ROBOTS MÓVILES AUTÓNOMOS

Ingeniería Matemática e Inteligencia Artificial

Práctica 1: Primeros Pasos con ROS 2

Informe de Laboratorio

Autores:
Liam Esgueva González
Jorge Carnicero Príncipe
Andrés Gil Vicente

Fecha:
27 de enero de 2026

Índice

1. Conoce tus armas

Pregunta #1: ¿Cuál es la versión y el nombre de la distribución de ROS instalada en el contenedor que se le ha proporcionado? ¿Cuándo finaliza su soporte? ¿Por qué?

La versión instalada es **ROS 2** y la distribución es **Humble Hawksbill** (versión 2). El soporte para esta distribución finaliza en **mayo de 2027**. Esto se debe a que las distribuciones de ROS 2 liberadas en años pares tienen un soporte extendido de 5 años, mientras que las liberadas en años impares solo tienen soporte de 1.5 años. Humble fue lanzada en mayo de 2022, por lo que tendrá soporte hasta mayo de 2027.

Para verificar esta información se utilizan los comandos:

Código 1.0: Comandos para verificar la distribución de ROS 2

```
printenv ROS_VERSION
printenv ROS_DISTRO
```

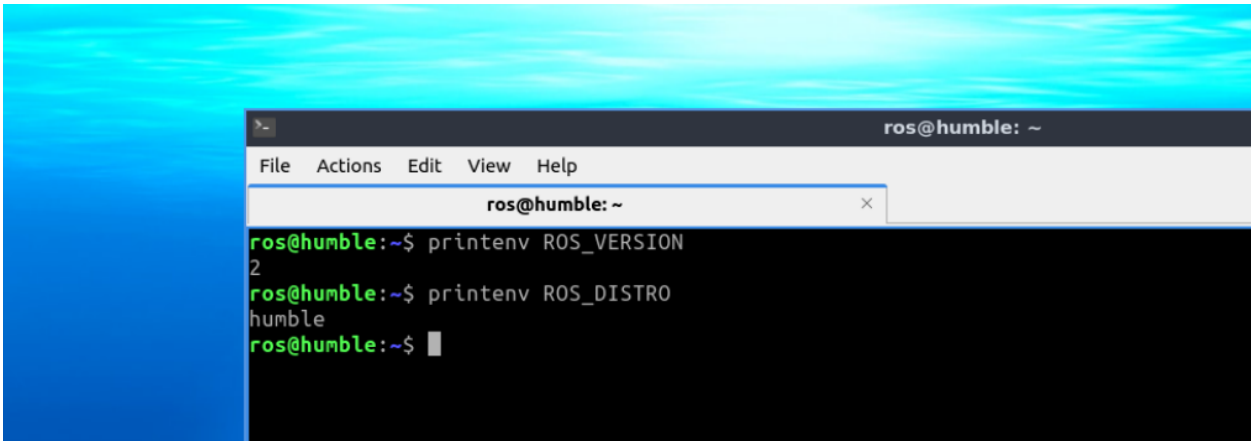

Figura 1: Verificación distribución de ROS2

2. ¿Dónde están las islas Galápagos?

Pregunta #2: ¿Qué otro comando se puede ejecutar con ros2 node? Escriba ros2 node-h para obtener ayuda.

Además de `ros2 node list`, otro comando disponible con `ros2 node` es `ros2 node info`, que muestra información detallada sobre un nodo específico, incluyendo sus publicadores, suscriptores, servicios y acciones.

