Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Puebla

Materia: Fundamentación de robótica

Clave: TE3001B

Grupo: 101

Profesores:
Alfredo García Suárez
Rigoberto Cerino Jiménez
Juan Manuel Ahuactzin Larios

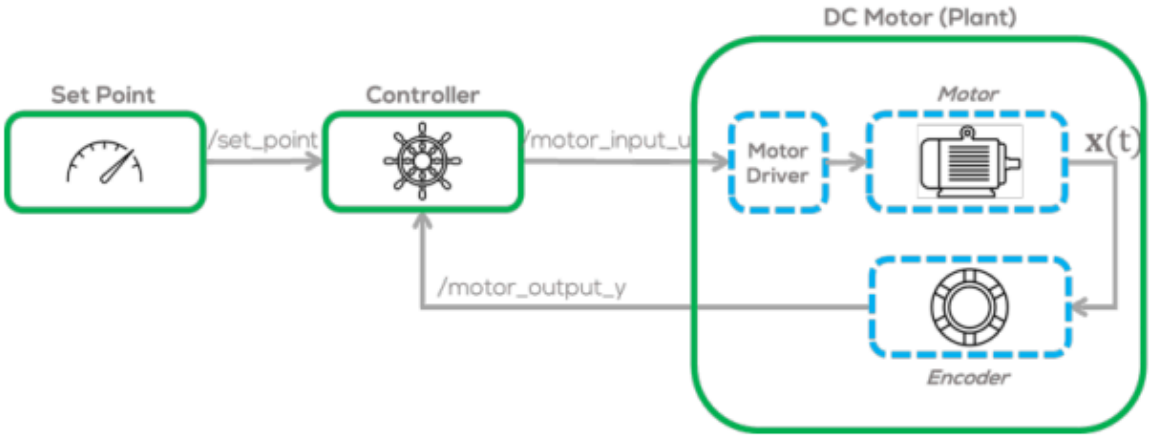
Actividades M_Sensores

Alumnos
Omar Perez Dominguez | A01738306
Ezequiel Luna Trejo | A01738153
Antonio Méndez Rodríguez | A01738269

22 de Febrero De 2026

Resumen

Este proyecto consistió en el diseño y puesta en marcha de un sistema de control para regular la velocidad de un motor de CD utilizando ROS 2 Humble. A partir de una base de paquetes proporcionada, se desarrolló un nodo controlador (controller.py) programado en Python con la librería NumPy y un archivo de automatización (***motor_launch.py***). El objetivo central fue cerrar el lazo de control para que el motor reaccione automáticamente a los cambios en la señal de referencia. Mediante el uso de ***rqt_plot*** y ***rqt_graph***, podemos graficar nuestras señales con el propósito de comprobar que el sistema es capaz de seguir la trayectoria deseada de forma estable, validando así nuestro sistema de control y la correcta integración de los nodos y el intercambio de mensajes personalizados.



Solución del problema

Análisis de la Planta y Configuración de Parámetros En lugar de un modelado gráfico, la solución partió del análisis de las especificaciones técnicas proporcionadas por Manchester Robotics para el motor. Se identificó que el sistema responde a una planta de primer orden, lo cual se tradujo directamente en la configuración del nodo ***dc_motor*** dentro del archivo de lanzamiento. Se establecieron parámetros críticos como la ganancia del sistema (K = 2.16) y la constante de tiempo (tau = 0.05), los cuales dictan la rapidez y la magnitud de la respuesta del motor ante un estímulo de voltaje.

Desarrollo del Nodo Controlador (***controller.py***)

La implementación del nodo controlador se realizó en Python mediante una arquitectura orientada a objetos en donde buscamos la eficiencia matemática y la flexibilidad, utilizando un algoritmo de control PI (Proporcional-Integral) diseñado para equilibrar una respuesta dinámica rápida con la eliminación total del error en estado estacionario. El sistema ejecuta un lazo de control de alta frecuencia (100 Hz) mediante un ***timer*** de ROS 2, donde se emplea la librería NumPy para procesar la señal de error y aplicar una técnica de Anti-windup a través de la función ***np.clip***, la cual limita la acumulación integral entre -100.0 y 100.0 para prevenir saturaciones e inestabilidad en el actuador. Asimismo, se integró una interfaz de parámetros dinámicos que permite la sintonización al momento de las ganancias K_p y K_i, respaldada por un ***parameters_callback*** que valida en tiempo real que los valores sean positivos, garantizando así una operación segura y un seguimiento de trayectoria estable según lo verificado en las pruebas experimentales, anexo una imagen del código elaborado sobre el nodo controller.py.

