Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Puebla

TE3001B: Fundamentación de robótica (Gpo 101)

Profesor: Dr. Rigoberto Cerino Jiménez

Reto semanal 2. Manchester Robotics

Equipo 5

Tania Sofia Arrazola Bello	A01738124
Iñaki Ahedo Madrid	A01738231
Marcos Allen Martínez Cortés	A01737939

23 de febrero de 2026

Resumen

En el siguiente reporte se presenta la solución al reto semanal 2 propuesto por Manchester Robotics el cual funcionó como continuación al reforzamiento de los conocimientos básicos de ROS 2 utilizando más funciones disponibles además de las previamente usadas como el modelo publicación-suscripción. Esta actividad consiste en utilizar los nodos previamente hechos en la clase los cuales simulan un sistema de lazo abierto en el que un nodo `set_point` actúa como entrada del sistema la cual es una señal senoidal que recibe el nodo `dc_motor` el cual simula un motor con la ecuación:

$$y[k+1] = y[k] + ((-1/\tau)\, y[k] + (K/\tau)\, u[k])\, T\_s \quad (1)$$

para hacer un controlador P, PI o PID.

Para esta solución se utilizaron nuevas herramientas de ROS2 como namespaces, parámetros en el archivo de lanzamiento y `rqt_reconfigure`. De igual forma en este reporte se explicará la metodología de la solución a este reto.

Objetivos

El objetivo principal de este reto es utilizar los nodos previamente hechos en clase en los que simula el comportamiento de un motor DC para crear un controlador para este sistema de primer orden. Este controlador puede ser proporcional, proporcional e integral o proporcional, integral y derivativo (PID). Los objetivos específicos de este reto fueron:

- Analizar la planta y crear un nuevo nodo llamado `controller` que sin el uso de librerías, programar un controlador para este.
- Ya sea en el archivo de lanzamiento o en un archivo `.yaml` especificar que las ganancias de Kp, Kd y Ki serán parámetros los cuales podrán ser modificados durante la ejecución del sistema.
- Utilizar la herramienta `rqt_reconfigure` para facilitar la configuración de estos parámetros.
- Utilizar namespaces para generar diferentes señales de entrada al sistema.
- Visualizar los resultados con la herramienta de `rqt_plot`.

Introducción

El funcionamiento del programa era con un sistema previo de motor dado por Manchester Robotics, era a lazo abierto el cual hacía que el generador de señales publicara continuamente una señal senoidal que representa el voltaje o esfuerzo aplicado al motor $u(t)$. Por su parte, el nodo del motor recibía ese impulso y calculaba su velocidad de salida $y(t)$ que estaba únicamente definida por su ganancia K y su constante de tiempo τ. El problema con este sistema es que al carecer de un lazo de retroalimentación que verifique el error existente de la relación entre la entrada y la salida, el motor reaccionaba a la señal senoidal y presentaba un retraso debido a su naturaleza. (Vilanova & Alfaro, 2011)

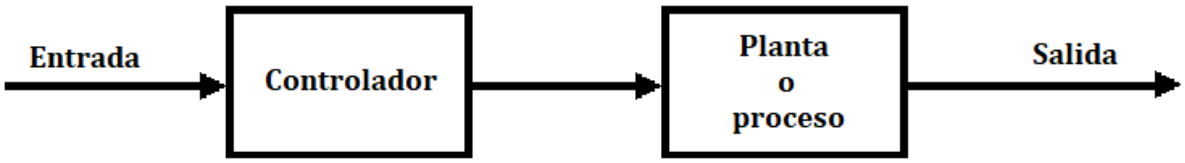


Figura 1: Sistema en lazo abierto.
https://www.researchgate.net/publication/334771046/figure/fig2/AS:786392066039808@1564501887099/Figura-5-Sistema-de-control-en-lazo-abierto.ppm

Para resolver esto y hacer que la señal de salida se pueda ajustar a que sea lo más parecido a la de entrada que en un contexto aplicado, se tiene que aplicar un controlador que tenga una retroalimentación y mida con una diferencia el resultado final con lo que se espera para después ajustar los parámetros y tenga un comportamiento lo más cercano a lo deseado. Para aquello existen varios controladores, en esta actividad se desarrolló un controlador PID. (Vilanova & Alfaro, 2011)

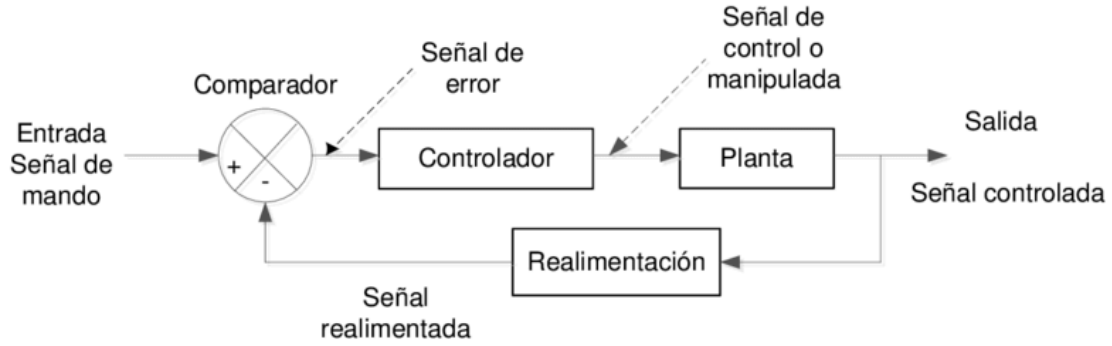


Figura 2: Sistema en lazo cerrado.
https://www.researchgate.net/figure/Diagrama-a-bloques-de-un-sistema-de-control-en-lazo-cerrado_fig1_267454753

Un controlador PID o controlador proporcional, integral y derivativo es un mecanismo de control que funciona mediante retroalimentación. La retroalimentación viene de medir la señal que se produjo a la salida $y(t)$ y compararla con la señal de entrada o el set point $r(t)$ y hacer una diferencia. A esta diferencia se le conoce como error $e(t)$ como se muestra en la ecuación 2:

$$e(t) = r(t) - y(t) \tag{2}$$

El control PID hace tres acciones con respecto a ese error:

Acción proporcional (K_p): Esta acción es la base del control. Su función es reaccionar a cuánto falta para llegar al valor deseado. Matemáticamente es muy simple debido a que multiplica al error actual por una constante K_p. En el caso de la simulación de un motor, esta constante funciona cuando la velocidad del motor es lejano a la velocidad deseada, la acción de esta constante es enviar un voltaje más alto. Entre menor sea la diferencia entre lo obtenido y lo esperado, la acción de K_p disminuye.

