



**Tecnológico  
de Monterrey**

**Reto semanal 5. Manchester Robotics**

Carlos Adrián Delgado Vázquez A01735818

Alfredo Díaz López A01737250

Juan Paulo Salgado Arvizu A01737223

Bruno Manuel Zamora García A01798275

29 de mayo del 2025

[\*\*Resumen\*\*](#)

En esta práctica se desarrolló un sistema de navegación autónoma para el robot móvil Puzzlebot, utilizando el framework ROS 2. Se integraron módulos de visión artificial para el seguimiento de líneas mediante cámara, y detección de semáforos con una red neuronal YOLOv8. El robot ajusta su comportamiento en función del entorno visual, combinando control de movimiento y procesamiento de imágenes en tiempo real.

## Objetivos

### Objetivo principal

Desarrollar un sistema de navegación autónoma para el robot móvil Puzzlebot en el entorno ROS, integrando visión artificial para el seguimiento de líneas y la detección de semáforos, con el fin de mejorar la capacidad de percepción y toma de decisiones en tiempo real.

### Objetivos particulares

- Implementar un módulo de visión que permita detectar y seguir líneas utilizando la cámara del robot.
- Procesar el error de seguimiento para ajustar dinámicamente las velocidades lineales y angulares del Puzzlebot.
- Integrar una red neuronal YOLOv8 entrenada para reconocer semáforos en sus tres estados (rojo, amarillo y verde).
- Diseñar una lógica de control que modifique el comportamiento del robot en función del estado del semáforo detectado.
- Validar la correcta interacción entre los distintos nodos ROS para asegurar un funcionamiento coordinado y en tiempo real.
- Evaluar el desempeño del sistema ante distintas condiciones visuales y escenarios de navegación.

## Introducción

La navegación autónoma en robots móviles es un área clave dentro de la robótica moderna, donde la percepción del entorno a través de sensores como cámaras permite desarrollar sistemas de control inteligentes y adaptativos. Esta práctica se enfoca en la integración de visión artificial para el seguimiento de líneas y la detección de semáforos, empleando el framework Robot Operating System (ROS) sobre una plataforma embebida NVIDIA Jetson Nano, con el robot Puzzlebot como plataforma experimental.

El procesamiento de imágenes en tarjetas embebidas como la Jetson Nano ha cobrado relevancia por su capacidad de ejecutar modelos de visión por computadora y aprendizaje profundo en tiempo real, con eficiencia energética y bajo costo. La Jetson cuenta con una GPU integrada basada en arquitectura CUDA Ilustración 1 Nvidia CUDA Ilustración 1, lo que permite acelerar tareas intensivas como la inferencia con redes neuronales convolucionales (NVIDIA, 2023).

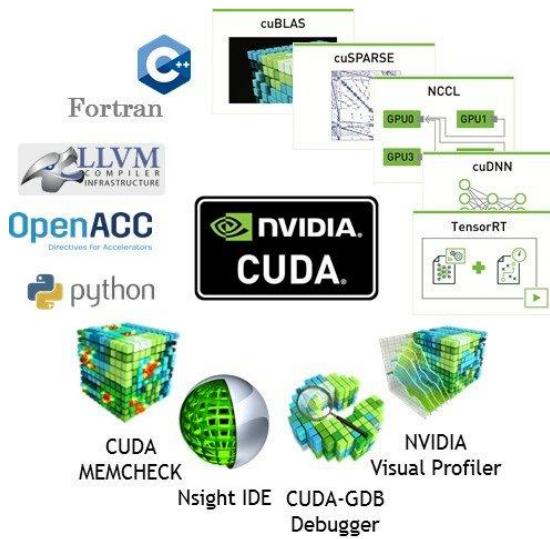


Ilustración 1 Nvidia CUDA

La interconexión entre la Jetson y la cámara CSI permite la adquisición directa de video en alta resolución, ideal para aplicaciones en visión artificial. Estas imágenes son procesadas por algoritmos que aplican técnicas de preprocessamiento como recorte, binarización y detección de centroides para identificar una línea guía en el entorno. El sistema calcula un error de posición respecto al centro del campo visual y ajusta las velocidades del robot en consecuencia, utilizando un esquema de control proporcional-integral-derivativo (PID) (*Raspberry Pi Camera Module V2.1 | Módulo De Cámara Raspberry Pi, Interfaz CSI-2, Resolución 3280 X 2464 Píxeles, 30fps | RS, n.d.*). El video se recorta para seleccionar una región de interés (ROI) en la parte inferior del encuadre, donde es más probable encontrar la línea guía. Posteriormente, se convierte a escala de grises y se aplica un algoritmo de binarización automática (como Otsu) para separar la línea del fondo.

Luego, mediante operaciones de morfología matemática (apertura o cierre), se eliminan imperfecciones como ruido o discontinuidades en la línea detectada. El sistema calcula el centroide de la línea binarizada y lo compara con el centro de la imagen para estimar un error de posición horizontal. Este error es enviado a un nodo de control que ajusta las velocidades lineal y angular

del robot usando un controlador PID, permitiendo que el Puzzlebot corrija su trayectoria y se mantenga sobre la línea.

Este enfoque permite una navegación suave y reactiva, incluso en trayectorias curvas, siempre que las condiciones visuales sean adecuadas. Para robustecer el sistema, se añadió ecualización adaptativa (CLAHE) y se ajustaron parámetros como el tamaño del kernel morfológico y el umbral de tolerancia inferior, mejorando el rendimiento ante cambios de iluminación o variaciones en la textura del suelo (*Morphological Filters–Bioimage Analysis Training Resources*, n.d.), como ejemplo en la Ilustración 2.

