

## REPORTE FINAL — Modelo de Auto Autonomo

### Abstract

En este documento se exponen estrategias en diferentes rubros de la autonomía móvil. Específicamente, se desarrollaron sobre la tarjeta NVIDIA Jetson Nano. Los tres elementos más significativos del documento son el control moderno para la posición y la orientación del robot. Visión computacional para el seguimiento de lineal y para reducir la carga de la red neuronal responsable de la clasificación y de la detección de señales de tránsito, así como los colores de los semáforos. También se exponen algunas limitaciones del modelo propuesto y se comparativas con otras implementaciones conocidas.

## 1. Introducción

El proyecto consiste en el diseñar e implementar el flujo de trabajo de un robot diferencial que fue entregado por Manchester Robotics. Se creó un repositorio con todos los materiales necesarios para llevar a cabo las tareas asignadas. Primero se inició con el control y la configuración para desarrollar sobre la Jetson Nano. Posteriormente, se empezó a trabajar la parte de visión, es decir, las metodologías del seguidor de línea. Para la parte de visión se utilizó visión clásica. Se construyó un conjunto de datos y se afinó una red neuronal que clasifica y detecta señales de tránsito y los semáforos. Además, cambiaron algunas cosas sobre el ecosistema de proporcionó Manchester Robotics, como la integración de CUDA a la librería de OpenCv.

## 2. Metodología

El desarrollo del proyecto se compone de tres elementos principales: control, visión computacional, y aprendizaje profundo. Todos los elementos se integran sobre ROS2 (Robot Operating System). El reto consiste en que un robot diferencial se pueda desplazar de manera autónoma sobre una pista con cruces, semáforos y señalamientos de tráfico. Para ello fue necesario construir un modelo matemático del robot, a partir de ello se diseñó un control lineal y angular. Posteriormente, se implementaron metodologías de visión clásica para que el robot diferencial pudiera seguir la línea de la pista que proporcionó Manchester Robotics. Finalmente, se entrenó una red neuronal profunda para que el robot pudiera responder a los señalamientos de tráfico.

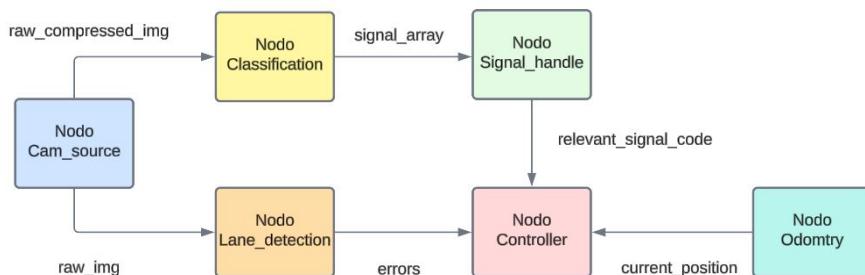


Figura 1: Esquema de los nodos creados para el proyecto.

## 2.1. Modelado del robot diferencial

Para diseñar el modelo cinemático del robot se propone un vector de estados. Dicho vector contiene la posición y la orientación. Por lo tanto,

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

Parte de la cinemática consiste en conocer el cambio, así que intuitivamente se deriva el vector de estados.

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix}$$

En general, se está interesado en relacionar las variables de estado. Para ello se utilizará el centro de rotación instantáneo. Se supone que el robot está dando una vuelta. Entonces, la siguiente expresión es válida

$$v = R\omega$$

Ahora bien, la entrada del sistema será la velocidad angular de las ruedas. Si el radio de las ruedas es constante. La velocidad lineal de cada rueda está dado por

$$v_r = R\omega_r$$

$$v_l = R\omega_l$$

Suponga que  $l$  es la distancia entre las ruedas del robot. Si se considera el centro de masa del robot como la mitad de  $l$ . Entonces, la velocidad lineal de cada rueda con respecto al centro instantáneo de rotación será

$$v_l = \omega \left( R_{ICR} - \frac{l}{2} \right)$$

$$v_r = \omega \left( R_{ICR} + \frac{l}{2} \right)$$

Ambas ecuaciones se operan como un sistema. Para encontrar la velocidad angular en términos de las velocidades lineales se restan las expresiones.

$$-v_l = -\omega R_{ICR} + \omega \frac{l}{2}$$

$$v_r = \omega R_{ICR} + \omega \frac{l}{2}$$

Se resuelve para la velocidad angular

$$\omega = \frac{v_r - v_l}{l}$$

Si se está interesado en encontrar una expresión para la velocidad lineal total del robot. Sume ambas ecuaciones y despeje  $v$ .

$$v = \frac{v_r + v_l}{2}$$

Ahora es posible conocer las variables de estado en término de las entradas del sistema que se desea controlar. En resumen, la cinemática del sistema está dada por el vector de estados.

$$\dot{\mathbf{x}} = \begin{bmatrix} v \cos \theta \\ v \sin \theta \\ \omega \end{bmatrix}$$

Claro, aquí tienes la sección Control del Robot Diferencial incluyendo la propuesta de dos controles diferentes, el control del robot mediante el modelo Tanh y la sección para la función de activación:

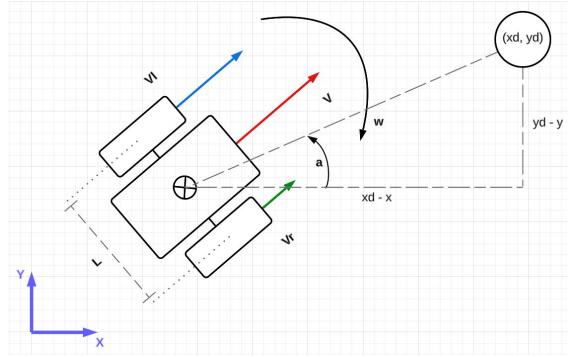


Figura 2: Esquema ilustrativo del robot diferencial.

### 3. Control

Dado que el modelado cinemático del robot no tiene acoplado el control angular con el control de desplazamiento, se proponen dos controles diferentes:

- **Control de desplazamiento:** controla la velocidad lineal del robot.
- **Control angular:** controla la velocidad angular del robot.

Para diseñar el control es necesario presentar algo de teoría de estabilidad. Una función de Lyapunov es continua y diferenciable. Llámese  $V(x)$  tal que  $V(0) = 0$  y  $V(x) > 0$  para todo  $x \in D$  y  $x \neq 0$ . Además,  $\dot{V}(x) \leq 0$ . Para el robot diferencial se necesitan dos controladores: un controlador para la velocidad lineal del robot y otro controlador para la velocidad angular del robot, es decir, la orientación del robot. Primero, se presenta el diseño del control lineal. Considere la métrica euclíadiana como la función de Lyapunov propuesta:

$$d = \sqrt{(x_d - x)^2 + (y_d - y)^2}$$

Ahora derive la expresión con respecto al tiempo:

$$\dot{V} = \dot{d}$$

Es necesario que la función sea negativa, por lo tanto, se añade una ganancia proporcional que servirá para justificar una de las propiedades necesarias:

$$V = -k_p d$$

El control proporcional no proporciona un comportamiento suave, ya que hay cambios muy agresivos de velocidad cuando el robot se aleja o se acerca mucho. Es por ello que se prefiere utilizar una función con las condiciones de Lyapunov, pero que sea más consistente con el comportamiento indistintamente de la distancia a la que se encuentra el robot de la posición deseada. Se utiliza una tanh, por lo tanto, el control resultante es:

$$V = -v_{\max} \tanh\left(\frac{k_p d}{v_{\max}}\right)$$

Para diseñar el control de la orientación, la metodología utilizada es similar a lo expuesto, sólo que la función propuesta de Lyapunov será distinta. Se define la dinámica de error angular como:

$$\theta_e = \theta - \theta_d$$

Se propone la función de Lyapunov como:

$$V = \frac{1}{2} \theta_e^2$$

Posteriormente se deriva y se sustituye con la velocidad angular:

$$\dot{V} = \theta_e \dot{\theta}_e = \theta_e \omega$$

Con  $\omega$  se puede diseñar un control proporcional integral. Para lograrlo, se propone el vector de estados:

$$\mathbf{x} = \begin{bmatrix} \theta_e \\ \int \theta_e dt \end{bmatrix}$$

Si se deriva el vector de estado:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\theta}_e \\ \theta_e \end{bmatrix}$$

Sustituya con la velocidad angular:

$$\dot{\mathbf{x}} = \begin{bmatrix} \omega \\ \theta_e \end{bmatrix}$$

El control resultante será:

$$u = -k_p(\theta - \hat{\theta}) - k_i \int (\theta - \hat{\theta}) dt$$

Para implementar el control descrito, se utilizó un nodo de odometría para conocer la posición actual del controlador, así como un nodo que implementa tanto el control lineal como el control angular.

