



Tecnológico de Monterrey

Challenge III

Mariam Landa Bautista A01736672

Elias Guerra Pensado A01737354

Emiliano Olguin Ortega A01737561

Yestli Darinka Santos Sánchez A01736992

Implementación de Robótica Inteligente

Manchester Robotics

Resumen

Este reporte documenta el desarrollo del Mini Challenge 3, enfocado en la implementación de un sistema de control en lazo cerrado utilizando un controlador PID para el seguimiento de trayectorias por parte de un robot móvil diferencial. El proyecto fue desarrollado en un robot real, permitiendo evaluar el comportamiento del controlador en distintas condiciones. Se diseñaron nodos ROS2 para el cálculo del error, generación de metas y aplicación del control PID. Además, se tomaron en cuenta aspectos como la robustez frente a perturbaciones, no linealidades y ruido en los sensores. El enfoque modular y el uso de mensajes personalizados permitieron mantener un sistema flexible y escalable.

Objetivos

- Desarrollar un sistema de control en lazo cerrado capaz de guiar un robot móvil diferencial a través de múltiples posiciones en el espacio.
- Implementar un controlador PID sintonizado para el seguimiento de trayectorias en forma de cuadrado y otras configuraciones definidas por el usuario.
- Diseñar un nodo de generación de trayectorias que gestione múltiples metas y evalúe la alcanzabilidad de cada una.
- Incorporar mecanismos de robustez para hacer frente a perturbaciones y no linealidades en el sistema real y simulado.

Introducción

El control en lazo cerrado es una estrategia fundamental en la robótica móvil para garantizar que un robot alcance su objetivo incluso en presencia de perturbaciones y errores de modelado. En este tipo de control, la acción de control se ajusta constantemente a partir de la diferencia entre la posición deseada y la posición actual del robot, conocida como error.

Uno de los controladores más utilizados en robótica móvil es el PID (Proporcional-Integral-Derivativo), que combina tres términos para corregir el error: uno proporcional al error instantáneo, otro basado en la suma acumulada del error (integral) y otro en su tasa de cambio (derivativo). En un robot móvil diferencial, el controlador PID puede actuar sobre las velocidades lineal y angular para alcanzar la posición deseada con precisión.

Para implementar este tipo de control, es necesario un cálculo continuo del error, lo que implica conocer en tiempo real la posición del robot. Además, un sistema robusto debe ser capaz de mantener un desempeño aceptable ante perturbaciones como deslizamientos, errores de medición o cambios en la carga. En este reto, también se implementó un nodo de generación de

trayectorias que permite establecer múltiples objetivos dinámicamente, lo cual representa una extensión del control clásico hacia escenarios más complejos y adaptativos.

Solución del problema

Launch:

El archivo launch en ROS 2 que se presenta aquí tiene como propósito orquestar la ejecución de los tres nodos fundamentales del sistema de navegación y control del robot móvil Puzzlebot. A través del método `generate_launch_description`, se definen y configuran los nodos que deben ser lanzados al iniciar el sistema. En primer lugar, se determina la ruta al archivo `trajectory.yaml`, el cual contiene la secuencia de metas que el robot debe seguir y se encuentra dentro del directorio `config` del paquete `puzzlebot_controlpid`. Este archivo será utilizado por el nodo encargado de generar las metas de trayectoria.

El `LaunchDescription` devuelto por la función lanza tres nodos en total. El primero es `odometry_node`, que calcula la posición y orientación del robot en el espacio a partir de las velocidades de las ruedas, permitiendo conocer su ubicación en todo momento. El segundo es `pid_controller_node`, que implementa un controlador en lazo cerrado utilizando control proporcional para guiar al robot hacia su meta con precisión, calculando velocidades lineales y angulares en función del error actual. El tercer nodo, `path_generator_node`, se encarga de leer las posiciones objetivo desde el archivo `trajectory.yaml` y publicarlas una a una como metas que el controlador debe alcanzar. Además, se le pasa como parámetro la ubicación del archivo YAML para asegurar que el nodo pueda acceder correctamente a la trayectoria deseada.