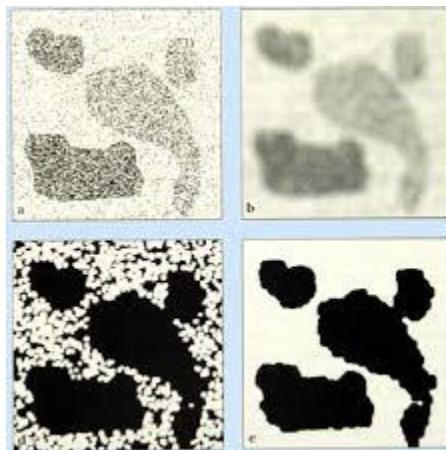


Ilustración 2 Ejemplo de filtrado morfológico

Para fortalecer la percepción del entorno, se utilizó una red neuronal YOLOv8, entrenada en la plataforma Roboflow, especializada en el etiquetado y procesamiento de datasets de visión artificial. YOLO (You Only Look Once) es un modelo de detección en tiempo real ampliamente utilizado por su velocidad y precisión al identificar objetos en imágenes (Jocher et al., 2023), como el ejemplo de Ilustración 3. En este caso, se entrenó para reconocer los tres estados de un semáforo: rojo, amarillo y verde. La salida de esta red fue integrada con la lógica de navegación del Puzzlebot para tomar decisiones dinámicas como detenerse, avanzar o reducir velocidad.

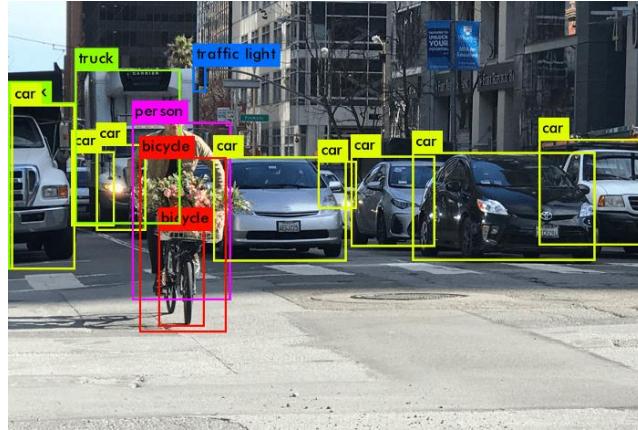


Ilustración 3 Red neuronal con YOLO

En conjunto, esta práctica combina visión artificial, control reactivo, aprendizaje profundo e integración embebida para construir un sistema de navegación autónoma eficiente, escalable y adaptado a los requerimientos modernos de la robótica móvil.

**Solución del problema** En esta sección presentamos la solución integral al reto de seguimiento de línea con semáforo para el robot Puzzlebot. Partimos de una arquitectura modular sobre ROS 2, en la que tres nodos: detección de línea, detección de semáforo y controlador de movimiento, intercambian información a través de tópicos para transformar la percepción en acciones precisas.

Primero, describimos la infraestructura hardware y software que sustenta el sistema: la cámara instalada en la Jetson Nano, el modelo YOLOv8 dedicado a la inferencia de semáforo y el pipeline de visión por computadora basado en OpenCV para calcular el error lateral de la pista. A continuación, detallamos cómo estos datos se integran en nuestro controlador, que emplea un PID exclusivamente en la componente angular y regula la velocidad lineal mediante rampas suaves, adaptándose dinámicamente al estado del semáforo (verde, amarillo o rojo).

Por último, explicamos los criterios de diseño que garantizan robustez y facilidad de parametrización: el uso de archivos YAML para ajustar Ganancias PID, deadzone, límites de velocidad y frecuencia de muestreo sin necesidad de recomilar, junto con mecanismos de seguridad ante la pérdida de línea o detecciones erráticas. De este modo, nuestra solución cumple con los requisitos de precisión, reactividad y tolerancia necesarios para un seguidor de línea autónomo capaz de respetar semáforos.

### **1. Desarrollo del nodo de detección de la línea:**

Para extraer el error lateral de la línea de forma robusta, en este caso solo habrá corrección de manera angular, el movimiento linear no tendrá corrección. El nodo realiza una secuencia de pasos de procesamiento de imagen y análisis de formas. A continuación, se describe cada etapa y su propósito, evitando detalles de sintaxis.

### 1. Ajuste de orientación

- a. La cámara puede estar montada al revés o inclinada, por lo que la imagen se gira  $180^{\circ}$  para garantizar que la línea siempre aparezca “derecha” en nuestro sistema de referencia.
- b.

### 2. Recorte de la Región de Interés (ROI)

- a. Se define una ventana rectangular en la parte inferior central de la imagen, eliminando las zonas superiores y laterales que suelen contener piso, paredes u otros objetos irrelevantes, en nuestro caso como solo detectaremos la linea central, queremos que no detecte las lineas de los exteriores.
- b. Dos parámetros ajustables controlan:
  - i. Cuánto bajar desde el borde superior (altura inicial de la ROI).
  - ii. Cuánto recortar de cada lado horizontal (ancho de los márgenes).

