

# Capybot - Robot diferencial autónomo con SLAM

Jorge Pérez Chávez

Escuela de Ingeniería y Ciencias  
Tecnológico de Monterrey, Jalisco  
Guadalajara, México  
A01023859@tec.mx

Luis Humberto Sánchez Vaca

Escuela de Ingeniería y Ciencias  
Tecnológico de Monterrey, Jalisco  
Guadalajara, México  
A01638029@tec.mx

Josué Rodríguez Mojica

Escuela de Ingeniería y Ciencias  
Tecnológico de Monterrey, Jalisco  
Guadalajara, México  
A01024035@tec.mx

## I. INTRODUCCIÓN

En el ámbito de la robótica, la localización y el mapeo simultáneos desempeñan un papel fundamental en permitir que los robots naveguen y tomen decisiones de manera autónoma en entornos desconocidos. Aprovechando esta capacidad, el presente informe se centra en la implementación de nuestro robot llamado Capybot, diseñado específicamente para abordar el desafío de medir la intensidad de la señal de WiFi en áreas desconocidas y crear mapas de calor que ayuden a mejorar la distribución de módems y repetidores. En este informe detallado, se describen los aspectos clave del diseño, la fabricación y la implementación del robot Capybot. Se abordan los diferentes componentes y algoritmos utilizados, como el filtro de Kalman para la estimación de la posición, el control de bajo nivel para el movimiento del robot, la utilización de la técnica de SLAM (Simultaneous Localization and Mapping) mediante el uso de Hector SLAM para generar mapas y proporcionar odometría y navegación autónoma, así como generación de trayectorias usando RRT y dos propuestas de exploración autónoma.

## II. JUSTIFICACIÓN DEL PROYECTO

### A. Contexto

El mapeo de señales WiFi se ha convertido en una herramienta muy útil debido a que los dispositivos inalámbricos son una parte integral de nuestras actividades cotidianas, y su rendimiento óptimo depende de tener una señal adecuada. Los mapas de calor de la cobertura de WiFi representan de manera visual la intensidad de la señal por medio de colores, dependiendo de la aplicación un color mostrará señales fuertes y otro color señales débiles o zonas muertas. Estos mapas son útiles para poder mejorar la cobertura de señal, ya que hay diversos elementos que pueden causar interferencias como paredes, puertas, muebles, techos, aparatos de radio e incluso señales vecinas de WiFi. Existen diversas aplicaciones que sirven para crear estos mapas de calor tanto para computadores como para celulares, los cuales cuentan con diversas herramientas de análisis posterior a hacer el mapeo, sin embargo no todas son de uso personal por lo que el costo de la aplicación puede elevarse considerablemente. Además estas herramientas requieren de contar con un plano del lugar, o de crear uno mediante la aplicación para que el mapeo sea realmente útil; una vez que se tiene cargado este mapa la aplicación requiere

de moverse a través del lugar con el dispositivo que se está usando para poder ir probando la intensidad de la señal y con esto generar el mapa de calor.

### B. Objetivo

El propósito del robot es funcionar como un facilitador de creación de estos mapas de calor WiFi y así se puedan hacer ajustes para mejorar la cobertura inalámbrica.

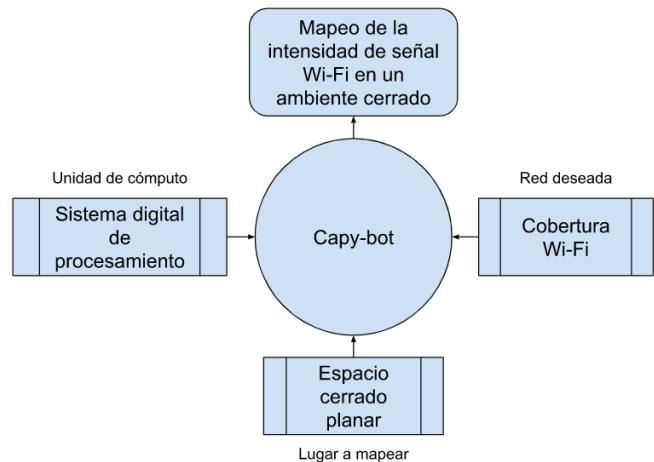


Fig. 1. Diagrama de funcionalidad

Originalmente, el proyecto estaba pensado para implementarse en el Puzzlebot, una plataforma de aprendizaje desarrollada por Manchester Robotics. Sin embargo, el motivo por el cual decidimos rediseñar la plataforma mecánica, es porque en semestres anteriores tuvimos demasiado ruido al trabajar con el Puzzlebot

Requerimientos de alto nivel:

- El robot tendrá la capacidad de navegar de manera autónoma por un cuarto plano.
- El robot tendrá la capacidad de detectar redes WiFi 802.11b 2.4Ghz.
- El robot contendrá piezas impresas pre-fabricadas o diseñadas para impresión en 3D. (Para evitar la manufactura de piezas complicadas)
- El robot generará un mapa de calor con las lecturas tomadas de la fuerza de la señal.

- El robot tendrá un ciclo de uso desde la batería de 10 mins mínimo.
- El subsistema de potencia debe de tener una manera de para todo en caso de emergencia.
- El subsistema de potencia deberá de tener protecciones en caso de que haya picos de corriente.
- El robot deberá de usar conectores que sean seguros y fáciles de conectar/desconectar entre sus componentes.

Limitaciones:

Presupuesto

- Los costos adicionales de componentes no deberán superar los \$2,000.
- El presupuesto podrá expandirse por gastos no previstos en un 50%.

Tiempo:

- El proyecto deberá quedar finalizado entre el 31/mayo y el 7/Junio del 2023.

