**Universidad Tecnológica Nacional**

Proyecto Final

---

# Robot experimental basado en ROS capaz de realizar localización y mapeo simultáneos (SLAM)

---


*Autores:*

- *Moragues, Juan Ignacio*

*Director:*

- *Burgos, Enrique Sergio*


*Proyecto final presentado para cumplimentar los requisitos académicos para acceder al título de Ingeniero Electrónico*

*en la*

**Facultad Regional Paraná**

Mayo de 2024

## Declaración de autoría:

Yo declaro que el Proyecto Final “Robot experimental basado en ROS capaz de realizar localización y mapeo simultáneos” y el trabajo realizado son propios.

Declaro:

- Este trabajo fue realizado en su totalidad, o principalmente, para acceder al título de grado de Ingeniero Electrónico, en la Universidad Tecnológica Nacional, Regional Paraná.
- Se establece claramente que el desarrollo realizado y el informe que lo acompaña no han sido previamente utilizados para acceder a otro título de grado o pre-grado.
- Siempre que se ha utilizado trabajo de otros autores, él mismo ha sido correctamente citado. El resto del trabajo es de autoría propia.
- Se ha indicado y agradecido correctamente a todos aquellos que han colaborado con el presente trabajo.
- Cuando el trabajo forma parte de un trabajo de mayores dimensiones donde han participado otras personas, se ha indicado claramente el alcance del trabajo realizado.

Firma:

Fecha:

## Agradecimientos

A mi docente director, Sergio Burgos por su paciencia y motivación durante el cursado de la carrera y durante el desarrollo del presente proyecto. A la Universidad Tecnológica Nacional, Facultad Regional Paraná, y a los profesionales docentes que la componen por la formación y enseñanzas que me impartieron a mi y a todos los alumnos que alguna vez formamos parte de la institución.

A mi novia, Alfonsina Carlen, por su apoyo incondicional en todos los proyectos que decido encarar. Es un pilar importante en mi vida desde hace mucho tiempo.

A mi familia, y en especial a mis padres, por ser quienes depositaron su confianza en mí a la hora de elegir estudiar Ingeniería. Por el apoyo tanto emocional como económico durante todo el transcurso de la carrera.

A mis compañeros y amigos que durante estos años compartimos mucho tiempo, ya sea de estudio, trabajos u ocio. Sin ellos el camino de estudiar Ingeniería Electrónica sería muy distinto.

**Universidad Tecnológica Nacional**

# *Abstract*

Facultad Regional Paraná

Ingeniero en Electrónica

# Robot experimental basado en ROS capaz de realizar localización y mapeo simultáneos (SLAM)

Moragues Juan Ignacio

## Abstract

This project consists of the design and development of an experimental robot capable of making a map of the environment in which it is located and, at the same time, locating itself. The development of the robot is based on the ROS operating system and existing algorithms within it. To achieve mapping and localization, the robot has different sensors:

- Lidar
- Encoder
- Inertial Sensors

By feeding the SLAM algorithm existing in the ROS operating system with the information provided by the sensors, the robot is capable of making a map of the environment in which it is located and locating itself in it.

For the robot hardware we use a Raspberry Pi where the operating system runs, a development board based on STM32 to control the motors, direct current motors powered by an H bridge, and battery protection and charging circuits.

After designing the functional robot and carrying out a series of tests and corrections, the developed robot meets the specifications initially proposed and lays the foundations for the development of more complex robots who need to solve the problem that SLAM poses.

## Keywords:

Robot, Robot Operating System, SLAM, LIDAR, STM32, Raspberry PI

## **Resumen**:

El presente proyecto consiste en el diseño y desarrollo de un robot experimental capaz de realizar un mapa del entorno en el que se encuentra y, al mismo tiempo, localizarse dentro de él. El desarrollo del robot se basa en el sistema operativo ROS (Robot Operating System) y sobre algoritmos existentes dentro del sistema. Para lograr el mapeo y localización el robot posee diferentes sensores:

- LIDAR
- Encoders
- Sensores Inerciales

Alimentando el algoritmo de SLAM existente en el sistema operativo ROS con la información proveída por los sensores, el robot es capaz de realizar un mapa del entorno en el que se encuentra y localizarse en él.

Para el hardware del robot utilizamos un Raspberry Pi donde se ejecuta el sistema operativo, una placa de desarrollo basada en STM32 para el control de los motores, motores de corriente continua alimentados mediante un puente H y circuitos de protección y carga de baterías.

Luego de realizar un diseño del robot funcional y de llevar a cabo una sucesión de pruebas y correcciones, el robot desarrollado cumple con las especificaciones planteadas en un principio y que sienta las bases para el desarrollo de robots más complejos, aplicados a diferentes campos de la ingeniería, que necesiten resolver el problema que el SLAM plantea.



## **Palabras Clave:**

Robot, Robot Operating System, SLAM, LIDAR, STM32, Raspberry PI

# Índice

# Lista de figuras

# Lista de tablas

# Lista de abreviaciones y símbolos

**ROS** Robot Operating System

**SLAM** Simultaneos Localization and Mapping

**PKG** Package

**TOF** Time of Flight

**IMU** Intertial Measurement Unit

**GNSS** Global Navigation Satellite System

**GPS** Global Positioning System

**LIDAR** Light Detection and Ranging

**PID** Proporcional, Integral y Derivativo

**PWM** Pulse Width Modulation

**UART** Universal Asynchronous Receiver-Transmitter

**POSE** Position Estimation

**MSG** Message

**TF** Transform

**URDF** Unified Robot Description Format

**DIFF DRIVE** Differential Drive Robot

**ODOM** Odometría

## Dedicado a

Mi novia Alfonsina Carlen por su paciencia y motivación para llegar a este logro. A mis padres, Silvana y Juan por su apoyo incondicional en esta y otras etapas de mi vida.

# Capítulo 1: Introducción.

En la actualidad existen vehículos autónomos aplicados a diferentes industrias, por ejemplo, robots de logística utilizados en almacenes o depósitos, robots camareros presentes en restaurantes u hoteles, robots aspiradores totalmente autónomos, hasta podemos encontrar robots autónomos de rescate utilizados en situaciones de riesgo, accidentes o desastres naturales.

Ante la creciente demanda de estos vehículos o robots autónomos, el presente proyecto plantea el desarrollo de un robot capaz de realizar mapeo y localización simultáneos (SLAM por sus siglas en inglés) y que, a su vez, sirva como base para el desarrollo de robots más complejos que necesiten tener resuelto el problema del SLAM.

Para el desarrollo del robot decidimos utilizar ROS (Robot Operating System). Este es un framework de desarrollo para aplicaciones robóticas. Provee un conjunto de bibliotecas de software, capas de abstracción de hardware, drivers, manejo de paquetes y herramientas que ayudan al diseño y creación de robots. ROS se encuentra bajo una licencia de código abierto, por lo que podemos hacer uso de todas sus funcionalidades libremente.

Actualmente, ROS se considera muy confiable dentro de la industria. Es la norma para enseñar robótica, desde proyectos estudiantiles, colaboraciones de varias instituciones hasta competencias a gran escala. Se encuentra dentro de robots comerciales que están en producción alrededor del mundo. En el mercado de vehículos autónomos, ROS ha ayudado a crear miles de millones de dólares en valor.

## 1.1 ¿Qué es ROS?

Robot Operating System (ROS) es una colección de frameworks para el desarrollo de software de robots. ROS se desarrolló originariamente en 2007 por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto de un Robot con Inteligencia Artificial.
A pesar de no ser un sistema operativo, ROS provee los servicios estándar de uno de estos tales como la abstracción del hardware, el control de dispositivos de bajo nivel,

el paso de mensajes entre procesos y el mantenimiento de paquetes. Tiene dos partes básicas, ROS propiamente dicho y ROS-PKG. Esta última consiste en una suite de paquetes aportados por la contribución de usuarios.

El objetivo primario de ROS es soportar el reuso de código en la investigación y desarrollo dentro de la robótica. ROS es una estructura distribuida de procesos llamados Nodos que permite el diseño individualizado, pero fácilmente acoplable a los demás procesos. Estos procesos se pueden agrupar en Paquetes y Pilas, que fácilmente pueden ser intercambiados, compartidos y distribuidos.

## 1.2 Mapeo y localización simultáneos (SLAM).

La localización y mapeo simultáneos es una técnica usada por robots y vehículos autónomos para construir un mapa de un entorno desconocido. Al mismo tiempo que construye el mapa, estima su trayectoria al desplazarse dentro del entorno. Dicho de otra manera, el SLAM busca resolver los problemas que plantea colocar un robot móvil en un entorno y posición desconocidos, y que él mismo sea capaz de construir incrementalmente un mapa consistente del entorno al tiempo que utiliza dicho mapa para determinar su propia localización.

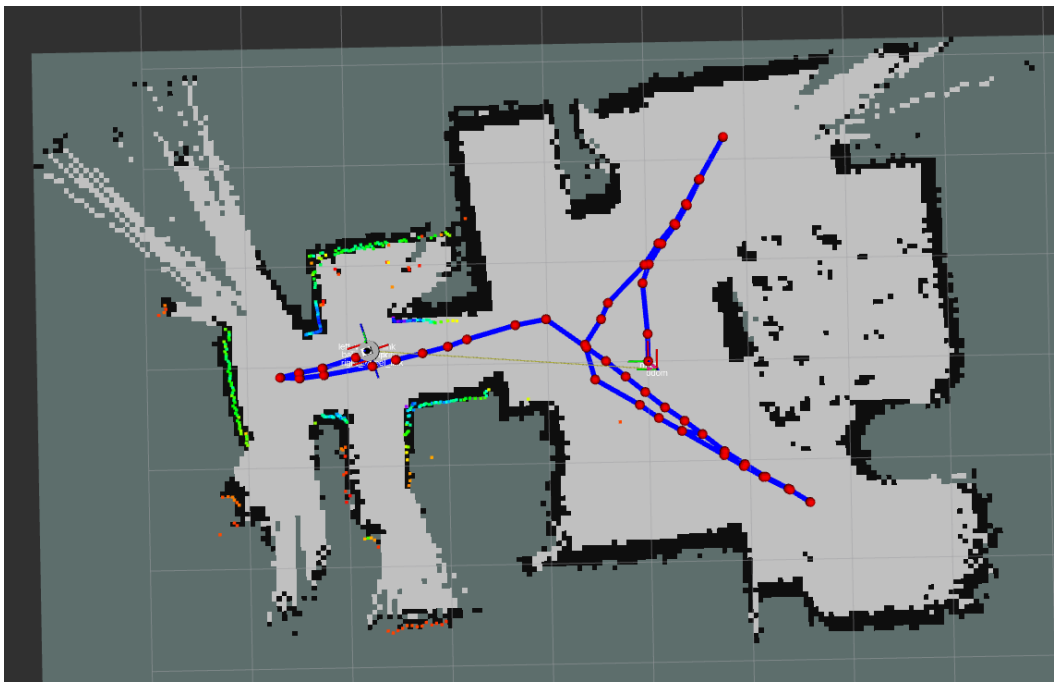

Fig. 1 - Ejemplo SLAM

Considere un robot aspirador. Sin SLAM, simplemente se moverá aleatoriamente dentro de una habitación y es posible que no pueda limpiar toda la superficie del piso.

Además, este método consume demasiada energía, por lo que la batería se agotará más rápidamente. Por otro lado, los robots con un algoritmo SLAM pueden utilizar información como la velocidad de las ruedas, datos de cámaras y otros sensores para determinar la cantidad de movimiento necesario. Esto se llama localización. El robot también puede utilizar simultáneamente la cámara y otros sensores para crear un mapa de los obstáculos de su entorno y evitar limpiar la misma zona dos veces. Esto se denomina mapeo.

Existen diferentes tipos de SLAM como ser, visual SLAM, lidar SLAM y multi sensor SLAM. En el presente proyecto nos enfocaremos en SLAM utilizando lidar. En comparación con las cámaras, ToF y otros sensores, los láseres son significativamente más precisos y se utilizan para aplicaciones con vehículos en movimiento a alta velocidad, como automóviles autónomos y drones. Los valores de salida de los sensores láser son generalmente datos de nubes de puntos 2D (x, y) o 3D (x, y, z). La nube de puntos del sensor láser proporciona mediciones de distancia de alta precisión y funciona eficazmente para la construcción de mapas con algoritmos SLAM. El movimiento se estima de forma secuencial registrando las nubes de puntos. El movimiento calculado (distancia recorrida) se utiliza para localizar el vehículo dentro del mapa. Los mapas de nubes de puntos 2D o 3D se pueden representar como mapas de cuadrícula o mapas de vóxeles.

Para un mejor funcionamiento, la localización de vehículos autónomos puede implicar la fusión de otras mediciones, como la odometría de las ruedas, el sistema global de navegación por satélite (GNSS) y datos de sensores inerciales (IMU).

## 1.3 Navegación de un entorno conocido.

Una vez que el robot conoce el entorno en el que se encuentra, podemos empezar a hablar de navegación autónoma. La autonomía de un robot móvil se basa principalmente en el sistema de navegación autónoma. En estos sistemas se incluyen diversas tareas como: planificación, percepción y control. La planificación en los robots móviles, consiste generalmente en establecer la misión, la ruta y la evasión de obstáculos.

El algoritmo de navegación nos pide como requisito el mapa donde se moverá el robot y el punto de comienzo. La posición del robot al comienzo de la navegación es muy

importante, no es necesario que sea extremadamente precisa, pero sí aproximada. Sin este punto o posición de comienzo, será muy difícil para el algoritmo de navegación conocer la localización. Y sin una posición aproximada de comienzo, no será posible establecer una ruta precisa.

## 1.4 Selección de la topología del Robot.

Existen muchos tipos de robots, sin ruedas, con diferente cantidad de ruedas, articulados, con dirección, brazos robóticos, etc. En el presente proyecto utilizaremos un tipo de robot denominado robot de accionamiento diferencial o “Diffdrive Robot”.
Un robot con ruedas diferenciales es un robot móvil cuyo movimiento se basa en dos ruedas accionadas por separado colocadas a cada lado del cuerpo del robot. Puede cambiar su dirección variando la velocidad relativa de rotación de sus ruedas y, por tanto, no requiere un movimiento de dirección adicional.

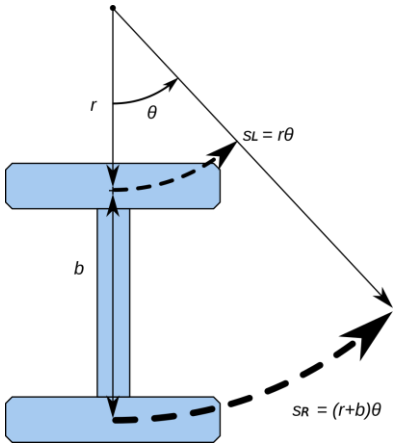


Fig. 2 - Diffdrive robot.

Si ambas ruedas se mueven en la misma dirección y velocidad, el robot irá en línea recta. Pero, si giran con igual velocidad en direcciones opuestas, como se desprende claramente del diagrama mostrado, el robot girará alrededor del punto central del eje. De lo contrario, dependiendo de la velocidad de rotación y su dirección, el centro de rotación puede caer en cualquier punto de la línea definida por los dos puntos de contacto de las ruedas. Mientras el robot viaja en línea recta, el centro de rotación está a una distancia infinita del robot. Dado que la dirección del robot depende de la velocidad y dirección de rotación de las dos ruedas motrices, estas cantidades deben detectarse y controlarse con precisión.

Los robots con ruedas diferenciales se utilizan ampliamente en robótica, ya que su movimiento es fácil de programar y puede controlarse bien. Prácticamente todos los robots de consumo del mercado actual utilizan dirección diferencial principalmente por su bajo coste y simplicidad.

# Capítulo 2: Desarrollo

El desarrollo del robot se puede dividir en 4 etapas, selección de los módulos de hardware, diseño del chasis donde se montarán los diferentes componentes del hardware, desarrollo del firmware del controlador de motores y, por último, desarrollo del software que se ejecutará sobre ROS.

## 2.1 Selección de módulos de hardware.

Por simplicidad y rapidez para el desarrollo del robot, se optó por el uso de módulos de hardware estándar que cumplen diferentes funciones. A continuación, presentamos un diagrama de bloques donde se pueden ver los diferentes módulos utilizados.

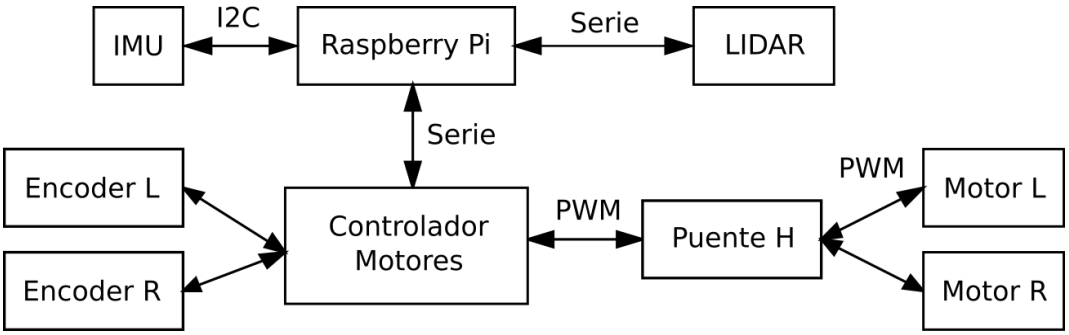


Fig. 3 - Diagrama de bloques de Hardware.

Para utilizar el sistema operativo ROS primeramente debemos seleccionar un sistema embebido capaz de ejecutar un Sistema Operativo linux. ROS como tal no es un sistema operativo, sino que es una colección de frameworks para el desarrollo de software orientado a robots. La versión de ROS a utilizar es la denominada “ROS 2 - Humble”, dicha versión solicita que se use específicamente la distribución de linux “Ubuntu 22.04”. Por estas razones seleccionamos como hardware principal una Raspberry Pi 4 - Model B de 4GB de RAM.

Para el LIDAR utilizamos el denominado LDS-01. Creemos que dicho LIDAR posee las características mínimas necesarias para cumplir con la tarea de SLAM. Las mismas son las siguientes.

- Rango de distancia: 120 ~ 3,500mm
- Exactitud de distancia de 120 a 499 mm: ±15mm
- Exactitud de distancia de 500 a 3,500mm: ±5.0%
- Precisión de distancia de 120mm a 499mm: ±10mm
- Precisión de distancia de 500mm a 3,500mm: ±3.5%
- Tasa de escaneo: 300±10 rpm
- Rango angular: 360°
- Resolución Angular 1°

Fig. 4 - LIDAR

El mapeo que realiza el robot se basa enteramente en las mediciones provistas por el LIDAR, por lo que este es un componente fundamental dentro del sistema.

Como IMU (Inertial Measurement Unit de sus siglas en inglés) seleccionamos la MPU9250. En este caso, este sensor ayudará al sistema ROS a calcular los movimientos del robot.

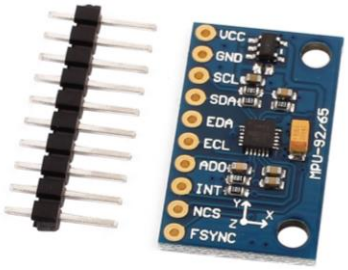


Fig. 5 - IMU MPU9250

Para controlar los motores utilizamos una placa de desarrollo de la marca ST basada en el microcontrolador STM32L432KC. Dicha placa se comunica con la Raspberry Pi mediante un puerto serie. La raspberry enviará comandos al microcontrolador con el fin de controlar la velocidad a la que se moverá el robot. Dicho microcontrolador

deberá garantizar que los motores se muevan a la velocidad solicitada. Esto lo hará mediante la realimentación provista por los encoders.

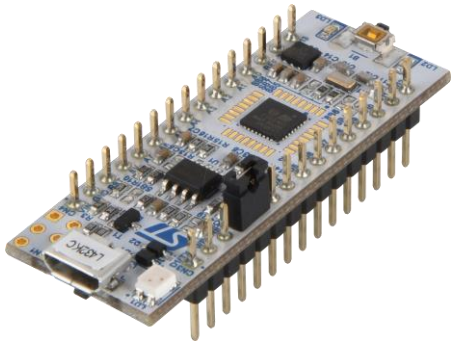

Fig. 6 - Nucleo STM32

Como la potencia de los motores es elevada como para que los pines de un microcontrolador los comanden, debemos colocar un circuito de potencia. Como tal seleccionamos un módulo basado en el circuito integrado L298n. Este circuito integrado posee en su interior dos circuitos puente H, lo que nos permite comandar dos motores por separado.

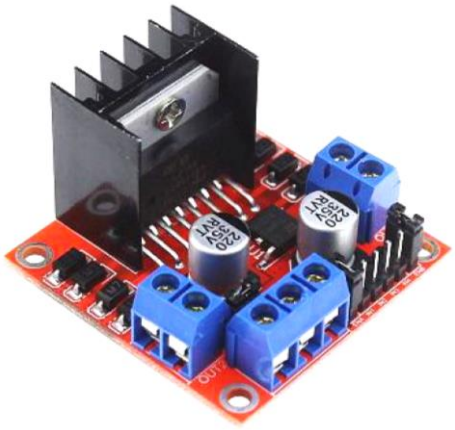

Fig. 7 - Puente H

Seleccionamos dos motores modelo FIT0522 con encoders de cuadratura incorporados. Los encoders de cuadratura, además de sensar la velocidad a la que se encuentra girando el motor, nos permite diferenciar el sentido de giro. Estos motores se alimentan con una tensión de 6[v] y tienen una potencia de aproximadamente 2.8[w]. Además, poseen una caja reductora, lo que les permite ejercer un mayor torque en las ruedas.