### 3. Reducción de color y ruido

- a. Convertimos la ROI a escala de grises para descartar información de color y centrarnos en la intensidad.
- b. Aplicamos un suavizado Gaussiano, que atenúa el “grano” y el ruido de alta frecuencia, facilitando así la posterior separación de la línea del fondo.

### 4. Ecualización local de contraste (CLAHE)

- a. En condiciones de iluminación variable, pequeñas zonas pueden quedar demasiado oscuras o quemadas.
- b. La ecualización adaptativa (CLAHE) mejora el contraste de cada bloque local de la imagen sin exagerar los ruidos, haciendo que la línea sea más homogéneamente visible.

### 5. Segmentación por umbral de Otsu

- a. Calculamos automáticamente el nivel de gris óptimo que separa el fondo de la línea.
- b. Invertimos la máscara resultante para que la línea (más oscura) quede representada como píxeles blancos sobre fondo negro, facilitando el análisis binario.

## 6. Cierre morfológico

- a. Conectar fragmentos de la línea y llenar pequeños huecos que hayan quedado tras la binarización.
- b. Se usa un elemento estructurante cuadrado, aplicando primero una dilatación y luego una erosión, repitiendo varias veces si es necesario.

## 7. Extracción de contornos

- a. Se identifican todos los contornos (bordes de regiones negras).
- b. Cada contorno representa un posible fragmento de la línea tras la binarización y limpieza.

## 8. Filtrado de contornos relevantes

- a. Para descartar artefactos y objetos ajenos, se eligen solo aquellos contornos cuya base (extremo inferior) toca o queda muy cerca del borde inferior de la ROI.
- b. De esta forma, aseguramos analizar únicamente las secciones de pista efectivamente presentes en la zona donde el robot “espera” encontrarla.

## 9. Cálculo de centroides

- a. Para cada contorno válido, se calcula su centro de masa:
  - i. El “momento” espacial  $m_{00}$  corresponde al área del contorno.
  - ii. El momento  $m_{10}$  acumula la posición horizontal de cada píxel del contorno.
  - iii. El cociente  $m_{10}/m_{00}$  da la coordenada X del centroide.
- b. Esto convierte cada trozo de línea en un punto representativo, independiente de su forma exacta.

## 10. Selección y promedio de centroides

- a. No basta con tomar un único contorno: la pista puede mostrar dos bordes cercanos al centro.

- b. Se ordenan los centroides por cercanía al eje medio de la ROI y se eligen los dos más próximos.
- c. El promedio de esas dos coordenadas X produce una estimación suave y estable de la posición central de la línea.

## 11. Cálculo del error lateral

- a. Se mide la diferencia en píxeles entre el centro de la ROI y la posición promedio de la línea.
- b. Para facilitar el control, este desplazamiento se normaliza dividiéndolo por la mitad del ancho de la ROI, dando un valor entre -1 (mucha desviación a la derecha) y +1 (mucha desviación a la izquierda).
- c. Finalmente, este valor de error normalizado se publica para que el controlador lo utilice en su ciclo PID.

Cada uno de estos pasos se basa en principios de visión por computadora: filtrado para reducir ruido, binarización para segmentar la pista, morfología para limpiar la máscara, y momentos de imagen para extraer características geométricas. El uso de centroides y la selección de los más cercanos al centro garantiza un seguimiento estable, incluso cuando la pista está compuesta de múltiples trazos o presenta discontinuidades.

## 2. Desarrollo del nodo del controlador:

Se describe con detalle cada etapa necesaria para que el nodo controlador mantenga el Puzzlebot alineado con la pista y reaccione adecuadamente a las señales de semáforo. Se enfatiza que solo existe lazo cerrado (PID) en la componente angular; la velocidad lineal se regula en lazo abierto mediante rampas y escalado.

## I. Preparación y parametrización

### 1. Definición de objetivos de control

- a. **Angular:** minimizar el error lateral de la línea con un PID que actúe sobre la velocidad de giro.
- b. **Lineal:** mantener una velocidad suave y segura, ajustada según la corrección angular y el semáforo, sin emplear PID.

## 2. Inicialización de estructuras

Variables de estado:

- a. error\_actual, prev\_error (único histórico necesario).
- b. integral (suma acumulada).
- c. current\_speed, target\_speed, prev\_semaphore.

Cálculo de intervalo de control  $\Delta t = 1 / \text{Hz\_control}$ .

## II. Adquisición de datos

### 1. Callback de error lateral

- a. Cada mensaje contiene el error normalizado  $\in [-1,1]$ .
- b. Al llegar, desplaza prev\_error  $\leftarrow$  error\_actual y almacena la nueva lectura en error\_actual.

### 2. Callback de semáforo

- a. Se recibe un entero (0=amarillo, 1=rojo, 2=verde).
- b. Si cambia respecto a prev\_semaphore, asignar nuevo target\_speed:
  - i. Verde  $\rightarrow v_{\text{default}}$  (velocidad normal)
  - ii. Amarillo  $\rightarrow v_{\text{default}} / 3$  (desacelerar hasta un tercio de la velocidad normal)
  - iii. Rojo  $\rightarrow 0$  (alto total)
  - iv. Sin clase  $\rightarrow$  seguir a la velocidad normal
- c. Guardar el estado en prev\_semaphore para evitar repeticiones.

## III. Bucle de control (cada $\Delta t$ segundos)

### 1. Actualización de velocidad lineal por rampa

- a. Si current\_speed < target\_speed, sumar step\_rampa hasta alcanzar target\_speed (rampa de aceleración).
- b. Si current\_speed > target\_speed, restar step\_rampa hasta bajarlo (rampa de desaceleración).

