con este código, el cuál se puede encontrar [aquí](#). En cuestión de dificultades no se encontraron muchos, gracias a tener el ejemplo de código mencionado anteriormente. Se realizaron cambios de prueba en la frecuencia del nodo, sin embargo no hay cambios en la estructura como tal.

Las instrucciones de instalación del repositorio se encuentran en el *README.md* del repositorio. La figura 5.22 muestra un ejemplo de las dependencias a instalar, y las instrucciones para instalar el paquete como tal.

## **omnidirectional\_robot**

Ros module for Arcos Lab omnidirectional robot.

## **Dependencies**

You'll need to install rospkg (In case you don't already have it) and pyserial in order to run this code.

```
pew new ros
pip install pyserial rospkg catkin_pkg numpy
```

**NOTE:** You'll need to use pew workon ros each time you want to use this package. Read more about pew in case you don't know what it is: <https://pypi.org/project/pew/>

## **Installation**

In order to use this ros driver, you need to clone this code in your catkin workspace:

```
cd ~/catkin_ws/src
git clone https://github.com/slealg/omnidirectional_robot/
cd ../
catkin_make
```

## **Ilustración**

Figura 5.22: Instrucciones de instalación e instalación de dependencias. Autoría propia.

El launch file que se debe correr para levantar el programa, es el que se encuentra en la dirección *omnidirectional\_robot \omnidirectional\_node \launch \bringup.launch*. Se levanta el nodo corriendo el siguiente comando, una vez que ya se tiene instalado el paquete con en Catkin correctamente.

```
1 rosrun omnidirectional_node bringup.launch
```

La visualización de los tópicos que se levantan cuando se corre este comando se puede observar en la figura 5.23. En este punto, se encuentran levantados todos los tópicos necesarios para realizar la navegación.

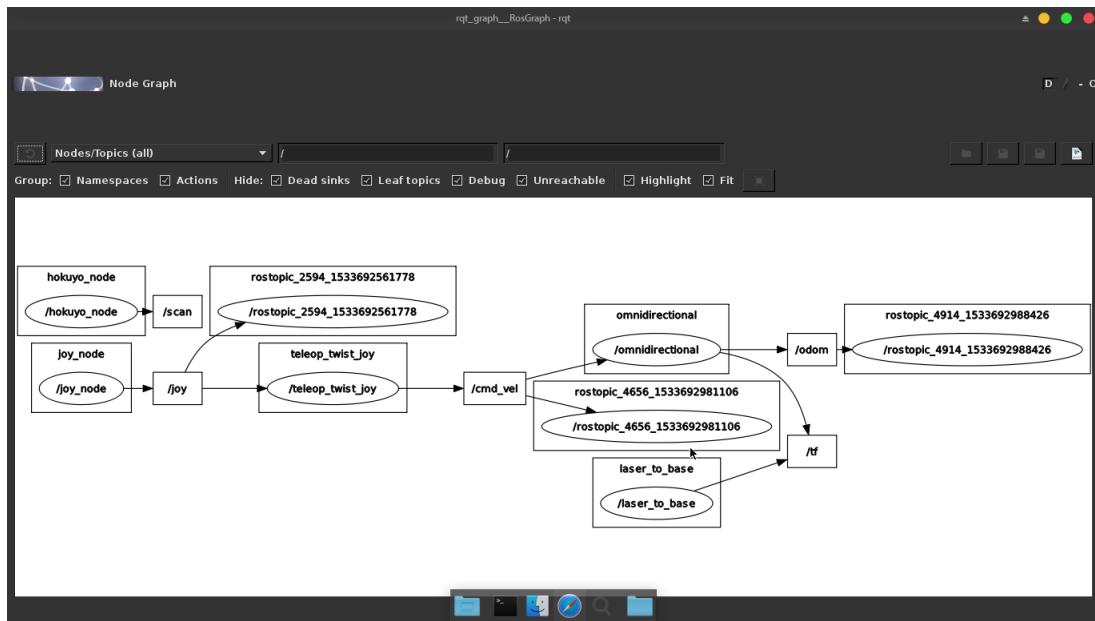


Figura 5.23: Visualización de las relaciones entre tópicos y nodos levantados en ROS. Autoría propia.

## 5.9. Cálculo de la odometría

El cálculo de la odometría se realizó hasta este punto, puesto que se necesitaba primero enviar comandos de movimiento hacia la base, y mover la base sobre alguna superficie para poder medir la veracidad de las mediciones de odometría.

### 5.9.1. Retos

Las ecuaciones que se siguieron son realmente las mostradas en las sección de diseño, y no hubieron dificultades en implementar las ecuaciones.

Sin embargo, se encontraron varios retos por resolver, algunos de los cuales están relacionados a las interrupciones systick.

Cuando sucede una interrupción en Systick, se modifica el valor de las variables globales de posición. Si en un dado momento, el ciclo principal está realizando la transferencia de esta variable por comunicación serial, y el ciclo systick realiza una interrupción de hardware, el valor enviado por el puerto serial será incorrecto. Para esto se crearon unas variables temporales que se actualizan únicamente si una bandera de actualización se activa.

Por lo tanto, cuando se inicia una nueva comunicación serial, se desactiva una señal que deja de actualizar la variable de posición global, mientras se esté realizando la transmisión de la variable. El diagrama mostrado en la figura 4.7 se muestra este proceso, y la bandera *update\_global\_pos* es la mencionada anteriormente.

## 5.10. Pruebas y corrección de errores

De las primeras pruebas que se realizó, fué verificar el movimiento correcto de la base omnidireccional utilizando el joystick. Se ajustaron los parámetros del PID de cada ruedas, para que el mismo fuera más rápido.

Realizando estas pruebas, fué donde se detectó el error mencionado del protocolo de comunicación, en donde la librería *libopencm3-plus* tenía una pulga. Se detectó puesto que en momentos aleatorios del funcionamiento del carrito el stm32f411 quedaba en un estado de abort.

Utilizando gdb se obtuvieron pistas del reto, sin embargo no fué hasta leer el código contenido en la librería que se logró identificar el reto.

Posteriormente se realizó la verificación de la información obtenida de la odometría de las ruedas. Las pruebas que se realizaron fué medir con una cinta 1 metro, realizar marcas en el piso y realizar pruebas con movimientos hacia adelante, laterales y rotacionales.

El cuadro 5.1 resume algunos resultados obtenidos de las pruebas.

Cuadro 5.1: Tabla de dos mediciones realizadas para comprobar el cálculo de odometría en la base. Autoría propia.