El problema con este controlador es que llega un punto en el que el error es tan pequeño que la señal resultante no tiene la suficiente fuerza para llegar al objetivo, siempre quedándose un poco abajo, esto se le conoce como error de estado estacionario.

Acción integral (K_i): La parte integral del controlador es la que soluciona el problema al solamente usar el control P. Esta acción se encarga de verificar los errores que se han acumulado anteriormente. Mira cuánto tiempo ha pasado en el estado estacionario y va sumando ese error acumulado. Al sumar este error, hace que no se quedé abajo del valor deseado. El problema es que si la ganancia de K_i es muy, el sistema puede llegar a tener oscilaciones hasta volverse inestable.

Acción Derivativa (K_i): La acción derivativa funciona como un amortiguador a los cambios repentinos que tiene el sistema o son causados por la acción proporcional. Este se basa en la razón de cambio del error, es decir, mide la pendiente de esta señal y si esta es muy inclinada, este hace un cambio más intenso y si es más estable, su acción es prácticamente 0.

Mientras que las demás acciones se enfocan en eliminar el error, esta acción se enfoca en la estabilidad. Actúa como una fuerza de oposición que frena la velocidad de respuesta del sistema cuando se está aproximando muy rápido al objetivo, evitando que el sistema se vuelva inestable y eliminando las oscilaciones que puede provocar ya sea la señal de entrada o las demás acciones. (Vilanova & Alfaro, 2011)

Si se utilizan las tres señales, se estará utilizando un controlador PID el cual suma estas tres acciones, repercutiendo en la señal de salida como se muestra en la ecuación 3:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \quad (3)$$

Para sintonizar las constantes de cada acción del controlador PID es importante realizarlo con la mejor precisión posible ya que en caso de un motor se necesita que este reaccione rápido más evitar el volverse inestable. Se utilizan varios métodos para la sintonización de estas variables, a continuación se presentan:

- **Método Heurístico (Prueba y error)**

Para este método primero se ajusta Ki y Kd en 0 y subir Kp hasta que el sistema reaccione con buena velocidad aunque oscila un poco. Después se aumenta Kd para evitar esas oscilaciones y amortiguar el sistema. Por último se aumenta Ki ligeramente para eliminar el error en estado estacionario, es decir, que la señal llegue exactamente al valor deseado.

- **Ziegler-Nichols**

Este método comienza en verificar el momento exacto donde la planta deja de ser estable y empieza a oscilar de forma constante. Para aquello, de igual forma Ki y Kd se inicializan en 0 y se aumenta gradualmente Kp hasta llegar a un punto en donde empezará a oscilar. Debe aumentar este valor hasta encontrar la ganancia crítica K_u, este sería el valor de Kp cuando se logra una oscilación constante. (Åström & Hägglund, 2004)

Después se mide el periodo crítico P_u el cual es el tiempo en segundos que tarda la oscilación en completar un ciclo completo. Una vez encontrados dichos valores, se diseña una tabla de constantes para calcular las tres ganancias del controlador. Estas ganancias son diferentes dependiendo el tipo de controlador que se quiera implementar. A continuación se presenta la tabla de parámetros:

Control	**Kp**	**Ki**	**Kd**
P	$0.5K_u$	-	-
PI	$0.45K_u$	$\frac{1}{12}P_u$	-
PID	$0.6K_u$	$0.5P_u$	$0.125P_u$

Tabla 1. Parámetros de Control

Namespaces en ROS2

Por otro lado, ROS2 implementa el uso de namespaces (espacios de nombres), una característica diseñada para organizar y agrupar nodos, tópicos y servicios dentro de un contexto específico. Su principal ventaja es que evitan las colisiones de nombres; es decir, hacen posible ejecutar múltiples instancias de un nodo con el mismo nombre de forma simultánea, siempre y cuando operen en namespaces distintos. En la práctica, esto permite particionar la red de ROS en subsistemas aislados que funcionan de manera independiente (Ramachandran, 2023).

Parámetros en ROS2

En ROS2, los parámetros son valores de configuración de nodos individuales que permiten modificar su comportamiento en el arranque o en tiempo de ejecución sin alterar el código fuente (Open Robotics, s.f.-b). Estructuralmente constan de una clave, un valor y un descriptor, direccionándose de forma jerárquica mediante namespaces y nombres. Es obligatorio que cada nodo declare previamente los parámetros que aceptará. Además, para gestionar modificaciones dinámicas, los nodos utilizan funciones callback (set parameter y on parameter event), las cuales reaccionan al intentar cambiar un valor o justo después de que un parámetro sea declarado, modificado o eliminado (Open Robotics, s.f.-b).

Servicios en ROS2

Los servicios son otro método de comunicación para los nodos en el grafo ROS. Se basan en un modelo de llamada y respuesta, a diferencia del modelo de publicación-suscripción de los temas. Mientras que los temas permiten a los nodos suscribirse a flujos de datos y recibir actualizaciones continuas, los servicios solo proporcionan datos cuando un cliente los solicita específicamente (*Understanding Services ROS 2 Documentation: Foxy Documentation*, s. f.). Puede haber muchos clientes de servicio utilizando el mismo servicio. Pero solo puede haber un servidor de servicio para un servicio.

Solución del problema

Para la solución de este reto, se diseñó una estructura utilizando el entorno de ROS para la creación de nodos que simulan el sistema de control.