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32
from rcl_interfaces.msg import SetParametersResult
import numpy as np


class Controller(Node):

    def __init__(self):
        super().__init__('ctrl')

        self.declare_parameter('Kp', 0.5)
        self.declare_parameter('Ki', 0.1)

        # Get Parameters
        self.Kp = self.get_parameter('Kp').value
        self.Ki = self.get_parameter('Ki').value


        self.y = 0.0
        self.sp = 0.0
        self.error_sum = 0.0

        # Publisher
        self.pub_u = self.create_publisher(Float32, 'motor_input_u', 10)

        # Subscribers

        self.sub_y = self.create_subscription(
            Float32,
            'motor_speed_y',
            self.callback_y,
            10
        )

        self.sub_sp = self.create_subscription(
            Float32,
            'set_point',
            self.callback_sp,
            10
        )


        # Timer (loop de control)
        self.sample_time = 0.01  # igual que motor_sys
        self.timer = self.create_timer(self.sample_time, self.control_loop)

        # Parameter Callback
        self.add_on_set_parameters_callback(self.parameters_callback)

        self.get_logger().info('Controller Node Started ')

```

```python
    def callback_sp(self, msg):
        self.sp = msg.data

    # Control Loop
    def control_loop(self):

        # Error
        error = self.sp - self.y

        # Integral (acumulación)
        self.error_sum += error

        # Anti-windup
        self.error_sum = np.clip(self.error_sum, -100.0, 100.0)

        # Control PI
        u = self.Kp * error + self.Ki * self.error_sum

        # Publicar señal de control
        msg = Float32()
        msg.data = float(u)
        self.pub_u.publish(msg)
    # Parameter validation
    def parameters_callback(self, params):
        for param in params:

            if param.name == "Kp":
                if param.value < 0.0:
                    self.get_logger().warn("Invalid Kp! Cannot be negative.")
                    return SetParametersResult(successful=False, reason="Kp < 0")
                else:
                    self.Kp = param.value
                    self.get_logger().info(f"Kp updated to {self.Kp}")

            if param.name == "Ki":
                if param.value < 0.0:
                    self.get_logger().warn("Invalid Ki! Cannot be negative.")
                    return SetParametersResult(successful=False, reason="Ki < 0")
                else:
                    self.Ki = param.value
                    self.get_logger().info(f"Ki updated to {self.Ki}")

        return SetParametersResult(successful=True)
# Main
def main(args=None):
    rclpy.init(args=args)

    node = Controller()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Automatización (***motor_launch.py***)

La automatización y orquestación del sistema se consolidó mediante un script de lanzamiento (***motor_launch.py***), el cual permite la gestión centralizada de los ciclos de vida de los nodos y la inyección de configuraciones críticas para la simulación. La arquitectura del archivo destaca por la instanciación del nodo ***motor_sys***, donde se definieron de forma explícita los parámetros físicos de la planta, incluyendo una ganancia del sistema de 2.16 y una constante de tiempo de 0.05 s, asegurando que el modelo del motor de CD opere bajo condiciones dinámicas realistas y consistentes en cada ejecución. Asimismo, el script coordina la ejecución concurrente del generador de referencias (***sp_gen***) y el nodo controlador personalizado (***ctrl***), utilizando las propiedades ***emulate_tty*** y ***output='screen'*** para garantizar que el flujo de datos y los mensajes de diagnóstico sean legibles directamente en la consola. Esta metodología de despliegue no solo optimiza el flujo de trabajo al eliminar la necesidad de ejecución manual por terminales separadas, sino que también garantiza la integridad de la red de comunicación al sincronizar el arranque de la planta con la activación del lazo de control PI, facilitando así la captura síncrona de datos para su posterior análisis en herramientas de visualización.

