



# Tecnológico de Monterrey

**Instituto Tecnológico de Estudios Superiores de Monterrey**

**Campus Puebla**

**Fundamentación de Robótica (Gpo 101)**

**Reporte: Reto 2 Manchester Robotics**

**Alumno**

José Diego Tomé Guardado A01733345

Pamela Hernández Montero A01736368

Victor Manuel Vázquez Morales A01736352

Fernando Estrada Silva A01736094

**Fecha de entrega**

Jueves 29 de Febrero de 2024

## **RESUMEN**

Durante la investigación, se exploraron los conceptos de namespaces, parameters y custom messages en ROS 2; implementando estos términos en 2 nodos que intercambian información para crear y reconstruir diferentes tipos de señales (senoidal, cuadrada, diente de sierra o triangular.).

## **OBJETIVOS**

El objetivo de la práctica consiste en desarrollar dos nodos que trabajarán juntos utilizando interfaces de mensajes para comunicarse de manera efectiva. Uno de los nodos creará y modificará una función predeterminada, mientras que el otro nodo utilizará los parámetros enviados a través de las interfaces de mensajes para reconstruir la salida de la función. Los objetivos particulares que se consideraron para la ejecución de nuestro programa fueron:

- ★ **Implementar el nodo creador de señal:** Desarrollar el primer nodo que será responsable de crear una salida de señal seleccionada por los parámetros desde la terminal utilizando YAMP y ajustar la función en consecuencia.
- ★ **Modificar las configuraciones de YAMP desde la terminal:** Se podrán ajustar las configuraciones desde la terminal utilizando YAMP, lo cual impactará en la modificación de los parámetros del nodo 1.
- ★ **Implementar launch:** Esto posibilitará iniciar el nodo junto con sus configuraciones de manera coordinada.
- ★ **Definir interfaces de mensajes:** Verificar que los archivos CMakeLists.txt y package.xml de cada paquete estén correctamente configurados para incluir las dependencias de los mensajes personalizados y garantizar una construcción exitosa del paquete.
- ★ **Implementar el nodo receptor y reconstructor de función:** Este estará encargado de recibir los parámetros enviados desde el nodo uno utilizando las interfaces de mensajes. Luego, utilizará estos parámetros para reconstruir la señal de salida de la función predeterminada.

## **INTRODUCCIÓN**

El presente trabajo se adentra en el ámbito de la comunicación entre nodos, centrándose en el entorno de ROS 2 (Robot Operating System 2). ROS 2 es una plataforma que facilita la comunicación entre diferentes componentes de un sistema robótico, como sensores, actuadores y algoritmos de procesamiento.

Es por ello que en este contexto, nos enfocamos en la utilización del lenguaje YAML para definir características de configuración en ROS 2, lo que nos permite ajustar dinámicamente los parámetros de los nodos y adaptar el comportamiento del sistema según las necesidades

específicas de cada aplicación. Esto se vuelve especialmente relevante en entornos donde la flexibilidad y la adaptabilidad son fundamentales.

El desarrollo de sistemas robóticos complejos implica la interacción entre múltiples componentes, cada uno con sus propias funcionalidades y requisitos. En este sentido, la comunicación efectiva entre nodos se convierte en un aspecto crítico para el funcionamiento adecuado del sistema. Para lograr esto, ROS 2 proporciona una serie de herramientas y conceptos, como namespaces, parameters y custom messages, que facilitan la interacción y la colaboración entre los diferentes componentes del sistema.

### Conceptos Clave a Explorar:

**Namespaces:** En ROS 2, los namespaces son una herramienta que permite organizar y estructurar los nodos y los tópicos dentro de un sistema. Funcionan como contenedores lógicos que agrupan elementos relacionados, lo que ayuda a evitar conflictos de nombres y a facilitar la modularidad y la escalabilidad del sistema.

“En pocas palabras, los Namespaces y Domain IDs en ROS 2 proporcionan una forma de dividir la red ROS en diferentes grupos aislados, lo que permite que varios robots operen de forma independiente sin interferir entre sí.” (Ramachandran. R. 2023)

Tal y como menciona Ramachandran es posible con los namespace agrupar en diferentes grupos permitiendo que aunque tengan el mismo nombre puedan operar en contextos diferentes, los espacios de nombres son los que les permiten operar de forma independiente.

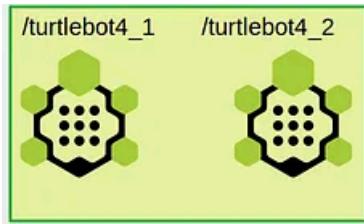


Fig 1. Ejemplo de un namespace (Ramachandran. R. 2023).

**Parameters:** Los parámetros en ROS 2 son valores configurables que pueden afectar el comportamiento de los nodos y los tópicos en tiempo de ejecución. Permiten ajustar diversas configuraciones, como velocidades, umbrales y otras variables, sin necesidad de recompilar el código. La integración del lenguaje YAML facilita la definición y gestión de estos parámetros de manera estructurada y legible.

Durante el inicio, los nodos leen los valores de los parámetros a través de la interfaz de línea de comandos o de archivos YAML para determinar su comportamiento. Estos valores pueden ser números enteros, de punto flotante, booleanos, cadenas o matrices. Al establecer diferentes valores de parámetros en diferentes contextos, un nodo puede modificar su comportamiento para funcionar en diversos escenarios. Por ejemplo, un nodo que controla la velocidad de un robot puede configurarse con distintos límites de velocidad según el entorno

en el que se despliega el robot. La estructura que conforma a los parámetros de puede ver reflejada en la figura 2.

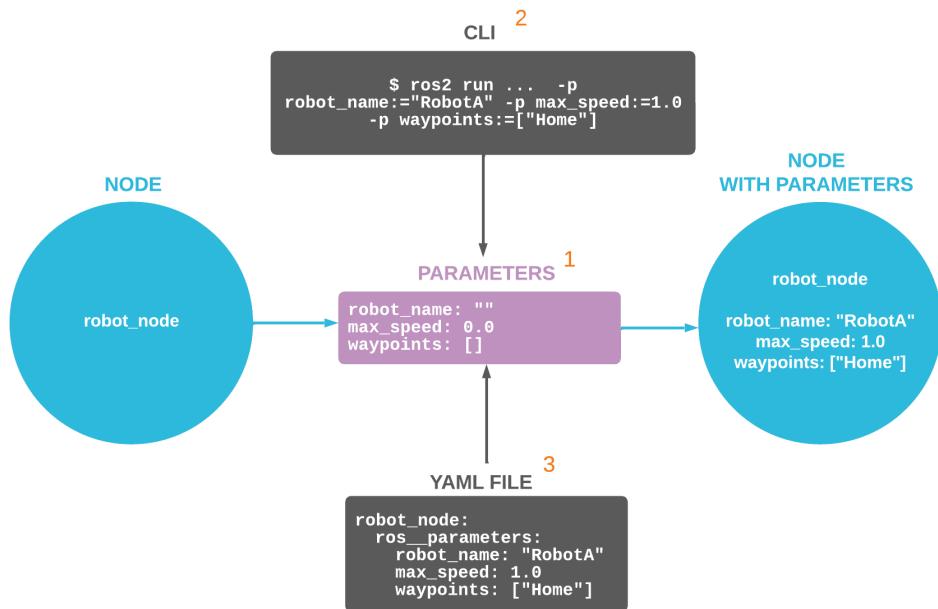


Fig 2. Estructura del uso de parámetros (Millán J. 2022).

Como se puede observar en la figura 2 los parámetros se definen y gestionan a través de la interfaz de línea de comandos (CLI) o mediante archivos de configuración YAML. Estos parámetros se pueden acceder y modificar durante la ejecución del sistema, lo que proporciona una flexibilidad considerable en la adaptación del comportamiento de los nodos.