1. **Modelado de la planta**

El primer paso consistió en programar la simulación del comportamiento de un motor DC. Se implementó una ecuación de diferencias de primer orden que representa la respuesta del motor ante un voltaje de entrada con la ecuación 1.

Para esto se configuraron como parámetros la ganancia K en 1.78 y la constante de tiempo τ en 0.5. El nodo `dc_motor.py` calcula la nueva velocidad cada 0.02 segundos basándose en el valor previo y la señal recibida en el tópico `motor_input_u`.

2. **Generador de señal**

Para poder evaluar el controlador, se desarrolló un nodo capaz de emitir distintos tipos de señales de referencia. Utilizando la función de ROS **Parameter Callback**, se configuró el nodo `set_point.py` para cambiar el tipo de señal `signal_type` en tiempo real. Esto se diferencia de los parámetros del motor que fueron declarados en el archivo de lanzamiento, esto permite que durante el tiempo de ejecución se pueda obtener una señal de tipo seno, escalón, cuadrada o rampa.

Este nodo también implementa un cliente de servicio `SetProcessBool` el cual asegura que la generación de la señal solo empieza cuando el motor ya está habilitado.

3. **Diseño del controlador PID**

Se creó un tercer nodo llamado `ctrl.py` el cual hace que el sistema pase de ser de lazo abierto a ser de lazo cerrado. Se basa en el cálculo del error $e(t)$ como vimos en la ecuación 2 en el que se resta el valor del setpoint con el valor de salida del motor.

De igual manera de como se definieron los parámetros del nodo que genera la señal, se utilizó **Parameter Callbacks** para definir las constantes del control las cuales son Kp, Kd y Ki. Esto permitirá que se modifiquen en tiempo de ejecución para hacer la sintonización de estas constantes. Este nodo se suscribe al tópico `motor_speed_y` que es la salida del sistema y al tópico `set_point` que es la entrada. Utilizando estos valores, se puede calcular el error que será utilizado por las constantes de cada una de las tres acciones que realiza el control:

- La acción proporcional P simplemente se calcula multiplicando de forma directa el error actual por la constante Kp. Esto representa la fuerza de reacción instantánea al error.
- Debido a que no se puede calcular la integral continua en la computadora, para la acción integral I se utilizó un método de aproximación en el que la variable `self.integral` funciona como un acomulador que en cada iteración suma el error actual y el paso temporal. Se puede explicar con la ecuación 4 en tiempo discreto:

$$I[k] = I[k-1] + (e[k] \cdot \Delta t) \tag{4}$$

- Para la acción derivativa D, se igual manera que la acción integral, se debe calcular de manera discreta. En este caso en lugar de derivar el error, el código almacena el error de la iteración pasada en `self.prev_error` y calcula la pendiente dividiendo la diferencia de errores entre el tiempo transcurrido como se muestra en la ecuación 5:

$$D[k] = \frac{e[k]-e[k-1]}{\Delta t} \quad (5)$$

Después de obtener estos cálculos, se tiene la señal de control (`control_signal`) que es la suma de estos tres cálculos multiplicados por sus respectivas ganancias que son Kp, Kd y Ki. Esto se publica como mensaje al tópico `motor_input_u` el cual es recibido por el motor para modificar sus propiedades y sea lo más parecido a la señal de entrada.

Sintonización de parámetros

Antes de ejecutar el archivo de lanzamiento con los tres nodos, se hizo la sintonización de los parámetros utilizando la herramienta de Simulink Control Design el cual, cuando se modela el comportamiento de un sistema a lazo cerrado con control PID, se puede hacer un **self-tuning** a las ganancias del control. En su interfaz gráfica se puede modificar qué tan agresivo es el controlador junto con el tiempo de respuesta deseado con una gráfica que indica cómo se comportaría moviendo tales parámetros y se puede ir ajustando con la referencia del sistema de entrada.

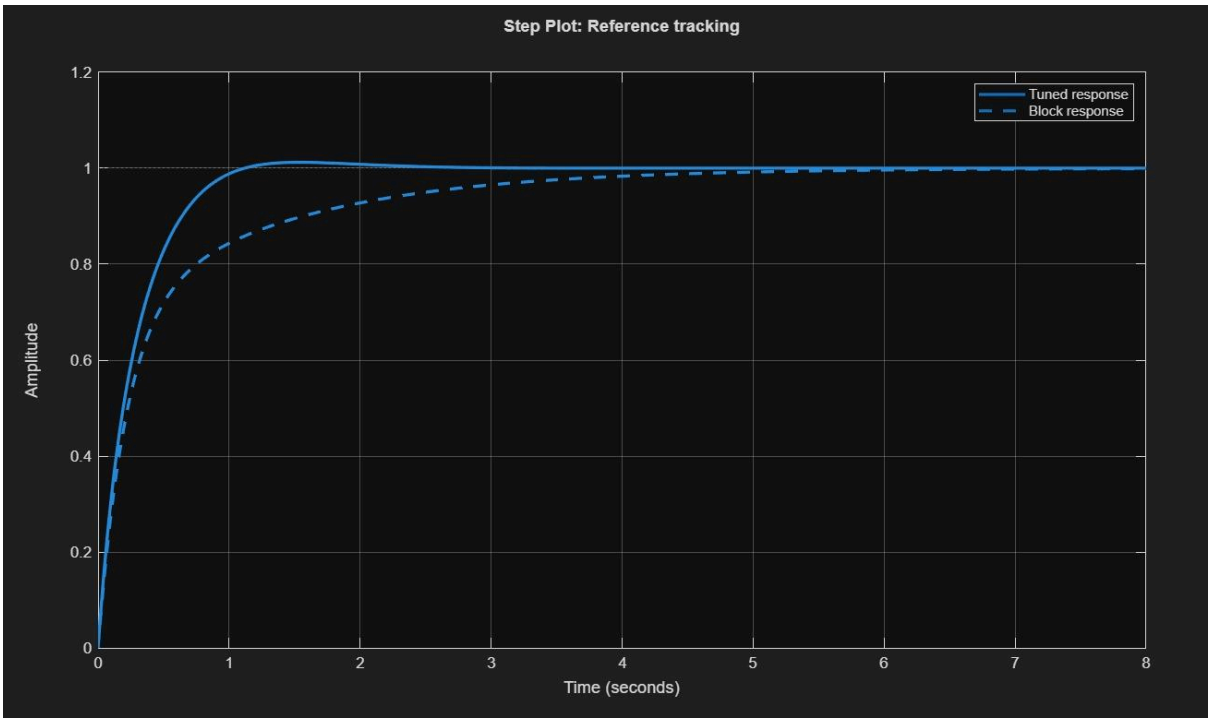


Figura 3. Interfaz gráfica de la herramienta de Simulink Control Design para la sincronización de parámetros