### 3.0.1. Implementación de Controles de Orientación y Posición

Inicialmente, el robot debe orientarse hacia el punto deseado. Esto se logra calculando el ángulo necesario para dirigirse hacia dicho punto usando la función arco tangente:

$$\theta_d = \arctan 2(y_d - y, x_d - x),$$

donde  $(x_d, y_d)$  son las coordenadas del destino y  $(x, y)$  la posición actual del robot. El error angular  $\theta_e$  se calcula como la diferencia entre el ángulo actual del robot  $\theta$  y  $\theta_d$ :

$$\theta_e = \theta - \theta_d.$$

Para asegurar que el error angular siempre se encuentre en el intervalo más corto entre  $-\pi$  y  $\pi$ , se ajusta  $\theta_e$  según sea necesario.

La velocidad angular  $\omega$  se controla utilizando el modelo tanh para corregir el error estacional de la respuesta del control:

$$\omega = -k_p(\theta - \hat{\theta}) - k_i \int (\theta - \hat{\theta}) dt$$

donde  $k_p$  es la ganancia proporcional,  $k_i$  es la ganancia integral,  $\theta$  es la orientación actual y  $\hat{\theta}$  es la orientación deseada. Este modelo ayuda a reducir la velocidad angular suavemente a medida que el robot se alinea con el ángulo deseado, evitando oscilaciones y sobreimpulsos.

La distancia hasta el punto deseado se calcula como:

$$d = \sqrt{(x_d - x)^2 + (y_d - y)^2}.$$

La activación del control de velocidad lineal se realiza con otra función tanh, que comienza a influir después de que el robot se ha orientado adecuadamente hacia el punto deseado:

$$\text{act} = -\frac{(\tanh(k_{\text{act}}(|\theta_e| - \pi/8)) - 1)}{2}.$$

La figura a continuación ilustra el comportamiento de la función de activación basada en la función tangente hiperbólica, que es crucial para ajustar la velocidad lineal del robot de manera efectiva.

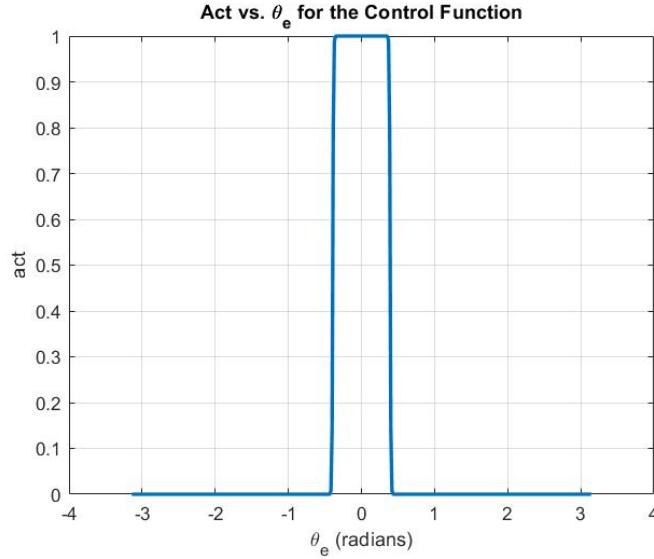


Figura 3: Comportamiento de la función de activación utilizando la función tanh para control de velocidad lineal.

La velocidad lineal  $V$  se ajusta en función de la distancia hasta el destino y la activación del control:

$$V = V_{\max} \tanh\left(\frac{k_{pt}d}{V_{\max}}\right) \cdot \text{act},$$

donde  $V_{\max}$  es la velocidad máxima y  $k_{pt}$  es una constante de proporcionalidad. Este método asegura que la velocidad lineal aumente o disminuya suavemente y que el robot se detenga de manera gradual al llegar al destino.

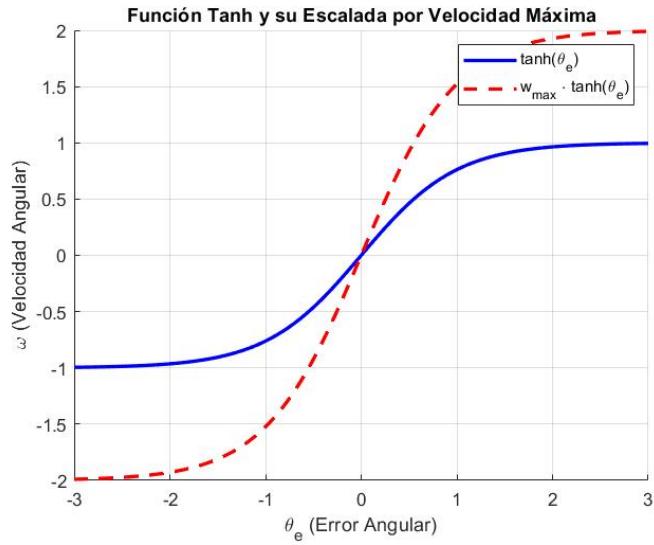


Figura 4: Comportamiento de la función tanh escalada por una velocidad máxima en caso de evaluar en control angular.

## 4. Nodo de Controlador

El nodo **Controller** sirve como gestión de los movimientos del robot. Este nodo recibe datos sobre la odometría del robot y los errores de posición y ángulo, y en función de estos datos, selecciona y aplica el control adecuado para mantener al robot en la trayectoria deseada.

**Inicialización y Parámetros** El nodo **Controller** inicializa varios parámetros que son cruciales para su funcionamiento, como los umbrales de error de distancia y ángulo, las velocidades máximas angular y lineal, y las constantes de los controladores proporcional e integral.

**Suscriptores y Publicadores** El nodo suscribe a varios tópicos para recibir la información necesaria:

- `/errors`: Para recibir errores de posición y ángulo.
- `/odom`: Para recibir la odometría del robot.
- `/current_state`: Para recibir el estado actual del robot.

Además, el nodo publica en el tópico `/cmd_vel` para enviar comandos de velocidad al robot.

**Selección del Control** El nodo determina el tipo de control a aplicar (control de línea o control de posición) en función del estado actual del robot. Esta selección sirve para manejar situaciones como cruces peatonales y giros.

**Control de Línea** El control de línea utiliza un controlador proporcional-integral (PI) para ajustar las velocidades angular y lineal del robot en función de los errores de ángulo y distancia. Esto ayuda a mantener al robot en la línea deseada.

**Control de Posición** El control de posición guía al robot hacia un punto objetivo específico. Utiliza los errores de ángulo y distancia para calcular las velocidades adecuadas, ajustando la trayectoria del robot hacia el punto objetivo.