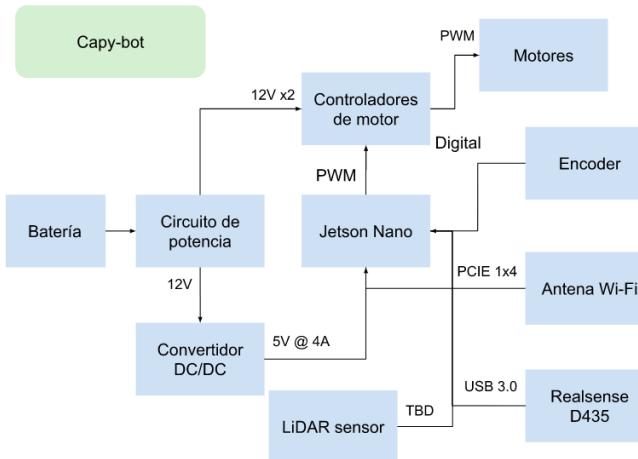


Fig. 2. Diagrama de bloques Capybot

Al terminar la definición de alcances del proyecto, se llegó al diagrama 3 de conexión entre nodos para desarrollar en el proyecto:

### III. DISEÑO Y CONSTRUCCIÓN

Comenzamos evaluando los recursos disponibles, como motores, microcontroladores, drivers para los motores, etc. También adquirimos materiales para el chasis, como perfiles de aluminio y MDF que podríamos cortar según nuestras necesidades. Se seleccionaron componentes clave para determinar el tamaño y la forma del chasis, incluyendo los motores, el microcontrolador, la batería y la tarjeta Jetson Nano.

Optamos por utilizar los motores "Yellow Jacket Planetary Gear" de la marca "goBilda". Estos motores cuentan con una caja de engranajes que proporciona un mayor torque, lo cual fue beneficioso para evitar limitaciones debido al peso y adaptarnos a nuestros recursos. Además, poseen un encoder de efecto Hall integrado del eje de salida. Asimismo, adquirimos los controladores de motor "1x15A Motor Controller" de la misma marca, que se controlan mediante señales PWM.

### Capybot ROS node graph

This is the initial (expected) ROS network between nodes and their topics

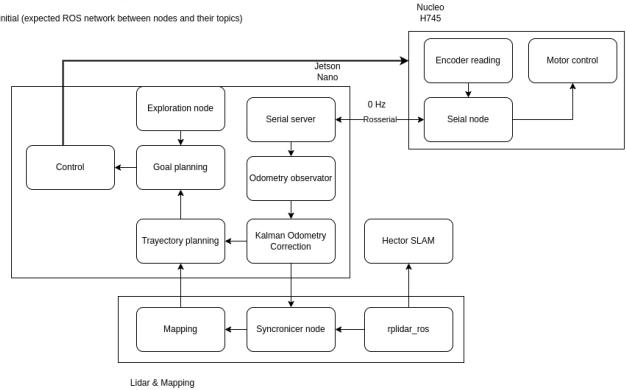


Fig. 3. Diagrama de nodos ROS en el Capybot

Respecto a la batería, seleccionamos una de la marca **DURAVOLT**, modelo D12-3.3. Que proporciona 12 voltios y es capaz de soportar altas cargas de amperaje (32 A). Este factor fue importante debido al tamaño de los motores y al hecho de que la Jetson puede llegar a consumir hasta 4 amperios. Calculamos una autonomía de entre 20 y 40 minutos, e incluso más dependiendo del uso, por lo que adquirimos 2 baterías para evitar retrasos mientras se recargan.

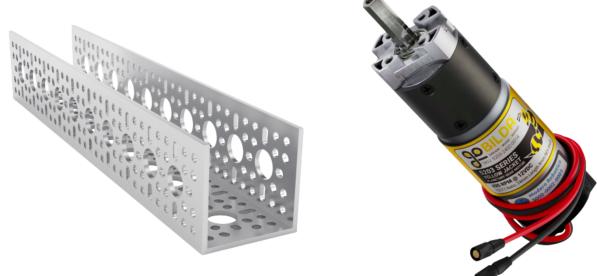
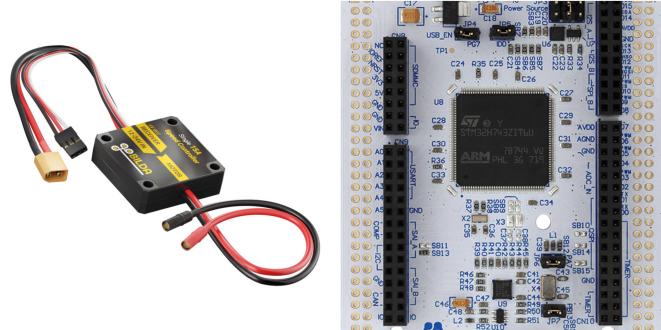


Fig. 4. Ensamblaje de las piezas modeladas en 3D

Con todos estos componentes en mente, comenzamos el diseño del chasis. Para sostener los motores, utilizamos perfiles de aluminio y añadimos un juego de engranajes para colocarlos de forma perpendicular a las ruedas y así obtener más espacio. Luego, fue necesario crear un diagrama para cortar el MDF

que unía las dos piezas. Utilizamos Onshape y aprovechamos la posibilidad de obtener modelos 3D de casi todos nuestros componentes para asegurar una precisión óptima. Optamos por utilizar Onshape porque es una herramienta en línea, lo que facilitó el intercambio de diseños entre los miembros del equipo sin depender de una única computadora o sistema operativo. Además, el no tener cargos por uso nos permitió reducir costos. También diseñamos llantas personalizadas para nuestro robot, con el mismo diámetro que las llantas Mecanum 70A Durometer Bearing Supported Rollers de 96 mm, las cuales actuaban como ruedas giratorias o "ruedas locas". Estas llantas se dividieron en dos partes, los rines y las llantas propiamente dichas, utilizando PLA y TPU, respectivamente. Sin embargo, debido a la rigidez y la resistencia a altas temperaturas, reemplazamos el PLA por PETG. Imprimimos todas estas piezas en 3D para una fabricación rápida y sencilla.

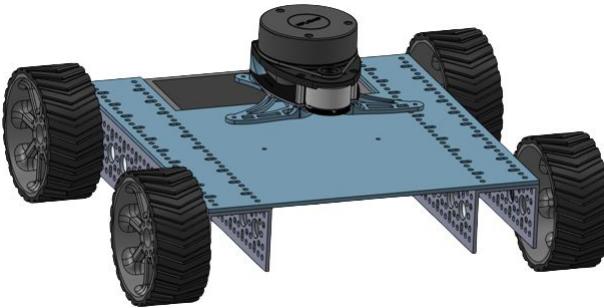


Fig. 5. Ensamblaje de las piezas modeladas en 3D

Más tarde, decidimos la distribución de los componentes. En la parte inferior, colocamos varios elementos, incluyendo la placa Núcleo, un módulo regulador Step Down 300W 9A XL4016 para reducir el voltaje y alimentar la Jetson. Este modelo mantiene constante el voltaje de salida, independientemente de las variaciones en el voltaje de entrada, y puede soportar hasta 9 amperios, brindando seguridad a la Jetson. También incluimos una PCB que diseñamos para proporcionar una mayor seguridad a todo el sistema. Aunque su diseño es sencillo, cumple su función y cuenta con 3 fusibles de 4 amperios para proteger el sistema en su totalidad. Diseñamos esta PCB utilizando Eagle y la encargamos a JLCPCB. Por último, los controladores de motores se atornillaron a los lados de los perfiles de aluminio.