Una vez encontrado el punto de estabilidad del sistema, la herramienta ajustará los parámetros del controlador automáticamente.

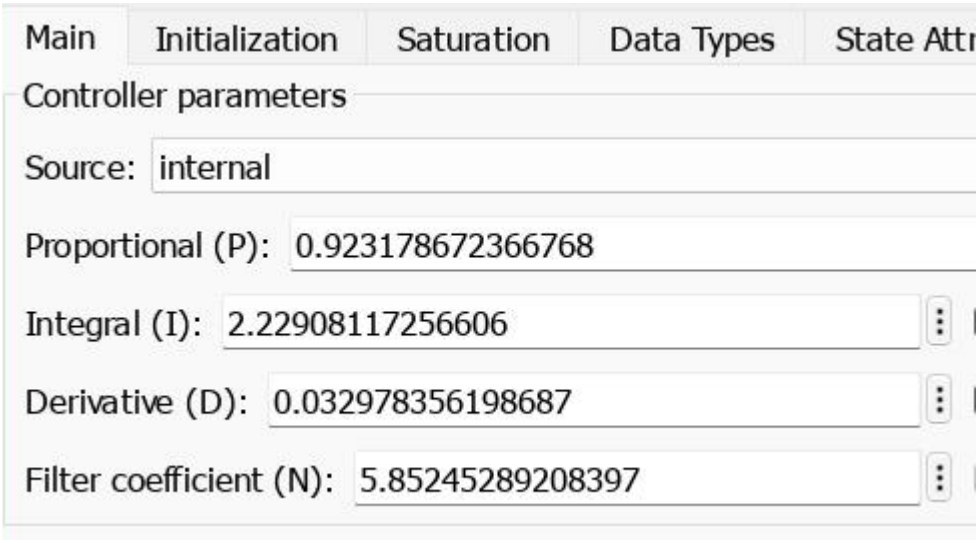


Figura 4. Sintonización automática de las ganancias del controlador

Los valores de las ganancias son utilizados para la simulación en ROS. Después del lanzamiento del programa, al utilizar la herramienta de `rqt_plot` al comparar la señal de entrada con la de salida, si no se obtienen los resultados esperados, las ganancias al ser declaradas como parámetros y utilizar la función **Parameter Callback** se pueden modificar en tiempo real utilizando el comando `ros2 param set <nombre del nodo> <nombre del parámetro> <nuevo valor>.`

4. Creación de archivos de lanzamiento

Para poder generar distintos tipos de ondas utilizando el mismo nodo de set point se deben utilizar namespaces ya que como se mencionó es una herramienta para hacer distintos grupos de un mismo nodo. Para aquello se creó un archivo de lanzamiento base (`motor_launch.py`) el cual instancia los tres nodos principales (motor, set point y controlador) para ser ejecutados a la vez. Aquí es donde se definen los parámetros del motor, sin embargo al ser declarados así, no se podrán modificar en tiempo de ejecución.

Se crea otro archivo de lanzamiento `challenge_launch.py` el cual incluye el anterior archivo. En este se usa la herramienta de **GroupAction** para definir tres grupos y después se utiliza la herramienta **PushRosNamespaces** para añadir los prefijos `/group1`, `/group2` y `/group3` a cada uno de los grupos. Esto hace que cada motor tenga su propio controlador y su propia referencia, estos operan de manera paralela para que la modificación de uno no afecte a otro.

Una vez creados los archivos de lanzamiento, en una terminal se compila desde la carpeta del **workspace** con `colcon build,` después se hace el `source install/setup.bash` y por último se corre el archivo de lanzamiento con `launch.`

Resultados

Al ejecutar el archivo de lanzamiento de `challenge_launch.py` se ejecutaran los tres nodos del paquete y se iniciarán las simulaciones de todos los grupos. Al ejecutar la herramienta de `rqt` se pueden observar cómo se iniciaron los nodos y si están comunicando con sus respectivos tópicos.