**Función de Callback del Controlador** Esta función se ejecuta a intervalos regulares y selecciona y aplica el control adecuado (línea o posición) basado en el estado actual del robot. Esto asegura una respuesta continua y adaptativa del robot a su entorno.

## 5. Nodo de Odometría

El nodo **Odometry** calcula la posición actual del robot basándose en las velocidades angulares de las ruedas y publica la posición actualizada en el tópico `/odom`.

**Suscriptores y Publicadores** El nodo suscribe a las velocidades angulares de las ruedas izquierda y derecha y publica la posición actual del robot.

**Cálculo de la Posición** El nodo utiliza las velocidades angulares de las ruedas y el tiempo transcurrido para calcular la nueva posición y orientación del robot. Esta posición se actualiza y publica continuamente, proporcionando datos cruciales para otros nodos como el **Controller**.

## 5.1. Esquema General del Nodo de Controlador y Odometría

- **Inicialización y Parámetros:** Configuración de parámetros iniciales.
- **Suscriptores y Publicadores:** Gestión de la comunicación con otros nodos.
- **Selección del Control:** Determinación del tipo de control basado en el estado del robot.
- **Control de Línea:** Ajuste de velocidades basadas en errores de ángulo y distancia.
- **Control de Posición:** Guiado del robot hacia puntos objetivos específicos.
- **Cálculo de la Posición:** Determinación continua de la posición actual del robot.

Estos nodos trabajan en conjunto para proporcionar un control preciso y eficiente del robot, manteniéndolo en la trayectoria deseada y adaptándose a diferentes situaciones en tiempo real.

## 6. Visión Computacional

Para resolver el problema del seguimiento de línea y la clasificación de señales de tránsito, se implementaron dos nodos principales en ROS 2: el nodo de cámara y el nodo de procesamiento de carril (lane processing).

Claro, aquí tienes la sección reescrita en LaTeX, utilizando pseudocódigo para los ejemplos de código y manteniendo el código real para el pipeline GStreamer:

## 7. Nodo de Cámara

El nodo de cámara se encarga de capturar imágenes en bruto (raw) y publicarlas en dos tópicos diferentes. Este nodo utiliza la biblioteca OpenCV junto con GStreamer para capturar imágenes de una cámara CSI (Camera Serial Interface).

**Inicialización del Nodo** El nodo `Camera` se inicializa con el nombre `camera`. Se configura el pipeline GStreamer en la cadena `filename_`. Se crea un objeto `cv::VideoCapture` llamado `cap_` con el pipeline GStreamer. Además, se crean dos publicadores:

- `image_publisher_` para publicar imágenes raw en el tópico `/camera/image_raw`.
- `compressed_publisher_` para publicar imágenes comprimidas en el tópico `/camera/image_compressed`.

**Configuración del Pipeline GStreamer** El pipeline GStreamer configurado en `filename_` se utiliza para capturar imágenes desde una cámara CSI:

```
"nvarguscamerasrc sensor-id=0 ! "
"video/x-raw(memory:NVMM), width=(int)854, height=(int)480, framerate=12/1, format=(string)NV12 ! "
"nvvidconv flip-method=2 ! "
"videoconvert ! "
"video/x-raw, format=(string)BGR ! appsink"
```

- `nvarguscamerasrc`: Fuente de la cámara Nvidia Argus. `sensor-id=0` indica el primer sensor.
- `video/x-raw(memory:NVMM)`: Especifica el formato de vídeo raw con memoria de Nvidia.
- `width`: Configuración del tamaño de la imagen en anchura (854 píxeles).
- `height`: Configuración del tamaño de la imagen en altura (480 píxeles).
- `framerate`: Configuración de la tasa de fotogramas (12 fotogramas por segundo).

- **format:** Configuración del formato de la imagen (NV12).
- **nvvidconv flip-method=2:** Conversión de vídeo con un método de volteo (flip).
- **videoconvert:** Conversión de vídeo a formato BGR.
- **appsink:** El final del pipeline, que permite la captura de imágenes en la aplicación.

**Captura y Publicación de Imágenes** La función `read_and_publish` se ejecuta en un bucle mientras el nodo está activo, capturando y publicando imágenes.

- **Captura de Imágenes:** `cap_.read(frame_)` captura un fotograma de la cámara. Si el fotograma no está vacío, se procede a publicar las imágenes.
- **Publicación de Imágenes Raw:** Se crea un mensaje de imagen raw utilizando `cv_bridge` y se publica en `image_publisher_`.
- **Publicación de Imágenes Comprimidas:** Se comprime la imagen usando `cv::imencode`. Se crea un mensaje de imagen comprimida y se publica en `compressed_publisher_`.
- Si no se puede capturar un fotograma, se registra un error y se espera brevemente antes de intentar capturar de nuevo.

## Funcionamiento del Nodo

### 1. Inicialización y Configuración:

- Se inicializa el nodo y se configura el pipeline GStreamer.
- Se crean los publicadores de imágenes raw y comprimidas.

### 2. Captura y Publicación en Bucle:

- Mientras el nodo esté activo, se capturan fotogramas de la cámara CSI.
- Cada fotograma se procesa y se publica en dos formatos: raw y comprimido.

### 3. Manejo de Errores:

- Si falla la captura de un fotograma, se registra un error y se intenta de nuevo después de un breve retraso.

Esta implementación permite que otros nodos en el sistema ROS 2 se suscriban a las imágenes publicadas y las utilicen para tareas como el seguimiento de línea y la clasificación de señales de tránsito. La modificación del pipeline da la libertad de poder reducir la latencia en el procesamiento completo del puzzlebot.

## 8. Nodo de Procesamiento de Carril

El nodo `lane_processing` recibe la imagen raw publicada por el nodo de cámara y aplica una serie de transformaciones para detectar y seguir la línea del carril. Las principales etapas de procesamiento son:

**Conversión a Escala de Grises** Primero, se convierte la imagen a escala de grises para facilitar la detección de bordes y reducir el costo computacional, ya que se trabaja con una matriz NxM en lugar de NxMx3 (formato RGB).

**Transformación de Perspectiva** Se aplica una transformación de perspectiva para estimar correctamente las dimensiones de cada marco y la magnitud de los objetos sobre la cámara. Esto se logra mediante el cálculo de un trapecio que representa el carril en el marco del video y la aplicación de la función `cv::warpPerspective`.

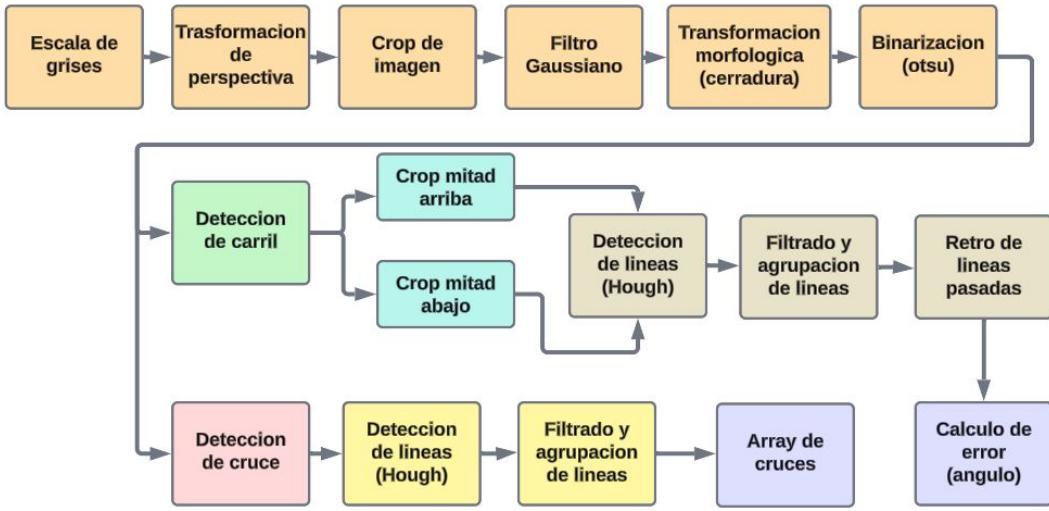


Figura 5: Pipeline del nodo de procesamiento de carril.

