Instituto Tecnológico y de Estudios Superiores de Monterrey

TE3001B Fundamentación de Robótica (Gpo 101)

Challenge 1

Autores:

Mariam Landa Bautista // A01736672

Bruno Manuel Zamora García // A01798275

Elías Guerra Pensado //

Profesor:

Juan Manuel Ahuactzin

Rigoberto Cerino

Alfredo García Suarez

Graciela D. Rodríguez Paz

Jueves 19 de Febrero de 2025 Semestre (6) Feb-Jul 2025

Campus Puebla

- **RESUMEN**

En este reporte se incluye el proceso para familiarizarse con el funcionamiento de ROS2 (Robot Operative System) realizando un ejemplo que aborda el uso de nodos y tópicos para simular un ambiente de comunicación entre dispositivos. Concretamente, se han creado nodos; el primero se encarga de publicar una señal senoidal a través de un tópico, posteriormente, el siguiente escuchará dicha señal y la utilizará como entrada para modificarla y para producir una nueva. La señal producida también será publicada. Finalmente, ambas señales serán graficadas simultáneamente para evidenciar los cambios de procesamiento.

- **OBJETIVOS**

El objetivo general de este proyecto es desarrollar e implementar un sistema en ROS2 que genere, procese y publique una señal senoidal en tiempo real, permitiendo su modificación y análisis a través de nodos específicos de generación y procesamiento.

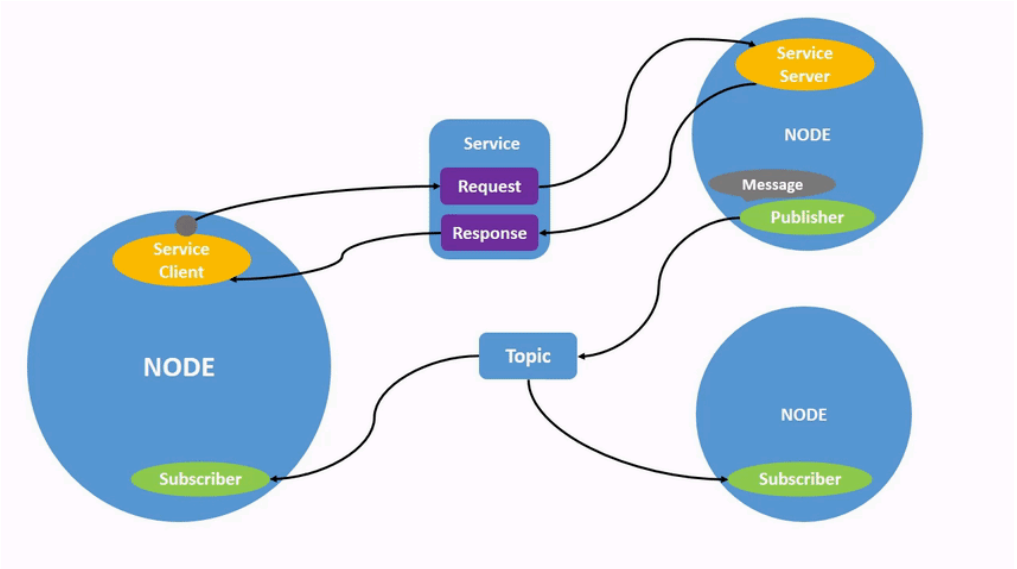
1. Explorar y familiarizarse con el framework ROS y sus herramientas para el desarrollo de aplicaciones robóticas.
2. Investigar las funcionalidades de ROS con el fin de incrementar el grado de interacción y mejorar el rendimiento de los sistemas robóticos.
3. Analizar la estructura y el funcionamiento de los nodos y las señales, comprendiendo la interconexión de sus componentes y sus capacidades físicas.

