Estudiantes:
Axel Javier Rosas Rodríguez | A01738607
Elian Cantalapiedra Sabugal | A01738462
Edwin Emmanuel Salazar Meza | A01738380

Profesor:
Rigoberto Cerino Jiménez

Nombre de la Institución:
Instituto Tecnológico y de Estudios Superiores de Tecnológico de Monterrey

Materia:
Fundamentación de la Robótica

Fecha:
23 de Febrero de 2026

Título del trabajo:
Reto semanal 2. Manchester Robotics

Resumen

En este trabajo se desarrolló e implementó un sistema de control en lazo cerrado para un motor de corriente directa (DC) utilizando el framework ROS 2. El reto consistió en modelar el comportamiento dinámico del motor, generar una señal de referencia y diseñar un controlador PID discreto que permitiera el seguimiento adecuado de dicha referencia. Organizamos la arquitectura del sistema mediante nodos independientes los cuales se comunican a través de tópicos, permitiendo una separación clara entre la generación del set point, el control y la simulación de la planta.

Finalmente, se evaluó el funcionamiento del sistema mediante herramientas de visualización de ROS 2, analizando la respuesta del motor frente a una referencia senoidal. Los resultados obtenidos permiten validar el correcto funcionamiento del controlador y la interacción entre los distintos componentes del sistema.

Objetivos

Objetivo General:

- Desarrollar e implementar un sistema de control en lazo cerrado para un motor de corriente directa (DC) utilizando ROS 2, integrando un modelo dinámico del motor, la generación de una señal de referencia y un controlador PID discreto que permita el seguimiento adecuado del set point.

Objetivos específicos:

- Modelar y discretizar un motor DC como nodo ROS 2.
- Diseñar e implementar un nodo generador de referencia senoidal configurable.
- Implementar un controlador PID discreto para calcular la señal de control.
- Comunicar los nodos vía tópicos de ROS 2.
- Ajustar y analizar parámetros PID para estabilidad y seguimiento de referencia.
- Evaluar el desempeño del sistema con herramientas de visualización.

Introducción

El control automático de sistemas dinámicos es un área fundamental en la ingeniería, particularmente en aplicaciones relacionadas con robótica y sistemas mecatrónicos. Los motores de corriente directa (DC) son ampliamente utilizados debido a su simplicidad, facilidad de control y bajo costo, por lo que resultan un caso de estudio ideal para el diseño y análisis de controladores.

En este minireto se aborda la implementación de un sistema de control para un motor DC empleando ROS 2 como plataforma de desarrollo. ROS 2 permite estructurar aplicaciones robóticas de manera modular mediante nodos que se comunican a través de tópicos, servicios y parámetros. En este contexto, los namespaces facilitan la organización de los recursos del sistema, mientras que los parameters permiten ajustar dinámicamente valores como ganancias del controlador o constantes del modelo sin modificar el código fuente.

El sistema desarrollado utiliza mensajes estándar para la comunicación entre nodos y se apoya en un controlador PID discreto para regular la velocidad del motor. Dicho controlador calcula la señal de control a partir del error entre la referencia y la salida del sistema, incorporando acciones proporcional e integral para mejorar el seguimiento de la señal deseada. La sintonización del controlador se realizó ajustando las ganancias con base en el comportamiento observado del sistema, buscando estabilidad y reducción del error en estado estacionario.

Metodología

1. Preparación de nodos base
Revisar los nodos proporcionados por MCR2:

- /motor_sys → Nodo del motor DC simulado.
- /sp_gen → Nodo generador de señales.

Verificar que ambos nodos funcionen correctamente y se comuniquen entre sí.

2. Crear el nodo de control /ctrl
Crear un nuevo nodo llamado controller o /ctrl.

Suscribirse a:
- /set_point → Entrada deseada.
- /motor_speed_y → Salida del motor.

Publicar en:
- /motor_input_u → Señal de control hacia el motor.

Implementar el controlador (P, PI o PID).

Realizar el ajuste inicial de PID (tuning empírico por pasos):
- Comenzar con todos los parámetros en 0, excepto kp.
- Ajustar primero kp hasta obtener una respuesta estable y razonable.
- Luego introducir ki progresivamente para eliminar el error en estado estacionario.
- Finalmente ajustar kd para mejorar la respuesta transitoria y reducir oscilaciones.