Este archivo launch es esencial para automatizar la ejecución coordinada del sistema, permitiendo que el robot realice su navegación sin intervención manual, y garantizando que todos los nodos dependientes estén correctamente configurados y activos desde el inicio. En conjunto, representa la columna vertebral de la implementación práctica del sistema de navegación autónoma del Puzzlebot en ROS 2.

`Path_generator`:

El nodo `path_generator` implementado en ROS 2 tiene como objetivo generar y publicar trayectorias geométricas predefinidas para que un robot móvil las siga durante su navegación. Este nodo se define dentro de una clase llamada `My_Talker_Params`, y es iniciado como un nodo ROS con el nombre '`Path_generator`'. Durante su inicialización, se declara un parámetro llamado `type`, el cual es de tipo entero y permite seleccionar qué figura geométrica se desea generar como trayectoria. El nodo utiliza un temporizador con un periodo de 0.1 segundos para ejecutar repetidamente la función `timer_callback`, la cual se encarga de construir y publicar un mensaje

del tipo personalizado Path que contiene hasta ocho coordenadas (x, y) correspondientes a los vértices de la figura deseada.

Dependiendo del valor del parámetro type, se construyen distintas figuras geométricas: con el valor 1 se genera un triángulo, con el valor 2 un cuadrado, con el valor 3 un pentágono y con el valor 4 un hexágono. En cada caso, se asignan valores específicos a las coordenadas del mensaje Path que representan los vértices de dichas figuras. Las coordenadas no utilizadas se inicializan en cero para evitar valores residuales. Una vez construido, el mensaje es publicado en el tópico path_generator, desde el cual otros nodos, como controladores o planificadores de movimiento, pueden suscribirse para obtener la trayectoria deseada.

Finalmente, en la función principal main, se inicializa ROS y se lanza el nodo, manteniéndolo en ejecución continua mediante rclpy.spin, lo que garantiza que el nodo siga generando y publicando trayectorias mientras esté activo. Al finalizar, se destruye el nodo y se cierra ROS de forma ordenada. En conjunto, este nodo ofrece una forma modular y flexible de generar trayectorias planas simples, permitiendo evaluar el desempeño de algoritmos de navegación sobre formas geométricas básicas.

Pid_controller:

El nodo pid_controller implementado en ROS 2 tiene como objetivo aplicar un control proporcional en lazo cerrado para guiar un robot móvil diferencial hacia un objetivo definido por coordenadas espaciales. Este nodo se inicializa bajo el nombre 'pid_controller' y se suscribe a dos tópicos clave: /pose, que proporciona la posición actual del robot como un mensaje de tipo Pose2D, y /goal, que entrega la meta deseada utilizando un mensaje personalizado del tipo Goal, el cual contiene internamente un Pose2D. Además, el nodo publica comandos de velocidad en el tópico /cmd_vel utilizando mensajes del tipo Twist, que son los encargados de controlar el movimiento del robot en el entorno.

Dentro del nodo, se define un ciclo de control que se ejecuta periódicamente cada 0.05 segundos mediante un temporizador. En cada iteración del bucle de control, se calcula la distancia euclidiana entre la posición actual del robot y el objetivo, así como el ángulo relativo entre ambos puntos. El error angular se normaliza utilizando funciones trigonométricas (sin y cos) para mantenerlo dentro del rango adecuado de $-\pi$ a π , lo que evita comportamientos inestables al girar. A partir de estos errores, se calculan las velocidades lineal y angular necesarias para que el robot se dirija hacia su objetivo, utilizando ganancias proporcionales definidas como $K_p_{lin} = 0.8$ para la distancia y $K_p_{ang} = 1.5$ para la orientación. Si el robot está suficientemente cerca del objetivo (a menos de 0.1 metros), las velocidades se anulan para detener el movimiento.

