Reto semanal 4. Manchester Robotics

Oscar Ortiz Torres A01769292

Yonathan Romero Amador A01737244

Ana Itzel Hernández García A01737526

Implementación de Robótica Inteligente

Grupo 501

Jueves 15 de mayo de 2025

Resumen

En este reto se presenta el desarrollo e integración de un sistema de navegación autónoma para el Puzzlebot Jetson Edition utilizando ROS2. El sistema está compuesto por cuatro nodos principales: generación de trayectorias (`pathGenerator`), localización por odometría (`localisation`), identificación de semáforos mediante visión por computadora (*`colorIdentificator`*) y control de movimiento (`controller`). Se implementaron técnicas como control PID y procesamiento de imágenes con OpenCV, empleando el modelo de color HSV para detección de señales visuales. La solución fue validada en un entorno de pruebas delimitado, asegurando el cumplimiento de restricciones de velocidad y respuesta adecuada ante señales de semáforo.

Objetivos

Objetivo general

Desarrollar e implementar un sistema de navegación autónoma para el Puzzlebot Jetson Edition utilizando ROS2, que integre generación de trayectorias, localización, identificación de señales visuales y control de movimiento, con el fin de realizar pruebas dentro de un área definida respetando las restricciones de velocidad y comportamiento ante semáforos.

Objetivos particulares

1. Nodo generador de trayectorias:
    a. Leer un archivo de parámetros que contenga coordenadas de puntos objetivo, rangos de velocidad lineal y angular, y límites del área de trabajo.
    b. Validar que los puntos objetivo se encuentren dentro del área de trabajo y que los rangos de velocidad estén dentro de los valores permitidos para el robot.

   c. Publicar la información procesada en el tópico `/pose` mediante un mensaje personalizado para su uso posterior.

2. Nodo de odometría:
   a. Obtener las velocidades de las ruedas izquierda y derecha del Puzzlebot.
   b. Calcular la pose del robot (x, y, θ) utilizando odometría diferencial.
   c. Publicar la pose estimada en el tópico `/odom` empleando el mensaje estándar `Odometry`.
3. Nodo de identificación de colores:
   a. Capturar y procesar imágenes desde la cámara del Puzzlebot.
   b. Detectar la presencia y color de semáforos dentro del entorno de pruebas.
   c. Publicar un identificador de color en el tópico `/color_id` mediante un mensaje `Int32`, diferenciando entre ausencia de semáforo y los colores rojo, amarillo y verde.
4. Nodo controlador:
   a. Recibir los datos de `/pose` para establecer puntos objetivos y configurar parámetros de velocidad.
   b. Utilizar la información de `/odom` para realizar control en lazo cerrado de las velocidades lineal y angular del robot.
   c. Publicar comandos de movimiento en el tópico `/cmd_vel` mediante mensajes `Twist`, asegurando un seguimiento adecuado de la trayectoria.
   d. Incorporar la lógica de comportamiento ante señales de semáforo detectadas en `/color_id`, ajustando las velocidades o deteniendo el movimiento según corresponda:

   i. Seguir con luz verde
   ii. Disminuir velocidad con luz amarilla
   iii. Detenerse con luz roja).

5. Integrar y validar el sistema completo:
   a. Realizar pruebas en el área de trabajo definida para verificar el correcto funcionamiento de cada nodo de manera individual y en conjunto.
   b. Evaluar el cumplimiento de los rangos de velocidad establecidos.
   c. Validar las respuestas del robot ante la detección de semáforos, asegurando un comportamiento seguro y eficiente.

Introducción

En esta actividad trata sobre el desarrollo y la evaluación de un sistema de control y visión por computadora aplicado a un puzzlebot con una Jetson Nano, ampliamente utilizado en entornos de investigación y automatización. Se le aplica un mecanismo para el seguimiento de trayectorias y la interacción con su entorno con el uso de controladores.

Una de las estrategias más comunes en los sistemas de control y automatización es la implementación de un PID (Proporcional-Integrador-Derivativo), el cual utiliza un mecanismo con retroalimentación en bucle cerrado que ajusta continuamente las salidas en función de la diferencia entre un punto de ajuste deseado y el valor medio.