### 2. Aplicación de zona muerta al error

- a. Si  $|\text{error\_actual}| < (\text{deadzone \%})$ , asignar error = 0 para evitar correcciones por ruido o vibraciones.

### 3. Cálculo del PID angular

- a. **Proporcional:**  $P = K_p \cdot \text{error\_actual}$
- b. **Integral:**
  - i. integral += error\_actual  $\cdot \Delta t$
  - ii. Limitar integral a un rango seguro (anti-windup).
- c. **Derivativo:**  $D = K_d \cdot ((\text{error\_actual} - \text{prev_error}) / \Delta t)$
- d. **Salida angular:**  

$$\omega = P + K_i \cdot (\text{integral}) + D$$

- e. **Saturación angular:**

$$\omega = \text{clamp}(\omega, -\omega_{max}^{[f_0]}, \omega_{max}^{[f_0]})$$

Actualizar prev\_error  $\leftarrow$  error\_actual.

#### 4. Mecanismo de parada total

- a. Si target\_speed == 0, forzar  $\omega = 0$  y  $v = 0$ , garantizando detención instantánea ante semáforo rojo.

#### 5. Escalado de la velocidad lineal según giro

Para mantener estabilidad en curvas:

$$v = \text{current\_speed} \times (1 - |\omega| / \omega_{max}^{[f_0]})$$

- a. Saturación lineal:

$$v = \text{clamp}(v, v_{min}^{[f_0]}, v_{max}^{[f_0]})$$

#### 6. Publicación de comando Twist

- a. Combinar linear.x = v y angular.z =  $\omega$  y publicar inmediatamente, asegurando frecuencia constante de ejecución.

### IV. Robustez y diagnósticos

#### 1. Anti-windup

- a. Limitar el término integral para evitar que arrastre la salida tras correcciones grandes.

#### 2. Detección de pérdida de señal de error

- a. Si no llega un nuevo error en N ciclos, considerar detenerse o girar suavemente en sitio hasta recuperarlo.

#### 3. Protección ante saturaciones

- a. Controlar que current\_speed,  $\omega$  y v nunca superen sus límites físicos.

#### 4. Registro de eventos

- a. Loggear cambios de semáforo, saturaciones o “sin error” para facilitar el ajuste de parámetros durante las pruebas.

### Valores para los parámetros del controlador:

*Tabla 1 Parámetros*

Parámetro	Valor
control.error_topic	/line_detector/error
control.cmd_vel_topic	/cmd_vel
control.loop_hz	30.0
control.speed.default	0.06
control.pid.Kp	0.3
control.pid.Ki	0.007
control.pid.Kd	0.095

control.max_v	0.15
control.min_v	0.03
control.max_w	0.7
control.error_deadzone	0.0

Estos parámetros Tabla 1 Parámetros, se establecieron de tal manera que el bot, siga la línea de la mejor manera posible, tanto las variables de velocidad como del PID, ya están sintonizadas para ser lo más robustas posibles.

### **3. Desarrollo del nodo de detección de semáforo en base a Yolo:**

#### **Entrenamiento del modelo detección de semáforo (YOLOv8):**

Para empezar con el entrenamiento del modelo, lo primero que hicimos fue tomar alrededor de 315 imágenes de cada clase (“verde”, “amarillo”, “rojo”), con la cámara de un celular y con la cámara del Puzzlebot como se ve en la Ilustración 5 como se ve en la Ilustración 4. Una vez que hicimos esto con ayuda de roboflow , etiquetamos cada clase en cada una de las 950 imágenes. Después, hicimos un “data augmentation” donde aplicamos filtros de iluminación, de ruido, de rotación, de distorsión y de contraste para hacer mas robusto el modelo, esto hizo que las 950 imágenes se multiplicaran por 7, al final el modelo termino con 6650 imágenes de entrenamiento.

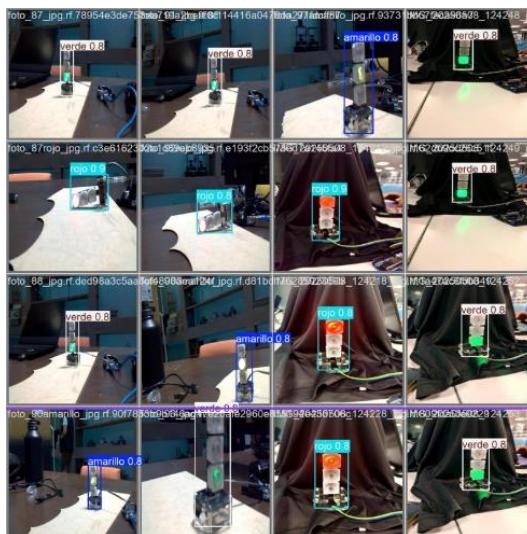


Ilustración 4 Predicción en Roboflow

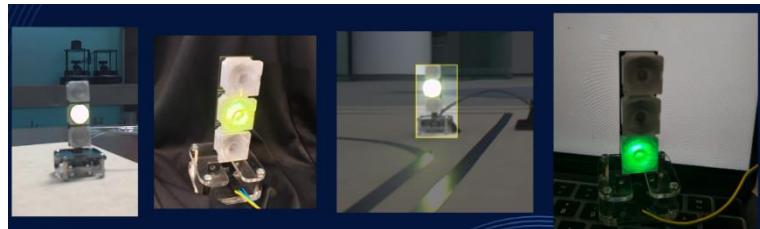


Ilustración 5 Entornos de fotos tomadas

A continuación, se describe, paso a paso, la lógica implementada en el nodo yolov8\_detector.py para detectar el semáforo y comunicar su estado al controlador. Cada sección está planteada como parte de una metodología de desarrollo para detectar el semáforo con el modelo Pytorch.