- **INTRODUCCIÓN**

Este proyecto explora los fundamentos de la comunicación en ROS 2, esencial para comprender su funcionamiento y sus principios clave. El Sistema Operativo

Robótico (ROS) es una plataforma de código abierto diseñada para simplificar el desarrollo de aplicaciones en robótica. Proporciona bibliotecas y herramientas de software que abstraen el hardware, gestionan dispositivos de bajo nivel e implementan funciones comunes. Su propósito principal es fomentar la reutilización de código en el ámbito de la investigación y el desarrollo, utilizando un sistema distribuido basado en procesos conocidos como nodos, los cuales pueden diseñarse y conectarse de manera flexible en tiempo de ejecución.

En ROS, los nodos se comunican a través de un sistema de intercambio de mensajes. Cada nodo es un proceso autónomo diseñado para realizar una tarea concreta, como controlar sensores o ejecutar algoritmos. Los mensajes son estructuras de datos que contienen información relevante, como valores obtenidos de sensores o comandos de control. Para transmitir y recibir estos mensajes, los nodos emplean tópicos, que actúan como canales unidireccionales donde algunos nodos publican datos y otros los reciben mediante suscripciones. Además, ROS incorpora un mecanismo de comunicación basado en servicios, que permite a un nodo solicitar directamente una función específica a otro mediante llamadas de procedimiento remoto.



ROS 2 ha sido diseñado desde sus bases con un enfoque modular y flexible, permitiendo una mayor escalabilidad y adaptación a distintos escenarios y requerimientos dentro de la robótica. Gracias a la implementación del estándar DDS (Data Distribution Service), la comunicación entre los distintos módulos del sistema se vuelve más eficiente, especialmente en entornos distribuidos y con restricciones de tiempo real, algo clave para desarrollos robóticos avanzados. En cuanto a la seguridad, se han integrado mecanismos como el cifrado de datos y la verificación de nodos, garantizando una transmisión de información segura y confiable, imprescindible en sistemas críticos.

- Nivel del sistema de archivos ROS: Estructura principal
- Metapaquetes: Grupos de paquetes relacionados, que pueden agruparse libremente.
- Paquetes: Unidades básicas del software ROS, conteniendo nodos, bibliotecas, y archivos de configuración.
- Manifiesto del paquete: Archivo que proporciona información sobre el paquete, como autor, licencia, y dependencias.
- Mensajes (.msg): Tipo de información enviada entre procesos ROS, definida en archivos .msg dentro de la carpeta msg de un paquete.
- Servicios (.srv): Interacciones de solicitud/respuesta entre procesos, definidas en archivos .srv dentro de la carpeta srv de un paquete.
- Repositorios: Conjuntos de paquetes mantenidos mediante sistemas de control de versiones (VCS) como Git o Subversion, facilitando la publicación mediante herramientas como Bloom.

- **SOLUCIÓN DEL PROBLEMA**

Configuración inicial:

Dadas las instrucciones proporcionadas, se debe crear un nuevo paquete, dentro de esta se crearán los nodos para procesar la señal senoidal. Primero creamos una carpeta, llamada “challenge1”.

```
mkdir challenge1
cd challenge1
mkdir src
cd src
```

Después de crear el espacio de trabajo, es recomendable generar el archivo ejecutable para asegurarse de que la distribución de archivos está funcionando correctamente. Para esto, se emplea el comando colcon, que simultáneamente crea los directorios build, install y log dentro de la carpeta src.

```
colcon build
```

Para agregar los elementos necesarios a las rutas de librerías y proporcionar comandos de shell o bash:

```
source install/setup.bash
```

Finalmente, para ejecutar el contenido dentro del nodo:

```
ros2 run challenge1 signal_generator
```

CREACIÓN DE NODOS:

Nodo signal_generator

Este archivo python ya es posible modificar, el cual publica la señal senoidal. Como buena práctica, deben incluirse las dependencias al inicio del programa y para generar la señal, se utilizará también las librería *numpy.*

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

Una vez construida la *Clase SignalGenerator* y, a diferencia del ejemplo de la clase, se publicará un tipo de dato Float32 para generar la señal. Después del constructor de inicialización, se asigna el nombre al nodo siendo *signal_generator* y se agrega la construcción de dos tópicos signal y time, correspondientemente. El mensaje de publicación será desplegado a una frecuencia de 10 hz.

```
class SignalGenerator(Node):
  def __init__(self):
    super().__init__('signal_generator')
    self.signal_pub = self.create_publisher(Float32, '/signal', 10)
    self.time_pub = self.create_publisher(Float32, '/time', 10)
    self.timer = self.create_timer(0.1, self.timer_callback)  # 10 Hz
    self.time = 0.0
```

