



Tecnológico de Monterrey

Reto semanal 4. Manchester Robotics

Carlos Adrián Delgado Vázquez A01735818

Alfredo Díaz López A01737250

Juan Paulo Salgado Arvizu A01737223

Bruno Manuel Zamora García A01798275

15 de mayo del 2025

Resumen

Se desarrolló un sistema en ROS2 para que el Puzzlebot navegue hacia una posición deseada usando odometría y controladores P, PI o PID. Además, se integró visión por computadora para detectar semáforos y modificar su comportamiento en tiempo real. El sistema combina localización, control y percepción visual para lograr una navegación autónoma más precisa y reactiva.

Objetivos

Objetivo principal

Desarrollar un sistema de navegación autónoma para el robot Puzzlebot en ROS2 que permita alcanzar posiciones deseadas con precisión mediante controladores, utilizando información de odometría y detección visual para adaptar su comportamiento en tiempo real y la reacción a un semáforo.

Objetivos particulares

- Calcular el error entre la posición actual y una meta deseada en tiempo real.
- Diseñar y probar controladores P, PI y PID para minimizar el error de posición.
- Integrar una cámara para capturar imágenes del entorno en tiempo real.
- Detectar colores y formas, específicamente semáforos, utilizando visión por computadora.
- Modificar el comportamiento del robot según el color detectado del semáforo.

Introducción

El controlador PID es una técnica de control en lazo cerrado que ajusta la salida del sistema en función del error entre una posición deseada y la posición actual. Combina tres acciones: una proporcional al error, otra que considera el acumulado del error (integral) y una más que anticipa su cambio (derivativa) (*Pid - ROS Wiki*, n.d.).

En un robot móvil diferencial como el Puzzlebot, el PID, Ilustración 1, se usa para corregir tanto la distancia como la orientación hacia un punto objetivo. A partir de la odometría, se calcula el error de posición y se generan comandos de velocidad lineal y angular para las ruedas, permitiendo que el robot se acerque con precisión al objetivo. Esta estrategia mejora la estabilidad del movimiento y reduce el error final, especialmente frente a pequeñas perturbaciones o desviaciones.

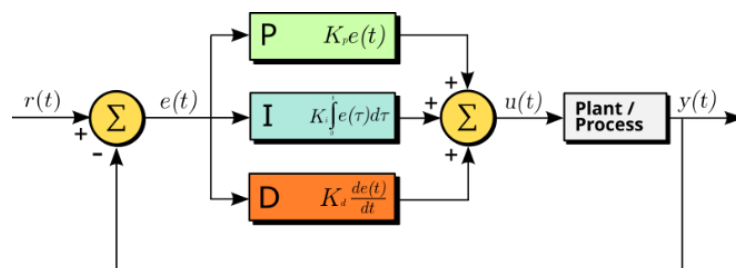


Ilustración 1 PID

La robustez de un controlador se refiere a su capacidad para mantener un rendimiento adecuado aun cuando existen incertidumbres, perturbaciones o variaciones en el sistema. En otras palabras,

un controlador robusto es capaz de seguir cumpliendo su objetivo (como llevar al robot a una posición deseada) aunque las condiciones reales no sean exactamente las previstas, por ejemplo, si hay errores en la odometría, deslizamiento de ruedas o cambios en el peso del robot.

El procesamiento de imágenes en tarjetas embebidas consiste en ejecutar algoritmos de visión por computadora directamente en dispositivos compactos de bajo consumo energético, como la NVIDIA Jetson Nano, Raspberry Pi, BeagleBone o similares. Estas tarjetas permiten capturar y analizar imágenes en tiempo real sin depender de una computadora externa, lo cual es fundamental en aplicaciones como la robótica móvil, donde se requiere autonomía y tiempos de respuesta cortos (*Get Started With Jetson Nano Developer Kit*, n.d.).



Ilustración 2 Jetson Nano NVIDIA

Para que una tarjeta embebida como la Jetson Nano, Ilustración 2, pueda procesar imágenes en tiempo real, es fundamental establecer una interconexión confiable con una cámara. Existen varios tipos de interfaces disponibles, dependiendo del modelo de la cámara y las necesidades del sistema: USB (Universal Serial Bus): Es la forma más común y directa de conectar cámaras web estándar (UVC). Es compatible con una gran variedad de dispositivos y fácil de implementar en sistemas operativos como Ubuntu. En la Jetson Nano, basta con conectar la cámara al puerto USB y utilizar herramientas como v4l2 o OpenCV para acceder al video en tiempo real.