## I. Definición de objetivos

1. Detectar el color actual del semáforo (verde, amarillo o rojo) a partir de la imagen de la cámara.
2. Publicar únicamente cuando haya un cambio de estado, para ahorrar tráfico en la red y evitar repeticiones innecesarias.
3. Suavizar la carga de cómputo reduciendo la frecuencia de inferencia, de modo que el sistema siga siendo reactivo pero no sature la CPU/GPU.
4. Mantener compatibilidad con el resto de nodos ROS 2 (línea y controlador) usando tipos de mensaje estándar (Int8 para el semáforo).

## II. Preparación y configuración inicial

1. **Carga del modelo YOLOv8 entrenado**
  - a. Indico la ruta exacta al fichero de pesos (lastfinal.pt), garantizando que el modelo se cargue en memoria antes de comenzar la inferencia.
2. **Instanciación de los publishers y subscribers**
  - a. Suscriptor a la imagen cruda desde /video\_source/raw.
  - b. Publicador de mensajes Int8 en /semaphore\_state.
  - c. Opcional: publicador de la imagen con bounding-boxes para depuración en /inference\_result.
3. **Configuración del temporizador de inferencia**
  - a. Fijo un período (por ejemplo, 0.33 s → 3 Hz) para invocar la inferencia de manera periódica, independientemente de la llegada de imágenes, evitando sobrecarga por callbacks continuos.

## III. Adquisición y almacenamiento de la imagen

### 1. Callback de imagen

- a. Cada vez que llega un mensaje de cámara, convierto el ROS Image a un arreglo OpenCV y lo rotulo 180° si es necesario.
- b. Guardo la última imagen válida en una variable interna para que la use el temporizador de inferencia.

## IV. Ejecución periódica de la inferencia

### 1. Invocación del modelo YOLO

- a. En cada tick del temporizador, paso la imagen almacenada al objeto model, obteniendo una lista de resultados (cajas detectadas).

### 2. Selección de la primera detección de semáforo

- a. Recorro las cajas devueltas; al primer hallazgo extraigo su clase (cls) y la convierto a entero.
- b. Si no hay detecciones, no publico nada y mantengo el último estado.

## V. Lógica de publicación y cambio de estado

### 1. Comparación con el estado previo

- a. Mantengo una variable interna prev\_class.
- b. Sólo proceso y publico cuando la nueva clase (new\_class) difiere de prev\_class.

### 2. Interpretación de la clase

- a. 2 → verde, 0 → amarillo, 1 → rojo.
- b. En caso de hallar una clase fuera de este rango, lanzo una advertencia y asumo “verde” por defecto.

### 3. Publicación de mensaje Int8

- a. Creo un mensaje con el valor de la clase y lo envío al tópico /semaphore\_state.
- b. Actualizo prev\_class = new\_class.

## VI. Feedback y depuración

### 1. Logs informativos

- a. Ante cada publicación, emito en consola un mensaje con emoji y nombre de la clase detectada (  ,  ,  ).

### 2. Imagen con cajas dibujadas (opcional)

- a. Si lo necesito para evaluar el rendimiento, genero un frame con bounding-boxes y lo publico en /inference\_result.

## VII. Ajustes de rendimiento

### 1. Frecuencia de inferencia

- a. Elegimos 3 Hz tras medir que a 10 Hz caía el rendimiento del robot; este valor mantiene la latencia baja sin saturar la Jetson.

### 2. Tamaño de imagen

- a. Si el procesamiento es lento, reducimos la resolución de entrada antes de la inferencia para acelerar el paso por la red.

### 3. Manejo de imágenes faltantes

- a. Si la cámara falla o la imagen está corrupta, el nodo sigue funcionando con la última imagen válida y emite una advertencia en logs.

### 4. Baja de resolución de la cámara:

Al correr ambos programas, puede llegar a ser muy pesado para nuestra Jetson Nano, por lo que se redujo la resolución de la cámara a 480x240, por lo que, tanto el YOLO como el seguidor de línea empiezan a procesar imágenes más ligeras, las cuales permiten tener una comunicación correcta entre ellas. Además, para mejorar el trabajo de estos archivos, decidimos correr el launch del seguidor de línea y el controlador, directamente en la Jetson Nano, y por otro lado, la red neuronal desde nuestro ordenador, de esta manera es más eficiente para el programa y menos pesado para la Jetson.