Posteriormente, se define un segundo método timer_callback, el cual se encargará de publicar la señal. Utiliza un timer para obtener el tiempo real en segundos y crea un mensaje Float32 para signal y time en sus respectivos tópicos. Finalmente se imprime en la terminal el valor del *time* y la función evaluada para cada instante de tiempo.

```
def timer_callback(self):
    signal = np.sin(self.time)
    time_msg = Float32()
    time_msg.data = self.time
    signal_msg = Float32()
    signal_msg.data = signal

    self.time_pub.publish(time_msg)
    self.signal_pub.publish(signal_msg)
    self.get_logger().info(f'Time: {self.time}, Signal: {signal}')

    self.time += 0.1
```

Nodo process:

Tal como su nombre lo sugiere, este segundo nodo será responsable de procesar la señal recibida tras suscribirse al nodo previo. Siguiendo el mismo enfoque, se crea un nuevo nodo en un archivo de Python llamado `process.py`. Una vez generado, este nodo deberá ser modificado para funcionar simultáneamente como listener y talker, permitiendo así la retransmisión de la señal ya procesada.

Asimismo, se incorporan las librerías necesarias junto con la dependencia del tipo de dato `Float32`.

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

La estructura de la clase sigue siendo la misma; no obstante, es necesario realizar las adaptaciones adecuadas para que el nuevo nodo funcione como un *listener* en lugar de un *talker*. Esto implica cambios en el nombre de la clase, así como en los métodos y objetos utilizados. En cuanto a los tópicos, se debe definir uno para la publicación de la señal procesada y dos suscriptores encargados de recibir la señal de entrada.

```
class ProcessNode(Node):

  def __init__(self):

    super().__init__('process')

    self.signal_sub = self.create_subscription(Float32, '/signal', self.signal_callback, 10)

    self.time_sub = self.create_subscription(Float32, '/time', self.time_callback, 10)

    self.proc_signal_pub = self.create_publisher(Float32, '/proc_signal', 10)

    # Parámetros de modificación de la señal

    self.phase_shift = np.pi / 20  # Desfase en radianes

    self.amplitud = 0.69  # Factor de escalado de la amplitud

    self.offset = 0.45  # Desplazamiento vertical

    self.smooth_filtro = 0.1 # Factor de suavizado para reducir cambios bruscos

    # Variables para almacenar el estado de la señal

    self.processed_signal = None  # Última señal procesada
```

```
    self.last_time = None  # Última marca de tiempo recibida

    self.time_threshold = 0.1  # Intervalo de actualización en segundos (10 Hz)
```

Lo siguiente es crear el segundo método para publicar la señal procesada. La función se ejecuta cada vez que un mensaje es recibido a través del tópico de *signal*. Se encarga de procesar la información de la señal ('msg.data'), la modifica y después la pública en el tópico de proc_signal.

```
  def signal_callback(self, msg):
    theta = np.arcsin(np.clip(msg.data, -1, 1))  # Convierte la señal a un ángulo válido
    theta_m = theta + self.phase_shift  # Aplica el desfase
     processed_signal = self.amplitud * np.sin(theta_m) + self.offset  # Modifica la amplitud y
agrega offset

    # Aplica un filtro para suavizar la señal en caso de cambios bruscos
    # El filtro funciona evitando picos bruscos en la señal que se procesa,
    # en cualquier cambio repentino en la señal, se amortigua con el valor pasado.
    if self.processed_signal is not None:
          processed_signal = self.smooth_filtro * processed_signal + (1 - self.smooth_filtro) *
self.processed_signal

    self.processed_signal = processed_signal  # Guarda la señal procesada
```

Para compilar este segundo nodo, se debe abrir otra terminal y ejecutar los comandos. Una vez construido y para verificar el funcionamiento adecuado, se debe volver a graficar la función.

```
ros2 run rqt_plot rqt_plot
```

launch file:

En este punto ya hemos verificado correctamente el funcionamiento de los 2 nodos mencionados teniendo los resultados esperados. Sin embargo, ejecutar cada nodo por separado puede ser complicado, porque implicaría abrir una cantidad extensa de terminales.