**Custom Messages:** En ROS 2, los mensajes personalizados son estructuras de datos definidas por el usuario que se utilizan para intercambiar información entre los nodos. Estos mensajes pueden contener cualquier tipo de datos, incluidos números, cadenas de texto, matrices y estructuras complejas. La creación de mensajes personalizados permite una comunicación más específica y eficiente entre los nodos.

Para utilizar mensajes personalizados primero se define la estructura del mensaje en un archivo de tipo .msg. Luego, se compila este archivo para generar el código fuente correspondiente en el lenguaje de programación seleccionado (por lo general, C++ o Python). Una vez compilados, los mensajes personalizados pueden ser utilizados por los nodos para intercambiar información de manera específica y eficiente.

En este trabajo se explorará en detalle la implementación y el funcionamiento de estos conceptos ayudando a proporcionar una comprensión clara y práctica de cómo utilizarlos en el desarrollo de sistemas robóticos.

## SOLUCIÓN AL PROBLEMA

Para la solución de este reto, se utilizaron los recursos brindados por Manchester Robotics.

Al igual que en la actividad anterior, se ha decidido implementar una estrategia similar. Para evitar cualquier tipo de malentendido o incluso desorganización, se ha creado un nuevo espacio de trabajo para este segundo reto. Para este nuevo directorio llamado *Challenge2*, se le ha proporcionado su propia carpeta fuente *src* para almacenar sus paquetes.

```
$ mkdir Challenge2  
Challenge2/ $ mkdir src
```

Retomando la solución implementada durante el reto 1 sobre procesamiento de señales con ros2, se solicita en esta ocasión crear un nuevo paquete llamado *signal\_params*, en el cual se incluirán parámetros del programa. Como bien se mencionó, un parámetro es una variable con valores predefinidos almacenados en un archivo distinto de la carpeta con acceso abierto al usuario.

Para crear el paquete de *signal\_params*, se utiliza el siguiente comando:

```
ros2 pkg create --build-type ament_python --license Apache-2.0 signals_params  
--dependencies rclpy std_msgs
```

En este nuevo paquete se incluye un nodo llamado *signal\_generator* junto con las dependencias ya antes utilizadas. En este archivo de python, se construye el generador de señal pero se incluirán los parámetros solicitados: amplitud, offset y phase shift, etc. Se incluye también la publicación de la señal construida, así como sus parámetros. Estos pueden ser modificados a través del valor de entrada proporcionado por el usuario en la terminal.

Para este caso, hemos decidido implementar 5 señales que el usuario podrá escoger desde terminal:

1. Senoidal
2. Cuadrada
3. Dientes de sierra
4. Triangular
5. Coseno

Es decir, en caso de seleccionar la opción 4, se asignan dichos parámetros a una señal correspondiente y esta podrá modificarse constantemente, incluso al momento de graficar en tiempo real.

Además, como buena práctica, es importante construir estos mismos parámetros dentro de un archivo YAML dentro de la carpeta *config* (ubicada dentro del paquete *signal\_params*). Este archivo nos servirá para almacenar configuraciones y parámetros de nodos y paquetes de

manera estructurada. Incluso pueden funcionar para definir la configuración del launch, especificando qué nodos se deben iniciar o bien, qué parámetros utilizar.

A continuación se muestra la construcción del archivo YAML, nótese que es de suma importancia asignarle el nombre del nodo, ya que en caso de cambiarlo, puede haber errores de compilación.

En términos generales, y como se mencionaba, el archivo incluye una serie de atributos con los que puede ser modificada una señal para convertirse en otra. Para este caso, hemos declarado un entero (selection) que servirá para cambiar el tipo de señal. De igual manera, se han construido una serie de arreglos para los distintos parámetros de la señal, en donde cada elemento corresponde a un tipo de señal:

```
signal_generator_node:  
ros__parameters:  
    selection: 1 #1: Sine, 2: Square, 3: Sawtooth 4: Triangle, 5: Cos  
    amplitude_array: [0.5, 1.0, 1.0, 3.0, 2.0]  
    frequency_array: [2.0, 1.0, 1.0, 1.5, 3.0]  
    offset_array: [0.0, 1.0, 2.0, 3.0, 5.0]  
    phase_shift: [1.0, 1.0, 1.0, 4.0, 2.0] #Desfase entre señal original y reconstruida
```

**Archivo yaml:** El funcionamiento de los parámetros es sencillo. Supongamos que el usuario cambia el valor de selection a 4, que corresponde a una señal triangular. Para este caso, la amplitud, frecuencia, offset y phase\_shift se tomarán del elemento 3 (4-1) de cada arreglo.

## Nodo signal\_generator

Para el desarrollo de este código, crearemos el archivo *signal\_generator.py*. Necesitaremos hacer uso de algunas de las librerías con las que ya hemos trabajado previamente, tales como *rclpy*, *numpy*, *time* y *std\_msgs*. Para esta nueva actividad, incluiremos de igual forma la librería *scipy*, la cual nos permitirá generar ondas de tipo cuadradas, diente de sierra y triangulares.

```
from scipy import signal
```

Ahora bien, procedemos a definir la clase *Signal\_Generator*, la cuál será la encargada de crear el nodo ‘*signal\_generator\_node*’. Ahora bien, declararemos los parámetros del nodo, los cuales servirán para configurar más adelante ciertos valores de interés para la generación de la señal:

```
self.declare_parameters(  
    namespace='',  
    parameters=[  
        ('selection', rclpy.Parameter.Type.INTEGER),
```

```

('amplitude_array', rclpy.Parameter.Type.DOUBLE_ARRAY),
('frequency_array', rclpy.Parameter.Type.DOUBLE_ARRAY),
('offset_array', rclpy.Parameter.Type.DOUBLE_ARRAY),
('phase_shift', rclpy.Parameter.Type.DOUBLE_ARRAY)
]
)

```

Ya que hemos declarado los parámetros de configuración del nodo, continuaremos con la creación del publisher, el cual mandará la señal calculada por el tópico ‘signal’ a una frecuencia de 1khz (0.001 s). Para ello, crearemos una variable de tipo *Float32()*, así como un timer que estará asociado con la función *signal\_callback*, la cuál sigue la siguiente lógica de programación:

1. Obtenemos el valor del parámetro ‘selection’, el cuál puede ir de 1 a 5 e indica el tipo de señal a graficar. En caso de que el parámetro sea modificado fuera del rango (menor a 0 o mayor que 5), se cargará el valor de 1 a nuestra variable local del mismo nombre (selection):

```

def signal_callback(self):
    selection = self.get_parameter('selection').get_parameter_value().integer_value
    if(selection>5 or selection<=0):
        selection = 1

```

2. Obtenemos el valor del resto de los parámetros y los guardamos en variables locales:

```

amplitude = self.get_parameter('amplitude_array')...
frequency = self.get_parameter('frequency_array')...
offset = self.get_parameter('offset_array')...
t = time.time()

```

3. Utilizando condicionales, creamos y calculamos el valor de la señal acorde al valor del parámetro ‘selection’. Posteriormente, lo imprimimos en terminal y publicamos el valor a través del publisher *signal\_pub*:

```

if(selection == 1): #Sine
    self.signal_value.data = amplitude*np.sin(2*np.pi*frequency*t) + offset
    ...
if(selection == 5): #Cos
    self.signal_value.data = amplitude*np.cos(2*np.pi*frequency*t) + offset
self.get_logger().info(f"Value: {self.signal_value.data}")
self.signal_pub.publish(self.signal_value)

```

**Nota:** Recordemos que el valor de *selection* indica el tipo de señal a graficar: 1 - Senoidal, 2 - Cuadrada, 3 - Dientes de sierra, 4 - Triangular, 5 - Coseno.

Ahora que hemos implementado el código de *signal\_generator*, no debemos olvidar modificar el archivo *setup.py*, específicamente en *entry\_points*:

```

entry_points={
    'console_scripts': [
        'signal_generator = signals_params.signal_generator:main'],
},

```

Ahora bien, ¿qué pasaría si intentamos correr el nodo desde terminal? A simple vista todo parece correctamente declarado e implementado, pero la realidad es que obtendremos errores al intentar ejecutarlo. Esto se debe a que no hemos inicializado aún los parámetros del nodo.

