Instituto Tecnológico y de Estudios Superiores de Monterrey

TE3001B  Fundamentación de Robótica (Gpo 101)

*Challenge 2*


Autores:

Mariam Landa Bautista // A01736672

Bruno Manuel Zamora García // A01798275

Elías Guerra Pensado // A01737354

Profesor:

Juan Manuel Ahuactzin

Rigoberto Cerino

Alfredo García Suarez


Jueves 27 de Febrero de 2025 Semestre (6) Feb-Jul 2025

Campus Puebla

**-Resumen**

Basándose en los conocimientos adquiridos en el Challenge anterior, en este segundo Challenge se incorporan nuevos conceptos como los *namespaces* y *parameters*. Este reporte presenta la solución al Challenge 2, cuyo objetivo principal es desarrollar un controlador para un motor de corriente continua simulado en ROS. Se detallan las instrucciones para la implementación del sistema, la estructura del nodo de control y la configuración dinámica de los parámetros. Además, se analizan los resultados obtenidos mediante herramientas como *rqt_plot* y *rqt_graph*, concluyendo con una evaluación del desempeño del sistema.

**-Objetivos**

En está sección se espera que repasemos los conceptos vistos en la sesión pasada. La actividad consiste en:

- Implementar un controlador P, PI o PID en ROS2 para un motor de corriente continua simulado.
- Configurar los nodos de comunicación entre el controlador, el generador de set points y el motor.
- Modificar y lanzar archivos de configuración en ROS para ejecutar la simulación.
- Analizar el comportamiento del sistema utilizando herramientas de ROS como rqt_plot y rqt_graph.
- Ajustar los parámetros del controlador en tiempo de ejecución.

**-Introducción**

Semanalmente se nos presenta un reto por Manchester Robotics, haciendo uso de los programas como ROS y algunos lenguajes de programación.

Manchester Robotics se encarga de darnos conocimientos previos al reto, donde resuelve dudas para finalizarlo con éxito, además nos proporciona código y algunos slide de apoyo. Esta semana tocamos los siguientes temas:

- Namespaces: En ROS2 es una forma de agrupar nodos, tópicos, servicios y parámetros dentro de un espacio de nombres común. Esto permite organizar mejor los elementos en sistemas complejos y evitar conflictos de nombres entre diferentes nodos o paquetes.

- ❖ Parameters (parámetros): Son valores configurables asociados a un nodo que permiten ajustar su comportamiento sin necesidad de modificar el código. Estos parámetros pueden ser utilizados para definir constantes, opciones de configuración o ajustes que pueden cambiar en tiempo de ejecución. Se almacenan en un espacio de nombres dentro del nodo y pueden ser accedidos o modificados mediante la API de ROS2 o herramientas como `ros2 param`.

  Por ejemplo, un controlador PID tiene los siguientes parámetros:

- `Kp` (Ganancia proporcional)
- `Ki` (Ganancia integral)
- `Kd` (Ganancia derivativa) Estos valores pueden definirse en un archivo YAML o modificarse dinámicamente con herramientas como `rqt_reconfigure`.

En el ecosistema de ROS2, los servicios representan un mecanismo de comunicación sincrónica entre nodos. A diferencia de los tópicos, donde se publica y suscribe información de manera asíncrona y continua, un servicio funciona bajo un patrón request-response: un nodo cliente envía una petición a un nodo servidor, y este último procesa la solicitud y responde con un resultado.

Esta comunicación sincrónica es útil en situaciones donde se requiere realizar una acción puntual, esperar el resultado de esa acción o intercambiar datos de manera transaccional. Por ejemplo, encender o apagar un dispositivo, configurar un valor puntual, realizar una consulta de estado o, como en este caso, iniciar o detener la simulación de un sistema.

2. Estructura de un Servicio

Definición del archivo .srv:

Un servicio en ROS2 se describe con un archivo .srv, donde se definen los campos de entrada (request) y los campos de salida (response). En este proyecto se usó el servicio SetProcessBool.srv, que contiene un booleano enable en la parte de solicitud y, en la respuesta, un booleano success y un string message. Esto permite enviar un valor de habilitación (True/False) y obtener una confirmación junto con un mensaje descriptivo.

Servidor de Servicio (Service Server):

Es el nodo que implementa la lógica del servicio. Recibe las peticiones de los clientes, ejecuta la acción deseada y envía la respuesta. En el código, los nodos motor_sys y ctrl exponen el servicio EnableProcess para iniciar o detener sus respectivos procesos (simulación del motor o cálculo del controlador).

Cliente de Servicio (Service Client):

Es el nodo que solicita la acción al servidor. Envía los datos de entrada y queda a la espera de la respuesta. En este proyecto, el nodo sp_gen (SetPointPublisher) actúa como cliente: envía una solicitud a EnableProcess para habilitar o deshabilitar el motor cuando inicia.

3. Implementación de los Servicios en el Código

3.1. Archivo SetProcessBool.srv

bool enable

---

bool success

string message

enable indica la acción que se desea (iniciar o detener).

success confirma si la operación se realizó correctamente.

message proporciona un texto adicional (por ejemplo, “Simulation Started Successfully”).

Este archivo se ubica en la carpeta srv dentro del paquete custom_interfaces y se compila con colcon build, lo que genera las clases de Python para su uso en ROS2.

3.2. Nodo Servidor: motor_sys

En motor_sys, se crea un servidor de servicio con:

self.srv = self.create_service(SetProcessBool, 'EnableProcess',
self.simulation_service_callback)

SetProcessBool es el tipo de servicio definido.