Código 2.0: Uso de ros2 node info

```
# Para obtener ayuda general
ros2 node -h

# Para obtener información de un nodo específico
ros2 node info /turtlesim
```

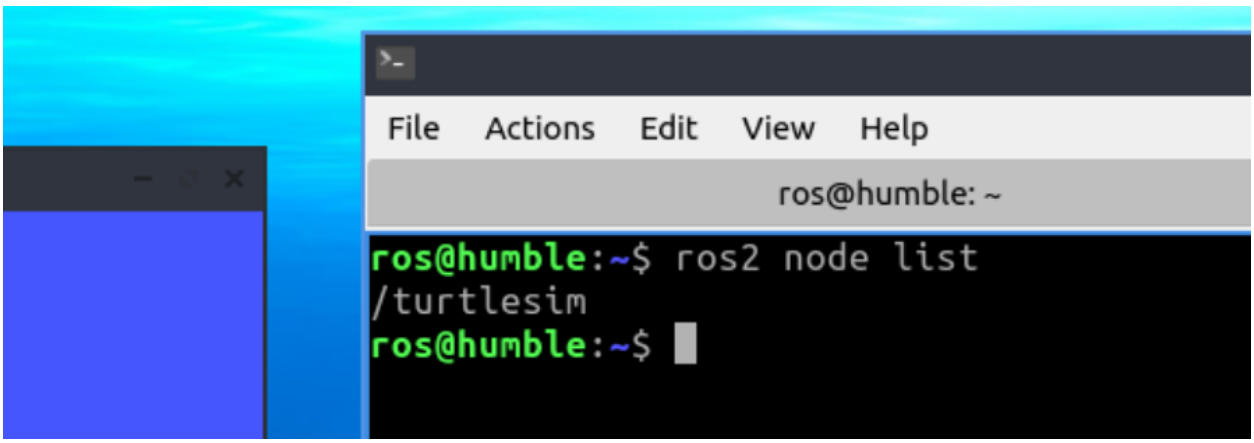

Figura 2: Listar todos los nodos en ejecución activos

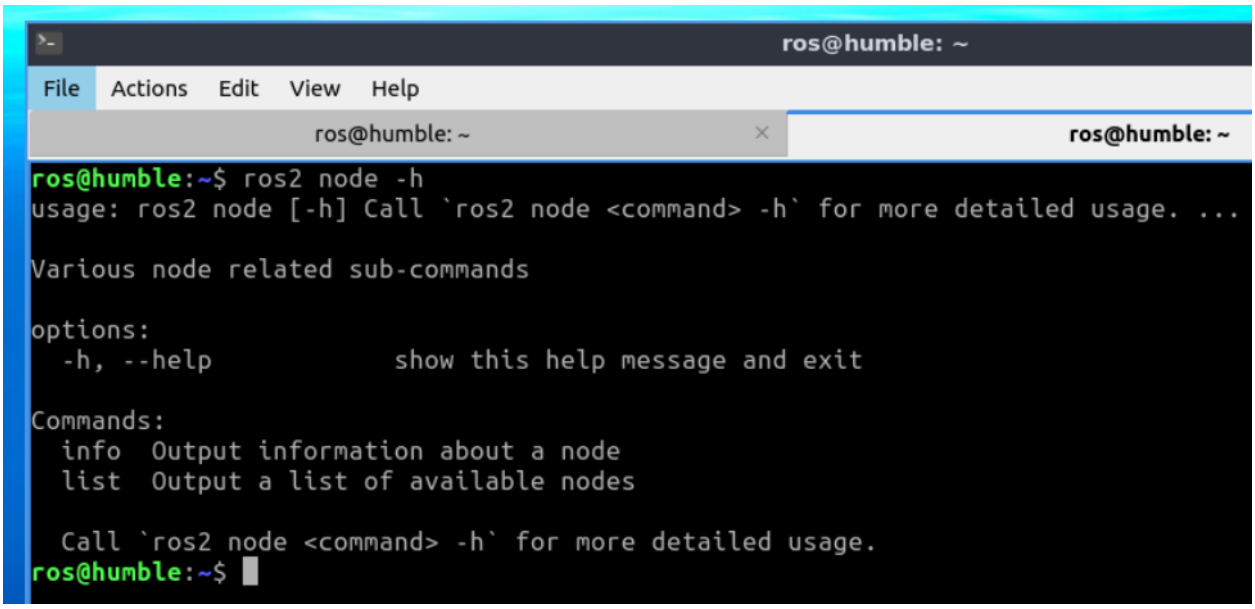

Figura 3: Pedir ayuda e información sobre un comando

Como se ve en la figura anterior, con el comando `ros2 node` se puede ejecutar tanto **info** como **list** .

