



Tecnológico de Monterrey

Challenge 2

Mariam Landa Bautista A01736672

Elias Guerra Pensado A01737354

Emiliano Olguin Ortega A01737561

Yestli Darinka Santos Sánchez A01736992

10 de Abril del 2025

Implementación de Robótica Inteligente

Manchester Robotics

Resumen

Este reporte documenta el desarrollo e implementación de un controlador en lazo abierto para un robot móvil, como parte del Mini Challenge 1 del curso MCR2. El reto consistió en diseñar un sistema capaz de dirigir al robot físico en la ejecución de una trayectoria cuadrada, de 2 metros por lado, utilizando un nodo programado en ROS2. El movimiento del robot fue estructurado mediante una máquina de estados finita (FSM), la cual permitió organizar el comportamiento en fases claras: avanzar, detenerse, girar 90 grados y repetir este ciclo hasta completar los cuatro lados del cuadrado.

Un aspecto fundamental del desafío fue asegurar la robustez del controlador, es decir, su capacidad para mantener un funcionamiento estable y preciso incluso ante posibles perturbaciones, errores de modelado, ruido sensorial o no linealidades propias del sistema físico. Para lograrlo, se incorporaron varias estrategias: tiempos de ejecución calibrados con márgenes de tolerancia, un sistema de autoajuste (autotuning) de parámetros según las condiciones iniciales, y una lógica que reduce la dependencia de sensores o retroalimentación durante la ejecución, lo cual es característico del control en lazo abierto.

El controlador ofrece al usuario dos modos de operación: definir una velocidad constante o establecer un tiempo total para completar la trayectoria cuadrada. A partir de este dato, el sistema calcula internamente los tiempos de avance recto y de giro, o bien las velocidades necesarias. Esta flexibilidad permite ajustar el comportamiento del robot a diferentes contextos o limitaciones prácticas, como espacio reducido o variaciones en la respuesta del motor.

Para el envío de comandos de movimiento, se utilizó el tópico estándar `/cmd_vel`, por medio del cual se especifican velocidades lineales y angulares mediante mensajes Twist. Aunque el sistema no se basa en retroalimentación directa, se hace referencia a la posición y orientación del robot a través del tópico `/pose`, como parte del análisis del comportamiento general del robot y el cálculo teórico de distancia y ángulo recorrido.

Objetivos

- Implementar un nodo en ROS2 capaz de controlar un robot móvil en una trayectoria cuadrada de 2 metros por lado.
- Diseñar un controlador en lazo abierto robusto que funcione en un robot real.
- Incluir un mecanismo de autoajuste (autotuning) que adapte el comportamiento del robot según parámetros definidos por el usuario.
- Utilizar una máquina de estados para estructurar el comportamiento del robot.

Introducción

A lo largo de este reporte se presentará un sistema de control para un robot móvil que ejecuta una trayectoria cuadrada y una trayectoria personalizada en un entorno real. Este tipo de tarea representa una base importante dentro del campo de la robótica móvil, ya que permite poner en práctica conceptos fundamentales como el control en lazo abierto, el uso de mensajes de control y la estimación del movimiento del robot.

En este contexto, los mensajes Pose representan la posición y orientación del robot dentro del espacio. Son útiles para conocer su estado actual y analizar su desplazamiento. Por otro lado, los mensajes Twist contienen información sobre las velocidades lineales y angulares del robot, y son los encargados de indicar cómo debe moverse en cada instante.

El control en lazo abierto se refiere a un tipo de control en el que las acciones del robot no dependen de la retroalimentación directa de su posición o entorno, sino que se basan en un modelo previo del comportamiento esperado. Esto requiere estimar con precisión cuánto debe avanzar o girar el robot para completar su trayectoria, sin depender de sensores externos durante la ejecución.

Para lograrlo, es necesario calcular la distancia recorrida por el robot en línea recta, utilizando su velocidad y el tiempo de movimiento, así como el ángulo girado, que se determina a partir de la velocidad angular y el tiempo destinado a cada giro. Estos cálculos permiten guiar al robot a través de los cuatro segmentos rectos y los cuatro giros de 90° que forman el trayecto cuadrado.

Este proyecto busca integrar todos estos elementos de manera práctica, sentando las bases para un entendimiento más profundo del comportamiento de los robots móviles en escenarios reales.