En la parte superior del chasis, colocamos la Jetson en una posición central para facilitar el acceso a todos sus puertos. Utilizamos separadores y un soporte diseñado para colocar el LiDAR por encima de la Jetson.

#### IV. IMPLEMENTACIÓN DE DRIVE-TRAIN

El Puzzlebot es un robot diferencial, y consideramos que cambiar el drive-train no era viable debido a las limitaciones de tiempo, pues no sólo cambia la construcción del sistema, sino también la cinemática.

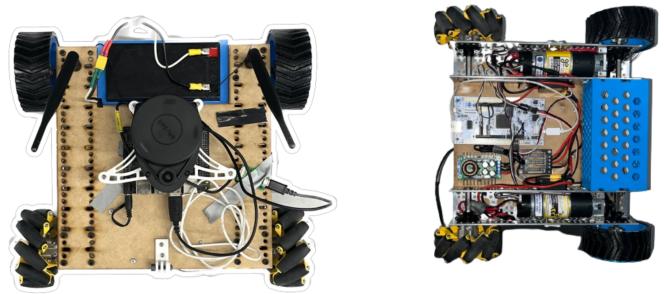


Fig. 6. Vista inferior y superior del CapyBot

##### A. Selección de módulo de procesamiento secundario

En primera instancia consideramos ejecutar todos los módulos de control directo en la Jetson, sin embargo después de investigar cómo funciona el GPIO en el micro, nos percatamos que no sería posible. Esto se debe a que el MCU no cuenta con relojes extras, y nosotros necesitábamos 4 (2 encoders y 2 generadores PWM). Debido a esto decidimos pasar a la Núcleo H745ZI-Q. Esta tarjeta cuenta con 12 relojes adicionales a los de bajo consumo de energía, además de que tenemos experiencia con el ambiente de programación obtenida en semestres anteriores.

##### B. Comunicación con ROS

Una vez seleccionada la tarjeta, procedimos a la implementación del sistema. Para facilitar la integración de ROS con la Núcleo, decidimos utilizar *rosserial*. Esto nos consumió cerca de una semana, ya que no había una guía clara sobre como hacerlo, y debido a que la Núcleo tiene un procesador ARM, no podíamos hacer la compilación directamente en una arquitectura x86. Más adelante, tuvimos que resolver problemas de compilación por la mezcla de C y C++ en un mismo proyecto. Esto porque las librerías de HAL (Hardware Access Layer) de STM, están escritas en C, pero se llaman en las clases de C++, y viceversa. Una vez que esos imprevistos fueron resueltos procedimos a la implementación del drive-train. Para ello se ocuparon dos variables clave, la resolución del encoder, y los rangos de PWM necesarios para controlar los motores.

Resolución encoder	537.7 ticks
Pulso PWM mínimo	1050 $\mu$ s
Pulso PWM máximo	1950 $\mu$ s

##### C. Lectura de encoders

Para la configuración de cada encoder, se utilizó un timer con entrada externa por dos canales, esto se debe a que el encoder de cuadratura genera dos señales; el sentido del encoder depende de su desfase; y donde cada flanco de subida representa un tick. Esto nos permitió configurar el encoder mode, donde automáticamente se incrementa/decrementa el valor del contador dependiendo del sentido en que gira el

motor. Además, configuramos el ARR a 538, ya que esto representaría una revolución de la llanta.

Contar los ticks del encoder es sólo el primer paso para obtener su velocidad. Debido a que la velocidad implementada en el modelo cinemático requiere que las llantas tengan unidades de rad/s. Para calcular la velocidad angular de nuestras llantas se ocupó fórmula mostrada en la ecuación (1).

$$\omega = \frac{2\pi \Delta Ticks * 1000}{encoderRes * freq} \quad (1)$$

Donde  $freq$  representa la frecuencia a la que se lee el valor de los ticks. Esta fue determinada por varios factores. El principal fue la frecuencia de Nyquist. Este dice que para poder muestrear una señal con fidelidad es necesario muestrear por lo menos al doble de la frecuencia a la que corre la señal deseada. Debido a que la velocidad máxima de nuestro motor es de 5.5 rev/s, la teoría decía que debemos muestrear a 10Hz. Sin embargo terminamos dejándolo en 50 Hz. Para corroborar la lectura de nuestro sistema, utilizamos un tacómetro para calibrar nuestro algoritmo. Pero, la lectura de la velocidad angular nos presentó un problema principal; la consistencia de la lectura. Este problema era originado por dos cosas. El cálculo de  $\Delta ticks$ , planteado como  $tick_{t-1} - tick_t$  funciona si los ticks son incrementales. Pero eso puede llegar a generar un overflow en memoria, por lo que una vez que la cuenta del reloj llega al valor de ARR, se regresa a 0. Esto genera que cada que se logra una revolución, se invierte el sentido y la magnitud de la medición. Esto fue solucionado de manera simple con la ecuación (2).

$$if(abs(\Delta tick) > ticksPerRevolution) \\ tick_t = tick_t - ticksPerRevolution \quad (2)$$

El segundo es originado por la frecuencia de lectura. En un inicio intentamos una frecuencia de 200Hz, pero a velocidades bajas nos generaba errores de sobre-muestreo. De igual manera, 10Hz limitaba la velocidad. A la cual podemos correr los nodos de odometría, donde buscamos alcanzar tiempos de muestreo cortos para obtener una mejor estimación. Es por esto que establecimos al final la frecuencia de 50 Hz.