## **Inicialización de parámetros del nodo**

Para inicializar los parámetros del nodo '*signal\_generator\_node*', haremos uso del archivo .yaml previamente creado y de un launch file. Para ello, en el directorio de nuestro paquete *signals\_params* deberemos crear una carpeta con el nombre *launch*. Dentro de esta carpeta, creamos un archivo con el nombre *project\_launch.py*. Dicho launch file seguirá la siguiente lógica de programación:

1. Importación de librerías:

```

import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

```

2. Dentro de la función *generate\_launch\_description()* deberemos crear una variable (*config*) que contenga la ubicación de nuestro archivo .yaml:

```

def generate_launch_description():
    config = os.path.join(
        get_package_share_directory('signals_params'),
        'config',
        'params.yaml')

```

3. Finalmente, declaramos el nodo *signal\_generator* considerando los parámetros del archivo *config*:

```

signal_generator = Node(
    package='signals_params',
    executable='signal_generator',
    output='screen',
    prefix = 'gnome-terminal --',
    parameters=[config]
)
ld = LaunchDescription([signal_generator])
return ld

```

Para el funcionamiento de nuestro launch file, debemos modificar el archivo *setup.py*, indicando la ubicación de nuestra carpeta launch, así como la ubicación de nuestro archivo *.yaml*. Para ello, agregaremos las siguientes líneas de código dentro de *data\_files*:

```
(os.path.join('share', package_name, 'launch'),  
glob(os.path.join('launch', '*launch.[pxy][yma]*'))),  
(os.path.join('share', package_name, 'config'),  
glob('config/*.yaml')))
```

De igual forma, es importante agregar la siguiente dependencia dentro del archivo *package.xml*:

```
<exec_depend>ros2launch</exec_depend>
```

### Custom messages:

Hasta el momento, todo parece funcionar perfectamente, por lo que podríamos proceder a crear el siguiente nodo, el cual es un reconstructor de la señal del *signal\_generator*. Ahora bien, para que este segundo nodo pueda reconstruir la señal, deberá conocer los parámetros de la señal, tales como su tipo, frecuencia, desfasamiento, etc.

Podríamos optar por crear una serie de tópicos para los distintos parámetros, pero esto puede resultar bastante ineficiente. En su lugar, optamos por crear un custom message, es decir, un mensaje personalizado que nos permitirá realizar el intercambio de información entre el nodo generador y el reconstructor.

Procederemos a declarar nuestro propio msg, pero para ello es necesario crear un nuevo paquete que contendrá los archivos ya mencionados. Nos situaremos dentro de nuestra carpeta src y ejecutaremos el siguiente comando:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 srv_int
```

Nos moveremos al directorio del paquete creado y haremos una nueva carpeta en donde guardaremos todos nuestros mensajes personalizados. En este caso, únicamente tendremos el archivo *Parameters.msg*:

```
$ cd srv_int/  
$ mkdir msg  
$ touch msg/Parameters.msg
```

**Nota:** Un archivo .msg debe iniciar con mayúscula, en caso contrario puede provocar errores de ejecución.

Dentro del archivo creado, declararemos las variables o atributos que tendrá nuestro *msg*, así como el tipo de dato para cada uno de ellos:

```
int64 type
float64 frequency
float64 offset
float64 time
float64 phase_shift
float64 amplitude
```

Finalmente, para poder hacer uso de este msg más adelante, deberemos modificar el archivo *CMakeLists.txt*, agregando las siguientes líneas de código:

```
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
"msg/Parameters.msg"
)
```

De igual forma, deberemos modificar el archivo *package.xml* (del paquete *srv\_int*), agregando las siguientes dependencias:

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Para verificar que todo se haya realizado de manera correcta, podemos compilar el paquete usando colcon build y posteriormente ejecutar el siguiente comando dentro de la carpeta *src*:

```
$ ros2 interface show srv_int/msg/Parameters
```

Este comando nos arrojará en terminal todo lo que tenemos dentro del archivo *Parameters.msg*.

**NOTA:** En caso de obtener errores relacionados con no encontrar el paquete, se recomienda cerrar la terminal

### **Modificación a la signal\_generator:**

Ahora que hemos creado el paquetes *srv\_int* (que contiene los custom msgs), procedemos a realizar algunas modificaciones en el nodo *signal\_generator* para que pueda mandar los parámetros de la señal por otro tópico.

Lo primero que debemos realizar es agregar la dependencia al paquete dentro del *setup.py* de nuestro paquete *signals\_params*:

```
<exec_depend>srv_int</exec_depend>
```

Posteriormente, estamos listos para utilizar nuestro mensaje dentro del código. Para importar este msg, agregamos la siguiente línea de código en la parte superior del archivo:

```
form srv_int.msg import Parameters
```

Lo siguiente que debemos hacer es crear un publisher para los parámetros, el cuál llevará datos de tipo *Parameters* cada 0.1s (10Hz) a través del tópico ‘*signal\_params*’:

```
self.parameters_pub = self.create_publisher(Parameters, 'signal_params', 10)
parameters_period = 0.1 #10hz
```

De igual manera, crearemos otro timer para publicar los parámetros, utilizando el periodo previamente declarado y “asociándolo” con la función *send\_parameters*:

```
self.timer_parms = self.create_timer(parameters_period, self.send_parameters)
```

No olvidemos declarar la variable correspondiente para almacenar los parámetros de la señal:

```
self.parameters_values = Parameters()
```

En términos generales, la función se encarga de acceder a cada uno de los parámetros de la señal, considerando primero el entero que indica el tipo de señal para posteriormente extraer los datos de amplitud, *frecuencia*, *offset* y *phase\_shift* de los parámetros de tipo array que contienen estos datos. Una vez que accede a estos datos, los asigna a cada uno de los “atributos” de la variable tipo *Parameters* para finalmente publicarla a través del publisher previamente creado. A continuación mostramos una parte de la función:

```
def send_parameters(self):
    selection = self.get_parameter('selection').get_parameter_value().integer_value
    self.parameters_values.type = selection
    self.parameters_values.amplitude = self.get_parameter('amplitude_array').get_
    parameter_value().double_array_value[selection-1]
    ...
    self.parameters_values.time = time.time()
    self.parameters_pub.publish(self.parameters_values)
```

## Nodo Reconstructor

Por otro lado, se solicita construir un segundo que actúe como reconstructor de señal. Como en prácticas anteriores, este nodo es muy parecido al publicador, siguiendo una estructura similar en cuanto a métodos y dependencias.

Este nodo debe suscribirse al tópico de *signal\_params* para construir una señal que incluya los parámetros proporcionados por el usuario. Como en prácticas anteriores, es de suma importancia indicar que se ha construido un nuevo nodo en el archivo de *setup.py* (dentro de *entry\_points*), colocando su nombre y tópico correspondiente.

```
'reconstructor = signals_params.reconstructor:main'
```

Después de crear la suscripción al tópico de *signal\_params*, deben inicializarse las nuevas variables para la señal construida (tipo de señal, frecuencia, offset, phase shift y amplitud). Se mantienen en 0 únicamente para declararlas y evitar errores.

```
class Signal_Reconstructor(Node):
    def __init__(self):
        super().__init__('reconstruction')
        self.subscription = self.create_subscription(Parameters,'signal_params',
                                                     self.reconstruction_callback,10)
        self.type = 0
        self.frequency = 0.0
        self.offset = 0.0
        self.time_susctract = 0.0
        self.phase_shift = 0.0
        self.amplitude = 0.0
```

Para reconstruir la señal, será necesario hacer uso igual del paquete `srv_int`. Como pudimos observar en la parte de arriba, la suscripción está asociada con el método `reconstruction_callback`, la cuál se encarga de copiar los parámetros recibidos del mensaje en las variables del nodo.