Solución del problema

Trayectoria cuadrada

El controlador fue implementado en Python usando la biblioteca de ROS 2 `rclpy`, dentro de un nodo llamado `open_loop_ctrl`. Su función es dirigir el movimiento de un robot móvil real en una trayectoria cuadrada de 2 metros por lado, sin usar retroalimentación sensorial, es decir, bajo un esquema de **control en lazo abierto**.

Estructura del nodo

El nodo está encapsulado en una clase llamada `SquareControllerUwuntu`, que hereda de `Node`. En su constructor (`__init__`), se inicializan todas las variables necesarias para el

control del movimiento, se crea el publicador de comandos de velocidad y se configura un temporizador que ejecuta un ciclo de control periódico cada 0.1 segundos.

Los elementos principales del nodo son:

- **Velocidades predefinidas:**
 - `self.linear_speed = 0.1`: velocidad lineal para avanzar recto.
 - `self.angular_speed = 0.5`: velocidad angular para realizar los giros.
- **Tiempos calculados:**
 - `forward_time = 2.0 / linear_speed`: calcula cuánto tiempo necesita el robot para recorrer 2 metros a la velocidad definida.
 - `rotate_time = ($\pi/2$) / angular_speed`: tiempo necesario para girar 90 grados.
- **Máquina de estados finita (FSM):**
 - El robot cambia de comportamiento según el valor de `self.state`, que puede ser:
 - 0: avanzar recto.
 - 1: girar 90°.
 - 2: detenerse (fin del ciclo).
 - Se inicializa también un contador `turns_made` para llevar registro de los giros realizados.

Control basado en tiempo

El nodo no usa información de posición o sensores para corregir su trayectoria. En su lugar, usa el tiempo como variable de control: se calcula cuánto tiempo debe durar cada acción (avanzar o girar) y se cambia de estado cuando ese tiempo ha transcurrido.

Esto se logra midiendo el tiempo transcurrido desde el inicio de cada estado, utilizando el reloj de ROS (`self.get_clock().now()`), y comparándolo con el tiempo requerido para completar la acción.

Función `control_loop()`

Este método es ejecutado periódicamente por el temporizador. En cada iteración:

1. **Se calcula el tiempo transcurrido** desde el inicio del estado actual.
2. **Se crean y publican los comandos de velocidad** (mensaje `Twist`) según el estado:
 - En el **estado 0**, se envía una velocidad lineal constante y se avanza recto hasta que se cumple `forward_time`.
 - En el **estado 1**, se detiene el avance y se inicia la rotación con una velocidad angular fija hasta que se cumple `rotate_time`.
 - Una vez se han hecho cuatro giros (es decir, se ha recorrido todo el cuadrado), el controlador pasa al **estado 2**, en el que se detiene completamente y se cancela el temporizador.
3. **Se imprimen mensajes de registro** que ayudan a depurar el comportamiento del nodo, mostrando el estado actual, tiempo transcurrido y el número de giros realizados.

Sincronización con el reloj de ROS

Al inicio, el nodo espera a que ROS tenga un reloj activo mediante la función `wait_for_ros_time()`. Esto es importante en algunos sistemas donde el reloj de ROS depende de una fuente externa o no está inmediatamente disponible.

ZigZag

Desarrollo del código del controlador de trayectoria en zigzag

Además del movimiento cuadrado, se diseñó un segundo controlador para ejecutar una trayectoria de tipo **zigzag**, con la finalidad de explorar trayectorias más complejas en un esquema

de control en lazo abierto. Este controlador también fue implementado en Python usando ROS 2, y está contenido en un nodo llamado `zigzag_ctrl`.

Objetivo del nodo

El objetivo principal de este nodo es realizar una secuencia específica de movimientos compuesta por varios avances y giros, alternando direcciones para formar un patrón en zigzag. Al igual que en el caso anterior, este nodo **no depende de retroalimentación sensorial**, sino que ejecuta cada movimiento durante un tiempo calculado previamente en función de velocidades y distancias/ángulos deseados.

Estructura del nodo

El controlador está implementado dentro de la clase `ZigzagController`, la cual hereda de `Node`. A continuación, se describen los principales elementos que componen el nodo:

- **Lista de movimientos (`self.sequence`):**

Se define una secuencia de pasos a seguir, representada como una lista de tuplas. Cada tupla contiene:

 - Un tipo de acción: `'rotate'` (giro) o `'forward'` (avance).
 - Un valor asociado: distancia en metros (para avanzar) o ángulo en radianes (para girar).
- **Velocidades definidas:**
 - `self.linear_speed = 0.1`: velocidad lineal constante para avanzar.
 - `self.angular_speed = 0.5`: velocidad angular constante para girar.
- **Temporizador (`self.timer`):**

Se crea un temporizador que ejecuta el ciclo de control (`control_loop`) cada 0.1 segundos.
- **Variables de control:**
 - `self.current_step`: índice actual en la lista de movimientos.