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():

    motor_node = Node(
        name="motor_sys",
        package='motor_control',
        executable='dc_motor',
        emulate_tty=True,
        output='screen',
        parameters=[{
            'sample_time': 0.01,
            'sys_gain_K': 2.16,
            'sys_tau_T': 0.05,
            'initial_conditions': 0.0,
        }]
    )

    sp_node = Node(
        name="sp_gen",
        package='motor_control',
        executable='set_point',
        emulate_tty=True,
        output='screen',
    )

    ctrl_node = Node(
        name="ctrl",
        package='motor_control',
        executable='controller',
        emulate_tty=True,
        output='screen',
    )

    l_d = LaunchDescription([
        motor_node,
        sp_node,
        ctrl_node
    ])

    return l_d
```

SImulación en Matlab simulink

Para llevar el comportamiento de un motor de CD al entorno digital, es necesario realizar una transición del mundo continuo al discreto. La ecuación de diferencia es la herramienta matemática que permite esta transformación, traduciendo la ecuación diferencial que rige la física del motor en un algoritmo recursivo que una computadora puede procesar paso a paso.

$$y[k+1] = y[k] + ((-1/tau) * y[k] + (K/tau) * u[k]) * Ts$$

Esta ecuación "gobierna" el sistema simulado, calculando el estado futuro de la velocidad ($y[k+1]$) a partir del estado actual ($y[k]$**)** y la entrada de control ($u[k]$), considerando siempre el tiempo de muestreo (Ts) y los parámetros físicos de ganancia (K) y constante de tiempo (tau).

La función de transferencia es una herramienta matemática que describe el comportamiento dinámico de un sistema en el dominio de la frecuencia compleja (s). Para este contexto, esta función representa el "vínculo" o relación directa entre la señal de voltaje que se envía como entrada y la velocidad angular que obtenemos como salida del motor. Al aplicar la Transformada de Laplace a las ecuaciones físicas del sistema, obtenemos una expresión de primer orden del tipo

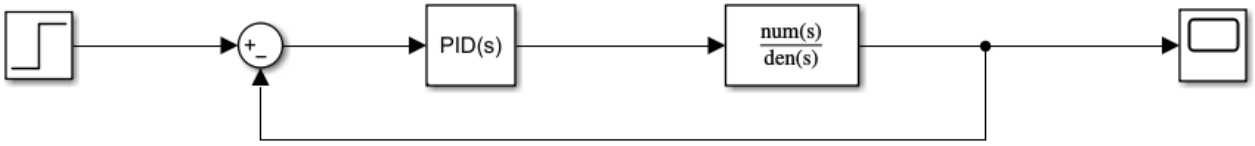
$$G(s) = \frac{Y(s)}{U(s)} = \frac{K}{Ts+1}$$

$Y(s)$: Es la salida del sistema (la velocidad del motor)