Operan utilizando tres términos básicos: el proporcional responde al error generando una salida proporcional minimizando rápidamente los errores, el integral aborda el error presente o desviaciones a largo plazo causando que el sistema se acerque al punto de ajuste con precisión y eliminando errores en el estado estacionario. Y el derivativo anticipa los cambios futuros en el

error evaluando su tasa de cambio, ayudando a amortiguar los cambios rápidos en el sistema. (*¿Qué Es un Controlador PID?*, 2025)

Un controlador PID se considera robusto cuando garantiza que el sistema en lazo cerrado mantiene un desempeño estable aun cuando existen errores de modelado o cambios en las condiciones del sistema, cumpliendo con los márgenes mínimos de ganancia y fase teniendo una buena respuesta ante incertidumbres del modelo, con una sensibilidad máxima (Ms) dentro de un rango aceptable como por ejemplo $Ms \approx 1.4$ y de igual forma mantiene su desempeño ante ajustes finos. (S/f)

A la par del control, el procesamiento de imágenes es esencial en la percepción del entorno, uno de los recursos implementados en este reto fue una *Cámara V2 para Raspberry Pi,* módulo oficial diseñado para añadir capacidades de captura de imagen y video. En sus características principales, cuenta con un sensor Sony IMX219, CMOS (Complementary Metal-Oxide-Semiconductor) de 8 megapíxeles.

Con una resolución de imagen de hasta 3280 x 2464 píxeles (8MP). Para la resolución de video cuenta con tres opciones: 1080p a 30 fps, 720p a 60 fps o 640 x 480 a 90 fps.Cuenta con un enfoque fijo y por medio de un conector CSI (Camera Serial Interface) mediante un cable plano de 15 contactos es compatible con la NVIDIA Jetson Nano debido a que ambas esta interfaz. Y sus dimensiones son de aproximadamente 25 x 24 x 9 mm. (Lozano, 2021)

OpenCV es una librería en python de visión por computadora de códigos abierto que se implementa en este reto, ofrece herramientas para tareas como transformación de imágenes, detección de objetos, etc. Incluye operaciones básicas como leer y mostrar imágenes y videos, transformar imágenes (rotar, redimensionar, recortar), cambiar entre espacios de color (RGB,

HSV, etc.), detección de bordes (Canny), Aplicación de máscaras y operaciones bitwise y aplicar filtros y operaciones morfológicas. (Kumari & Kumari, 2023)

La función cv2.HoughCircles() en OpenCV utilizado para detectar círculos, internamente incluye una etapa de detección de bordes mediante el filtro Canny implicando que no es necesario el uso de una aplicación manual de detección de bordes previo al llamado de la función. En sus parámetros se encuentra un umbral superior para el detector Canny, un umbral para la acumulación en la transformada de Hough que controla la sensibilidad en la detección de los círculos, los rangos de los radios a detectar y la resolución del acumulador. (Detectar círculo en imagen binaria de contornos en Python, s/f)

Asimismo, la transformada de Hough es una técnica trascendental de la visión por computadora para la detección de figuras geométricas en imágenes que se describen mediante parámetros matemáticos. Se basa en transformar puntos detectados, es decir bordes, a un espacio de parámetros (espacio de Hough) donde cada punto "vota" por las posibles figuras que podrían pasar. Para la detección de circunferencias se ocupan tres parámetros distintos, centro (a,b) y radio (r) incrementando un acumulador 3D, el cual es la discreción del espacio de parámetros en una matriz. (Sosa-Costa, 2019)

El procesamiento de imagen se implementa el modelo de color HSV (Hue, Saturation, Value), amplicamiente utilizado en la parte de detección y seccmentación de colores ya que nos permite separar el componente de color (matiz) de la intensidad y el brillo, Los valores típicos en OpenCV de los canales son: H(0-179), S(0-255) y V(0-255). Considerando que para el color rojo se usan dos rangos debido a que este matiz está en los extremos de la escala circular. (*Descomponer los Colores de una Imagen*, 2024)

Solución del problema

Diseño e implementación del nodo `pathGenerator`

El nodo se encarga de leer el archivo de parámetros params.yaml y validar que los datos ingresados por el usuario sean correctos, proporcionando retroalimentación en caso de errores.