- `self.state_start_time`: tiempo en el que se inició el paso actual.

Funcionamiento del método `control_loop()`

Este método es llamado periódicamente por el temporizador y se encarga de ejecutar la lógica de movimiento paso a paso. El funcionamiento general es el siguiente:

1. Verifica si se completó la secuencia:

Si se han ejecutado todos los pasos de la lista, se detiene el robot, se publica un mensaje `Twist` en cero (para detener el movimiento) y se cancela el temporizador.

2. Ejecuta el paso actual:

- Se extrae el tipo de acción y el valor desde la secuencia.
- Se calcula el tiempo que debe durar la acción:

- Para avanzar: $\text{duración} = \text{distancia} / \text{velocidad_lineal}$

- Para girar: $\text{duración} = \text{ángulo} / \text{velocidad_angular}$

- Se publica el comando de movimiento correspondiente:

- Si es 'forward', se establece una velocidad lineal positiva.

- Si es 'rotate', se establece una velocidad angular con el signo correspondiente (izquierda o derecha).

3. Transición de pasos:

Cuando se cumple el tiempo requerido (`elapsed_time >= duration`), se pasa al siguiente paso de la secuencia y se reinicia el temporizador.

4. Mensajes de depuración:

El nodo imprime mensajes detallados en consola sobre el paso actual, la distancia o ángulo ejecutado y cuándo se completa cada etapa.

Sincronización con el reloj de ROS

Al igual que en el nodo cuadrado, se incluye una función `wait_for_ros_time()` para esperar a que el reloj de ROS esté activo antes de iniciar el movimiento. Esto asegura una sincronización temporal adecuada en entornos ROS 2, especialmente cuando el reloj no está disponible de inmediato.

Durante el desarrollo del controlador, utilizamos el sistema de compilación `colcon`, el cual es estándar en ROS 2 para compilar paquetes y manejar sus dependencias. En el proceso de programación y prueba del nodo encargado de controlar al robot, notamos que en algunas ocasiones los cambios realizados en el código fuente no se veían reflejados al ejecutar el paquete, a pesar de compilarlo nuevamente.

Para resolver este problema, fue necesario limpiar manualmente los archivos generados por compilaciones anteriores, ya que estaban interfiriendo con la correcta actualización del código. Para ello, se utilizó el siguiente comando:

```
colcon build --packages-select nombre_del_paquete --cmake-target clean
```

Este comando ejecuta el objetivo `clean` de CMake, el cual elimina los archivos temporales de construcción, tales como binarios, archivos intermedios o configuraciones en caché que podrían impedir que los nuevos cambios surtan efecto. Después de ejecutar este comando, volvimos a compilar el paquete con:

```
colcon build --packages-select nombre_del_paquete
```

Gracias a esta limpieza, se solucionaron los problemas relacionados con la falta de actualización del código, y pudimos continuar con el desarrollo y prueba del controlador sin inconvenientes.

Identificación de la región lineal de los motores

Durante las pruebas experimentales realizadas con el robot real, se identificó que los motores presentan una zona de funcionamiento útil limitada por una velocidad mínima y una máxima efectiva.

Se observó que la velocidad mínima a la que los motores comienzan a moverse de manera consistente y pueden ejecutar correctamente los trayectos planificados es de aproximadamente 0.05 m/s. Por debajo de este valor, los motores no generan suficiente torque para superar la inercia del sistema, lo que produce movimientos intermitentes o directamente nulos.

Por otro lado, la velocidad máxima útil se encontró en torno a 0.2 m/s. Aunque los motores pueden girar más rápido, se detectó que al superar este umbral, el sistema presenta un fallo en la

jetson, lo que genera la desconexión de la señal de control y detiene inesperadamente el movimiento del robot.

Este fenómeno indica que el sistema podría estar experimentando un sobreconsumo de corriente o una caída de voltaje que interfiere con la alimentación de la electrónica, por estas razones, todos los controladores implementados en este reto fueron ajustados para trabajar dentro de la región lineal operativa del robot, es decir, entre 0.05 m/s y 0.2 m/s. Esto asegura que los movimientos sean estables, continuos y reproducibles, sin comprometer la integridad del sistema ni la precisión de la trayectoria.