3. El Maestro Astilla entrena a Michelangelo

Pregunta #3: ¿Cómo se llama el tema (*topic*) donde `/teleop_turtle` publica los comandos de movimiento? Adjunte una captura de pantalla de `rqt_graph` donde se vea el nombre.

El tema (topic) donde `/teleop_turtle` publica los comandos de movimiento se llama:

`/turtle1/cmd_vel`

En `rqt_graph`, la flecha sale del nodo `/teleop_turtle` (publicador) y entra en `/michelangelo` (suscriptor), pasando a través del topic `/turtle1/cmd_vel`.

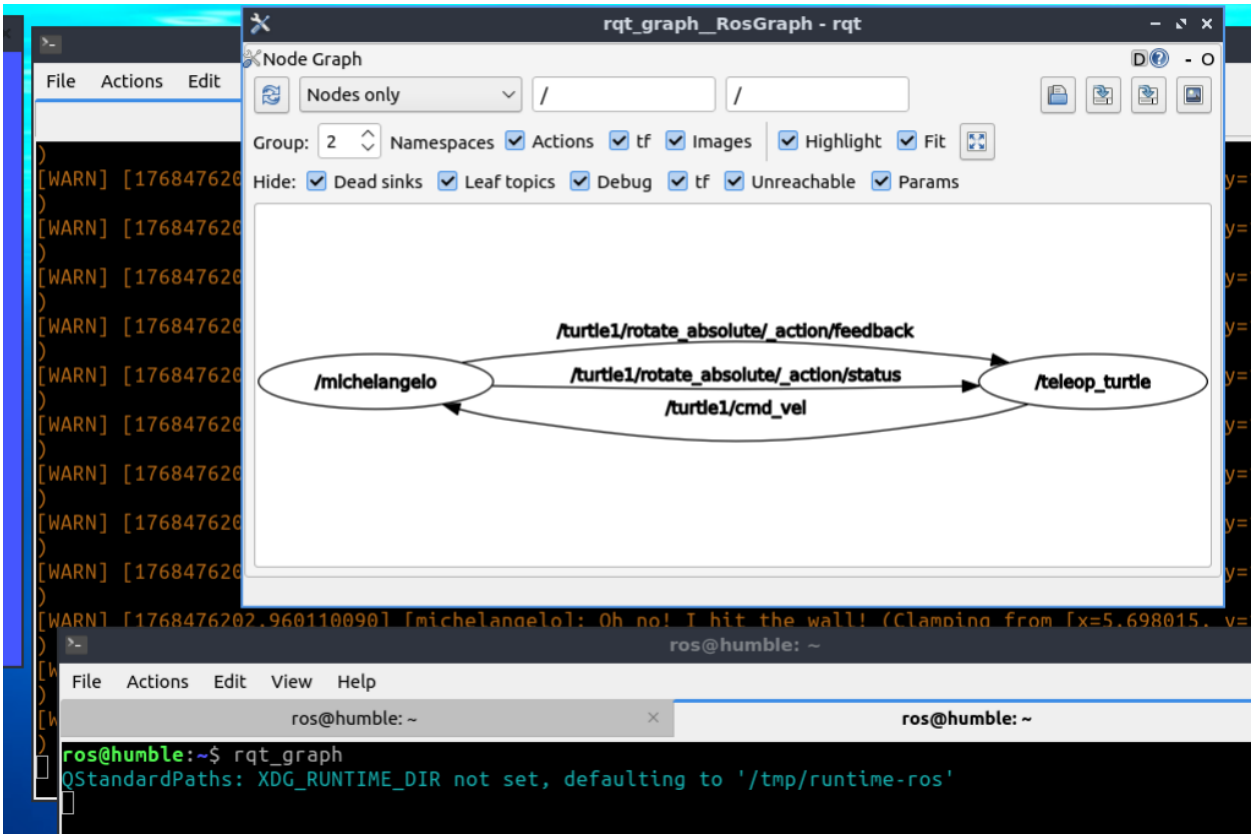

Figura 4: Grafo de relaciones entre nodos mostrando el topic `/turtle1/cmd_vel`

Como se ve en la figura anterior, para visualizar el grafo de relaciones se utiliza el siguiente comando:

Código 3.0: Comando para abrir rqt_graph

```
rqt_graph
```

Pregunta #4: ¿Qué recibe /michelangelo de /teleop_turtle?.