#### D. Control de actuadores

Por otro lado, para generar los pulsos de PWM se configuró un reloj con entrada de `SYSCLOCK`. Debido a que la frecuencia de este reloj es de 75 MHz, seleccionamos un prescaler de 74, para que el reloj incrementara cada  $1\mu s$ . Para variar el ciclo de trabajo, simplemente se alteraba el CCR1, ajustándolo con la fórmula de la ecuación (3).

$$CCR1 = \frac{(x - in_{min}) * (out_{max} - out_{min})}{(in_{max} - in_{min})} + out_{min} \quad (3)$$

Donde las variables representan:

Variable	Significado
$out_{min}$	Pulso PWM mínimo
$out_{max}$	Pulso PWM máximo
$in_{min}$	Velocidad mínima rad/s
$out_{min}$	Velocidad mínima rad/s
$x$	Velocidad deseada

Una vez que logramos controlar las velocidades de manera independiente de cada motor y tuvieron la oportunidad de probar el drive-train, surgió un imprevisto extra. Pues a pesar de tener una misma referencia, las llantas no giraban la misma velocidad. Para solucionar esto, tuvimos que implementar un control a bajo nivel para cada motor. Debido al uso de los motores, el problema del control como lo habíamos planteado era el error de estado estacionario. Por lo que decidimos implementar un control PI.

Para diseñar este control PI, partimos del entendimiento que nuestro error viene dado por la ecuación (4).

$$e = V_{ref} - V_s \quad (4)$$

Donde  $V_s$  es la velocidad del sistema y  $V_{ref}$  la velocidad deseada. Dado que buscamos que el error tienda a 0, se analiza su dinámica expresada en la ecuación (5).

$$u = -ke \\ \dot{e} \rightarrow 0 \\ \dot{u} = -kV_{ref} \quad (5)$$

Lo anterior nos indica que el estado en que nuestro control actúa es la aceleración, por lo que si buscamos controlar la velocidad debemos de integrar el resultado del control, y de esta manera estaremos actuando en el espacio deseado.

Conforme avanzamos en la implementación, para hacer que ambos motores girasen como se había planteado en el modelo cinemático, negamos la referencia recibida por un motor, así como invertir el sentido de lectura siempre que leyéramos del mismo. Sin embargo, conforme los nodos dependientes de esta inversión fueron creciendo, empezaron a generar problemas por omisiones de nuestra parte. Por esto mismo decidimos crear un método para invertir el sentido del motor, donde, el error a la referencia es quien tiene que ser negado, pues si no el sistema se desestabiliza.

Finalmente, para facilitar la implementación de ambos requerimientos, creamos una clase para cada uno de los módulos:

## V. MODELO MATEMÁTICO ROBOT DIFERENCIAL

### A. Espacio de configuración

El siguiente paso para controlar el robot es necesario conocer a priori el espacio sobre el que se va a controlar. En este caso buscamos controlar  $x, y, \theta$  del robot, por lo que hay 3 DOFs. Por eso mismo la configuración del sistema se ve en la ecuación (6).

$$q = \begin{bmatrix} x \\ y \\ \phi \end{bmatrix} \quad (6)$$

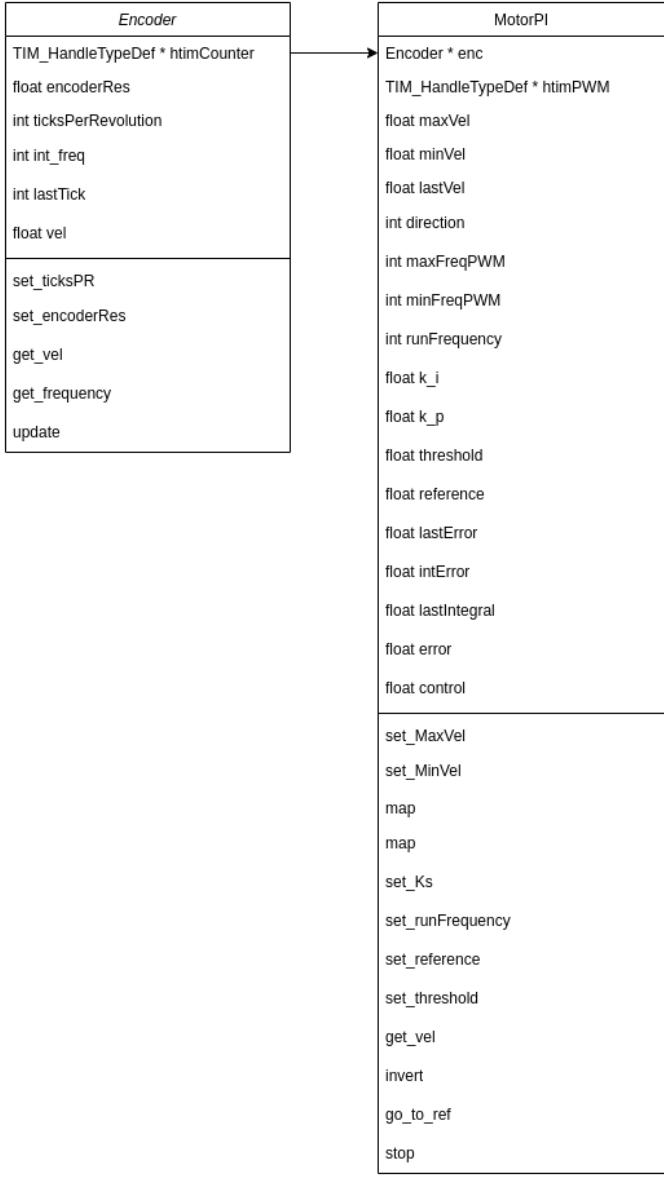


Fig. 7. UML diagram for implementation on STM32

### B. Marco de referencias

El marco de referencias es determinado por la Fig. 8.