Fig. 8 - Motor FIT0522

Por último, para alimentar el robot utilizamos 4 baterías 18650 en conjunto con 2 fuentes reguladas. Tenemos 2 circuitos de alimentación independientes, 2 baterías en serie alimentan una fuente, y las otras 2, también en serie, alimentan la fuente restante. Utilizamos esta configuración debido a que los motores se alimentan con una tensión de 6[v], y la Raspberry Pi y el Lidar con 5[v].

Para cargar las baterías utilizamos dos módulos de carga TP5100, estos módulos pueden cargar una batería o 2 baterías en serie.


Fig. 9 - Módulo cargador TP5100

## 2.2 Diseño del chasis.

Para montar los diferentes componentes de hardware necesitaremos diseñar un chasis. Para ello utilizamos el software FreeCAD. A continuación, mostraremos una imagen del robot completo, luego iremos detallando cada parte.

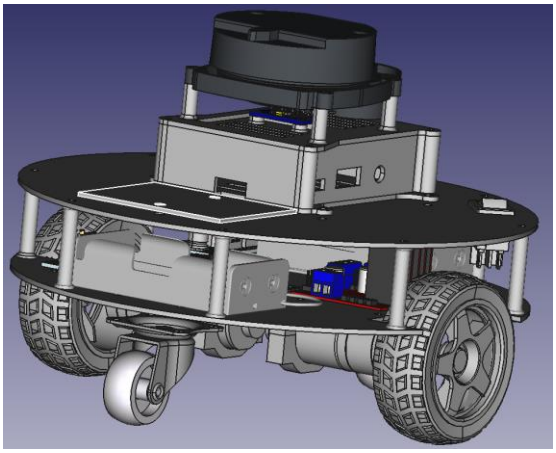

Fig. 10 - Diseño robot completo.

Primero necesitamos una carcasa para la Raspberry Pi, sobre la misma montaremos el Lidar y la IMU.

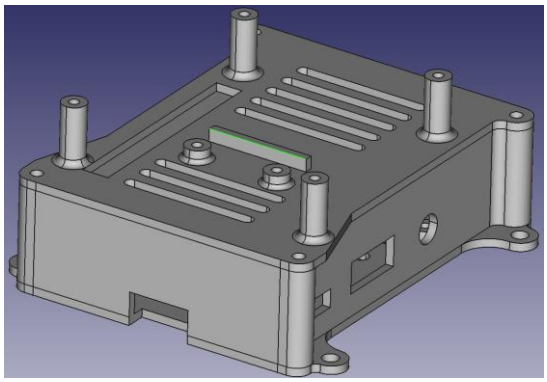

Fig. 11 - Carcasa Raspberry Pi

Para montar todos los componentes diseñamos una base. Sobre esta base montaremos las baterías, los módulos de carga, el módulo puente H, las fuentes, los motores y las dos ruedas de apoyo.

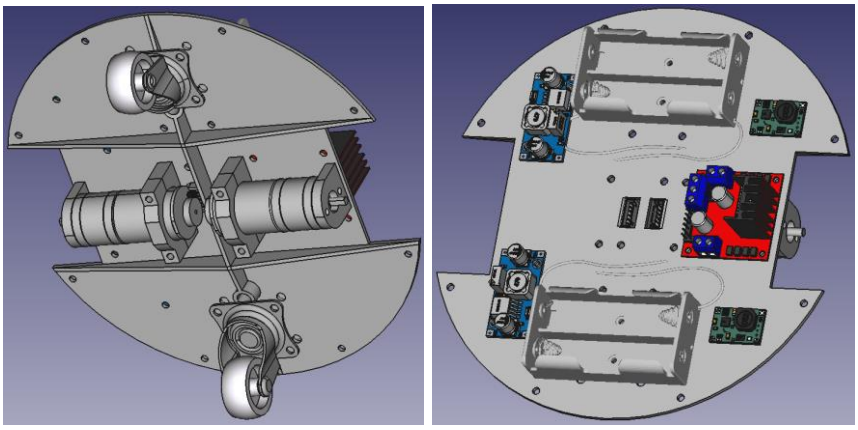

Fig. 12 - Base del Robot

Sobre esta base colocaremos una tapa en la cual podremos montar la carcasa de la Raspberry Pi, el controlador de motores y una llave de encendido.

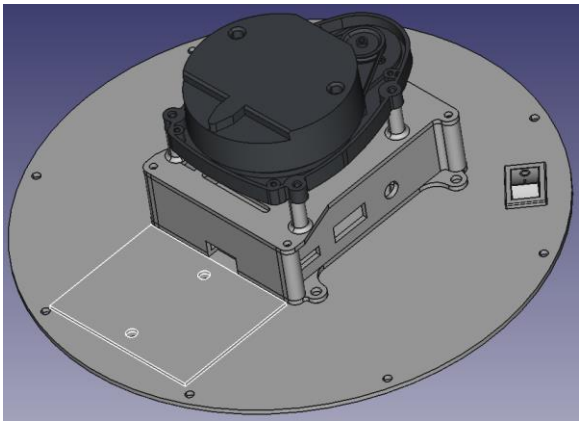

Fig. 13 - Tapa del Robot

Con respecto a las ruedas, la idea original era utilizar unas ruedas adquiridas en un kit de robot. Cuando comenzamos con las primeras pruebas vimos que dichas ruedas no tenían una buena adherencia al suelo. Por este motivo decidimos diseñar unas ruedas similares y añadirles “goma eva” y caucho para lograr una mejor adherencia.

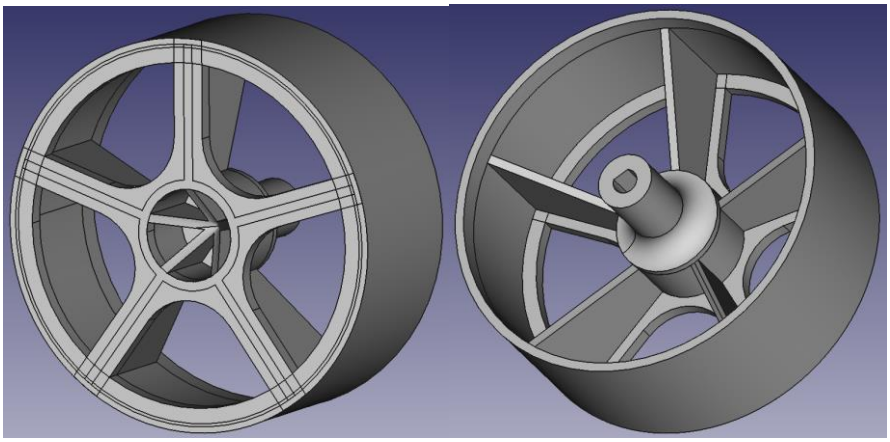

Fig. 14 - Ruedas

## 2.3 Desarrollo firmware controlador de motores.

Para mejorar la respuesta de los motores implementamos un controlador PID como controlador de motores.

Un controlador PID o controlador proporcional, integral y derivativo, es un mecanismo de control que a través de un lazo de retroalimentación permite regular la velocidad,

temperatura, presión y flujo entre otras variables de un proceso en general. El controlador PID calcula la diferencia entre la variable real y la variable deseada.

El algoritmo del control PID consta de tres parámetros distintos: el proporcional, el integral, y el derivativo. El valor proporcional depende del error actual, el integral depende de los errores pasados y el derivativo es una predicción de los errores futuros. La suma de estas tres acciones es usada para ajustar el proceso por medio de un elemento de control, como la posición de una válvula de control o la potencia suministrada a un calentador.

El error es la desviación existente entre el valor de la variable de proceso y el valor consigna, o "Set Point". Se expresa matemáticamente como una función en el tiempo:

$$e(t) = SP - PV(t)$$

donde SP es el set point y PV(t) es la variable de proceso.

### 2.3.1 Entradas y Salidas.

Para desarrollar el firmware utilizamos el STM32 CubeIDE. Dicha interfaz de desarrollo es la provista por STM para programar sus productos. Comenzaremos mostrando las entradas y salidas necesarias.

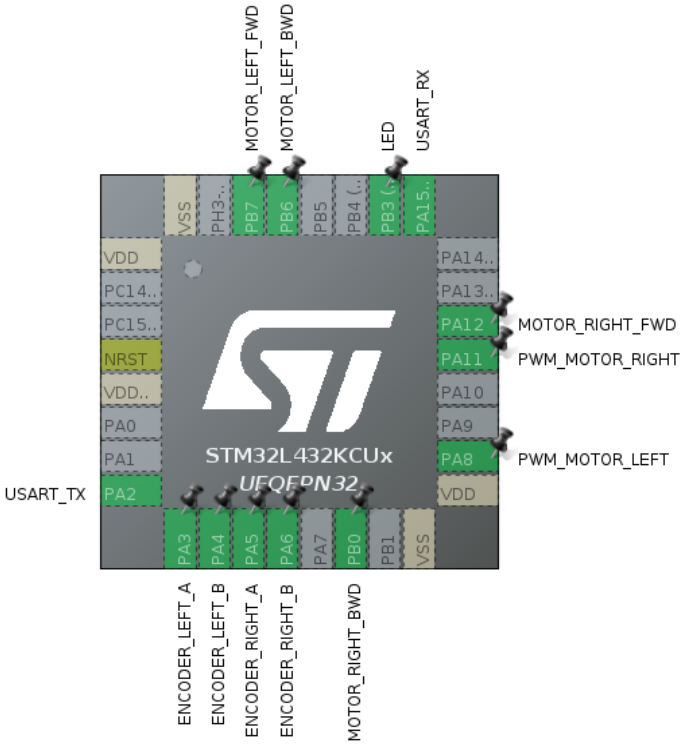


Fig. 15 - Entradas/Salidas STM32L4

El control de cada motor requiere de una señal de PWM y dos señales digitales (pines de entrada salida de propósito general) para ser controlados. Estos indicarán el sentido de giro mientras que la señal PWM establecerá la velocidad. Además, a fin de tener realimentación del giro de cada motor, cada encoder requerirá dos entradas digitales por tratarse de encoders de cuadratura. Y, por último, necesitamos una UART para poder comunicarnos con la Raspberry Pi. Resumiendo:

- PA2 - USART Tx.
- PA3 - Encoder izquierdo A,
- PA4 - Encoder izquierdo B.
- PA5 - Encoder derecho A.
- PA6 - Encoder derecho B.
- PA8 - PWM Motor izquierdo.
- PA11 - PWM Motor derecho.
- PA12 - Motor derecho hacia adelante.
- PA15 - USART Rx.
- PB0 - Motor derecho hacia atrás.
- PB6 - Motor izquierdo hacia atrás.
- PB7 - Motor izquierdo hacia adelante.

### 2.3.2 Lógica encoders de cuadratura.

Un encoder de cuadratura, también conocido como encoder rotatorio incremental, mide la velocidad y la dirección de un eje giratorio. Los encoders de cuadratura pueden utilizar diferentes tipos de sensores, tanto los ópticos como de efecto Hall son comúnmente utilizados. No importa qué tipo de sensores se utilicen, la salida suele ser dos formas de onda cuadradas desfasadas 90°, como se muestra a continuación.

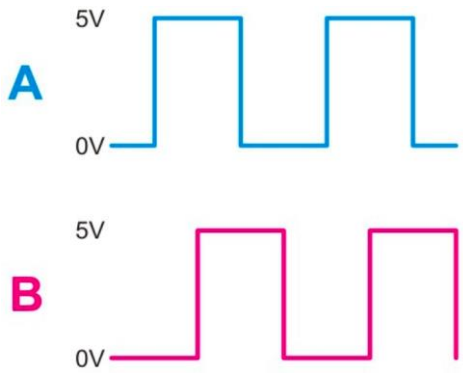


Fig. 16 - Señales encoders de cuadratura

Si solo desea monitorear la velocidad de rotación, puede usar cualquiera de las salidas y simplemente medir la frecuencia. La razón para tener dos salidas es que también se puede determinar la dirección de rotación del eje observando el patrón de números binarios generados por las dos salidas.

Dependiendo del sentido de rotación, obtenemos:

- 00 = 0
- 01 = 1
- 11 = 3
- 10 = 2

O, en el otro sentido:

- 00 = 0
- 10 = 2
- 11 = 3
- 01 = 1

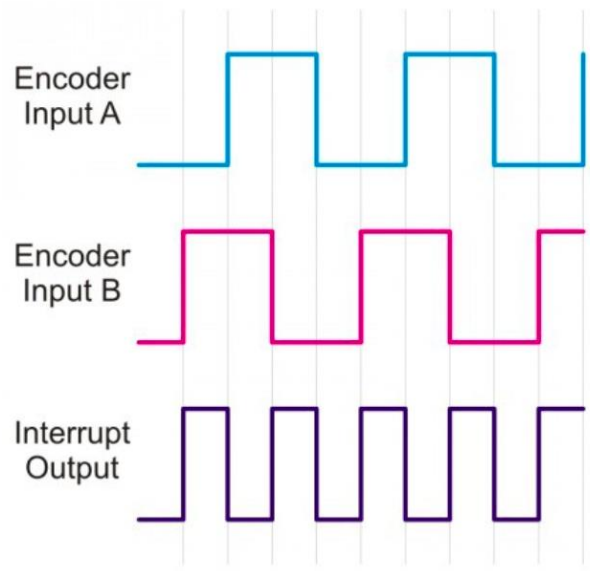


Fig. 17 - Señal de interrupción

Los valores binarios anteriores se convierten a 0,1,3,2 o 0,2,3,1 según la dirección. Este patrón se repite continuamente. Al utilizar el valor actual del codificador para indexar una dimensión de una matriz y el valor anterior para indexar la otra dimensión, se puede obtener rápidamente una salida -1, 0 o +1. La matriz quedaría de la siguiente forma.

| | | Valor Actual | | | |
|---|---|---|---|---|---|
| | | 00 | 01 | 10 | 11 |
| Valor Anterior | 00 | 0 | -1 | +1 | X |
| | 01 | +1 | 0 | X | -1 |
| | 10 | -1 | X | 0 | +1 |
| | 11 | X | +1 | -1 | 0 |

Tabla 1 - Matriz encoders cuadratura.

Como se puede ver, si el valor no cambia, la salida es 0, si la secuencia es 0, 1, 3, 2 tenemos una salida de -1. Y, por último, si la secuencia es 0, 2, 3, 1 tenemos una salida de +1. X representa un estado no permitido y lo más probable es que ocurra si las salidas del encoder cambian demasiado rápido para que el firmware pueda leerlas correctamente.

Desarrollamos una librería basada en handlers, cada handler corresponde a un encoder y guarda el estado del mismo cada vez que hay un cambio.

```
typedef struct{
    GPIO_TypeDef *portA;          /* Puerto correspondiente a la entrada A
*/
    uint16_t pinA;                /* Pin correspondiente a la entrada A */

    GPIO_TypeDef *portB;          /* Puerto correspondiente a la entrada B
*/
    uint16_t pinB;                /* Pin correspondiente a la entrada B */

    int8_t  phase;                /* Valor guardado de las entradas A y B
*/
    int32_t count;                /* Cuenta del encoder */

    uint8_t newInt_flag;          /* Flag de interrupción */
}ENCODER_Handle;
```

Tabla 2 - Handler encoders

Configuramos las entradas de los encoders como interrupciones en flanco de subida y flanco de bajada. De esta manera podemos disparar el procesamiento de los encoders cada vez que se produce un cambio. Para esto tenemos una función,

*ENCODER_process(),* que lee los pines del encoder cada vez que se produce una interrupción del mismo.

```
void ENCODER_process(ENCODER_Handle *enc){

	uint8_t s = enc->phase & 0x03;

	if( HAL_GPIO_ReadPin(enc->portA, enc->pinA) == GPIO_PIN_SET )
	{
		s |= 0x04;
	}
	if( HAL_GPIO_ReadPin(enc->portB, enc->pinB) == GPIO_PIN_SET )
	{
		s |= 0x08;
	}

	switch (s) {
		case 0: case 5: case 10: case 15:
			break;
		case 1: case 7: case 8: case 14:
			enc->count++;
			break;
		case 2: case 4: case 11: case 13:
			enc->count--;
			break;
		case 3: case 12:
			enc->count += 2;
			break;
		default:
			enc->count -= 2;
			break;
	}
	enc->phase = (s >> 2);

	enc->newInt_flag = 1;
}
```

Tabla 3 - Encoder process.

Dentro del *switch* analizamos el valor anterior y el actual utilizando los casos que vimos anteriormente en la matriz.

### 2.3.3 Implementación PID.

Para el caso del PID también desarrollamos una librería basada en handlers.

```
typedef struct{
	/* Controller gains */
	float Kp;
	float Ki;
	float Kd;

	/* Derivative low-pass filter time constant */
	float tau;
```

```
      /* Output Limits */
      float limMin;
      float limMax;

      /* Integrator limits */
      float limMinInt;
      float limMaxInt;

      /* Sample time in seconds */
      float T;

      /* Controller "memory" */
      float integrator;
      float prevError;
      float differentiator;
      float prevMeasurement;

      /* Controller Output */
      float out;
}PIDController_Handle;
```

Tabla 4 - PID Handler

El handler almacena las constantes proporcionales, derivativas e integrales, además de límites configurables, el tiempo de muestreo y la salida.

Como sabemos, la salida del PID es la suma del término proporcional, el derivativo y el integral.

$$G(s) = K_P + K_I \frac{1}{S} + K_D \frac{S}{S\tau + 1}$$

Implementamos la función *PIDController_Update* que toma como argumentos el handler correspondiente al PID, el setpoint y la medición de la variable y, nos da como resultado, la salida del sistema.

```
float PIDController_Update( PIDController_Handle *pid, float setpoint,
float measurement ){

      /* Error Signal */
      float error = setpoint - measurement;

      /* Proportional*/
      float proportional = pid->Kp * error;

      /* Integrator */
      pid->integrator = pid->integrator + 0.5f * pid->Ki * pid->T *
                        (error + pid->prevError);

      /* Anti wind-up via dynamic integrator clamping */
      float limMinInt, limMaxInt;

      /* Compute integrator limits */
      if (pid->limMax > proportional){
```

```
            limMaxInt = pid->limMax - proportional;
      }
      else{
            limMaxInt = 0.0f;
      }

      if (pid->limMin < proportional){
            limMinInt = pid->limMin - proportional;
      }
      else{
            limMinInt = 0.0f;
      }

      /* Clamp Integrator */
      if (pid->integrator > limMaxInt){
            pid->integrator = limMaxInt;
      }
      else if (pid->integrator < limMinInt){
            pid->integrator = limMinInt;
      }

      /* Derivative (band-limited differentiator) */
      pid->differentiator = (2.0f * pid->Kd * (measurement -
                           pid->prevMeasurement) + (2.0f * pid->tau -
                           pid->T) * pid->differentiator)
                           / (2.0f * pid->tau + pid->T);

      /* Compute output and apply limits */
      pid->out = proportional + pid->integrator + pid->differentiator;

      if (pid->out > pid->limMax){
            pid->out = pid->limMax;
      }
      else if (pid->out < pid->limMin){
            pid->out = pid->limMin;
      }

      /* Store error and measurement for later use */
      pid->prevError = error;
      pid->prevMeasurement = measurement;

      /* Return controller output*/
      return pid->out;
}
```

Tabla 5 - PID Update

En la función podemos ver cómo calculamos cada uno de los términos y luego los sumamos. También podemos ver que al final controlamos que la salida se encuentra dentro de los límites configurados.

Un detalle a tener en cuenta es que tendremos un PID por cada rueda del robot ya que debemos controlar la velocidad de cada una por separado.

### 2.3.4 Protocolo de comunicación.

Para comunicar la Raspberry Pi con nuestro controlador de motores debemos establecer un protocolo de comunicación. El puerto de comunicación que utilizaremos es UART.

Con dicho protocolo de comunicación debemos ser capaces de iniciar o detener los motores, establecer la velocidad de cada una de las ruedas, solicitar la cuenta de cada uno de los encoders o inicializarla en cero.

Los comandos establecidos son los siguientes. Todos los comandos devuelven la cuenta de los encoders, esto se debe a un tema de compatibilidad con la librería que utilizaremos en ROS para comunicarnos con el controlador.

| **Comando** | **Respuesta**[1] | **Descripción** |
|---|---|---|
| e | <enc izq> <enc der>\r\n | Devuelve la cuenta de los encoders, a la izq el correspondiente a la rueda izq y a la derecha el de la rueda derecha. |
| r | <enc izq> <enc der>\r\n | Vuelve a cero las cuentas de los encoders. |
| s | <enc izq> <enc der>\r\n | Pone a 0 el setpoint de los PID y el valor de PWM de las ruedas. De esta manera detiene los motores. |
| m <setpoint> <setpoint> | <enc izq> <enc der>\r\n | Establece el set point del PID para cada rueda. La unidad del set point es rad/s, corresponde a la velocidad angular de la rueda. El sentido de giro se lo damos con el signo. |
| o <rawpwm> <rawpwm> | <enc izq> <enc der>\r\n | Establece el valor del pwm correspondiente a cada rueda omitiendo la acción del controlador PID. El sentido de giro se indica con el signo. |

Tabla 6 - Protocolo del controlador de motores.