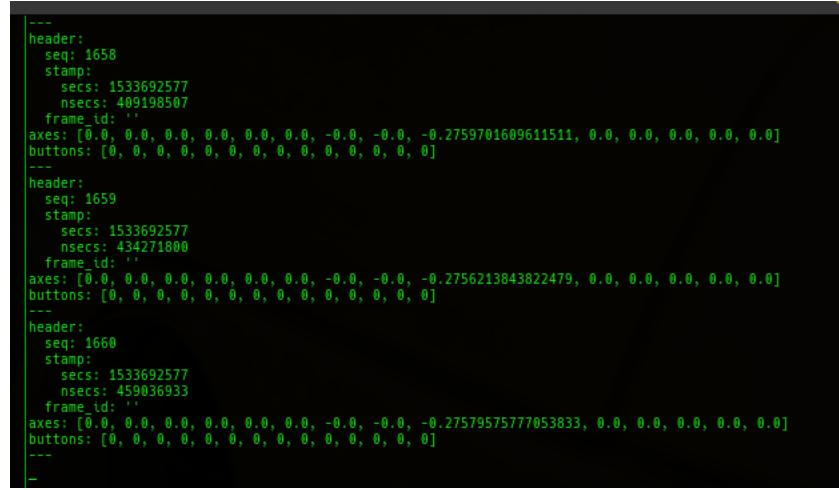
Variable	Medición 1	Medición 2	Medición esperada (m)
X_x	1.034 m	1.045 m	1.000 m
X_y	1.096 m	1.087 m	1.000 m
$\alpha_z$	405°	395°	365°

Como se puede observar, se muestran resultados bastante desviados de los valores reales. Esto se debe a errores o en el cálculo de la odometría, o en los parámetros de las ecuaciones de la kinemática.

Estos errores se deben corregir para poder implementar correctamente el algoritmo de SLAM.

## 5.11. DualShock 4 en ROS

Para conectar el control de PS4 a ROS, se utilizó un paquete llamado [ps4-ros](#). Las instrucciones de instalación en el repositorio son bastante sencillas de seguir, por lo que esta etapa no demostró mucho. Una de las dependencias es utilizar un paquete llamado [ros-joy](#) el cual contiene la información del tipo de mensaje que se debe publicar para la información de un joystick.



```

---  

header:  

  seq: 1658  

  stamp:  

    secs: 1533692577  

    nsecs: 409198507  

    frame_id: ''  

axes: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.0, -0.0, -0.2759701609611511, 0.0, 0.0, 0.0, 0.0, 0.0]  

buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  

---  

header:  

  seq: 1659  

  stamp:  

    secs: 1533692577  

    nsecs: 434271800  

    frame_id: ''  

axes: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.0, -0.0, -0.2756213843822479, 0.0, 0.0, 0.0, 0.0, 0.0]  

buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  

---  

header:  

  seq: 1660  

  stamp:  

    secs: 1533692577  

    nsecs: 459036933  

    frame_id: ''  

axes: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.0, -0.0, -0.27579575777053833, 0.0, 0.0, 0.0, 0.0, 0.0]  

buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
---  

-
```

Figura 5.24: Muestra de la salida del nodo *ps4-ros*. Autoría propia.

El producto que genera este módulo, es la publicación de botones y estados de los joystick, en un tópico llamado *\joy* (ver figura 5.24). Esta información debe ser convertida primero a información compatible para el tópico *\cmd\_vel*. El nodo se levanta corriendo el siguiente comando:

```
1 rosrun ps4-ros ps4-ros
```

Un módulo llamado *teleop\_twist\_joy* realiza la función de convertir la información de un mensaje del tipo *\joy* a información del tipo *\twist*, en un tópico llamado *cmd\_vel*. El nodo se levanta utilizando el siguiente comando:

```
1 roslaunch teleop_twist_joy bringup.launch
```

En la figura 5.25 se muestra la conexión establecida entre el control de PS4 y la computadora principal.



Figura 5.25: Uso del control de PS4 para enviar comandos hacia el robot de velocidad. Autoría propia.

Algunos videos de muestra se pueden mostrar en el siguiente [link](#). Acá se prueba el movimiento lateral, y frontal. Este [link](#) muestra el movimiento de la odometría y TF utilizando movimientos en el joystick.

## 5.12. Implementación de Navegación

Para este punto, se logró implementar correctamente lo siguiente:

- Se habilita la recepción de comandos de velocidad a través del tópico llamado *cmd\_vel*.
- Se publica información de la odometría del robot a través del tópico llamado *odom*
- Se publica la información de los sensores láser en el tópico llamado *laser*

- Se publica la información de las transformaciones entre el mapa, la posición global, la posición de la base del robot, la base del láser, y la información de la odometría y la información del láser correspondientemente.
- Se habilita el nodo llamado *gmapping* para iniciar el proceso de mapeo.

Para habilitar la navegación en el robot, se creó una guía en el repositorio en donde están las instrucciones necesarias para realizar el inicio del mapeo: [NAVIGATION](#)

Básicamente, una vez que se tiene corriendo el nodo *teleop\_twist\_joy* se debe correr el nodo *omnidirectional\_node*, el cuál levanta los procesos de comunicación con la base.

```
1 rosrun omnidirectional_node bringup.launch
```

Finalmente, corremos el nodo que controla el láser utilizado para realizar las pruebas: *hokuyo\_node*. Y por último correr el proceso de mapeo sobre el tópico *scan*:

```
1 rosparam set hokuyo_node/port /dev/ttyACM1
2 rosrun hokuyo_node hokuyo_node
```

En los comandos de arriba, se debe verificar cuál es el puerto serial al cuál se tiene conectado el sensor láser.

Y finalmente, iniciar el proceso de mapeo:

```
1 rosrun gmapping slam_gmapping scan:=scan
```

Para salvar el mapa que se crea, se debe ejecutar el siguiente comando:

```
1 rosrun map_server map_saver -f mymap
```

### 5.12.1. Configuración

Se deben definir varios parámetros importantes, los cuales son:

- *map\_update\_interval*: Es el tiempo en segundos entre dos actualizaciones del mapa. Idealmente, la actualización sería instantánea, sin embargo es muy costoso en términos de procesamiento realizarlo de esta manera. El intervalo por defecto es de 5 segundos.
- *max\_Urange*: El rango máximo para el cuál el láser brinda información válida. La información que se encuentre más lejos que esta distancia será descartada.
- *iterations*: La cantidad de iteraciones del scanmatcher.
- *particles*: El número de partículas usadas en el filtro.
- *xmin*, *ymin*, *xmax*, *y ymax*: Las coordenadas juntas describen el tamaño del mapa.
- *base\_frame*: Indica cuál marco corresponde a la base del robot en el árbol de transformaciones.

NAME	TOPIC	MESSAGE TYPE
ROBOT FOOT-PRINT	/local_costmap/robot_footprint	geometry_msgs/PolygonStamped
LOCAL COSTMAP	/move_base/local_costmap/costmap	nav_msgs/GridCells
OBSTACLES LAYER	/local_costmap/obstacles	nav_msgs/GridCells
INFLATED OBSTACLES LAYER	/local_costmap/inflated_obstacles	nav_msgs/GridCells
STATIC MAP	/map	nav_msgs/GetMap or nav_msgs/OccupancyGrid
GLOBAL PLAN	/move_base/TrajectoryPlannerROS/global_plan	nav_msgs/Path
LOCAL PLAN	/move_base/TrajectoryPlannerROS/local_plan	nav_msgs/Path
2D NAV GOAL	/move_base_simple/goal	geometry_msgs/PoseStamped
PLANNER PLAN	/move_base/NavfnROS/plan	nav_msgs/Path
LASER SCAN	/scan	sensor_msgs/LaserScan
KINECT POINT-CLOUD	/camera/depth_registered/points	sensor_msgs/PointCloud2

Figura 5.26: Algunos de los requisitos necesarios para la navegación con ROS. Tomado de [14].