Primero, se verifica que los vectores `coordenadas_x` y `coordenadas_y` no estén vacíos y que ambos tengan la misma longitud. Luego, se obtienen los límites del sistema de coordenadas del área de trabajo, almacenándolos en las variables `xmin_lim`, `xmax_lim`, `ymin_lim` y `ymax_lim`, a partir de los parámetros `area_largo` y `area_ancho`.

Para validar las coordenadas locales del robot, cada par de coordenadas se transforma al marco de referencia del área de trabajo utilizando una matriz de transformación homogénea. Esta transformación emplea un vector de traslación definido por el parámetro `origen_robot`, y se comprueba que cada punto transformado se encuentre dentro de los límites del área de trabajo.

A continuación, se verifica que los valores de los parámetros `lin_vel` y `ang_vel` estén definidos y se encuentren dentro del rango operativo del robot.

Si durante el proceso se detecta algún error, la función `calculate_path()` se interrumpe. En caso contrario, si todos los datos son válidos, se activa la bandera `path_verified`, la cual es utilizada por la función `publicar_mensaje()` para enviar los mensajes al tópico `/pose`.

Los mensajes de retroalimentación al usuario son los presentados en la Tabla 1.

Tabla 1

Mensajes de retroalimentación del nodo PathGenerator

Mensaje mostrado	Descripción del caso
No se han declarado vectores de coordenadas completos	Aparece cuando las listas de coordenadas `x` o `y` están vacías.
El tamaño de los vectores de coordenadas X y Y no coinciden, hay coordenadas incompletas.	Ocurre cuando las listas `x` y `y` no tienen la misma cantidad de elementos.
El punto (x, y) transformado a (x', y') se encuentra en el rango del área de trabajo	Se muestra cuando una coordenada transformada cae dentro del área permitida de trabajo.
El punto (x, y) transformado a (x', y') no esta dentro de los limites del área de trabajo (x=[xmin, xmax], y=[ymin, ymax])	Se muestra cuando una coordenada transformada cae fuera del área permitida de trabajo.
El valor lineal minimo no esta dentro del rango	El valor mínimo de velocidad lineal está fuera del rango permitido.
El valor lineal maximo no esta dentro del rango	El valor máximo de velocidad lineal está fuera del rango o no es mayor que el mínimo.
Los límites de la velocidad lineal están incompletos	El vector `lin_vel` no tiene exactamente dos elementos.
El valor angular minimo no esta en rango	El valor mínimo de velocidad angular está fuera del rango permitido.
El valor angular maximo no esta en rango	El valor máximo de velocidad angular está fuera del rango o no es mayor que el mínimo.
Los límites de la velocidad angular están incompletos	El vector `ang_vel` no tiene exactamente dos elementos.
Errores en el parameter file (/config/params.yaml)	El sistema detectó errores en los parámetros y por eso no se publica nada.
Publicado mensaje N	El nodo publicó exitosamente la coordenada número N.
Todos los mensajes han sido publicados	Se han enviado todos los mensajes del camino verificado.

Desarrollo del nodo `localisation`

Este nodo es el encargado de implementar la odometría diferencial para la estimación en tiempo real de la posición (x, y) y la orientación θ del Puzzlebot a partir de las velocidades angulares de ambas ruedas.

Su funcionamiento consiste en:

- Suscripción a los encoders: se suscribe a los tópicos `VelocityEncR` y `VelocityEncL`, los cuales contienen las velocidades angulares de la rueda derecha e izquierda.
- Cálculo de las velocidades: se convierten las velocidades de cada rueda en velocidades tangenciales usando el radio de las ruedas y con el uso del modelo cinemático del robot se obtienen las velocidades lineal y angular, utilizando las fórmulas 1 y 2.

$$V = \frac{V_R + V_L}{2} = r\frac{\omega_R + \omega_L}{2} \quad (1)$$

$$\omega = \frac{V_R - V_L}{l} = r\frac{\omega_R - \omega_L}{l} \quad (2)$$

- Estimación de la posición y orientación: se utiliza la velocidad angular y lineal, se actualizan las coordenadas (x,y) y el ángulo θ en cada iteración integrando el delta de tiempo, siguiendo las ecuaciones 3, 4 y 5.

$$\dot{x} = r\frac{\omega_R + \omega_L}{2} \cdot \cos\theta \quad (3)$$

$$\dot{y} = r\frac{\omega_R + \omega_L}{2} \cdot \sin\theta \quad (4)$$

$$\dot{\theta} = r\left(\frac{\omega_R - \omega_L}{l}\right) \quad (5)$$

- Publicación de odometría: los datos estimados de posición y orientación se publica en el tópico `/odom` como mensaje `nav_msgs/Odometry`, de igual manera se incluye la velocidad lineal y angular para el uso de los otros nodos.