1. Todos los comandos tienen la misma respuesta, y se corresponde con el valor actual de los encoders izquierdo y derecho.

### 2.3.5 Respuesta PID.

A continuación, compararemos la respuesta de los motores utilizando el PID contra comandos de PWM directos.

Primero, graficamos la respuesta de los motores ante diferentes comandos.

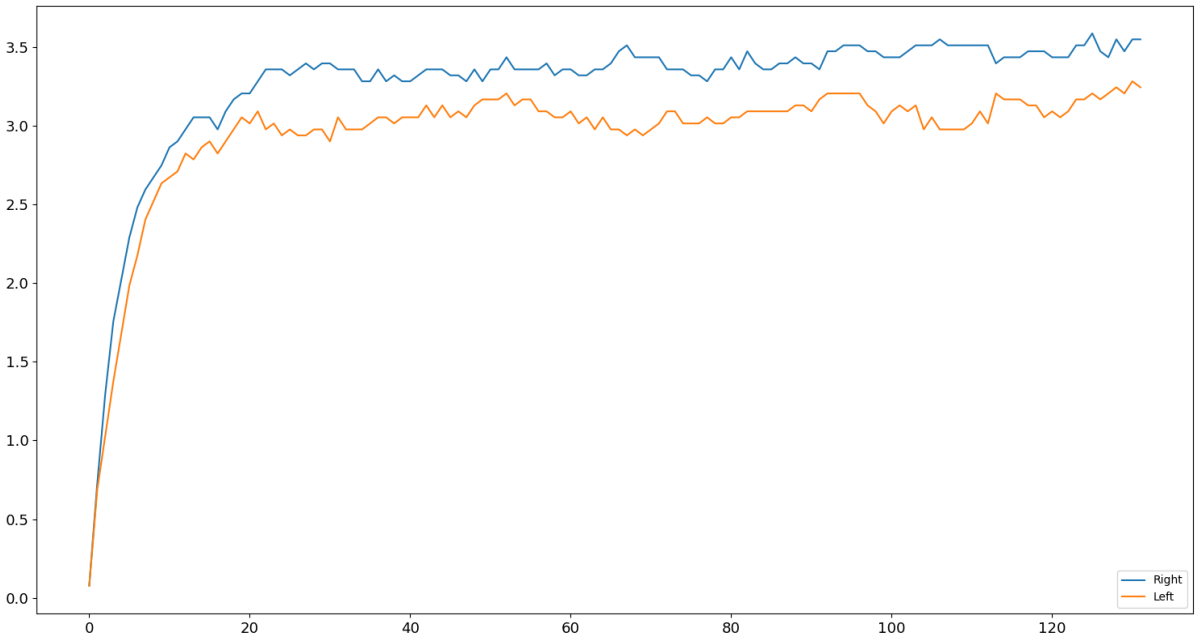


Fig. 18 - Comando RAW (o 1500 1500)

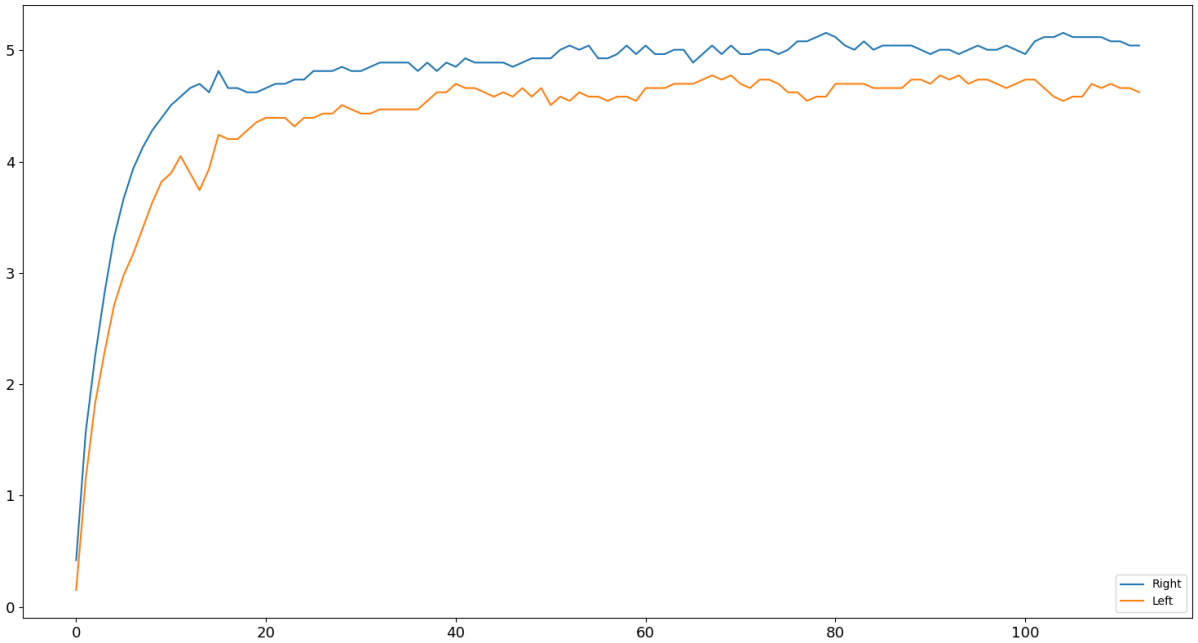


Fig. 19 - Comando RAW (o 2000 2000)

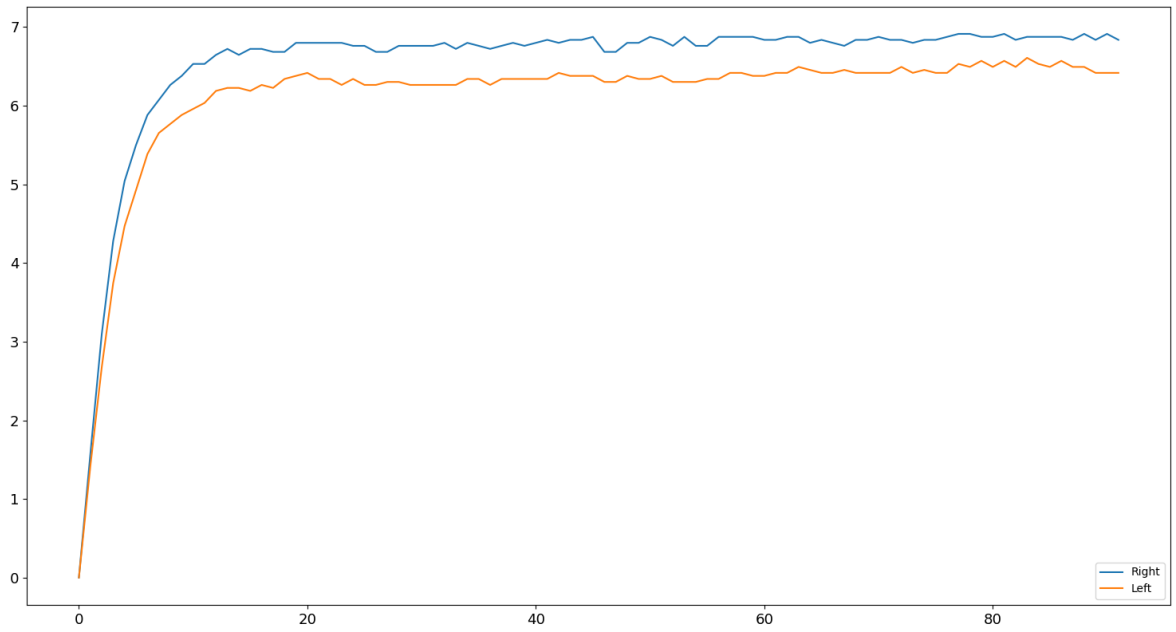


Fig. 20 - Comando RAW (o 3000 3000)

Como vemos en las imágenes, enviamos 3 comandos con valores directos al PWM, 1500, 2000 y 3000. Cada uno de estos comandos genera una velocidad aproximada de 3.5, 5.0 y 6.8 [rad/s] respectivamente. Lo que se puede apreciar de las gráficas es que, en todos los casos, la velocidad del motor derecho fue superior a la del motor izquierdo. Siendo que el comando PWM fue el mismo para ambos casos, podemos concluir que por temas constructivos, ya sea del Puente H o de los mismos motores, ambas respuestas no son iguales. Esta diferencia de velocidad produce que el robot no avance en línea recta, sino que tiende a doblar en la dirección de la rueda con menor velocidad.

Otro punto a tener en cuenta es que el valor de 1500 es el mínimo valor que podemos comandar para que los motores logren vencer la inercia y avanzar.

Ahora mostraremos la respuesta con comandos PID. En las siguientes imágenes vemos la respuesta del PID cuando establecemos una velocidad de 2, 3, 4 y 5 [rad/s]. Lo primero que podemos apreciar es que las velocidades de ambos motores ahora son idénticas. Con esto logramos que el robot avance en línea recta.

En las gráficas de 2 y 3 [rad/s] vemos que al principio el PID configura una velocidad bastante mayor al set point enviado en el comando, esto se debe a que, si el PID comandaba directamente la velocidad seteada, el robot no se movería ya que no podría vencer la inercia.

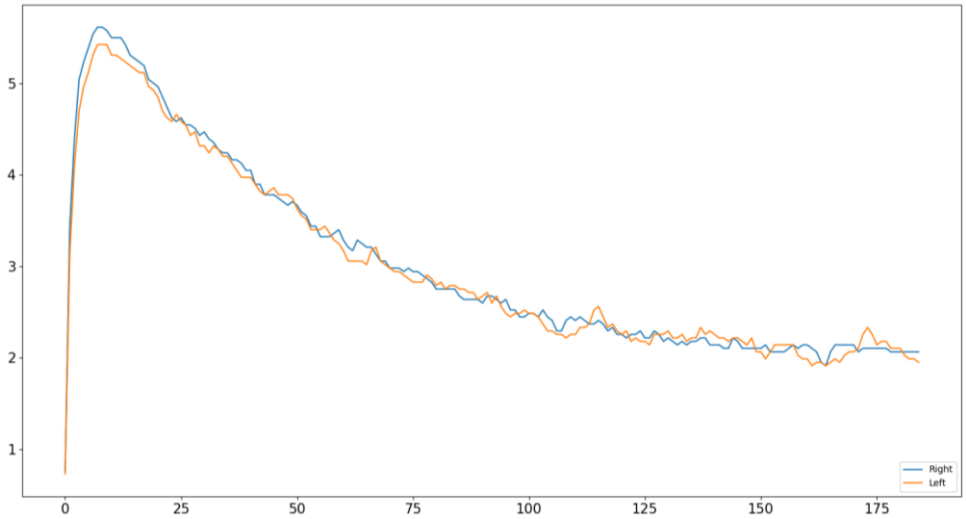


Fig. 21 - Comando PID (m 2 2)

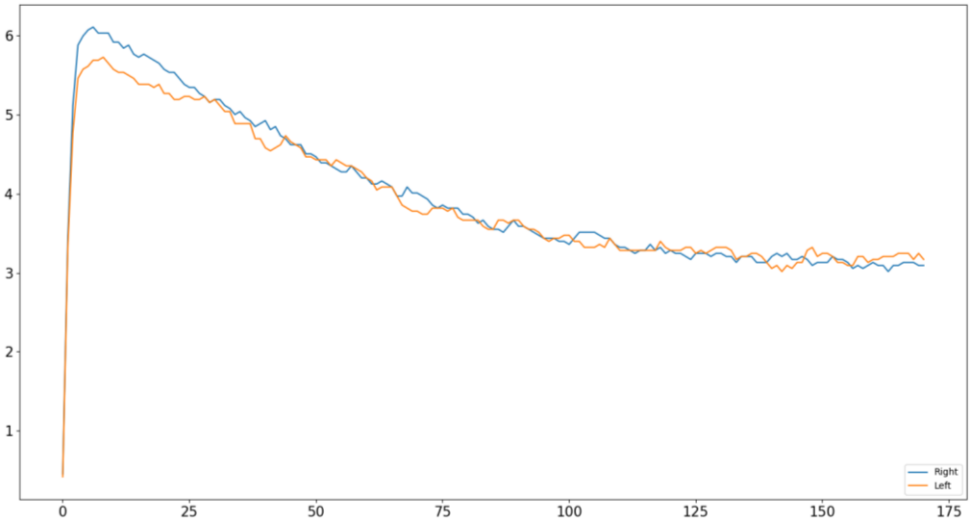


Fig. 22 - Comando PID (m 3 3)

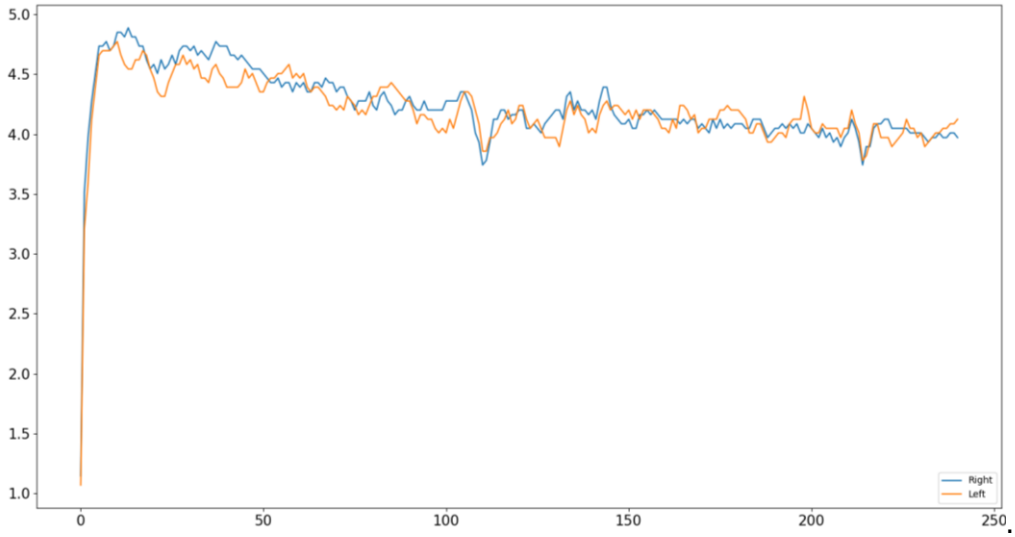


Fig. 23 - Comando PID (m 4 4)

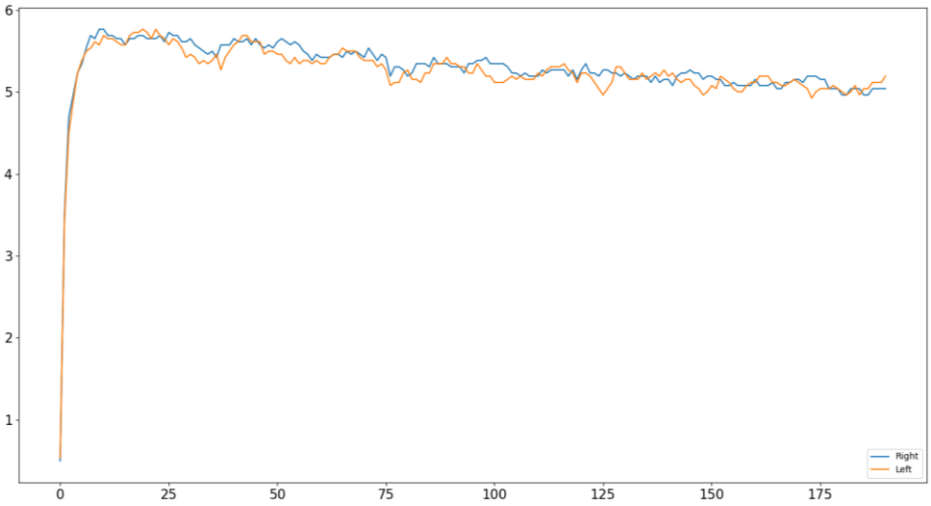


Fig. 24 - Comando PID (m 5 5)

De esta manera, aparte de lograr que las velocidades de ambas ruedas sean iguales, también podemos llegar a velocidades menores comparado con comandos directos de PWM.

## 2.4 Desarrollo en ROS.

Los paquetes en ROS se comunican a través de tópicos. A su vez, en ROS, existen muchos tipos de datos, por lo que dichos tópicos deben transportar un solo tipo de dato. Estos tipos de datos nos ayudan a entender que datos espera un paquete (y en qué tópicos), y que datos publica. Por ejemplo, un paquete denominado *IMU Complementary Filter*, toma datos crudos provenientes de una IMU (Acelerómetro, Giroscopio y opcionalmente Magnetómetro) y, a partir de dichos datos, calcula el Quaternion. El tipo de mensaje en ROS que encapsula los datos provenientes de la imu se denomina *sensor_msgs/Imu.msg* y está compuesto de la siguiente manera.

```
-   std_msgs/Header header
-   geometry_msgs/Quaternion orientation
-   float64[9] orientation_covariance
-   geometry_msgs/Vector3 angular_velocity
-   float64[9] angular_velocity_covariance
-   geometry_msgs/Vector3 linear_acceleration
-   float64[9] linear_acceleration_covariance
```

Tabla 7 - Composición sensor_msgs/Imu.msg

Por lo que vemos, está compuesto por un Header de tipo *std_msgs/Header*, el quaternion del tipo *geometry_msgs/Quaternion* con su matriz de covarianza, la

velocidad angular del tipo *geometry_msgs/Vector3* con su matriz de covarianza y, por último, la aceleración lineal del tipo *geometry_msgs/Vector3* también con su matriz de covarianza.

Dicho paquete espera los datos crudos del tipo que acabamos de mencionar en el tópico *imu/data_raw*, toma los datos, calcula el quaternion y publica en el tópico */imu/data* mensajes del mismo tipo. La diferencia de los mensajes en cada uno de los tópicos es que en el caso de *imu/data_raw* el campo del quaternion se encuentra vacío (todos los valores en cero), mientras que en */imu/data* ya se encuentra el quaternion calculado. Este mismo principio lo usan todos los paquetes dentro de ROS para poder tomar y publicar datos.

Otro concepto que debemos conocer es el de las TF o transformaciones. TF es un paquete que permite al usuario realizar un seguimiento de múltiples marcos de coordenadas a lo largo del tiempo. TF mantiene la relación entre los marcos de coordenadas en una estructura de árbol almacenada en el tiempo y permite al usuario transformar puntos, vectores, etc. entre dos marcos de coordenadas cualesquiera en cualquier momento deseado. Lo que quiere decir esto es que, teniendo en cuenta el robot que diseñamos, tendremos un marco de coordenadas (o frame) por cada componente del mismo (ruedas, IMU, LIDAR e incluso el robot). Estos “frames” son muy importantes ya que nos permiten conocer la posición de cada componente respecto de otro marco de coordenadas. Para el caso de los componentes que conforman el robot, estas transformaciones serán fijas ya que los componentes no tienen movimiento respecto de la base del robot. Estas transformaciones fijas, en conjunto con la forma y dimensiones del robot se describen mediante un archivo denominado Formato Unificado de Descripción del Robot (URDF o Unified Robot Description Format).

Así como existen frames fijos también existen frames continuos, por ejemplo, la odometría. La odometría es un concepto fundamental en robótica, que sirve como medio para estimar la posición y el movimiento de un robot mediante el análisis de datos de sensores. Existen diferentes métodos para calcular la odometría del robot según los sensores disponibles. Al ser un frame continuo debemos, mediante los diferentes sensores presentes en el robot, calcular en todo momento la odometría. Si

por algún motivo, mientras el robot está en movimiento, dejamos de calcular dicha transformación, perderemos la referencia respecto de la posición del robot.

Otro frame importante es la posición global del robot. Con la odometría podemos conocer la posición del robot respecto de su punto de partida, pero no conocemos la posición real del mismo respecto de su entorno. Esta función la cumple el frame global. En nuestro caso que trabajaremos con la posición del robot respecto de un mapa, este frame se denominará “map”. En el caso que tengamos un robot que se posicione con coordenadas globales (utilizando GPS por ejemplo), dicho frame suele denominarse “global”.

El responsable de calcular la transformación del frame “map” al frame “odometría”, en nuestro caso, será el paquete que realiza el SLAM. Como el sensor que utilizaremos para realizar el mapeo es el Lidar, en definitiva, el responsable de proveer dicha transformación será el Lidar.

Un punto a destacar es que la transformación map -> odometría no necesariamente debe calcularse de manera continua. Si volvemos al ejemplo del GPS, en dicho caso tendremos las coordenadas actualizadas una vez por segundo, por lo que la transformación se calculará una vez por segundo. A continuación, mostramos un diagrama de bloques de los frames presentes en nuestro robot.