### 5. Pista a usar

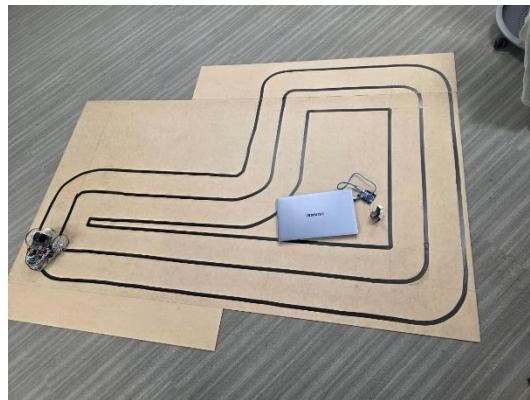


Ilustración 6 Diseño de la pista

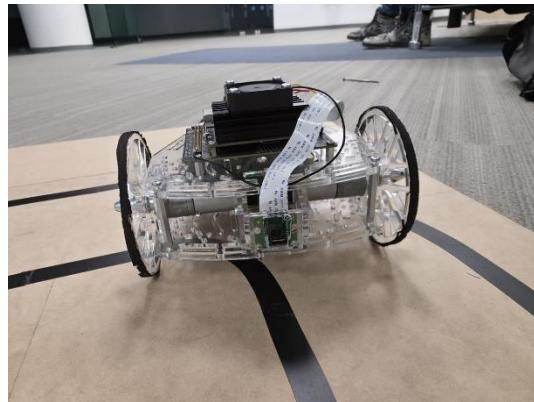


Ilustración 7 Posición de la cámara

Como vemos en la **Error! No se encuentra el origen de la referencia.**, se realizó un circuito en el cual nuestro Puzzlebot se moverá, además la posición de la cámara fue modificada, como se ve en la **Error! No se encuentra el origen de la referencia.**, esto para que la cámara tenga más “tiempo” para detectar las curvas, y que se hagan los movimientos lo más cercano a la curva.

## Resultados

En esta sección presentamos los resultados obtenidos tras la integración y puesta a prueba de nuestro sistema de seguimiento de línea con detección de semáforo. Mostramos primero el grafo de nodos (`rqt_graph`) y el diagrama de flujo que ilustran la arquitectura de mensajes y la secuencia de procesamiento. A continuación, incluimos capturas y videos representativos en los que se observa la respuesta del robot ante distintos estados del semáforo (sin semáforo, rojo, amarillo y verde), así como el seguimiento de la línea en escenarios rectos y curvos. Finalmente, presentamos la gráfica del error lateral a lo largo de la trayectoria, acompañada de un breve análisis cuantitativo de su evolución, que permite evaluar la precisión y la estabilidad de la corrección angular implementada. Con estos elementos visuales y métricos validamos el cumplimiento de los objetivos de precisión, reactividad y robustez, y sentamos las bases para futuras mejoras.

### Rqt\_graph:

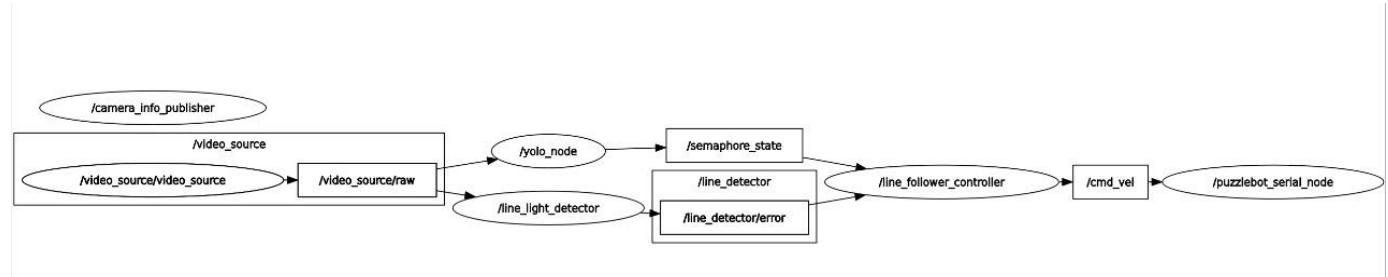


Ilustración 8 Diagrama de nodos

### Diagrama de flujo:

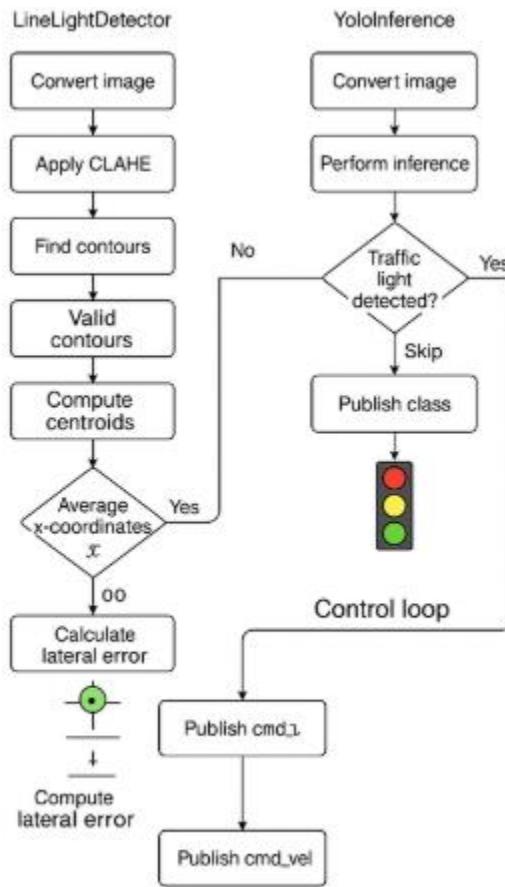


Ilustración 9 Diagrama de flujo del funcionamiento

### Videos de demostración:

Video de funcionamiento del sistema:

[https://drive.google.com/drive/folders/1LFqY7yWr9X8Y443Z3XR0uJjQ1gBcc92q?usp=drive\\_link](https://drive.google.com/drive/folders/1LFqY7yWr9X8Y443Z3XR0uJjQ1gBcc92q?usp=drive_link)