CSI (Camera Serial Interface): Es una interfaz de alta velocidad diseñada específicamente para la transmisión de video en sistemas embebidos. En la Jetson Nano, la conexión CSI se realiza mediante el puerto MIPI-CSI ubicado en la placa, que permite conectar cámaras como la Raspberry Pi Camera Module v2, Ilustración 3. Esta opción ofrece mejor rendimiento que USB, ya que utiliza canales dedicados de la GPU y evita la sobrecarga del bus principal (*NVIDIA Jetson Nano*, n.d.).



Ilustración 3 Raspberry Pi Camera Module v2

La segmentación de imágenes es un proceso fundamental en visión por computadora, ya que permite dividir una imagen en regiones o grupos de píxeles con características similares, como

color, textura o intensidad. Su objetivo principal es aislar los objetos de interés del fondo o de otras partes irrelevantes de la imagen, para facilitar tareas posteriores como detección, seguimiento o reconocimiento (*Visión Por Computador _ AcademiaLab*, n.d.)

1. Segmentación por umbral (thresholding):

Consiste en convertir una imagen en escala de grises a una imagen binaria, estableciendo un valor de corte. Si un píxel tiene un valor mayor al umbral, se considera blanco (objeto); si es menor, negro (fondo). Se puede usar umbral adaptativo y método de Otsu, que calcula automáticamente el mejor umbral para separar regiones, como en la Ilustración 4 (*OpenCV: Image Thresholding*, n.d.).

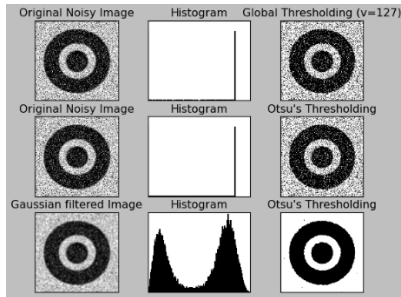


Ilustración 4 Umbrales

2. Segmentación por color en espacio HSV:

El espacio de color HSV (Hue, Saturation, Value), Ilustración 5, permite segmentar objetos de color de forma más robusta que el espacio RGB, ya que separa el tono (color puro) de la intensidad y saturación. Es ideal para detectar objetos como luces de semáforo, señales o marcadores de colores específicos. Se define un rango de valores de tono, saturación y valor, y se aplica una máscara lógica para extraer los píxeles que cumplen con esas condiciones (Valentino, 2024). Se realiza usando filtros HSV con rangos de umbral definidos para cada color. Este método es muy útil en entornos controlados donde los colores tienen significados específicos, como un semáforo rojo (detener) o verde (avanzar). La detección de colores puede combinarse con el cálculo de áreas y momentos para verificar la presencia y ubicación del objeto.

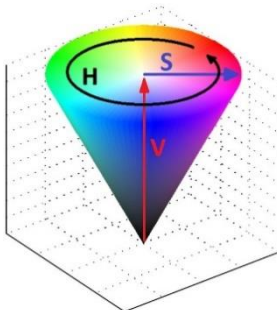


Ilustración 5 Espacios HSV

3. Segmentación basada en regiones o bordes:

Detecta bordes mediante operadores como Canny, Sobel o Laplaciano, y a partir de estos delimita contornos o regiones cerradas, como se ve en la Ilustración 6. Esto es útil para

detectar formas bien definidas o separar objetos por contraste de bordes. Los contornos son líneas cerradas que definen los bordes de los objetos. En OpenCV, se obtienen con funciones como `cv2.findContours()` tras aplicar una binarización. A partir de los contornos, se pueden detectar formas geométricas simples mediante el análisis de número de vértices, aproximación de polígonos, análisis de circularidad o relaciones de aspecto.

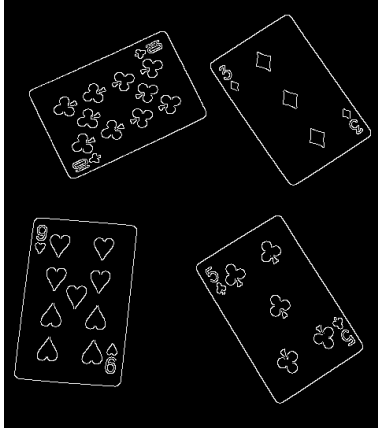


Ilustración 6 Detección de bordes

4. Segmentación por clustering (agrupamiento):
Métodos como K-means o Mean Shift agrupan píxeles con características similares. Se utilizan en tareas más complejas, como segmentación de escenas, donde los objetos no están claramente separados por color o borde.
5. Segmentación semántica o con redes neuronales (Deep Learning):
Utiliza modelos entrenados para segmentar clases específicas de objetos a nivel de píxel. Es muy robusta, pero requiere gran capacidad de cómputo y datos entrenados, lo cual puede ser una limitación en tarjetas embebidas si no se usa una red optimizada como DeepLab, ENet o YOLO-Seg, viéndose como en la Ilustración 7 (Ramos, 2023).