```
def reconstruction_callback(self, msg):
    self.type = msg.type
    self.frequency = msg.frequency
    self.offset = msg.offset
    self.time_subtraction= time.time() - msg.time
    self.phase_shift = msg.phase_shift
    self.amplitude = msg.amplitude
```

La variable `time_subtraction` se usa para evitar un desfase en la señal en caso de que el tiempo para cada nodo sea diferente.

Finalmente, la función *reconstructed\_signal* se encarga de reconstruir la señal respetando los parámetros recibidos en el tópico del publicador junto de acuerdo al temporizador establecido. Para aclarar, se define un condicional numérico (1-5) que corresponde a cada una de las distintas señales:

```
def reconstructed_signal(self):
    t = time.time()
    argument = 2*np.pi*self.frequency*(t-self.time_susctruction)+self.phase_shift
    if (self.type == 1): #Sine
        self.signal_value.data = self.amplitude*np.sin(argument) + self.offset
    ...

```

```

        elif(self.type == 5): #Cos
            self.signal_value.data = self.amplitude*np.cos(argument) + self.offset

```

De igual forma, este nodo se encarga de publicar la señal a través de su propio publisher, mandando el mensaje por el tema ‘reconstructed\_signal’:

```

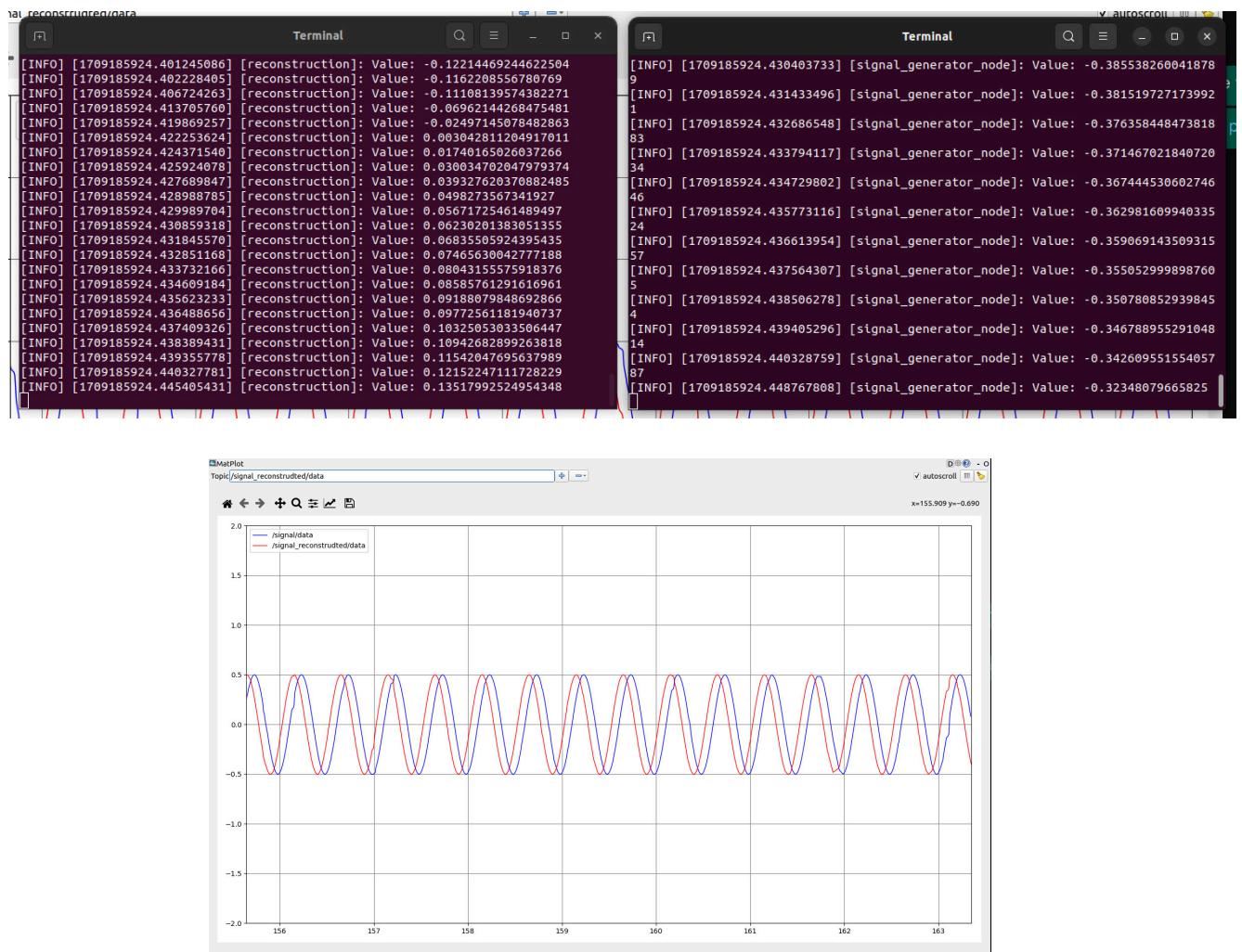
self.get_logger().info(f"Value: {self.signal_value.data}")
self.publisher.publish(self.signal_value)

```

Dado que se han acumulado un considerable número de archivos de vital importancia para el programa, se ha construido un archivo launch con el fin de ejecutar múltiples nodos simultáneamente sobre una sola terminal. Modificamos nuestro archivo launch para lanzar de igual forma el nodo reconstructor, así como para lanzar rqt\_plot y rqt\_graph.

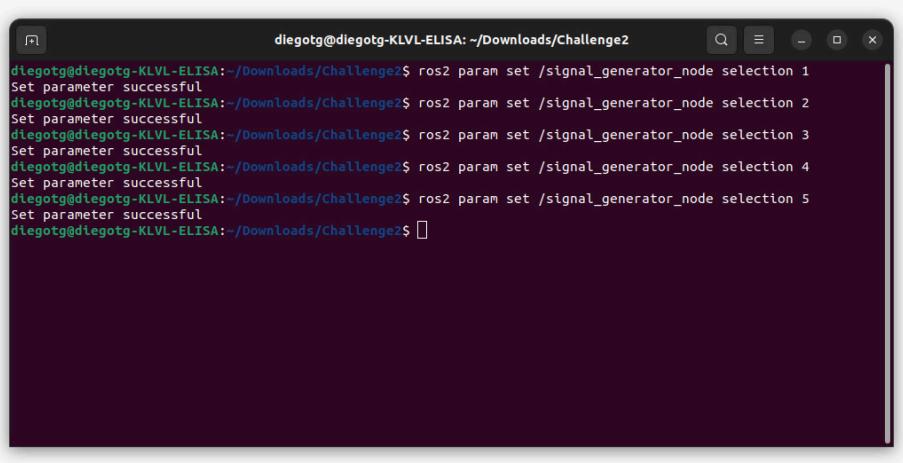