El nodo `/michelangelo` recibe del nodo `/teleop_turtle` mensajes de tipo:

`geometry_msgs/msg/Twist`

a través del topic:

`/turtle1/cmd_vel`

Estos mensajes contienen información sobre la velocidad lineal y angular. Para visualizar estos mensajes en tiempo real:

Código 3.1: Visualización de mensajes en el topic

```
ros2 topic echo /turtle1/cmd_vel
```

El resultado muestra una estructura como la que se ve en la siguiente figura:

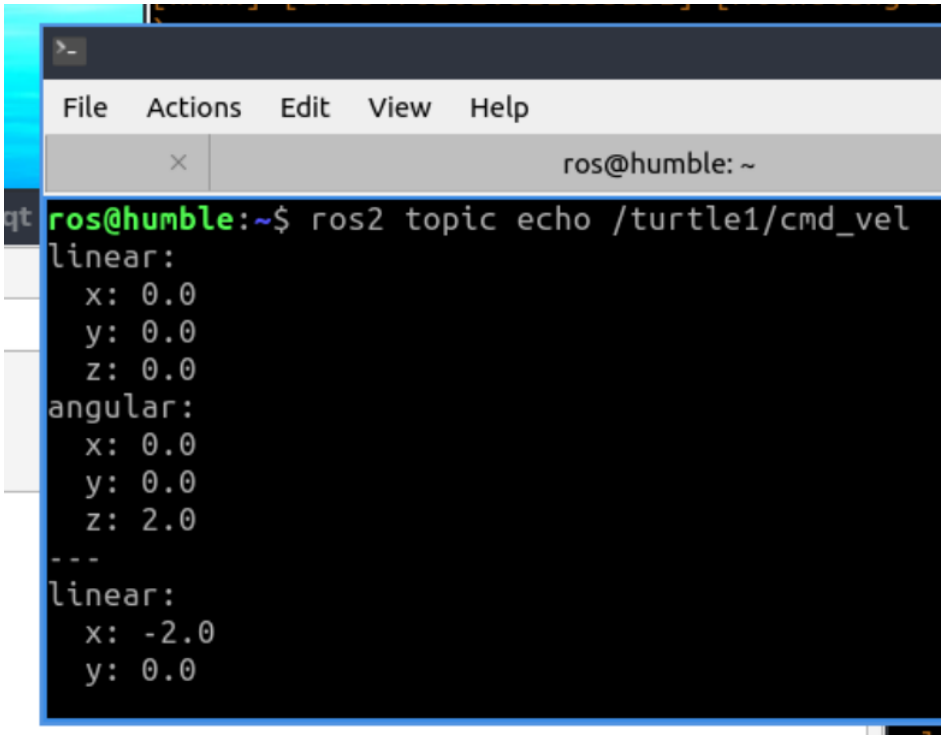

Figura 5: Estructura mensaje de tipo Twist

Donde `linear.x` representa la velocidad lineal de avance (0.0 m/s y luego -2.0 m/s en este caso) y `angular.z` representa la velocidad angular de giro (2.0 rad/s en este caso). Al pulsar las teclas en `teleop`, vemos que estos valores van cambiando y que la tortuga se va moviendo acorde con ellos.

Pregunta #5: ¿Qué tipos de datos se intercambian entre el nodo de teleoperación y TurtleSim?.

Los tipos de datos que se intercambian entre el nodo de teleoperación y TurtleSim son mensajes de tipo `geometry_msgs/msg/Twist`, que contienen dos estructuras de tipo `Vector3`:

- Para velocidades lineales (`linear`)
- Para velocidades angulares (`angular`)

Cada `Vector3` tiene tres componentes de números en coma flotante (x, y, z) que representan las velocidades en los tres ejes del espacio.