### C. Cinemática del sistema

Como se observa en la figura 8, el robot está compuesto por llantas holonómicas, y no holonómicas, pero utilizamos el modelo cinemático de un robot no holonómico debido a que las omniwheels no están actuadas y sólo fungen como estabilizadores mecánicos. Hay dos motivos por los cuales es relevante conocer la dinámica del sistema, el primero es controlar el sistema percé, pero también estimar su estado dados las entradas de control, para generar odometría del robot. Dicho lo anterior, la cinemática del modelo se ve representada por la ecuación (7)

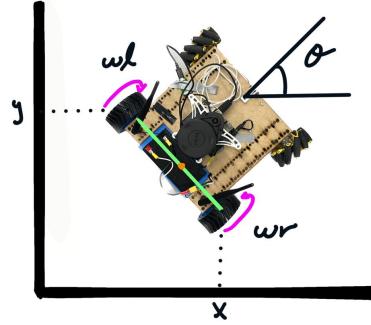


Fig. 8. Marco de referencias Capybot

$$\dot{q} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \frac{r}{2} \cos(\phi) - \frac{hr}{d} \sin(\phi) & \frac{r}{2} \cos(\phi) + \frac{hr}{d} \sin(\phi) \\ \frac{r}{2} \sin(\phi) + \frac{hr}{d} \cos(\phi) & \frac{r}{2} \sin(\phi) - \frac{hr}{d} \cos(\phi) \\ \frac{r}{d} & -\frac{r}{d} \end{bmatrix} \begin{bmatrix} w_l \\ w_r \end{bmatrix} \quad (7)$$

Hemos de mencionar que este sistema no es lineal en tiempo continuo. Por es necesario verificar su controlabilidad. Bajo la definición del teorema de rango de Kalman, donde se requiere que haya un mayor número de actuadores que estados, el sistema no sería controlable. Sin embargo, al realizar un par de conjeturas es posible obtener un controlador. En el análisis de control de sistemas no lineales, existen tres condiciones de controlabilidad, "global controllability, small-time local controllability, and small-time local accessibility". STLC (Small-time local accesibility) por sus siglas en inglés nos interesa debido a que contiene todo el espacio dimensional, y su configuración inicial. Por lo que la planta puede ser representada por 8.

$$\dot{x}(t) = Ax(t) + Bu(t)$$

$$y(t) = Cx(t) + Du(t) \quad (8)$$

Donde  $x(t)$  representa nuestro vector de estados y  $u(t)$  la salida producida por el controlador. La matriz  $A$  la dinámica del sistema,  $B$  los actuadores en el sistema,  $C$  representa  $A^{-1}$  y  $D$  las ganancias para obtener una salida deseada.

### D. Control implementado

Previo a la ejecución de este control, es necesario resaltar un par de cosas. Al observar detalladamente la estimación de nuestro sistema. Notése que todos los términos de la matriz (7), son fraccionados por  $d$ ; por lo que es imprescindible que  $d \neq 0$ . Además de eso, no es de nuestro interés controlar en determinada pose, podemos simplificar el modelo para reducir su complejidad.

$$q, q_d = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$e = q_d - q$$

$$u = A^{-1}(q_d - Ke) \quad (9)$$

Sin embargo, debido a las restricciones de tiempo en el proyecto no nos fue hacer un algoritmo de generación de trayectorias, por lo que se asume  $q_d = 0$ .

Tras muchas iteraciones para lograr el comportamiento deseado en el drive-train, llegamos a los valores de:

$$K = \begin{bmatrix} 0.12 & 0 \\ 0 & 0.12 \end{bmatrix} \quad (10)$$

De igual manera, saturamos la velocidad de cada llanta a 8 rad/s, para evitar accidentes con la plataforma, ya que su velocidad máxima es de 32 rad/s.

## VI. FILTRO DE KALMAN

Idealmente, la estimación dada por los algoritmos de Dead-reckoning es muy cercana al ground-truth. Sin embargo, eso nunca es verdad; pues se desprecian muchos factores externos como drift, fuerzas externas, y ruido en la lectura de sensores. El filtro de Kalman busca proveer de una estimación de estado óptimo de un sistema lineal. Consigue esto generando una relación directa entre el estado del sistema y la lectura de cada sensor, por medio de una etapa de predicción y corrección. Es capaz de filtrar, suavizar y predecir el comportamiento de una señal.

### A. Observabilidad del sistema

Previo a implementar el filtro de Kalman en un sistema es necesario conocer si este es observable. Eso refiere a que podemos conocer todos los estados del sistema sin tener una medición directa de cada estado. Esto es relevante, ya que para la estimación de odometría necesitamos mediciones en  $\dot{x}$ ,  $\dot{y}$ ,  $\dot{\theta}$ . Sin embargo no se incorporaron sensores para capturar estos estados. La observabilidad del Filtro de Kalman es de utilidad, ya que nos permite estimar estas mediciones. Otra característica del filtro de Kalman es que, debido a que la planta se retro-alimenta, se obtiene un ciclo de control sobre el error. Lo que nos permite alterar la taza de convergencia al estado real ante los distintos sensores.

### B. Kalman, acercamiento estocástico

Debido a que tanto el modelo de la planta, como nuestros sensores cuentan con defectos, y estos provocan ruido en la estimación del sistema, el filtro de Kalman utiliza las matrices  $P$ ,  $Q$  y  $R$  para rastrear la matriz de covarianza de error en la predicción, matriz de covarianza del ruido del modelo y matriz de covarianza de ruido en sensores respectivamente. De manera imprescindible, al suponer que todo el ruido en el sistema es descrito por una distribución Gaussiana, así como que nuestro modelo es lineal, se mantendrá un modelo donde el ruido sigue siendo Gaussiano al multiplicar estas matrices, incluso a pesar de ser integrado a través del tiempo.

$$\dot{x} = F\hat{x}BU + PH^T R^{-1}(Z - H\hat{x})$$

$$\dot{P} = FP + PF^T + GQG^T - PH^T R^{-1}HP \quad (11)$$

Debido a que el Filtro de Kalman sólo funciona con sistemas lineales, y nuestro modelo no lo es, tuvimos que linearizar el sistema bajo la premisa de que en ángulos pequeños,  $\cos 0 = 1$  y  $\sin 0 = 0$ . Por lo que obtenemos la ecuación (12)

$$F = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{hr}{d} & \frac{hr}{d} \\ \frac{r}{d} & -\frac{r}{d} \end{bmatrix} \quad (12)$$

De manera similar, definimos como identidad la matriz  $Q$  ya que el ruido del modelo debe ser relativamente constante e incremental y  $H$  porque como la entrada de estados es un sensor virtual generado por la odometría, somos capaces de leer todos los estados de la planta. Adicionalmente, como en este momento no es de nuestro interés predecir, ya que el control se ejecuta a una mayor frecuencia que la estimación de Kalman, definimos la matriz  $P$  como una matriz nula. El proceso para determinar los valores de la matriz  $R$ , que es característica para cada sensor fue de igual manera, prueba y error hasta que el filtro se comportara de manera deseada.