En lo que respecta al módulo *move\_base*, basa sus técnicas de planeación en la ubicación actual y el destino final. En el código de launch file, se tiene la sintaxis común de un archivo launch, donde se deben definir siete parámetros.

- *base\_global\_planner*: Selecciona el plugin a utilizar. Por defecto, se utiliza el de la serie 1.1
- *controller\_frequency*: Es un parámetro que fija la frecuencia con la que se mandan comandos de velocidad hacia la base.

Además, en el archivo de configuración para el mapa de costos, *costmap\_common\_params* se deben definir ciertos parámetros que usará el *global\_planner*. Por ejemplo:

- *obstacle\_range* and *raytrace\_range*: La distancia mínima en metros que será considerada a la hora de tomar información de un obstáculo y ponerlo en el mapa de costos. *Raytrace\_range* es la distancia máxima que será considerada cuando se toma espacio libre alrededor de un robot y se agrega en un mapa de costos.

### 5.12.2. Resultados del mapeo

Para esta sección, se requirió depurar el código desarrollado del lado del stm32f4. La odometría de las ruedas tenía un error mayor al 10 % en movimientos lineales y de hasta un 40 % en movimientos translacionales, lo cuál significa que el proceso de mapeo no funcionaba correctamente.

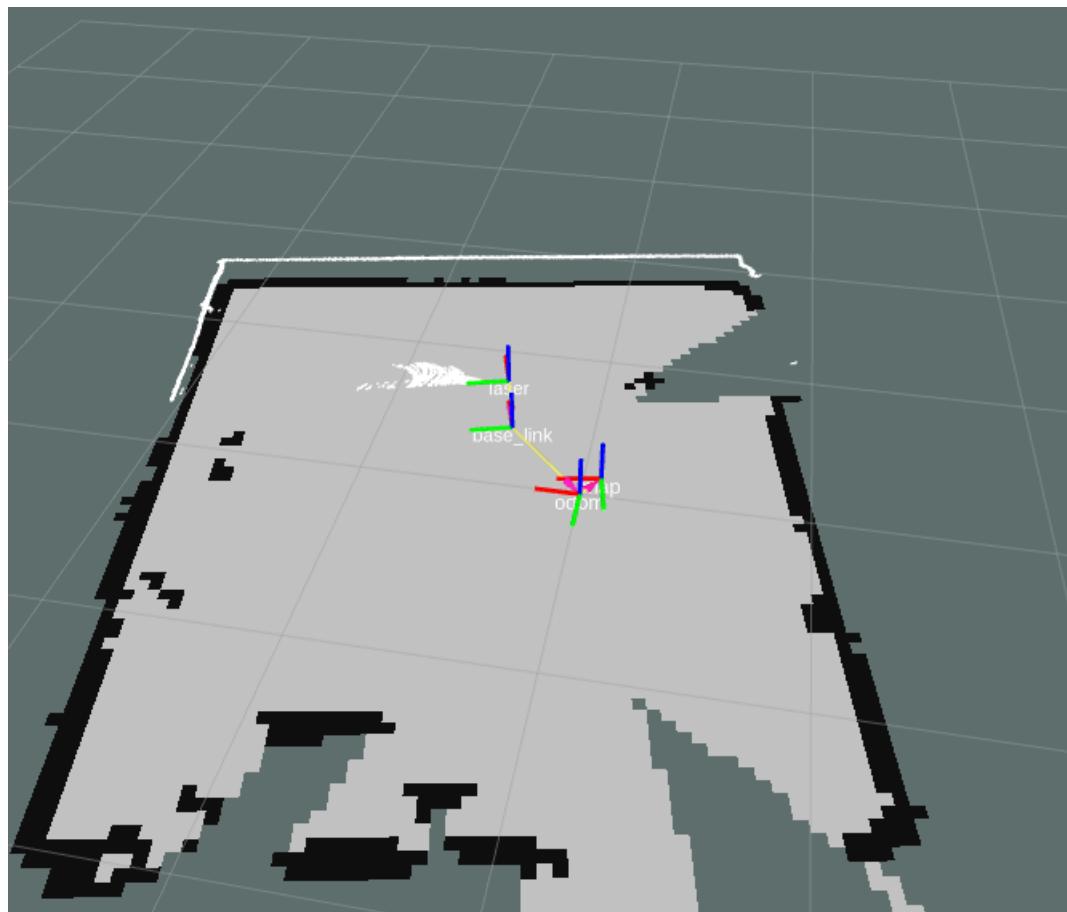


Figura 5.27: Mapeo realizado en el cuarto de la cocina del INII. Autoría propia.

Para esta corrección, se requirió depurar la manera en la que se calcula la velocidad del lado del microcontrolador, y además implementar nuevas funciones en la librería de comunicación serial para determinar cuál era el fallo.

Finalmente, se logró corregir el error, y se iniciaron los primeros mapeos, los cuales se muestran a continuación:

Adicionalmente, se realizaron mediciones para comparar los resultados del mapeo vs las distancias en la realidad. A continuación el cuadro 5.2 muestra algunas de las pruebas realizadas.

Prueba	Medida real	Medida en mapa	Error
Distancia entre paredes	3.5 metros	3.7 metros	5.7 %
Ángulos en paredes	90 grados	90.5 grados	0.5 %

Cuadro 5.2: Medidas comparativas entre el mapa generado por la navegación y la realidad



Figura 5.28: Fotografía real de la cocineta en la que realizó el mapeo anterior.

### 5.12.3. Retos

Como se mencionó anteriormente, para esta sección se debieron corregir algunos retos encontrados durante la implementación de los cálculos de la odometría del robot.

Sin embargo, propiamente con esta sección no se encontraron errores importantes.

El único error que cabe rescatar, es que inicialmente se inició con una transformación del láser, en la dirección relativa a la base incorrecta. Esto causaba que virtualmente, el láser estuviera atrás del centro del robot, pero en la realidad estaba delante del robot.

Este error pudo ser detectado por el profesor Federico Ruiz, al sugerir realizar movimientos circulares en los cuáles se evidencia más este error.