**Recorte de Imagen** Se recorta la parte superior de la imagen para eliminar elementos no útiles para el seguimiento de línea, centrándose en la parte del carril visible en la parte inferior del marco.

**Filtro Gaussiano** Se aplica un filtro gaussiano con un kernel de  $5 \times 5$  para eliminar elementos irrelevantes o ruidosos, como suciedad en la pista.

**Transformaciones Morfológicas** Se aplican erosiones y dilataciones para ajustar los bordes del carril. La erosión hace los bordes más estrechos calculando el mínimo sobre el kernel, mientras que la dilatación los amplía calculando el máximo. Con esto también logramos borrar los bordes detectados de la pista en forma de rompecabeza

**Binarización con Otsu** Se utiliza el método de binarización de Otsu para segmentar la imagen. Este método encuentra el umbral óptimo para diferenciar los píxeles de la línea del carril de los del fondo.

**Detección de Líneas con Hough Probabilístico** Se utiliza la transformación de Hough probabilística para detectar líneas en la imagen. Las líneas detectadas se filtran según su pendiente para identificar las que representan el carril. Las líneas verticales se utilizan para seguir en línea recta, mientras que las horizontales ayudan a detectar cruces.

**Selección de Línea Relevante** El algoritmo de selección de línea relevante se basa en la detección y el procesamiento de las líneas del carril en las imágenes capturadas. Este proceso incluye la detección de líneas utilizando la transformación de Hough, la agrupación de líneas similares en clusters, y la extrapolación del centroide relevante para garantizar que el robot siga la trayectoria correcta incluso en ausencia de lecturas de la cámara.

**Detección de Líneas** Se utiliza la transformación de Hough probabilística para detectar líneas en la imagen. Las líneas detectadas se filtran según su pendiente para identificar las que representan el carril.

**Creación de Clusters** Para agrupar las líneas detectadas, se utiliza una analogía con la creación de clusters. Cada línea se representa por su punto medio, y se aplica un algoritmo de clustering basado en la distancia euclídea. El proceso es el siguiente:

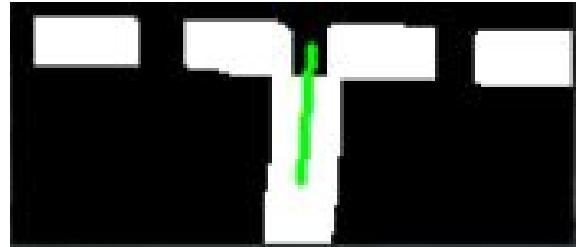
1. **Cálculo de Puntos Medios:** Se calcula el punto medio de cada línea detectada.
2. **Inicialización de Clusters:** Se inicializan los clusters asignando un identificador único a cada punto medio.
3. **Asignación de Vecinos:** Para cada punto medio, se identifican sus vecinos basándose en una distancia umbral  $eps$ .
4. **Formación de Clusters:** Si el número de vecinos es mayor o igual a un número mínimo de puntos ( $minPts$ ), se crea un nuevo cluster o se asigna al cluster existente.
5. **Cálculo del Centroide:** Para cada cluster, se calcula el centroide promediando las coordenadas de los puntos medios de las líneas agrupadas.

**Extrapolación del Centroide Relevante** Para mantener el seguimiento de la trayectoria del carril en caso de no tener lecturas de la cámara, se realiza una extrapolación del centroide relevante. Este proceso se realiza utilizando los historiales de los centroides de los frames anteriores.

1. **Almacenamiento del Historial:** Se mantiene un historial de los centroides de los frames anteriores.
2. **Extrapolación de Centroides:** Si no se detecta una nueva línea en la imagen actual, se extrapola la posición del centroide utilizando regresión lineal sobre los historiales almacenados.
3. **Predicción de Nuevos Puntos:** Se utiliza la tendencia de los datos históricos para predecir la posición del centroide en el siguiente frame.



(a) Detección de cruce utilizando la imagen recortada de la parte superior.



(b) Detección de líneas en la imagen.

Figura 6: Detección de cruce y detección de líneas en la imagen después de detectar el centroide relevante del crop de arriba y abajo, para posterior extrapolacion

**Resumen del Algoritmo** El algoritmo completo para la selección de línea relevante y la extrapolación de centroides sigue los siguientes pasos:

- **Conversión a Escala de Grises:** Convertir la imagen a escala de grises.
- **Transformación de Perspectiva:** Aplicar una transformación de perspectiva.
- **Recorte de Imagen:** Recortar la parte superior e inferior de la imagen.
- **Filtro Gaussiano:** Aplicar un filtro gaussiano para suavizar la imagen.
- **Transformaciones Morfológicas:** Aplicar erosiones y dilataciones.
- **Binarización con Otsu:** Segmentar la imagen usando el método de Otsu.
- **Detección de Líneas:** Detectar líneas utilizando la transformación de Hough probabilística.
- **Creación de Clusters:** Agrupar líneas similares en clusters y calcular el centroide.
- **Extrapolación de Centroides:** Extrapolar el centroide relevante usando la historia de frames anteriores en caso de no tener lecturas de la cámara.



Figura 7: Detección de líneas de Hough sin procesamiento de la imagen.

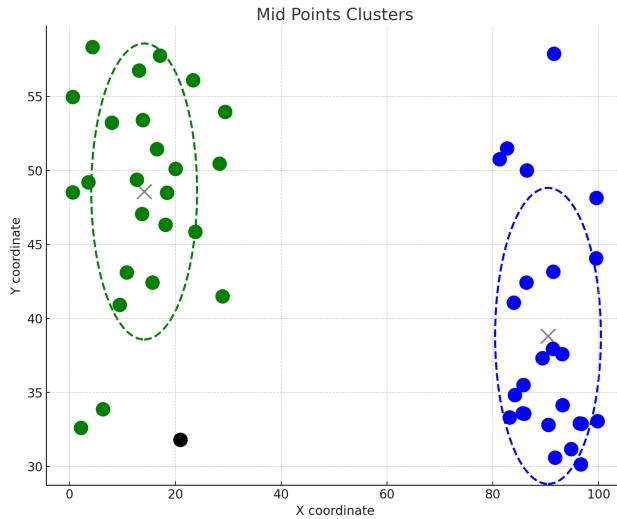


Figura 8: Puntos medios de las líneas detectadas y áreas de agrupación de clusters. Los círculos indican los centroides de cada cluster.

**Beneficios del Algoritmo** Este algoritmo permite un seguimiento robusto de la línea del carril incluso en condiciones donde la visibilidad puede ser intermitente. La creación de clusters asegura que las líneas detectadas sean representativas del carril, y la extrapolación del centroide permite mantener la trayectoria correcta del robot en ausencia de nuevas lecturas de la cámara.

Para implementar este nodo se desarrollaron varias librerías: `lane_params`, `cv_wrapper`, `lane_funcs` y `perspective`.

**Inicialización del Nodo** El nodo `LaneProcessingNode` se inicializa con el nombre `lane_processing_node`. Se suscribe al tópico `/camera/image_raw` para recibir las imágenes de la cámara. Se crea el publicador de tipo custom:

- `publisher_errors_` para publicar los errores detectados (error de angulo y cruce).