$$R_{odom} = \begin{bmatrix} 0.5 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.0 \\ 0.0 & 0.0 & 0.8 \end{bmatrix} \quad (13)$$

$$R_{visualOdom} = \begin{bmatrix} 0.5 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.0 \\ 0.0 & 0.0 & 0.3 \end{bmatrix} \quad (14)$$

### C. Resultados

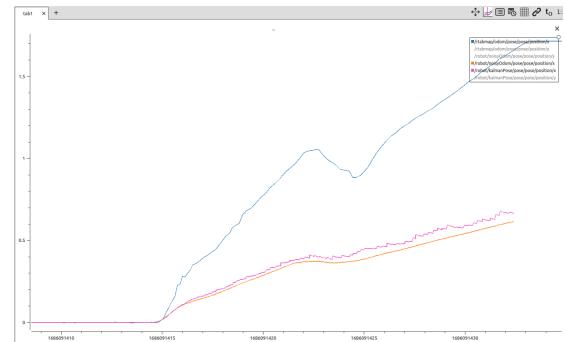


Fig. 9. Filtro de Kalman en X

Como se observa en la gráfica 10 y 9, la implementación del filtro nunca tiende a acercarse a las mediciones de la odometría visual. Eso se debe a que en los mapas donde probamos la odometría visual no tenían suficientes características para que nos diera mediciones precisas, incluso si el robot avanzaba y giraba lento, la odometría visual se perdía de manera constante. Posteriormente mencionamos cómo solucionamos este problema.

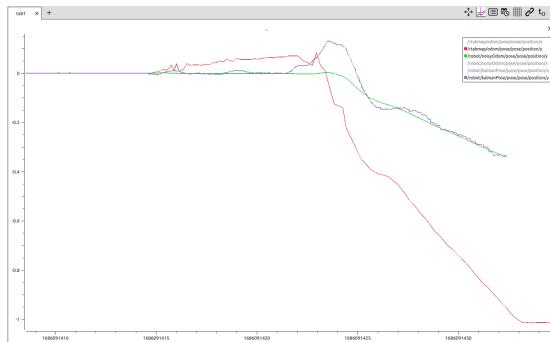


Fig. 10. Filtro de Kalman en Y

#### D. Generación gradilla ocupacional

Con el objetivo de hacer que el robot fuera autónomo, primero debíamos hacer que pudiera reconocer su entorno y saber donde había obstáculos con los cuales pudiera chocar y por lo tanto debía evitar para continuar con su exploración. Con este fin utilizamos el sensor LiDAR que nos da mediciones de a que distancia y en que ángulo respecto al robot se encuentra un posible obstáculo. Juntando esto con la odometría para hacer la localización del robot podríamos generar un mapa del entorno del robot.

Para poder crear este mapa se desarrollo un nodo que recibiera los datos tanto de odometría como de las lecturas del LiDAR, y tuviera como salida una gradilla de ocupación con probabilidades de que las celdas estuvieran ocupadas, libres o fueran desconocidas. Hay que tener en cuenta ciertas consideraciones que se hacen para crear este mapa, como el hecho de que el LiDAR da datos en dos dimensiones, por lo que la distancia y ángulo de la lectura siempre están a la misma altura y por lo tanto no es posible ver obstáculos más bajos que no alcances la altura a la que se coloque el sensor. También es importante tomar en cuenta que el mapa que se va a generar a través de este algoritmo va a tener un tamaño definido al inicio y solo funciona con obstáculos estáticos, lo cual nos ahorra complejidad computacional.

El primer paso para crear el mapa de ocupación es inicializar el mapa del tamaño establecido con todas las celdas con un valor de cero, estas celdas van a contener tres valores, la posición y el ángulo de cada celda con respecto al robot y la probabilidad de que estén ocupadas, cada celda tiene un tamaño en metros preestablecido, lo cual representa la resolución de nuestro mapa. Al principio se considera que el robot esta en el centro de la gradilla, por lo que se puede calcular la posición de cada celda de manera sencilla. Una vez que el robot empieza a moverse entonces hay que volver a recalcular la posición y el ángulo de cada celda en base a los valores que obtengamos de la odometría, para esto simplemente hay que restar la nueva posición al valor que se tenía previamente ya que si se va a valores negativos con respecto al marco de referencia, entonces hay que sumar las coordenadas, en cambio en valores positivos se esta acercando y hay que restar.

Una vez que podemos saber en que parte del mapa se encuentra el robot, lo que hay que hacer es tomar los valores del LiDAR y revisar si las celdas se encuentran ocupadas o libres, para lo cual se checa si el láser esta indicando que hay un obstáculo en cada casilla, para esto se calcula que si la celda esta dentro del ángulo marcado por el láser, y si además se encuentra a una distancia cercana al LiDAR, determinada por un umbral, entonces la celda esta en una zona ocupada, en cambio si esta dentro del rango del ángulo pero la distancia marcada por el sensor es mayor, esta zona se considera libre, ya que no hubo nada que estuviera en medio evitar que el láser llegará más lejos.

Ya con las celdas determinadas como zonas libres u ocupadas entonces se pasa a calcular las probabilidades de ocupación dependiendo del tipo, y se acumulan estas probabilidades para cada celda, llegando a un valor de cero si se esta completamente seguro de que es libre y de uno si se esta completamente seguro de que esta ocupada.

Con esto podíamos determinar las celdas que se considerarán como obstáculos y visualizarlas de forma gráfica utilizando *rviz* como se ve en la Fig. 11.

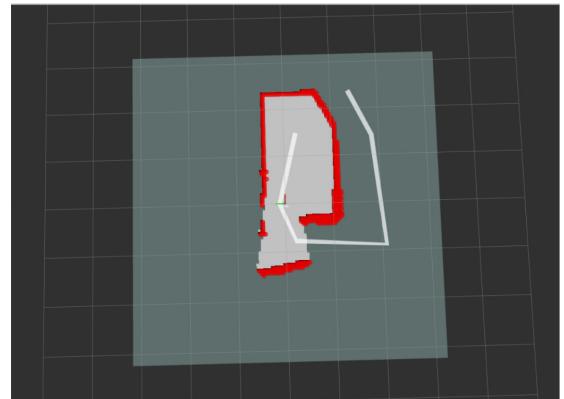


Fig. 11. Mapa de ocupación mostrando en rojo los obstáculos

Nota: este algoritmo no pudo ser usado de forma fiable en el robot ya que era bastante pesado computacionalmente y por cuestiones de tiempo no pudimos buscar la manera de optimizarlo por lo que decidimos usar Hector SLAM para la generación de las gradilla de ocupación. De la misma manera, Hector SLAM provee de una estimación de odometría visual implementada por el algoritmo de ICP. Al ser más confiable que todas la demás odometría, decidimos utilizar esto como entrada principal para el control.