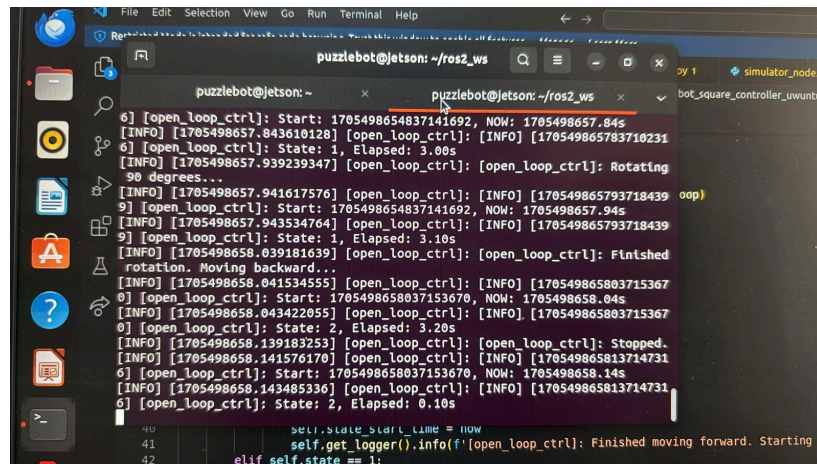
Resultados

Cuadrado

<https://youtu.be/q0EEs5-SWF4?si=Ka2bUXMQa2IMsWEP>

En el video se muestra como el robot avanza dos metros y se detiene para hacer un giro de 90° y seguir avanzando repitiendo la misma secuencia hasta terminar el cuadrado en la misma posición en la que empezó.

Impresión de consola



```
puzzlebot@jetson: ~ -/ros2_ws
6] [open_loop_ctrl]: Start: 1705498657.843610128, NOW: 1705498657.84s
[INFO] [1705498657.843610128] [open_loop_ctrl]: [INFO] [1705498657.843610128]
6] [open_loop_ctrl]: State: 1, Elapsed: 3.00s
[INFO] [1705498657.939239347] [open_loop_ctrl]: [open_loop_ctrl]: Rotating
90 degrees...
[INFO] [1705498657.941617576] [open_loop_ctrl]: [INFO] [1705498657.941617576]
9] [open_loop_ctrl]: Start: 1705498657.941617576, NOW: 1705498657.94s
[INFO] [1705498657.943534764] [open_loop_ctrl]: [INFO] [1705498657.943534764]
9] [open_loop_ctrl]: State: 1, Elapsed: 3.10s
[INFO] [1705498658.039181639] [open_loop_ctrl]: [open_loop_ctrl]: Finished
rotation. Moving backward...
[INFO] [1705498658.041534555] [open_loop_ctrl]: [INFO] [1705498658.041534555]
0] [open_loop_ctrl]: Start: 1705498658.041534555, NOW: 1705498658.04s
[INFO] [1705498658.043422055] [open_loop_ctrl]: [INFO] [1705498658.043422055]
0] [open_loop_ctrl]: State: 2, Elapsed: 3.20s
[INFO] [1705498658.139183253] [open_loop_ctrl]: [open_loop_ctrl]: Stopped.
[INFO] [1705498658.141576170] [open_loop_ctrl]: [INFO] [1705498658.141576170]
6] [open_loop_ctrl]: Start: 1705498658.141576170, NOW: 1705498658.14s
[INFO] [1705498658.143485336] [open_loop_ctrl]: [INFO] [1705498658.143485336]
6] [open_loop_ctrl]: State: 2, Elapsed: 0.10s
[INFO] [1705498658.143485336] [open_loop_ctrl]: [INFO] [1705498658.143485336]

40 self.state_start_time = now
41 self.get_logger().info(f'[open_loop_ctrl]: Finished moving forward. Starting
42 elif self.state == 1:
```