$U(s)$: Es la entrada (el voltaje o señal de control que envías desde el nodo /***ctrl***)

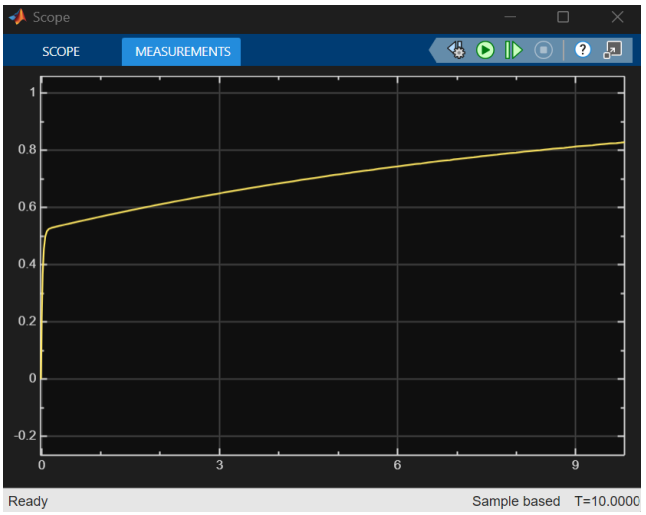
Dónde parámetros clave como la ganancia (K) y la constante de tiempo (tau) definen qué tan potente y qué tan rápido es el motor. Esta representación es fundamental para la etapa de simulación, ya que permite predecir la estabilidad del sistema y sintonizar las ganancias del controlador PI

Experimento 1 — Respuesta al escalón unitario

Escalón = 1



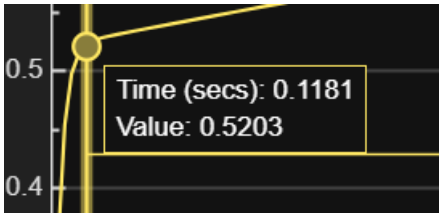
Scope:



Statistics

Transfer Fcn	Max	Min	Peak to Peak	Mean	Standard Deviation	Median	RMS
Time	10.0000 s	401.9018 us					
Value	0.830201445307506	0.008609258226462795	0.8216	0.6819	0.1332	0.7079	0.6947

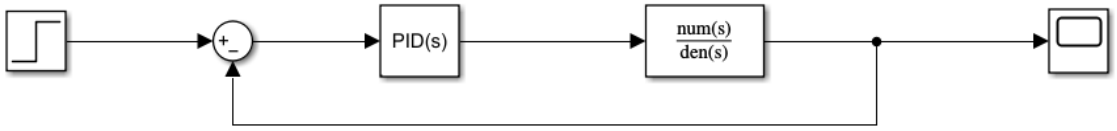
Medir:
Valor final:
0.8302
Tiempo de subida:



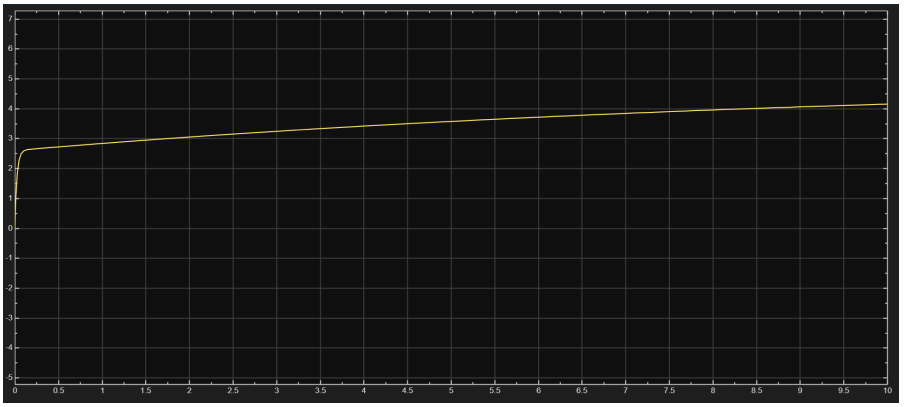
Tiempo establecimiento:
10 segundos
Forma de la curva:
Sobreamortiguada

Experimento 2 — Respuesta al escalón unitario

Escalón: 5



Scope:



Statistics

Transfer Fcn	Max	Min	Peak to Peak	Mean	Standard Deviation	Median	RMS
Time	10.0000	0.0000					
Value	4.150990687228411	0	4.1510	3.3348	0.8275	3.5216	3.4352

Analizar:

proporcionalidad:

Podemos ver que si es un sistema proporcional debido a que la entrada ahora es 5, nuestro valor maximo a los 10s ahora es 4.15 aproximadamente mientras que en el experimento 1 era 0.83 que si lo multiplicas por 5 nos da el aproximado 4.15 esto demuestra que es un sistema lineal.

rapidez:

Al observar el inicio de la curva, el Tiempo de Subida sigue siendo el mismo (cerca de 0.11s). Esto confirma que la rapidez de respuesta depende de la física del motor (tau) y no del tamaño del escalón.

Comportamiento dinámico:

El sistema presenta un comportamiento dinámico de primer orden sobreamortiguado

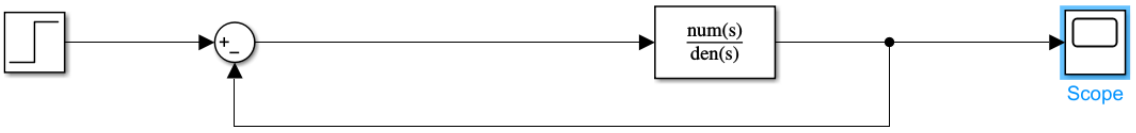
¿Qué tipo de sistema es?

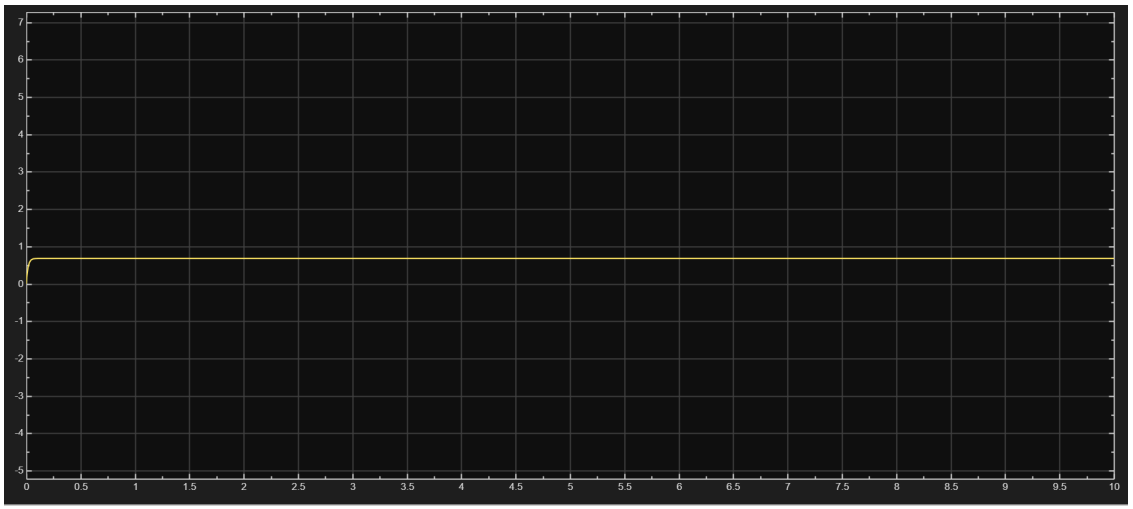
El sistema analizado es un sistema de primer orden, caracterizado por tener un único polo en su función de transferencia de la forma

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