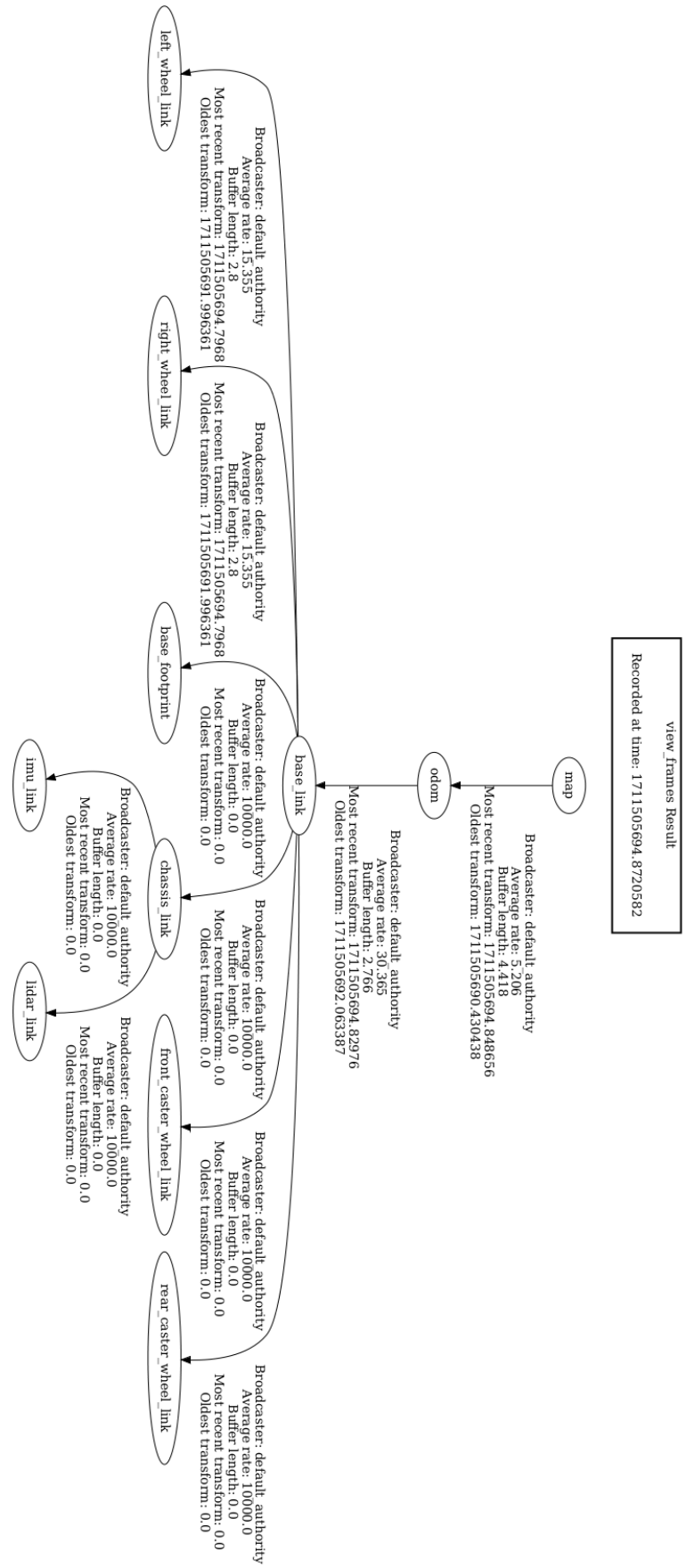


Fig. 25 - TF Frames

### 2.4.1 Descripción del robot (Archivo URDF).

El archivo URDF es un archivo XML que incluye la descripción física de un robot. Es esencialmente un modelo 3D con información de la forma, masa, articulaciones, ruedas, motores, etc.

Los archivos URDF también se utilizan en simuladores para probar el comportamiento del robot antes de su implementación en el mundo real, o en la creación de un doble digital para la visualización bidireccional en tiempo real del estado de un dispositivo robot.

Una vez que tenemos el archivo URDF, se lo pasaremos a un paquete denominado *robot_state_publisher*. Dicho paquete será el encargado de publicar la información de nuestro robot en el tópico *robot_description* con un tipo de mensaje *std_msgs/msg/String*.

Utilizando la herramienta de visualización *rviz2* podemos ver el robot que describimos mediante el archivo URDF.

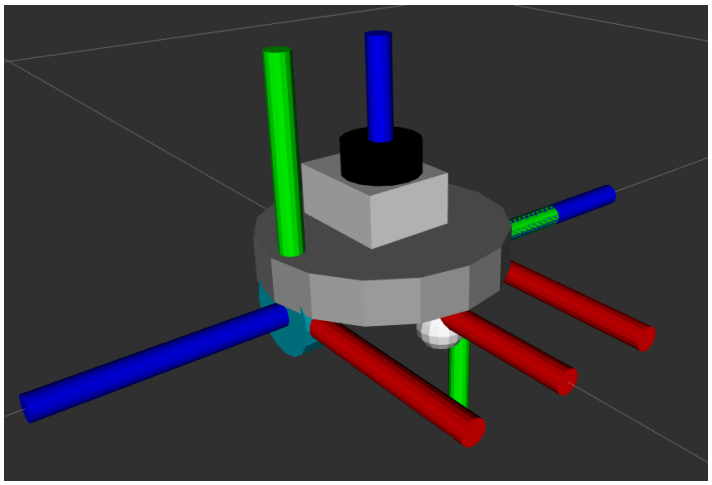

Fig. 26 - Visualización del robot en Rviz

En la imagen, aparte de ver el modelo 3D del robot, vemos también 3 conjuntos de ejes. Cada uno de estos conjuntos señala el origen del Frame correspondiente a cada una de las ruedas y al robot. Así mismo, también indican la orientación que tiene cada uno de ellos, en rojo se indica el eje X, en verde el eje Y y, por último, en azul el eje Z.

### 2.4.2 Interfaz de hardware.

Para la interfaz de hardware, es decir la comunicación entre ROS y el controlador de motores, utilizamos un paquete denominado *ros2_control*. Dicho paquete es un framework para el control en tiempo real de robots utilizando ROS 2. El objetivo de *ros2_control* es simplificar la integración de nuevo hardware y superar algunos inconvenientes.

Cualquiera sea nuestro hardware, para usar *ros2_control* con él, necesitamos algo llamado interfaz de hardware. Este es un fragmento de código que se comunica con el hardware y lo expone a *ros2_control* de la manera que dicho paquete lo espera. Dicha interfaz actúa como una abstracción, de modo que, como usuarios, todo lo que necesitamos entender es la forma en que se representa nuestro hardware, que es a través de interfaces de comando e interfaces de estado. Las interfaces de comando nos permiten actuar (o enviar información al robot) mientras que las interfaces de estado solo permiten observar valores que indican su condición de funcionamiento actual.

En nuestro caso, en el robot que estamos desarrollando, solo podemos controlar la velocidad de las ruedas por lo que nuestra interfaz de hardware tendrá dos interfaces de comando, una por cada motor. Y, para el caso de las interfaces de estado, utilizando los encoders podemos medir la velocidad y la posición de las ruedas por lo que tendremos 4 interfaces de estado, 2 por cada encoder.

Para que nuestra interfaz se comunique correctamente con *ros2_control* debemos implementar una serie de funciones. Dichas funciones inicializan la interfaz, dan a conocer a *ros2_control* las diferentes interfaces (de comando y de estado), le dicen que hacer al activar o desactivar la interfaz y, por último, como leer o escribir el puerto serie para comunicarse con el controlador. Dichas funciones son las siguientes.

```
  DIFFDRIVE_STM32_PUBLIC
  hardware_interface::CallbackReturn on_init(
    const hardware_interface::HardwareInfo & info) override;

  DIFFDRIVE_STM32_PUBLIC
  std::vector<hardware_interface::StateInterface>
export_state_interfaces() override;

  DIFFDRIVE_STM32_PUBLIC
  std::vector<hardware_interface::CommandInterface>
export_command_interfaces() override;
```

```
  DIFFDRIVE_STM32_PUBLIC
  hardware_interface::CallbackReturn on_activate(
    const rclcpp_lifecycle::State & previous_state) override;

  DIFFDRIVE_STM32_PUBLIC
  hardware_interface::CallbackReturn on_deactivate(
    const rclcpp_lifecycle::State & previous_state) override;

  DIFFDRIVE_STM32_PUBLIC
  hardware_interface::return_type read(
    const rclcpp::Time & time, const rclcpp::Duration & period) override;

  DIFFDRIVE_STM32_PUBLIC
  hardware_interface::return_type write(
    const rclcpp::Time & time, const rclcpp::Duration & period) override;
```

Tabla 8 - Funciones interfaz de hardware.

Una vez que implementamos dichas funciones estamos en condiciones de iniciar *ros2_control.* Una vez iniciado, si le pedimos a *ros2_control* que nos liste las interfaces existentes, nos devuelve lo siguiente.

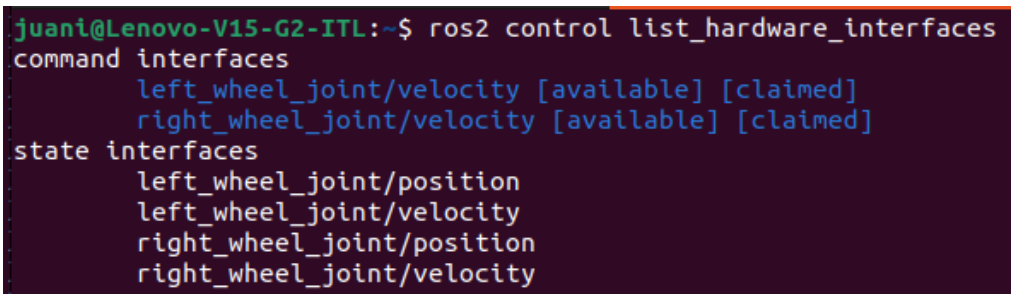


Fig. 27 - Interfaces de hardware

Para configurar la interfaz de hardware, utilizamos también un archivo XML. Dentro de este archivo configuramos, entre otras cosas, el puerto al que debe conectarse la interfaz (en nuestro caso un puerto serie), la velocidad de comunicación y la cantidad de cuentas por revolución que entregan los encoders.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:macro name="raspibot_ros2_control" params="name prefix">

    <ros2_control name="${name}" type="system">
      <hardware>
        <plugin>diffdrive_stm32/DiffDriveSTM32Hardware</plugin>
        <param name="left_wheel_name">left_wheel_joint</param>
        <param name="right_wheel_name">right_wheel_joint</param>
        <param name="loop_rate">30</param>
        <param name="device">/dev/ttyACM0</param>
        <param name="baud_rate">921600</param>
        <param name="timeout_ms">1000</param>
```

```
        <param name="enc_counts_per_rev">3290</param>
      </hardware>
      <joint name="${prefix}left_wheel_joint">
        <command_interface name="velocity"/>
        <state_interface name="position"/>
        <state_interface name="velocity"/>
      </joint>
      <joint name="${prefix}right_wheel_joint">
        <command_interface name="velocity"/>
        <state_interface name="position"/>
        <state_interface name="velocity"/>
      </joint>
    </ros2_control>

  </xacro:macro>

</robot>
```

Tabla 9 - Configuración interfaz de hardware.

A su vez necesitaremos algo denominado *Controlador*. Los controladores son la forma en que el resto del ecosistema ROS interactúa con ros2_control. Por un lado, escuchan un topic de ROS para obtener información de control (como ser posiciones de las articulaciones o velocidades). Toman esta información, utilizan algún tipo de algoritmo para determinar las velocidades, posiciones, etc. apropiadas del actuador, y lo envían a las interfaces de hardware correspondientes.

Si bien las interfaces de hardware se crearán específicamente para diferentes hardware de robots, los controladores dependen de la aplicación y, dado que existen algunas aplicaciones comunes, el paquete *ros2_controllers* proporciona una serie de controladores que deberían cubrir las necesidades de la mayoría de los casos. El robot en el que estamos trabajando, como comentamos anteriormente, es de accionamiento diferencial, por lo que, naturalmente, usaremos *diff_drive_controller*.

A continuación, mostramos el archivo de configuración que utilizamos para configurar el controlador. En este caso se utiliza un archivo YAML.

```
controller_manager:
  ros__parameters:
    update_rate: 30  # Hz

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    raspibot_base_controller:
      type: diff_drive_controller/DiffDriveController

raspibot_base_controller:
```

```
ros__parameters:
  left_wheel_names: ["left_wheel_joint"]
  right_wheel_names: ["right_wheel_joint"]

  wheel_separation: 0.169
  wheel_radius: 0.034

  wheel_separation_multiplier: 1.0
  left_wheel_radius_multiplier: 1.0
  right_wheel_radius_multiplier: 1.0

  publish_rate: 30.0
  odom_frame_id: odom
  base_frame_id: base_link
  pose_covariance_diagonal : [0.001, 0.001, 0.001, 0.001, 0.001, 0.01]
  twist_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.01]

  open_loop: false
  enable_odom_tf: false
```

Tabla 10 - Configuración ros2_control

Una vez que tenemos la interfaz de hardware y el controlador configurados, procedemos a testear el movimiento del robot. Además de corroborar que el robot se mueva en línea recta, debemos constatar que la distancia real recorrida por el robot sea lo más aproximada posible a la distancia recorrida que reporta en ROS. Esto lo realizamos comandando al robot que se mueva, midiendo la distancia recorrida con un metro y, luego, utilizando la herramienta de visualización *rviz2* medimos la distancia reportada en ROS. En las siguientes imágenes podemos visualizar el recorrido realizado por el robot. Colocamos una cinta métrica paralela al recorrido para poder visualizar en ella la distancia.

Fig. 28 - Test de distancia recorrida

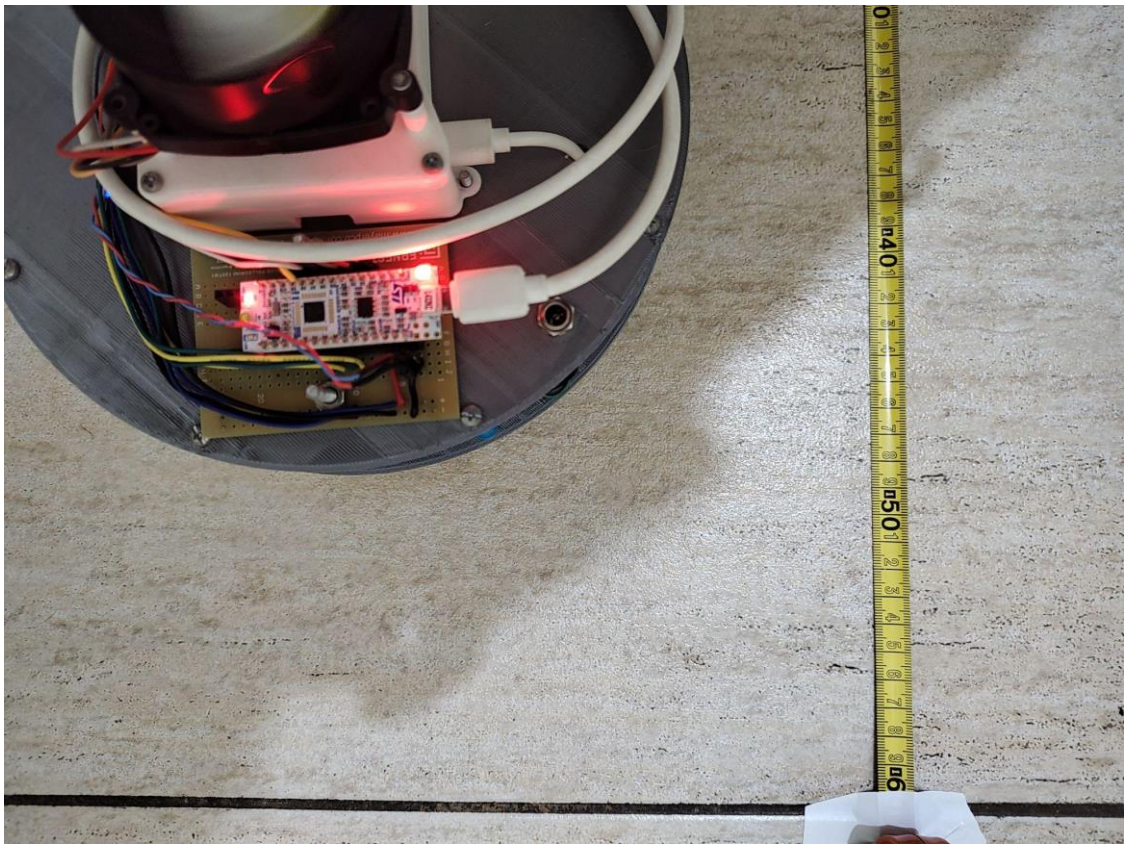


Fig. 29 - Distancia Recorrida

Luego, en el programa rviz, podemos ver la distancia sensada por ROS a partir de los encoders y la interfaz de hardware desarrollada.

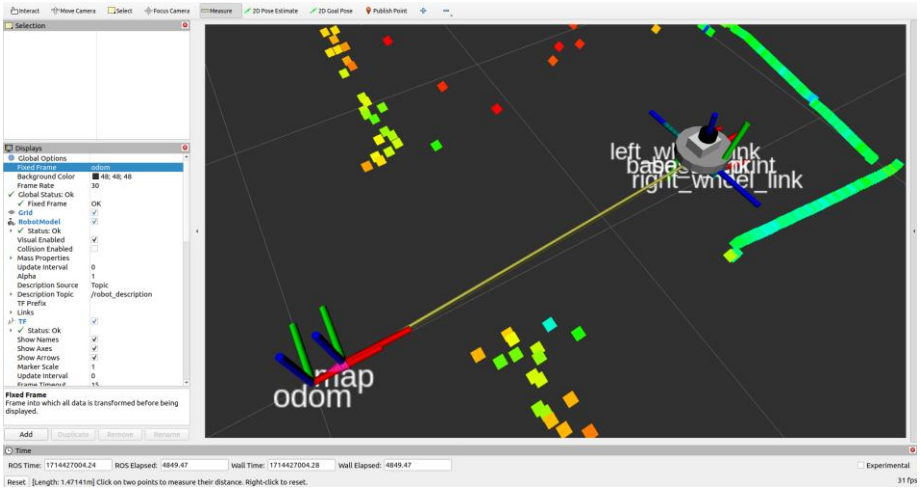


Fig. 30 - Distancia en Rviz

En la esquina inferior izquierda de la pantalla podemos visualizar la medición. La misma nos dice 1,4714 [m], muy aproximada a la medición realizada con la cinta métrica.

La odometría se publica bajo el tópico */raspibot_base_controller/odom* con un tipo de mensaje *nav_msgs/Odometry*.

Para comandar al robot que se mueva, utilizamos un paquete denominado *teleop_twist_keyboard*. Dicho paquete captura las teclas del teclado y nos deja utilizarlo a modo de “joystick”. Luego, de acuerdo a las teclas que pulsamos, publica en el topic */cmd_vel* mensajes del tipo *geometry_msg/Twist.msg*.

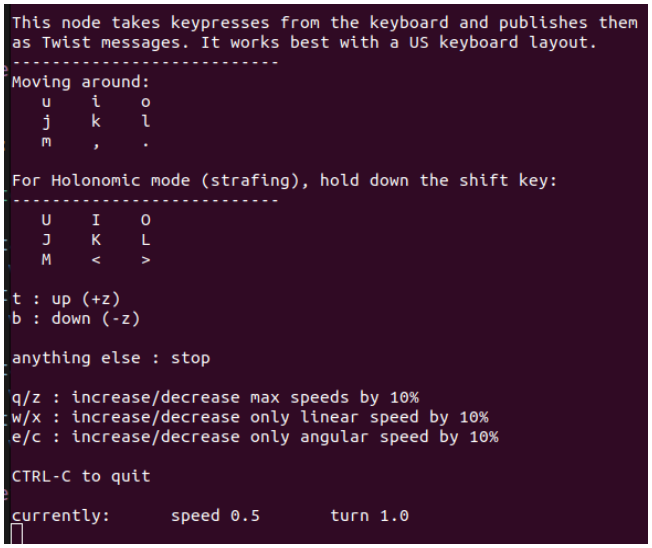


Fig. 31 - Teleop twist keyboard

### 2.4.3 Lidar.

Al igual que existen paquetes en ROS que nos resuelven diferentes problemas, también existen drivers de sensores encapsulados en paquetes de ROS. Este es el caso de nuestro lidar. Como comentamos anteriormente, el modelo del Lidar es el LDS-01, por lo que utilizaremos el paquete *hls_lfcd_lds_driver*. Los únicos dos parámetros que nos pide este paquete son, el puerto (*/dev/ttyUSB0*) donde está conectado el Lidar, y el frame (*lidar_link*) donde se encuentra el mismo. Si ejecutamos el paquete y visualizamos en Rviz podemos ver las mediciones que nos entrega el sensor.

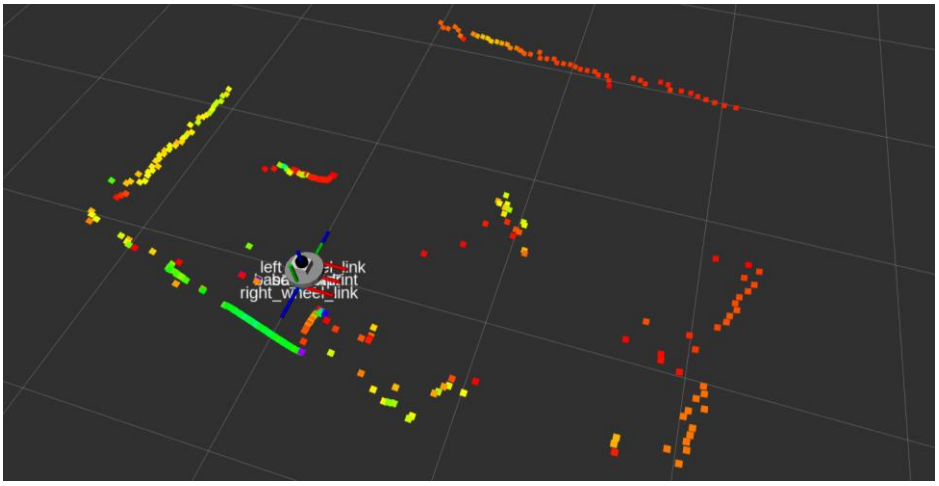

Fig. 32 - Mediciones Lidar

El driver publica las mediciones con tipo de mensaje *sensor_msgs/LaserScan* en el tópico */scan*. Cada punto que vemos en la imagen corresponde a una medición del Lidar en un ángulo concreto. El color de cada punto es solo ilustrativo, es un indicativo de la distancia a la que se encuentra dicho punto.

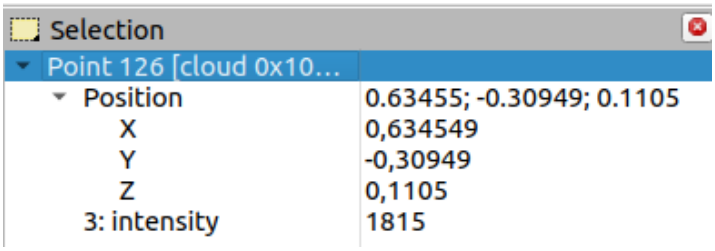


Fig. 33 - Punto Lidar

Si seleccionamos un punto de los que vimos en la imagen anterior, podemos ver la información de dicho punto, es decir, la posición en coordenadas XYZ y la intensidad del láser en esa medición concreta.

### 2.4.4 IMU.

Respecto del IMU, no encontramos un paquete de ROS que nos provea el driver de dicho sensor, por lo que tuvimos que desarrollarlo. El IMU se encuentra conectado por I2C directo a la Raspberry PI.

Nuestro driver tiene que, comunicarse con el sensor, solicitar los valores del Giroscopio y Acelerómetro y publicarlos en un tópico con tipo de mensaje *sensor_msgs/Imu.msg*.

La función principal de nuestro Driver es la siguiente. En ella podemos ver como creamos el mensaje del tipo correcto, solicitamos los datos al sensor y publicamos el mensaje.

