Reto Semanal 2. Manchester Robotics

Oscar Ortiz Torres A01769292

Yonathan Romero Amador A01737244

Ana Itzel Hernández García A01737526

Fundamentación de robótica

Grupo 101

Jueves 27 de Febrero 2025

Resumen

Este proyecto se enfocó en la implementación de un sistema de control en lazo cerrado en ROS2 para simular el control de un motor de corriente continua (DC) utilizando controladores PID. Se integraron tres nodos principales: `motor_sys_ROSario, sp_gen_ROSario` y `control_pid_ROSario`, los cuales se encargan de simular el comportamiento del motor, generar la señal de referencia y ejecutar el controlador PID, respectivamente. El sistema fue diseñado para permitir la modificación dinámica de parámetros en tiempo real y utilizar herramientas de visualización como rqt_plot y rqt_graph para monitorear y analizar el rendimiento del sistema.

Además, se implementaron servicios para controlar el encendido y apagado del sistema de control. Se consiguió el desarrollo exitoso de los nodos y la creación de un sistema funcional de lazo cerrado, que responde adecuadamente a cambios en los parámetros del controlador y tipo de señales de entrada.

Objetivos

1. Implementación de un sistema de control en lazo cerrado en ROS2
    a. Integrar tres nodos (`motor_sys`, `sp_gen` y `ctrl`) para la simulación de un motor DC con control de tipo P, PI o PID.
    b. Configurar la comunicación entre los nodos mediante tópicos adecuados.
2. Desarrollo e implementación del nodo de control (`ctrl`)
    a. Diseñar el algoritmo de control, eligiendo entre controladores P, PI o PID.
    b. Calcular los parámetros del controlador basándose en la función de transferencia dada.

   c. Garantizar la correcta suscripción a `/set_point` y `/motor_output_y` y la publicación en `/motor_input_u`.

3. Gestión de parámetros dinámicos y configuración en ejecución
   a. Utilizar un archivo launch para definir los parámetros del sistema.
   b. Permitir la modificación de parámetros durante la ejecución con `rqt_reconfigure`, y a través de terminal, asegurando la validación de la entrada del usuario.
4. Uso de herramientas de visualización y análisis en ROS2
   a. Implementar `rqt_plot` para visualizar la respuesta del sistema en tiempo real.
   b. Emplear `rqt_graph` para analizar la estructura de comunicación entre nodos.
5. Automatización y control del sistema mediante *launch* files y *services*
   a. Ejecutar los tres nodos simultáneamente usando *namespaces* en el archivo *launch*.
   b. Implementar *services* para iniciar y detener los nodos `motor_sys` y `ctrl`.

Introducción

En el desarrollo de sistemas de control en robótica es primordial comprender la estructura y funcionamiento de los diversos elementos que permiten la comunicación y el control de los dispositivos. Bajo esta perspectiva, conceptos como *namespaces*, *parámetros* y *custom messages* son determinantes en la organización y transmisión de datos dentro de un sistema basado en ROS2.

Los *Namespaces* en ROS2 permiten que el sistema ejecute dos nodos similares sin generar conflicto entre los nombres de los nodos o tópicos. Cuando el archivo contiene una gran cantidad de nodos, se recomienda definir un namespace global, de modo que cada nodo anidado

herede automáticamente dicho namespace. (*Managing Large Projects — ROS 2 Documentation: Humble Documentation*, s. f.)

Los *parámetros* están asociados con nodos individuales y se utilizan tanto para la configuración inicial como la ejecución, sin necesidad de modificar el código. Su duración está vinculada al ciclo de vida del nodo.

Los parámetros se identifican mediante el nombre del nodo, su namespace, el nombre del parámetro y, opcionalmente, el namespace del parámetro. Pueden almacenar valores de tipo bool, int64, float64, string o byte. De manera predeterminada, todos los nodos deben declarar todos los parámetros que utilizarán a lo largo de su funcionamiento. (*Parameters — ROS 2 Documentation: Humble Documentation*, s. f.)

Custom Messages permite definir estructuras de datos específicas para la transmisión de información entre nodos, especialmente cuando las interfaces predefinidas no son suficientes. Gracias a su flexibilidad, estos mensajes optimizan la comunicación al adaptarse a las necesidades del sistema.

Para su creación, primero se debe generar un paquete de interfaces que contenga los archivos .msg y .srv, donde se define la estructura de los datos especificando los tipos como int64, float64 o utilizando mensajes de otros paquetes.1

Una vez creados los mensajes y servicios personalizados, es necesario configurar los archivos CMakeLists.txt y package.xml para que ROS 2 los reconozca y genere el código correspondiente, permitiendo su uso en nodos. (*Creating Custom Msg And Srv Files — ROS 2 Documentation: Jazzy Documentation*, s. f.)

Los *controladores PID* son uno de los mayores empleados debido a su versatilidad, estos reciben datos de entrada de sensores, calcula el error entre el valor deseado y el valor sensado.

Esto lo hace a través de 3 métodos, de ahí su nombre PID. El proporcional, el integral y el derivativo.