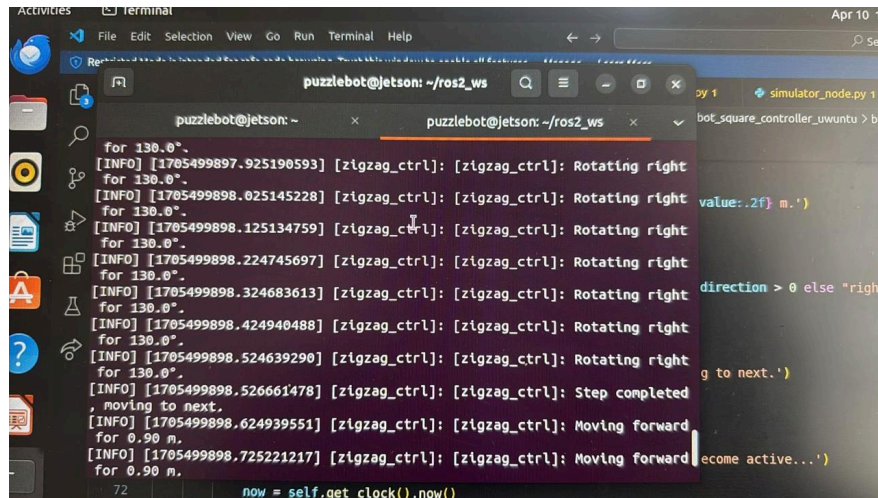
En la terminal se puede observar como muestra los movimientos del robot y en cuanto tiempo los realiza, muestra cuando avanza, se detiene y realiza los giros hasta terminar el cuadrado.

Trayectoria personalizada

https://youtube.com/shorts/rC4nB0yVeCs?si=qYfutM1_MQ0mdACP

En el video se muestra como el robot realiza un giro, para avanzar y volverse a detener para después volver a girar todo eso lo realiza hasta completar la trayectoria personalizada.

Impresión de consola



```
for 130.0°.  
[INFO] [1705499897.925190593] [zigzag_ctrl]: [zigzag_ctrl]: Rotating right  
for 130.0°.  
[INFO] [1705499898.025145228] [zigzag_ctrl]: [zigzag_ctrl]: Rotating right  
for 130.0°.  
[INFO] [1705499898.125134759] [zigzag_ctrl]: [zigzag_ctrl]: Rotating right  
for 130.0°.  
[INFO] [1705499898.224745697] [zigzag_ctrl]: [zigzag_ctrl]: Rotating right  
for 130.0°.  
[INFO] [1705499898.324683613] [zigzag_ctrl]: [zigzag_ctrl]: Rotating right  
for 130.0°.  
[INFO] [1705499898.424940488] [zigzag_ctrl]: [zigzag_ctrl]: Rotating right  
for 130.0°.  
[INFO] [1705499898.524639290] [zigzag_ctrl]: [zigzag_ctrl]: Rotating right  
for 130.0°.  
[INFO] [1705499898.526661478] [zigzag_ctrl]: [zigzag_ctrl]: Step completed  
moving to next.  
[INFO] [1705499898.624939551] [zigzag_ctrl]: [zigzag_ctrl]: Moving forward  
for 0.90 m.  
[INFO] [1705499898.725221217] [zigzag_ctrl]: [zigzag_ctrl]: Moving forward  
for 0.90 m.  
now = self.get_clock().now()
```

En la terminal se puede observar como muestra los movimientos del robot y en cuanto tiempo los realiza, muestra cuando avanza, se detiene y realiza los giros hasta terminar el zigzag

Conclusión

A lo largo de este proyecto se cumplieron de forma satisfactoria los objetivos planteados. Se logró implementar un nodo en ROS2 capaz de controlar un robot móvil en una trayectoria cuadrada de 2 metros por lado, haciendo uso de un controlador en lazo abierto robusto, estructurado mediante una máquina de estados. Además, se desarrolló un segundo nodo para ejecutar una trayectoria personalizada en zigzag, lo que permitió validar la flexibilidad y adaptabilidad de la lógica implementada.

Sin embargo, se identificaron ciertas limitaciones y dificultades con respecto al suelo y el estado de las llantas y el mismo robot ya que al no tener las mejores llantas llega a desviarse lo que impide la realización de los giros de manera precisa, asimismo, se presentaron complicaciones técnicas porque para entrar al sftp, tuvimos problemas ya que no se actualizaban los códigos cargados a la jetson, lo cual tuvimos que recurrir al comando `colcon build --packages-select tu_paquete --cmake-target clean`, la cual nos sirve para borrar el historial de los colcon hechos, para compilarlo nuevamente. Una vez ejecutado este comando, hacemos `colcon build` y el source como normalmente lo hacemos.

Una posible mejora futura sería implementar un código que en el que se pudiera correr las dos trayectorias condicionado de que si oprimía el número uno realizará la trayectoria del cuadrado y si oprimía el número dos realizaba el zigzag. En conclusión, el proyecto demostró una sólida comprensión de los principios de control en robótica móvil, además de fomentar habilidades prácticas en programación, resolución de problemas y adaptación al entorno físico real.