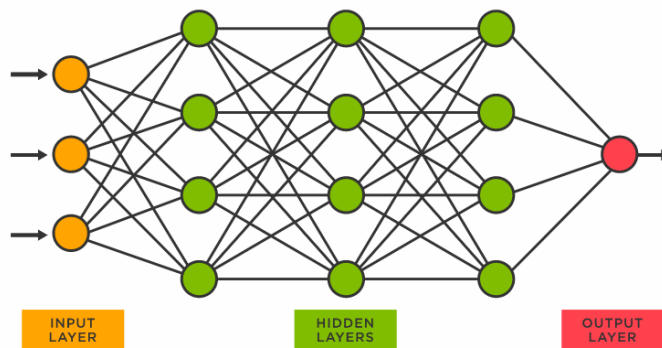


Ilustración 7 Redes neuronales

Para mejorar la robustez de estos métodos, se emplean filtros de suavizado como Gaussian Blur, Median Blur o técnicas morfológicas como erosión y dilatación, que ayudan a reducir el ruido y mejorar la definición de objetos detectados.

La combinación de control robusto y percepción visual permite que el robot móvil no solo se desplace con precisión hacia un punto objetivo, sino que también reaccione ante estímulos visuales del entorno, como la detección de un semáforo, parecido al de la Ilustración 8. Esta integración entre control y visión es esencial para lograr un comportamiento autónomo adaptable, eficiente y funcional en escenarios reales.

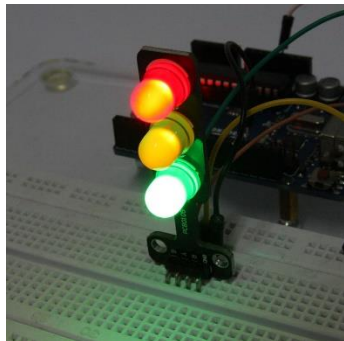


Ilustración 8 Semáforo Adafruit

Con esta base teórica y tecnológica, se procede al diseño y desarrollo de la solución propuesta, en la cual se integran todos estos componentes en un sistema funcional de navegación inteligente.

Solución del problema

Partiendo de la base teórica y tecnológica presentada, en esta sección se expone la solución implementada para el Puzzlebot, que integra el cálculo y compensación de errores de odometría, el diseño de controladores PID, la detección de semáforos mediante visión por computadora y una arquitectura modular de nodos en ROS2. Se detallan los procedimientos para medir y corregir las desviaciones en distancia y ángulo, así como la configuración de filtros y rangos HSV para segmentar con precisión los colores del semáforo. A continuación, se describe la interacción entre los nodos `path_generator`, `color_detection`, `controller` y `serial`, y la lógica de control que adapta dinámicamente la velocidad del robot según el estado de la señal (detenerse, desacelerar o avanzar). Finalmente, se presentan los criterios de parametrización y los formatos de archivos YAML utilizados para definir trayectorias, asegurando un sistema robusto y capaz de reaccionar en tiempo real a las condiciones del entorno.

1. Cálculo de errores de velocidad lineal y angular:

Medición de errores en distancia lineal y angular (giro a la derecha y giro a la izquierda):

Distancia real, Tabla 1 Movimiento lineal:

Tabla 1 Movimiento lineal

Distancia real	Distancia medida (enconders)	Error
1 m	.945 m	5.5
1 m	.943 m	5.7

3 m	2.823 m	5.9
3 m	2.817 m	6.1
5 m	4.687 m	6.3
5 m	4.675	6.5

El error promedio para la distancia lineal es de -5.5%, por lo que para el código final será necesario agregar esa diferencia para compensar el error.