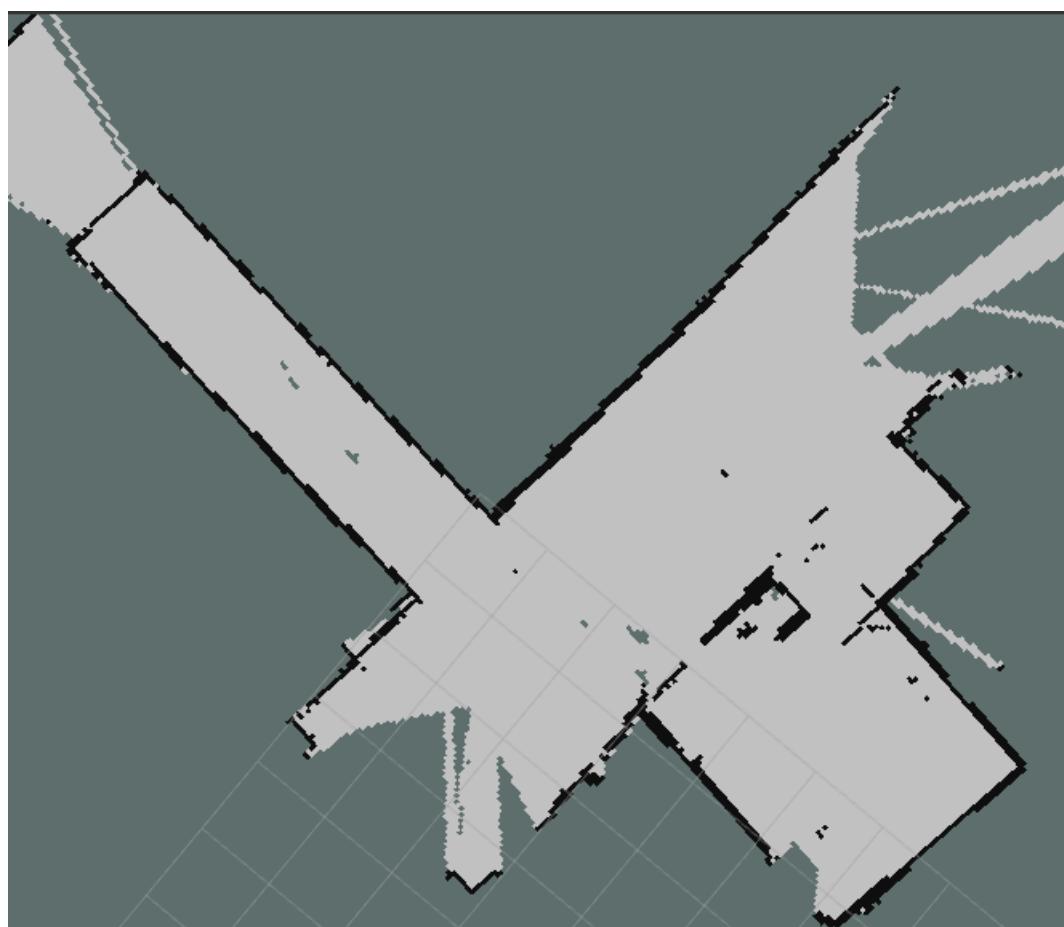


Figura 5.29: Mapeo realizado en el ala Sur del INII. Autoría propia.

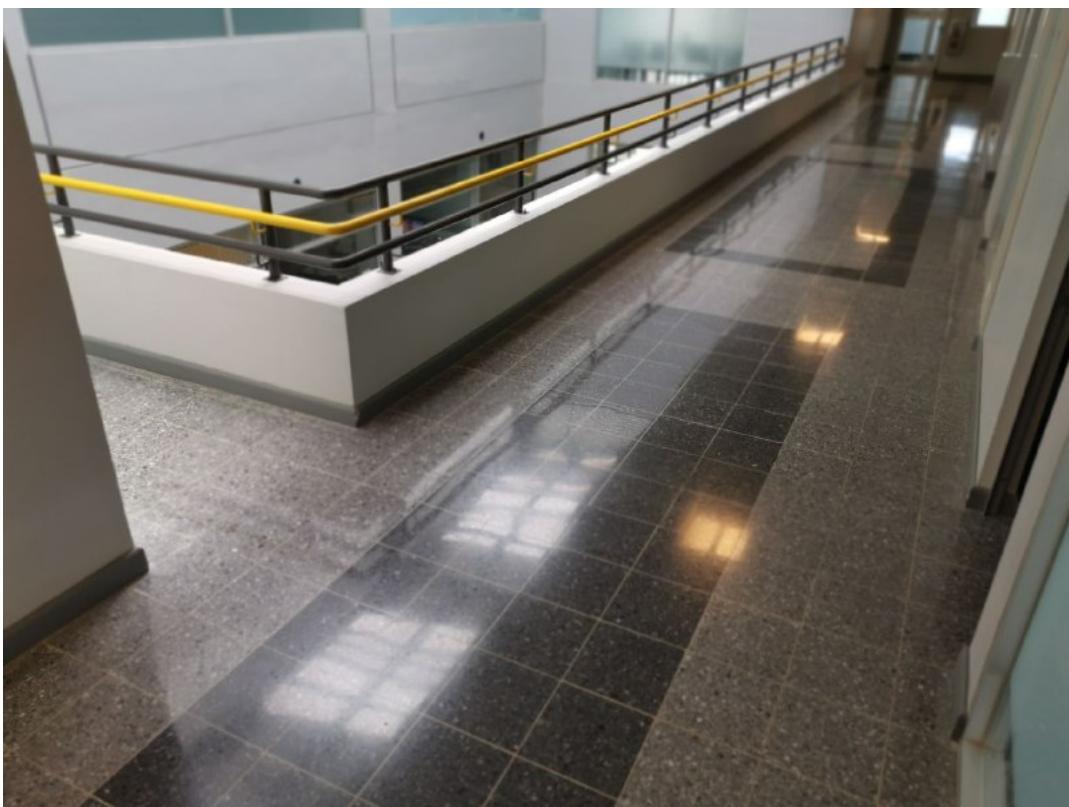


Figura 5.30: Fotografía real del pasillo en el que se realizó el mapeo anterior.