'EnableProcess' es el nombre del servicio.

self.simulation_service_callback es la función que procesa la petición.

Cuando el cliente envía enable=True, la variable simulation_running se pone en True y el nodo empieza a actualizar el modelo dinámico en el timer_cb. Si recibe enable=False, se detiene la simulación.

3.3. Nodo Servidor (opcional): ctrl

De manera similar, en el nodo del controlador ctrl también se podría exponer un servicio "EnableProcess" para iniciar o detener la acción de control. Así se implementa la misma lógica: si enable=True, se activa el cálculo de la señal de control; si no, se detiene. Esto se hace para practicar la posibilidad de habilitar/deshabilitar distintos nodos de manera centralizada.

3.4. Nodo Cliente: sp_gen

En el nodo sp_gen (SetPointPublisher), se crea un cliente de servicio:

self.cli = self.create_client(SetProcessBool, 'EnableProcess')

y se espera a que esté disponible:

while not self.cli.wait_for_service(timeout_sec=1.0):

self.get_logger().info('Service not available, waiting again...')

Después, se construye la petición:

request = SetProcessBool.Request()

request.enable = True

Finalmente, se envía la solicitud de manera asíncrona con:

future = self.cli.call_async(request)

future.add_done_callback(self.response_callback)

La función response_callback procesa la respuesta, revisando si success es True y mostrando el mensaje de confirmación. De esta forma, sp_gen puede habilitar el motor al iniciar, o en cualquier otro momento, según la lógica deseada.

4. Ventajas y Escenarios de Uso

Control de Estado: Permite habilitar o deshabilitar subsistemas de forma ordenada. En el ejemplo, se “enciende” o “apaga” la simulación y el controlador.

Interacciones Puntuales: A diferencia de los tópicos, donde el flujo de datos es continuo, un servicio encaja mejor cuando se requiere una acción discreta (iniciar, detener, configurar un valor único, etc.).

Simplicidad: La comunicación request-response es intuitiva, pues el cliente sabe si su petición fue exitosa o no y obtiene un mensaje informativo.

Extensibilidad: Se pueden agregar más servicios con distintos tipos de solicitudes y respuestas (por ejemplo, servicios para modificar parámetros avanzados, reiniciar el sistema, etc.).

5. Ejecución y Prueba

Lanzar los nodos:

Se utiliza un archivo de lanzamiento (por ejemplo, challenge_launch.py) para iniciar los nodos motor_sys, ctrl y sp_gen.

Verificar Servicios Disponibles:

Con Ros2 service list, se puede comprobar que el servicio /motor_sys/EnableProcess está en ejecución.

Llamar al Servicio Manualmente:

En una terminal, se puede llamar al servicio con:

ros2 service call /motor_sys/EnableProcess custom_interfaces/srv/SetProcessBool "{enable: true}"

Si el nodo responde con success: true, el motor se activa. Luego, con enable: false, se detiene.

Respuesta en la Consola:

Se observa en los logs cómo motor_sys imprime “Simulation Started” o “Simulation Stopped”, y cómo el cliente sp_gen recibe el resultado.

6. Conclusión

Los servicios en ROS2 son un pilar importante para implementar interacciones sincrónicas entre nodos. En este proyecto, se aprovechó un archivo .srv personalizado (SetProcessBool.srv) para iniciar o detener la simulación del motor y, potencialmente, del controlador. La implementación incluyó un servidor en motor_sys (y opcionalmente en ctrl), y un cliente en sp_gen, que al arrancar solicita la activación del proceso.

Este enfoque de servicios resulta útil para separar responsabilidades: los nodos servidores controlan la lógica interna de habilitar/deshabilitar sus funciones, mientras que los nodos clientes o incluso la consola de ROS2 pueden disparar esas acciones de forma ordenada. De esta manera, se logra un diseño modular y escalable, alineado con la filosofía de ROS2 de facilitar la colaboración y la extensibilidad de los sistemas robóticos.

**-Solución del problema**

1. Descarga y configuración del paquete: Se descargó y compiló el paquete "motor_control".
2. Modificación del nodo de control: Se creó un nuevo nodo llamado "/ctrl" que suscribe a "/set_point" y "/motor_output_y", y publica en "/motor_input_u".
3. Configuración del lanzamiento: Se implementó un archivo de lanzamiento para inicializar todos los nodos necesarios.
4. Pruebas del sistema: Se utilizaron comandos de ROS2 para enviar mensajes y analizar la respuesta del sistema.
5. Ajuste del controlador: Se utilizó rqt_reconfigure para modificar los parámetros en tiempo de ejecución.
6. Visualización de datos: Se emplearon rqt_plot y rqt_graph para monitorear el desempeño del sistema.