Se define como un sistema estable, lo que garantiza que la salida siempre buscará un punto de equilibrio ante una entrada constante. Asimismo, se clasifica como un sistema de Tipo 0, dado que la planta original no cuenta con integradores puros en el origen. Finalmente, el motor demuestra ser un sistema lineal e invariante en el tiempo (LTI), una propiedad que se validó en el Experimento 2 al observar que la rapidez de la respuesta, o tiempo de subida, se mantiene constante independientemente de la magnitud del escalón de entrada aplicado.

Al retirar el controlador PI y cerrar el lazo, el sistema se comporta como un control proporcional unitario. El valor final medido (0.6835) coincide con el cálculo teórico para un sistema de Tipo 0, evidenciando un error de estado estable. Asimismo, el tiempo de establecimiento se redujo drásticamente a aproximadamente 0.06 s, demostrando que la retroalimentación negativa incrementa la rapidez de la respuesta a costa de la precisión final.





Statistics

Transfer Fcn	Max	Min	Peak to Peak	Mean	Standard Deviation	Median	RMS
Time	10.0000	0.0000					
Value	0.683521438429591 7	0	0.6835	0.6656	0.1005	0.6833	0.6731

Preparación para PID

Sin diseñar PID aún, responder:

▹**¿Qué problema tiene el motor sin control?**
Cuando el motor se implementa sin ningún tipo de control, no cuenta con ningún tipo de corrección y debido a su ausencia provoca un error estacionario baja robustez y falta de precisión

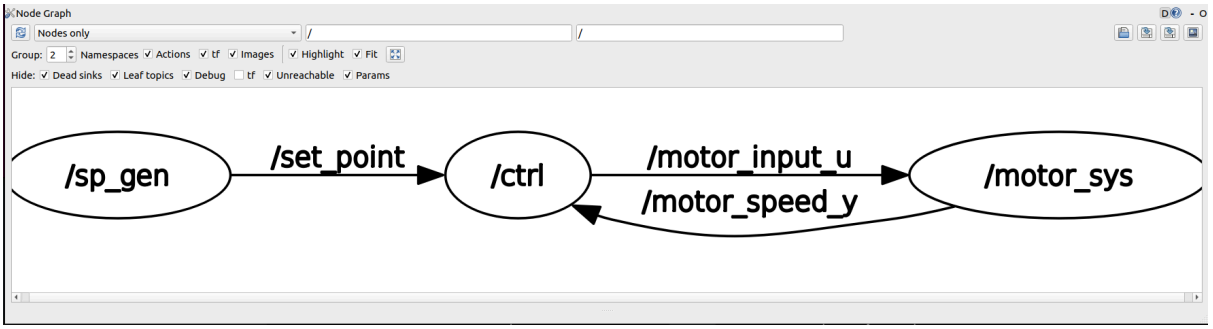
▹**¿Seguiría bien una referencia variable?**
El seguimiento de una referencia variable depende de la dinámica del sistema y del controlador. En sistemas de primer orden, el controlador PI permite buen seguimiento, aunque puede aparecer desfase cuando la referencia varía rápidamente debido a la constante de tiempo del sistema.

▹**¿Por qué se necesita PID?**
No siempre es necesario y depende de cada sistema ya que el controlador PID se utiliza para mejorar simultáneamente la rapidez, estabilidad y precisión del sistema. En este caso el sistema mostró una respuesta estable con solo el proporcional y integral.

▹**¿Qué esperas que haga el término P?**
El término proporcional genera una acción de control directamente proporcional al error instantáneo, permitiendo reducir el error y mejorar la rapidez de respuesta del sistema.