Es por ello que se diseñó "challenge_launch.py" para ejecutar simultáneamente los nodos. Esto lo hicimos con la información proporcionada por manchester robotics.

Crearemos una carpeta para launch file dentro de nuestro paquete.

```
cd Challenge1/src/Challege1
mkdir launch
cd launch
```

Una vez en la carpeta, creamos nuestro archivo y nos aseguramos de darle los permisos necesarios para la ejecución.

```
chmod +x challenge_launch.py
```

Nuestro código de launch es el siguiente:

Es un archivo challenge_launch.py. Ejecutar ambos nodos, signal_generator y process, en diferentes terminales, una para cada nodo.

Abrir un plotter utilizando rqt_plot, en donde se puedan observar ambas señales.

Además el launch file deberá abrir una gráfica, utilizado rqt_graph, en donde se puedan visualizar todos los elementos activos, tanto nodos como tópicos.

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

```
def generate_launch_description():
  return LaunchDescription([
    Node(
      package='challenge1',
      executable='signal_generator',
      name='signal_generator',
      output='screen'
    ),
    Node(
      package='challenge1',
      executable='process',
      name='process',
      output='screen'
    ),
    Node(
      package='rqt_plot',
      executable='rqt_plot',
      name='rqt_plot',
      arguments=['/signal/data', '/proc_signal/data']
    ),
    Node(
      package='rqt_graph',
      executable='rqt_graph',
      name='rqt_graph',
      output='screen'
    )

  ])
```

Tal y como se puede observar, dentro de esta función se agregan los nodos que se desean ejecutar, indicando el paquete y el ejecutable en donde se encuentran.

Dentro de nuestro archivo setup.py se añaden unas líneas de código, la cual nos permite que el paquete actúe como un paquete de ejecución de archivos globales.

```
            (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch',
'*'))),
```

Dado esto, nuestro código setup.py queda de la siguiente manera:

```
package_name = 'challenge1'
setup(
  name=package_name,
  version='0.0.0',
  packages=find_packages(exclude=['test']),
  data_files=[
     ('share/ament_index/resource_index/packages',
        ['resource/' + package_name]),
     ('share/' + package_name, ['package.xml']),
     (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch', '*'))),
  ],
  install_requires=['setuptools'],
  zip_safe=True,
  maintainer='mariamuwu',
  maintainer_email='mariamuwu@todo.todo',
  description='TODO: Package description',
  license='TODO: License declaration',
  tests_require=['pytest'],
  entry_points={
     'console_scripts': [
        'signal_generator = challenge1.signal_generator:main',
        'process = challenge1.process:main'
     ],
  },
```

```
)
```