**Nodo controller:**

```
#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

from std_msgs.msg import Float32

import numpy as np

from rcl_interfaces.msg import SetParametersResult

class Controller(Node):

  def _init_(self):

    super()._init_('ctrl')

    # Declarar parámetros del controlador
```

```
        self.declare_parameter('Kp', 0.2)
        self.declare_parameter('Ki', 0.5)
        self.declare_parameter('Kd', 0.0)
        self.declare_parameter('sample_time', 0.02)
        # Límite de saturación de la señal de control
        self.declare_parameter('control_sat', 5.0)
        # Activar o no el anti-windup
        self.declare_parameter('enable_antiwindup', True)
        # Cargar valores de parámetros
        self.Kp = self.get_parameter('Kp').value
        self.Ki = self.get_parameter('Ki').value
        self.Kd = self.get_parameter('Kd').value
        self.sample_time = self.get_parameter('sample_time').value
        self.control_sat = self.get_parameter('control_sat').value
        self.enable_antiwindup = self.get_parameter('enable_antiwindup').value
        # Variables internas
        self.previous_error = 0.0
        self.integral = 0.0
        self.current_set_point = 0.0
        self.current_output = 0.0
        # Publicador de la señal de control
        self.control_pub = self.create_publisher(Float32, 'motor_input_u', 10)
        # Suscriptores
        self.create_subscription(Float32, 'set_point', self.set_point_callback, 10)
        self.create_subscription(Float32, 'motor_output_y', self.motor_output_callback, 10)

        # Timer para ejecución periódica
        self.timer = self.create_timer(self.sample_time, self.timer_callback)
        # Callback para reconfiguración de parámetros en tiempo real
        self.add_on_set_parameters_callback(self.parameter_update_callback)
```

```
        self.get_logger().info("Controller node with saturation & anti-windup started.")

    def set_point_callback(self, msg: Float32):
        self.current_set_point = msg.data

    def motor_output_callback(self, msg: Float32):
        self.current_output = msg.data

    def timer_callback(self):
        # Calcular el error
        error = self.current_set_point - self.current_output
        # Parte integral
        self.integral += error * self.sample_time
        # Derivada
        derivative = (error - self.previous_error) / self.sample_time
        # Ley PID ideal
        control_signal = self.Kp * error + self.Ki * self.integral + self.Kd * derivative
        # Aplicar saturación
        if control_signal > self.control_sat:
            control_saturated = self.control_sat
            # Anti-windup
            if self.enable_antiwindup:
                # Revertir la acumulación integral
                self.integral -= error * self.sample_time
        elif control_signal < -self.control_sat:
            control_saturated = -self.control_sat
            # Anti-windup
            if self.enable_antiwindup:
                self.integral -= error * self.sample_time
        else:
            control_saturated = control_signal
        # Guardar error actual
        self.previous_error = error
```

```
    # Publicar la señal de control
    msg_out = Float32()
    msg_out.data = control_saturated
    self.control_pub.publish(msg_out)

def parameter_update_callback(self, params):
    """Callback para actualizar parámetros con rqt_reconfigure."""
    result = SetParametersResult()
    result.successful = True
    for param in params:
        if param.name == "Kp":
            if param.value < 0.0:
                self.get_logger().warn("Kp must be >= 0")
                result.successful = False
                result.reason = "Invalid Kp"
            else:
                self.Kp = param.value
        elif param.name == "Ki":
            if param.value < 0.0:
                self.get_logger().warn("Ki must be >= 0")
                result.successful = False
                result.reason = "Invalid Ki"
            else:
                self.Ki = param.value
        elif param.name == "Kd":
            if param.value < 0.0:
                self.get_logger().warn("Kd must be >= 0")
                result.successful = False
                result.reason = "Invalid Kd"
            else:
                self.Kd = param.value
```

```
        elif param.name == "sample_time":
            if param.value <= 0.0:
                self.get_logger().warn("sample_time must be > 0")
                result.successful = False
                result.reason = "Invalid sample_time"
            else:
                self.sample_time = param.value
                self.timer.cancel()
                self.timer = self.create_timer(self.sample_time, self.timer_callback)
        elif param.name == "control_sat":
            if param.value <= 0.0:
                self.get_logger().warn("control_sat must be > 0")
                result.successful = False
                result.reason = "Invalid control_sat"
            else:
                self.control_sat = param.value
        elif param.name == "enable_antiwindup":
            self.enable_antiwindup = bool(param.value)
    return result
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()
```

Este código es un controlador PID implementado en ROS 2 que regula una señal de control para un motor. Su objetivo es recibir un setpoint (valor deseado), medir la salida del motor y ajustar la señal de control para minimizar el error.

Estamos usando un controlador PID, pero en ambos se usó un PI

1. **Definición del nodo (“Controller”)**

El nodo se llama “ctrl” y hereda de “Node” en ROS 2.

2. **Parámetros del controlador PID**

Se definen y cargan parámetros como:

- Ganancias PID: "Kp", "Ki", "Kd"

- Tiempo de muestreo ("sample_time")

- Saturación de la señal de control ("control_sat")

3. **Publicador y Suscriptores**

- Publicador: "motor_input_u" (envía la señal de control)

- Suscriptores: "set_point": Recibe el valor deseado / "motor_output_y": Recibe la salida del motor

4. **Bucle de control ("timer_callback")**

Este se ejecuta cada "sample_time" y realiza los cálculos del PID:

1. Calcula el error: Diferencia entre "set_point" y "motor_output_y".
2. Calcula el término integral: Suma acumulada del error.
3. Calcula el término derivativo: Cambio del error en el tiempo.
4. Genera la señal de control: $u = Kp \cdot e + Ki \cdot \int edt + Kd \cdot \frac{de}{dt}$
5. Aplica saturación: Si el valor supera "control_sat", lo limita.
6. Aplica anti-windup: Corrige la acumulación del error si se activa.

7. Publica la señal de control.

5. **Ajuste dinámico de parámetros ("parameter_update_callback")**

Permite modificar parámetros en tiempo real con "rqt_reconfigure".

6. **Función principal ("main")**

Inicializa el nodo y ejecuta "rclpy.spin(node)" hasta que se detenga.