## ANÁLISIS DE RESULTADOS

Como podemos observar, utilizando el comando launch vamos a tener nuestras dos terminales trabajando con valores de la señal generada en el nodo “signal\_generator\_node”, así como los valores arrojados por el nodo ‘reconstruction’ en la terminal del lado izquierdo. Inicialmente, nuestro programa graficara la señal senoidal, esto gracias a la manera en la que definimos y declaramos los parámetros en el archivo .yaml.

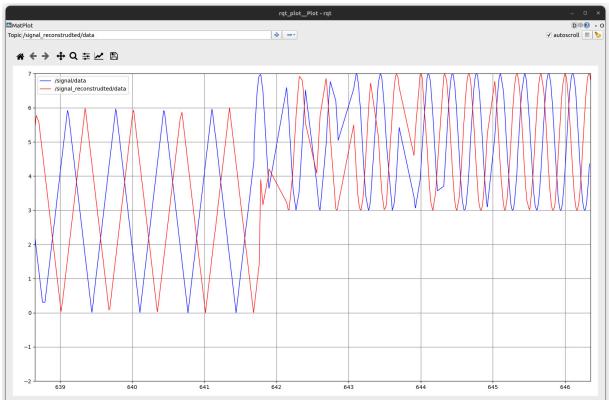
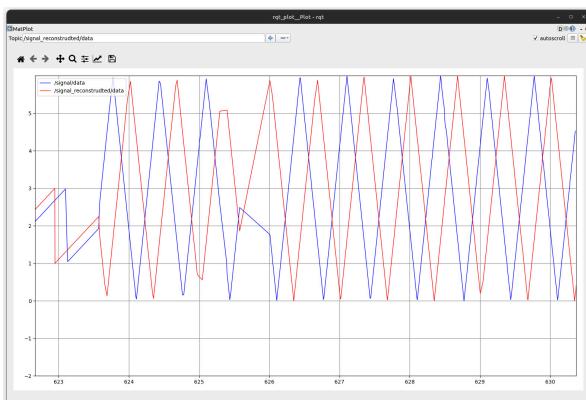
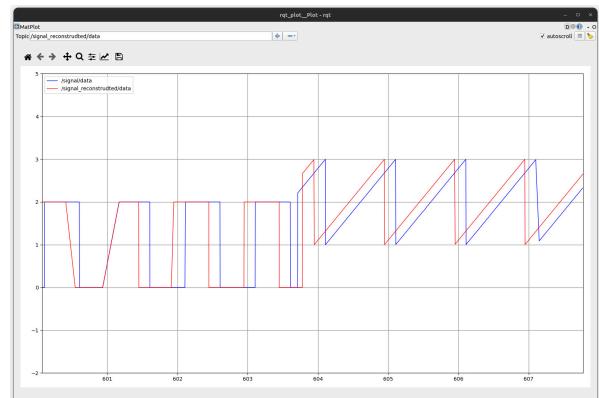
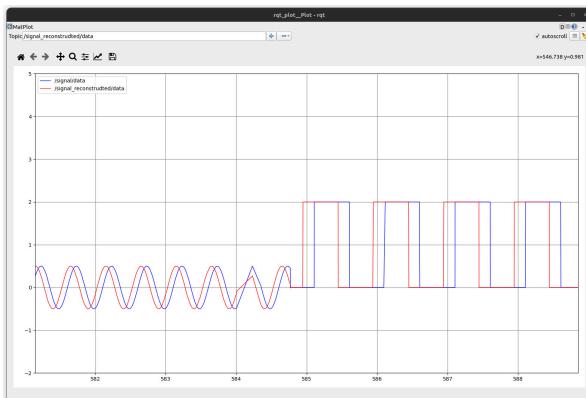
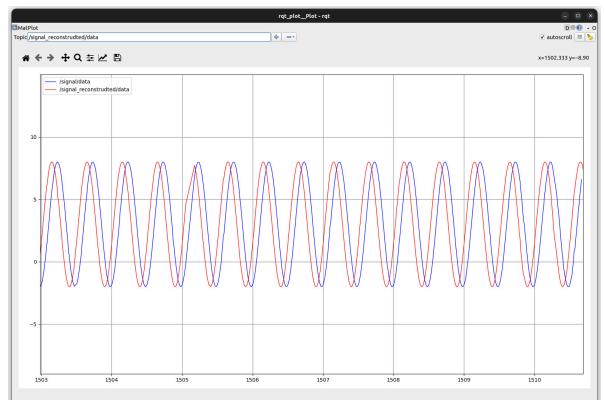


A continuación, iremos mostrando el funcionamiento del programa tras modificar los distintos parámetros con los que cuenta nuestro nodo:

## ● Selection



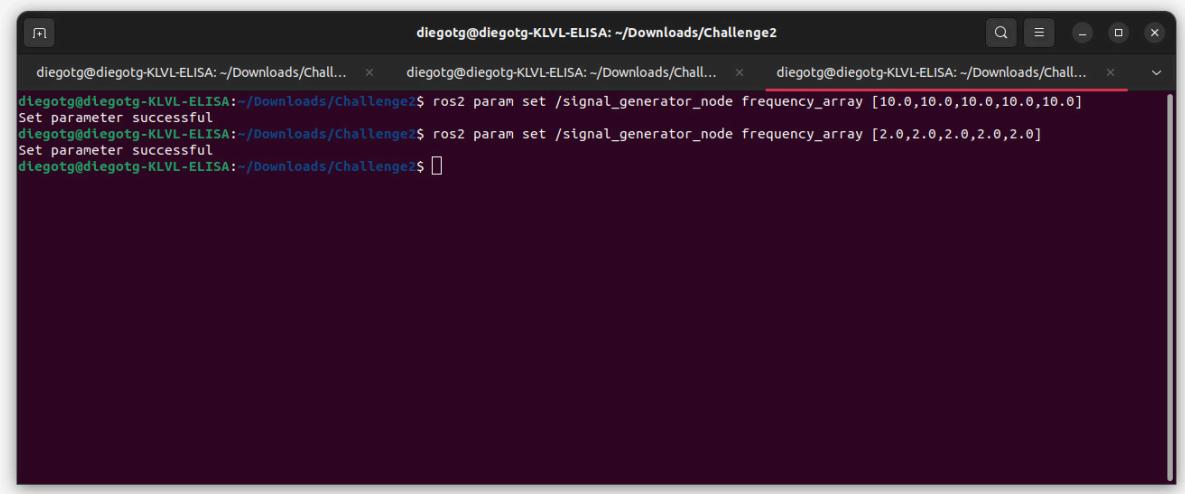
```
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node selection 1
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node selection 2
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node selection 3
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node selection 4
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node selection 5
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$
```



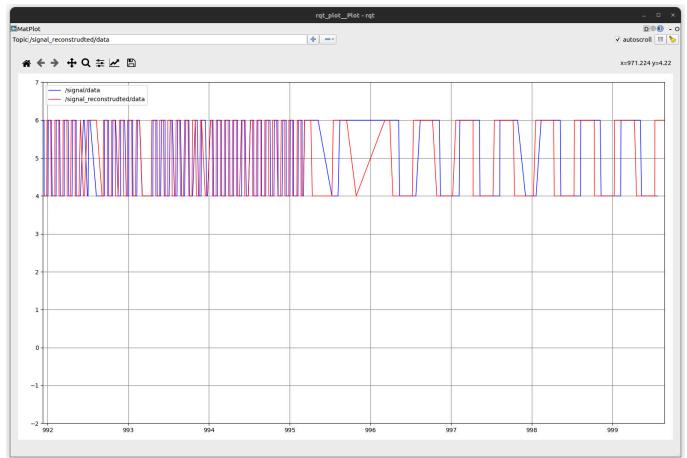
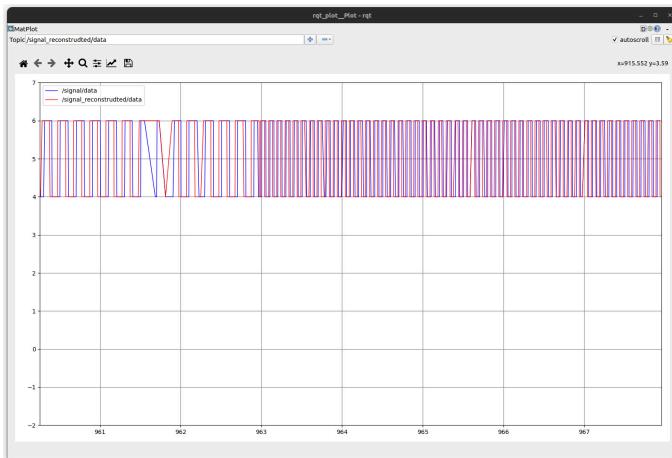
Primero vamos a tener el uso de Selection lo cual desde la terminal vamos a poder ejecutar el comando de “`ros2 param set /signal_generator_node selection`”. Con ello pondremos el número de la señal que vamos a querer representar quedando de la siguiente forma:

1 = Sine , 2 = Square , 3 = Sawtooth, 4 = Triangle, 5 = Cos. De esta manera vamos a poder ver cómo se inicia con la onda senoidal (1) y luego conforme se hace la selección de la señal se va cambiando dependiendo del número que haya ingresado.

- Frequency