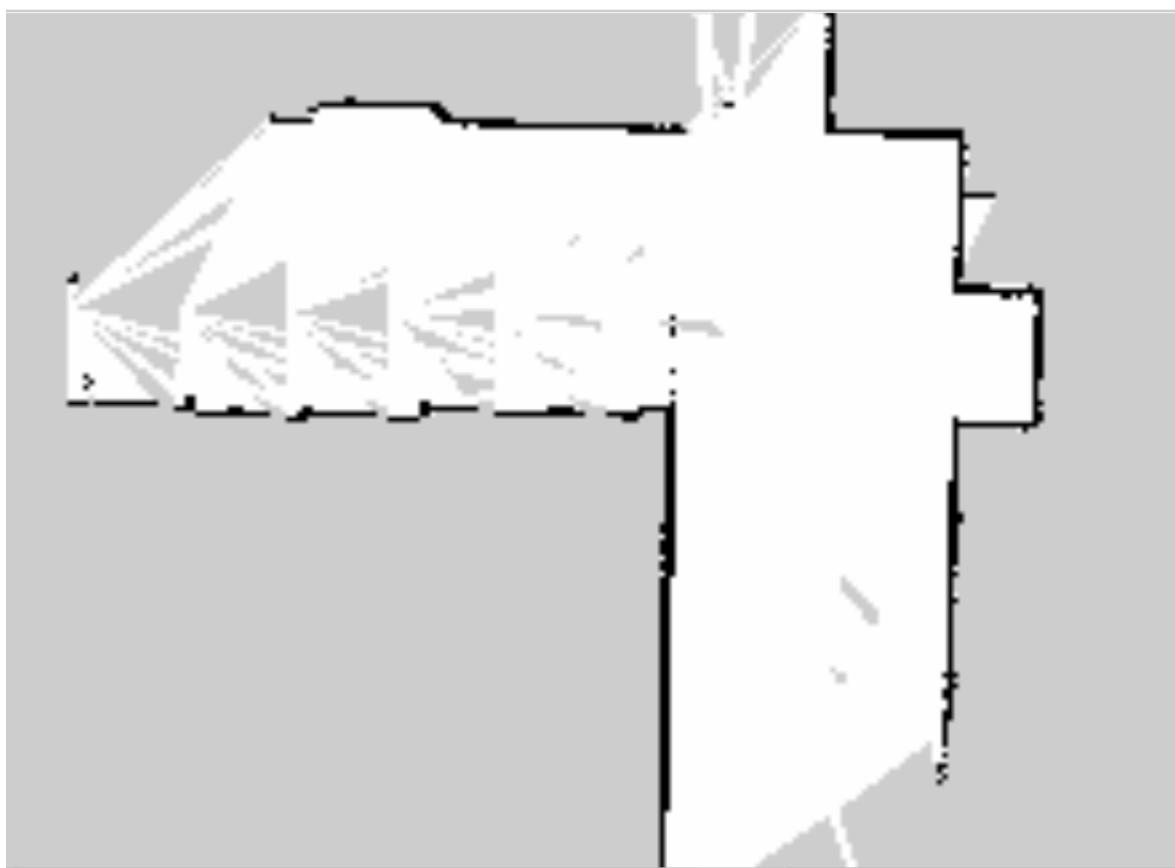


Figura 5.31: Mapeo realizado en la esquina a las afueras del ARCOS-LAB. Autoría propia.

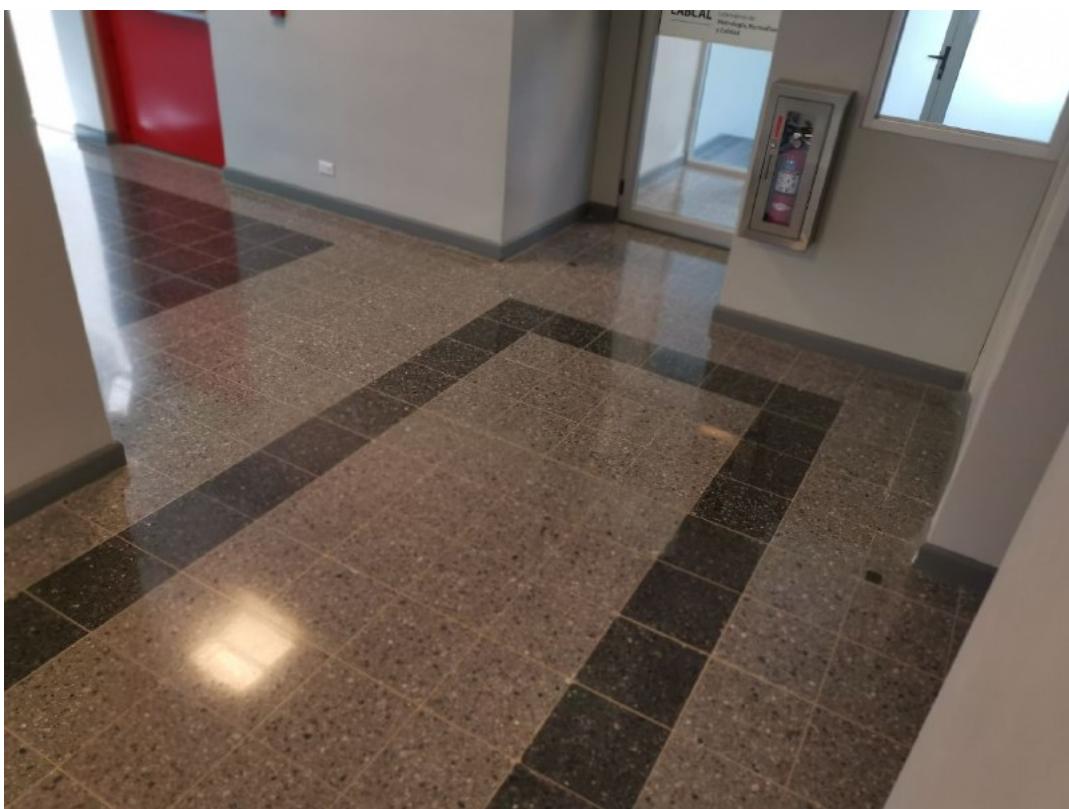


Figura 5.32: Fotografía real del pasillo en el que se realizó el mapeo anterior.

## CAPÍTULO 6

# CONCLUSIONES Y RECOMENDACIONES

A modo de cierre, el correspondiente capítulo viene a culminar el proceso de implementación, enumerando las conclusiones principales de este proyecto. Es importante mencionar que

### 6.1. Conclusiones

El laboratorio de investigación ARCOS-Lab ha llevado a cabo varios procesos de diseño en robot omnidireccionales. A través de las iteraciones se han encontrado una serie de errores que impiden a las bases realizar navegación autónoma, lo cuál era uno de los objetivos principales de dichas bases.

El presente proyecto viene a completar la posibilidad de que con este robot omnidireccional se realice navegación autónoma. Por lo tanto, se presentan los diseños adecuados eléctricamente, y en cuestiones de software para proporcionar la infraestructura básica que permita realizar navegación.

A continuación, se enumeran las principales conclusiones expuestas en este proyecto:

- Se desarrolló el software de control para las ruedas y los sensores de odometría. El software de control para las ruedas, se diseño utilizando un PID en lazo cerrado con la información de la odometría.
- Se desarrolló una infraestructura de comunicación en Python, que conecta la computadora principal y el cuerpo del robot.
- Se logró implementar correctamente el software de comunicación entre dos sensores de profundidad y la computadora principal, los cuales son un Kinect, y un Hokuyo.
- Se realizó la integración de todos los módulos, tanto de software como de hardware y se realizaron pruebas de funcionamiento para cada módulo del sistema, donde se detectaron ciertos errores en el cálculo de la odometría de las ruedas, los cuales fueron corregidos.
- Se realizó la implementación del sistema de SLAM en la computadora principal.
- Se realizaron experimentos de funcionamiento para validar el sistema de mapeo y localización simultánea.

## **6.2. Recomendaciones**

A modo de recomendaciones, se propone diseñar además de la infraestructura en Hardware y Software, los métodos mediante los cuáles se realizará la depuración de las implementaciones. En particular, la depuración de errores en las conexiones eléctricas, y protocolos de comunicación serial no son triviales, puesto que son tareas con las que generalmente no se tiene mucho contacto.

Establecer la técnica que se utilizará para la depuración resulta entonces crucial en estos aspectos, para la eficiencia del proceso. La lectura de otros proyectos similares puede brindar una idea de cómo realizar la depuración correctamente.