#### E. Navegación sobre gradilla ocupacional

Una vez obtenido el mapa de ocupación, entonces podíamos a pasar a generar trayectorias para que el robot pudiera llegar de un punto A a un punto B considerando los obstáculos que se tenían en el entorno. Para esto implementamos el algoritmo "Rapidly Exploring Random Trees (RRT)", el cual consiste en la generación de puntos aleatorios para explorar si se puede llegar a estos y eventualmente crear un camino hasta el destino. Este algoritmo recibe como entradas la posición actual del

robot, el punto al que se quiere llegar, el tamaño de la zona de seguridad que tendrán los obstáculos para evitar que este pase demasiado cerca y se choque, las posiciones de los obstáculos y las coordenadas que definen el tamaño del espacio por donde se puede navegar. Como salida entrega una serie de puntos que llevan desde la posición actual hasta el destino deseado evitando los obstáculos. Para su implementación se cuenta con dos ramas que parten desde el origen y destino respectivamente, lo primero que se debe hacer es revisar si las ramas se pueden conectar, es decir, que no haya colisiones entre ellas en línea recta, en caso de que no haya se conectan y el algoritmo termina de encontrar el camino, lo único que resta es conectar ambas ramas; en caso contrario habrá que generar un punto aleatorio en el mapa y buscar el nodo más cercano de cada rama a ese punto, se extrae el nodo en dirección al punto y se verifica que no haya colisiones, en cuyo caso el nodo se conecta a la rama y se vuelve a verificar la conexión entre ramas hasta encontrar un camino. Dado que el algoritmo incluye la generación de puntos aleatorios no hay forma de que este termine en caso de que no haya camino posible (si se está en un área cerrada y el destino está fuera de esta) por lo que es necesario definir un número máximo de iteraciones que el algoritmo intentará conectar ambas ramas, en caso de no ser posible simplemente se devuelve una lista vacía.

Adicionalmente se está validando continuamente la trayectoria para que en caso de que se vean nuevos obstáculos que generen una colisión con la ruta ya generada esta sea recalculada y se envíe la nueva ruta para evitar que el robot pueda chocar.

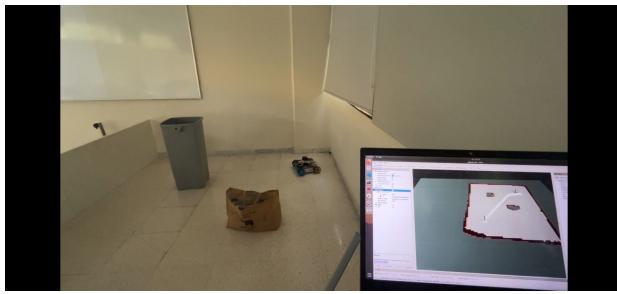


Fig. 12. Robot real y simulación del mapa con generación de caminos.

Dado que este algoritmo utiliza puntos aleatorios puede que el camino a seguir de vueltas que hagan que el robot se desvíe en lugar de avanzar hacia el destino, por lo que una vez que se tiene la serie de puntos generados se pasa por una optimización en la cual se determina si dos puntos no contiguos pueden ser conectados de forma directa, verificando que no haya colisión entre ellos, obteniendo menos puntos de esta forma y un camino más directo.

En cuanto la planta tuvo la capacidad de poder navegar a cualquier punto en el espacio planar, creamos dos algoritmos para determinar de manera autónoma cuál es el siguiente punto a explorar.

#### F. Modo mapeo

Este nodo fue creado con el objetivo de generar el mapa de calor y explorar el mapa al mismo tiempo. Una vez se ha definido el mapa de ocupación, se genera una nueva gradilla con un tamaño definido dependiendo de la cantidad de puntos de interés en los que serán tomadas las mediciones de intensidad de señal.

Con el objetivo de determinar los puntos de interés, se define un tamaño de la gradilla y se toman las dimensiones máximas que puede tener el mapa que ocupación, para dividirlo de manera uniforme y determinar los puntos medios de cada celda, obteniendo así las coordenadas de los puntos de interés a los cuales deberá intentar navegar el robot.



Fig. 13. Mapa de calor de la intensidad de señal.

Con estos puntos generados se mandan uno por uno al algoritmo de generación de trayectorias para que intente generar una ruta hacia el punto de interés, en caso de que no haya una ruta posible entonces este punto será omitido y se intentará llegar al siguiente, en cambio si es posible llegar se esperará hasta que el robot esté en el punto y se tomará una lectura de la señal para actualizar el mapa de calor y de igual forma se pasará al siguiente punto. Una vez que se haya intentado tomar lectura de todos los puntos el algoritmo termina. La visualización del mapa de calor puede ser vista en tiempo real desde *rviz* como muestra la Fig. 13

#### G. Modo exploración

El objetivo de este modo es conocer en su totalidad el espacio cerrado sobre el que el robot se encuentra. Debido a que ya contamos un mapa con valores que indican la probabilidad que esté sea un obstáculo o un espacio que desconocido, reinterpretamos este mapa como una escala de grises. Bajo el mismo concepto, creamos máscaras binarias para determinar las zonas de obstáculos, zonas desconocidas, así como zonas conocidas.

