UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Desarrollo y comunicación de nodos en ROS2 con nodos publicadores y suscriptores

Eugenio Ivorra

Objetivos

- En esta práctica, los objetivos son desarrollar una comprensión sólida y práctica de cómo se comunican los nodos en ROS2 mediante la creación y manipulación de nodos publicadores y subscriptores en Python, y aplicar este conocimiento usando el simulador turtlesim:
  - Comprender la estructura y el funcionamiento de los nodos publicadores y subscriptores en ROS2.
  - Crear un nodo publicador en Python que pueda interactuar con el simulador turtlesim para mover la tortuga.
  - Crear un nodo subscriptor en Python que pueda recibir y procesar mensajes de posición de la tortuga desde el simulador turtlesim.
  - Aprender a definir y añadir dependencias de mensajes en el fichero `package.xml`.
  - Integrar y aplicar conceptos de publicadores y subscriptores en ROS2 mediante actividades prácticas y experimentación con diferentes tipos de mensajes y topics.

Índice general

1 Nodo Publicador

1.1 Crear paquete

Para crear un nodo en ROS2, es necesario tener un paquete. Los paquetes permiten separar el código en bloques reutilizables, donde cada paquete es una unidad independiente. Por ejemplo, podríamos tener un paquete para gestionar una cámara, otro para las ruedas del robot y otro más para planificar el movimiento del robot en el entorno.

Vamos a crear un paquete de Python en el espacio de trabajo de ROS2 que creamos en la primera práctica. Navegaremos primero al directorio `src` de nuestro espacio de trabajo `cd ~/ros2_ws/src` y para crear un paquete, ejecutamos el siguiente comando:

```
ros2 pkg create mi_paquete_python --build-type ament_python --dependencies
↪  rclpy
```

Aquí, `mi_paquete_python` es el nombre de nuestro nuevo paquete, y estamos especificando que es un paquete de Python con dependencia de `rclpy`, la biblioteca de Python para ROS2.

Muchos archivos y carpetas se han creado en un nuevo directorio llamado `mi_paquete_python`. Puedes explorar estos archivos y carpetas usando un editor de texto, como Visual Studio Code tal y como se ve en la figura 1.1.

En el directorio `mi_paquete_python`, encontrarás otro directorio con el mismo nombre y dentro un archivo python llamado `__init__.py` que no tenemos que modificar. Al mismo nivel que ese archivo se añadirán tus nodos de Python.

Cada paquete de ROS2 contendrá un archivo `package.xml`. Este archivo contiene información del paquete y sus dependencias. Si decides compartir tu paquete con la comunidad o necesitas añadir una licencia comercial, deberás editar las etiquetas correspondientes en este archivo.

Para compilar el paquete, navega al directorio raíz de tu espacio de trabajo y ejecuta:

```
colcon build
```

Si encuentras errores durante la compilación relacionados con `setuptools`, podrías necesitar ajustar la versión de `setuptools`:

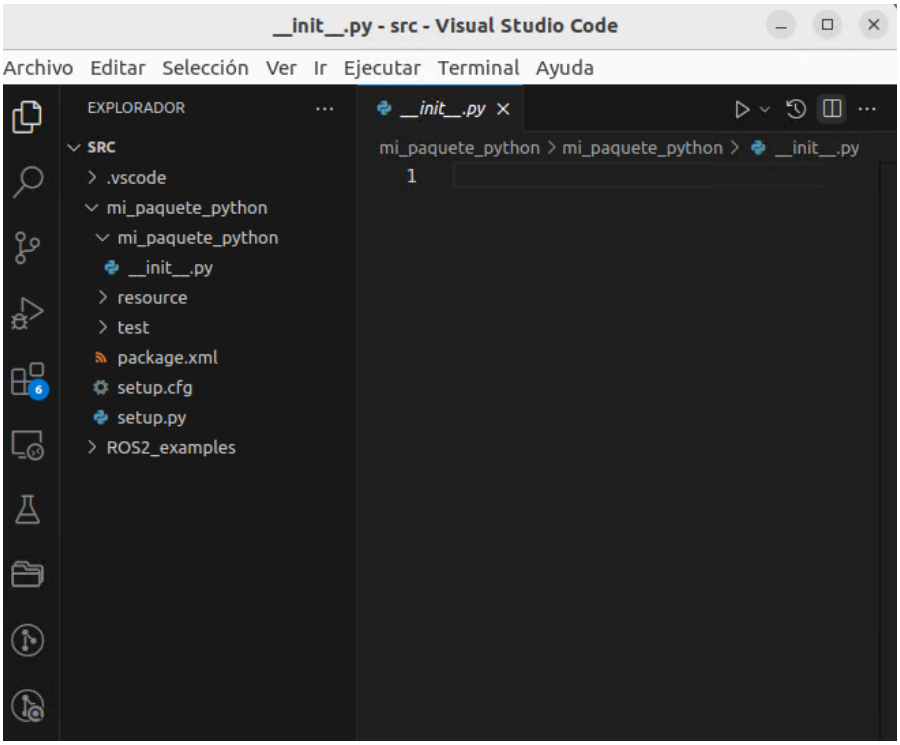


Figura 1.1: Estructura de un paquete abierto con el Visual Code

```
pip3 install setuptools==50.8.2.0
```

Una vez ajustada la versión, ejecuta nuevamente `colcon build`, y si todo funciona correctamente, verás un mensaje indicando que la construcción fue exitosa.

Finalmente, si deseas compilar un paquete específico, puedes usar:

```
colcon build --packages-select mi_paquete_python
```

Este comando es especialmente útil cuando tienes varios paquetes y deseas compilar uno específico para ahorrar tiempo de desarrollo.

Con estos pasos, tu paquete de Python en ROS2 está listo para albergar cualquier nodo de Python que desees desarrollar.

1.2 Estructura de un nodo publicador

Navega a `/ros2_ws/src/ROS2_examples/rclpy/topics/minimal_publisher/examples_rclpy_minimal_publisher`. Recuerda que este directorio es un paquete de Python con el mismo nombre que el paquete ROS 2 en el que está anidado y es el que probamos en la primera práctica.

Abre el archivo `publisher_member_function.py` usando tu editor de texto preferido (recomiendo Visual Code).

1.2.1 Examina el código

Las primeras líneas de código después de los comentarios importan la librería `rclpy` para que se pueda usar su clase `Node`.

```
import rclpy
from rclpy.node import Node
```

La siguiente declaración importa el tipo de mensaje string que el nodo usa para estructurar los datos que pasa en el topic.

```
from std_msgs.msg import String
```

Estas líneas representan las dependencias del nodo. Estas dependencias deben agregarse a `package.xml`, lo que comprobarás en la siguiente sección.

Luego, se crea la clase `MinimalPublisher`, que hereda de (o es una subclase de) `Node`.

```
class MinimalPublisher(Node):
```

A continuación, se define el constructor de la clase. `super().__init__` llama al constructor de la clase `Node` y le da el nombre de tu nodo, en este caso `minimal_publisher`.

`create_publisher` declara que el nodo publica mensajes del tipo `String` (importado del módulo `std_msgs.msg`), sobre un topic llamado `topic`, y que el "tamaño de la cola" es 10. El tamaño de la cola es una configuración de QoS (calidad de servicio) requerida que limita la cantidad de mensajes en cola si un suscriptor no los está recibiendo lo suficientemente rápido.

Luego, se crea un temporizador con un `callback` para ejecutar cada 0.5 segundos. `self.i` es un contador usado en el `callback`.

```
def __init__(self):
    super().__init__('minimal_publisher')
    self.publisher_ = self.create_publisher(String, 'topic', 10)
    timer_period = 0.5  # seconds
    self.timer = self.create_timer(timer_period, self.timer_callback)
    self.i = 0
```

`timer_callback` crea un mensaje con el valor del contador añadido, y lo publica en la consola con `get_logger().info`.

```
def timer_callback(self):
    msg = String()
    msg.data = 'Hello World: %d' % self.i
```

```
    self.publisher_.publish(msg)
    self.get_logger().info('Publishing: "%s"' % msg.data)
    self.i += 1
```

Finalmente, se define la función principal.

```
def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MinimalPublisher()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
```

Primero se inicializa la biblioteca `rclpy`, luego se crea el nodo y después se "hace girar" el nodo para que se llamen sus `callbacks` sin que termine el programa.