## APÉNDICE A

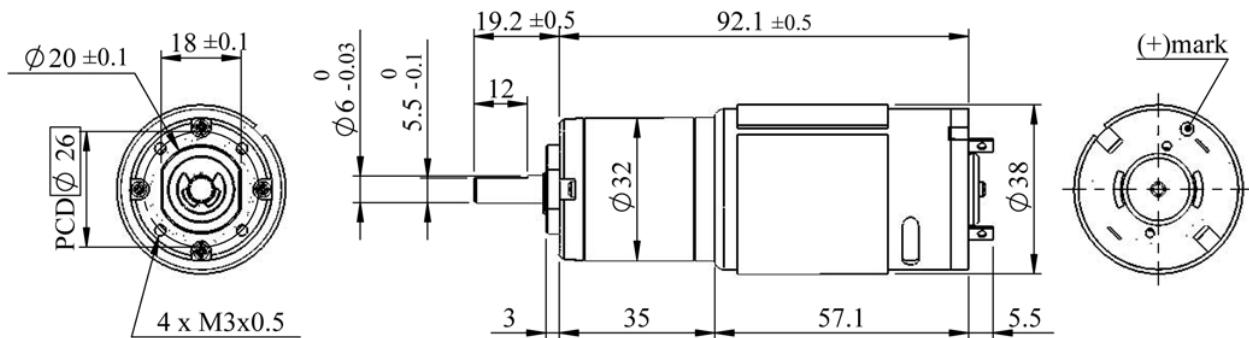
### **ACTOBOTICS 638276**

A continuación, la siguiente página muestra las especificaciones del fabricante de los motores DC Actobotics utilizados en el proyecto.

DIRECTION OF ROTATION

## SPECIFICATIONS

### DC Planetary Gear Brush Motor



(unit:mm)

**A. Operating Conditions:**

1	Operating Voltage Range	6~12	VDC	4	Operating Temperature	-10~+60	°C
2	Rated Voltage	12	VDC	5	Storage Temperature	-30~+85	°C
3	Rated Load	10	kgf-cm	6	Test Position	Horizontal	~

**B. Electrical Characteristics:**

1	Max. No-load Current	0.53	A	6	Max. Stall Current	20	A
2	No-load Speed	118±12	rpm	7	Insulation Resist.(500V)	20	MΩ
3	Rated-load Current	2	A	8	Dielectric Strength	250	VAC
4	Rated-load Speed	104±10	rpm	9	Motor Brush Type	Graphite	~
5	Min. Stall Torque	69	kgf-cm	10	Output Power at Max.Eff.	11	W

**C. Mechanical Characteristics:**

1	Gear Type	Planetary	~	7	Max. Shaft Radial Load	3.5	kgf
2	Gear Ratio	1/71	~	8	Max. Shaft Runout	0.05	mm
3	Gear Material	Metal	~	9	Max. Shaft End Play	0.30	mm
4	Rated Tolerance Torque	15	kgf-cm	10	Bearing Type	Dual Ball	~
5	Moment. Tolerance Torque	30	kgf-cm	11	Net Weight	360±20	grams
6	Max. Shaft Axial Load	2.5	kgf				

## APÉNDICE B

### **STM32F4 ENCODER**

A continuación, las siguientes tres páginas representan la información contenida en el manual de usuario del stm32f411 con respecto a el funcionamiento de los encoders de cuadratura, y la configuración pertinente del lado del stm.

The OPM waveform is defined by writing the compare registers (taking into account the clock frequency and the counter prescaler).

- The  $t_{DELAY}$  is defined by the value written in the TIMx\_CCR1 register.
- The  $t_{PULSE}$  is defined by the difference between the auto-reload value and the compare value (TIMx\_ARR - TIMx\_CCR1).
- Let us say the user wants to build a waveform with a transition from '0' to '1' when a compare match occurs and a transition from '1' to '0' when the counter reaches the auto-reload value. To do this, enable PWM mode 2 by writing OC1M=111 in the TIMx\_CCMR1 register. The user can optionally enable the preload registers by writing OC1PE='1' in the TIMx\_CCMR1 register and ARPE in the TIMx\_CR1 register. In this case the compare value must be written in the TIMx\_CCR1 register, the auto-reload value in the TIMx\_ARR register, generate an update by setting the UG bit and wait for external trigger event on TI2. CC1P is written to '0' in this example.

In our example, the DIR and CMS bits in the TIMx\_CR1 register should be low.

The user only wants one pulse (Single mode), so '1' must be written in the OPM bit in the TIMx\_CR1 register to stop the counter at the next update event (when the counter rolls over from the auto-reload value back to 0). When OPM bit in the TIMx\_CR1 register is set to '0', so the Repetitive Mode is selected.

Particular case: OCx fast enable:

In One-pulse mode, the edge detection on TIx input set the CEN bit which enables the counter. Then the comparison between the counter and the compare value makes the output toggle. But several clock cycles are needed for these operations and it limits the minimum delay  $t_{DELAY}$  min we can get.

If the user wants to output a waveform with the minimum delay, the OCxFE bit in the TIMx\_CCMRx register must be set. Then OCxRef (and OCx) are forced in response to the stimulus, without taking in account the comparison. Its new level is the same as if a compare match had occurred. OCxFE acts only if the channel is configured in PWM1 or PWM2 mode.

### 12.3.16 Encoder interface mode

To select Encoder Interface mode write SMS='001' in the TIMx\_SMCR register if the counter is counting on TI2 edges only, SMS='010' if it is counting on TI1 edges only and SMS='011' if it is counting on both TI1 and TI2 edges.

Select the TI1 and TI2 polarity by programming the CC1P and CC2P bits in the TIMx\_CCER register. When needed, the user can program the input filter as well. CC1NP and CC2NP must be kept low.

The two inputs TI1 and TI2 are used to interface to an incremental encoder. Refer to [Table 48](#). The counter is clocked by each valid transition on TI1FP1 or TI2FP2 (TI1 and TI2 after input filter and polarity selection, TI1FP1=TI1 if not filtered and not inverted, TI2FP2=TI2 if not filtered and not inverted) assuming that it is enabled (CEN bit in TIMx\_CR1 register written to '1'). The sequence of transitions of the two inputs is evaluated and generates count pulses as well as the direction signal. Depending on the sequence the counter counts up or down, the DIR bit in the TIMx\_CR1 register is modified by hardware accordingly. The DIR bit is calculated at each transition on any input (TI1 or TI2), whatever the counter is counting on TI1 only, TI2 only or both TI1 and TI2.

Encoder interface mode acts simply as an external clock with direction selection. This means that the counter just counts continuously between 0 and the auto-reload value in the

TIMx\_ARR register (0 to ARR or ARR down to 0 depending on the direction). So user must configure TIMx\_ARR before starting. in the same way, the capture, compare, prescaler, repetition counter, trigger output features continue to work as normal. Encoder mode and External clock mode 2 are not compatible and must not be selected together.

In this mode, the counter is modified automatically following the speed and the direction of the incremental encoder and its content, therefore, always represents the encoder's position. The count direction correspond to the rotation direction of the connected sensor.

*Table 48* summarizes the possible combinations, assuming TI1 and TI2 do not switch at the same time.