```
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node frequency_array [10.0,10.0,10.0,10.0,10.0]
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node frequency_array [2.0,2.0,2.0,2.0,2.0]
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$
```

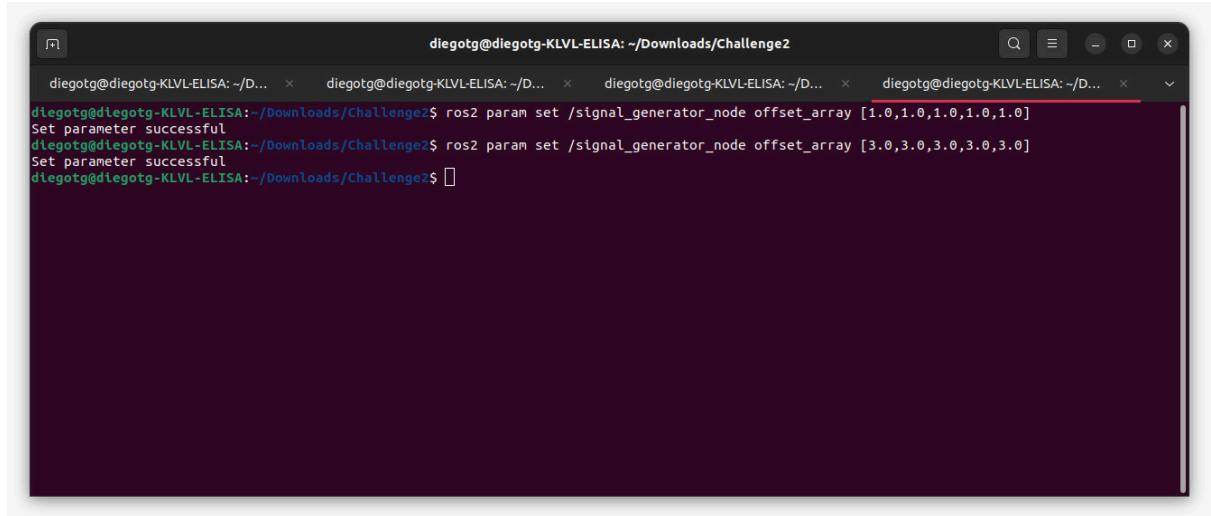


Explicando otro parámetro vamos a tener el de frequency en cual vamos a necesitar de un arreglo. En esta ocasión se probó hacerlo con la señal 2 que corresponde a square. Primeramente se asignó un frequency\_array de 10.0 para cada elemento, lo que ocasionó qué se indicará qué la frecuencia fuera constante en 10 Hz para cada ciclo, lo que significa que la señal va a completar 10 ciclos en un segundo, los ciclos estarán más juntos y se observa una señal comprimida.

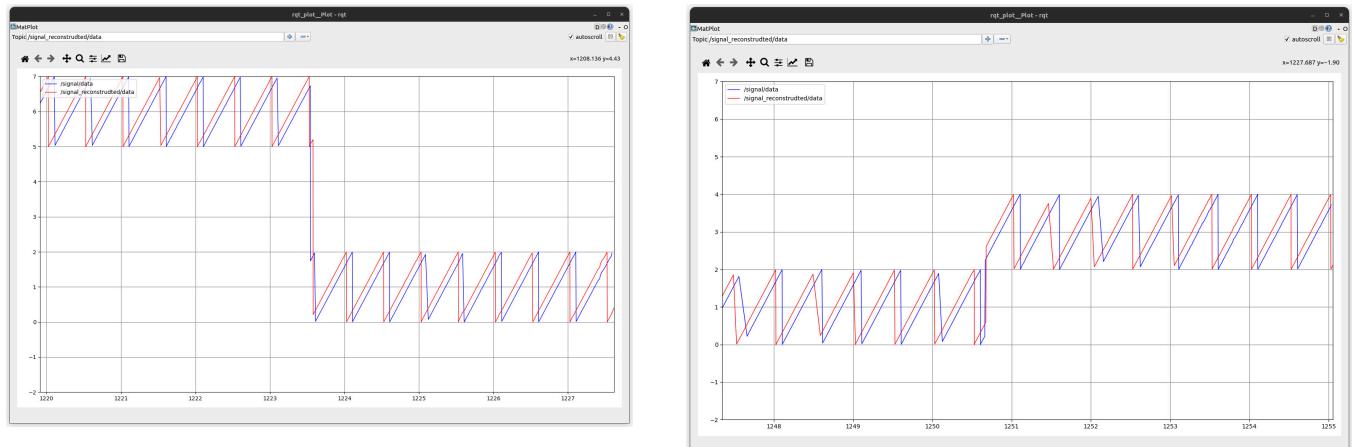
Ahora cuando usamos un array de frecuencia de [2.0,2.0,2.0,2.0,2.0].De esta forma, estamos indicando una frecuencia más baja de 2 Hz para cada ciclo, la señal tardará más tiempo en completar un ciclo y esto generará la impresión de que la señal se amplía. Tenemos que recalcar que el array que utilizamos desde este parámetro para los siguientes fue porque la terminal no solo podemos cambiar un elemento, esto por qué el array es de 5 elementos y

cada elemento representa una de las 5 selecciones de señales. Por esto mismo al cambiar un elemento diferente de un señal en la qué no estamos, no debería de afectar a nuestra gráfica, pero nosotros decidimos por temas de un mejor entendimiento para los ejemplos dejar el mismo valor en cada elemento del array.

- **Offset**