**Nodo motor:**

```
# Imports
import rclpy  # Biblioteca principal de ROS 2
from rclpy.node import Node  # Clase base para nodos en ROS 2
from std_msgs.msg import Float32  # Tipo de mensaje estándar para números flotantes
from rcl_interfaces.msg import SetParametersResult  # Mensaje para actualización de
parámetros
from custom_interfaces.srv import SetProcessBool  # Servicio personalizado para
activar/desactivar simulación

# Definición de la clase DCMotor
class DCMotor(Node):
   def _init_(self):
       super().__init__('motor_sys')  # Inicializa el nodo con el nombre
'motor_sys'

       # Declaración de parámetros configurables
       self.declare_parameter('sample_time', 0.02)  # Tiempo de muestreo en
segundos
       self.declare_parameter('sys_gain_K', 1.75)  # Ganancia del sistema K
       self.declare_parameter('sys_tau_T', 0.5)  # Constante de tiempo Tau
       self.declare_parameter('initial_conditions', 0.0)  # Condiciones iniciales

       # Carga de parámetros en variables internas
       self.sample_time = self.get_parameter('sample_time').value
       self.param_K = self.get_parameter('sys_gain_K').value
       self.param_T = self.get_parameter('sys_tau_T').value
       self.initial_conditions = self.get_parameter('initial_conditions').value

       # Inicialización del mensaje de salida
       self.motor_output_msg = Float32()

       # Variables internas del sistema
       self.input_u = 0.0  # Entrada del motor
```

```
        self.output_y = self.initial_conditions  # Salida del motor con su condición inicial
        self.simulation_running = False  # Estado de la simulación

        # Creación de publicadores y suscriptores
        self.motor_input_sub = self.create_subscription(Float32, 'motor_input_u', self.input_callback, 10)  # Suscriptor para la señal de entrada
        self.motor_speed_pub = self.create_publisher(Float32, 'motor_output_y', 10) # Publicador de la señal de salida

        # Creación del temporizador para la simulación
        self.timer = self.create_timer(self.sample_time, self.timer_cb)  # Se ejecuta cada sample_time segundos

        # Configuración del callback para la actualización de parámetros en tiempo real
        self.add_on_set_parameters_callback(self.parameters_callback)

        # Creación del servicio para activar/desactivar la simulación
        self.srv = self.create_service(SetProcessBool, 'EnableProcess', self.simulation_service_callback)

        # Mensaje de inicio en la terminal
        self.get_logger().info('Dynamical System Node Started \U0001F680')

    # Callback del temporizador
    def timer_cb(self):
        if not self.simulation_running:
            return  # Si la simulación está detenida, no hace nada

        # Simulación del motor de corriente continua usando la ecuación diferencial discreta
        # Ecuación: y[k+1] = y[k] + ((-1/τ) y[k] + (K/τ) u[k]) T_s
        self.output_y += (-1.0 / self.param_T * self.output_y + self.param_K / self.param_T * self.input_u) * self.sample_time

        # Publicación del resultado
        self.motor_output_msg.data = self.output_y
        self.motor_speed_pub.publish(self.motor_output_msg)

    # Callback del suscriptor para recibir la entrada del motor
    def input_callback(self, input_sgn):
        self.input_u = input_sgn.data  # Almacena la entrada recibida

    # Callback del servicio para iniciar/detener la simulación
```

```
    def simulation_service_callback(self, request, response):
        if request.enable:
            self.simulation_running = True
            self.get_logger().info("🚀 Simulation Started")
            response.success = True
            response.message = "Simulation Started Successfully"
        else:
            self.simulation_running = False
            self.get_logger().info("🛑 Simulation Stopped")
            response.success = True
            response.message = "Simulation Stopped Successfully"

        return response

    # Callback para la actualización dinámica de parámetros
    def parameters_callback(self, params):
        for param in params:
            # Validación de la ganancia del sistema
            if param.name == "sys_gain_K":
                if param.value < 0.0:
                    self.get_logger().warn("Invalid sys_gain_K! It cannot be
negative.")
                    return SetParametersResult(successful=False, reason="sys_gain_K
cannot be negative")
                else:
                    self.param_K = param.value  # Actualiza la variable interna
                    self.get_logger().info(f"sys_gain_K updated to {self.param_K}")

            # Validación de la constante de tiempo Tau
            if param.name == "sys_tau_T":
                if param.value < 0.0:
                    self.get_logger().warn("Invalid sys_tau_T! It cannot be
negative.")
                    return SetParametersResult(successful=False, reason="sys_tau_T
cannot be negative")
                else:
                    self.param_T = param.value  # Actualiza la variable interna
                    self.get_logger().info(f"sys_tau_T updated to {self.param_T}")

        return SetParametersResult(successful=True)


# Función principal para ejecutar el nodo
def main(args=None):
    rclpy.init(args=args)  # Inicializa ROS 2
```

```
    node = DCMotor()  # Crea una instancia de la clase DCMotor

    try:
        rclpy.spin(node)  # Mantiene el nodo en ejecución
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()  # Destruye el nodo al finalizar
        rclpy.try_shutdown()  # Cierra ROS 2 correctamente

# Punto de entrada del programa
if _name_ == '_main_':
    main()
```

Este código implementa un nodo en ROS 2 que simula el comportamiento dinámico de un motor de corriente continua.

**1. Parámetros:** Define parámetros como el tiempo de muestreo, la ganancia del sistema y la constante de tiempo.

**2. Publicadores y Suscriptores:**

- Recibe la señal de entrada del motor a través del tópico "motor_input_u".
- Publica la velocidad del motor en "motor_output_y".

**3. Simulación:**

- Usa una ecuación diferencial discreta para calcular la velocidad del motor en función de la entrada.
- Se ejecuta en un temporizador con un período definido ("sample_time").