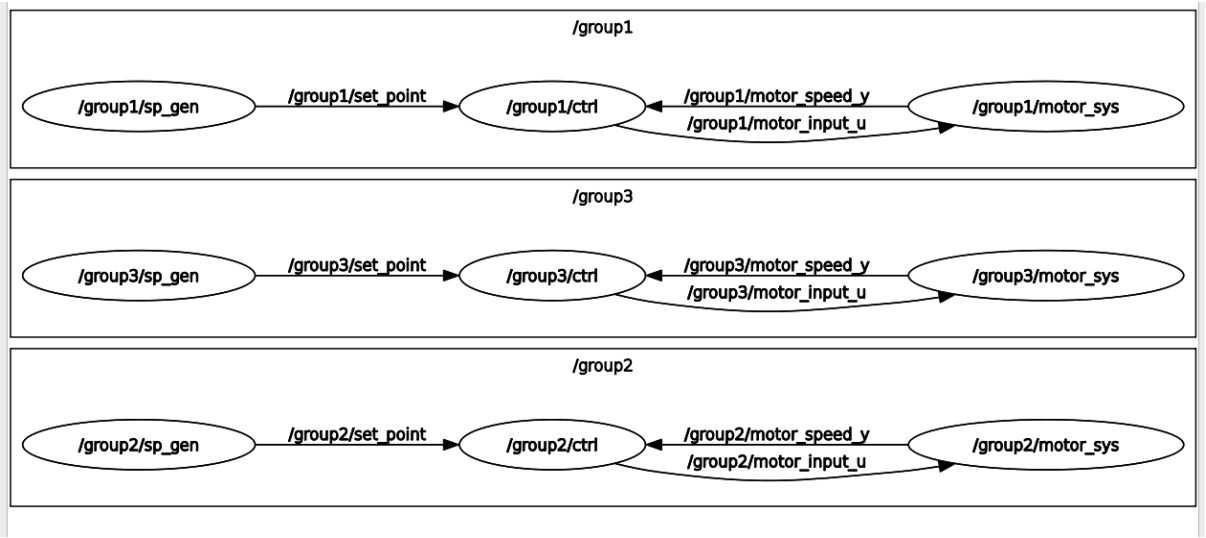


Figura 5. Nodos y topicos con rqt_graph

Se utilizó la herramienta de `rqt_plot` para visualizar las gráficas de la señal de entrada y la señal de salida con respecto al tiempo. Aqui es donde se puede ver la diferencia entre el comportamiento real con lo esperado y a partir de eso, se podrán ajustar las ganancias del controlador PID.

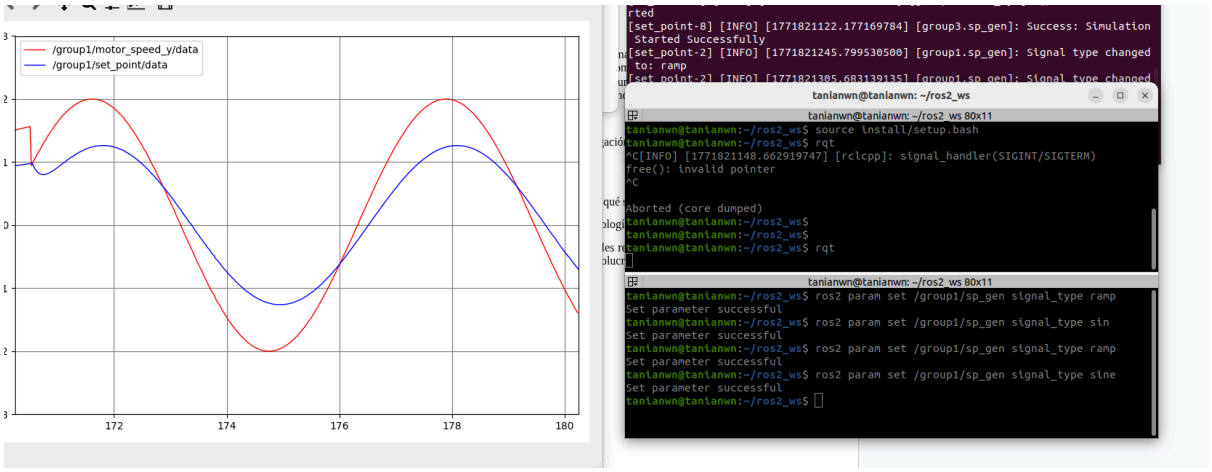


Figura 6. Ejemplo de ejecución señales de entrada y salida con kp=1 kd=0 y ki=0

Para modificar estos parámetros se puede hacer de dos maneras. La primera es utilizando la herramienta de ROS `rqt_reconfigure` la cual toma en cuenta las variables que se declararon como parámetro para poder ser modificadas durante el tiempo de ejecución en una interfaz gráfica. En esta se selecciona los parámetros del nodo de control y ahí tendrá la opción de modificar Kp, Kd y Ki con respecto a los valores que se sintonizaron con la herramienta de Simulink. De igual manera se puede modificar individualmente los parámetros de cada grupo creado.