```
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node offset_array [1.0,1.0,1.0,1.0,1.0]
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node offset_array [3.0,3.0,3.0,3.0,3.0]
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$
```

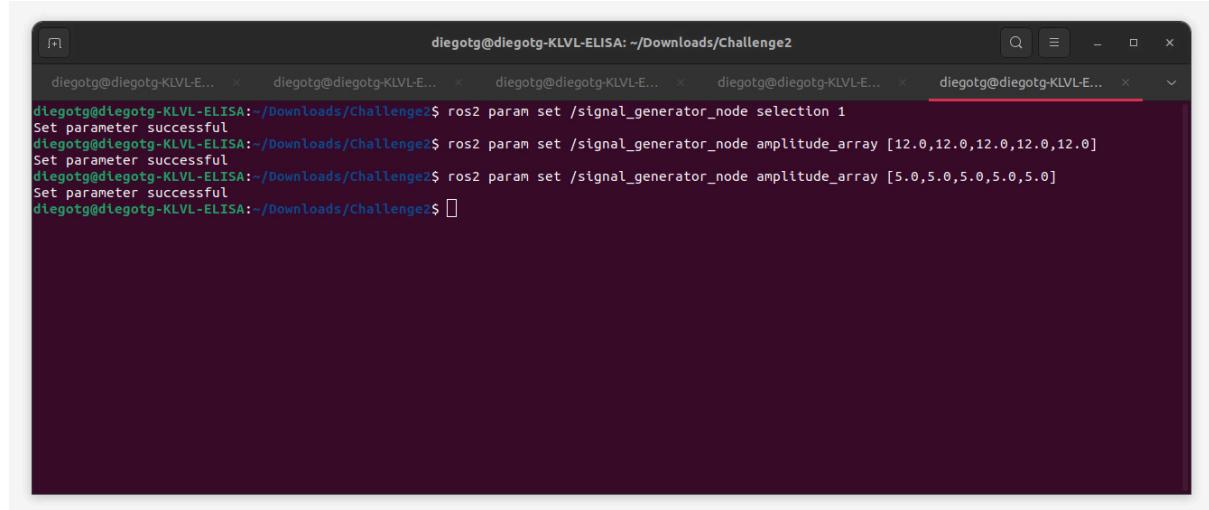


Ahora con el uso de offset vamos a observar cómo funciona este parámetro va a determinar el valor inicial de la señal antes de comenzar con su aumento lineal lo que afectará en manera la posición inicial qué tendrá la señal en el eje y. Por lo que podemos observar al principio pusimos un array de offset  $[1.0, 1.0, 1.0, 1.0, 1.0]$  indicaremos que la señal comenzará en 1.0 en el eje y antes de comenzar con su aumento lineal, la señal comenzará generando desde 1.0 centrada alrededor de 1.0 en el eje y.

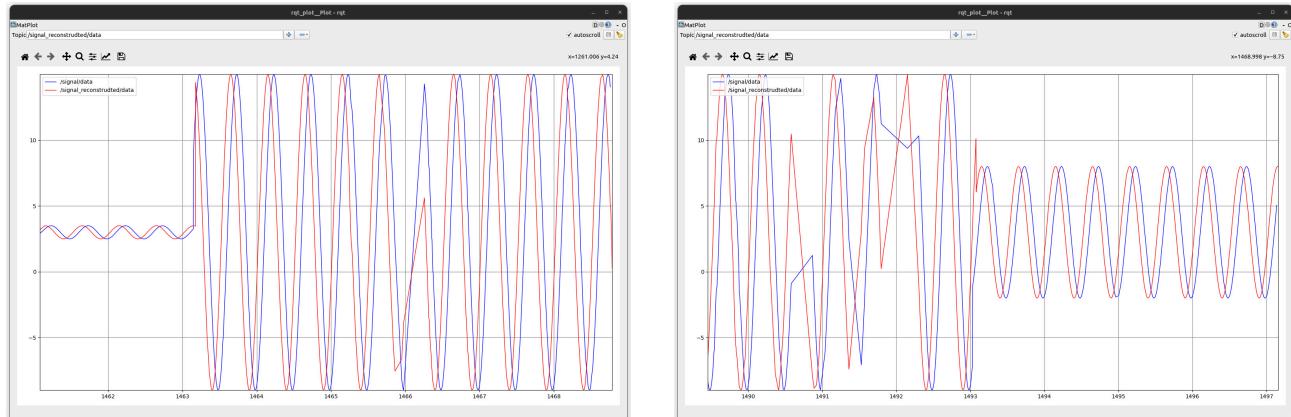
De otra forma usaremos un array de offset de  $[3.0, 3.0, 3.0, 3.0, 3.0]$  indicamos que la señal comenzará en el eje y en 3.0, comenzando desde aquí por lo que podemos ver en las gráficas de cómo se hace el cambio, al principio la teníamos predeterminada en un punto pero al meterle el offset array de 1.0 vemos como bajo centrado en 1.0 en el eje y. Mientras que

cuando le pusimos el offset array de 3.0 observamos también el cambio que la señal subió para centrarse en 3.0 en el eje y.

## Amplitud



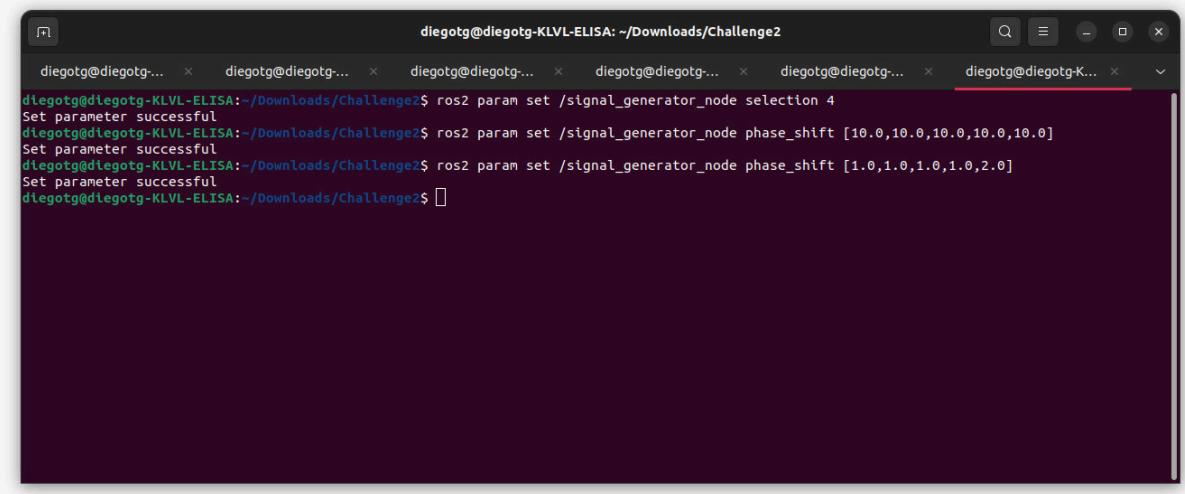
```
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2
diegotg@diegotg-KLVL-ELISA:~/Downloads/Challenge2$ ros2 param set /signal_generator_node selection 1
Set parameter successful
diegotg@diegotg-KLVL-ELISA:~/Downloads/Challenge2$ ros2 param set /signal_generator_node amplitude_array [12.0,12.0,12.0,12.0,12.0]
Set parameter successful
diegotg@diegotg-KLVL-ELISA:~/Downloads/Challenge2$ ros2 param set /signal_generator_node amplitude_array [5.0,5.0,5.0,5.0,5.0]
Set parameter successful
diegotg@diegotg-KLVL-ELISA:~/Downloads/Challenge2$ 
```



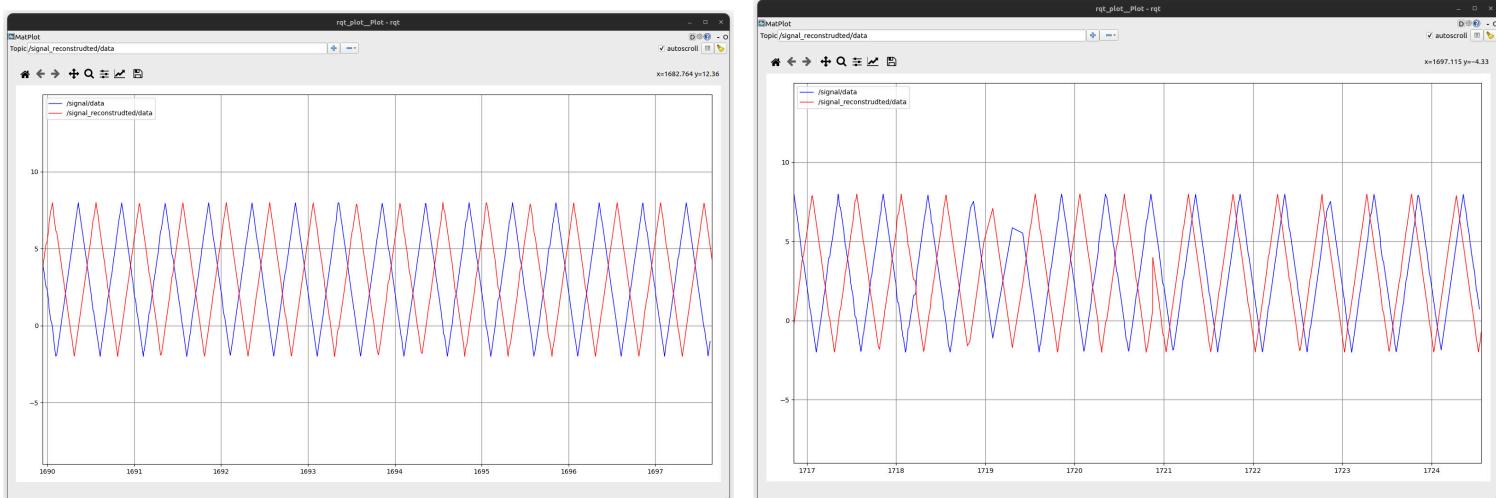
Tenemos el parámetro de amplitud para ajustarlo ahora en una señal senoidal lo cual va a modificar la magnitud de oscilación de la onda, la amplitud determinara la distancia máxima desde el punto medio de la onda hasta su punto más alto o bajo.

Cuando utilizamos un array de amplitud de [12.0,12.0,12.0,12.0,12.0] estamos indicando qué la amplitud será de 12.0 para cada ciclo de la onda, significando que la distancia desde el punto medio de la onda será más alto resultando en ondas más grandes. Por otro lado con el array de 5.0 indicamos una amplitud más baja y en caso contrario observamos que resulte en unas ondas más pequeñas en la altura en comparación con las de 12.0.

## Phase Shift