Distancia angular (izquierda y derecha), Tabla 2 Movimiento angular:

Tabla 2 Movimiento angular

Giro a la izquierda			Giro a la derecha		
Ángulo real (grados)	Ángulo medido (encoder)	Error	Ángulo real (grados)	Ángulo medido (encoder)	Error
90	84.11	6.54%	90	81.76	9.15%
180	166	7.3%	180	164.5	7.8%
360	332	7.8%	360	332.5	7.69%

El error promedio para estas pruebas fue de -7.71%, por ello, será necesario compensar este error, agregando esta cifra al ángulo medido para que se acerque lo más posible al ángulo real. El cálculo de los anteriores errores es necesario para que el PID funcione adecuadamente, puesto que si el PID solo recibiera los datos medidos de los encoders tendría mediciones incorrectas porque como se observa tanto a la velocidad lineal como angular miden menos de los que realmente nosotros medimos visualmente.

2. Detección de semáforos en base a espacio de colores HSV:

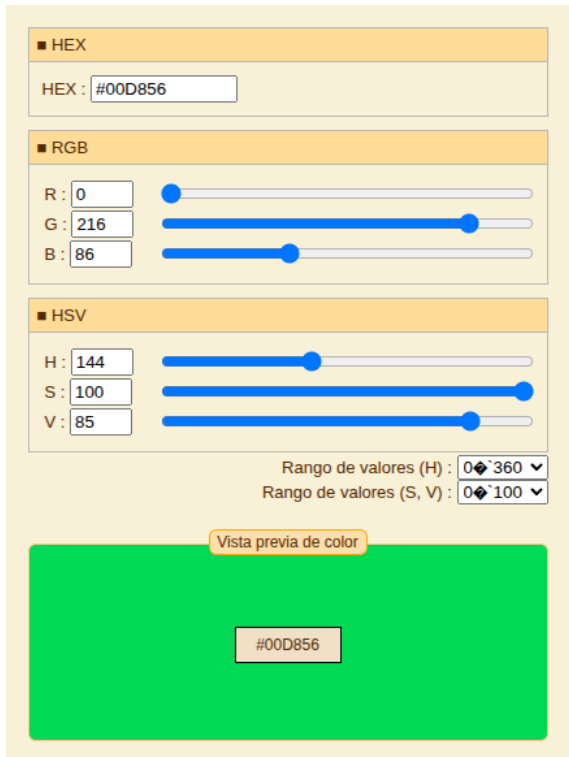
```
# Rangos HSV para rojo (dos rangos), verde y amarillo
self.red_lower1 = np.array([ 0, 100, 100], np.uint8)
self.red_upper1 = np.array([ 10, 255, 255], np.uint8)
self.red_lower2 = np.array([160, 100, 100], np.uint8)
self.red_upper2 = np.array([179, 255, 255], np.uint8)

self.green_lower = np.array([45, 80, 120], np.uint8)
self.green_upper = np.array([75, 250, 255], np.uint8)

self.yellow_lower = np.array([25, 15, 140], np.uint8)
self.yellow_upper = np.array([45, 250, 255], np.uint8)
```

Ilustración 9 Rangos HSV

Haciendo pruebas con el semáforo llegamos a los valores necesarios para que detectara los 3 colores (verde, amarillo y rojo), en un rango de un ambiente con iluminación media a baja, como se ve Ilustración 9. Los valores se obtuvieron con ayuda de una aplicación en línea, esto con el fin de que obtuviéramos los valores exactos, Ilustración 10.



The image shows a web-based color picker interface. It has three main sections: HEX, RGB, and HSV. The HEX section shows the value #00D856. The RGB section shows R: 0, G: 216, and B: 86, each with a corresponding slider. The HSV section shows H: 144, S: 100, and V: 85, each with a corresponding slider. Below these sections, there are two dropdown menus for 'Rango de valores (H)' and 'Rango de valores (S, V)'. At the bottom, there is a large green rectangular preview area labeled 'Vista previa de color' and a small box containing the hex code #00D856.

Ilustración 10 Cálculo de valores HSV

3. Filtros para detección de semáforo:

Se decidió utilizar el espacio de color HSV en lugar del tradicional RGB para la detección de colores del semáforo (rojo, amarillo y verde) debido a su mayor robustez frente a variaciones de iluminación. A diferencia del espacio RGB, donde el color y el brillo están mezclados en los tres canales, HSV separa el matiz (Hue), la saturación (Saturation) y el valor de brillo (Value), lo que permite identificar los colores basándose únicamente en el matiz, independientemente de cuán iluminada esté la escena.