**4. Servicio:** Permite iniciar o detener la simulación mediante el servicio "EnableProcess".

**5. Parámetros Dinámicos:** Permite modificar en tiempo real la ganancia y la constante de tiempo mediante "rqt_reconfigure".

En general, este nodo simula el comportamiento de un motor DC y permite controlarlo y monitorearlo dentro de un entorno ROS 2.

**Nodo Setpoint:**

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
import numpy as np
from std_msgs.msg import Float32
from custom_interfaces.srv import SetProcessBool
from rcl_interfaces.msg import SetParametersResult
```

```
class SetPointPublisher(Node):
   def _init_(self):
       super()._init_('sp_gen')

       # Declarar parámetros para configurar la señal generada
       self.declare_parameter('amplitude', 2.0)  # Amplitud de la señal
       self.declare_parameter('omega', 1.0)  # Frecuencia en Hz
       self.declare_parameter('signal_type', 'sin')  # Tipo de señal (sin, square,
triangle, sawtooth)

       # Obtener valores iniciales de los parámetros
       self.amplitude = self.get_parameter('amplitude').value
       self.freq = self.get_parameter('omega').value
       self.signal_type = self.get_parameter('signal_type').value

       # Publicador en el tópico "set_point" para enviar la señal generada
       self.signal_publisher = self.create_publisher(Float32, 'set_point', 10)

       # Crear un temporizador para generar la señal periódicamente (cada 0.1 s)
       self.timer_period = 0.1  # Segundos
       self.timer = self.create_timer(self.timer_period, self.timer_cb)

       # Cliente del servicio para habilitar o deshabilitar el proceso del sistema
       self.cli = self.create_client(SetProcessBool, 'EnableProcess')
       while not self.cli.wait_for_service(timeout_sec=1.0):
           self.get_logger().info('Service not available, waiting again...')

       # Variables para el mensaje de la señal y control de estado
       self.signal_msg = Float32()
       self.start_time = self.get_clock().now()  # Guardar el tiempo inicial
       self.system_running = False  # Estado del sistema

       # Callback para actualización dinámica de parámetros
       self.add_on_set_parameters_callback(self.parameter_callback)

       self.get_logger().info("SetPoint Node Started \U0001F680")

       # Enviar una solicitud al servicio para iniciar la simulación
automáticamente
       self.send_request(True)

   def timer_cb(self):
       if not self.system_running:
           return  # Si la simulación no está en marcha, salir
```

```
        # Calcular el tiempo transcurrido en segundos
        elapsed_time = (self.get_clock().now() - self.start_time).nanoseconds / 1e9

        # Ángulo de la señal (fase) en radianes
        theta = 2.0 * np.pi * self.freq * elapsed_time

        # Generación de la señal según el tipo seleccionado
        if self.signal_type == 'sin':
            self.signal_msg.data = self.amplitude * np.sin(theta)
        elif self.signal_type == 'square':
            self.signal_msg.data = self.amplitude * np.sign(np.sin(theta))
        elif self.signal_type == 'triangle':
            self.signal_msg.data = self.amplitude * (2.0 / np.pi) *
np.arcsin(np.sin(theta))
        elif self.signal_type == 'sawtooth':
            saw_val = 2.0 * ((self.freq * elapsed_time) - np.floor(0.5 + self.freq *
elapsed_time))
            self.signal_msg.data = self.amplitude * saw_val
        else:
            self.get_logger().warn("Unknown signal type, defaulting to 'sin'")
            self.signal_msg.data = self.amplitude * np.sin(theta)

        # Publicar la señal generada en el tópico "set_point"
        self.signal_publisher.publish(self.signal_msg)

    def send_request(self, enable: bool):
        """ Envía una solicitud al servicio EnableProcess para activar o detener la
simulación. """
        request = SetProcessBool.Request()
        request.enable = enable

        future = self.cli.call_async(request)
        future.add_done_callback(self.response_callback)

    def response_callback(self, future):
        """ Maneja la respuesta del servicio EnableProcess. """
        try:
            response = future.result()
            if response.success:
                self.system_running = True
                self.get_logger().info(f"Success: {response.message}")
            else:
                self.system_running = False
                self.get_logger().warn(f"Failure: {response.message}")
```

```
        except Exception as e:
            self.system_running = False
            self.get_logger().error(f"Service call failed: {e}")

    def parameter_callback(self, params):
        """ Callback para actualizar parámetros en tiempo de ejecución. """
        successful = True
        reason = ''

        for param in params:
            if param.name == 'amplitude':
                if param.value < 0.0:
                    successful = False
                    reason = "Amplitude cannot be negative"
                    self.get_logger().warn(reason)
                else:
                    self.amplitude = param.value
                    self.get_logger().info(f"Amplitude updated to {self.amplitude}")

            elif param.name == 'omega':
                if param.value < 0.0:
                    successful = False
                    reason = "omega (freq in Hz) cannot be negative"
                    self.get_logger().warn(reason)
                else:
                    self.freq = param.value
                    self.get_logger().info(f"Frequency updated to {self.freq} Hz")

            elif param.name == 'signal_type':
                if param.value not in ['sin', 'square', 'triangle', 'sawtooth']:
                    successful = False
                    reason = "signal_type must be 'sin', 'square', 'triangle' or
'sawtooth'"
                    self.get_logger().warn(reason)
                else:
                    self.signal_type = param.value
                    self.get_logger().info(f"signal_type updated to
{self.signal_type}")

        result = SetParametersResult()
        result.successful = successful
        result.reason = reason
        return result

def main(args=None):
```

```
    rclpy.init(args=args)
    node = SetPointPublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