Al analizar los datos de la pose publicados por el nodo, se observó la propagación del error en las mediciones calculadas con las integrales numéricas para los tres factores. Para solucionar este problema, se hicieron mediciones para cada uno de los factores, para x y y en un rango de 1 a 4 m y para θ en un rango de 0 a 2π, posteriormente se calculó la proporción del error entre el dato real y el dato medido, finalmente, se obtuvo el promedio entre las 4 mediciones de cada componente. Este procedimiento brindó los resultados de la Tabla 2.

Tabla 2

Datos de pruebas para la corrección de la pose del robot

	Real	**Medido**	**Factor**	**Factor promedio**
Pose | x (m)	1	0.9107	1.09806	1.09997
	2	1.8176	1.10035	
	3	2.7283	1.09959	
	4	3.6335	1.10087	
Pose | y (m)	1	0.8761	1.14142	1.10806
	2	1.81	1.10497	
	3	2.7022	1.11021	
	4	3.6169	1.10592	
Pose | theta (rads)	1.5708	1.3856	1.13366	1.10088
	3.1415	2.8346	1.10827	
	4.71239	4.3095	1.09349	
	6.28319	5.7729	1.08839	

Los valores obtenidos en la columna “Factor promedio” son los valores guardados por las variables `factor_x`, `factor_y` y `factor_th`, los cuales se utilizan para multiplicar los valores calculados con las integrales numéricas en cada iteración de ejecución del nodo.

Implementación del nodo `colorIdentificator`

El nodo de `colorIdentificator`, obtiene el `/raw` del nodo `ros_deep_learning`. Obteniendo la imagen a través del tópico `/image`. Una vez obtenida la imagen esta se procesa, primero se invierte la cámara con flip para poder obtenerla correctamente debido a que la cámara está posicionada de forma invertida debido a limitaciones físicas. Una vez invertida se recorta al 55% de la altura del frame original, obteniendo el área de interés de nuestro Puzzlebot.

Una vez obtenido el frame ajustado convertimos nuestra imagen al formato HSV, esto debido a que este formato de imagen es el mejor para la separación de colores requerido para observar el semáforo y poder detectar en qué color está. Hay que tener en cuenta que en OpenCV los valores HSV son distintos a los normales. Puesto a que en esta librería H está dividido a la mitad, S y V están de `0` al `255` en vez de `0` a `100`.

Teniendo claro la forma en la que se manejan los valores creamos múltiples máscaras para poder hacer el procesamiento de la imagen correspondiente

Tabla 3

Valores de las máscaras en HSV para realizar la visión por computadora

Máscara	H	S	V
White	0 - 92	40 - 255	190 - 255
Red (Bajo)	0 - 5	100 - 255	0 - 255
Red (Alto)	175 - 180	100 - 255	0 - 255
Yellow	25 - 35	15 - 255	165 - 255
Green	60 - 80	100 - 255	100 - 255

Primero nuestro algoritmo crea una máscara de blancos. La función de esta es obtener los píxeles de mayor nivel de intensidad, ya que en este espacio están nuestras fuentes de luz. Esta binarización nos ayuda a reducir el ruido de las siguientes máscaras de colores y delimitando el área. También se aplica un ligero blur, esto para suavizar bordes y eliminar ligeros ruidos que puedan llegar a existir. Para la limitación usamos la función `HoughCircles` en la máscara de blancos la cual usando un detector de bordes obtendrá los centroides de los círculos en la imagen.