```
void MPU9250Driver::handleInput()
{
  auto message = sensor_msgs::msg::Imu();
  message.header.stamp = this->get clock()->now();
  message.header.frame_id = "base_link";

  // Direct measurements
  message.linear_acceleration_covariance = {1e-3,1e-3,1e-3,1e-3,1e-3,1e-3,1e-3,1e-3,1e-3};
  message.linear_acceleration.x = mpu9250_->getAccelerationX();
  message.linear_acceleration.y = mpu9250_->getAccelerationY();
  message.linear_acceleration.z = mpu9250_->getAccelerationZ();
  message.angular_velocity_covariance = {1e-6, 0, 0, 0, 1e-6, 0, 0, 0, 1e-6};
  message.angular_velocity.x = mpu9250_->getAngularVelocityX()*GRADS_TO_RADS;
  message.angular_velocity.y = mpu9250_->getAngularVelocityY()*GRADS_TO_RADS;
  message.angular_velocity.z = mpu9250_->getAngularVelocityZ()*GRADS_TO_RADS;

  //calculateOrientation(message);
  message.orientation_covariance[0] = -1;
  message.orientation.x = 0;
  message.orientation.y = 0;
  message.orientation.z = 0;
  message.orientation.w = 0;
  publisher_->publish(message);
}
```

Tabla 11 - Input handler IMU

A la hora de solicitar la velocidad angular del giroscopio, el sensor nos entrega el valor en grados por segundos. En ROS, la unidad de velocidad angular es en rad/s, por lo que debemos convertir las velocidades del giroscopio a dicha unidad antes de colocarlas en el mensaje.

Utilizando el comando *ros2 topic echo /imu/data_raw*, podemos visualizar los datos correspondientes al imu, publicados por el driver.

```
juani@Lenovo-V15-G2-ITL:~/Proyecto_Final$ ros2 topic echo /imu/data_raw
header:
  stamp:
    sec: 1714427341
    nanosec: 971008445
  frame_id: base_link
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0
orientation_covariance:
- -1.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
- 0.0
angular_velocity:
  x: 0.0011464553167938915
  y: -0.025469027795419846
  z: -0.0009617967381679737
angular_velocity_covariance:
- 1.0e-06
- 0.0
- 0.0
- 0.0
- 1.0e-06
- 0.0
- 0.0
- 0.0
- 1.0e-06
linear_acceleration:
  x: -0.013536650390625016
  y: -0.002771037597656123
  z: 9.998162314453142
linear_acceleration_covariance:
- 0.01
- 0.01
- 0.01
- 0.01
- 0.01
- 0.01
- 0.01
- 0.01
- 0.01
---
```

Fig. 34 - Datos IMU

### 2.4.5 Configuración de la Odometría.

Si bien podríamos realizar SLAM solo con la trayectoria que calcula *ros_control* con los datos provenientes de los encoders, podemos mejorar dicho cálculo si le sumamos los datos provenientes de la IMU. La IMU también nos ayudará a detectar si las ruedas del robot “patinan”. Esto último es muy importante, ya que, de lo contrario, ante la eventualidad de que ocurra que las ruedas patinen, los encoders informaran que el robot se está moviendo, cuando en realidad, está quieto en el lugar.

Para fusionar los datos de los encoders con los datos de la IMU, utilizamos un paquete denominado *robot_localization*. Dicho paquete ofrece la posibilidad de utilizar un filtro de kalman para fusionar datos que tienen que ver con el movimiento del robot.

Podemos fusionar la cantidad de sensores que queramos, siempre y cuando los tipos de mensajes que ingresemos sean algunos de los siguientes.

- nav_msgs/Odometry
- sensor_msgs/Imu
- geometry_msgs/PoseWithCovarianceStamped
- geometry_msgs/TwistWithCovarianceStamped

Como vimos anteriormente, *ros2_control* publica datos de odometría con tipo de mensaje *nav_msgs/Odometry*, y nuestro driver de la IMU publica datos con tipo de mensaje *sensor_msgs/Imu*. Por lo tanto, estamos en condiciones de fusionar ambos sensores. A continuación, mostramos las configuraciones de *robot_localization.*

```
### ekf config file ###
ekf_localization_node:
    ros__parameters:

        frequency: 30.0

        two_d_mode: true

        print_diagnostics: true

        publish_tf: true

        map_frame: map              # Defaults to "map" if unspecified
        odom_frame: odom            # Defaults to "odom" if unspecified
        base_link_frame: base_link  # Defaults to "base_link"
        world_frame: odom           # Defaults to value of odom_frame

        odom0: raspibot_base_controller/odom
        odom0_config: [false,  false,  false,
                       false, false, false,
                       true, true, false,
                       false, false, true,
                       false, false, false]
        odom0_queue_size: 30
        odom0_differential: false
        odom0_relative: true

        imu0: imu/data_raw
        imu0_config:   [false, false, false,
                        false, false, true,
                        false, false, false,
                        false, false, true,
                        true, false, false]
        imu0_queue_size: 30
        imu0_differential: false
```

Tabla 12 - Configuración robot_localization

*Robot_localization*, además de publicar los datos fusionados en el tópico */odometry/odom_to_base_link*, también publica la transformación (TF) desde la odometría a la base del robot (base_link). Si quisiéramos podríamos desactivar la publicación del TF mediante la opción *publish_tf* en el archivo de configuración.

Otras configuraciones presentes en el archivo son, la frecuencia de publicación de los datos, los nombres de los distintos frames y los vectores de configuración. En este caso, como hemos puesto *odom* como world_frame, la transformación calculada, como hemos dicho anteriormente, será la de *odom* a *base_link.*

Los vectores de configuración indican a *robot_localization* qué datos de cada sensor debe utilizar para fusionar. Cada valor del vector de configuración representa lo siguiente.

$$config\_vector = \begin{bmatrix} x & y & z \\ roll & pitch & yaw \\ v_x & v_y & v_z \\ v_{roll} & v_{pitch} & v_{yaw} \\ a_x & a_y & a_z \end{bmatrix}$$

Fig. 35 - Vector de configuración.

Como podemos ver, para el caso de *odom0*, que corresponde a la odometría proveniente de las ruedas, fusionamos la velocidad en X, la velocidad en Y y la velocidad de giro sobre el eje Z (yaw). Para el caso de *imu0*, fusionamos el ángulo de orientación en el eje Z (yaw), la velocidad de giro sobre el eje Z y, por último, la aceleración en el eje X.

### 2.4.6 Mapeo y localización simultáneos (SLAM).

Para el caso del SLAM utilizamos un paquete denominado *slam_toolbox*. Este paquete tomará las mediciones del Lidar y la transformación TF odom->base_link y a partir de ellas comenzará a armar un mapa.

El tipo de mapa generado se denomina “ocupancy grid” o grilla de ocupación. El nombre se debe a que está compuesto por celdas, cada una con sus coordenadas XY y con un valor que nos indica si dicha celda se encuentra ocupada o no. Cada celda está marcada con un número que indica la probabilidad de que la celda

contenga un objeto. El número suele ser de 0 (espacio libre) a 100 (probablemente 100% ocupado). Las áreas no escaneadas (es decir, por el LIDAR, el sensor ultrasónico o algún otro sensor de detección de objetos) se marcan con -1.

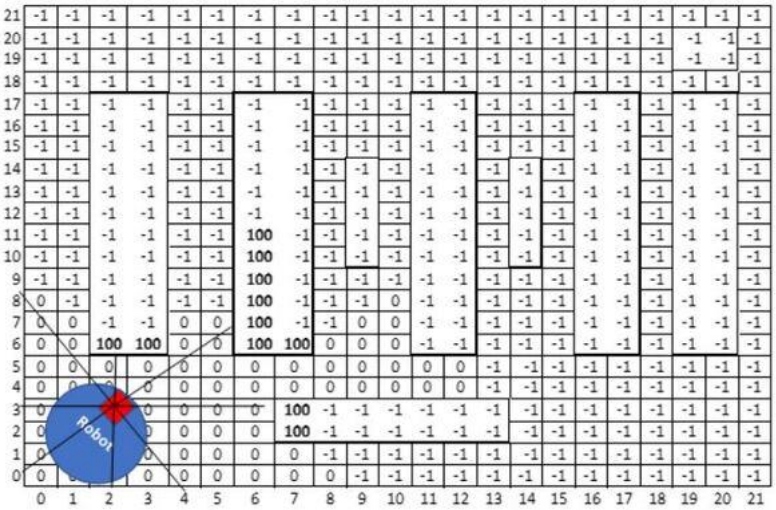


Fig. 36 - Ejemplo de grilla de ocupación.

Al igual que los paquetes que utilizamos anteriormente, *slam_toolbox* posee un archivo de configuración. A continuación, mostramos el archivo utilizado y pasamos a detallar algunas de las opciones.

```
slam_toolbox:
  ros__parameters:

    # Plugin params
    solver_plugin: solver_plugins::CeresSolver
    ceres_linear_solver: SPARSE_NORMAL_CHOLESKY
    ceres_preconditioner: SCHUR_JACOBI
    ceres_trust_strategy: LEVENBERG_MARQUARDT
    ceres_dogleg_type: TRADITIONAL_DOGLEG
    ceres_loss_function: HuberLoss

    # ROS Parameters
    odom_frame: odom
    map_frame: map
    base_frame: base_footprint
    scan_topic: /scan
    scan_queue_size: 10
    use_map_saver: true
    mode: mapping #localization

    debug_logging: false
    throttle_scans: 1
    transform_publish_period: 0.0 #if 0 never publishes odometry
    map_update_interval: 5.0
    resolution: 0.05
    max_laser_range: 3.5 #for rastering images
    minimum_time_interval: 0.5
    transform_timeout: 0.2
    tf_buffer_duration: 30.
```

```
    stack_size_to_use: 400000000
    enable_interactive_mode: false

    # General Parameters
    use_scan_matching: true
    use_scan_barycenter: true
    minimum_travel_distance: 0.25
    minimum_travel_heading: 0.25
    scan_buffer_size: 10
    scan_buffer_maximum_scan_distance: 2.0
    link_match_minimum_response_fine: 0.1
    link_scan_maximum_distance: 0.5
    loop_search_maximum_distance: 3.0
    do_loop_closing: true
    loop_match_minimum_chain_size: 10
    loop_match_maximum_variance_coarse: 3.0
    loop_match_minimum_response_coarse: 0.35
    loop_match_minimum_response_fine: 0.45

    position_covariance_scale: 1000.0
    yaw_covariance_scale: 1000.0

    # Correlation Parameters - Correlation Parameters
    correlation_search_space_dimension: 0.5
    correlation_search_space_resolution: 0.01
    correlation_search_space_smear_deviation: 0.1