Esto se realiza a través de un ciclo cerrado de control el cual se define como un sistema el cual posee una retroalimentación por el cual generar un error. Con esto en mente tenemos que el componente proporcional de nuestro PID depende únicamente del error, esta ganancia determina la relación entre la respuesta de salida y la señal de error. Por lo que esta ayuda a aumentar la velocidad de respuesta de un sistema, aunque esto conlleva un problema debido a que esta velocidad aumenta la oscilación.

Otro de los componentes del controlador es el controlador integral, que suma el término del error con respecto al tiempo, esto hace que el error con tiempo se reduzca a 0. Aunque aumentar esta ganancia aumentará la velocidad en la que eliminará el error, teniendo cuidado de no desestabilizar el sistema, saturando al controlador conocido como windup.

Por último tenemos al controlador integral, este hace que disminuya la salida si el proceso va rápidamente, esta derivada es proporcional a la tasa de cambio. Debido a esto esta ganancia es susceptible al ruido.(*Explicación Sobre el Controlador PID y la Teoría*, 2006)

Solución del problema

La función de transferencia qué representa el sistema del motor es la siguiente:

$$\frac{Y(s)}{U(s)} = \frac{K}{Ts+1}$$

Donde:

$Y(s)$ es la salida del sistema

$U(s)$ es la entrada del sistema

K es la ganancia estática del sistema

T es la constante de tiempo del sistema

En el código de la planta proporcionada, K tiene valor de 1.75 y T de 0.5. Con dichos valores en la función, se simuló el sistema en Simulink con una señal de escalón unitario en lazo abierto:

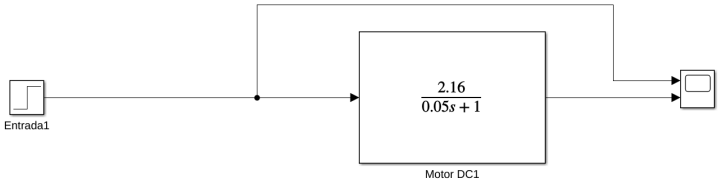

Imagen 1. Sistema de lazo abierto

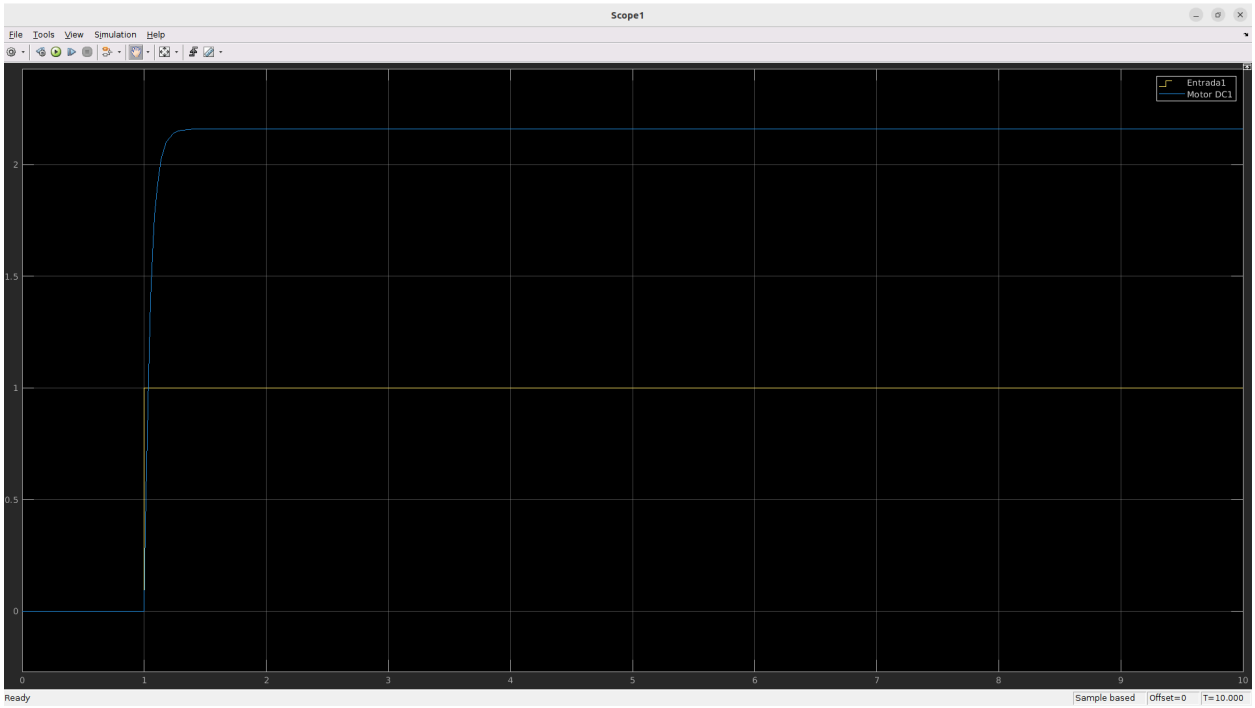

Imagen 2. Respuesta del sistema de lazo abierto

Posteriormente, se simuló en lazo cerrado con una ganancia unitaria para la parte del sensor y un bloque PID Controller para diseñar el controlador:

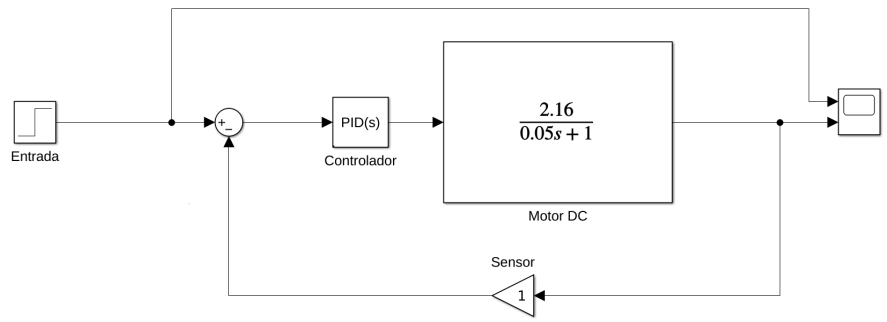

Imagen 3. Sistema de lazo cerrado

Se utilizó la herramienta PID Tuner de MATLAB para la definición de los parámetros del controlador:

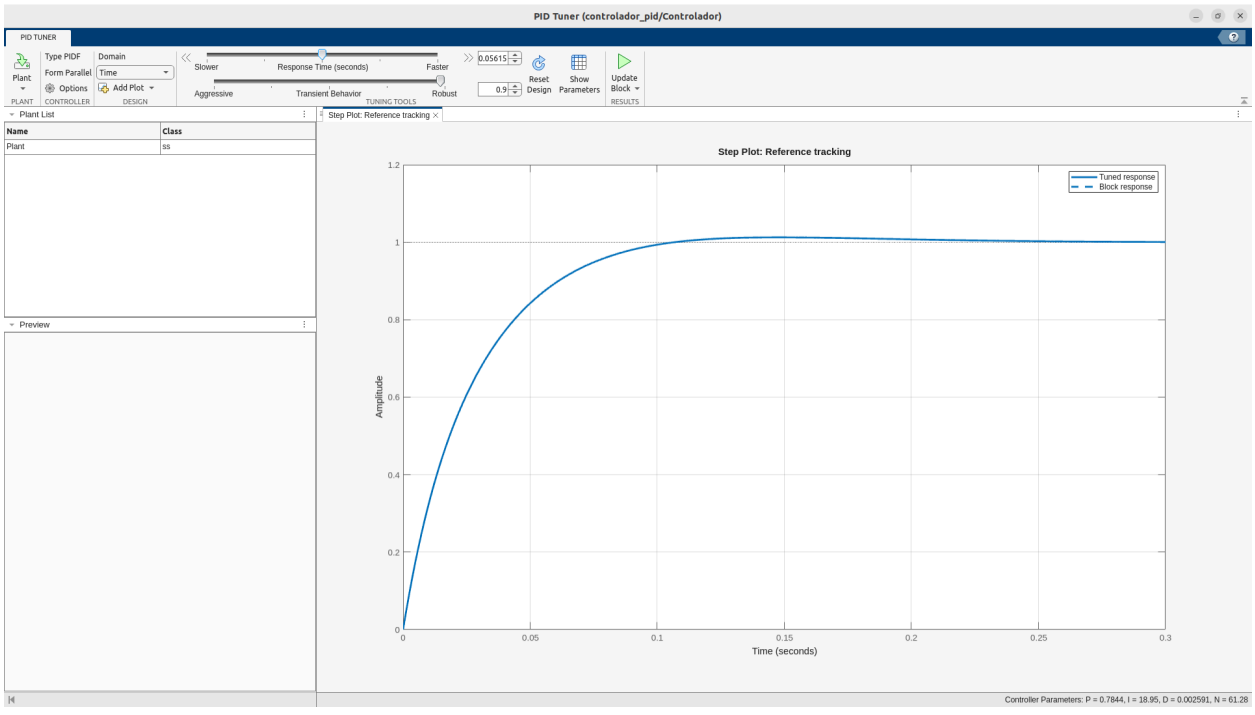

Imagen 4. Ajustes de controlador en PID Tuner

Los parámetros obtenidos fueron:

$$k_p = 0.784448352257978$$

$$k_i = 18.9480680587262$$

$$k_d = 0.00259088726253201$$

Configurando el bloque con los parámetros anteriores, la respuesta al escalón unitario mejoró de la siguiente manera:

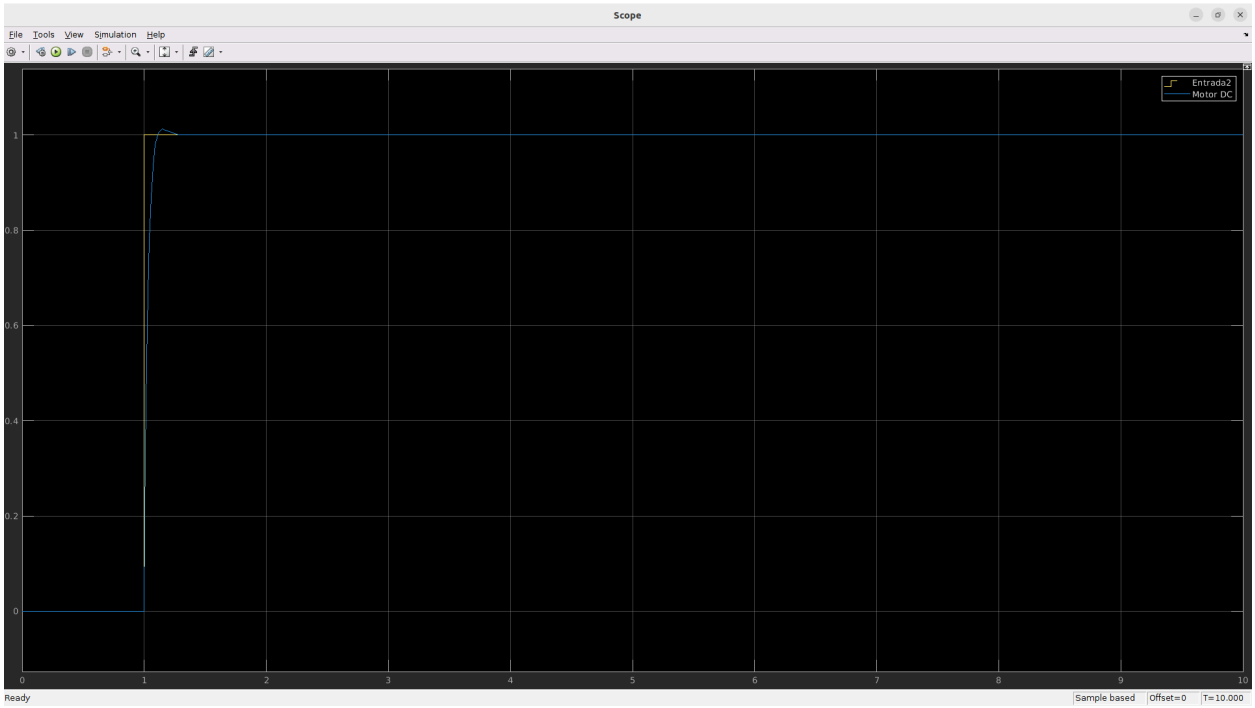

Imagen 5. Respuesta de sistema de lazo cerrado

Los nodos de ROS2 qué interactúan dentro del paquete `motor_control_ROSario` se desarrollaron de la siguiente manera:

1. Nodo `set_point_node_ROSario` (*set_point_ROSario.py*)

   Este nodo genera la señal de referencia para el sistema de control. Se utilizó como base el código proporcionado por Manchester Robotics, realizando las siguientes modificaciones:

a. Se añadió el parámetro `type_flag`, permitiendo seleccionar el tipo de señal de salida:
    i. 0: señal senoidal
    ii. 1: señal cuadrada
b. Se incorporaron los parámetros `amplitude` y `omega` para ajustar la amplitud y la frecuencia angular de la señal, permitiendo modificarlos en tiempo de ejecución.
c. Publica los valores de la señal de referencia en el tópico `/set_point_ROSario`.
d. Opera a 50 Hz y utiliza mensajes de tipo Float32, con una cola de 10 mensajes.

2. Nodo `control_pid_ROSario` (*control_pid_ROSario.py*)

   Este nodo implementa un controlador PID, encargado del calcular la señal de control basada en la referencia y la retroalimentación del sistema. Sus características principales incluyen:

   a. Parámetros configurables `kp`, `ki` y `kd`, lso cuales pueden modificarse dinámicamente durante la ejecución. Se verifica qué los valores ingresados sean mayores o iguales a 0.
   b. Suscripción a los tópicos:
      i. `/set_point_ROSario`: recibe la referencia del sistema.
      ii. `/motor_speed_y_ROSario`: recibe la salida del motor simulada.
   c. Publicación en el tópico `/motor_input_u_ROSario`, enviando la señal de control para ajustar la velocidad del motor.

d. Cálculo en cada iteración de los términos:

   i. Error entre la referencia y la salida del motor.

   ii. Término proporcional, integral y derivativo del controlador PID.

      1. $u = k_p * e(t) + k_i \int_0^t e(t)dt + k_d \frac{dt}{t} e(t)$

   iii. Sumatoria de los términos para generar la señal de control.

e. Almacena variables interenas para la siguiente iteración y opera a 100 Hz con mensajes de tipo Float32 y una cola de 10 mensajes.

3. Nodo `dc_motor_ROSario` (*dc_motor_ROSario.py*)

   Este nodo simula el comportamiento de un motor de corriente directa de primer orden, utilizando la ecuación diferencial correspondiente. Se utilizó el código base de Manchester Robotics, con las siguientes especificaciones:

   a. Parámetros configurables par ala simulación

      i. Constante de ganancia K.

      ii. Constante de tiempo T

      iii. Condición inicial del sistema

   b. Se suscribe al tópico `/motor_input_u_ROSario` para recibir la señal de control generada por el nodo PID.

   c. Publica en el tópico `/motor_speed_y_ROSario`, proporcionando la retroalimentación necesaria para el lazo cerrado.

   d. Opera a 50 Hz con mensajes de tipo Float32 y una cola de 10 mensajes.

Para permitir la modificación de parámetros en tiempo de ejecución, se utilizaron callbacks a través de la función `add_on_set_parameters_callback()`. Dentro de estos

callbacks, se realizan las verificaciones necesarias para garantizar el correcto funcionamiento de los nodos y evitar valores no permitidos.