Una vez obtenidos los centroides de los círculos estos se expanden de forma que sea el tamaño del radio del círculo detectado más del 10% de este radio. Ya que nuestras fuentes de luz solo ven colores con alta luminosidad por lo que en el caso de los leds de nuestro semáforo será difícil distinguir de que color son por que los pixeles con mayor saturación se ubican al contorno de nuestros círculos por lo que la ampliación del radio ayuda a obtener una mejor precisión.

Obtenidos los círculos se crean 3 máscaras independientes, estas con cada color que se desea segmentar. Recordando que la mascara roja debido al espacio HSV esta dividida en dos

areas de color. Al separar cada color se utiliza la operación `Bitwise_and` con la máscara de los círculos limitando los colores observados a sólo los círculos.

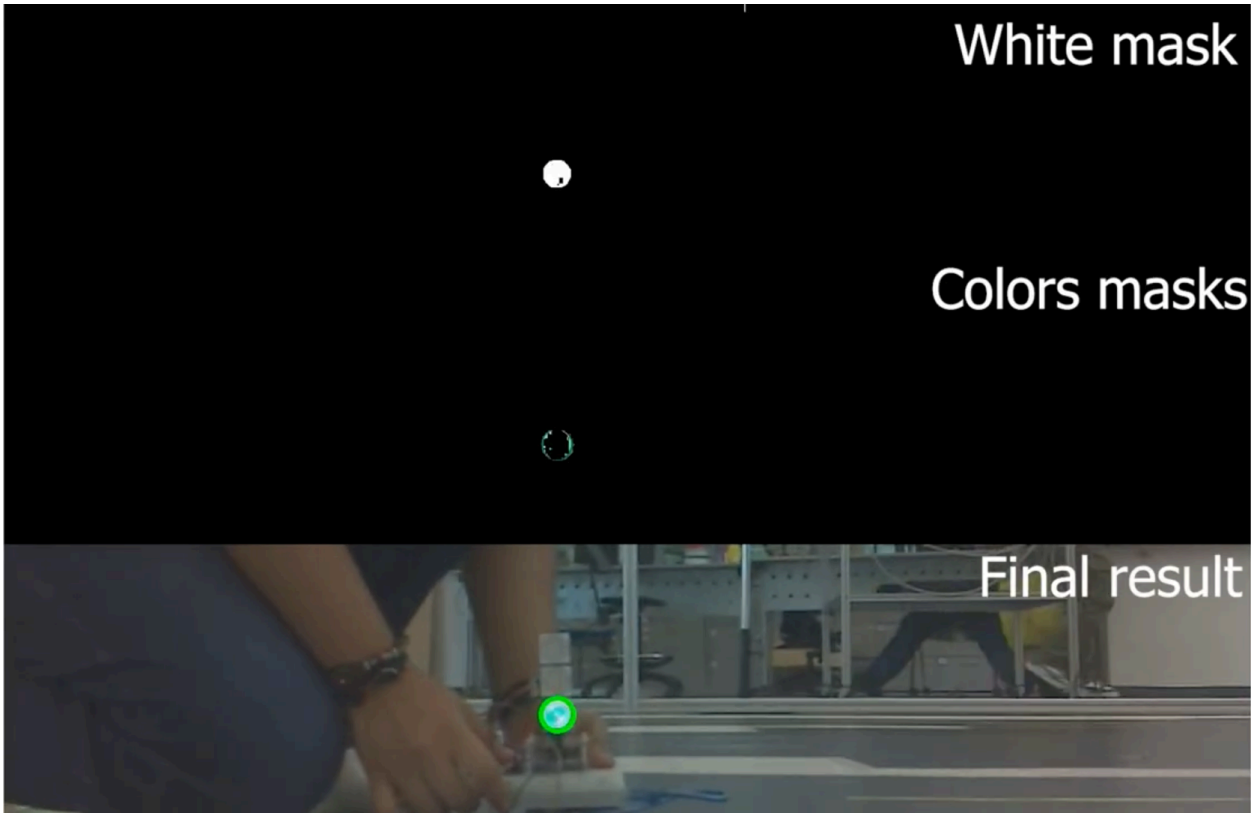

Imagen 1. Captura del vídeo demostrativo, reflejando el procesamiento de imagen.