if _name_ == '_main_':
    main()
```

Este código implementa un nodo en ROS 2 que genera señales de referencia (Set Point) para un sistema de control.

**1. Parámetros:**

- Define la amplitud, la frecuencia y el tipo de señal (seno, cuadrada, triangular o diente de sierra).
- Permite la actualización dinámica de parámetros mediante "rqt_reconfigure".

**2. Publicador:**

- Publica la señal generada en el tópico "set_point" cada 0.1 segundos.

**3. Generación de Señales:**

- Calcula valores de señales periódicas (seno, cuadrada, triangular y diente de sierra) usando funciones matemáticas de "numpy".

**4. Cliente de Servicio:**

- Se conecta al servicio "EnableProcess" para habilitar o detener la simulación del sistema.
- Envía una solicitud automática al inicio para arrancar el sistema.

**5. Manejo de Parámetros en Tiempo Real:**

- Modifica la amplitud, frecuencia y tipo de señal en tiempo de ejecución con validaciones de entrada.

En general, este nodo sirve como generador de señales de prueba para un sistema de control en ROS 2.

**Challenge launch:**

```
import os
from ament_index_python.packages import get_package_share_directory

# Importa las clases necesarias para la creación del lanzamiento
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    # Definición de los nombres de los nodos
    # - ctrl (controlador)
    # - motor_sys (sistema del motor)
    # - sp_gen (generador de setpoint)

    # Obtener las direcciones de los archivos YAML con los parámetros de
configuración
    controller_yaml = os.path.join(
        get_package_share_directory('motor_control'),
        'config',
        'controller_params.yaml'
    )

    motor_yaml = os.path.join(
        get_package_share_directory('motor_control'),
        'config',
        'dc_motor_params.yaml'
    )

    setpoint_yaml = os.path.join(
        get_package_share_directory('motor_control'),
        'config',
        'set_point_params.yaml'
    )

    # Definición de los nodos para el Grupo 1
    motor_node_1 = Node(name="motor_sys_1",
                        package='motor_control',
                        executable='dc_motor',
                        emulate_tty=True,
                        output='screen',
```

```
                    namespace="group1",
                    parameters=[motor_yaml]
                    )

sp_node_1 = Node(name="sp_gen_1",
                 package='motor_control',
                 executable='set_point',
                 emulate_tty=True,
                 output='screen',
                 namespace="group1",
                 parameters=[setpoint_yaml]
                 )

controller_node_1 = Node(name="ctrl_1",
                         package='motor_control',
                         executable='ctrl',
                         emulate_tty=True,
                         output='screen',
                         namespace="group1",
                         parameters=[controller_yaml]
                         )

# Definición de los nodos para el Grupo 2
motor_node_2 = Node(name="motor_sys_2",
                    package='motor_control',
                    executable='dc_motor',
                    emulate_tty=True,
                    output='screen',
                    namespace="group2",
                    parameters=[motor_yaml]
                    )

sp_node_2 = Node(name="sp_gen_2",
                 package='motor_control',
                 executable='set_point',
                 emulate_tty=True,
                 output='screen',
                 namespace="group2",
                 parameters=[setpoint_yaml]
                 )

controller_node_2 = Node(name="ctrl_2",
                         package='motor_control',
                         executable='ctrl',
                         emulate_tty=True,
```

```
                            output='screen',
                            namespace="group2",
                            parameters=[controller_yaml]
                            )

# Definición de los nodos para el Grupo 3
motor_node_3 = Node(name="motor_sys_3",
                    package='motor_control',
                    executable='dc_motor',
                    emulate_tty=True,
                    output='screen',
                    namespace="group3",
                    parameters=[motor_yaml]
                    )

sp_node_3 = Node(name="sp_gen_3",
                 package='motor_control',
                 executable='set_point',
                 emulate_tty=True,
                 output='screen',
                 namespace="group3",
                 parameters=[setpoint_yaml]
                 )

controller_node_3 = Node(name="ctrl_3",
                         package='motor_control',
                         executable='ctrl',
                         emulate_tty=True,
                         output='screen',
                         namespace="group3",
                         parameters=[controller_yaml]
                         )

# Crear la descripción del lanzamiento con todos los nodos definidos
l_d = LaunchDescription([
    motor_node_1, sp_node_1, controller_node_1,
    motor_node_2, sp_node_2, controller_node_2,
    motor_node_3, sp_node_3, controller_node_3
])