Se desarrollaron dos archivos de lanzamiento en ROS2 para facilitar la ejecución del sistema:

1. *challenge2_ROSario_launch.py*
    a. Define individualmente los tres nodos qué conforman al sistema,
    b. Especifica para cada nodo:
        i. Nombre del nodo
        ii. Paquete y ejecutable
        iii. Tipo de salida
        iv. Valores iniciales de los parámetros
2. *challenge2_ROSario1_launch.py*
    a. Define globalmente el paquete y el archivo de lanzamiento principal (*challenge2_ROSario_launch.py*).
    b. Establece los namespaces del sistema de `group1`, `group2` y `group3` de la siguiente manera:
        i. Obtiene la dirección del paquete y el archivo de lanzamiento principal
        ii. Crea las direcciones fuente de cada grupo
        iii. Asignar las descripciones de cada namespace
    c. Adicionalmente, ejecuta herramientas de visualización y análisis en ROS2:
        i. `rqt_plot`: para graficar las señales del sistema en tiempo real.

ii. `rqt_graph`: para visualizar la estructura del grafo de nodos y sus interconexiones

iii. `rqt_reconfigure`: para modificar parámetros en tiempo de ejecución.

En este caso se utilizó un servicio para poder encender o apagar nuestro controlador, para esto se utilizó un nuevo paquete *servicio_ctrl* dentro de esta paqueteria se determinó el *SwitchCtrl.srv* el cual tiene 3 variables

1. *bool enable* : esta variable es la que toma el valor de switch en nuestro nodo.
2. *bool success* : esta variable de debugging se utiliza para poder informar al usuario que los cambios han sido aplicados.
3. *string message* : esta variable guarda el mensaje para el usuario. Informando del cambio de estado del nodo.

Una vez creado el nuevo paquete se implementa el servicio en el *control_pid_ROSario.py* para esto se importa nuestro nuevo paquete creado, dentro de la definición de nuestro nodo crearemos el servicio *EnableCtrl_ROSa* el cual sera nuestro servicio en base a nuestro archivo *SwitchCtrl.srv* .

Además se tiene que definir una nueva función *control_service_callback()* el cual utilizaremos para leer nuestro servicio, dependiendo de las variables se procesarán los datos con un PID o simplemente se mandará la entrada al motor sin un procesamiento. Así teniendo la capacidad de cambiar entre un control de lazo cerrado a uno abierto.

Para poder cambiar nuestro servicio ocuparemos el comando *`ros2 service call /<namespace>/EnableCtrl_ROSa servicio_ctrl/srv/SwitchControl '{enable: <bool>}'`* Con

este comando cambiamos nuestro archivo .srv para así prenderlo con *True* o apagar nuestro controlador con *False*. Así mismo podemos seleccionar que controlador queremos cambiar de estado debido a que podemos seleccionar el namespace al que le queremos modificar el servicio.

Resultados

A continuación se presentan las evidencias y análisis sobre el funcionamiento y desempeño del sistema de control del motor, a través de simulaciones y ejecuciones de en ROS2. Los resultados se obtuvieron mediante el uso de diversas herramientas de visualización y análisis.

Gráfica obtenida utilizando `rqt_graph` que muestra la interacción entre entre los tres nodos del sistema: `sp_gen_ROSario`, `control_pid_ROSario` y `motor_sys_ROSario`. Se puede observar cómo la señal de referencia generada por el nodo `sp_gen_ROSario` es enviada al controlador PID, que a su vez genera una señal de control publicada en el nodo `motor_sys_ROSario`.. Este nodo simula la salida del motor y envía la retroalimentación necesaria al controlador.