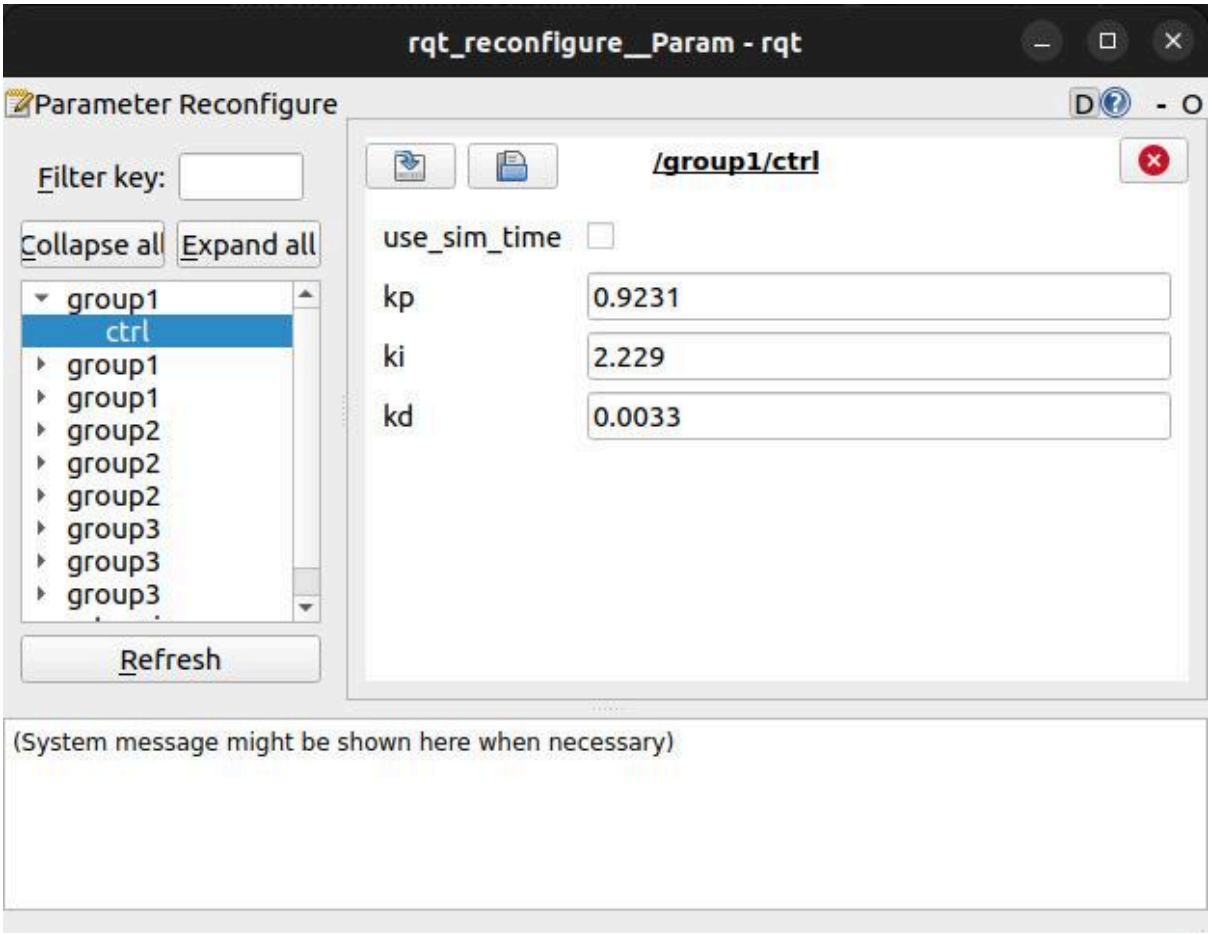


Figura 7. Modificación de parámetros utilizando rqt_reconfigure

La segunda manera de modificar los parámetros es desde la terminal utilizando la función `param set` especificando el grupo, el nodo y la variable que se quiera modificar. Utilizando estas dos funciones también se puede cambiar el tipo de señal de entrada que recibe el nodo de motor, se puede obtener señales seno, escalón, cuadrada o rampa. Después de cambiar el valor de las ganancias desde la terminal, se obtuvieron lo siguientes resultados:

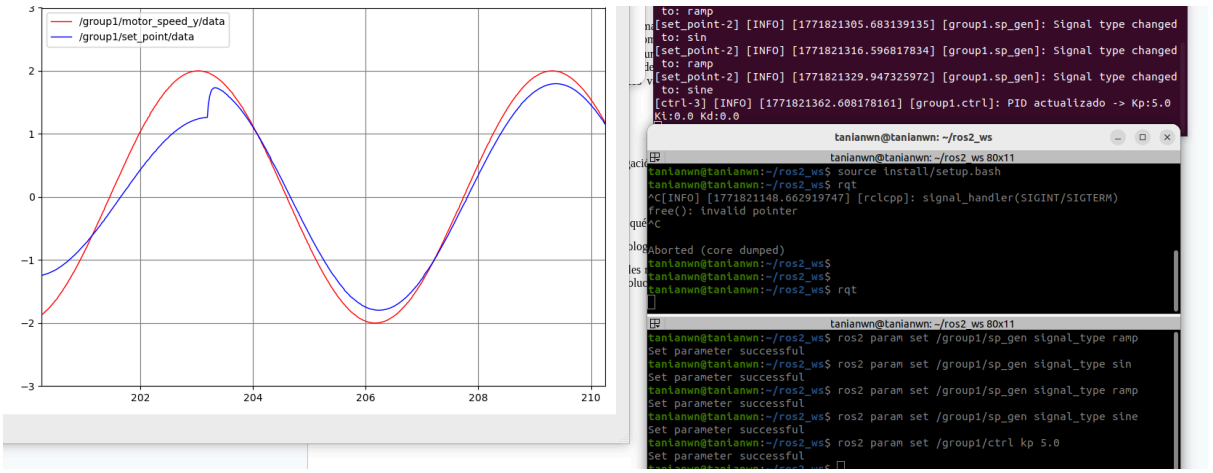


Figura 8. Gráficas de entrada y salida con kp=5, kd=0, ki=0

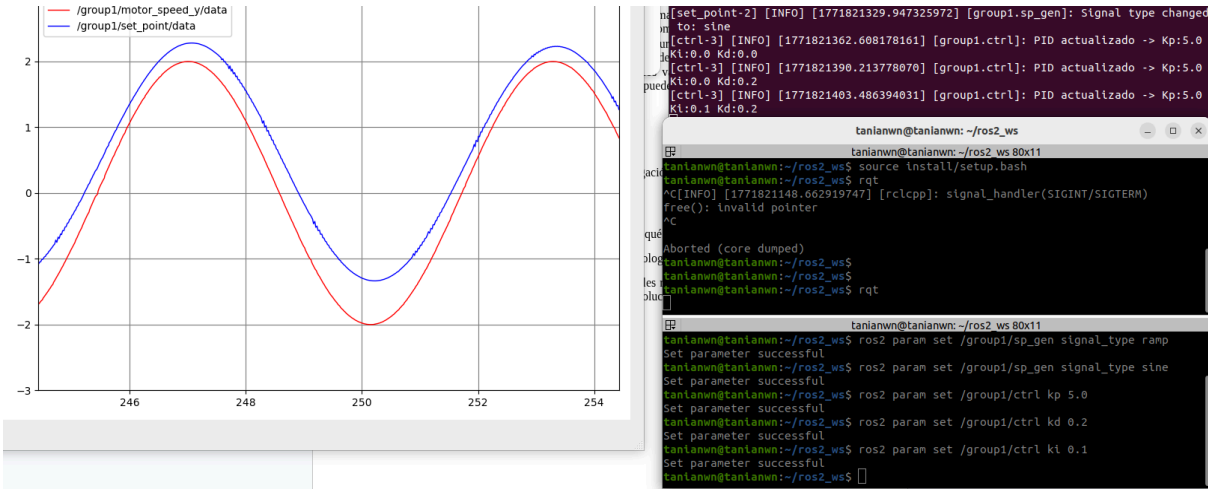
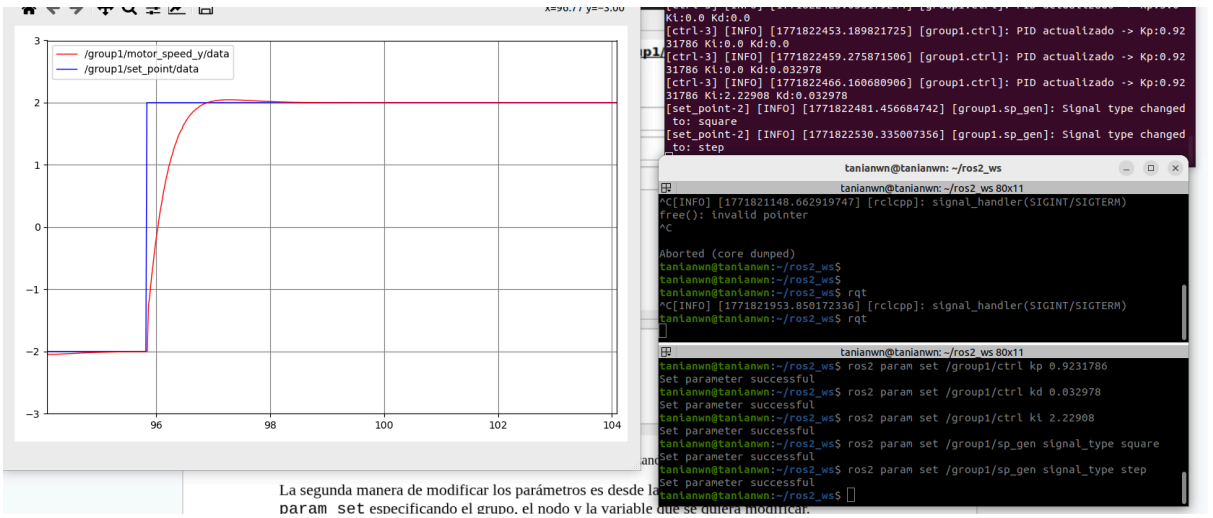