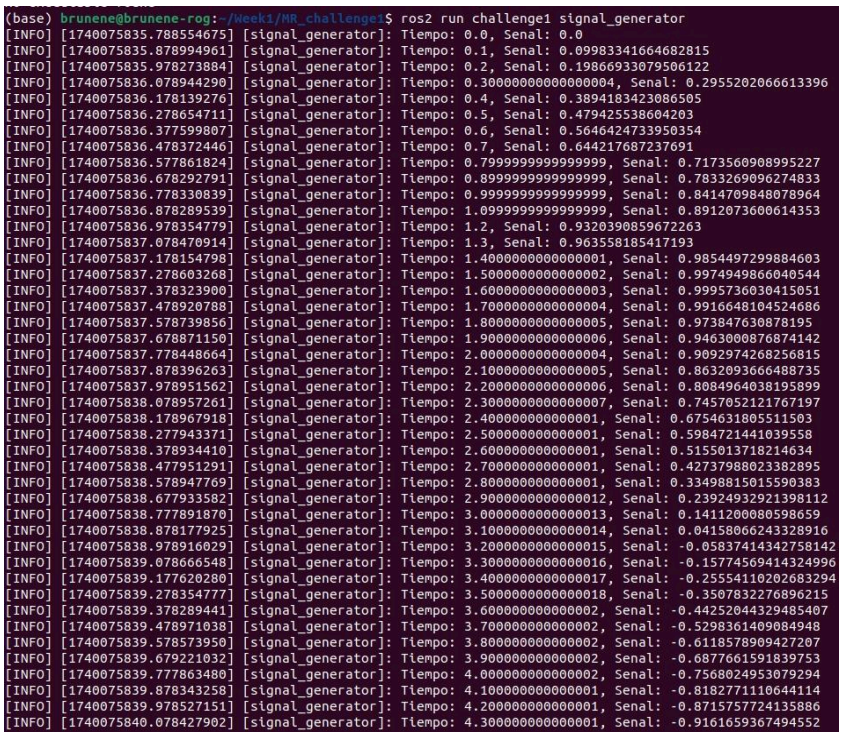
Ahora bien, para verificar que nuestro launch file funcione correctamente, debemos ubicarnos en la carpeta Challenge y realizar la compilación de los paquetes:

```
colcon build
source install/setup.bash
```

Finalmente, ha llegado la hora de probar el funcionamiento de nuestro launch file:

```
ros2 launch challenge1 challenge_launch.py
```