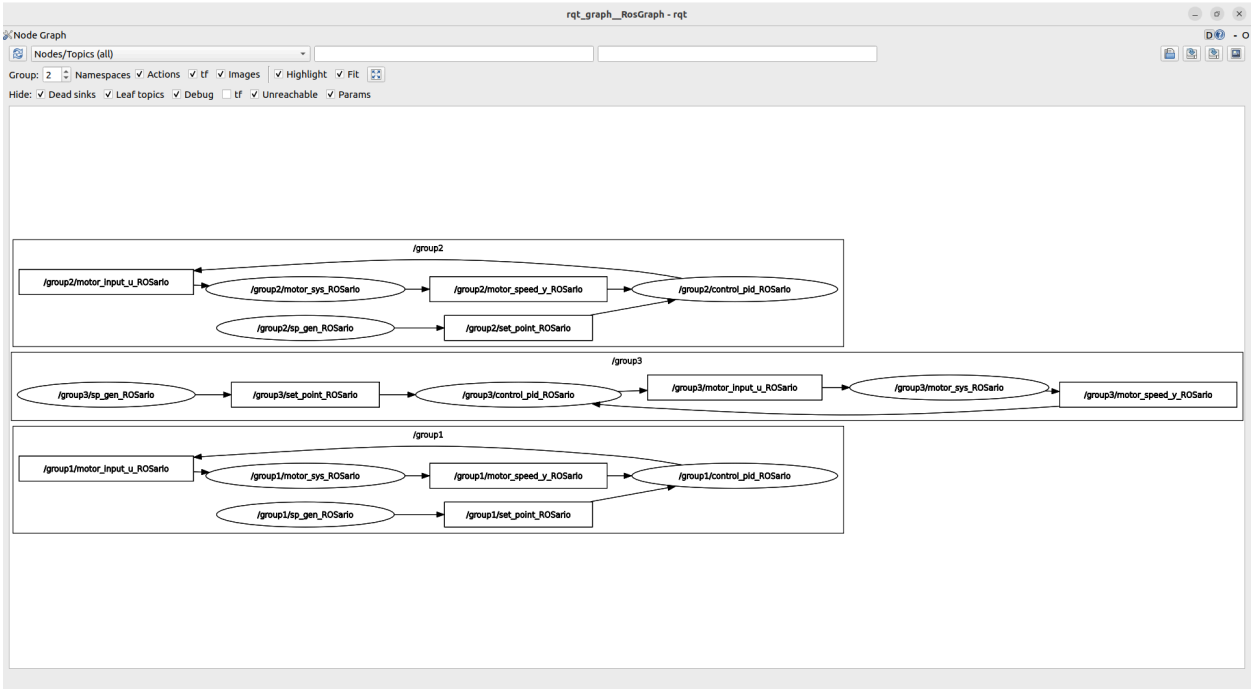

Imagen 6. Grafo de los motores del sistema

Usando `rqt_plot`, se visualiza en tiempo real la respuesta del sistema al aplicar una señal senoidal. En la gráfica se muestra como la salida del motor, en el tópico `motor_speed_y_ROSario`, responde a la referencia enviada desde el nodo `set_point_ROSario`, permitiendo observar la efectividad del controlador PID en la estabilización y seguimiento de la señal de referencia. Mientras que `motor_input_u_ROSario` representa la señal de entrada al motor.

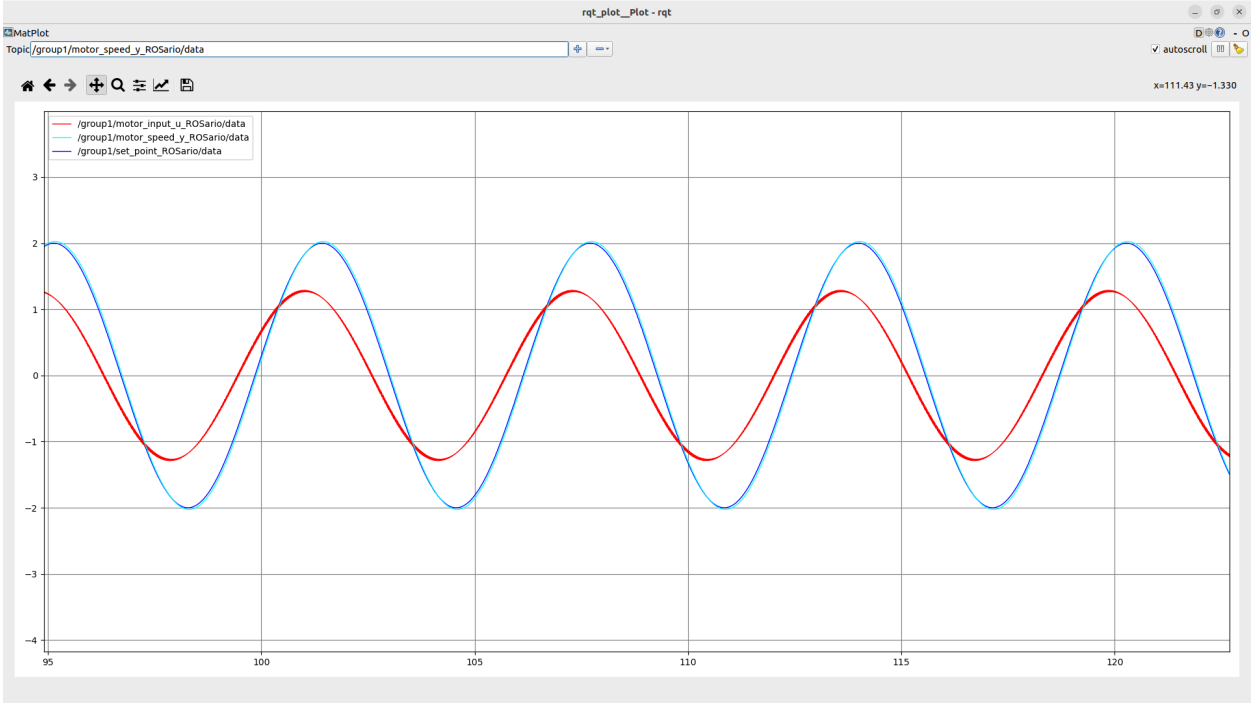

Imagen 7. Graficación de los tópicos de group1

Por medio de la herramienta `rqt_reconfigure` se modifica la señal de entrada a través de la opción se puede eligiendo `type_flag` donde 0.0 es para integrar una señal senoidal y 1.0 para una señal cuadrada, respetando la magnitud y frecuencia angular. Los resultados mostraron que el sistema es capaz de adaptarse a diferentes tipos de señales, tomando en cuenta que al ser la señal cuadrada cuenta con un cambio abrupto, tomándole un poco de tiempo al sistema adaptarse mientras que en la senoidal se adapta rápidamente.