El programa funciona en ROS2 como un nodo llamado `color_detection_node`, cuyo objetivo es detectar el color del semáforo en una imagen capturada por cámara. Primero, se suscribe al tópic `/video_source/raw` para recibir imágenes en tiempo real desde una cámara, utilizando el tipo de mensaje `sensor_msgs/msg/Image`. Estas imágenes se convierten de formato ROS a OpenCV mediante `CvBridge`, y luego se les aplica un filtro Gaussiano para suavizar el ruido. Posteriormente, la imagen se transforma al espacio de color HSV, que permite segmentar de

manera más precisa los colores rojo, amarillo y verde mediante rangos definidos de matiz. En el caso del rojo, se usan dos rangos debido a su posición en los extremos del espectro HSV. A cada máscara binaria generada se le aplican operaciones morfológicas de erosión y dilatación para eliminar ruido y unir regiones. Una vez segmentados los colores, el nodo evalúa cuál tiene mayor presencia en la imagen y determina el color visible del semáforo, siguiendo una jerarquía de prioridad: rojo > amarillo > verde. Cuando el color detectado cambia, el nodo publica esta información en el tópico `detected_color` utilizando mensajes tipo `std_msgs/msg/String`, permitiendo que otros nodos del sistema (como un controlador de robot) puedan reaccionar en tiempo real ante el estado del semáforo detectado.