    # Correlation Parameters - Loop Closure Parameters
    loop_search_space_dimension: 250.0
    loop_search_space_resolution: 0.05
    loop_search_space_smear_deviation: 0.03
```

Tabla 13 - Configuración slam_toolbox

Las opciones que se encuentran debajo de *#Plugin params* se refieren a los diferentes algoritmos utilizados para el mapeo. Luego, debajo de *#ROS Parameters* configuramos los diferentes frames, *odom_frame, map_frame* y *base_frame*, y el tópico donde se encontrarán los datos del LIDAR.

También podemos configurar la resolución del mapa (*resolution*), la frecuencia con la que se actualiza el mapa (*map_update_interval*) y la distancia máxima que mide el LIDAR (*max_laser_range*).

Después, bajo *#General Parameters* y *#Correlation Parameters* encontramos configuraciones orientadas al “matcheo” del scan. Además de para mapear, *slam_toolbox* utiliza el LIDAR para localizar el robot dentro del mapa. El “matcheo” del scan es lo que se utiliza para encontrar coincidencia entre el relevamiento que se está realizando (scan) y el mapa creado, de esta manera es como *slam_toolbox* logra localizar el robot.

A continuación, detallamos los tópicos publicados por *slam_toolbox*.

- /map -> Mensajes del tipo *nav_msgs/Occupancy_Grid.*
- /pose -> Mensajes del tipo *geometry_msgs/PoseWithCovarianceStamped.*

Bajo el tópico /map se publican los mensajes que tienen que ver con el mapeo y, luego, bajo el tópico /pose se publican los mensajes que tienen que ver con la localización del robot (el nombre “pose” viene de “position estimation”, que en inglés significa estimación de la posición). Además de los tópicos, podemos configurar a *slam_toolbox* para que publique la transformación map->odom. En nuestro caso optamos por ejecutar otro nodo de *robot_localization* para que fusione la odometría publicada por el nodo anterior del mismo paquete, con los datos publicados por *slam_toolbox* en el tópico /pose. De esta manera obtenemos datos más suavizados de localización. Cabe destacar que la transformación map->odom es muy importante para conocer la posición exacta del robot dentro del mapa.

Luego de estas configuraciones, iniciamos nuevamente el robot y ejecutamos la herramienta rviz para poder visualizar el mapeo. Al iniciar rviz podemos ver como *slam_toolbox* comienza a armar el mapa a partir de los datos del LIDAR.

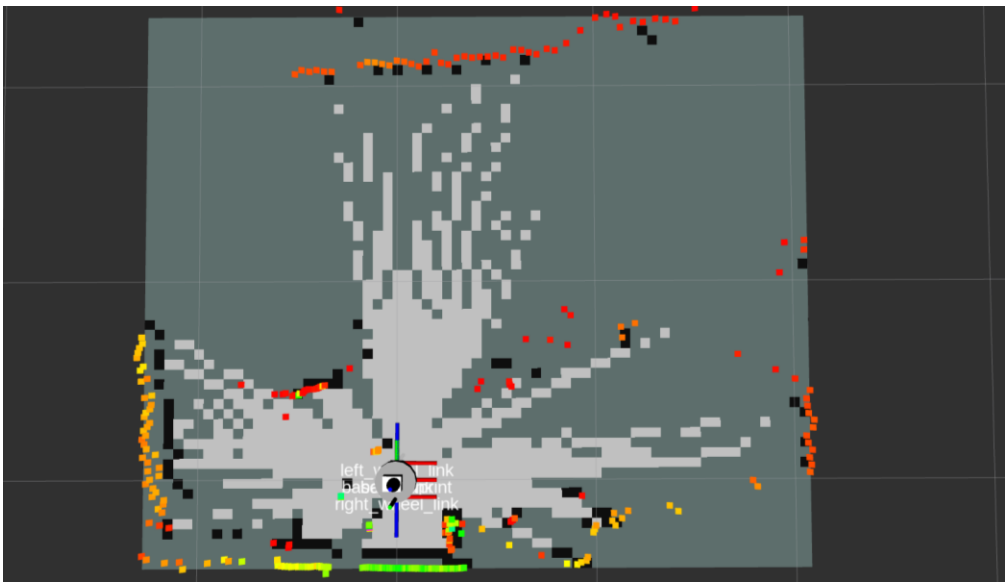

Fig. 37 - Slam Toolbox

A continuación, mostraremos una serie de capturas para entender cómo se va armando el mapa a medida que movemos el robot.

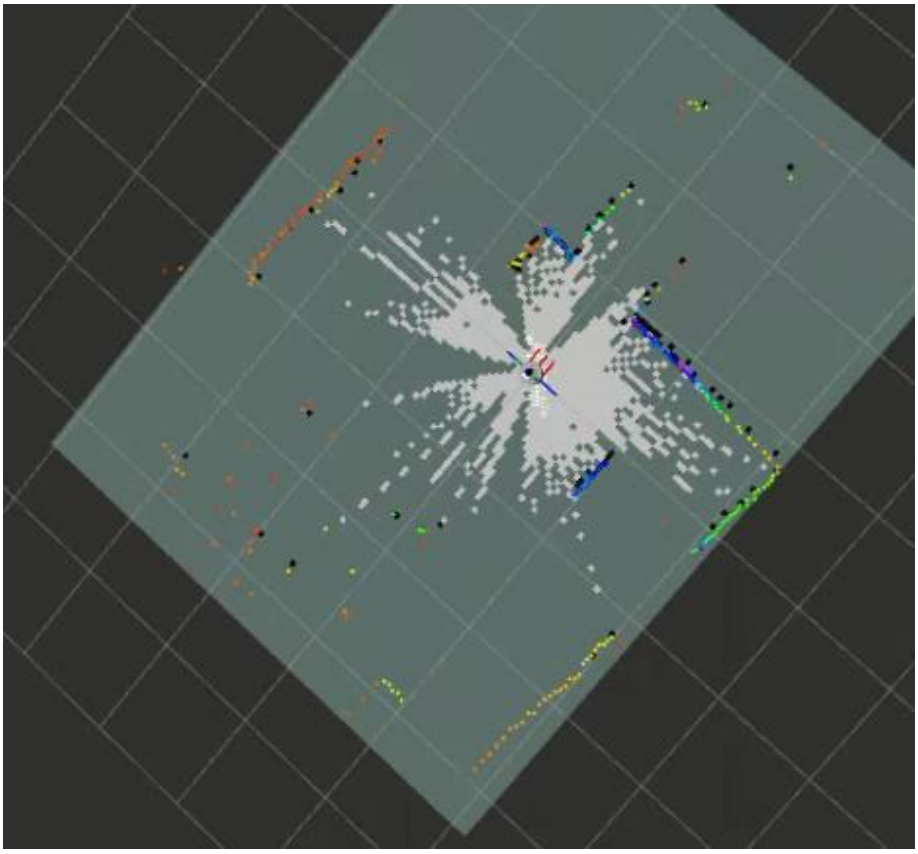

Fig. 38 - Mapa 1

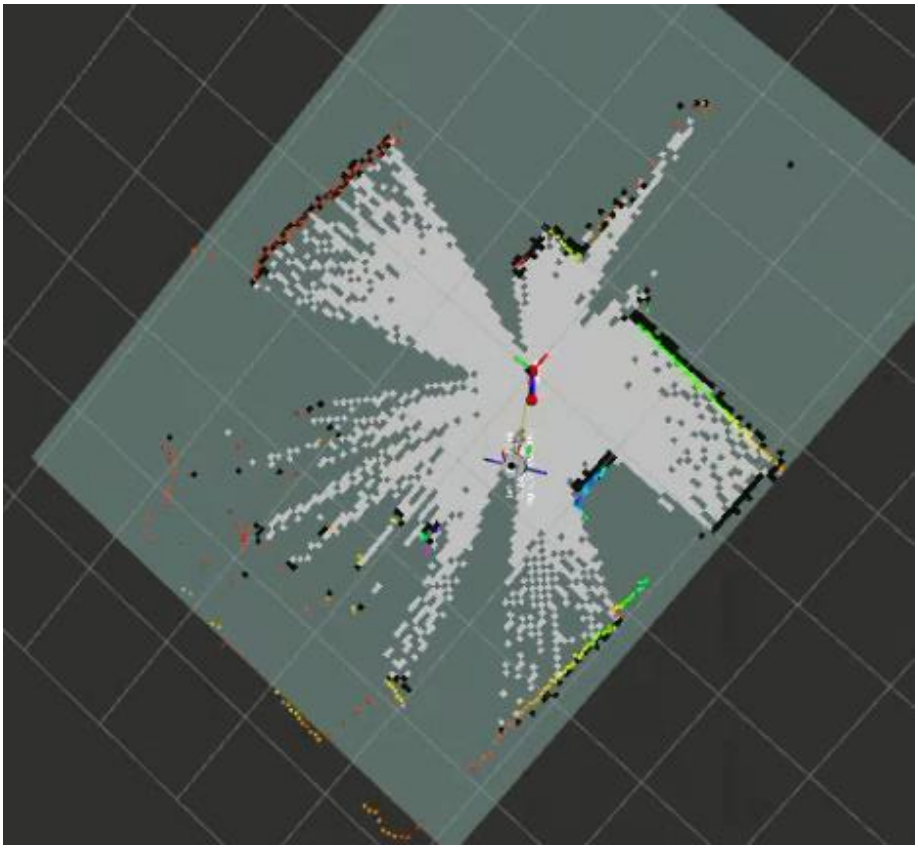

Fig. 39 - Mapa 2

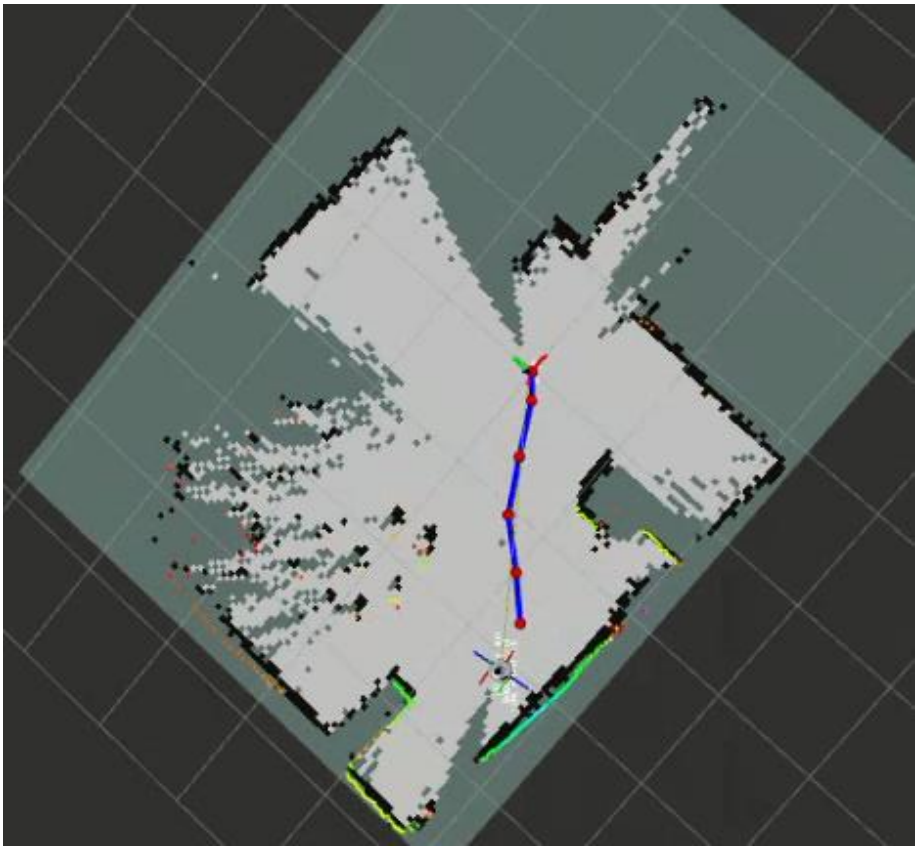

Fig. 40 - Mapa 3

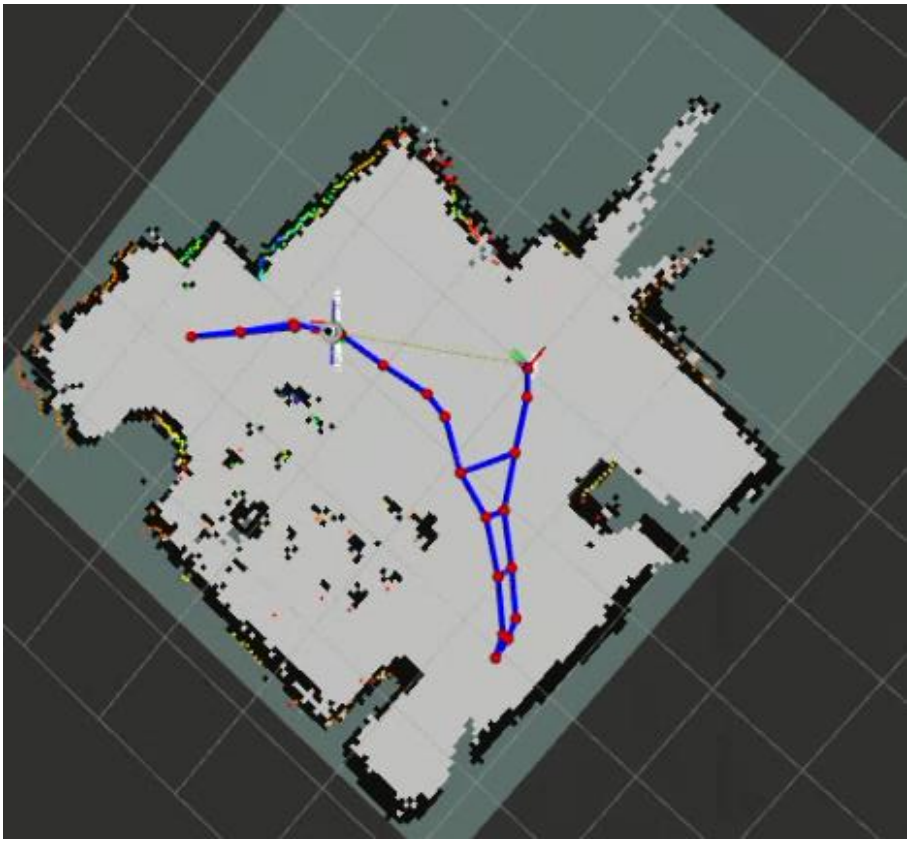

Fig. 41 - Mapa 4

Fig. 42 - Mapa 5

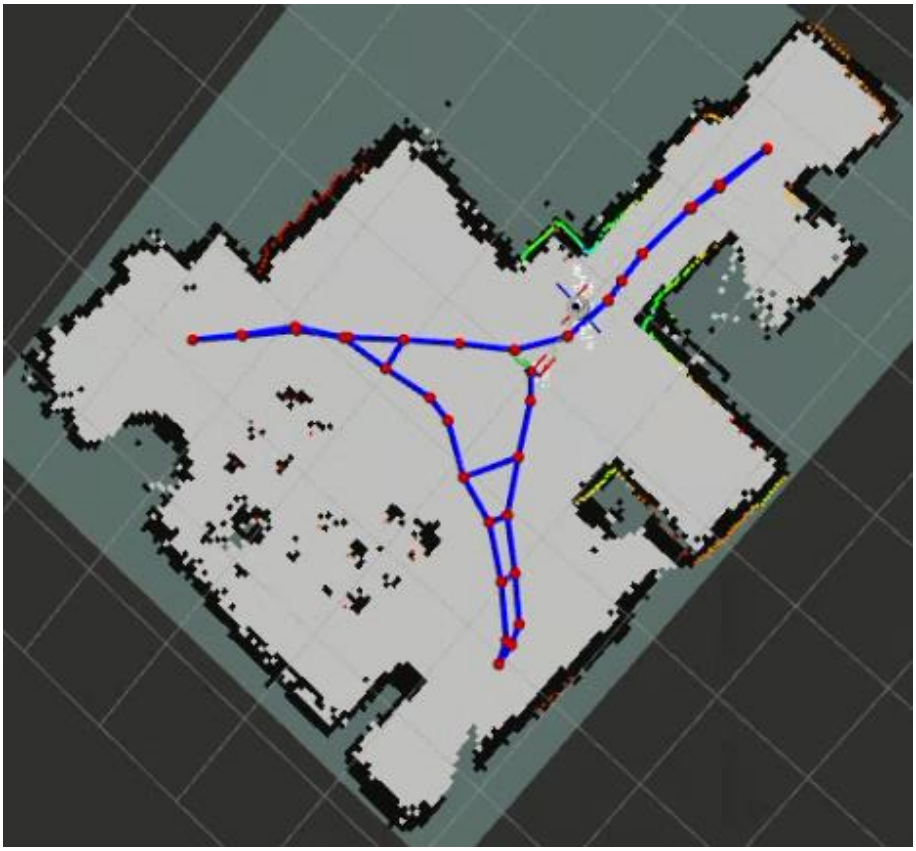

Fig. 43 - Mapa 6

En la secuencia de imágenes podemos observar perfectamente como Slam Toolbox va armando el mapa. Las líneas azules corresponden a la trayectoria del robot, mientras que los puntos rojos son los lugares donde Slam Toolbox calculó la posición del robot respecto del mapa. A continuación, compararemos el mapa realizado con un plano del lugar. De esta manera veremos si las dimensiones del mapa realizado por Slam Toolbox son correctas.

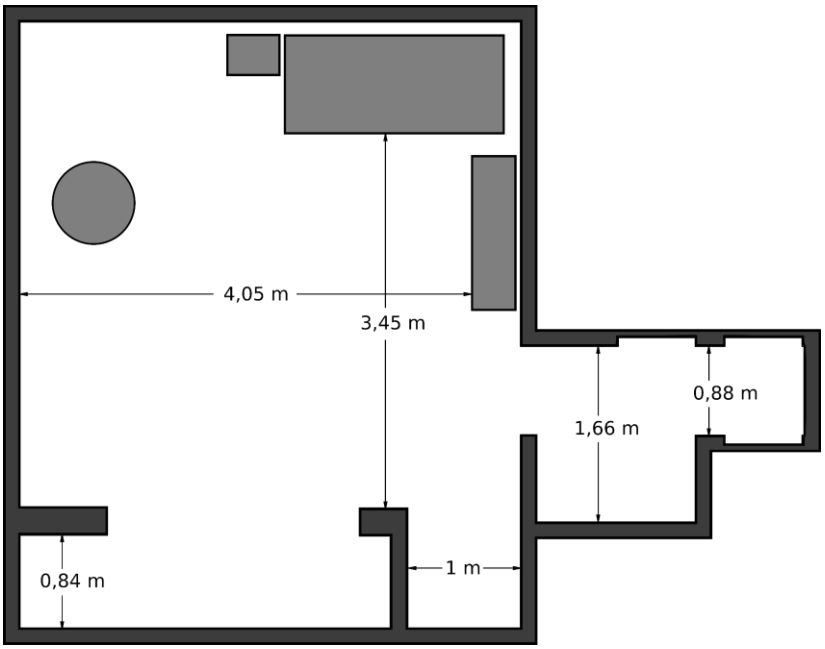


Fig. 44 - Plano del lugar

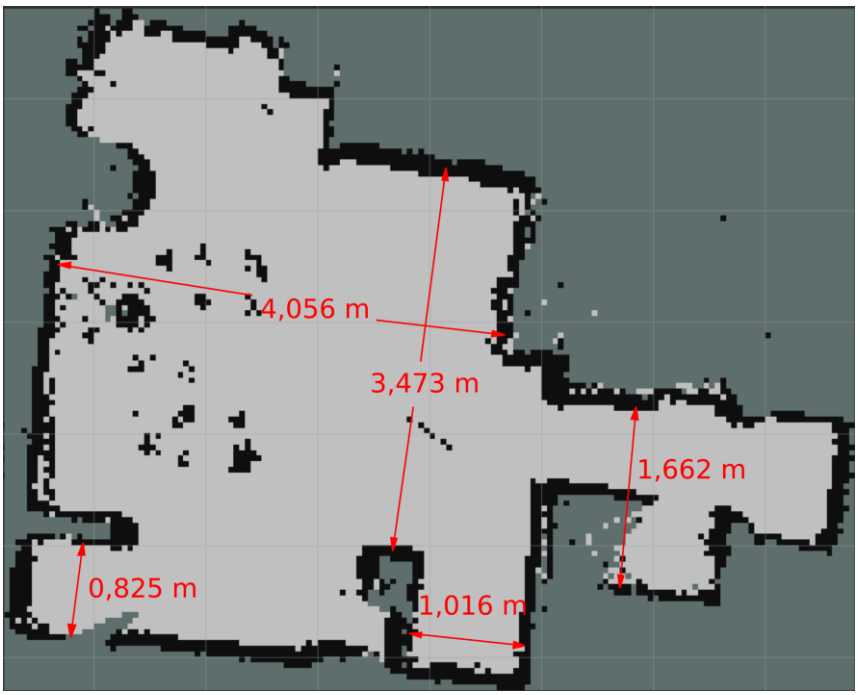


Fig. 45 - Mapa completo

Utilizando la herramienta de medición presente en rviz2 podemos medir las dimensiones del mapa. Comparándolas con el mapa del lugar observamos que son muy aproximadas a las reales.

### 2.4.7 Navegación autónoma.

Para la navegación autónoma utilizaremos el paquete *Nav2*. Dicho paquete es el sucesor con respaldo profesional de ROS Navigation Stack que implementa las mismas tecnologías que impulsan los vehículos autónomos. Este proyecto permite que los robots móviles, de casi cualquier clase o tipo, naveguen a través de entornos complejos para completar tareas. No solo puede moverse del punto A al punto B, sino que también puede realizar otro tipo de acciones como seguimiento de objetos y evasión de obstáculos. Nav2 es un framework de navegación de alta calidad y grado de producción en el que confían más de 100 empresas en todo el mundo.

La información esperada por *Nav2* para funcionar es, las transformaciones TF, el mapa, el archivo de configuración y cualquier información de sensores que pueda ser relevante para la navegación.

Como *Nav2* es un conjunto de muchas herramientas y cada una posee un archivo de configuración, la cantidad de opciones de configuración es muy grande. Por este motivo utilizamos como archivo de configuración el archivo de ejemplo provisto por el paquete. Los únicos cambios que hicimos a dicho archivo fueron los nombres de los tópicos que, en nuestro caso, eran diferentes.