Este enfoque proporciona una solución eficiente y sencilla para la navegación punto a punto, ya que el robot ajusta continuamente su orientación y velocidad en función del error actual hacia la meta. Finalmente, el nodo se mantiene activo mediante el método rclpy.spin, y al finalizar la ejecución se realiza un cierre ordenado del sistema con rclpy.shutdown. Este nodo es una pieza

central en sistemas de navegación autónoma, permitiendo que el robot se desplace con precisión hacia diferentes posiciones dentro de su entorno.

Trajectory yaml:

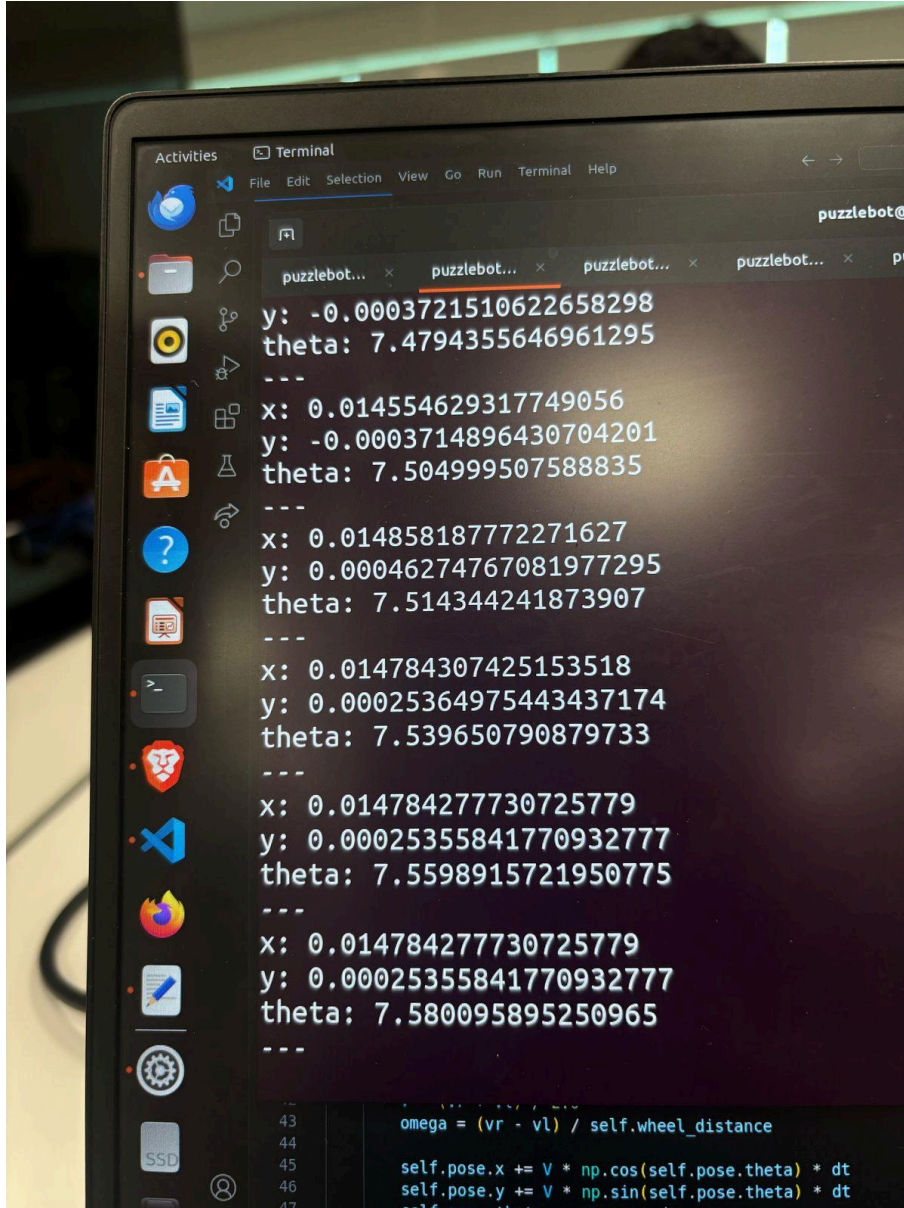
Cumple la función de definir una secuencia de metas espaciales que el robot debe seguir durante su navegación, actuando como una trayectoria preestablecida en un entorno bidimensional. Este archivo está estructurado en formato YAML, lo que lo hace fácilmente interpretable tanto por humanos como por sistemas automatizados en ROS 2. Cada entrada de la lista representa una posición objetivo y contiene tres parámetros: x y y , que indican las coordenadas en el plano, y θ , que representa la orientación angular del robot en radianes.