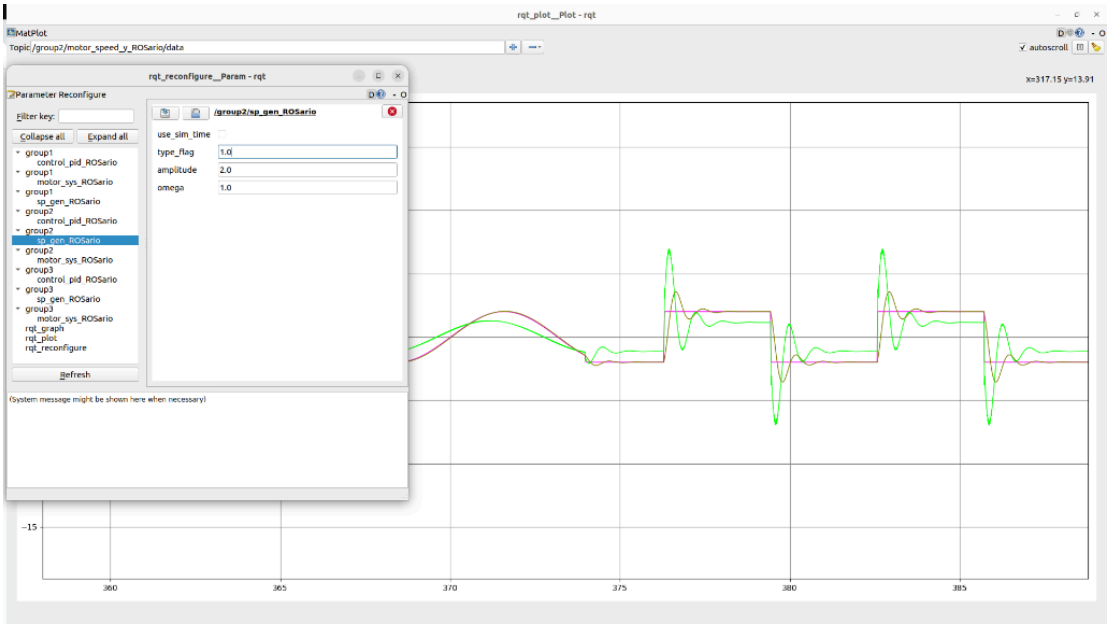

Imagen 8. Graficación de respuesta de group2 al cambio del tipo de señal

El sistema cuenta con una modificación dinámica de los parámetros del controlador PID durante la ejecución del sistema `rqt_reconfigure`. Los parámetros K_p, K_i y K_d fueron ajustados en tiempo real, lo que permite observar cómo la respuesta del sistema se afecta por los cambios de las ganancias del controlador. Resultando que el sistema es capaz de ajustarse a los nuevos valores de los parámetros y mantener el control en la salida del motor.

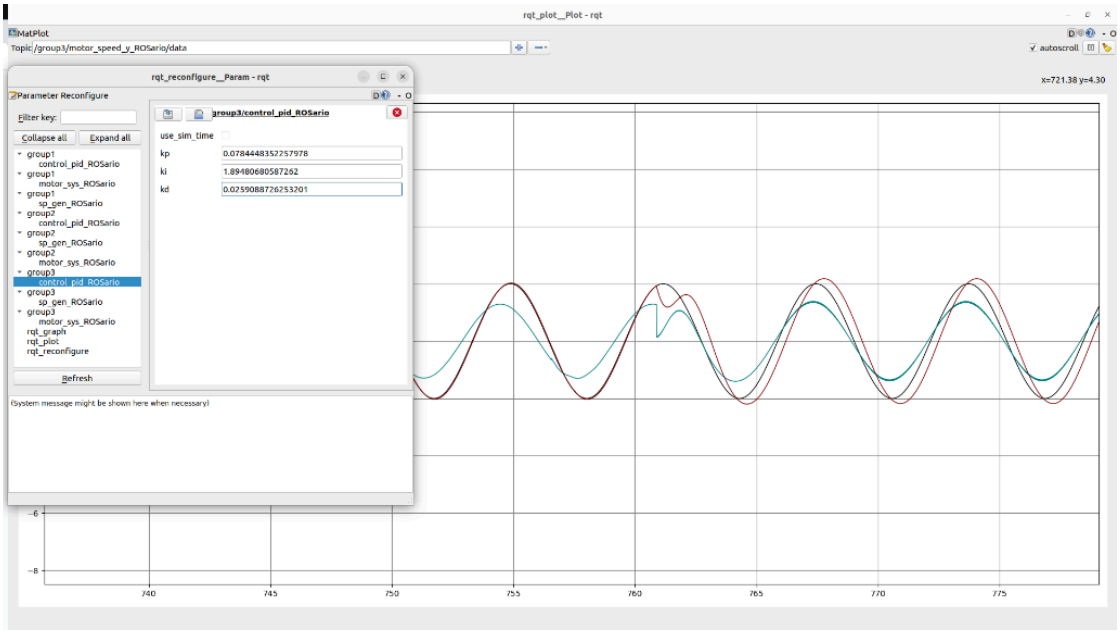

Imagen 9. Graficación de respuesta de group2 al cambio de los parámetros Kp, Ki y Kd