**Captura y Procesamiento de Imágenes** El nodo recibe una imagen, la convierte a escala de grises, aplica una transformación de perspectiva y realiza un recorte. Luego, se aplica un filtro gaussiano, transformaciones

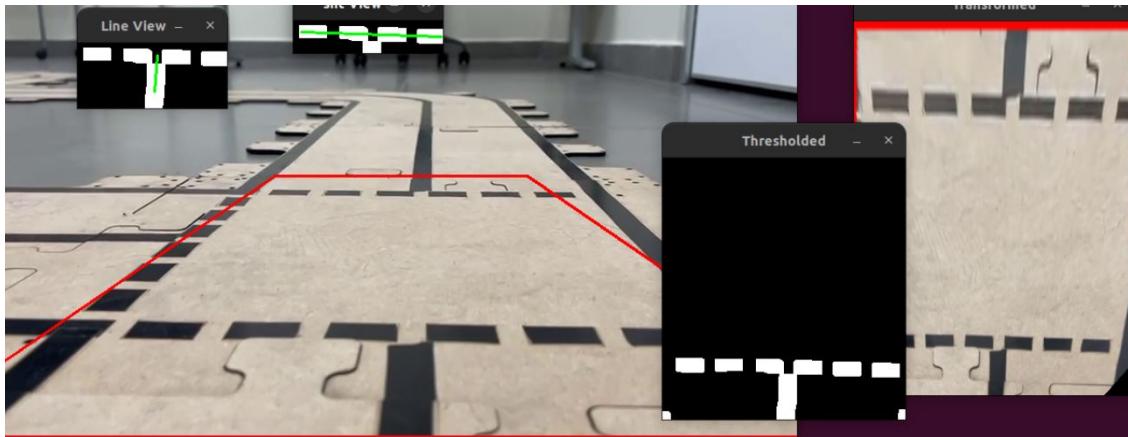


Figura 9: Test del nodo de procesamiento de carril.

morfológicas y binarización de Otsu. Finalmente, se detectan las líneas utilizando la transformación de Hough probabilística y se calcula el error de ángulo y distancia.

**Publicación de Resultados** Se publican los resultados procesados a través de los publicadores creados. Si no se puede procesar un fotograma, se registra un error y se intenta de nuevo.

#### Funcionamiento del Nodo

##### 1. Inicialización y Configuración:

- Se inicializa el nodo y se configura la suscripción al tópico de imágenes.
- Se crean los publicadores de ángulo, distancia y errores.

##### 2. Captura y Procesamiento de Imágenes:

- Mientras el nodo esté activo, se reciben fotogramas de la cámara.
- Cada fotograma se procesa siguiendo las etapas descritas: conversión a escala de grises, transformación de perspectiva, recorte, filtro gaussiano, transformaciones morfológicas, binarización de Otsu y detección de líneas.

##### 3. Publicación de Resultados:

- Se publican los errores de ángulo y distancia, así como los errores detectados.
- Si falla el procesamiento de un fotograma, se registra un error y se intenta de nuevo.

Esta implementación permite que otros nodos en el sistema ROS 2 se suscriban a los resultados publicados y los utilicen para tareas como el control del movimiento del robot y la detección de cruces.

## 9. Aprendizaje Profundo

Para resolver el problema de detección de objetos y clasificación se utilizó un modelo ya entrenado conocido como YOLO que quiere decir (You Only Look Once). Básicamente, lo que se hizo fue tomar la versión más reciente del modelo, seccionarla y volver a entrenarla con las señales de tránsito, es decir, menos categorías. Realmente, se aplicó una técnica llamada aprendizaje por transferencia, la cual consiste en seccionar la red neuronal en el clasificador de salida. YOLO es un tipo de red neuronal convolucional. Las redes neuronales convolucionales pueden variar mucho en su arquitectura, pero la idea general es que se aplican convoluciones sobre la imagen para extraer características, así es como funciona un filtro y eventualmente pasa a un clasificador. Entonces, la parte que se hizo fue cambiar el clasificador. Para entrenar el modelo se tomaron varias fotografías de las señales de tránsito impresas y los semáforos.

## 10. Transferencia de Aprendizaje y Proceso de Entrenamiento

Para entrenar el modelo YOLO v8 se utilizó la técnica de transferencia de aprendizaje, la cual permite reutilizar un modelo preentrenado en un gran conjunto de datos y ajustarlo para un nuevo conjunto de datos específico.

### Transferencia de Aprendizaje

- **Modelo Preentrenado:** Se utilizó un modelo YOLO v8 preentrenado en el conjunto de datos COCO, conocido por su capacidad para detectar una variedad de objetos comunes.
- **Ajuste Fino (Fine-tuning):** El modelo preentrenado se ajustó con un conjunto de datos específico de señales de tránsito y semáforos. Este proceso incluyó el ajuste de los pesos del modelo para mejorar la precisión en la detección de estos objetos específicos.

### Proceso de Entrenamiento

- **Preprocesamiento de Imágenes:** Las imágenes se ajustaron en tamaño y se normalizaron para que sean compatibles con el modelo YOLO v8, como se muestra en las figuras 10a, 10b y 10c.
- **Entrenamiento y Validación:** Se entrenó el modelo utilizando una división de los datos en conjuntos de entrenamiento y validación. Durante este proceso, se monitorearon las métricas de rendimiento del modelo.

**Métricas de Evaluación** Es crucial evaluar no solo la precisión (accuracy) del modelo, sino también otras métricas como el recall y la matriz de confusión para validar la confiabilidad del modelo.

- **Precisión (Accuracy):** Indica el porcentaje de predicciones correctas sobre el total de predicciones.
- **Recall:** Mide la capacidad del modelo para identificar correctamente todas las instancias positivas de una clase.
- **Matriz de Confusión:** Proporciona una visión detallada de las predicciones del modelo, mostrando los verdaderos positivos, falsos positivos, verdaderos negativos y falsos negativos.

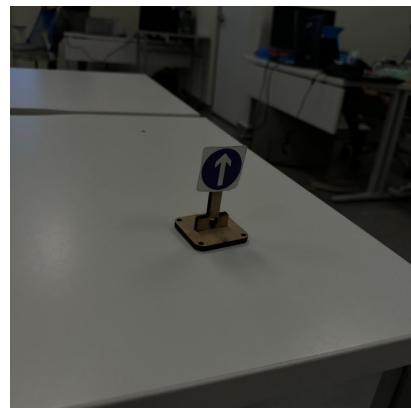
**Consideraciones Especiales** Para este modelo, no se pudieron aplicar técnicas de aumento de datos como la rotación de imágenes, ya que hay señales de tráfico específicas para direcciones izquierda y derecha que podrían generar conflictos.



(a) Preprocesamiento de imagen 1



(b) Preprocesamiento de imagen 2



(c) Preprocesamiento de imagen 3

Figura 10: Ejemplos de imágenes preprocesadas para entrenamiento de YOLO v8.

## 11. Nodo de Clasificación

El nodo `classification` recibe la imagen raw comprimida publicada por el nodo de cámara y utiliza YOLO v8 para clasificar señales de tránsito y semáforos. YOLO v8 es un modelo de aprendizaje profundo optimizado para la detección rápida y precisa de objetos.

**Preprocesamiento** Las imágenes comprimidas se descomprimen y se preprocesan para ser compatibles con el modelo YOLO v8. Este preprocesamiento incluye el ajuste de tamaño y la normalización de los píxeles.

**Inferencia con YOLO v8** Se ejecuta la inferencia utilizando el modelo YOLO v8, que devuelve las coordenadas de los cuadros delimitadores y las etiquetas de las clases detectadas en la imagen.

**Postprocesamiento** Los resultados de la inferencia se postprocesan para filtrar detecciones de baja confianza y se integran en el sistema de control del robot para responder adecuadamente a las señales de tránsito y semáforos detectados.