1.2.2 Añadir dependencias

Navega un nivel atrás al directorio donde se han creado para ti los archivos `setup.py`, `setup.cfg`, y `package.xml` al crear un paquete de python.

Abre `package.xml` con tu editor de texto.

Aquí son importantes llenar las etiquetas `<description>`, `<maintainer>` y `<license>`:

```
<description>Examples of minimal publisher/subscriber using
 ↪  rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

Después de las líneas anteriores, se añaden las siguientes dependencias correspondientes a las declaraciones de importación de tu nodo:

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

Esto declara que el paquete necesita `rclpy` y `std_msgs` cuando se ejecuta su código.

1.2.3 Añadir un punto de entrada

Abre el archivo `setup.py`. Nuevamente, ajusta los campos `maintainer`, `maintainer_email`, `description` y `license` a tu `package.xml`:

```
maintainer='YourName',
maintainer_email='you@email.com',
description='Examples of minimal publisher/subscriber using rclpy',
license='Apache License 2.0',
```

Para que se pueda ejecutar el nodo hay que indicar un nombre de ejecutable e indicarle qué fichero y función tiene que ejecutar. Esto se realiza con la siguiente línea dentro de los corchetes de `console_scripts` del campo `entry_points`:

```
entry_points={
        'console_scripts': [
                'publisher_member_function =
                ↪  'examples_rclpy_minimal_publisher.publisher \
                ↪  _member_function:main',
        ],
},
```

1.2.4 Verificar `setup.cfg`

El contenido del archivo `setup.cfg` debería estar correctamente rellenado de forma automática, de esta manera:

```
[develop]
script_dir=$base/lib/examples_rclpy_minimal_publisher
[install]
install_scripts=$base/lib/examples_rclpy_minimal_publisher
```

Esto simplemente le dice a `setuptools` que ponga tus ejecutables en `lib`, porque `ros2 run` los buscará allí.

Ahora vamos aplicar estos conocimientos en crear tu propio publicador en el paquete que hemos creado antes.

1.3 Crear nodo publicador

Ahora que ya tienes preparado el workspace y el paquete vamos a crear un nodo publicador con Python. Dirígete al directorio fuente de nuestro espacio de trabajo de ROS2.

```
cd ~/ros2_ws/src/mi_paquete_python/mi_paquete_python
```

A continuación crearemos un archivo llamado "robot_news_station.py". Este nodo se llamará robot_news_station y será responsable de publicar en una estación de noticias de robots. Haremos el archivo ejecutable con el siguiente comando:

```
# Es importante hacerlo ejecutable
chmod +x robot_news_station.py
```

A continuación se muestra un código base que puedes usar.

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node


class MyCustomNode(Node): # MODIFY NAME
    def __init__(self):
        super().__init__("node_name") # MODIFY NAME


def main(args=None):
    rclpy.init(args=args)
    node = MyCustomNode() # MODIFY NAME
    rclpy.spin(node)
    rclpy.shutdown()


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

Escribe el nodo para que se inicialice y cree un publicador y un temporizador. El temporizador llamará a una función llamada `publish_news` a 2 Hz. En concreto, el mensaje que dará es: *"Hi, this is C3PO from the robot news station"* en el topic `robot_news`.

Puedes lanzar el nodo, pero no hará nada visible hasta que agreguemos una línea para instalarlo en el archivo `setup.py`.

Después de configurar el archivo `setup.py`, necesitas volver a compilar el paquete e instalar el nodo.

Acuérdate también de actualizar el fichero `package.xml` con las nuevas dependencias necesarias.

```
colcon build --packages-select mi_paquete_python --symlink-install
source install/setup.bash
ros2 run mi_paquete_python robot_news_station
```

Ahora el nodo estará en ejecución y publicando mensajes en el topic "robot_news". Puedes verificar los topics activos y ver los mensajes publicados en "robot_news" utilizando las herramientas de línea de comandos de ROS2.

```
ros2 topic list
ros2 topic echo /robot_news
```

1.3.1 Publicar mensajes Twist para mover la tortuga

Para mover la tortuga del simulador `turtlesim` hay que publicar en el topic `'turtle1/cmd_vel'` un mensaje del tipo `twist`. Los cambios necesarios para poder usar este tipo de mensaje y como usarlo en el nodo se muestran aquí:

```
...
from geometry_msgs.msg import Twist
...