▹**¿Qué hará el término I?**
El término integral acumula el error en el tiempo, permitiendo eliminar el error estacionario y mejorar la precisión del sistema en estado estable.

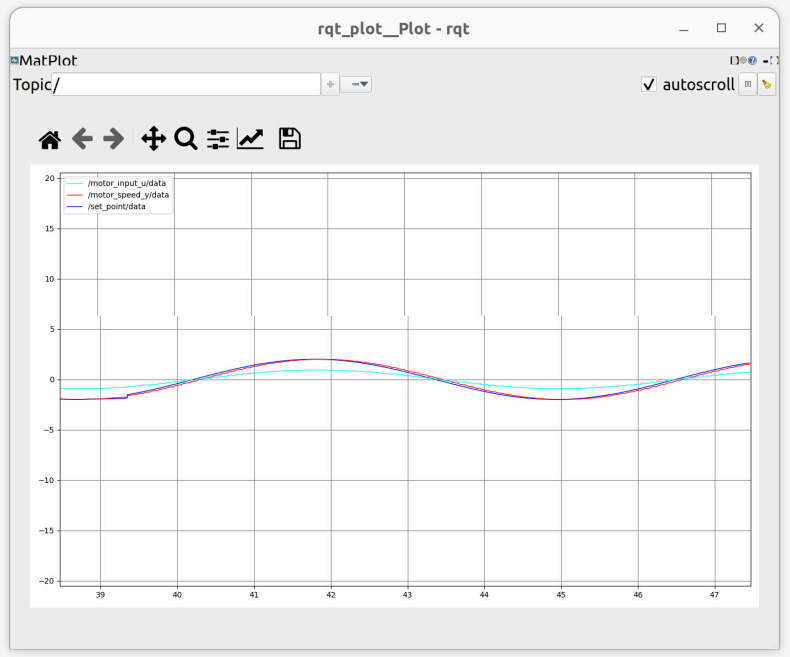
Resultados

La validación de la arquitectura de comunicación se realizó mediante la generación del diagrama de computación del sistema a través de la herramienta ***rqt_graph***, la cual permite visualizar de manera clara el flujo de datos y la interconectividad entre los procesos de ROS 2. En el diagrama obtenido, se identifica un lazo de control cerrado perfectamente estructurado donde el nodo de referencia, denominado ***n__sp_gen***, actúa como el generador de trayectoria al publicar la velocidad deseada en el tópico ***/set_point.*** Esta señal es recibida por el nodo controlador central, ***/ctrl***, el cual desempeña un rol crítico de intermediario al estar suscrito simultáneamente a la referencia y a la retroalimentación proveniente del motor a través del tópico ***/motor_speed_y***. Como resultado del procesamiento interno mediante el algoritmo PI y la lógica de Anti-windup implementada en Python, el controlador pública la señal de mando calculada en el tópico ***/motor_input_u***, la cual es consumida por el nodo de planta ***n__motor_sys***. Este último cierra el ciclo de retroalimentación al enviar la velocidad real alcanzada de vuelta al controlador, confirmando así que la topología de la red es robusta, que no existen nodos aislados y que el intercambio de mensajes personalizados se ejecuta de forma síncrona y eficiente bajo el esquema de diseño establecido.



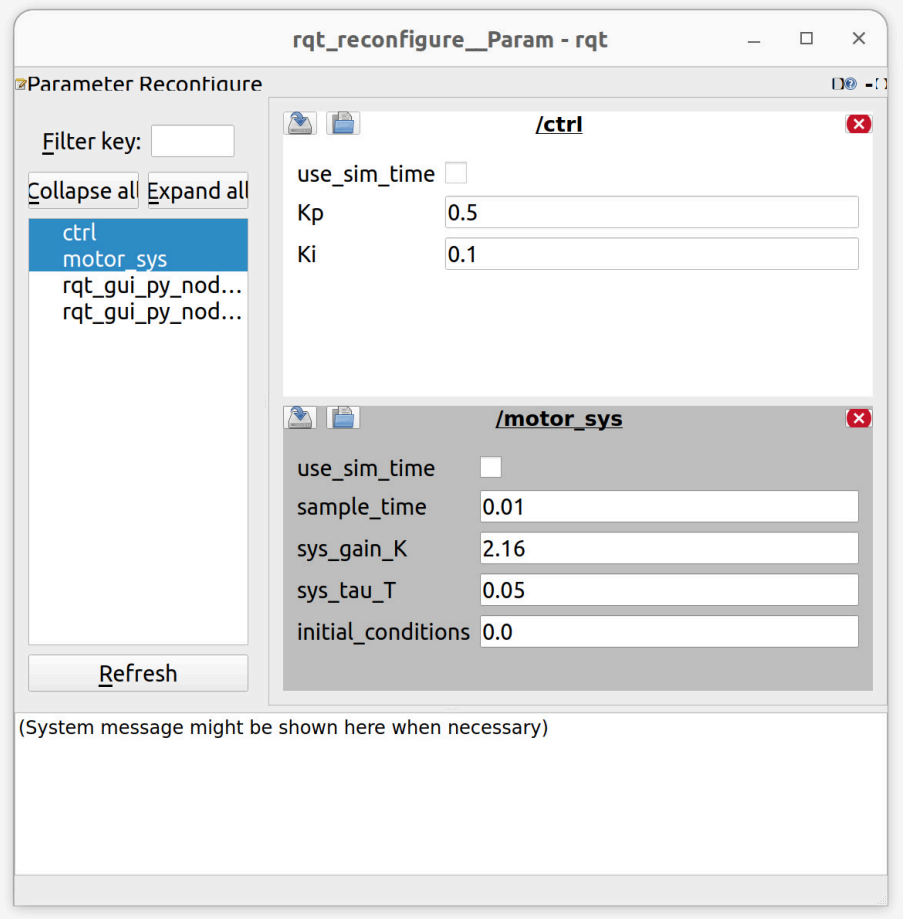
El análisis de la respuesta temporal del sistema evidencia un desempeño altamente satisfactorio del controlador PI en el seguimiento de trayectorias dinámicas. Se observa que la velocidad real del motor (representada por la señal roja) converge de manera casi instantánea con la señal de referencia (línea azul), manteniendo un error de seguimiento despreciable a lo largo de todo el ciclo de operación. Esta precisión es una prueba directa de la efectividad de la acción integral programada, la cual compensa