Para obtener la estructura completa del mensaje ejecutamos los siguientes comandos:

Código 3.2: Comandos para inspeccionar tipos de mensajes

```
# Ver el tipo de mensaje de un topic
ros2 topic info /turtle1/cmd_vel

# Ver la estructura interna del mensaje
ros2 interface show geometry_msgs/msg/Twist
```

Obteniendo el resultado que muestra la figura a continuación:

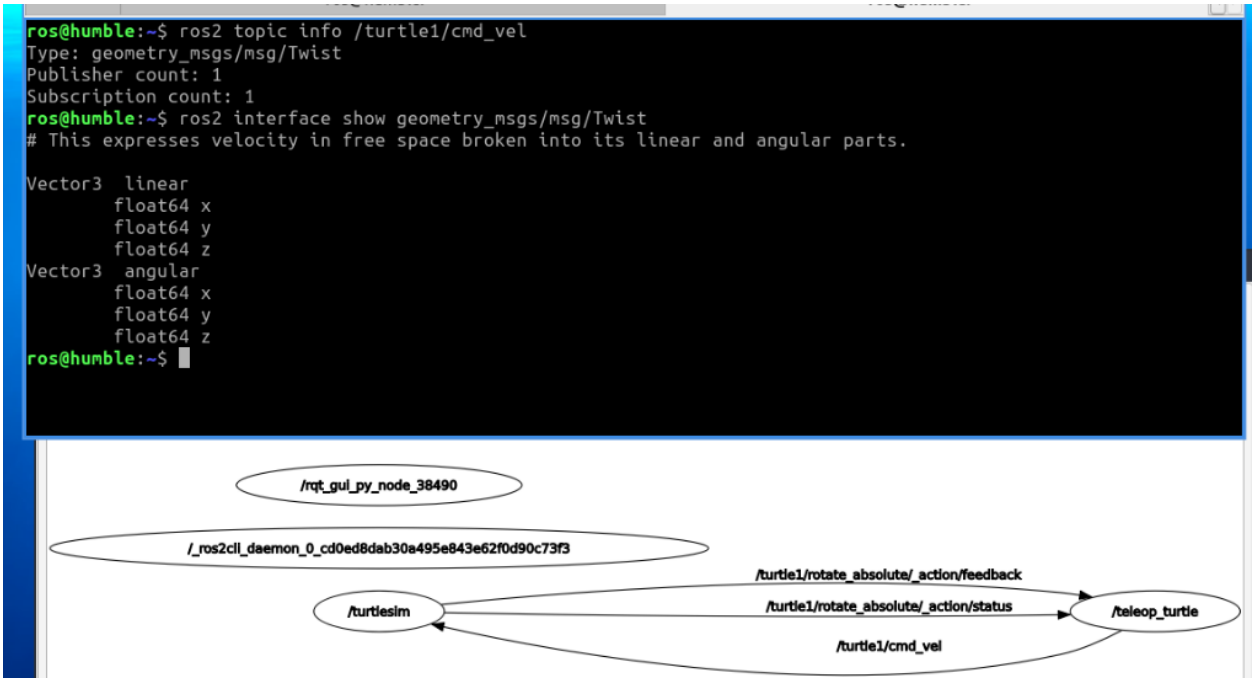

Figura 6: Interfaz del mensaje de tipo Twist

4. ¡Ey! ¿Qué pasa Raphael? Aquí Mikey

Pregunta #6: ¿Qué campos hay que completar en package.xml? ¿Qué es exactamente una licencia en este contexto? ¿Puede citar algunas licencias de software?.

En el archivo `package.xml` hay que completar los siguientes campos:

- `<description>`: Una breve descripción del propósito del paquete.
- `<maintainer>`: Email de la persona responsable del mantenimiento.
- `<license>`: La licencia bajo la cual se distribuye el código.