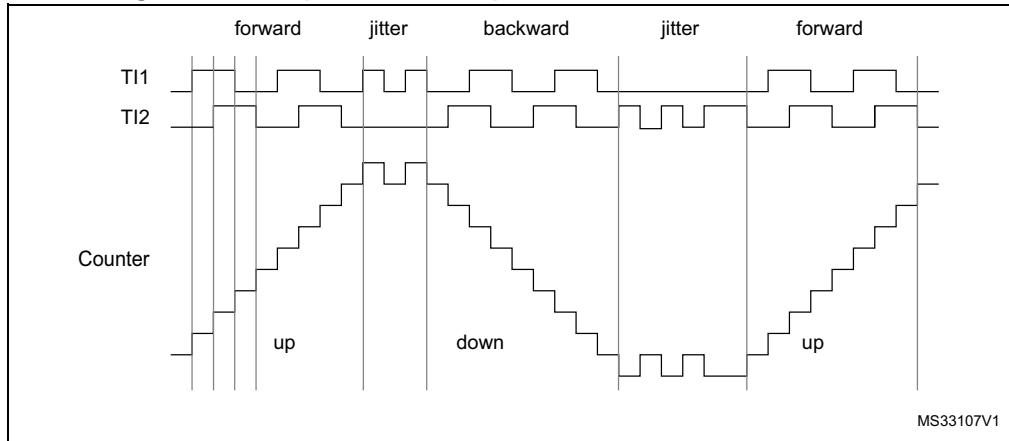
**Table 48. Counting direction versus encoder signals**

Active edge	Level on opposite signal (TI1FP1 for TI2, TI2FP2 for TI1)	TI1FP1 signal		TI2FP2 signal	
		Rising	Falling	Rising	Falling
Counting on TI1 only	High	Down	Up	No Count	No Count
	Low	Up	Down	No Count	No Count
Counting on TI2 only	High	No Count	No Count	Up	Down
	Low	No Count	No Count	Down	Up
Counting on TI1 and TI2	High	Down	Up	Up	Down
	Low	Up	Down	Down	Up

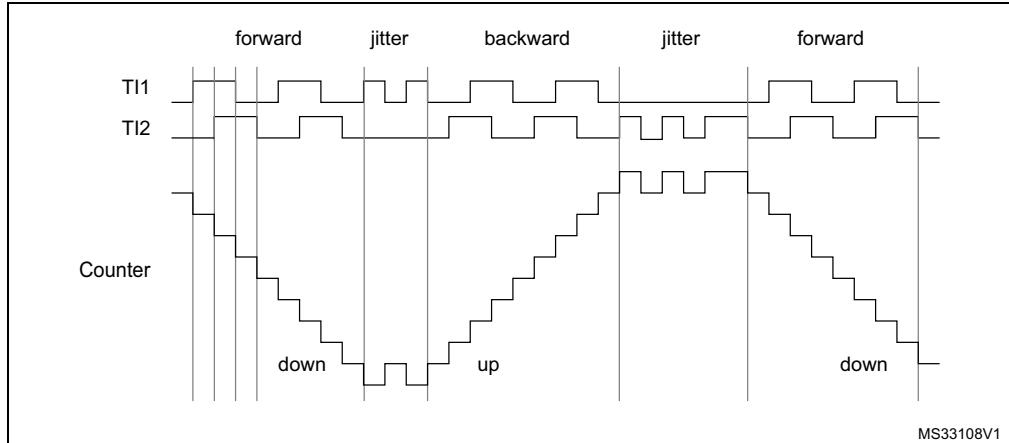
An external incremental encoder can be connected directly to the MCU without external interface logic. However, comparators are normally be used to convert the encoder's differential outputs to digital signals. This greatly increases noise immunity. The third encoder output which indicate the mechanical zero position, may be connected to an external interrupt input and trigger a counter reset.

*Figure 80* gives an example of counter operation, showing count signal generation and direction control. It also shows how input jitter is compensated where both edges are selected. This might occur if the sensor is positioned near to one of the switching points. For this example we assume that the configuration is the following:

- CC1S='01' (TIMx\_CCMR1 register, TI1FP1 mapped on TI1).
- CC2S='01' (TIMx\_CCMR2 register, TI1FP2 mapped on TI2).
- CC1P='0', CC1NP='0', and IC1F = '0000' (TIMx\_CCER register, TI1FP1 non-inverted, TI1FP1=TI1).
- CC2P='0', CC2NP='0', and IC2F = '0000' (TIMx\_CCER register, TI1FP2 non-inverted, TI1FP2= TI2).
- SMS='011' (TIMx\_SMCR register, both inputs are active on both rising and falling edges).
- CEN='1' (TIMx\_CR1 register, Counter enabled).

**Figure 80. Example of counter operation in encoder interface mode.**

[Figure 81](#) gives an example of counter behavior when TI1FP1 polarity is inverted (same configuration as above except CC1P='1').

**Figure 81. Example of encoder interface mode with TI1FP1 polarity inverted.**

The timer, when configured in Encoder Interface mode provides information on the sensor's current position. The user can obtain dynamic information (speed, acceleration, deceleration) by measuring the period between two encoder events using a second timer configured in capture mode. The output of the encoder which indicates the mechanical zero can be used for this purpose. Depending on the time between two events, the counter can also be read at regular times. This can be done by latching the counter value into a third input capture register if available (then the capture signal must be periodic and can be generated by another timer). When available, it is also possible to read its value through a DMA request generated by a real-time clock.

## APÉNDICE C

### **ROBOCLAW INSTRUCCIONES**

A continuación se muestran tres páginas con información relevante para la escritura de una librería de comunicación hacia el controlador de motores Roboclaw.

**Commands 0 - 7 Compatibility Commands**

The following commands are used in packet serial mode. The command syntax is the same for commands 0 thru 7:

Send: *Address, Command, ByteValue, CRC16*

Receive: [0xFF]

Command	Description
0	Drive Forward Motor 1
1	Drive Backwards Motor 1
2	Set Main Voltage Minimum
3	Set Main Voltage Maximum
4	Drive Forward Motor 2
5	Drive Backwards Motor 2
6	Drive Motor 1 (7 Bit)
7	Drive Motor 2 (7 Bit)
8	Drive Forward Mixed Mode
9	Drive Backwards Mixed Mode
10	Turn Right Mixed Mode
11	Turn Left Mixed Mode
12	Drive Forward or Backward (7 bit)
13	Turn Left or Right (7 Bit)

**0 - Drive Forward M1**

Drive motor 1 forward. Valid data range is 0 - 127. A value of 127 = full speed forward, 64 = about half speed forward and 0 = full stop.

Send: [Address, 0, Value, CRC(2 bytes)]

Receive: [0xFF]

**1 - Drive Backwards M1**

Drive motor 1 backwards. Valid data range is 0 - 127. A value of 127 full speed backwards, 64 = about half speed backward and 0 = full stop.

Send: [Address, 1, Value, CRC(2 bytes)]

Receive: [0xFF]

**2 - Set Minimum Main Voltage (Command 57 Preferred)**

Sets main battery (B- / B+) minimum voltage level. If the battery voltages drops below the set voltage level RoboClaw will stop driving the motors. The voltage is set in .2 volt increments. A value of 0 sets the minimum value allowed which is 6V. The valid data range is 0 - 140 (6V - 34V). The formula for calculating the voltage is: (Desired Volts - 6) x 5 = Value. Examples of valid values are 6V = 0, 8V = 10 and 11V = 25.