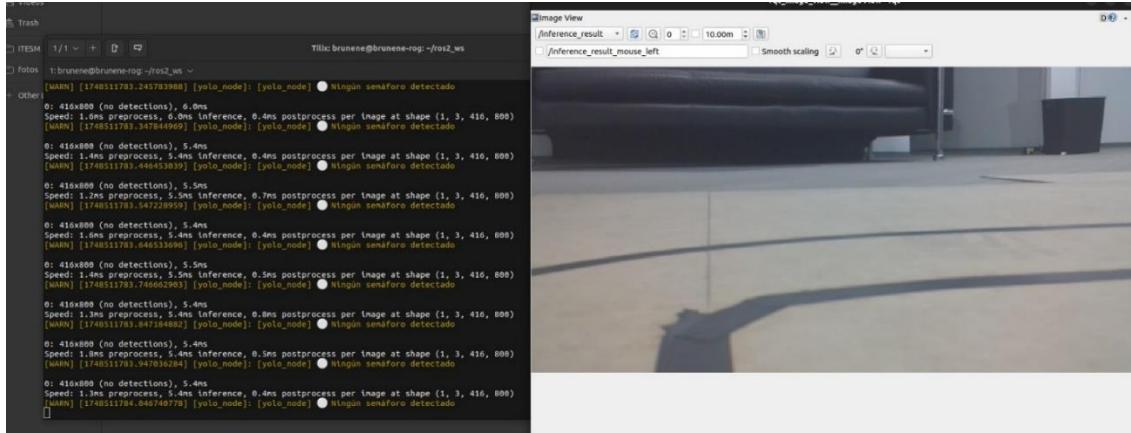


Ilustración 10 Detección sin semáforo

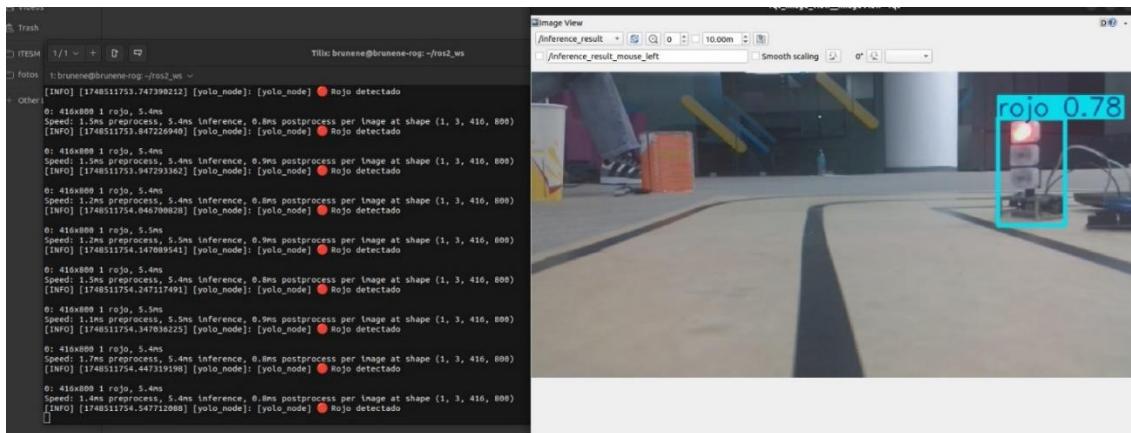


Ilustración 11 Detección de semáforo rojo

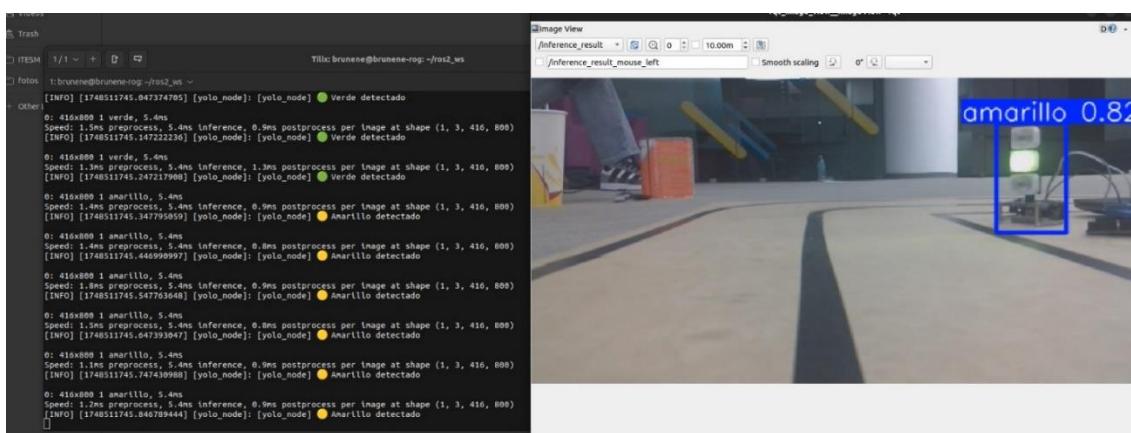


Ilustración 12 Detección de semáforo amarillo

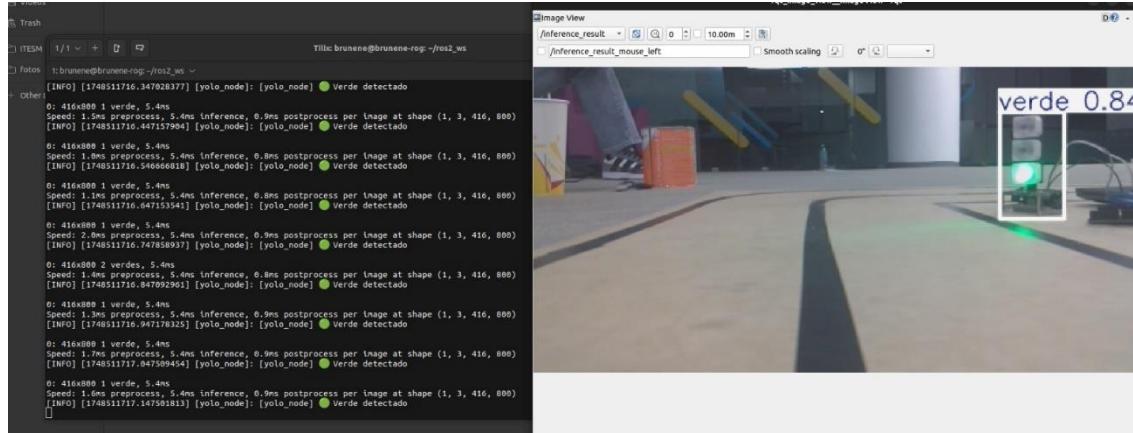


Ilustración 13 Detección de semáforo verde

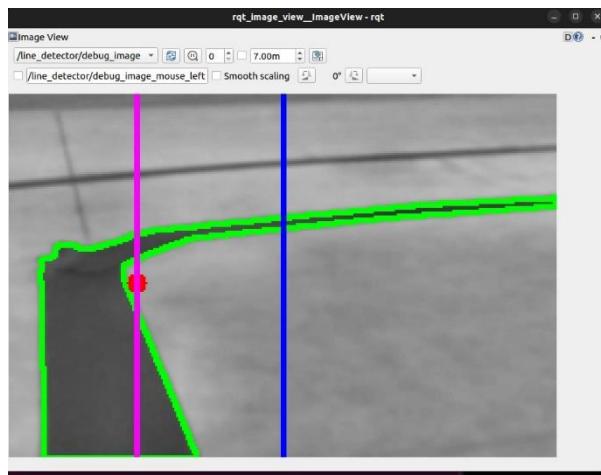


Ilustración 14 Detección de linea

Como podemos ver en Ilustración 10, Ilustración 11, Ilustración 12, Ilustración 13 y Ilustración 14 nuestro modelo fue capaz de identificar el semáforo y la ausencia de este, para regular las velocidades y la trayectoria del modelo a seguir.

### Grafica del error:

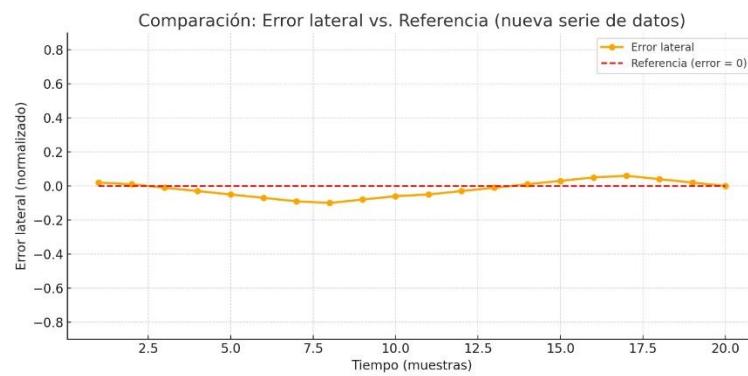


Gráfico 1

## Conclusiones

Se logró implementar un sistema funcional de navegación autónoma en el Puzzlebot usando ROS2, seguimiento de líneas y detección de semáforos con visión artificial. Los objetivos principales fueron cumplidos, ya que el robot pudo seguir la línea y reaccionar adecuadamente a los distintos estados del semáforo.

Sin embargo, en condiciones de mucha iluminación o visibilidad parcial de la línea, el desempeño fue menos estable. También se observó sensibilidad en la detección del semáforo cuando cambiaba la distancia.

Estas limitaciones se deben a restricciones del hardware y a la falta de predicción en el sistema de control. Como mejora, se propone optimizar el preprocesamiento de imagen, entrenar con datos más variados y considerar técnicas de predicción para una navegación más robusta.

En general, la práctica permitió integrar y comprender los principales componentes de un sistema autónomo basado en percepción visual y control reactivo.

## Bibliografía o referencias

NVIDIA Corporation. (2023). *Jetson Nano Developer Kit User Guide*. Disponible en: <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>

Jocher, G., Chaurasia, A., Qiu, J., & Stoken, A. (2023). *YOLOv8: Real-time Object Detection*. Ultralytics. <https://docs.ultralytics.com/>

*Raspberry Pi Camera Module V2.1 | Módulo de cámara Raspberry Pi, interfaz CSI-2, resolución 3280 x 2464 píxeles, 30fps | RS*. (n.d.). <https://es.rs-online.com/web/p/camaras-para-raspberry-pi/9132664>

*Morphological filters – Bioimage Analysis Training Resources*. (n.d.). - Bioimage Analysis Training Resources. [https://neubias.github.io/training-resources/filter\\_morphological/index.html](https://neubias.github.io/training-resources/filter_morphological/index.html)