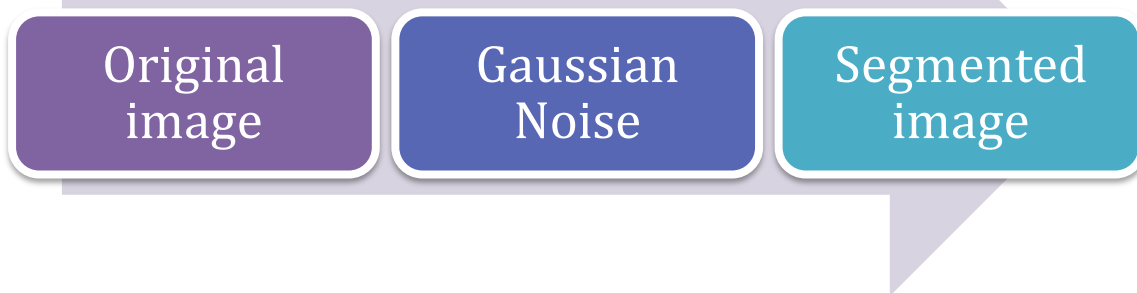


Ilustración 11 Detección color rojo

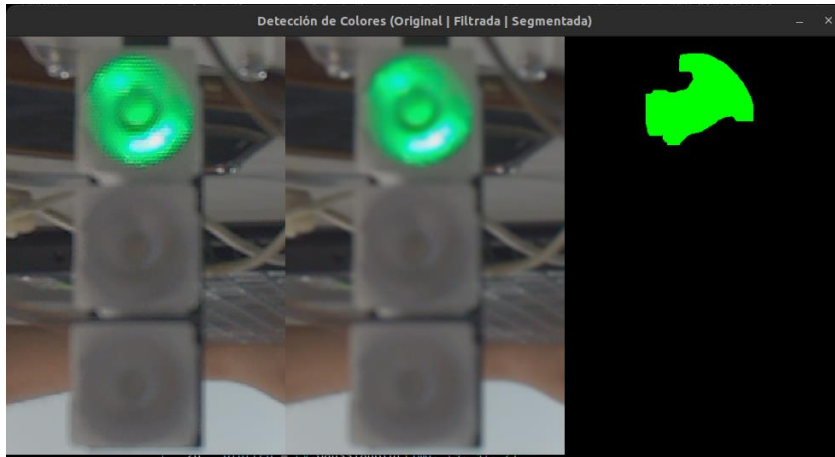


Ilustración 12 Detección color verde

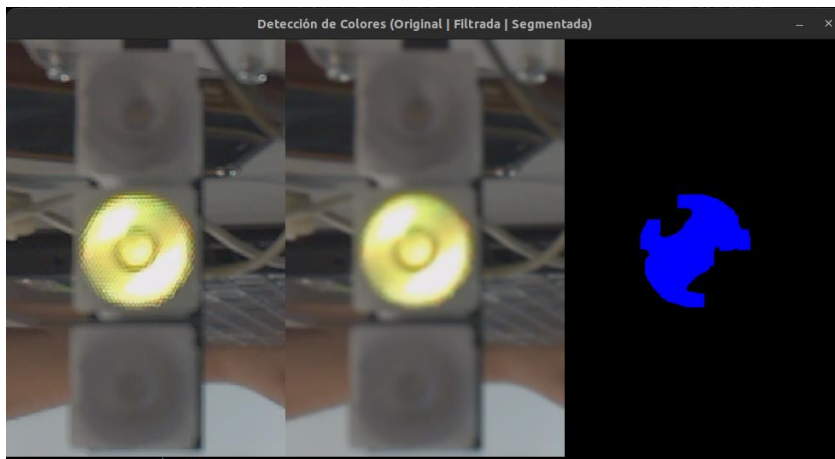


Ilustración 13 Detección color amarillo

4. Funcionamiento de nodos trabajando en conjunto:

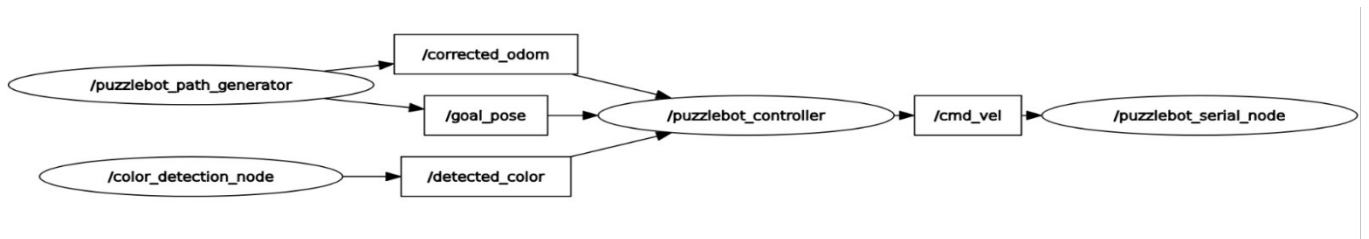


ILUSTRACIÓN 14 DIAGRAMA DE NODOS

Explicación de los nodos en rqt_graph, **Error! Reference source not found.:**

El grafo rqt que se genera muestra cómo se comunican los cuatro nodos principales mediante tópicos de ROS2. Para describirlo se presentaría así:

- Nodo `puzzlebot_path_generator`
 - Publica en `/corrected_odom` (tipo `nav_msgs/Odometry`) la odometría corregida.
 - Publica en `/goal_pose` (tipo `geometry_msgs/PoseStamped`) el siguiente waypoint que debe alcanzar el robot.
- Nodo `color_detection_node`
 - Publica en `/detected_color` (tipo `std_msgs/String`) el color detectado del “semáforo” (rojo, amarillo o verde), tras aplicar filtros HSV y morfología a la imagen.
- Nodo `puzzlebot_controller`
 - Se suscribe a:
 - `/corrected_odom` → para conocer la posición actual.
 - `/goal_pose` → para recibir el objetivo a alcanzar.
 - `/detected_color` → para ajustar su lógica según el color:
 - “red” → detenerse (velocidad = 0)
 - “yellow” → reducir velocidad
 - “green” → aplicar el control PID normal
 - Publica en `/cmd_vel` (tipo `geometry_msgs/Twist`) la velocidad lineal y angular resultante.
- Nodo `puzzlebot_serial_node`
 - Se suscribe a `/cmd_vel` y traduce cada comando Twist en paquetes serie que envía al microcontrolador de los motores.

Flujo general de información:

1. `puzzlebot_path_generator` procesa la odometría cruda y genera los tópicos `/corrected_odom` y `/goal_pose`.
2. `color_detection_node` analiza la imagen de cámara y publica `/detected_color`.
3. `puzzlebot_controller` combina posición, objetivo y estado de semáforo para calcular el comando de velocidad `/cmd_vel`.
4. `puzzlebot_serial_node` recibe `/cmd_vel` y envía los comandos finales al hardware.
5. *Determinación de trayectorias a recorrer (uso de parámetros .yaml):*

Se definio la siguiente trayectoria, en la cual se hace un tipo “rayo” (Ilustración 14) . El rayo avanzará 2 metros, posteriormente gira 90 grados a la izquierda y avanza un metro y

finalmente gira 90 grados a la derecha y avanza otros 2 metros de distancia.

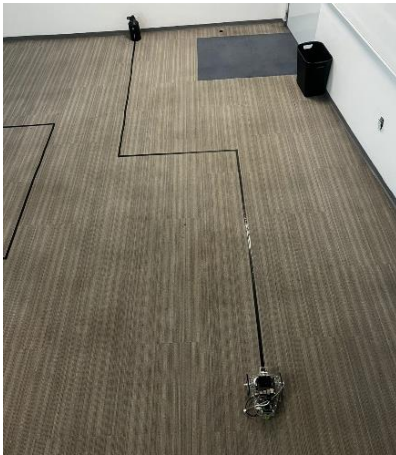


Ilustración 14 Trayectoria (rayo)