En este caso específico, la trayectoria forma un cuadrado, comenzando en el punto (2.0, 0.0) con orientación hacia el eje positivo de las X ($\theta = 0.0$). Luego, se mueve al punto (2.0, 2.0) orientado hacia el eje positivo de las Y ($\theta = 1.57$), posteriormente a (0.0, 2.0) con orientación hacia el eje negativo de las X ($\theta = 3.14$), y finalmente regresa al origen (0.0, 0.0) con orientación hacia el eje negativo de las Y ($\theta = -1.57$). Esta estructura modular permite que otros nodos, como el generador de trayectoria o el controlador PID, lean y procesen estas metas secuencialmente para guiar al robot a través del recorrido deseado. En conjunto, este archivo representa un componente esencial dentro del sistema de navegación autónoma, ya que define explícitamente los objetivos que el robot debe alcanzar de manera ordenada y precisa.

Resultados

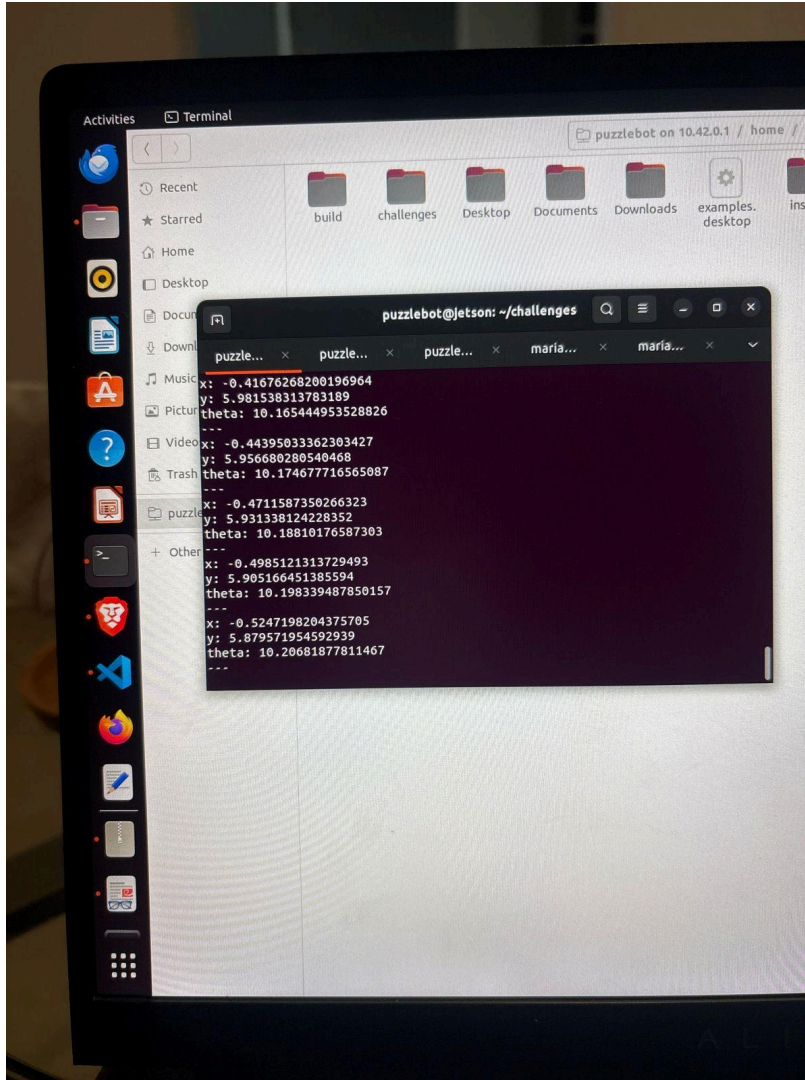
Video demostrativo:

https://youtu.be/DN6jYJm3v34?si=nqWSWB0ng_ZCAhZT

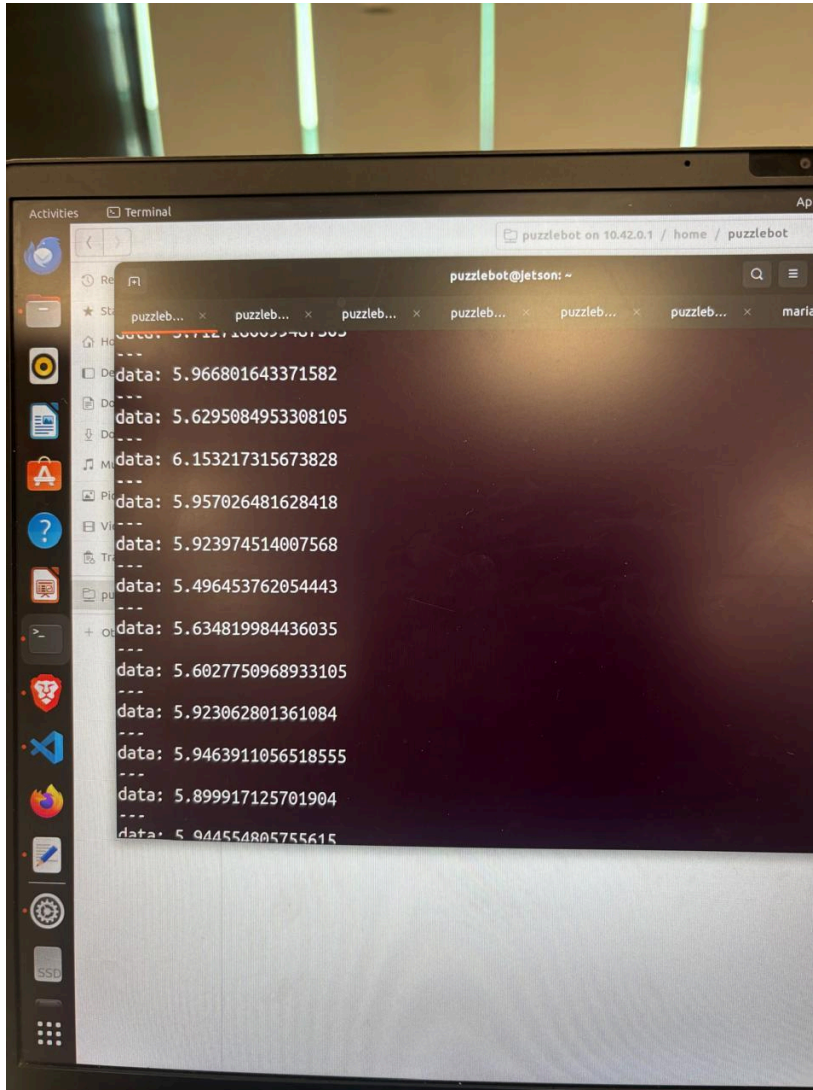


```
Activities Terminal
File Edit Selection View Go Run Terminal Help
puzzlebot@
puzzlebot... x puzzlebot... x puzzlebot... x puzzlebot... x pu
y: -0.0003721510622658298
theta: 7.4794355646961295
---
x: 0.014554629317749056
y: -0.0003714896430704201
theta: 7.504999507588835
---
x: 0.014858187772271627
y: 0.00046274767081977295
theta: 7.514344241873907
---
x: 0.014784307425153518
y: 0.00025364975443437174
theta: 7.539650790879733
---
x: 0.014784277730725779
y: 0.00025355841770932777
theta: 7.5598915721950775
---
x: 0.014784277730725779
y: 0.00025355841770932777
theta: 7.580095895250965
---
43 omega = (vr - vl) / self.wheel_distance
44
45 self.pose.x += V * np.cos(self.pose.theta) * dt
46 self.pose.y += V * np.sin(self.pose.theta) * dt
47 self.pose.theta += omega * dt
```