Uno de los paquetes presentes dentro de *Nav2* es *Costmap 2D*. Dicho paquete implementa un mapa de costos basado en cuadrículas 2D para representaciones ambientales. El mapa de costos se utiliza en el planificador y el controlador para crear el espacio donde se moverá el robot y para comprobar si hay colisiones o áreas de mayor coste para negociar.

Un mapa de costo o *Costmap*, es un mapa que se crea a partir del mapa del entorno creado con SLAM. La diferencia entre el Costmap y la grilla de ocupación (Ocupancy Grid) que hablábamos anteriormente, es que el valor que tienen las celdas en el Costmap no representa que tan ocupada está, si no que representa el “costo” que tiene moverse a través de dicha celda. El costo lo establece qué tan cerca está la celda a una celda ocupada. Esto se debe a que, tanto la posición del robot, como la

posición de los obstáculos o de las celdas ocupadas, son estimaciones (de ahí el nombre Pose o position estimation). Por lo tanto, las celdas cercanas a obstáculos tienen un mayor costo ya que no se sabe con certeza la ubicación exacta.

Que una celda tenga más costo no quiere decir que el robot no vaya a circular nunca por esa zona, si no que a la hora de planear la ruta se tratarán de evitar las celdas con mayor costo. En el caso que el único camino posible sea circular por dichas celdas, la ruta pasará por ahí. De esta manera, podemos concluir que lo que hace el costmap es definir áreas donde es seguro moverse y áreas que deben, en lo posible, evitarse. En la siguiente imagen podemos observar un ejemplo de Costmap, vemos como las celdas cercanas a celdas ocupadas cambian de color, indicando un mayor costo.

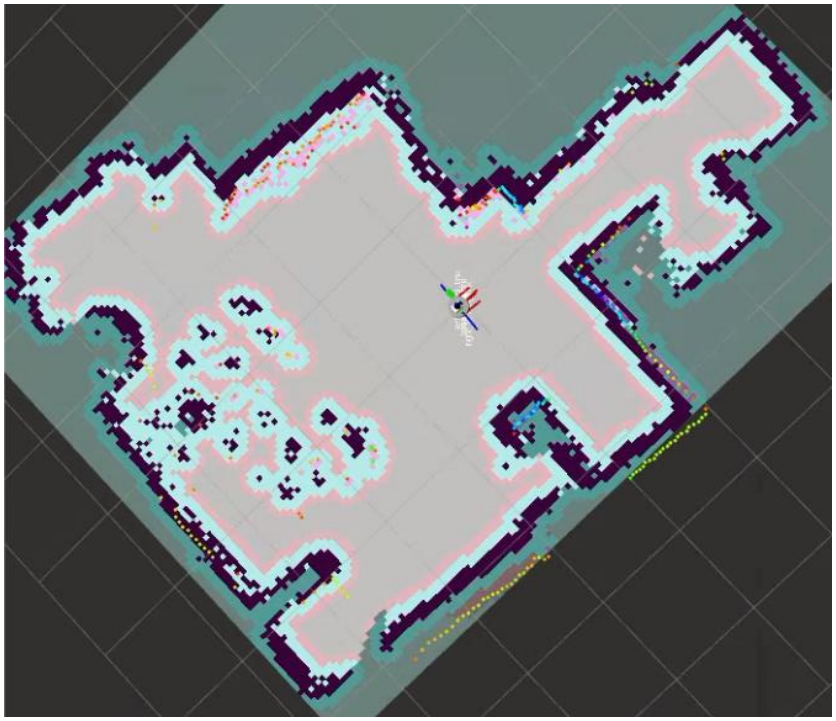

Fig. 46 - Ejemplo Costmap

A continuación, presentaremos una serie de imágenes que muestran el funcionamiento de *Nav2,* como colocamos la posición inicial, luego una posición objetivo y cómo se calcula la ruta que seguirá el robot para llegar al destino solicitado.

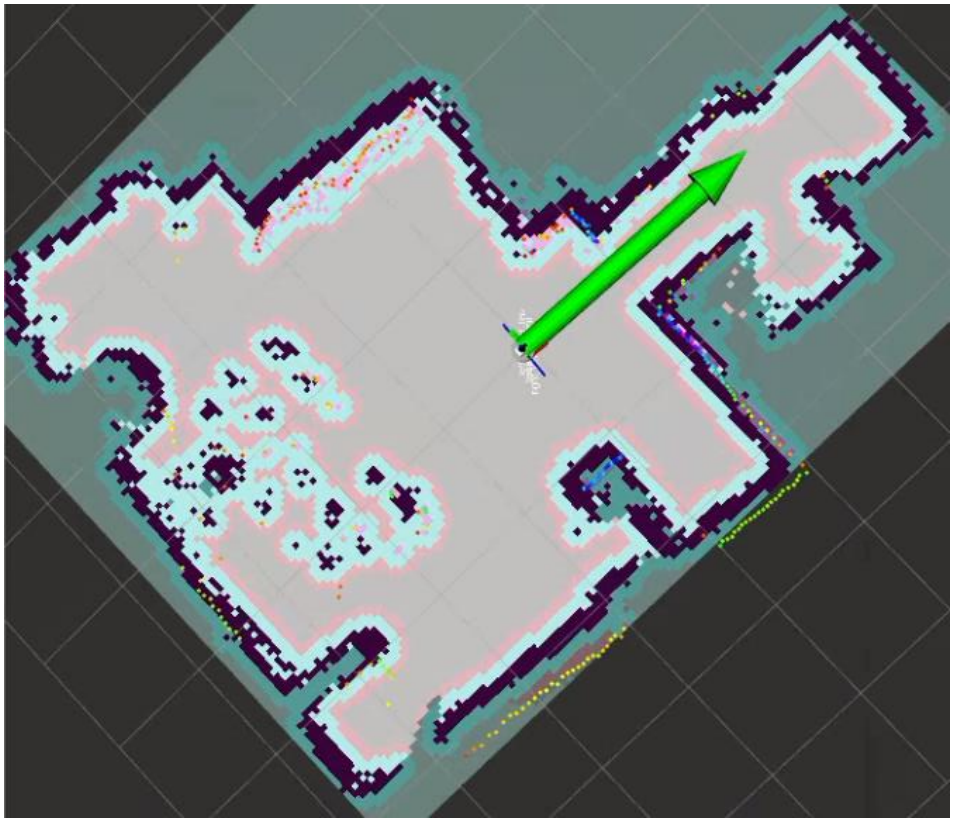

Fig. 47 - Posición inicial.

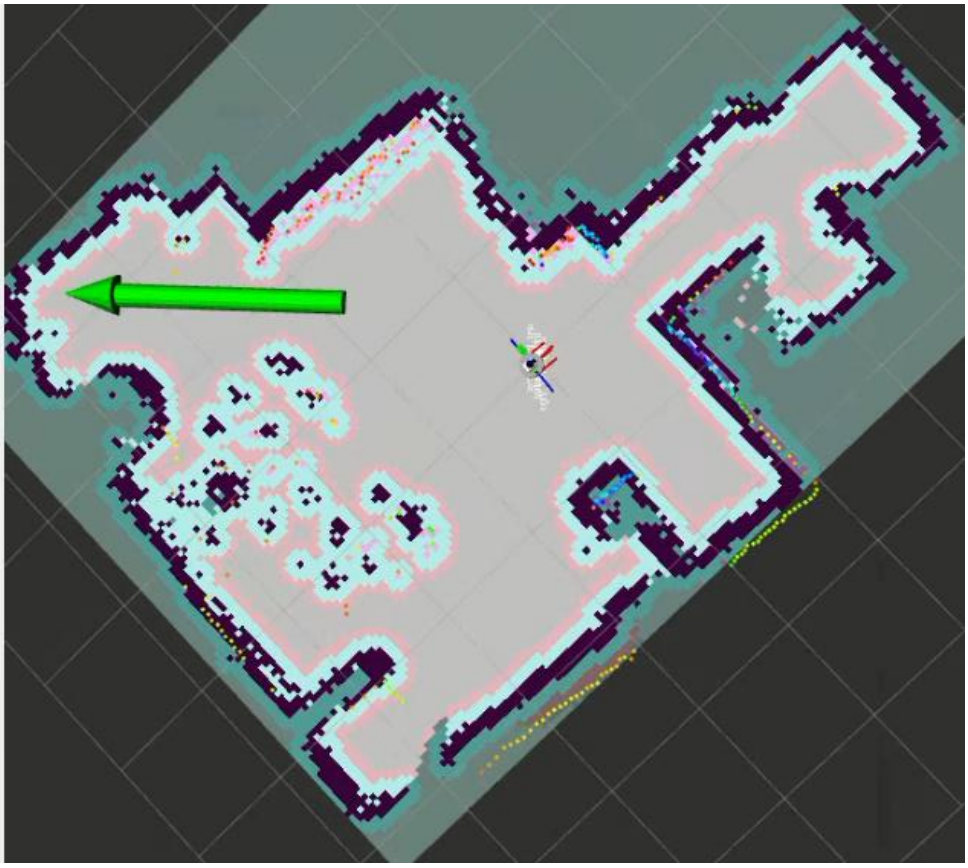

Fig. 48 - Primer punto objetivo.

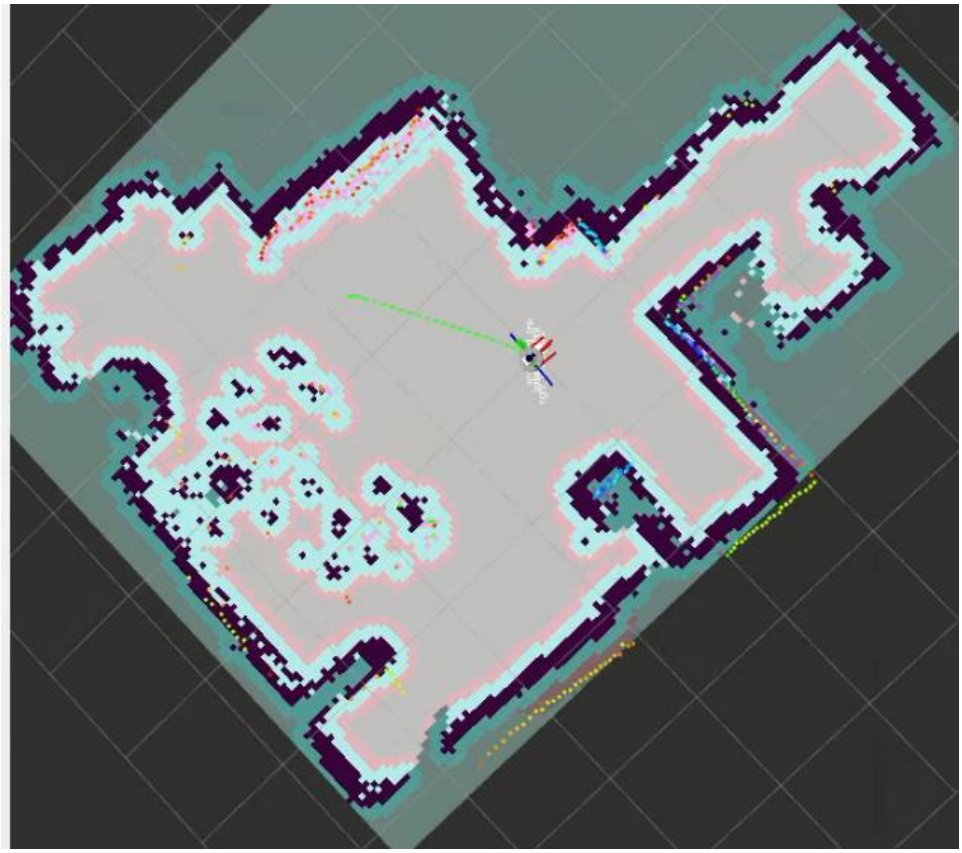

Fig. 49 - Ruta del robot.

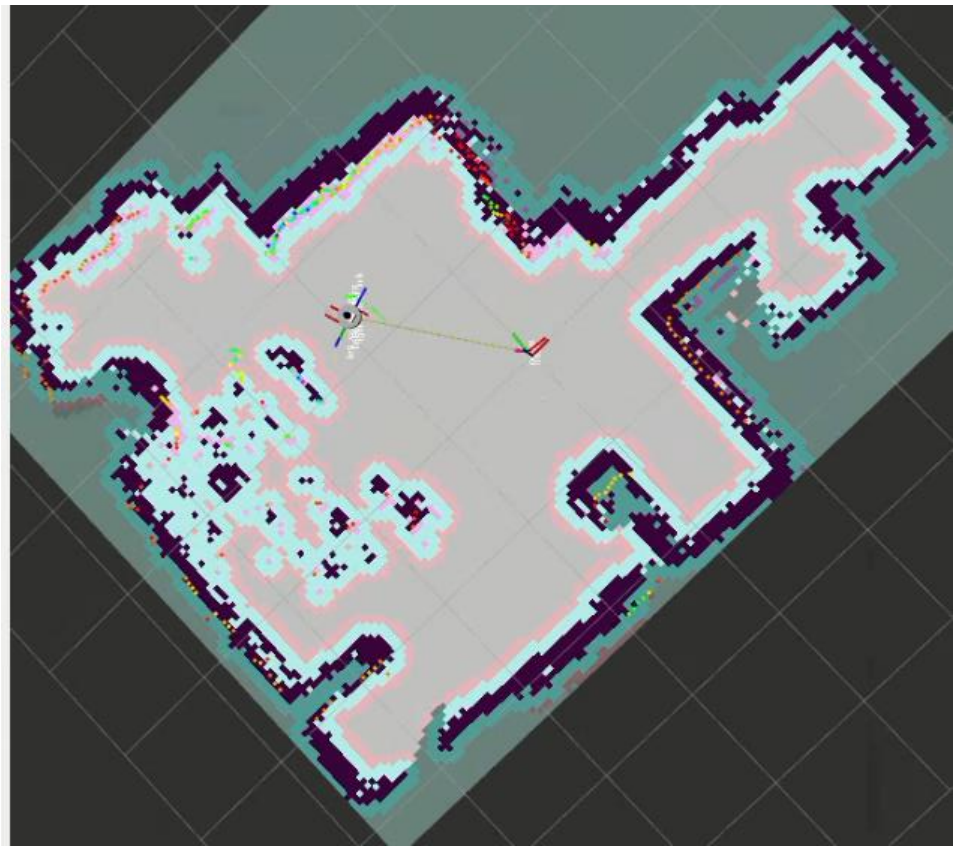

Fig. 50 - Llegada al objetivo.

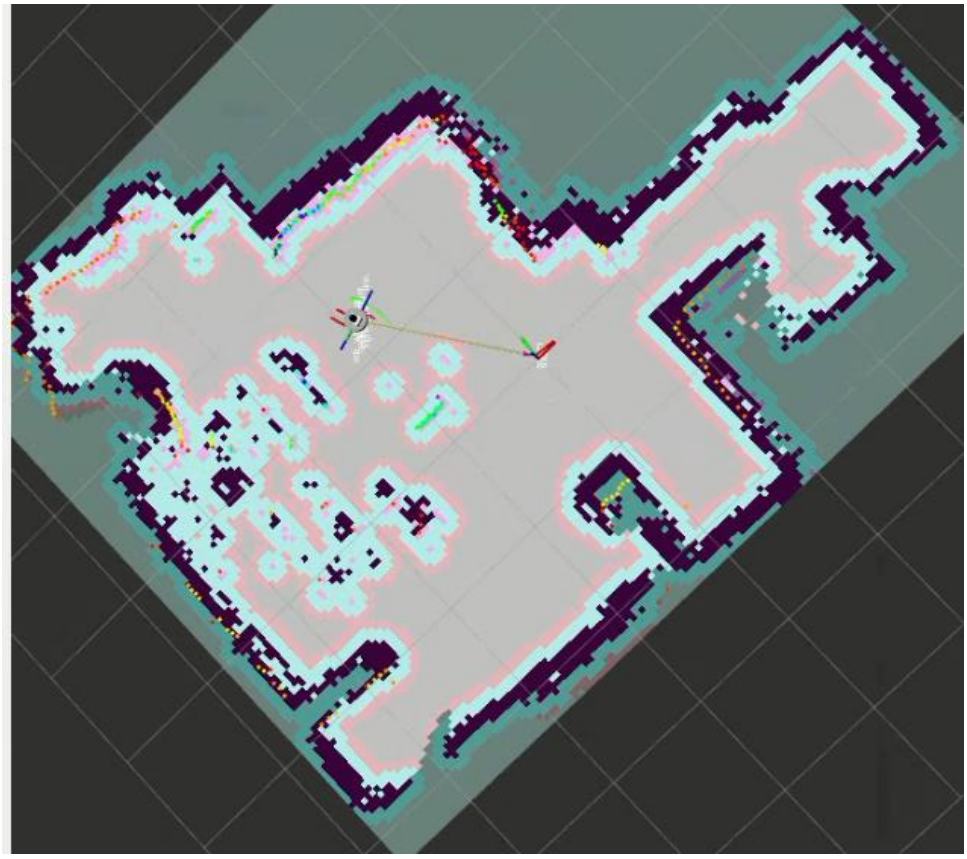

Fig. 51 - Colocamos un obstáculo.

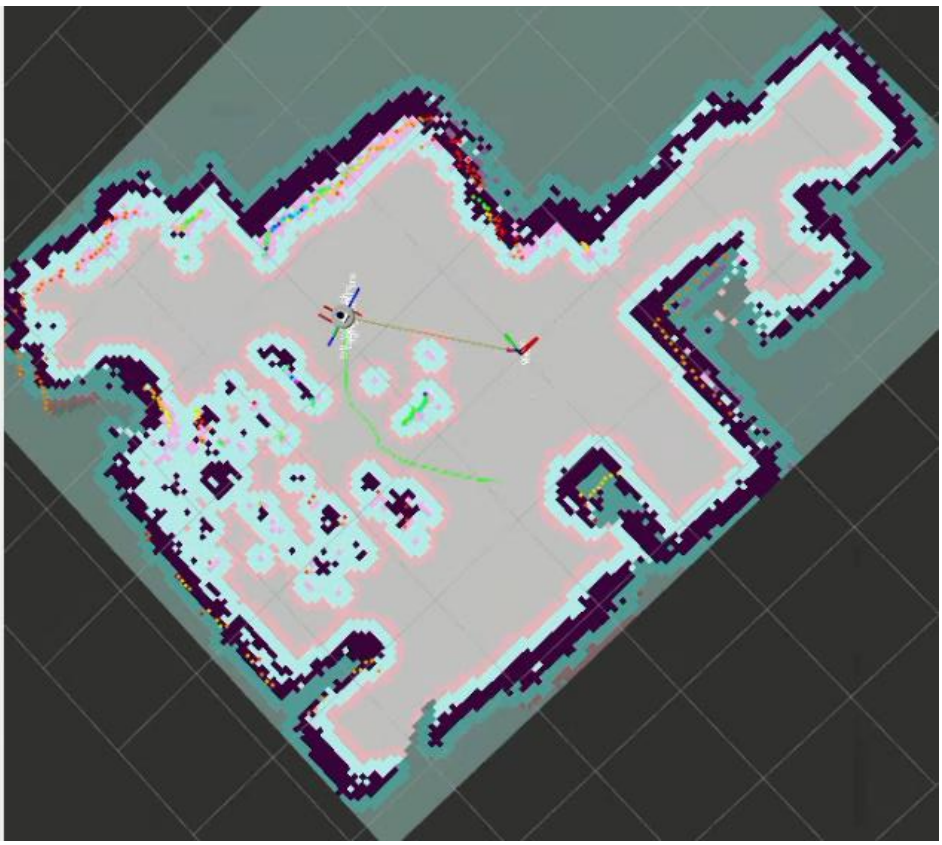

Fig. 52 - Nuevo punto objetivo.

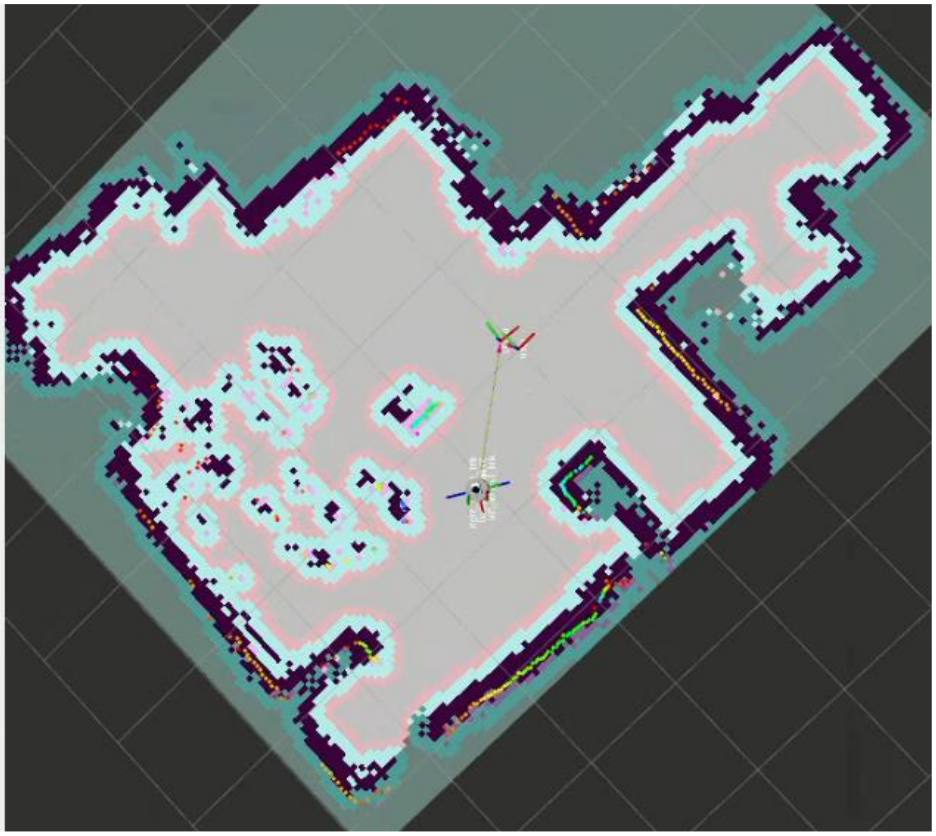

Fig. 53 - Llegada al punto objetivo.

En la serie de imágenes pudimos observar cómo *Nav2* calcula la trayectoria del robot para llegar a los puntos que fuimos solicitando y como, a su vez, se adapta ante nuevos obstáculos que no estaban presentes en el mapa original.

## Capítulo 3: Resultados

Si bien en los resultados presentados en cada una de las partes vemos un buen desempeño del robot, para llegar a ese punto tuvimos que ir probando diferentes configuraciones de cada uno de los paquetes ROS que utilizamos. Al final del presente proyecto, en el anexo, presentamos una serie de imágenes de cómo fue evolucionando y mejorando el mapeo SLAM.

También tuvimos diferentes problemas relacionados con el hardware. En un principio estábamos utilizando unas ruedas provenientes de un “Kit Robot para Arduino”. Dichas ruedas nos dieron muchos problemas de adherencia, lo que hacía que nuestra odometría sea muy inexacta. Esto nos obligó a diseñar nuestras propias ruedas.

Otro problema relacionado con el hardware fueron los pulsos por vuelta entregados por los encoders. Cuando configuramos la interfaz de hardware nos basamos en información proveniente del “datasheet” de los motores. Dicha información era errónea por lo que tuvimos que contar los pulsos con la ayuda del controlador para dar con la información correcta.

Una vez que obtuvimos un desempeño satisfactorio del robot dentro de nuestro lugar de pruebas, decidimos llevarlo a un entorno más grande y así poder testear nuevamente el mapeo. Aquí nos encontramos con una limitación importante. Como comentamos anteriormente, el paquete *slam_toolbox* calcula la posición respecto del mapa utilizando información del Lidar. Dicho Lidar tiene un rango de medición acotado (max 3.5m) y, como el nuevo entorno tiene dimensiones más grandes que 3.5m, nos comenzamos a topar con distintos problemas. Nos dimos cuenta que, si el lidar no aportaba información suficiente como para calcular el movimiento del robot, *slam_toolbox* suponía que el robot no se había movido, o se había movido muy poco. Esto provoca un desfase entre la posición calculada por *slam_toolbox* y la posición real del robot, haciendo que el mapa pierda sus dimensiones reales o, en el peor de los casos, pierda totalmente su forma.

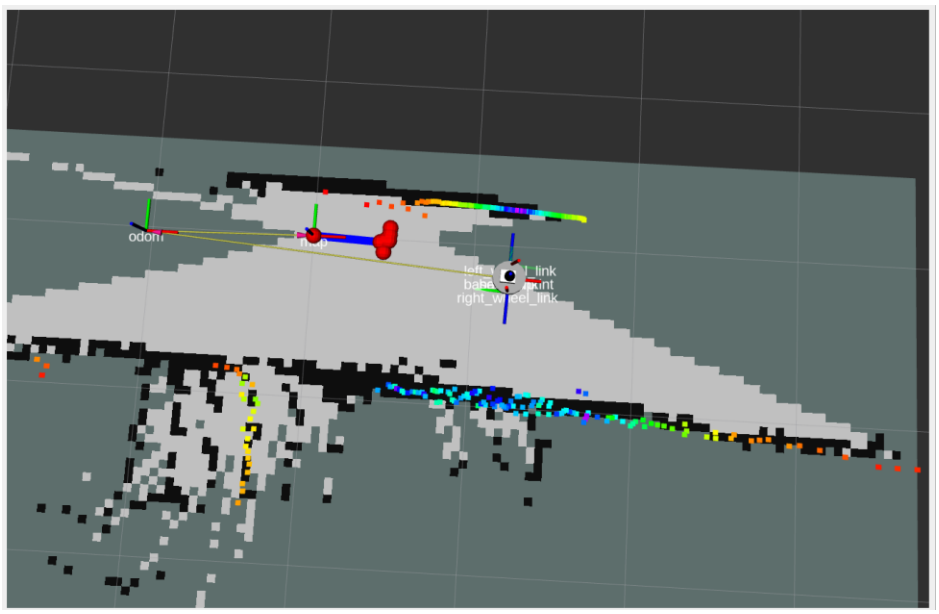

Fig. 54 - Primer Problema.