6. Lógica de funcionamiento con el semáforo (diagrama de bloques):

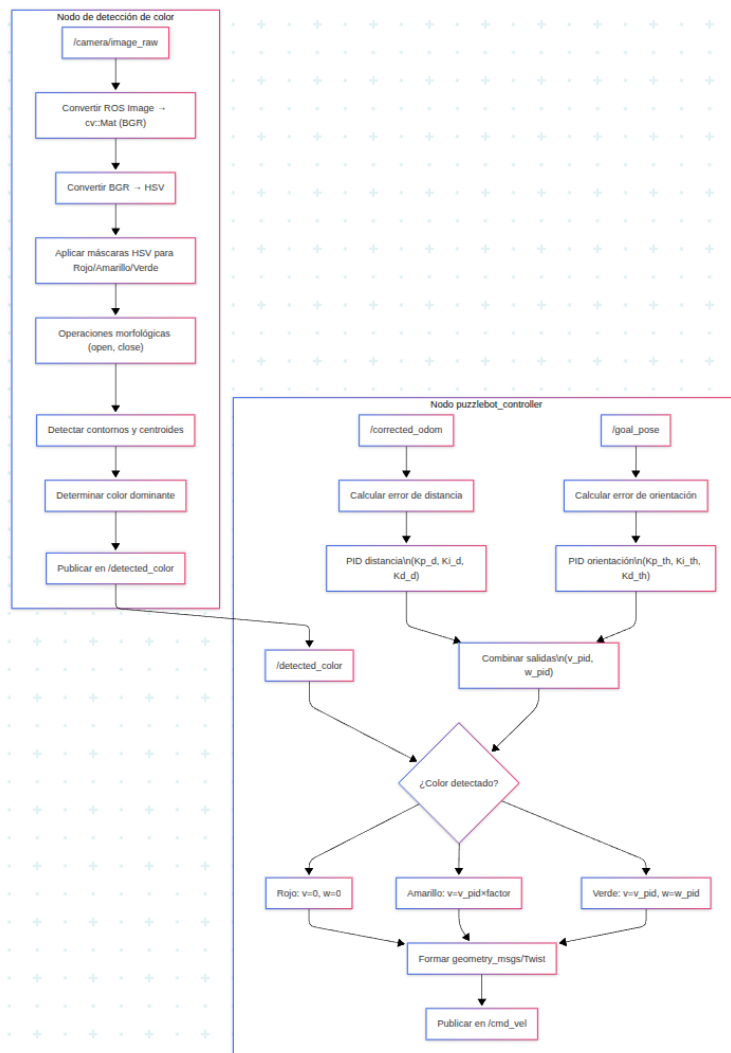


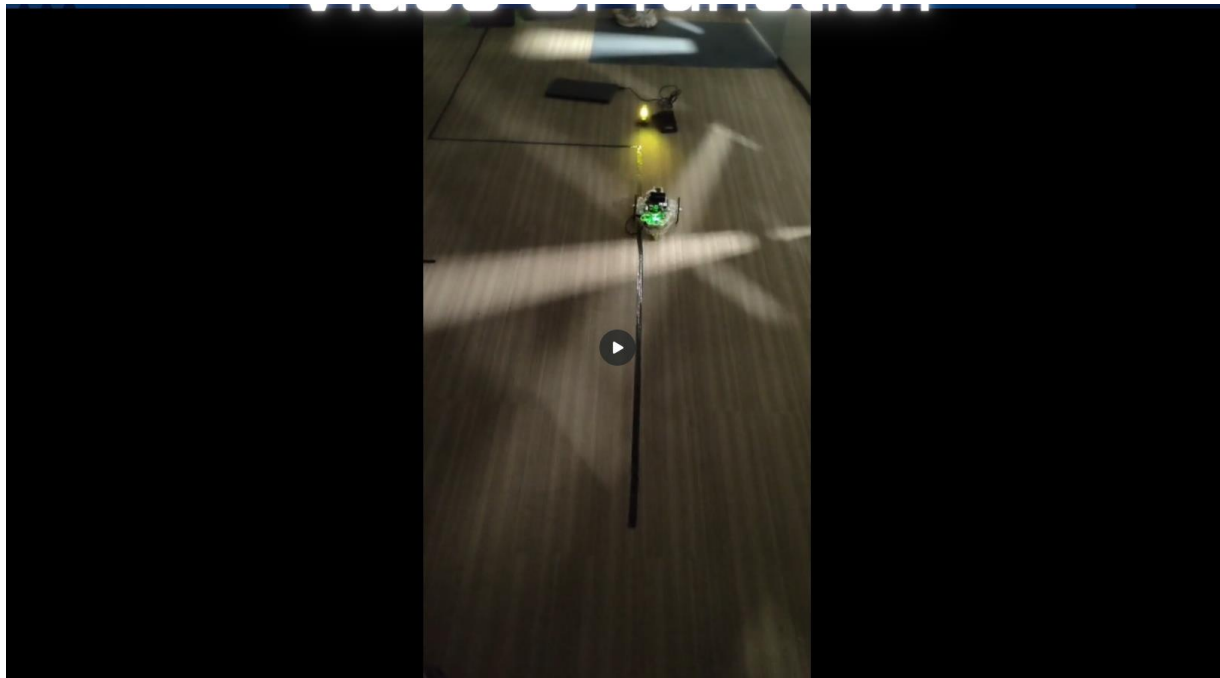
ILUSTRACIÓN 16 DIAGRAMA A BLOQUES DE LA LÓGICA DEL SEMÁFORO

La lógica de funcionamiento basada en el semáforo se articula a través de dos bloques principales: el nodo de detección de color y el nodo de control. El nodo de detección de color captura la imagen proveniente de la cámara, la convierte al espacio HSV y aplica umbrales y operaciones morfológicas para aislar las regiones correspondientes a rojo, amarillo o verde. A partir de esta segmentación, emite un mensaje de tipo `std_msgs/String` indicando el estado actual del “semáforo” y lo publica en el tópico `/detected_color`.

Por su parte, el nodo de control suscribe tanto a la odometría corregida y al waypoint objetivo como al estado de semáforo. En el bloque de conmutación interna, cuando recibe “red” detiene inmediatamente el robot anulando cualquier consigna de velocidad; si detecta “yellow” atenúa la velocidad lineal, limitando la marcha pero manteniendo el cálculo PID para corregir la orientación; y al recibir “green” restablece el lazo de control completo, aplicando la ley proporcional–integral–derivativa sobre el error de distancia y ángulo para guiar el robot hacia su meta. De este modo, el diagrama de bloques muestra cómo el estado del semáforo actúa como una señal de habilitación y ajuste sobre el regulador PID, garantizando que el robot se detenga, desacelere o avance con normalidad según el color detectado. Ilustración 16

Resultados

1. Videos funcionamiento:



En el siguiente link mostramos como nuestro sistema completo funciona:
<https://drive.google.com/file/d/1GGJWRlxAq0s5sIXD1Q6rLrPe1-TaAVeO/view?usp=sharing>

Conclusiones