# Crear un objeto de tipo Twist
self.vel_msg = Twist()
...

#Para modificar la velocidad de avance:
self.vel_msg.linear.x= 1.0

#Para modificar la velocidad de giro:
self.vel_msg.angular.z= 1.0
...
```

Además, tendremos que añadir la dependencia `"geometry_msgs"` en el fichero `package.xml`.

1.4 ACTIVIDAD

Realiza cuatro nodos de ROS2 para mover las 4 tortugas que hiciste en la práctica anterior. Acuérdate de parar el movimiento de la tortuga al terminar de dibujar. Solo tienen que realizar el movimiento una vez, con lo que no es necesario programar un temporizador. A modo de recordatorio se muestra esta tabla resumen:

Nombre nodo	Forma geométrica a dibujar
mover_Leonardo	Cuadrado
mover_Raphael	Triángulo
mover_Michelangelo	Círculo
mover_Donatello	Rectángulo

2 Nodo Subscriptor

2.1 Estructura de un nodo subscriptor

Abra el archivo /home/euivmar/ros2_ws/src/ROS2_examples/rclpy/topics/minimal_subscriber/examples_rclpy_minimal_subscriber/subscriber_member_function.py con su editor de texto. En estos ejemplos el publicador y el subscriptor se encuentran en paquetes distintos pero sería perfectamente posible que estuvieran en el mismo paquete.

2.1.1 Examine el Código

El código del nodo subscriptor es casi idéntico al del publicador. El constructor crea un subscriptor con los mismos argumentos que el publicador. Recuerde del capítulo anterior que el nombre del topic y el tipo de mensaje utilizados por el publicador y el subscriptor deben coincidir para permitirles comunicarse.

```
self.subscription = self.create_subscription(
    String,
    'topic',
    self.listener_callback,
    10)
```

El constructor del subscriptor y el callback no incluyen ninguna definición de temporizador, ya que no lo necesita. Su callback se llama tan pronto como recibe un mensaje.

La definición del callback simplemente imprime un mensaje de info en la consola, junto con los datos que recibió. Recuerde que el publicador define `msg.data = 'Hello World: %d' % self.i`

```
def listener_callback(self, msg):
    self.get_logger().info('Escuché: "%s"' % msg.data)
```

La definición de `main` es casi exactamente la misma, reemplazando la creación y ejecución del publicador con el subscriptor.

```
minimal_subscriber = MinimalSubscriber()
rclpy.spin(minimal_subscriber)
```

Dado que este nodo tiene las mismas dependencias que el publicador, el archivo `package.xml` será idéntico al publicador. Recuerda que en el fichero `setup.py` tiene que aparecer el punto de entrada para el nodo subscriptor

2.2 Crear nodo subscriptor

Vamos a crear un subscriptor en Python para escuchar a la estación de radio que acabamos de crear en el capítulo anterior.

En este caso, vamos a crear un nodo denominado `smartphone` y, dentro de él, vamos a crear un subscriptor de topics. Como podrían imaginar, un nodo que representa a un smartphone podría tener muchas funcionalidades, como leer y publicar el estado de la batería, enviar un mensaje de texto a un contacto, aumentar o disminuir el volumen del altavoz, etc. Así que, el subscriptor que creemos aquí será solo una parte de este nodo.

Dirígete al directorio fuente de nuestro espacio de trabajo de ROS2.

```
cd ~/ros2_ws/src/mi_paquete_python/mi_paquete_python
```

Aquí, vamos a crear un archivo llamado `smartphone.py` y lo haremos ejecutable para que podamos editar el archivo desde ahí.

```
chmod +x smartphone.py
```

Como puedes adivinar, usaremos la misma plantilla de nodos que para el publicador, así que será bastante fácil crear la clase base para el nodo que llamaremos `SmartphoneNode`. Y simplemente llamaremos al nodo `smartphone`. Este nodo `smartphone` se subscribirá al topic "robot_news" y, cada vez que se publique un mensaje en ese topic, el nodo imprimirá el contenido del mensaje recibido en el log, informando que lo "oyó". Además imprime en el log la información de que el nodo se ha inicializado.

Las dependencias son las mismas que el publicador con lo que no necesitamos modificar el fichero `package.xml`. Sin embargo, en el fichero `setup.py` tenemos que añadir una linea para añadir el nodo `smartphone` a los nodos instalables.

2.2.1 Compilación y Prueba

Una vez creado el subscriptor, regresa a tu espacio de trabajo ROS 2 y compila de nuevo.

Ahora, puedes ejecutar tu nodo publicador y tu nuevo nodo subscriptor y deberías poder ver los mensajes de la estación de noticias del robot en el log del `smartphone`.