las perturbaciones y la dinámica del modelo físico para eliminar el error en estado estacionario que caracterizaría a un control puramente proporcional. Asimismo, la señal de control o voltaje de entrada (línea cian) presenta una evolución suave y coherente con las variaciones de la referencia, lo que indica que la sintonización de las ganancias K_p y K_i es óptima para la constante de tiempo de la planta.

La ausencia de oscilaciones divergentes o sobretiros pronunciados en las crestas de la señal confirma no solo la estabilidad del lazo cerrado, sino también la correcta implementación de la lógica de Anti-windup, la cual previene que la saturación del actuador degrade la respuesta del sistema ante cambios continuos en la consigna, validando así la robustez del diseño desarrollado para el reto.



Como se observa en el diagrama que generamos (***rqt_graph***), el flujo de información es bidireccional y síncrono; el nodo ***/ctrl*** procesa la diferencia entre la referencia de ***/set_point*** y la velocidad real del motor para enviar una señal de mando al nodo ***n__motor_sys***. Esta robustez estructural se complementa con la flexibilidad operativa visualizada en ***rqt_reconfigure***, donde el uso de parámetros dinámicos permitió establecer ganancias de $K_p = 0.5$ y $K_i = 1$ en tiempo real, optimizando el comportamiento del sistema sin interrumpir la simulación.

De estos parámetros que elegimos se refleja directamente en la respuesta temporal obtenida en ***rqt_plot***, donde se aprecia un seguimiento de trayectoria. La señal de velocidad del motor (línea roja) se superpone con precisión a la señal de referencia (línea azul), lo que valida que la acción integral implementada logró eliminar el error en estado estacionario. Asimismo, la estabilidad de la señal de control (línea cian) confirma que la lógica de Anti-windup y la sintonización de parámetros fueron las adecuadas para la dinámica de la planta (K=2.16, tau=0.05), garantizando un sistema autónomo, estable y capaz de reaccionar eficazmente a cambios continuos en la velocidad deseada.



Conclusiones