En la figura anterior se observa como *slam_toolbox* intenta recalcular varias veces la posición del robot (superposición de puntos rojos). Como la información del lidar no es suficiente (no existen puntos de medición en la dirección en la que el robot se

estaba moviendo), *slam_toolbox* cree que el robot se mantuvo en el lugar. Para intentar paliar esta situación, configuramos el segundo nodo de *robot_localization* (el que calcula la transformación map->odom mediante la información provista por *slam_toolbox* y la odometría) para que solo utilice la información de la odometría. Aquí es donde nos dimos cuenta que *slam_toolbox* calcula las dimensiones del mapa solo con información del Lidar, y no con la odometría del robot.

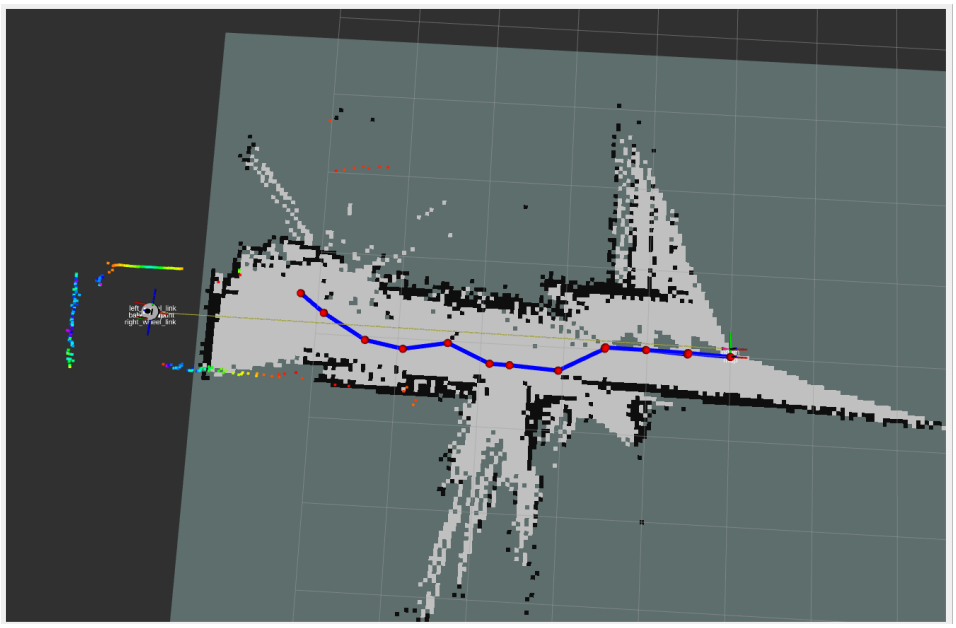

Fig. 55 - Segundo Problema.

En la figura anterior vemos como, al calcular la transformación map->odom con solo información de la odometría, la posición del robot termina saliendo del mapa. Esto nos indica que las dimensiones del mismo no son correctas.

# Capítulo 4: Discusión y Conclusión

El presente proyecto surgió con el objetivo de desarrollar una plataforma robótica capaz de moverse de manera autónoma. Para ello se seleccionó ROS como el framework de desarrollo por ser de código abierto y, a su vez, una de las herramientas más usadas en la industria.

El proyecto ha demostrado con éxito la capacidad de utilizar ROS como plataforma para desarrollar un robot autónomo capaz de realizar tareas de mapeo y localización simultáneas. Mediante la implementación de algoritmos de SLAM, nuestro robot ha sido capaz de navegar en entornos desconocidos, creando mapas detallados mientras se localiza dentro de ellos.

Durante el desarrollo enfrentamos varios desafíos, desde la selección de los componentes de hardware, el diseño del robot completo hasta el entendimiento de las diferentes herramientas y paquetes de ROS que utilizamos.

Si bien el desempeño de nuestro robot fue más que satisfactorio, entendemos el potencial de continuar mejorándolo. Para ello se podría considerar el cambio del sensor Lidar por uno de mayor rango, la utilización de otros sensores que aporten información relevante para el mapeo (como ser cámaras o Lidar’s 3D), o la utilización de otro paquete de SLAM diferente a *slam_toolbox.*

Creemos que nuestro proyecto ayudará a la investigación y desarrollo de otros robots más complejos aplicables a diferentes industrias. Al mismo tiempo, sienta las bases del conocimiento para futuros alumnos o desarrolladores que deseen realizar un proyecto similar o relacionado a la robótica en el marco de la Facultad Regional Paraná.

# Capítulo 5: Literatura Citada


1. ¿Qué es ROS?
   Link: https://openwebinars.net/blog/que-es-ros/
2. Introducción a ROS.
   Link:http://wiki.ros.org/es/ROS/Introduccion#:~:text=Siguiente-,Qu%C3%A9%20es%20ROS%3F,procesos%20y%20manejo%20de%20paquetes.
3. Localización y mapeo simultáneos.
   Link:https://es.wikipedia.org/wiki/Localizaci%C3%B3n_y_modelado_simult%C3%A1neos
4. ¿Qué es SLAM?
   Link: https://www.mathworks.com/discovery/slam.html
5. Robot de accionamiento diferencial.
   Link: https://en.wikipedia.org/wiki/Differential_wheeled_robot
6. Controlador PID.
   Link :https://es.wikipedia.org/wiki/Controlador_PID
7. ¿Cómo usar un encoder de cuadratura?
   Link:https://cdn.sparkfun.com/datasheets/Robotics/How%20to%20use%20a%20quadrature%20encoder.pdf
8. Implementación de un controlador PID.
   Link:https://www.youtube.com/watch?v=zOByx3Izf5U&ab_channel=Phil%E2%80%99sLab
9. Acerca de TF2.
   Link: https://docs.ros.org/en/foxy/Concepts/About-Tf2.html
10. ROS 2 Control.
    Link: https://control.ros.org/master/index.html
11. Conceptos de ROS 2 Control.
    Link:https://beta.articulatedrobotics.xyz/tutorials/mobile-robot/applications/ros2_control-concepts/
12. Driver LIDAR.
    Link: http://wiki.ros.org/hls_lfcd_lds_driver
13. Driver MPU9250.
    Link: https://github.com/hiwad-aziz/ros2_mpu9250_driver
14. Wiki Robot Localization.
    Link: https://docs.ros.org/en/melodic/api/robot_localization/html/index.html

15. ¿Qué es una grilla de ocupación?
    Link: https://automaticaddison.com/what-is-an-occupancy-grid-map/
16. Wiki Slam Toolbox.
    Link: https://github.com/SteveMacenski/slam_toolbox
17. Wiki Navigation 2.
    Link: https://navigation.ros.org/index.html

# Capítulo 6: Anexo

## 6.1 Mapas.

Fig. 56 - Mapa 1

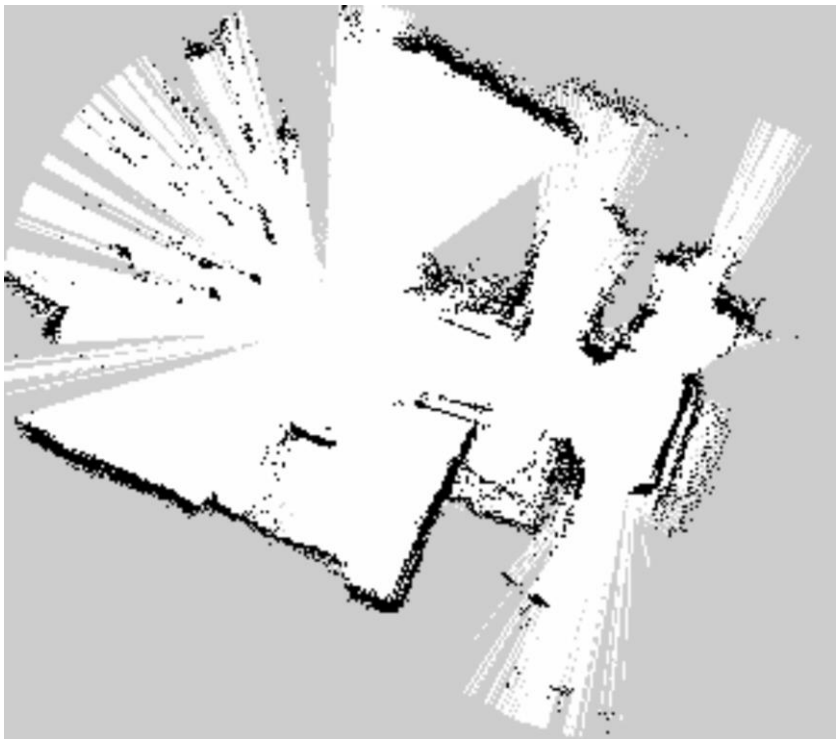

Fig. 57 - Mapa 2

Fig. 58 - Mapa 3

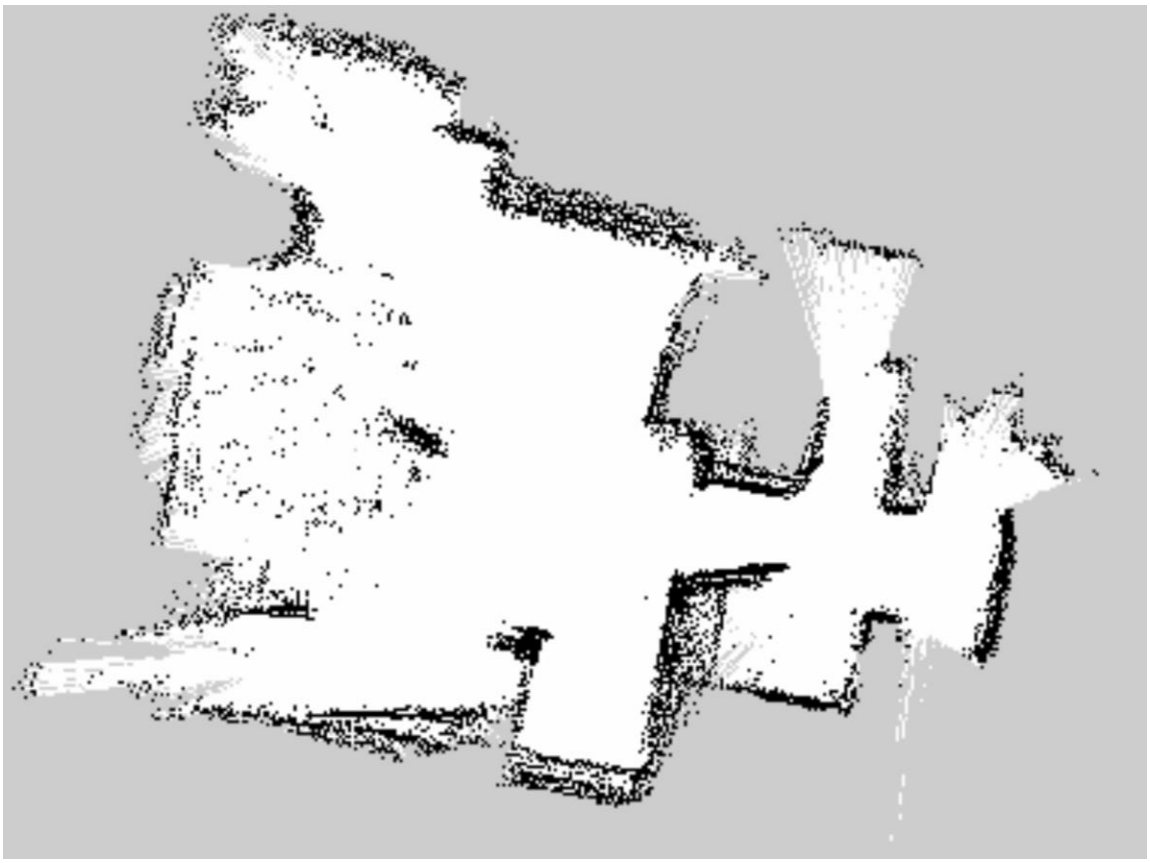

Fig. 59 - Mapa 4

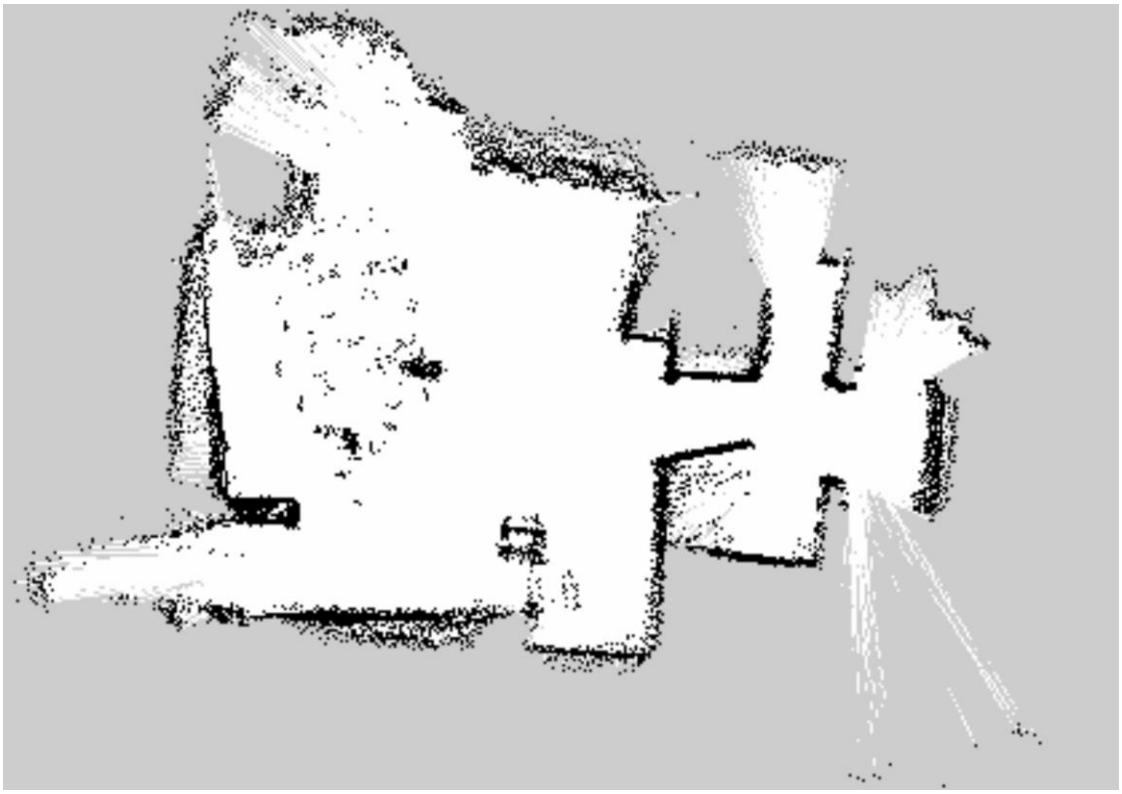

Fig. 60 - Mapa 5

Fig. 61 - Mapa 6

Fig. 62 - Mapa 7

Fig. 63 - Mapa 8

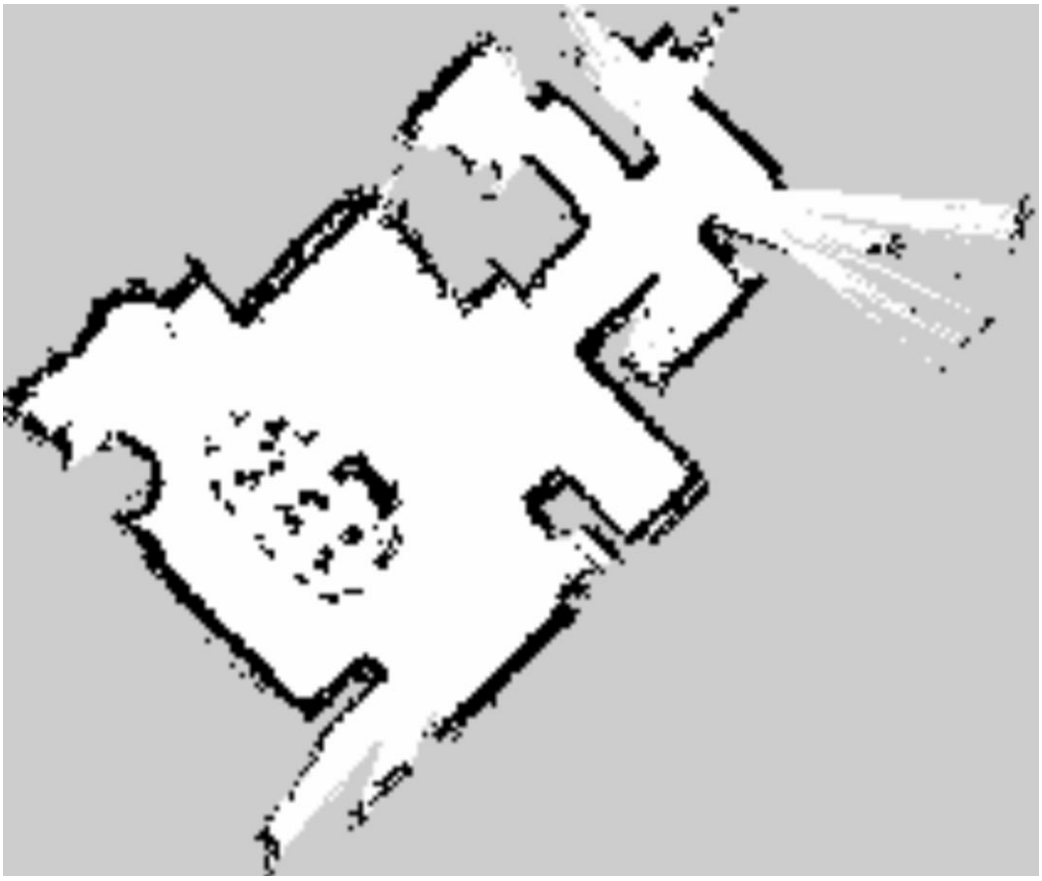

Fig. 64 - Mapa 9

Fig. 65 - Mapa 10

Fig. 66 - Mapa 11

## 6.2 Fotos robot armado.

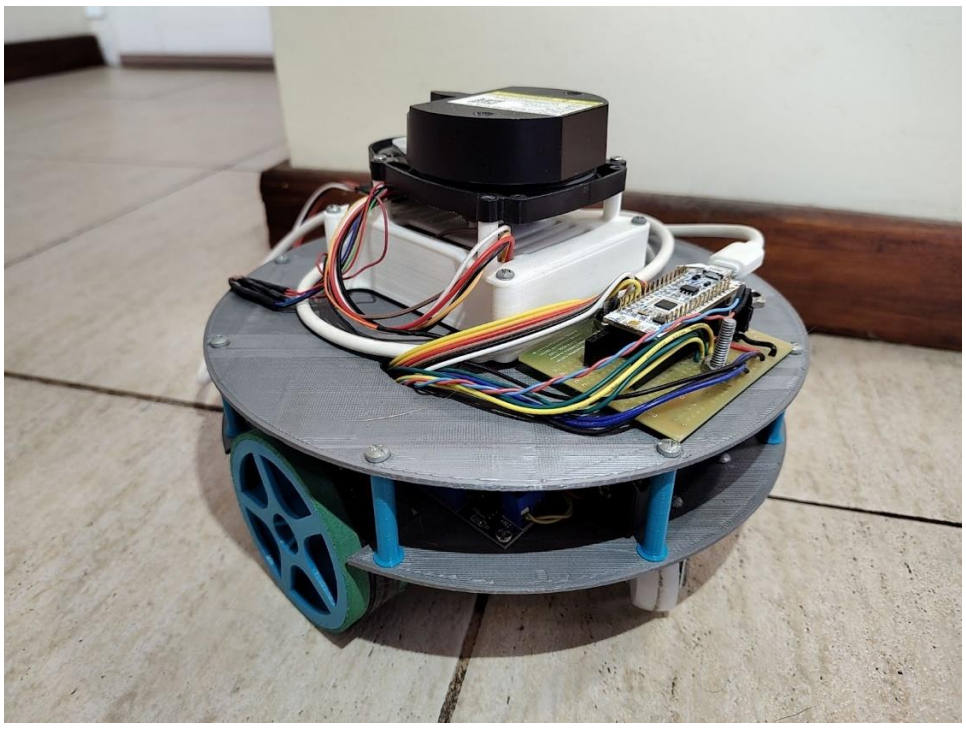

Fig. 67 - Robot Completo 1

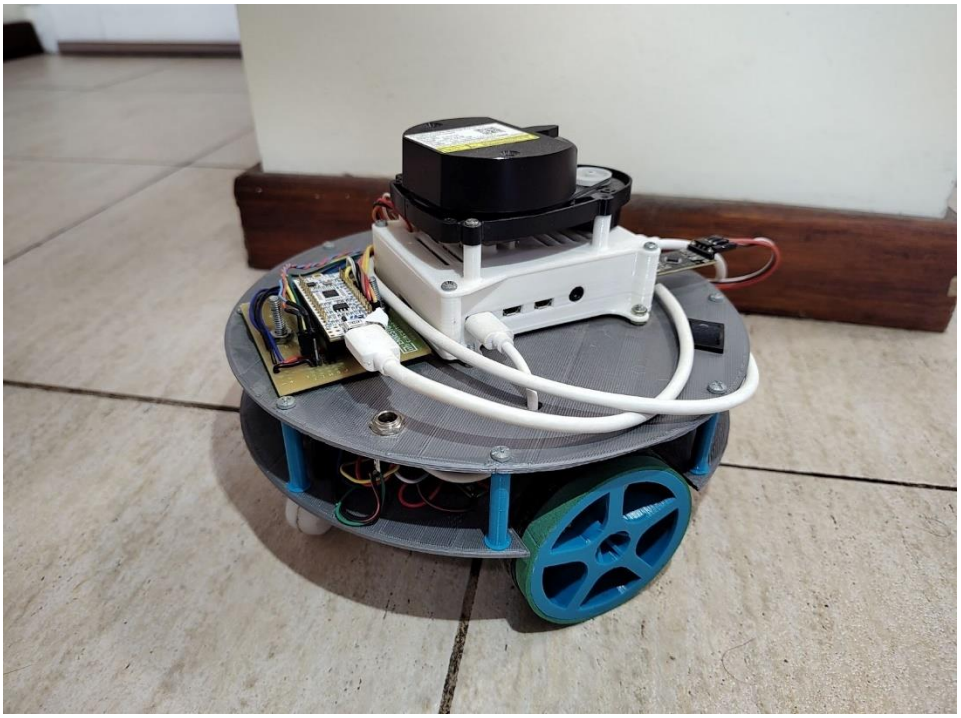

Fig. 68 - Robot Completo 2