3. Configuración del generador de set point
Modificar /sp_gen para permitir diferentes tipos de señales:
- Sinusoidal
- Cuadrada
- Triangular

Definir un parámetro que determine el tipo de señal y que sea modificable en tiempo real.

4. Creación y modificación del Launch file
Crear un launch file (motor_launch.py) que incluya:

- Nodo /motor_sys.
- Nodo /sp_gen.
- Nodo /ctrl.

Configurar los parámetros de los tres nodos desde el launch file.

Implementar 3 grupos independientes usando namespaces:
- group1 señal sinusoidal
- group2: señal cuadrada
- group3: señal triangular

Cada grupo tiene su propia instancia de /motor_sys, /sp_gen y /ctrl.

Verificar que todos los nodos se ejecuten continuamente al lanzar el launch

5. Visualización y ajuste

Usar rqt_graph para verificar conexiones entre nodos y tópicos.

Usar rqt_reconfigure para modificar parámetros en tiempo real.

Usar rqt_plot para graficar:
- /motor_output_y
- /set_point.
- motor_input_u.

Analizar la respuesta del sistema en cada grupo y ajustar kp, ki, kd de ser necesario.

Solución del problema

Nodo sp_gen

```python
#Class Definition
class SetPointPublisher(Node):
    def __init__(self):
        super().__init__('set_point_node')

        self.declare_parameter('signal_type')
        self.declare_parameter('amplitude')
        self.declare_parameter('omega')

        # Retrieve sine wave parameters
        self.amplitude = 2.0
        self.omega  = 1.0
        self.timer_period = 0.1 #seconds

        #Create a publisher and timer for the signal
        self.signal_publisher = self.create_publisher(Float32, 'set_point', 10)
        self.timer = self.create_timer(self.timer_period, self.timer_cb)

        #Create a messages and variables to be used
        self.signal_msg = Float32()
        self.start_time = self.get_clock().now()

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

Partiendo del nodo sp_gen proporcionado por MCR2, se declaran los parámetros que se pueden actualizar en tiempo real (signal_type, amplitude y omega).

Esto permite definir en un archivo Launch, los parámetros de una señal específica para cada grupo y verificar la eficiencia del control.

```python
def timer_cb(self):
    # Get the signal type parameter
    self.omega = self.get_parameter('omega').get_parameter_value().double_value
    self.amplitude = self.get_parameter('amplitude').get_parameter_value().double_value
    signal_type = self.get_parameter('signal_type').get_parameter_value().string_value
    elapsed_time = (self.get_clock().now() - self.start_time).nanoseconds / 1e9
```

Se define una función callback ejecutada cada 0.1 segundos, lee los parámetros actuales y calcula el tiempo transcurrido desde el inicio del nodo. Esto permite ajustar la amplitud, frecuencia y tipo de señal que se busca representar en tiempo real.

```python
# Generate selected signal
if signal_type == 'sine':
    self.signal_msg.data = self.amplitude * np.sin(self.omega * elapsed_time)
elif signal_type == 'square':
    self.signal_msg.data = self.amplitude * (1.0 if np.sin(self.omega * elapsed_time) >= 0 else -1.0)
elif signal_type == 'triangle':
    self.signal_msg.data = self.amplitude * (2 * abs((elapsed_time / (2 * np.pi)) % 1 - 0.5) - 0.5)
else:
    self.get_logger().warn(f"Unknown signal type '{signal_type}'. Defaulting to sine wave.")
    self.signal_msg.data = self.amplitude * np.sin(self.omega * elapsed_time)

# Publish the signal
self.signal_publisher.publish(self.signal_msg)
```

Dentro de la misma función callback, se calcula el valor actual de la señal según el tipo de señal (sine, square, triangle) y lo publica en el topic /set_point.

Nodo motor_sys

```python
# Declare parameters
# System sample time in seconds
self.declare_parameter('sample_time', 0.01)
# System gain K
self.declare_parameter('sys_gain_K', 1.75)
# System time constant Tau
self.declare_parameter('sys_tau_T', 0.5)
# System initial conditions
self.declare_parameter('initial_conditions', 0.0)

# DC Motor Parameters
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').
```

Se definen los parámetros que describen el motor: tiempo de muestreo, ganancia K, constante de tiempo τ y condiciones iniciales.

Donde K y τ son esenciales para modelar cómo responde el motor a una entrada de control.

Necesitamos guardar el estado del motor (y) y la última entrada (u) para calcular el siguiente paso de la simulación.