Tras la ejecución y validación del sistema de control para el motor de CD, se concluye que los objetivos planteados al inicio de este reto se cumplieron satisfactoriamente. El éxito del reto está en la correcta integración de la arquitectura de software con la teoría de control clásica; se logró diseñar un nodo controlador funcional en Python que, mediante el uso de NumPy, permite un procesamiento de señales eficiente y síncrono. La implementación del algoritmo PI fue fundamental para alcanzar la precisión requerida, ya que la acción integral permitió eliminar el error en estado estacionario que presentaba la planta ante cambios en la referencia.

El cumplimiento total de los objetivos se debe, en gran medida, al aprovechamiento de las herramientas avanzadas de ROS 2. El uso de parámetros dinámicos y la interfaz de ***rqt_reconfigure*** fueron piezas clave que permitieron la sintonización fina de las ganancias K_p y K_i en tiempo real, facilitando la búsqueda de un equilibrio entre rapidez de respuesta y estabilidad sin necesidad de interrumpir el flujo de la simulación. Asimismo, la incorporación de una técnica de Anti-windup mediante ***np.clip*** resultó esencial para mantener la integridad del sistema, evitando que la saturación del actuador provocará oscilaciones divergentes o comportamientos erráticos.

A pesar de los resultados positivos, una posible mejora a la metodología implementada sería la transición hacia un controlador PID completo. La acción derivativa permitiría anticipar la tendencia del error, lo que reduciría aún más el sobretiro (overshoot) en cambios bruscos de trayectoria y mejoraría el tiempo de asentamiento del motor. Además, para futuras iteraciones del proyecto.

Referencias Bibliográficas (APA)

- **Manchester Robotics.** (2024). *DC Motor Control Challenge 2: ROS 2 Humble Integration Guide*.
- **Open Robotics.** (2026). *ROS 2 Humble Hawksbill Documentation*. Recuperado de https://docs.ros.org/en/humble/
- **Ogata, K.** (2010). *Ingeniería de Control Moderna* (5ª ed.). Pearson Educación.
- **Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E.** (2020). *Array programming with NumPy*. Nature, 585(7825), 357-362.
- **Python Software Foundation.** (2026). *Python Language Reference, version 3.10*. Recuperado de https://www.python.org/