Figura 9. Controlador con Kp 5.0, Kd 0.2 y Ki 0.1. Se muestra un poco de ruido



La segunda manera de modificar los parámetros es desde la
`param set` especificando el grupo, el nodo y la variable que se quiera modificar.

Figura 10. Cambio de señal a step con el controlador de Matlab

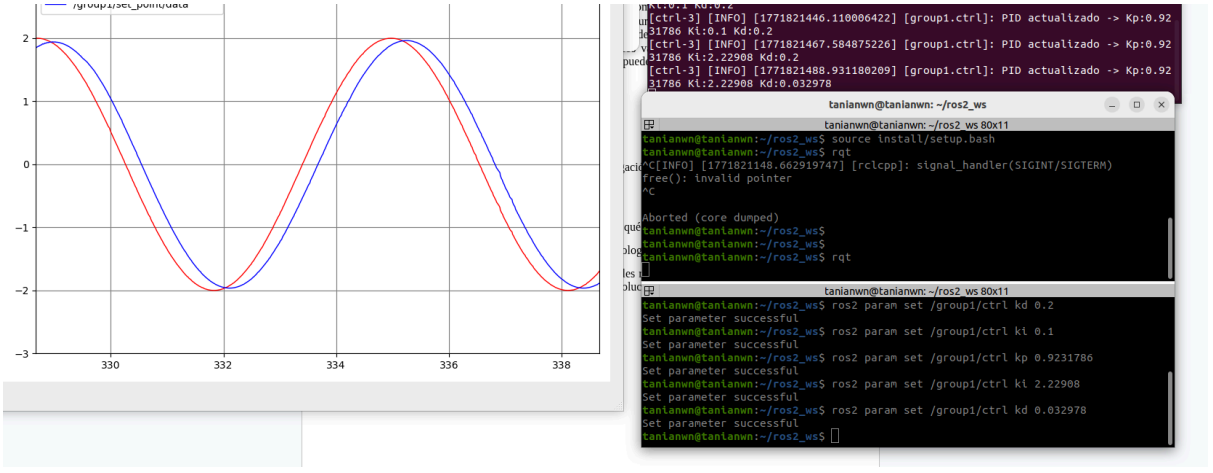


Figura 11. Controlador con la señal original sinusoidal utilizando los parámetros de Matlab