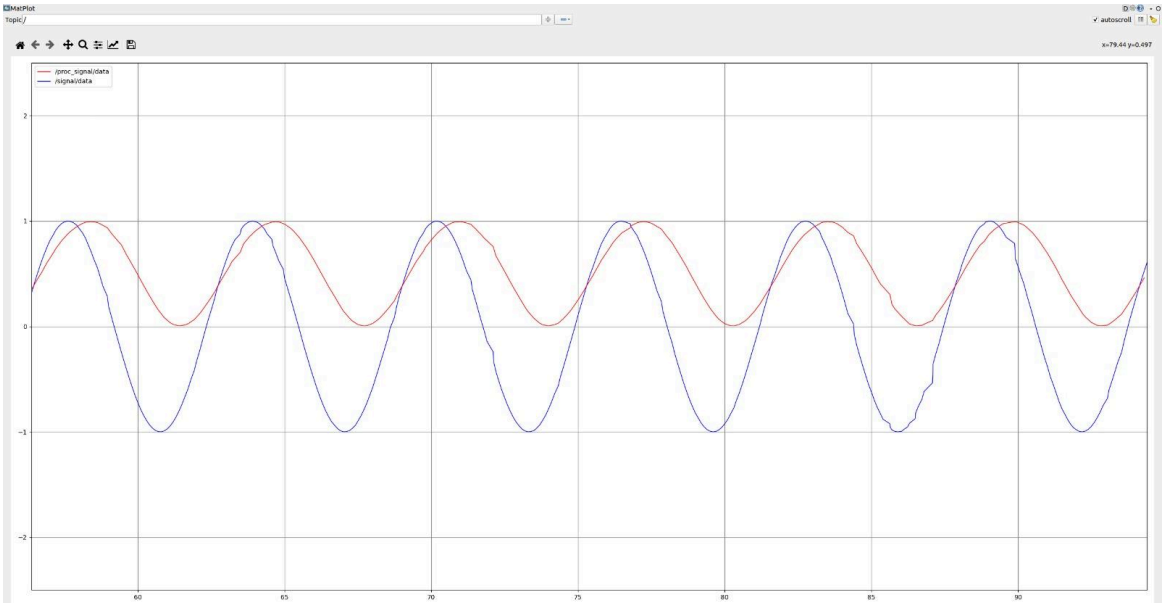
- **RESULTADOS**



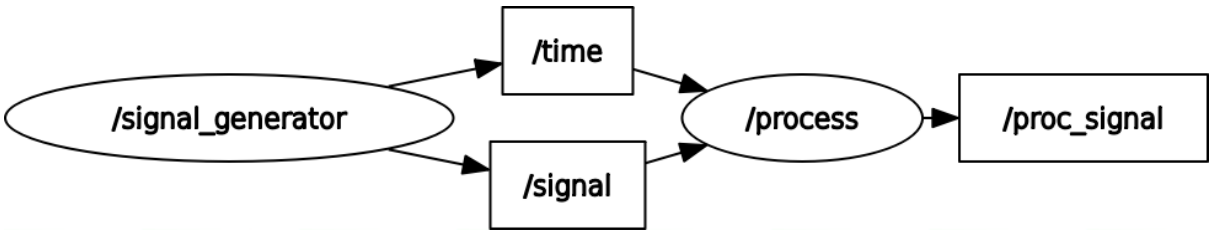
La salida de signal_generator.py, nos muestra el cambio en el tiempo cada 0.1 s, y cada vez que pasa uno de estos ciclos nos da el valor de la señal en base al cálculo del seno con respecto al valor del tiempo que transcurre en ese momento.

```
(base) brunene@brunene-rog:~/Week1/MR_challenge1$ ros2 run challenge1 process
[INFO] [1740075941.796173156] [process]: Tiempo: 53.70s - Señal procesada: 0.357
[INFO] [1740075941.985999255] [process]: Tiempo: 53.90s - Señal procesada: 0.350
[INFO] [1740075942.186094058] [process]: Tiempo: 54.10s - Señal procesada: 0.297
[INFO] [1740075942.284542983] [process]: Tiempo: 54.20s - Señal procesada: 0.271
[INFO] [1740075942.485968173] [process]: Tiempo: 54.40s - Señal procesada: 0.212
[INFO] [1740075942.685711295] [process]: Tiempo: 54.60s - Señal procesada: 0.148
[INFO] [1740075942.785868177] [process]: Tiempo: 54.70s - Señal procesada: 0.116
[INFO] [1740075942.985989059] [process]: Tiempo: 54.90s - Señal procesada: 0.054
[INFO] [1740075943.185370051] [process]: Tiempo: 55.10s - Señal procesada: 0.001
[INFO] [1740075943.285789636] [process]: Tiempo: 55.20s - Señal procesada: -0.018
[INFO] [1740075943.485981992] [process]: Tiempo: 55.40s - Señal procesada: -0.042
[INFO] [1740075943.685578870] [process]: Tiempo: 55.60s - Señal procesada: -0.046
[INFO] [1740075943.785927722] [process]: Tiempo: 55.70s - Señal procesada: -0.040
[INFO] [1740075943.985908681] [process]: Tiempo: 55.90s - Señal procesada: -0.014
[INFO] [1740075944.185969739] [process]: Tiempo: 56.10s - Señal procesada: 0.030
[INFO] [1740075944.285766265] [process]: Tiempo: 56.20s - Señal procesada: 0.059
[INFO] [1740075944.485984129] [process]: Tiempo: 56.40s - Señal procesada: 0.128
[INFO] [1740075944.685822030] [process]: Tiempo: 56.60s - Señal procesada: 0.210
[INFO] [1740075944.785110491] [process]: Tiempo: 56.70s - Señal procesada: 0.255
[INFO] [1740075944.985636413] [process]: Tiempo: 56.90s - Señal procesada: 0.351
[INFO] [1740075945.185894341] [process]: Tiempo: 57.10s - Señal procesada: 0.450
[INFO] [1740075945.285640953] [process]: Tiempo: 57.20s - Señal procesada: 0.500
[INFO] [1740075945.485994773] [process]: Tiempo: 57.40s - Señal procesada: 0.598
[INFO] [1740075945.685933332] [process]: Tiempo: 57.60s - Señal procesada: 0.690
[INFO] [1740075945.786346896] [process]: Tiempo: 57.70s - Señal procesada: 0.732
[INFO] [1740075945.985559631] [process]: Tiempo: 57.90s - Señal procesada: 0.809
[INFO] [1740075946.185746076] [process]: Tiempo: 58.10s - Señal procesada: 0.871
[INFO] [1740075946.286363694] [process]: Tiempo: 58.20s - Señal procesada: 0.898
[INFO] [1740075946.485948499] [process]: Tiempo: 58.40s - Señal procesada: 0.943
[INFO] [1740075946.685943993] [process]: Tiempo: 58.60s - Señal procesada: 0.976
[INFO] [1740075946.785927129] [process]: Tiempo: 58.70s - Señal procesada: 0.986
[INFO] [1740075946.985674851] [process]: Tiempo: 58.90s - Señal procesada: 0.994
[INFO] [1740075947.185582362] [process]: Tiempo: 59.10s - Señal procesada: 0.984
[INFO] [1740075947.286075135] [process]: Tiempo: 59.20s - Señal procesada: 0.972
[INFO] [1740075947.485657354] [process]: Tiempo: 59.40s - Señal procesada: 0.935
[INFO] [1740075947.685528527] [process]: Tiempo: 59.60s - Señal procesada: 0.881
[INFO] [1740075947.784659853] [process]: Tiempo: 59.70s - Señal procesada: 0.848
[INFO] [1740075947.985454179] [process]: Tiempo: 59.90s - Señal procesada: 0.772
[INFO] [1740075948.185079130] [process]: Tiempo: 60.10s - Señal procesada: 0.684
[INFO] [1740075948.285607700] [process]: Tiempo: 60.20s - Señal procesada: 0.637
[INFO] [1740075948.485369577] [process]: Tiempo: 60.40s - Señal procesada: 0.538
[INFO] [1740075948.685943549] [process]: Tiempo: 60.60s - Señal procesada: 0.436
[INFO] [1740075948.785698206] [process]: Tiempo: 60.70s - Señal procesada: 0.386
[INFO] [1740075948.984680630] [process]: Tiempo: 60.90s - Señal procesada: 0.287
```