```python
#Set variables to be used
self.input_u = 0.0
self.output_y = self.initial_conditions
```

```python
#Timer Callback
def timer_cb(self):
    #DC Motor Simulation
    #DC Motor Equation 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.para
    #Publish the result
    self.motor_output_msg.data = self.output_y
    self.motor_speed_pub.publish(self.motor_output_msg)

#Subscriber Callback
def input_callback(self, input_sgn):
    self.input_u = input_sgn.data
```

Por último se simula el motor DC de primer orden con dinámica realista.

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

La integración discreta permite que la simulación evolucione paso a paso.

Nodo ctrl

Se crea un nodo ROS2 llamado 'controller', declaran los parámetros del PID (kp, ki, kd) y el tiempo de muestreo Ts, los cuales podrán modificarse desde un launch file o rqt reconfigure en tiempo real.

Lo que permite al controlador ser configurable sin tocar el código.

```python
class Controller(Node):
    def __init__(self):
        super().__init__('controller')

        #Parametros
        self.declare_parameter('kp', 0.82)
        self.declare_parameter('ki', 23.75)
        self.declare_parameter('kd', 0.0)
        self.declare_parameter('Ts', 0.01) #Tiempo de muestreo

        #Variables de estado
        self.motor = 0.0 #Estado actual del motor (velocidad)
        self.ref = 0.0 #Referencia actual del set point
        self.error_prev = 0.0
        self.integral = 0.0
```

```python
#Callbacks
def setpoint_callback(self, msg):
    self.ref = msg.data

def output_callback(self, msg):
    self.motor = msg.data
```

Se crean dos callback: setpoint_callback actualiza la referencia actual (ref) y output_callback actualiza la velocidad del motor (motor).

Lo que mantiene la información en tiempo real para que el PID calcule el error.

Se crea otra función que lee los parámetros PID actuales.

Calcula el error entre la eferencia y la velocidad del motor.

Calcula la parte integral (self.integral) y la derivada (derivative).

Publica la señal de control en /motor_input_u.

Calcula la señal de control u usando la ecuación PID discreta:

```python
#PID
def control_loop(self):
    kp = self.get_parameter('kp').value
    ki = self.get_parameter('ki').value
    kd = self.get_parameter('kd').value
    Ts = self.get_parameter('Ts').value

    #Diferencia entre la referencia y la velocidad del motor
    error = self.ref - self.motor

    self.integral += error * Ts
    derivative = (error - self.error_prev) / Ts

    #Calculo de la señal de control
    u = kp * error + ki * self.integral + kd * derivative

    #Publicar la entrada de control
    msg = Float32()
    msg.data = u
    self.motor_input_pub.publish(msg)

    #Actualizar variables de estado
    self.error_prev = error
```

$$u[k] = K_p e[k] + K_i \sum e[k] T_s + K_d \frac{e[k] - e[k-1]}{T_s}$$

Archivo de Lanzamiento

```python
def create_group(group_name, signal_type):
    return GroupAction([
        PushRosNamespace(group_name),
```

Se crea un grupo de nodos que comparten un namespace (group_name).

El namespace permite que los nodos de cada grupo tengan topics independientes, evitando conflictos entre los grupos.

```python
Node(
    name="motor_sys",
    package='motor_control',
    executable='dc_motor',
    emulate_tty=True,
    output='screen',
    parameters=[{
        'sample_time': 0.01,
        'sys_gain_K': 1.75,
        'sys_tau_T': 0.05,
        'initial_conditions': 0.0,
```

```python
Node(
    name="sp_gen",
    package='motor_control',
    executable='set_point',
    emulate_tty=True,
    output='screen',
    parameters=[{
        'amplitude': 2.0,
        'omega': 1.0,
        'signal_type': signal_type,
```