Una vez obtenidas las máscaras de colores limitadas por círculos, se cuenta cuantos pixeles existen en cada máscara y se obtiene el máximo para determinar cuál color es el más prominente, esto se aplica de dos formas. El robot genera un `color_id` y se publica en el tópico `/color_id`. Esto controlando en qué estado se encuentra nuestro robot, esto tomándolo de forma global en la imagen. También para fines demostrativos se hace que cada círculo se dibuja con el color más predominante en el círculo con un mínimo de 50 pixeles para ser considerado de un color en específico.

Una vez procesado el frame cada círculo se dibuja con su color y se guarda el frame para una vez finalizado se pueda guardar en un archivo MP4.

Desarrollo del nodo `controller`

Este nodo es el nodo encargado de la implementación del controlador en lazo cerrado, tiene la tarea de hacer que el robot siga una trayectoria formada por puntos objetivos, primero implementando una rotación y luego una traslación hacia cada punto con uso de un controlador PID para ambos movimientos.

El flujo de trabajo qué sigue es:

- Recepción objetivos: se subscribe al tópico `/pose` para recibir cada punto destino (msg.path.position.x, msg.path.position.y), velocidad máxima y mínima de la parte lineal y la velocidad máxima y mínima de la velocidad angular. Cada punto se guarda en la variable `path_queue`.
- Lectura de la odometría del robot: se suscribe al tópico `/odom` para obtener la posición (X,Y) y la orientación actual del robot. Con el uso de `transforms3d`, el cuaternión se transforma a los ángulos de Euler, de los cuales se usa únicamente yaw, guardandolo en la variable `Th_robot`.
- Recepción de la identificación del color del semáforo: a través de sus suscripción al tópico `/color_id`, se guarda el valor recibido por el nodo `colorIndentificator` en la variable `color_state`.
    - Para evitar la recepción de ruido en los valores de entrada, se filtra el dato recibido en función del dato actual del color identificado, siguiendo el flujo de los colores del semáforo, asegurando una correcta lógica de seguimiento.

- Ciclo de control: se ejecuta `control_loop()` como la máquina de estados mostrada en la Imagen 2, conmutando entre los estados de 3 variables distintas
    - `robot_busy`: si es `false`, el sistema obtiene el siguiente punto objetivo de `path_queue` y declara valores en las variables para qué el sistema de navegación inicie de manera exitosa. Si es True, entra a la máquina de estados dada por la variable `color_state`.
    - `color_state`: esta varbiable tiene los valores posibles 0 (sin detección de semáforo), 1 (detección de rojo), 2 (detección de amarillo) y 3 (detección de verde). Si es 1, la velocidad lineal y angular se declaran en 0 para detener el movimiento del robot. Si es 0 o 3, entra a la máquina de estados de `mov_state` para controlar el movimiento de trayectoria del robot. Si es 2, entra a una máquina de estados similar a la anterior, dependiendo de `mov_state`, pero en vez de usar los PID, va decrementando gradualmente pero aún tomando en cuenta el punto objetivo, esto para evitar qué siga avanzando en caso de qué llegue al punto requerido.
    - `mov_state`: esta variable conmuta entre los valores 0 (movimiento rotacional), 1 (traslación al objetivo) y 2 (reposo).
        - Si es 0, el sistema calcula el error angular entre la pose del robot y el objetivo, lo envía a la función `pid_controller_angular()`, limita la salida del controlador si en necesario (esto por el rango de velocidades qué soporta el robot) y la almacena en la variable `angular_speed`.
        - Si es 1, el sistema calcula el error tanto de la distancia como del ángulo entre el robot y el objetivo, los envía a las funciones

`pid_controller_lineal()` y `pid_controller_angular()`, limita las salidas si es necesario y las almacena en las variables `linear_speed` y `angular_speed`.

- Si es 0, almacena 0 en las variables `linear_speed` y `angular_speed` para detener completamente el robot.