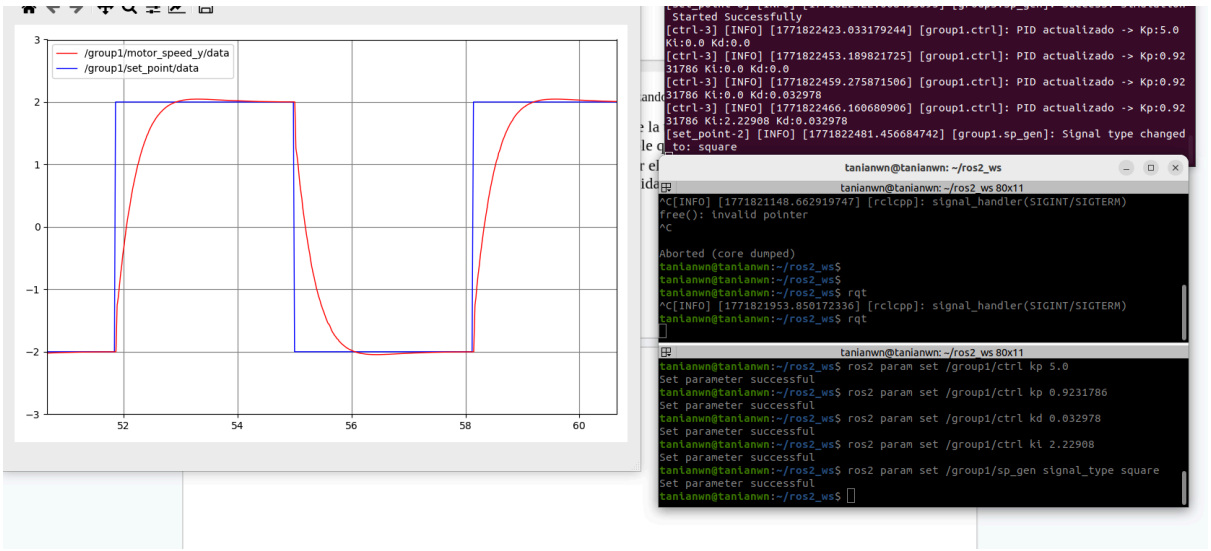


Figura 12. Cambio de señal por un square utilizando los parámetros de matlab

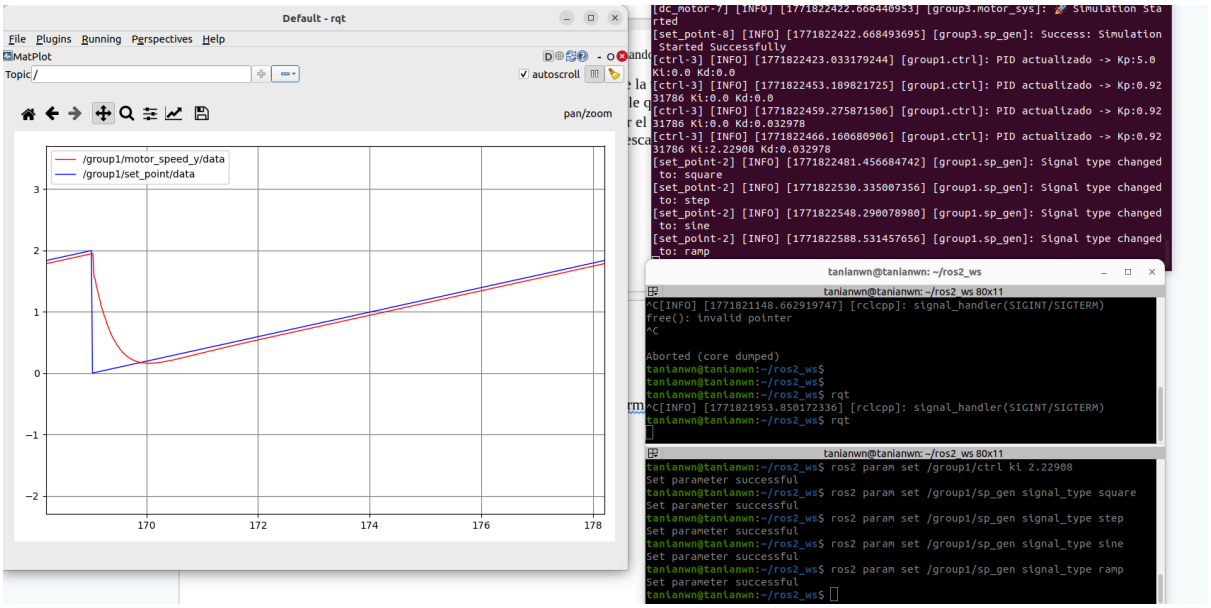


Figura 13. Cambio de señal a ramp utilizando los parámetros de matlab

Se puede observar que gracias al control aplicado al motor, la señal de este se aproxima a la señal de entrada al que este se le aplica.

Conclusiones

Los objetivos del reto se completaron de manera exitosa ya que se logró diseñar un sistema de control de lazo cerrado mediante la programación de un controlador PID. El método de sintonización que se usó fue importante debido a que se obtuvieron los resultados esperados que es corregir la señal del motor para que sea lo más parecida a las de entrada. Gracias a las herramientas y funciones de ROS 2 como los parámetros y las gráficas de rqt_plot fue mucho más sencilla la implementación y sintonización del controlador ya que ofrecen funciones que facilitan el cambio de los valores de algunas variables. De igual manera, los namespaces ayudaron a ejecutar múltiples plantas al mismo tiempo sin colisión

de datos. A pesar de que los objetivos se cumplieron, el sistema sigue presentando ciertas limitaciones que son propias de la programación del controlador. Al ser en tiempo discreto, lo que más afectó fue a la ganancia derivativa debido a que puede generar ruido en la señal de control al ser sensible a cambios bruscos de la señal. Además, los parámetros calculados con la herramienta de Simulink puede que requieran un poco más de ajuste en la ejecución en ROS para alcanzar una mejor estabilidad. Para la mejora del controlador se pueden utilizar otros métodos de programación de un controlador PID para que la integración y la derivada sean mucho más exactas. También se puede añadir un control anti-windup para evitar sobre impulsos de la acción integral.

Referencias

Åström, K., & Hägglund, T. (2004). Revisiting the Ziegler–Nichols step response method for PID control. Journal Of Process Control, 14(6), 635-650. https://doi.org/10.1016/j.jprocont.2004.01.002

Open Robotics. (s. f.-a). *Creating custom msg and srv files*. ROS 2 Documentation: Humble Documentation. Recuperado 21 de febrero de 2026, de https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Custom-ROS2-Interfaces.html

Open Robotics. (s. f.-b). *Parameters*. ROS 2 Documentation: Humble Documentation. Recuperado 21 de febrero de 2026, de https://docs.ros.org/en/humble/Concepts/Basic/About-Parameters.html

Open Robotics. (s. f.-c). *Services*. ROS 2 Documentation: Humble Documentation. Recuperado 21 de febrero de 2026, de https://docs.ros.org/en/humble/Concepts/Basic/About-Services.html

Ramachandran, R. (2023, 17 abril). *Domain ID and Namespace in ROS 2 for Multi-Robot Systems*. Medium. https://blog.robotair.io/domain-id-and-namespace-in-ros-2-for-multi-robot-systems-9a939ae3fa40

Vilanova, R., & Alfaro, V. M. (2011). Control PID robusto: Una visión panorámica. Revista Iberoamericana de Automática E Informática Industrial RIAI, 8(3), 141-158. https://doi.org/10.1016/j.riai.2011.06.003

Understanding services — ROS 2 Documentation: Foxy documentation. (s. f.). https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html