Para el postprocesamiento se generó otro nodo llamado `signal_handle`, que maneja las señales detectadas para determinar el estado actual del robot en función de las señales de tránsito.

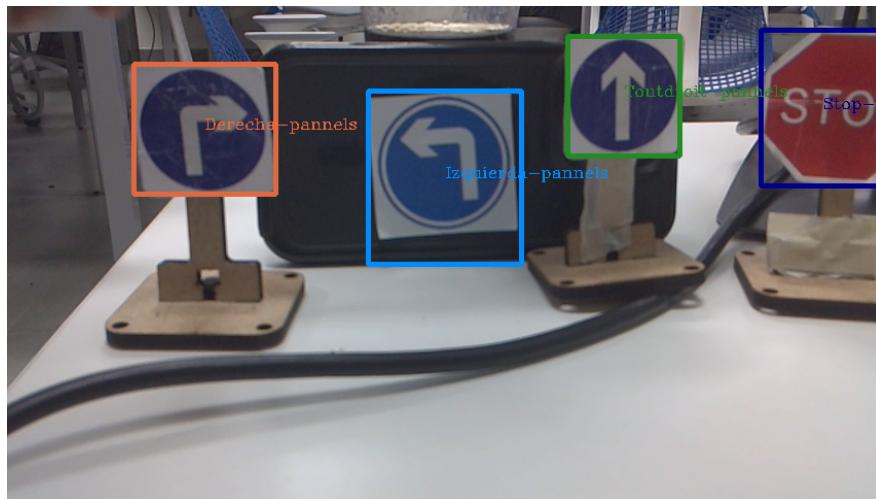


Figura 11: Clasificación de señales.

## 12. Nodo de Manejo de Señales

El nodo `SignalHandlerNode` recibe los resultados de clasificación publicados por el nodo de clasificación y procesa estas señales para determinar la acción a realizar por el robot.

### Suscripción y Publicación

- `subscriber_`: Suscripción al tópico `classification_results`" para recibir los resultados de la clasificación.
- `publisher_`: Publicación en el tópico `current_state`" para informar el estado actual del robot.

### Inicialización

- `signal_history`: Una cola que almacena el historial de señales detectadas.
- `state_queue`: Una cola que almacena los estados del robot basados en las señales detectadas.

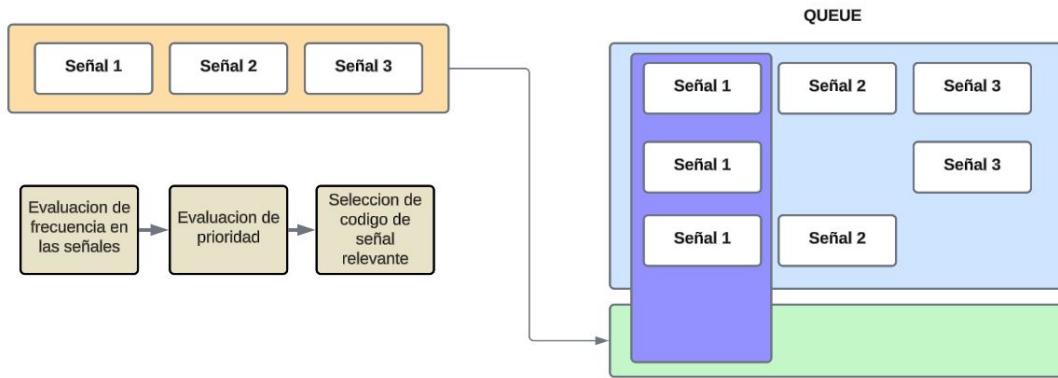


Figura 12: Pipeline para procesamiento de multiples señales.

- `priority_map` y `vote_threshold`: Mapas que definen las prioridades y umbrales de voto para cada tipo de señal.

#### Lógica de Manejo de Señales

- **Actualización del Historial de Señales:** Las señales detectadas se almacenan en una cola de historial para mantener un registro de las detecciones recientes.
- **Obtención de Señales Más Frecuentes:** Se calcula la frecuencia de cada señal en el historial y se seleccionan las señales que alcanzan el umbral de votos necesarios.
- **Filtrado por Prioridad:** Las señales seleccionadas se filtran por prioridad para determinar las señales más importantes.
- **Actualización de la Cola de Estados:** La cola de estados se actualiza con los nuevos estados basados en las señales priorizadas.
- **Procesamiento de la Cola de Estados:** Se procesa la cola de estados para ejecutar la acción correspondiente y publicar el estado actual del robot.

**Código del Nodo de Clasificación** El nodo de clasificación se llama `ClassificationNode` y realiza las siguientes funciones:

- Suscribirse al tópico de la cámara para recibir imágenes comprimidas.
- Descomprimir y preprocesar las imágenes para que sean compatibles con YOLO v8.
- Ejecutar la inferencia con YOLO v8 para detectar señales de tránsito y semáforos.
- Publicar los resultados de la clasificación en el tópico `/classification_results`.

**Código del Nodo de Manejo de Señales** El nodo de manejo de señales se llama `SignalHandlerNode` y realiza las siguientes funciones:

- Suscribirse al tópico `/classification_results` para recibir los resultados de la clasificación.
- Procesar las señales detectadas para determinar el estado actual del robot.
- Publicar el estado actual del robot en el tópico `/current_state`.

## 13. Experimentos

### 13.1. Control

En general, hubo varios problemas de sintonización con el control. Inicialmente, se probaron dos funciones de saturación para el control de la velocidad angular y para el control de la velocidad lineal.

El control angular usando un modelo tanh no fue lo suficientemente responsive en cuanto al seguimiento de línea y tampoco fue posible sintonizarlo de manera deseada, ya que para el control de línea, el control angular con tanh producía demasiadas oscilaciones que terminaban perdiendo la referencia en curva y provocando que se perdiera debido a la latencia del sistema cuando se incorporaron los nodos finales. Es por ello que al final se optó por simplificar el enfoque de control por un proporcional integral (PI). Teóricamente, no es posible alcanzar la referencia con un control proporcional debido al error estacionario. Entonces, se aplicó la parte integral. El problema de la parte integral es que acumula el error y eventualmente no es posible controlar el robot cuando pierde referencia. La solución más pragmática conocida es reiniciar la variable del control integral periódicamente.

Posteriormente, se hizo una sintonización con pruebas sencillas sobre la pista. El control está hecho sobre varias capas. Por ejemplo, en el firmware que subió Manchester Robotics, las ganancias de cada motor se configuraban en un JSON que actualizaba la hackerboard a través de conexiones inalámbricas. Una vez sintonizado, se trabajó sobre la capa implementada en ROS 2. Cabe señalar que los diferenciales de tiempo no eran consistentes, por lo que se utilizó un reloj por cada ciclo. Con ello, mejoró la calidad de la odometría, y por consiguiente, la respuesta del control. La velocidad del control no fue un rubro relevante por la velocidad a la que se desplazó el robot diferencial.

Para validar el control, se hicieron pruebas en CoppeliaSim con un robot diferencial, obteniendo la aprobación para el control de posición.