Las zonas de interés siempre estarán al borde de lo desconocido, y conocido, siempre y cuando no sea un obstáculo. Por

## VII. CONCLUSIONES

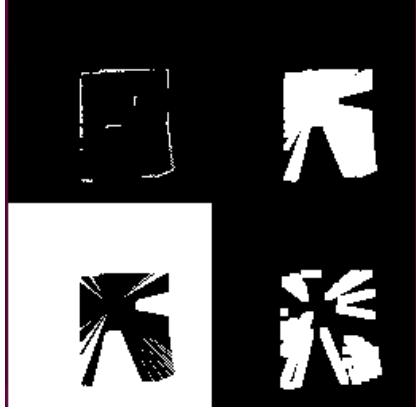


Fig. 14. Máscaras binarias. SI: Obstáculos. SD: Puntos conocidos . II: Zonas desconocida. ID: Zonas de interés.

lo que se definen de la siguiente manera:

$$\begin{aligned}
 M &= \text{Map}_{i,j} \\
 M_{\text{obs}} &= M \in \{x \mid t_{\text{isObstacle}} \leq x \leq 100\} \\
 M_{\text{known}} &= M \in \{x \mid x \leq t_{\text{isObstacle}}\} \\
 M_{\text{unknown}} &= M \in \{x \mid x > 100\} \\
 M_{\text{noGo}} &= M_{\text{obs}} \cup M_{\text{unknown}} \\
 M_{\text{inZone}} &= M_{\text{known}} \cap M_{\text{noGo}} \\
 M_{\text{interest}} &= M_{\text{obs}} \Delta M_{\text{inZone}}
 \end{aligned} \quad (15)$$

Donde  $t_{\text{isObstacle}}$  es el umbral escala 1-100, que determina si un objeto es obstáculo. Además, la región de obstáculos fue dilatada para exaltar su área. Con el objetivo de lograr la intersección en común con las zonas de interés.

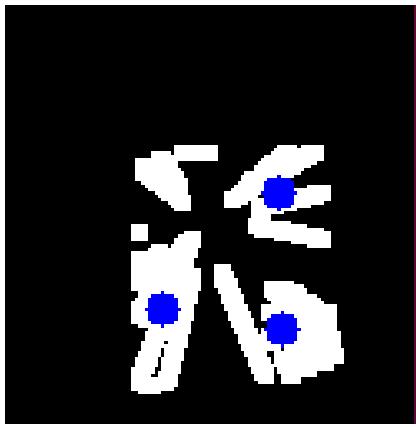


Fig. 15. Puntos de interés en modo exploración.

Dada la máscara con zonas de interés, se utilizó la función de *findContours* de OpenCV para obtener las agrupaciones de puntos en la imagen. Posteriormente, utilizamos momentos para calcular su área, y centro de masa. Para que de esta manera nos sea posible filtrar aquellos puntos no significativos, así como un único punto por zona de interés. Dando como resultado:

El trabajo descrito en este reporte muestra la alta dificultad que representa el reto de que un robot pueda navegar de forma autónoma en un entorno desconocido, requiriendo de múltiples algoritmos complejos tanto computacionalmente como matemáticamente como lo es el filtro de Kalman, para el cual tuvimos bastantes complicaciones al momento de implementarlo ya que no lográbamos terminar de entender como funcionaba, sin embargo al final logramos corregirlo y utilizarlo para tener un mejor cálculo de la odometría utilizando las lecturas de los encoders, aunque finalmente no pudimos usar este modelo debido a que se tienen bastantes factores externos que la hacen poco precisa a pesar de usar Kalman al momento de hacer giros con el robot, aún así fue una gran experiencia de aprendizaje sobre este algoritmo tan útil en el campo de la robótica.

También aprendimos a hacer mapas de ocupación a partir de las lecturas del LiDAR, lo cual ajustando los parámetros funcionaba de manera bastante precisa, aunque era muy pesada computacionalmente la implementación usada para poder tenerla en tiempo real, de igual forma esto nos dio la base para posteriormente hacer la implementación de RRT para la generación de trayectorias, lo cual nos funcionó bastante bien, inclusive cuando se tenían obstáculos no vistos previamente, ya que se detectaba la colisión y se recalculaba la ruta para evitar choques, aunque dependíamos de que tan cerca se pasará de estos obstáculos desconocidos, ya que el rango mínimo de nuestro sensor era de 30 cm y el espacio de pruebas era reducido.

Con este proyecto también decidimos ir más allá y armar nuestro propio robot prácticamente desde cero, lo cual nos trajo complicaciones en cuestión del hardware pero también nos permitió aprender más en el ámbito de diseño tanto modelado 3D como electrónico, ya que tuvimos que hacer el circuito que alimentaría nuestros componentes y hacer que fuera seguro. Además de que pudimos aplicar conocimientos de sistemas embebidos de semestres pasado para hacer la programación del controlador de bajo nivel para los motores.

Finalmente logramos hacer que el robot pudiera navegar de forma completamente autónoma, dándole puntos de interés calculados en tiempo de ejecución para que este explorará y tomará muestras para nuestro diferenciador con la intensidad de señal de WiFi, a pesar de que no pudimos terminar de probarlo por la dificultad de encontrar un espacio apto para esto, pudimos ver en varias ocasiones como empezaba explorar el mapa y recalcular para evitar colisiones, generando una vista completa de su entorno, con el único detalle de que no llegamos a tomar lecturas, pero con la satisfacción de que a pesar de todas las complicaciones que hubo en el camino y el corto periodo de tiempo que se tenía para lograrlo pudimos implementar de manera exitosa prácticamente todo lo que nos propusimos en un inicio, aprendiendo bastante de conceptos muy importantes en la robótica.

## REFERENCIAS

- [1] DNSstuff.com. (2023, March 8). Best Wi-Fi Heat Mapping Tools. Recuperado de <https://www.dnsstuff.com/wi-fi-heat-maps>
- [2] NetSpot. (2023, June 8). Best Wi-Fi Mapper Apps. Recuperado de <https://www.netspotapp.com/wifi-heat-map/best-wifi-mapper-apps.html>
- [3] Eye Networks. (s/f). Identify Dead Zones and Map Your Wi-Fi with a Heatmapper. Recuperado de <https://eyenetworks.no/en/map-wifi-with-heatmapper/>
- [4] Engineering Media. (2023, March 8). The Kalman Filter. Recuperado de <https://engineeringmedia.com/controlblog/the-kalman-filter>
- [5] Gelb, A. (2014). Advanced Kalman filtering, least-squares and modeling: A practical handbook. Wiley.
- [6] Sastry, S. S., & Murray, R. M. (2018). Modern Robotics. Vol. 1. Foundations of Robotics (3rd ed.). MIT Press.
- [7] Anvari, I. (2013). Non-holonomic Differential Drive Mobile Robot Control & Design : Critical Dynamics and Coupling Constraints.