El proyecto logró cumplir sus objetivos principales: se implementó un controlador PID que permitió al Puzzlebot alcanzar con precisión una posición deseada, y se integró un sistema de visión por computadora en una tarjeta embebida para detectar colores y formas en tiempo real, como semáforos.

Sin embargo, se identificaron limitaciones en la segmentación de imágenes bajo diferentes condiciones de iluminación y en la precisión de la odometría debido al deslizamiento o errores acumulados. Para mejorar estos aspectos, proponemos el uso de redes neuronales ligeras que permitan una detección más robusta y adaptable a variaciones del entorno visual, manteniendo la eficiencia en hardware embebido.

En resumen, se demostró una comprensión adecuada de los procesos de control y percepción visual en robótica móvil, y se sentaron las bases para una solución más avanzada basada en inteligencia artificial.

Bibliografía o referencias

OpenCV: Image thresholding. (n.d.).

https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html

NVIDIA DeepStream SDK. (n.d.). NVIDIA Developer. <https://developer.nvidia.com/deepstream-sdk>

Get started with Jetson Nano Developer Kit. (n.d.). NVIDIA Developer. <https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit>

NVIDIA Jetson Nano. (n.d.). NVIDIA Developer. <https://developer.nvidia.com/embedded/jetson-nano>

Help center. (n.d.). <https://www.mathworks.com/help/control/ug/pid-control-design.html>

pid - ROS Wiki. (n.d.). <https://wiki.ros.org/pid>

Visión por computador _ AcademiaLab. (n.d.). https://academia-lab.com/enciclopedia/vision-por-computador/#google_vignette

Valentino. (2024, November 4). *HSV Color: Descubre qué es y cómo utilizarlo en el arte y el diseño*. Pinturas Chile. https://pinturaschile.cl/que-es-hsv-color/#google_vignette

Ramos, E. (2023, August 15). *¿Qué es una red neuronal y cómo funciona?* Ebac. <https://ebac.mx/blog/red-neuronal>