La salida process.py nos muestra el tiempo que transcurre en signal generator y el valor de la señal modificada con los parámetros solicitados. Estas modificaciones se pueden ver más claramente en la parte donde se muestra el código.



Esta es la graficación de cómo las señales de entrada y la de salida se ven con respecto al tiempo. Esta gratificación se realiza con la ayuda de rqt_plot.



La última imagen es la representación gráfica de cómo nuestros nodo y tópicos se conectan entre sí para para dar una salida que es la señal procesada esta se hace con la ayuda de rqt_graph.

- CONCLUSIONES

Durante el desarrollo del challenge, logramos cumplir satisfactoriamente los objetivos propuestos, nos familiarizamos con ROS2 de manera introductoria, lo que nos permitió hacer uso adecuado de sus funcionamientos básicos. Además creamos correctamente los nodos signal generator y process, así como los tópicos /signal y /proc_signal. A pesar de que tenemos resultados muy certeros a lo que se nos solicitó en el reto, sabemos que pudimos tener mejores y más precisos.

El uso de las herramientas rqt_plot y rqt_graph nos permitió visualizar el esquema de manera adecuada, mostrando la conexión entre los nodos y los tópicos que facilitan la transmisión de información.

Y aunque la forma en la que manejamos la señal original aún puede llegar a tener errores si se modifican los valores ya determinados en el código, esta

puede ser una pauta para tener más experiencia de cómo trabajar para los siguientes retos y tomar en cuenta más variables que podrían llegar a afectar el correcto funcionamiento de nuestros códigos y la graficación de nuestras variables principales.

- **BIBLIOGRAFÍAS**

Martínez Bedoya, J. F. (s.f.). *Tutorial ROS2*. Semillero ARES. https://ares.davinsony.com/tutorial/ROS2/

Open Source Robotics Foundation. (s.f.). *Introducción a ROS*. ROS Wiki. https://wiki.ros.org/es/ROS/Introduccion

Open Source Robotics Foundation. (s.f.). *Conceptos de ROS*. ROS Wiki. https://wiki.ros.org/es/ROS/Conceptos

Manchester Robotics Ltd. (2025). *MCR2 MiniChallenge 1* [Presentación de diapositivas]. GitHub. https://github.com/ManchesterRoboticsLtd/TE3001B_Intelligent_Robotics_Implementation_2025/blob/main/Week%201/Challenge/MCR2_MiniChallenge_1.pptx