return l_d
```

Este código es un script de lanzamiento en ROS 2, que se encarga de inicializar y configurar múltiples nodos relacionados con el control de un motor de corriente continua.

El script define y lanza tres grupos de nodos (group1, group2, group3), donde cada grupo incluye:

1. "motor_sys": Simula el comportamiento del motor de corriente continua.
2. "sp_gen": Genera una señal de referencia o setpoint para el sistema.
3. "ctrl": Controlador que regula el comportamiento del motor en función del setpoint.

**1. Carga de archivos YAML de configuración**

- Se obtienen las rutas de los archivos YAML ("controller_params.yaml", "dc_motor_params.yaml", "set_point_params.yaml") dentro del paquete "motor_control".
- Estos archivos contienen los parámetros necesarios para configurar cada nodo.

**2. Creación de nodos para cada grupo**

- Se definen los nodos del motor, el setpoint y el controlador para tres grupos diferentes ("group1", "group2", "group3").
- Cada nodo es instanciado con el paquete "motor_control", asignándole un nombre único y los parámetros correspondientes.

**3. Agrupación de nodos y ejecución**

- Todos los nodos se agregan a un objeto "LaunchDescription", que se devuelve para que ROS 2 pueda ejecutarlos de manera simultánea.

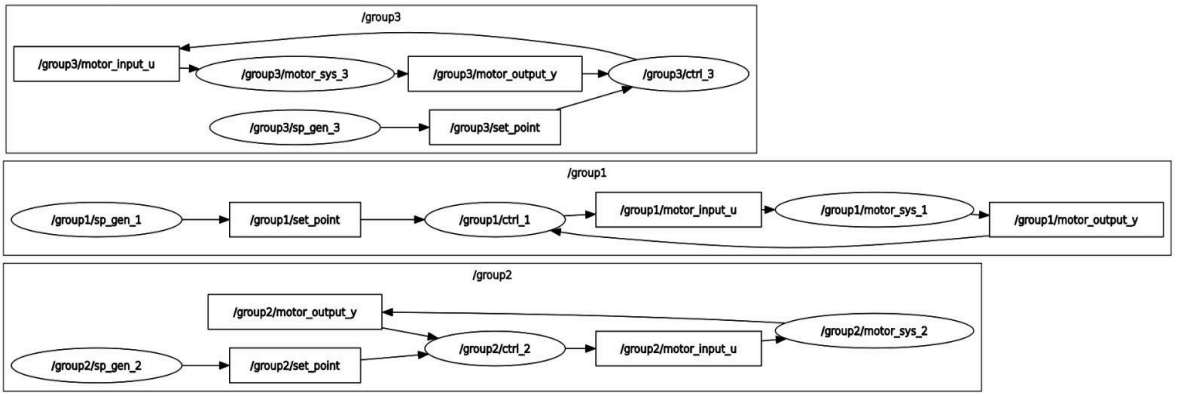
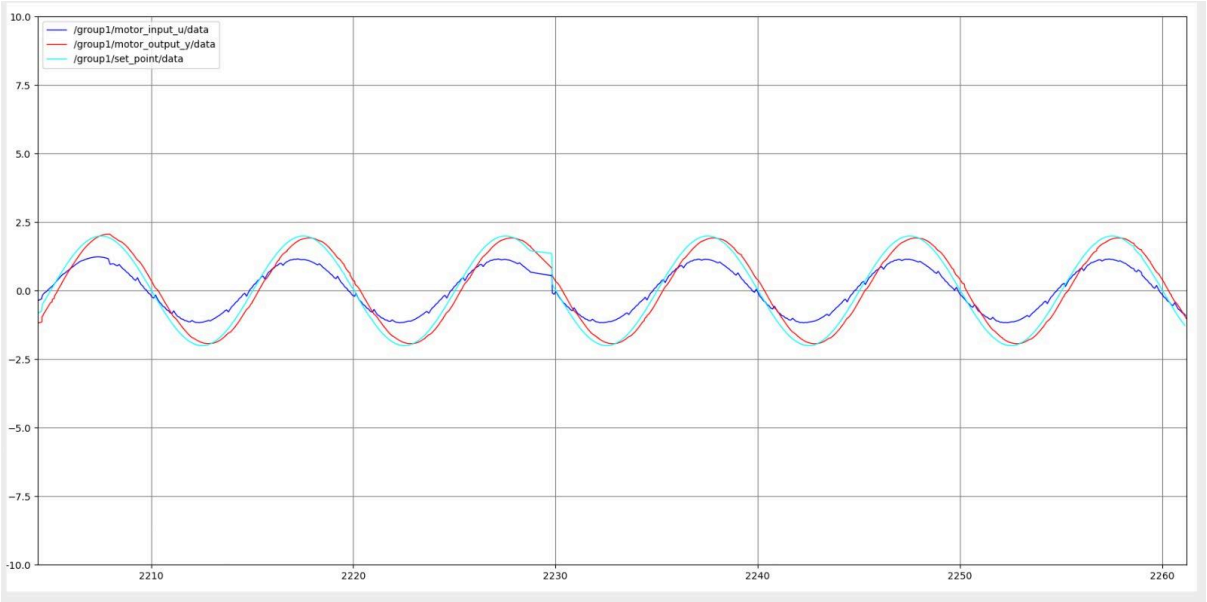
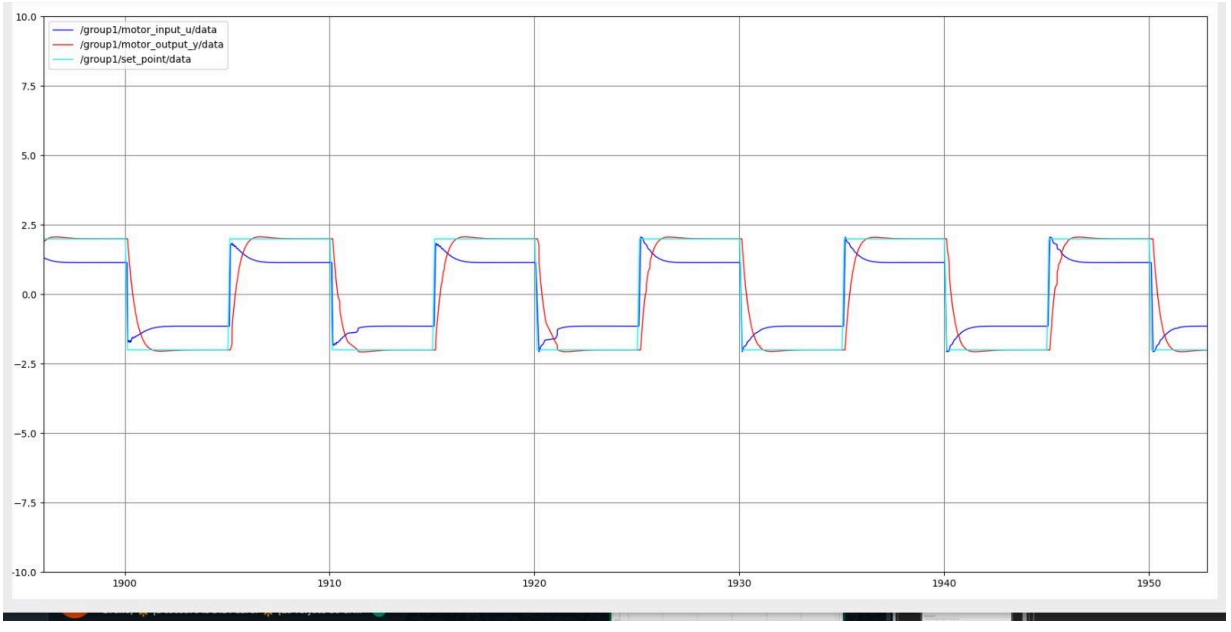



Diagrama de bloques que muestra la estructura y flujo de señales en un sistema con tres grupos de control de motores. Cada grupo incluye un generador de setpoint, un controlador y un motor con retroalimentación, formando sistemas de control en lazo cerrado.

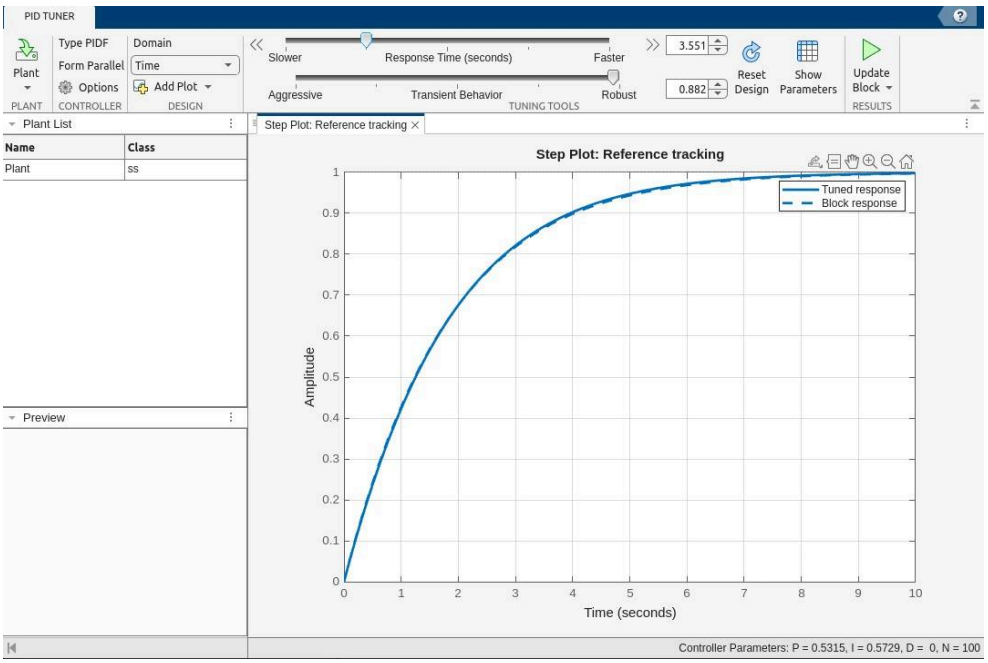



Resultados de señal senoidal