```
ros2 run mi_paquete_python smartphone
```

¡Felicidades! Ahora tienes dos nodos que se están comunicando a través de un topic de ROS 2. Como puedes ver, una vez que comprendes cómo funcionan los topic, crear un publicador o subscriptor de topics dentro de un nodo realmente no es tan difícil.

2.3 ACTIVIDAD

Crea un nodo subscriptor que diga la posición que ocupa un tortuga de `turtlesim`. Investiga en qué topic se publica y qué tipo de mensaje espera utilizando las herramientas que vimos en la práctica anterior.

3 ACTIVIDAD: Controlador para turtlesim

3.1 Objetivo

El objetivo de este ejercicio es desarrollar un nodo de ROS2 en Python que controle la tortuga en turtlesim para que se mueva a una posición objetivo, definida por las coordenadas x y y. El nodo debe suscribirse a la posición actual de la tortuga y publicar comandos de velocidad para dirigirla a la posición deseada mediante un controlador proporcional.

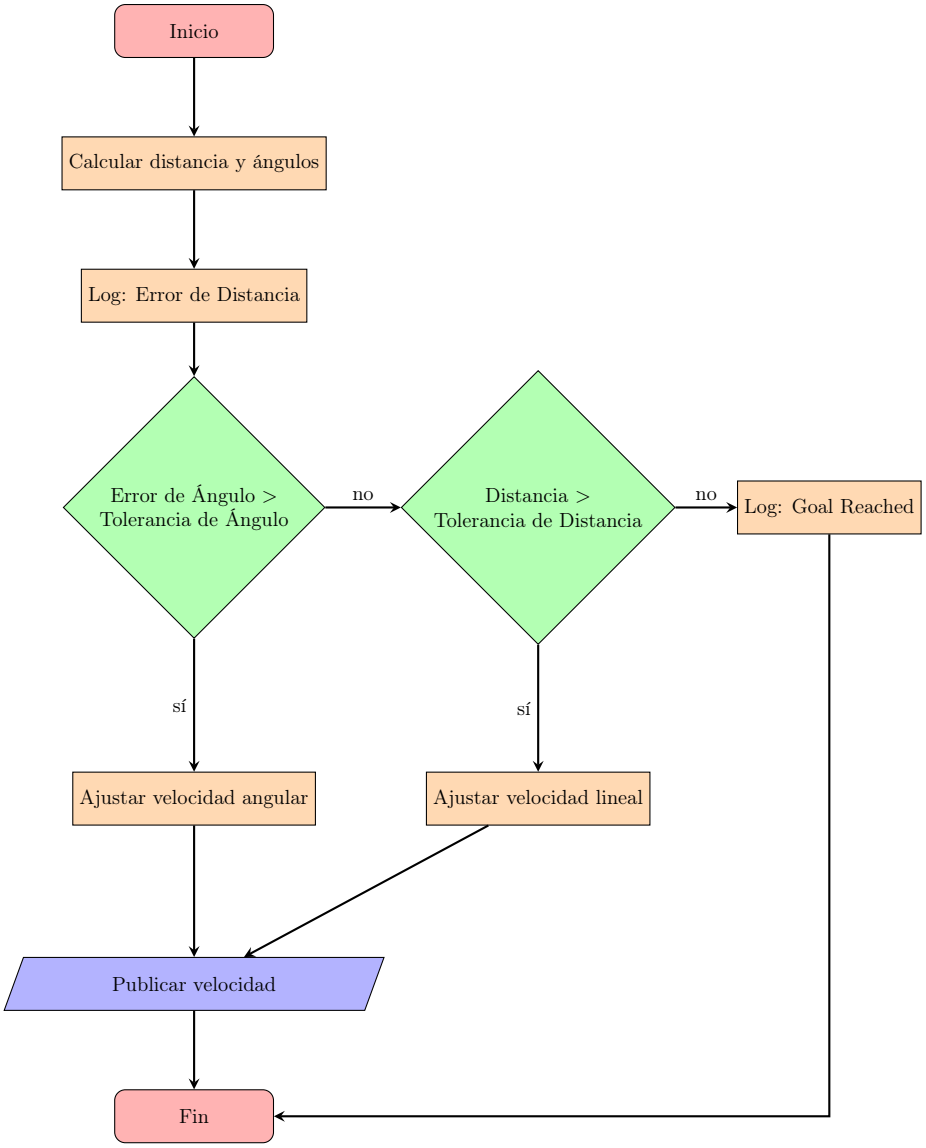
3.2 Especificaciones

El nodo debe llamarse `TurtleController` y debe estar suscrito al tópico `turtle1/pose` del tipo de mensaje `Pose` y publicar en el tópico `turtle1/cmd_vel` del tipo de mensaje `Twist`.

3.3 Controlador Proporcional para alcanzar una posición

3.3.1 Diagrama de Flujo

El siguiente diagrama representa el flujo de operaciones del controlador proporcional:



3.3.2 *Matemáticas del Controlador Proporcional*

El controlador proporcional a implementar en el código utiliza una constante proporcional *kp* para ajustar la velocidad lineal del robot de manera proporcional al error de distancia y para el error de ángulo. Para más detalles ver https://es.wikipedia.org/wiki/Control_proporcional.

$$\text{distance} = \sqrt{(x_{\text{goal}} - x_{\text{pose}})^2 + (y_{\text{goal}} - y_{\text{pose}})^2} \tag{3.1}$$

$$\text{angle\_to\_goal} = \arctan 2(y_{\text{goal}} - y_{\text{pose}}, x_{\text{goal}} - x_{\text{pose}}) - \theta_{\text{pose}} \tag{3.2}$$

Donde $x_{\text{pose}}, y_{\text{pose}}$ son las coordenadas actuales del robot y $x_{\text{goal}}, y_{\text{goal}}$ son las coordenadas deseadas. Para calcular las velocidades necesarias, solo nos queda multiplicar el error de distancia y de ángulo calculados por los valores de nuestro controlador. Ajustar los valores por ajuste experimental sabiendo que estarán en un rango desde 0.1 a 5.

$$\text{velocity\_message.linear.x} = kp \times \text{distance} \tag{3.3}$$

$$\text{velocity\_message.angular.z} = kp \times \text{angle\_to\_goal} \tag{3.4}$$

3.4 Instrucciones

1. Crea un archivo Python llamado `turtle_controller.py`.

2. Importa los paquetes y módulos necesarios.

3. Define una clase `TurtleController` que herede de `Node`. Dentro de esta clase, inicializa el nodo, los subscriptores y publicadores, y define los callbacks y funciones necesarias.