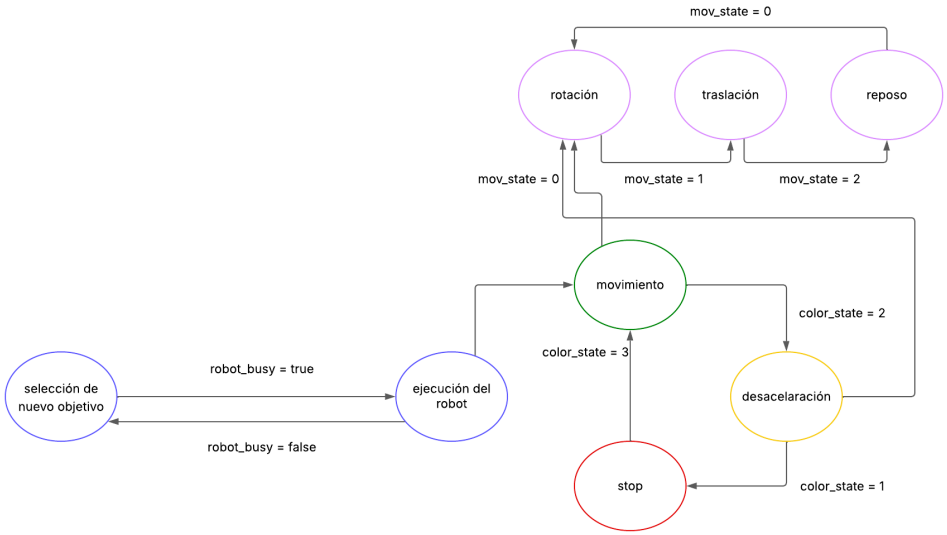

Imagen 2. Flujo de procesamiento de `control_loop()`

- Controlador PID: se utilizan dos, un PID lineal y un PID angular que incluyen anti-windup para la saturación de la parte integral del controlador. Para detener el proceso de control, se hace una comparación de los dos tipos de errores calculados contra valores de errores mínimos, siendo de 0.0488692 radianes y 5 cm, para el error angular y lineal respectivamente.
- Publicación de velocidades: Por último, el nodo publica la velocidad en un mensaje tipo `Twist` en el tópico `/cmd_vel`, donde `angular.z` almacena la variable `angular_speed` para girar y `linear.x` almacena la variable `linear_speed`.

Ejecución del sistema

Se diseñó el archivo `traffic_nav_launch.py` con el propósito de ejecutar todos los nodos necesarios mediante una sola instrucción. Para ello, se empleó la acción `ExecuteProcess` para lanzar los nodos `micro_ros_agent` y `video_source.ros2_launch`, los cuales pertenecen a paquetes externos al desarrollado para la solución del reto.

Dado que estos nodos requieren un tiempo específico para inicializarse correctamente y los nodos desarrollados dependen de la información que publican, se utilizó la acción `TimerAction`. Esta acción permite retrasar la ejecución del nodo `video_source.ros2_launch` por 5 segundos tras el inicio del `nodo micro_ros_agent`, y posteriormente, iniciar los nodos desarrollados del sistema 10 segundos después.

Resultados

Al ejecutar el comando ros2 run rqt_graph rqt_graph, obtuvimos el gráfico de interconexión de los nodo, obteniendo lo mostrado en la Imagen 3, lo cual representa una correcta comunicación entre los módulos del sistema.