```python
Node(
    name="ctrl",
    package='motor_control',
    executable='controller',
    emulate_tty=True,
    output='screen',
    parameters=[{
        'kp': 0.82,
        'ki': 23.75,
        'kd': 0.0,
        'Ts': 0.01,
```

Se definen los parámetros correspondientes para cada uno de los 3 nodos. El motor (motor_sys), que simula la dinámica del motor; el generador de referencia (sp_gen), que crea la señal de set point (seno, cuadrada o triangular) según el parámetro signal_type y el controlador PID (ctrl), que recibe la referencia y la velocidad del motor, calcula la señal de control usando los parámetros kp, ki y kd.

Finalmente se define la función generate_launch_description(), que crea los tres grupos de control llamando a create_group con sus respectivos namespaces y tipos de señal y luego retorna un LaunchDescription que incluye los tres grupos.

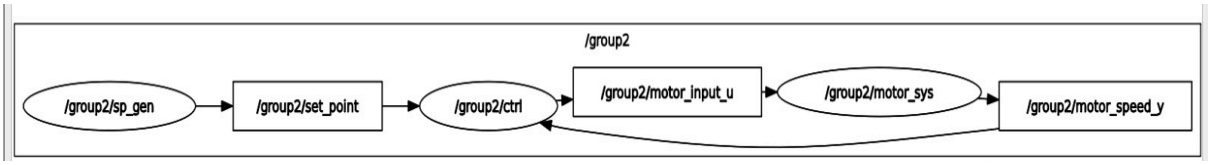
```python
def generate_launch_description():

    group1 = create_group('group1', 'sine')
    group2 = create_group('group2', 'square')
    group3 = create_group('group3', 'triangle')