Por medio de la línea de comandos podemos activar nuestro controlador, ,por lo que podemos ver en la gráfica el momento donde nosotros establecemos nuestro *enable* en ***True*** ya que nuestra línea roja (*motor_speed_y_ROSario/data*) empieza a sobreponerse a nuestra entrada. Esto a través de `ros2 service call /group2/EnableCtrl_ROSa servicio_ctrl/srv/SwitchControl '{enable: true}'` , si queremos volver a apagar nuestro controlador solo tenemos que establecer nuestro *enable* en ***False***

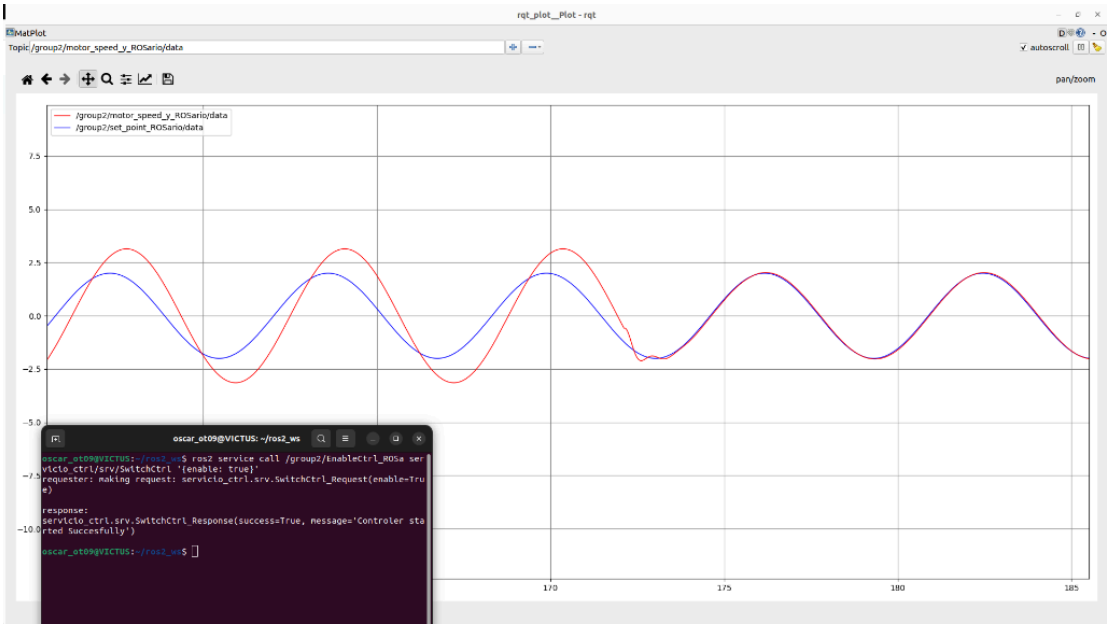

Imagen 10. Graficación de respuesta de group2 a la activación del nodo controlador

Conclusiones

Este reto resultó ser mucho más complejo, por lo que no todos nuestros objetivos pudieron ser completados debido a limitaciones de tiempo. Sin embargo, logramos implementar correctamente el nodo controlador PID, lo que nos permitió desarrollar un sistema de lazo cerrado con el nodo motor. Cumpliendo con nuestros objetivos creamos un lanzador para poder crear tres namespaces con sus nodos correspondientes cada uno con los parámetros definidos en el lanzamiento.También conseguimos crear un servicio que nos permite controlar el estado del sistema, lo que facilita encenderlo y apagarlo. Sin embargo, debido a la falta de tiempo, no pudimos integrar esta función con los motores.

Uno de nuestros mayores desafíos fue el trabajo en equipo, especialmente debido a problemas con nuestra metodología de envío de avances. Además, enfrentamos inconvenientes con paquetes que tenían nombres idénticos, por lo que en futuros retos deberíamos definir nombres más específicos desde el inicio para así evitar confusiones.

Bibliografía

Explicación sobre el controlador PID y la teoría. (2006, 31 agosto). NI. https://www.ni.com/es/shop/labview/pid-theory-explained.html?srsltid=AfmBOorcTNI0uV8EJFsKPu0vZHTNUTDS5hcTmMY9KbV8WS4EP132p6Cl

Managing large projects — ROS 2 Documentation: Humble documentation. (s. f.). https://docs.ros.org/en/humble/Tutorials/Intermediate/Launch/Using-ROS2-Launch-For-Large-Projects.html#namespaces

Parameters — ROS 2 Documentation: Humble documentation. (s. f.). https://docs.ros.org/en/humble/Concepts/Basic/About-Parameters.html

Creating custom msg and srv files — ROS 2 Documentation: Jazzy documentation. (s. f.). https://docs.ros.org/en/jazzy/Tutorials/Beginner-Client-Libraries/Custom-ROS2-Interfaces.html

Anexos

Repositorio: https://github.com/oscarOT09/motor_control_ROSario1

Archivo de simulación (Simulink):

https://drive.google.com/file/d/1DwkbE_cEkpdTgPpmlHK8OFzgl3ECspsp/view?usp=sharing