Resultados de señal cuadrada


Para los resultados hicimos uso de matlab ajustando los valores del PID  Realizar ajustes de Kp, Ki, Kd, para mejorar el desempeño, simultáneamente lo hicimos de manera empírica.




## -Conclusiones

- Se logró implementar un controlador funcional en ROS2 para el motor de corriente continua simulado.
- La estructura de publicación y suscripción de ROS2 permitió la comunicación efectiva entre los nodos del sistema.
- El uso de rqt_plot y rqt_graph facilitó el análisis del comportamiento del sistema.
- La configuración en tiempo de ejecución de los parámetros permitió mejorar la respuesta del sistema y probar diferentes estrategias de control.
- Se reforzaron los conocimientos sobre el desarrollo de nodos en ROS2 y la implementación de sistemas de control en entornos simulados.

## -Referencias


➔ *¿Qué es un controlador PID?* (2024, 19 junio). Soluciones de Adquisición de Datos (DAQ). https://dewesoft.com/es/blog/que-es-un-controlador-pid

➔ Ramachandran, R. (2025, 14 febrero). Importance of Domain ID and Namespace in ROS 2 for Multi-Robot Systems | Robotair. *Medium*.

http://blog.robotair.io/domain-id-and-namespace-in-ros-2-for-multi-robot-systems-9a939ae3fa40

➔ ManchesterRoboticsLtd. (s. f.). *TE3001B_Intelligent_Robotics_Implementation_2025/Week 2/Presentations/PDF/MCR2_ROS_Practicalities_V2_watermark.pdf at main · ManchesterRoboticsLtd/TE3001B_Intelligent_Robotics_Implementation_2025*. GitHub. https://github.com/ManchesterRoboticsLtd/TE3001B_Intelligent_Robotics_Implementation_2025/blob/main/Week%202/Presentations/PDF/MCR2_ROS_Practicalities_V2_watermark.pdf

➔ *Parameter API design in ROS*. (s. f.). https://design.ros2.org/articles/ros_parameters.html