Una licencia en este contexto es el permiso legal que se concede sobre el código, definiendo qué pueden hacer otras personas con el paquete (usarlo, copiarlo, modificarlo, redistribuirlo) y bajo qué condiciones. Algunas licencias de software comunes son por ejemplo:

- Apache License 2.0
- MIT License
- BSD License (2-Clause / 3-Clause)
- GNU GPL v3 (General Public License)
- GNU LGPL

Además, en `package.xml` se especifican las dependencias de ejecución mediante etiquetas `<exec_depend>`, que indican los paquetes necesarios para ejecutar el código.

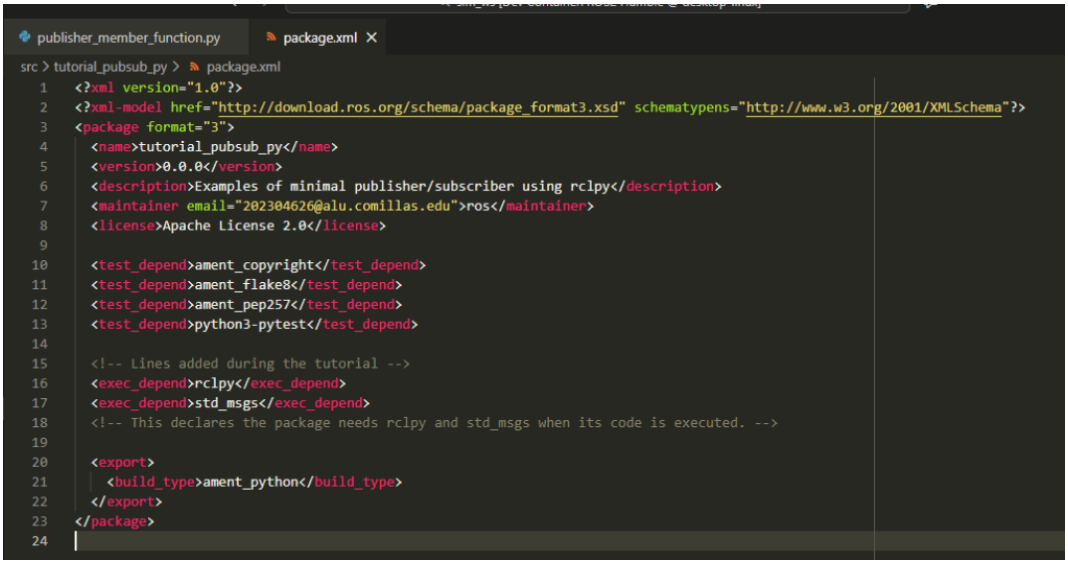

Figura 7: Ejemplo de package.xml

Pregunta #7: ¿Qué hay que añadir a setup.py?.

En el archivo `setup.py` hay que añadir los **entry points**, que registran cuáles son y cómo se llaman los nodos/scripts ejecutables que se podrán lanzar desde la terminal. Esto nos permite ejecutar los nodos con el comando:

```
ros2 run <paquete><ejecutable>
```

En este archivo también se deben completar los campos de `description`, `license` y `maintainer`, igual que en el `package.xml`. Veamos el ejemplo del tutorial propuesto para el publisher y el subscriber básicos:

Código 4.0: Configuración de entry points en setup.py

```
entry_points={
    'console_scripts': [
        'talker = tutorial_pubsub_py.publisher_member_function:main',
        'listener = tutorial_pubsub_py.subscriber_member_function:main',
    ],
},
```

Esto significa que línea dentro de `console_scripts` define un comando ejecutable que apunta a una función `main()` dentro de un módulo Python específico.

Pregunta #8: ¿Cuáles son los pasos para construir y ejecutar un nodo?.

Los pasos para construir y ejecutar un nodo son los siguientes:

1. Crear un paquete (si aún no se tiene ninguno creado):

Código 4.1: Creación de un paquete de python

```
ros2 pkg create --build-type ament_python --license Apache-2.0 py_pubsub
```

2. Descargar un ejemplo de plantilla de nodo:

En el tutorial se nos proporciona la plantilla tanto del publisher como del subscriber, pero también podríamos codificar nosotros mismos dichos scripts de python.