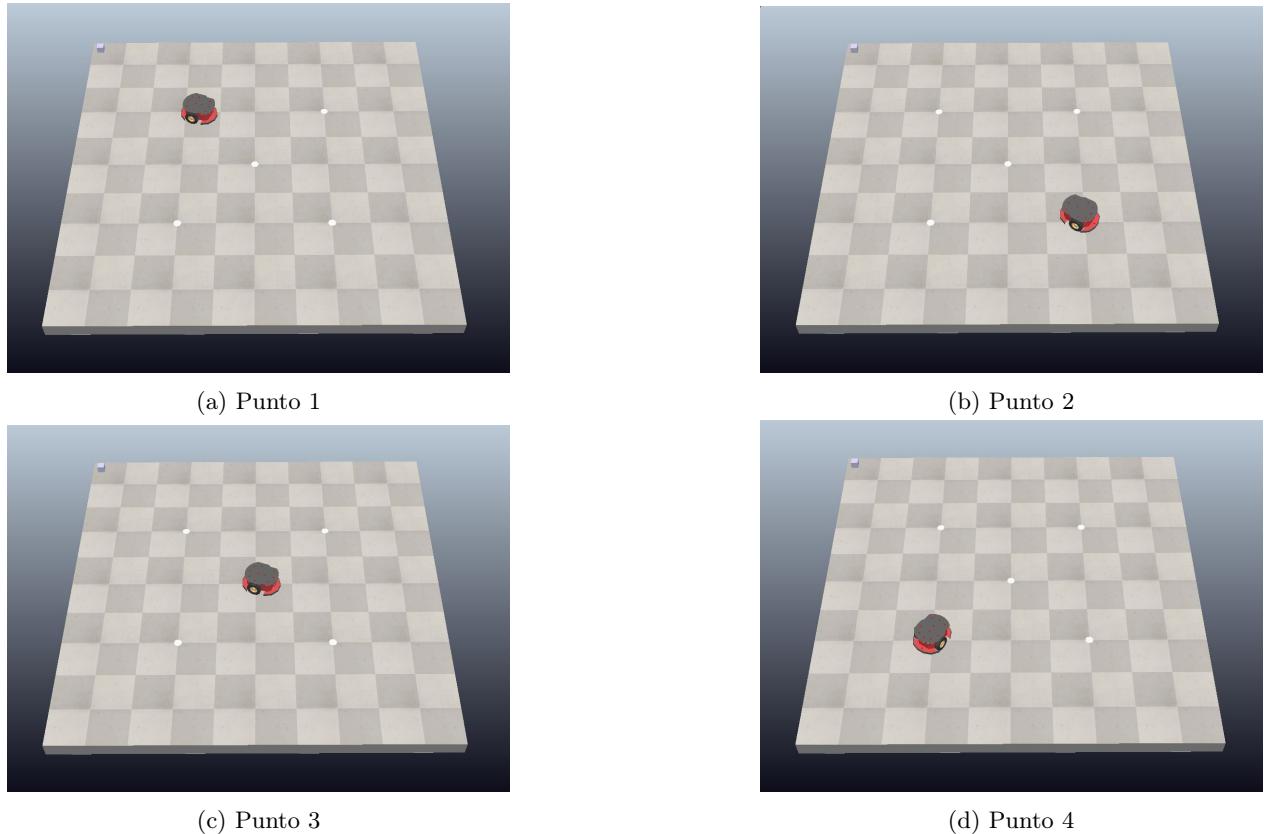


Figura 13: Imágenes de las respuestas de la simulación en los diferentes puntos de interés.

## 13.2. Visión Computacional

Para probar lo descrito en la sección anterior se grabó un video con la cámara del celular desde la perspectiva del robot diferencial. Se abrió el video con las herramientas de OpenCV en la computadora y se empezó a diseñar la metodología que se aplicaría para el seguimiento de línea. Una vez esquematizado, se migró a la cámara del robot. La idea de grabar un video con el celular fue por las limitaciones computacionales que tiene la plataforma de desarrollo. En ese momento no se había hecho uso de CUDA. Además, la transmisión de información por red es demasiado costosa cuando se está trabajando con sistemas embebidos. Otro elemento importante son las variables de entorno conocidas y las que son controlables como la iluminación del lugar. En general, se considera que esta fue la parte más difícil del reto, ya que no hay una solución única.

Se hicieron distintos algoritmos de agrupación de líneas para detectar una única línea. Se probó sacando los promedios de los puntos superiores e inferiores del resultado de detección de líneas por Hough, comparando pendientes y sacando una del promedio de cada grupo. Se seleccionaba el grupo comparando las distancias entre puntos medios, sin embargo, esta implementación era demasiado ruidosa en el momento de la aplicación.

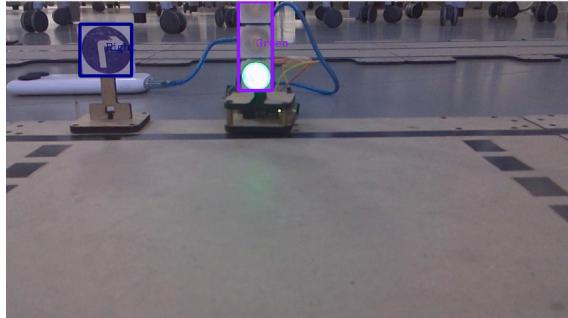
Posteriormente, se utilizó la analogía del clustering, dividiendo el frame en arriba y abajo previo a la detección de líneas, el agrupamiento y la retroalimentación de las líneas anteriores. Este método es más barato computacionalmente, ya que solo se calculan distancias euclídeas para evaluar y sacar un centroide, además de robustecerlo con la retroalimentación y extrapolación.



Figura 14: Comparacion de agrupamiento.

## 13.3. Aprendizaje Profundo

La parte de experimentación consistió principalmente en la generación del conjunto de datos. Se tomaron varias imágenes de las señales de tránsito y además se aumentó el conjunto de datos al aplicar transformaciones geométricas, es decir, se que se cambió la orientación de las imágenes, con ello se esperaba desarrollar un modelo más robusto. El único problema de haber hecho esto es que las señales para dar vuelta a la izquierda o a la derecha se confundieron, y por lo tanto el modelo tenía problemas para clasificar. No obstante, se cambió el conjunto de datos por un conjunto pasado, donde no se tienen las rotaciones de las señales. Cuando se utilizó dicho conjunto de datos para entrenar el modelo todo funcionó correctamente. En entrenamiento se llevó a cabo en la nube a través de servicios de pago. Después se quantizó el modelo con TinyYOLO para que fuera capaz de correr sobre la Jetson Nano. Desafortunadamente, la tarjeta no tiene soporte para los optimizadores que publicó NVIDIA RT hace poco.



(a) Vuelta a la derecha y verde



(b) Vuelta a la izquierda



(c) Señal de alto



(d) Señal de frente

Figura 15: Imágenes de las respuestas de la simulación en los diferentes puntos de interés.

## 14. Resultados

### 14.1. Control

Fue necesario reducir la velocidad lineal y angular del robot por motivos de latencia entre los algoritmos de visión computacional clásica y el costo computacional de la red neuronal. Por este motivo no fue un problema la rapidez con la que el control converge. Una de las principales dificultades fue la sintonización del control, ya que había demasiados elementos no lineales que no son tomados en cuenta en el diseño del control. Por ejemplo, la caja de engranes de cada uno de los motores. Por lo tanto, la simulación del control para su sintonización no fue de mucha ayuda.

### 14.2. Visión computacional

Es difícil justificar las decisiones de diseño en la parte de visión computacional debido a la naturaleza del problema. La mayoría de los recursos existentes en internet utilizan exclusivamente soluciones con aprendizaje profundo, lo cual es casi impensable tomando en cuenta la cantidad de recursos computacionales disponibles. Es por ello que hay un balance entre una solución que sea implementable y lo sofisticado que puede ser el construir dicha solución.

### 14.3. Aprendizaje profundo

Probablemente el modelo de aprendizaje profundo fue lo más sencillo de desarrollar, ya que muchas de las herramientas ya estaban implementadas y el subproblema correspondiente ya está resuelto a muchas escalas, así que únicamente fue cuestión de afinar lo ya conocido. Sin embargo, uno de los problemas que a veces pueden surgir son la visualización de los datos, es decir, a medida que aumenta el tamaño de los datos, es más difícil tener la certeza de que todos son correctos. En general, en aprendizaje profundo es una técnica que se ha popularizado mucho en los últimos años.