    return LaunchDescription([
        group1,
        group2,
        group3
```

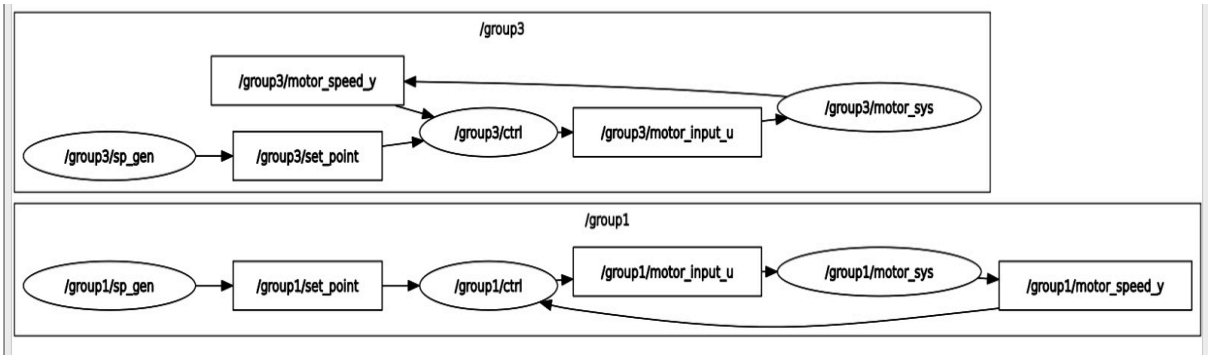
Resultados

A continuación se mostrarán los resultados obtenidos tras la elaboración y ejecución de los nodos mediante el archivo de lanzamiento, que permitirá observar la respuesta del control a distintos tipos de señales (senoidal, cuadrada, triangular).

Integración de los nodos

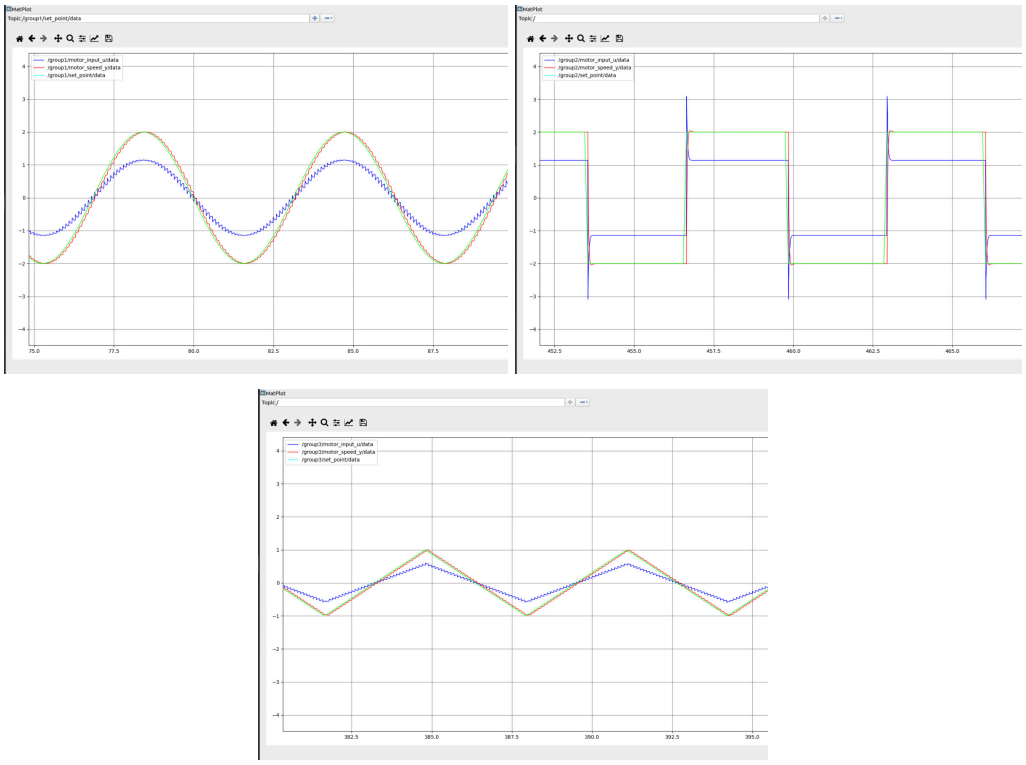
Al ejecutar rqt_graph en una terminal diferente a en la que se lanzó el launch, se logra visualizar que se cumple el comportamiento esperado, donde se cuenta con la presencia de 3 grupos, donde cada uno sigue la estructura de un nodo sp_gen, en publica una señal al nodo ctrl y este a su vez recibe y publica una señal para el nodo motor_sys.





Visualización de las señales

Al ejecutar el archivo launch, se puede verificar de forma gráfica la diferencia entre la señal generada por el nodo sp_gen y la señal generada por el motor tras la aplicación del control. Para cada grupo el control es lo suficientemente preciso para seguir la forma de la señal, independientemente si es senoidal, cuadrada o triangular.



La señal de color verde representa la señal generada por el nodo sp_gen, la señal de color rojo siendo la correspondiente a la velocidad del motor y por ultimo la señal azul, que es la repuesta producida por el control.

Conclusiones

En el desarrollo de este mini reto se logró implementar de manera exitosa un sistema de control en lazo cerrado para un motor de corriente directa (DC) utilizando ROS 2. La arquitectura basada en nodos permitió separar claramente las funciones de generación de referencia, control y simulación del sistema, facilitando tanto el desarrollo como el análisis del comportamiento del sistema completo.

Los objetivos planteados al inicio del trabajo se cumplieron satisfactoriamente, ya que se logró modelar el motor DC como un sistema dinámico de primer orden, implementar un controlador PID discreto y establecer la comunicación entre los nodos mediante tópicos. Asimismo, el uso de parámetros configurables permitió ajustar en tiempo real tanto el modelo del motor como las ganancias del controlador, lo cual resultó fundamental para la sintonización del sistema.

A partir de los resultados obtenidos y visualizados mediante herramientas como rqt_graph y rqt_plot, se pudo observar que el controlador PID fue capaz de seguir adecuadamente distintas señales de referencia (senoidal, cuadrada y triangular). Aunque se presentan ligeros errores transitorios y retardos inherentes a la dinámica del sistema y al tiempo de muestreo, el comportamiento general del sistema se mantiene estable y con un error en estado estacionario reducido.

En caso de que algunos objetivos no se cumplieran completamente, esto podría atribuirse a las limitaciones propias del modelo simplificado del motor, así como a la ausencia de fenómenos reales como saturación, ruido o fricción no lineal. No obstante, estas simplificaciones permiten un mejor entendimiento del funcionamiento del controlador y del entorno ROS 2.

Como posible mejora a la metodología implementada, se podría incorporar un modelo más complejo del motor, incluir saturaciones en la señal de control o implementar técnicas de control más avanzadas. Además, una extensión natural de este trabajo sería la implementación del sistema en un entorno físico real, integrando sensores y actuadores reales para evaluar el desempeño del controlador bajo condiciones no ideales.