```
class TurtleController(Node):
    # ...
    def __init__(self):
        # ...
    # Función de callback del subscriptor
    def pose_callback(self, msg):
        # ...

    # Función de controlador
    def go_to_goal(self):
        goal = Pose()
        goal.x = float(sys.argv[1])
        goal.y = float(sys.argv[2])
```

4. Implementa la función `go_to_goal` para que la tortuga se mueva hacia el punto objetivo. Usa un controlador proporcional y ajusta el error de control y la constante proporcional según sea necesario. En este caso se ha modificado para que acepte entradas como argumentos por línea de comandos.

```
    def go_to_goal(self):
        goal = Pose()
        goal.x = float(sys.argv[1])
        goal.y = float(sys.argv[2])
```

5. Define la función `main` para inicializar `rclpy`, instanciar la clase `TurtleController` y llamar a `rclpy.spin` para mantener el nodo en ejecución.

6. Añade las dependencias necesarias y el punto de entrada a los ficheros `package.xml` y `setup.py`

7. Prueba con diferentes localizaciones y diferentes valores de ganancias del controlador.

Bibliografía

Tutorial turtlesim (2023). http://wiki.ros.org/turtlesim/Tutorials.

Web de los tutoriales de ROS (2023). https://docs.ros.org/en/humble/Tutorials.html.

Web de ROS2 Humble (2023). https://docs.ros.org/en/humble/index.html.