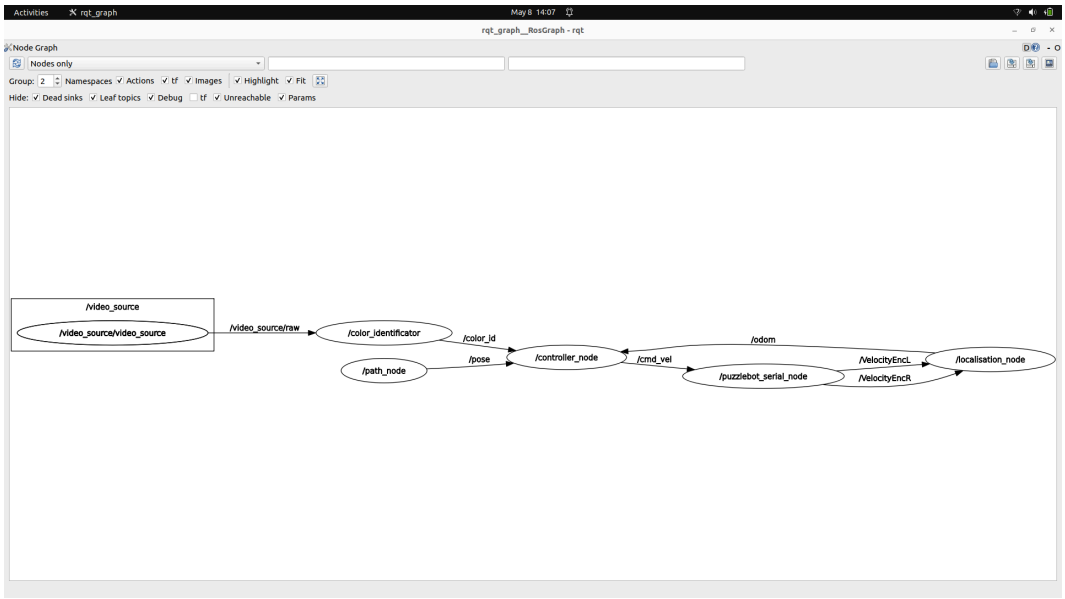

Imagen 3. Diagrama de interconexiones del paquete de ROS2

El sistema se probó en un espacio de trabajo definido, midiendo su largo y ancho, ubicando el origen del robot respecto al espacio a navegar, así como los puntos a seguir y las velocidades en un rango en el qué el robot tuviera tiempo suficiente para identificar los colores.

En el video de funcionamiento de la sección de Anexos se puede observar como el robot sigue satisfactoriamente los puntos objetivos, así como la realización de una correcta identificación de los colores del semáforo con un corto tiempo de respuesta, permitiendo realizar las acciones correspondientes sin retrasos.

Conclusiones

Este sistema de navegación autónoma nos permitió intrigar de forma efectiva los componentes necesarios para la operación coordinada y segura dentro de un entorno controlado, donde cada nodo cumplio con una funcionalidad específica favoreciendo la arquitectura modular y a su vez lograron exitosamente el cumplimiento de los objetivos propios de cada nodo..

Las pruebas confirmaron que el sistema responde de forma consistente ante cambios en el entorno, respetando los rangos de velocidad y modulando su comportamiento de acuerdo con ka señales captadas, la correcta interconexión entre los nodos y el buen desempeño de los robot en un escenario de prueba controlado, donde se buscaba una iluminación constante evitando superficies reflejantes.

Este reto demuestra la importancia de la integración entre percepción, planificación y control así como la necesidad de diseñar sistemas capaces de adaptarse al entorno en tiempo real para una navegación segura y eficiente. Además de corroborar la viabilidad de implementar soluciones robustas mediante ROS2 y el procesamiento de tarjetas embebidas como la Jetson nano.

Bibliografía

¿Qué es un controlador PID? (2025, 6 abril). Soluciones de Adquisición de Datos (DAQ). https://dewesoft.com/es/blog/que-es-un-controlador-pid

Lozano, R. (2021, 10 agosto). ¿Como usar camara v2.1 para raspberry? *Talos Electronics*. https://www.taloselectronics.com/blogs/tutoriales/camara-para-raspberry-v2

Descomponer los colores de una imagen. (2024, 1 septiembre). FIRST Robotics Competition Documentation. https://docs.wpilib.org/es/latest/docs/software/vision-processing/wpilibpi/image-thresholding.html

Kumari, S., & Kumari, S. (2023, 14 enero). Beginner's guide to OpenCV in Python. *Codedamn News*. https://codedamn.com/news/python/beginners-guide-to-opencv-in-python

Sosa-Costa, A. (2019, 30 agosto). *La transformada de línea de Hough - ▷ Cursos de Programación de 0 a Experto © Garantizados*. ▷ Cursos de Programación de 0 A Experto © Garantizados. https://unipython.com/la-transformada-linea-hough/

(S/f). Scielo.cl. Recuperado el 15 de mayo de 2025, de https://www.scielo.cl/scielo.php?script=sci_arttext&pid=S0718-33052017000100028

Detectar círculo en imagen binaria de contornos en Python. (s/f). Stack Overflow en español. Recuperado el 15 de mayo de 2025, de https://es.stackoverflow.com/questions/505583/detectar-c%C3%ADrculo-en-imagen-binaria-de-contornos-en-python

Anexos

Repositorio de GitHub

Video de funcionamiento

Video de presentación