```
diegotg@diegotg-... x diegotg@diegotg-... x diegotg@diegotg-... x diegotg@diegotg-... x diegotg@diegotg-... x diegotg@diegotg-... x diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node selection 4
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node phase_shift [10.0,10.0,10.0,10.0,10.0]
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$ ros2 param set /signal_generator_node phase_shift [1.0,1.0,1.0,1.0,2.0]
Set parameter successful
diegotg@diegotg-KLVL-ELISA: ~/Downloads/Challenge2$
```

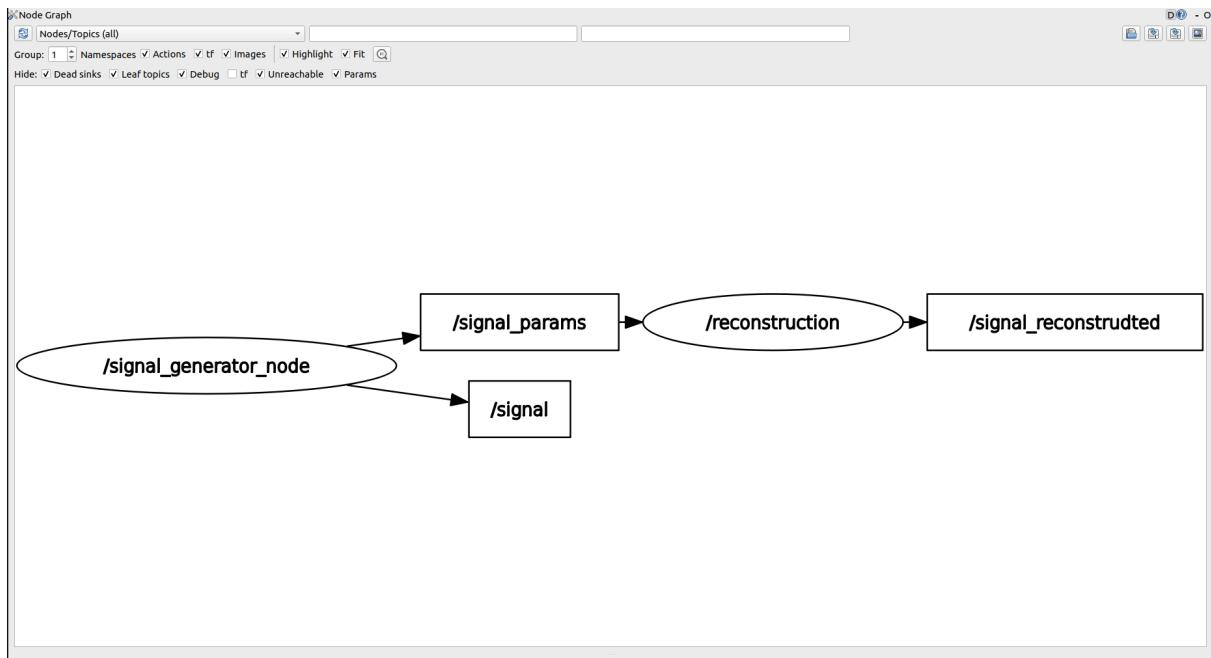


Finalmente vamos a ajustar nuestro parámetro de phase shift el cual se pueden observar como existe un desfase qué determina cuánto se adelanta o se va a retrasar la señal con respecto a la otra. Utilizando primero un array de desfase de 10.0 indicamos que existirá un desfase de 10 unidades de tiempo para cada ciclo de las señales, empezando el ciclo de la otra señal con un desfase y es por ello que observamos que no se ven juntas.

Por otro lado cuando utilizamos un array de 1.0 vamos a indicar que existirá un pequeño desfase ya no tan grande como el anterior, lo que en este caso las señales comenzaron casi simultáneamente observando en la gráfica que se ven mas pegadas entre ellas con solo una pequeña diferencia de tiempo entre ellas.

En este caso podemos observar lo que comentamos sobre los valores de cada elemento del array, ya qué en este ocasión tenemos seleccionada la señal triangular que es el número 4 y tenemos el segundo phase shift array de [1.0,1.0,1.0,1.0,2.0] teniendo un valor de 2.0 en el quinto elemento y esto no afectaría nuestro resultado debido a que solo esta tomando el valor que está en la cuarta posición del array debido a nuestra selección.

## Gráfico de nodos y tópicos:



## MODO DE EJECUCIÓN

Para probar los resultados podrás descargar el archivo .zip ubicado en este mismo repositorio. Una vez descomprimido, deberás usar los siguientes comandos:

```
source install/setup.bash  
ros2 launch signals_params project.launch.py
```

Para modificar los parámetros será mediante la siguiente estructura:

```
ros2 param set /signal_generator_node /param param_set
```

Es importante que el tipo de dato asignado corresponda con el tipo del parámetro.

## CONCLUSIONES

Los objetivos principales de la práctica fueron alcanzados con éxito, demostrando la capacidad de modificar la señal entre los nodos y logrando una comunicación efectiva entre ellos, así como la modificación de parámetros desde la terminal. Si bien la implementación de mensajes personalizados presentó ciertas dificultades, pudimos superarlas gracias a una investigación exhaustiva en foros y la documentación oficial de ROS 2.

La ventaja de utilizar YAML para recibir parámetros desde la terminal fue fundamental. Esta elección nos permitió simplificar y estandarizar el proceso, garantizando una representación clara y legible de los datos.

Encontramos que una posible solución o mejora a la metodología implementada sería proporcionar ejemplos gráficos más claros y detallados en la documentación oficial de ROS 2, especialmente en lo que respecta Custom messages. Esto ayudaría a comprender mejor el proceso y facilitaría su implementación en futuros proyectos. Además, seguir explorando el funcionamiento de YAML para la configuración de parámetros podría mejorar nuestros códigos futuros para realizar diferentes pruebas.

## REFERENCIAS

Documentación ROS 2 rodante. Implementing custom interfaces. Fecha de consulta: 29 de Febrero del 2024.

<https://docs.ros.org/en/rolling/Tutorials/Beginner-Client-Libraries/Single-Package-Define-And-Use-Interface.html>

Documentación MathWorks. ROS 2 Custom Message Support. The MathWorks, Inc. Fecha de consulta: 29 de Febrero del 2024.

<https://la.mathworks.com/help/ros/ug/ros2-custom-message-support.html>

Manchester Robotics. (2024). Robot Operating System - ROS [Diapositivas de PowerPoint]. [https://github.com/ManchesterRoboticsLtd/TE3001B\\_Robotics\\_Foundation\\_2024/blob/main/Week1/Slides/2%20-%20MCR2\\_ROS\\_BASICS.pdf](https://github.com/ManchesterRoboticsLtd/TE3001B_Robotics_Foundation_2024/blob/main/Week1/Slides/2%20-%20MCR2_ROS_BASICS.pdf)

Millán J. (26 de septiembre del 2022). *How to Use ROS 2 Parameters*. Foxglove. <https://foxglove.dev/blog/how-to-use-ros2-parameters>

Ramachandran R. (17 de abril del 2023). *Domain ID and Namespace in ROS 2 for Multi-Robot Systems*. Medium

<https://rragesh.medium.com/domain-id-and-namespace-in-ros-2-for-multi-robot-systems-9a939ae3fa40#:~:text=Namespaces%20and%20Domain%20IDs%20are,nodes%2C%20topics%2C%20and%20services.>

The robotics back end. *ROS2 Create Custom Message (Msg/Srv)*. Fecha de consulta: 29 de Febrero del 2024. <https://roboticsbackend.com/ros2-create-custom-message/>