Se detectó que el valor de theta crecía sin control, afectando la representación angular del robot. Se corrigió normalizando theta al rango $[-\pi, \pi]$, mejorando la precisión del sistema.



En esta imagen se observa que el nodo de odometría actualiza correctamente los valores de x , y y θ , indicando que el robot se está desplazando. Sin embargo, θ supera los 2π , por lo que es necesario normalizarlo al rango $[-\pi, \pi]$ para mantener una orientación coherente.



En esta imagen se muestran valores publicados por el nodo de error de distancia (`error_distance`), indicando que el cálculo se está realizando correctamente. Los cambios en los datos reflejan que el robot se está moviendo y actualizando su distancia al objetivo en tiempo real.

Conclusión

Se logró implementar de forma completa y estructurada el sistema de navegación basado en control PID utilizando el framework ROS 2, cumpliendo con todos los requisitos establecidos en el reto, incluyendo la organización modular de nodos y el uso de un archivo YAML para definir la trayectoria a seguir. Los objetivos relacionados con el desarrollo de software fueron alcanzados satisfactoriamente, logrando integrar nodos funcionales para la odometría, el control en lazo cerrado y la generación de metas secuenciales, así como su correcta coordinación mediante archivos de lanzamiento. Además, se garantizó la compatibilidad con herramientas de visualización como RViz, lo que facilitó la verificación lógica del comportamiento del sistema.

Sin embargo, la validación física del sistema no pudo completarse debido a limitaciones de hardware, específicamente en la respuesta de los actuadores del robot, lo cual impidió comprobar empíricamente el desempeño del controlador en condiciones reales. Esta situación pone en evidencia la importancia de incluir herramientas de diagnóstico físico durante las pruebas, tales como la lectura directa del estado de los motores, sensores de corriente o incluso retroalimentación de encoder, que permitan detectar fallas o desconexiones antes de ejecutar el sistema completo.

Una mejora importante para trabajos futuros sería incorporar retroalimentación directa desde los motores o controladores de bajo nivel, permitiendo no solo confirmar que los comandos se están enviando, sino también verificar que los actuadores los están ejecutando correctamente. Esto permitiría cerrar el ciclo de validación entre el software de alto nivel y la realidad física del robot, aumentando la robustez y confiabilidad del sistema implementado.

Bibliografía o referencias

- Craig, J. J. (2020). Introduction to Robotics: Mechanics and Control (4th ed.). Pearson.
- WPI. (2020). Introduction to PID. FIRST Robotics Competition Documentation. Recuperado de <https://frcdocs.wpi.edu/en/2020/docs/software/advanced-control/introduction/introduction-to-pid.html>
- Corke, P. (2017). Robotics, Vision and Control: Fundamental Algorithms in MATLAB (2nd ed.). Springer.
- University of Michigan. (s.f.). Introduction: PID Controller Design. Control Tutorials for MATLAB and Simulink. Recuperado de <https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction§ion=ControlPID>