Send: [Address, 2, Value, CRC(2 bytes)]

Receive: [0xFF]

**3 - Set Maximum Main Voltage (Command 57 Preferred)**

Sets main battery (B- / B+) maximum voltage level. The valid data range is 30 - 175 (6V - 34V). During regenerative breaking a back voltage is applied to charge the battery. When using a power supply, by setting the maximum voltage level, RoboClaw will, before exceeding it, go into hard braking mode until the voltage drops below the maximum value set. This will prevent overvoltage conditions when using power supplies. The formula for calculating the voltage is: Desired Volts x 5.12 = Value. Examples of valid values are 12V = 62, 16V = 82 and 24V = 123.

Send: [Address, 3, Value, CRC(2 bytes)]

Receive: [0xFF]

**4 - Drive Forward M2**

Drive motor 2 forward. Valid data range is 0 - 127. A value of 127 full speed forward, 64 = about half speed forward and 0 = full stop.

Send: [Address, 4, Value, CRC(2 bytes)]

Receive: [0xFF]

**5 - Drive Backwards M2**

Drive motor 2 backwards. Valid data range is 0 - 127. A value of 127 full speed backwards, 64 = about half speed backward and 0 = full stop.

Send: [Address, 5, Value, CRC(2 bytes)]

Receive: [0xFF]

**6 - Drive M1 (7 Bit)**

Drive motor 1 forward or reverse. Valid data range is 0 - 127. A value of 0 = full speed reverse, 64 = stop and 127 = full speed forward.

Send: [Address, 6, Value, CRC(2 bytes)]

Receive: [0xFF]

**7 - Drive M2 (7 Bit)**

Drive motor 2 forward or reverse. Valid data range is 0 - 127. A value of 0 = full speed reverse, 64 = stop and 127 = full speed forward.

Send: [Address, 7, Value, CRC(2 bytes)]

Receive: [0xFF]

Command	Description
104	Read CTRLs
133	Set M1 Maximum Current
134	Set M2 Maximum Current
135	Read M1 Maximum Current
136	Read M2 Maximum Current
148	Set PWM Mode
149	Read PWM Mode

**21 - Read Firmware Version**

Read RoboClaw firmware version. Returns up to 48 bytes(depending on the Roboclaw model) and is terminated by a line feed character and a null character.

Send: [Address, 21]  
 Receive: ["RoboClaw 10.2A v4.1.11",10,0, CRC(2 bytes)]

The command will return up to 48 bytes. The return string includes the product name and firmware version. The return string is terminated with a line feed (10) and null (0) character.

**24 - Read Main Battery Voltage Level**

Read the main battery voltage level connected to B+ and B- terminals. The voltage is returned in 10ths of a volt(eg 300 = 30v).

Send: [Address, 24]  
 Receive: [Value(2 bytes), CRC(2 bytes)]

**25 - Read Logic Battery Voltage Level**

Read a logic battery voltage level connected to LB+ and LB- terminals. The voltage is returned in 10ths of a volt(eg 50 = 5v).

Send: [Address, 25]  
 Receive: [Value.Byte1, Value.Byte0, CRC(2 bytes)]

**26 - Set Minimum Logic Voltage Level**

**Note: This command is included for backwards compatibility. We recommend you use command 58 instead.**

Sets logic input (LB- / LB+) minimum voltage level. RoboClaw will shut down with an error if the voltage is below this level. The voltage is set in .2 volt increments. A value of 0 sets the minimum value allowed which is 6V. The valid data range is 0 - 140 (6V - 34V). The formula for calculating the voltage is: (Desired Volts - 6) x 5 = Value. Examples of valid values are 6V = 0, 8V = 10 and 11V = 25.

Send: [Address, 26, Value, CRC(2 bytes)]  
 Receive: [0xFF]

## BIBLIOGRAFÍA

- [1] Pieter Abbeel. gMapping. *Transactions in Robotics*, 2006.
- [2] J. A. Batlle and A. Barjau. Holonomy in mobile robots. *Robotics and Autonomous Systems*, 57(4):433–440, 2009.
- [3] Maren Bennewitz Daniel Maier, Armin Hornung. Real-Time Navigation in 3D Environments Based on Depth Camera Data. *IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*, 12th, 2012.
- [4] David Lu. global\_planer a path library and node. [http://wiki.ros.org/global\\_planer](http://wiki.ros.org/global_planer), 2012.
- [5] Foote, Tully. TF transform tree. <http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF>, 2009.
- [6] Gerkey, Brian. Amcl localization system. <http://wiki.ros.org/amcl>, 2009.
- [7] Stan Giblisco and Dr. Simon Monk. *Teach Yourself Electricity and Electronics*. Professional, McGraw-Hill, 6th edition, 2016.
- [8] Dr. Farid Golnaraghi, Dr. Benjamin C., and C. Kuo. *Automatic Control Systems*. McGraw-Hill Profesional, 10th edition, 2017.
- [9] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved techniques for grid mapping with Rao-Blackwellized particle filters. *IEEE Transactions on Robotics*, 2007.
- [10] Roseli A F Romero (Institute of Mathematical Helio Azevedo (Renato Archer IT Center and ICM/USP), José Pedro R. (Institute of Mathematical and Computer Sciences) and Computer Sciences). Cognitive and Robotic Systems: Speeding up Integration and Results. *IEEE Robotics & Automation Magazine*, 2017.
- [11] Kevin Lai, Liefeng Bo, Xiaofeng Ren, and Dieter Fox. A large-scale hierarchical multi-view RGB-D object dataset. In *Proceedings - IEEE International Conference on Robotics and Automation*, 2011.
- [12] Roberto Manduchi, Andres Castano, Ashit Talukder, and Larry Matthies. Obstacle detection and terrain classification for autonomous off-road navigation. *Autonomous Robots*, 2005.

- [13] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Mg. ROS: an open-source Robot Operating System. *Icra*, 2009.
- [14] Rodrigo Longhi Guimaraes, Andre Schneider de Oliveira, Joao Fabro. ROS Navigation: concepts and tutorial. *Federal University of Technology*, 2016.
- [15] Radu Bogdan Rusu and S. Cousins. 3D is here: point cloud library. *IEEE International Conference on Robotics and Automation*, 2011.
- [16] Hamid Taheri, Bing Qiao, and Nurallah Ghaeminezhad. Kinematic Model of a Four Mecanum Wheeled Mobile Robot. *International Journal of Computer Applications*, 2015.
- [17] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. Probabilistic robotics (intelligent robotics and autonomous agents series). *Intelligent robotics and autonomous agents, The MIT* ..., 45:52, 2005.
- [18] Thiemo Wiedemeyer. IAI Kinect2. [https://github.com/code-iai/iai\\_kinect2](https://github.com/code-iai/iai_kinect2), 2014 – 2015. Accessed June 12, 2015.