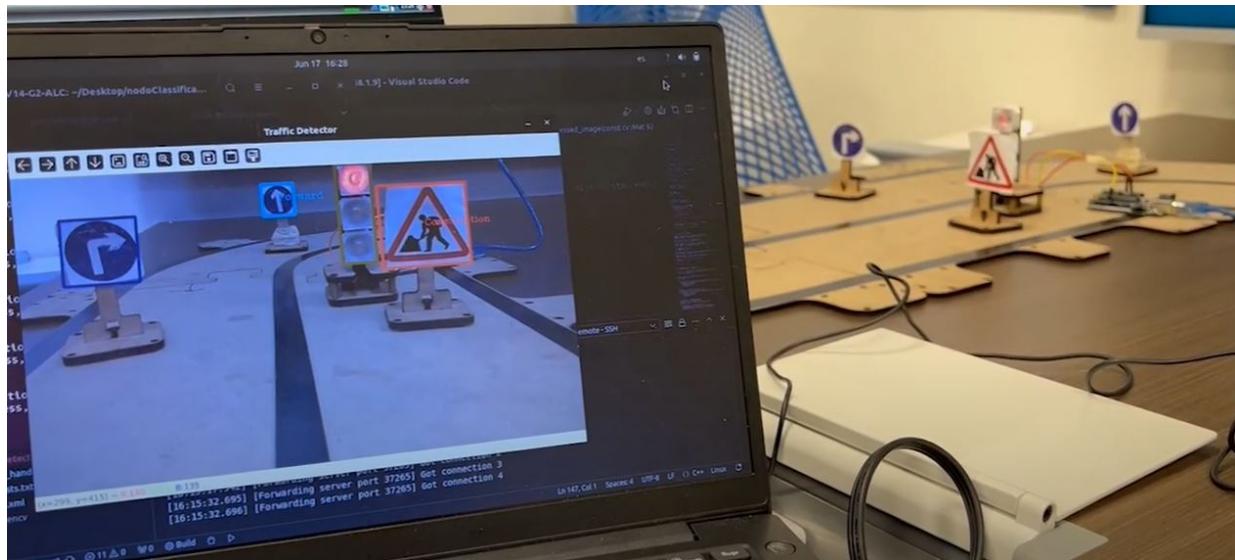
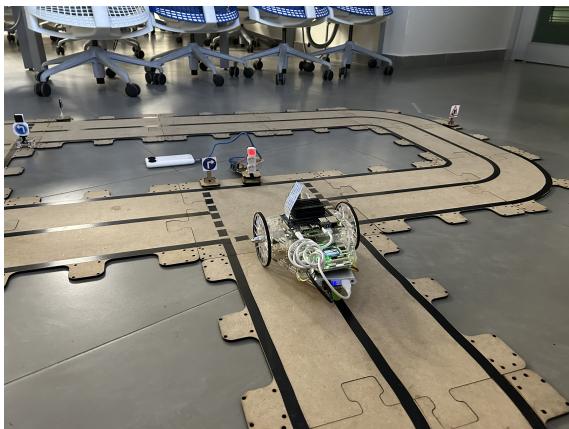


Figura 16: Clasificación de señales.

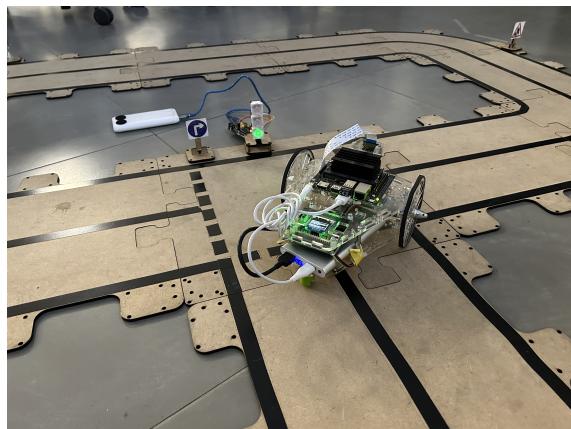
Para una demostración en video de la detección de líneas y cruce, consulte el siguiente enlace: [https://www.youtube.com/watch?v=ZkQhuwi-L\\_w&t=15s](https://www.youtube.com/watch?v=ZkQhuwi-L_w&t=15s).

## 15. Conclusión

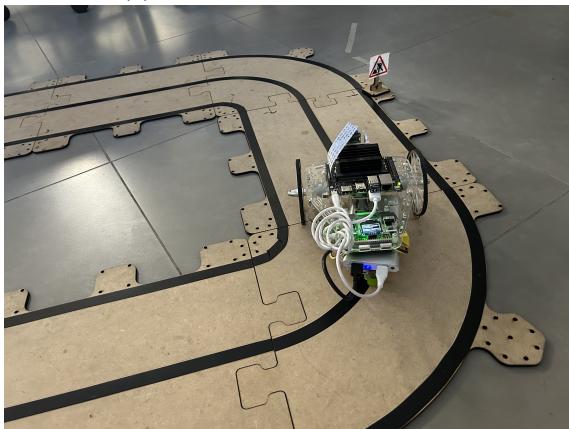
En conclusión, el control fue lo suficientemente bueno para el contexto de la aplicación. No obstante, es posible explorar otras estrategias de control que convergen más rápido. Suponiendo que se cambie la plataforma de desarrollo. Más aún, no se aprovecharon todos los recursos. Una de las principales limitaciones fue omitir el uso de GPUs y la dificultad de portar todo correctamente a C++. Se recomienda explorar controladores como modelos predictivos que se conocen por ser bastante responsivos. La parte de visión computacional está resuelta exclusivamente con metodologías clásicas que operan sobre matrices en escala de grises. Se considera que esta fue la parte más complicada del proyecto porque hay muchos factores que influyen cómo se diseña la solución del seguimiento de línea. Además, aquí fue donde el equipo se percató de las limitaciones computacionales que tiene la Jetson Nano. Podría ser pertinente explorar otras soluciones más robustas a factores como perturbaciones en el entorno. No hay mucho que agregar en la parte de redes neuronales. Casi todos los problemas de la industria están resueltos con la metodología de presentada en este proyecto.



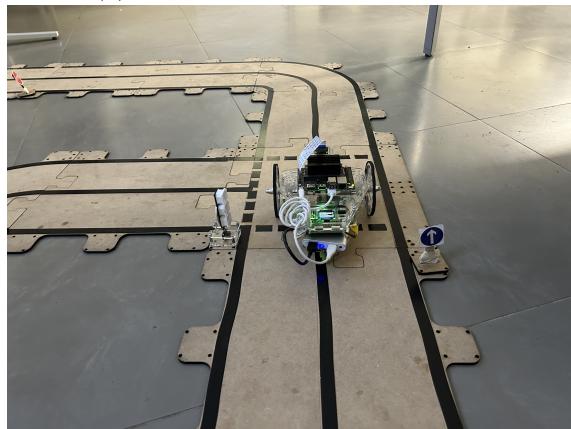
(a) Detección de semáforo rojo



(b) Detección de vuelta a la derecha



(c) Detección de construcción



(d) Detección de señal de derecho

Figura 17: Detección de señales de tránsito por el robot.

## Referencias

- [1] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*, volume Second edition. The MIT Press, 2011.
- [2] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer Tracts in Advanced Robotics. Springer Berlin Heidelberg, 1 edition, 2011. 43 b/w illustrations, 295 illustrations in colour.

Submitted by Inés Alejandro García Mosqueda, Oliver Josel Hernández Rebollar, Paul Enrique on 19 de junio de 2024.