Código 4.2: Descarga de plantilla de ejemplo de nodo publisher

```
wget https://raw.githubusercontent.com/ros2/examples/humble/rclpy
↪ /topics/minimal_publisher/examples_rclpy_minimal_publisher/
↪ publisher_member_function.py
```

3. Añadir las dependencias necesarias:

Como ya se ha mencionado anteriormente, debemos rellenar cierta información en el fichero `package.xml`. Un ejemplo de las líneas que se deben añadir o completar es el siguiente:

Código 4.3: Configuración de las dependencias en package.xml

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

# Añadir
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

También se ha comentado ya, que es necesario configurar correctamente los **entry points** en el fichero `setup.py`, así como rellenar coherentemente los campos necesarios:

Código 4.4: Configuración de setup.py

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

# Añadir
entry_points={
    'console_scripts': [
        'talker = tutorial_pubsub_py.publisher_member_function:main',
        'listener = tutorial_pubsub_py.subscriber_member_function:main',
    ],
},
```

También se debe comprobar que el contenido del fichero `setup.cfg` es el adecuado, concordando con los nombres que se estén utilizando.

4. Instalar las dependencias del workspace:

Una vez ya tenemos todo configurado, para ejecutar los nodos que hayamos creado debemos abrir una terminal y ejecutar los siguientes comandos:

Código 4.5: Instalación de dependencias con rosdep

```
rosdep install -i --from-path src --rosdistro humble -y
```

5. Compilar el paquete específico:

Código 4.6: Compilación del paquete con colcon

```
colcon build --symlink-install --packages-select tutorial_pubsub_py
```

6. En una nueva terminal, cargar el entorno del workspace:

Código 4.7: Source del workspace

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

7. Ejecutar el nodo deseado:

Código 4.8: Ejecución de nodos

```
# Ejecutar el publicador
ros2 run tutorial_pubsub_py talker

# Ejecutar el suscriptor (en otra terminal)
ros2 run tutorial_pubsub_py listener
```

Donde `ros2 run` es el subcomando para lanzar ejecutables, `tutorial_pubsub_py` es el nombre del paquete, y `talker` o `listener` son los nombres de los ejecutables según los habíamos definido en el `setup.py`.

Pregunta #9: En sus propias palabras, explique brevemente cómo se comunica el nodo parlante (talker) con el oyente (listener).

El nodo *talker* crea un objeto publisher especificando el topic por donde enviará los mensajes y el tipo de mensaje que va a enviar (por ejemplo, `String` o `Twist`). En el ejemplo del tutorial se utiliza un timer que ejecuta periódicamente una función de callback, pero en otros casos podría controlarse de forma diferente según se quiera. En cada llamada, la función de callback crea un mensaje del tipo especificado, le asigna contenido (lo rellena) y lo publica a través del publisher mediante el topic correspondiente.

Por otro lado, el nodo *listener* crea un objeto de suscripción especificando a qué topic desea suscribirse y qué tipo de mensajes espera recibir. Esto debe coincidir con lo que especificó el nodo `talker` ya que si no la comunicación no se producirá correctamente. Además, este nodo define también una función de callback que se ejecuta automáticamente cada vez que le llega un mensaje del `talker`. Esta función procesa el mensaje recibido, por ejemplo mostrándolo por pantalla mediante un log, o hace cualquier acción que se desee utilizando la información de dicho mensaje.

Para esto, también debemos configurar las dependencias en `package.xml` y los entry points en `setup.py`, de modo que ROS 2 pueda localizar y ejecutar ambos nodos sin problemas.

5. Tortugas a la carrera

Pregunta #10: Utilice las herramientas de terminal de ROS 2 y rqt para descubrir los nodos que operan el TurtleBot3, así como los publicadores y suscriptores que tienen. Explore también la estructura y el contenido de los mensajes e intente explicar con sus palabras para qué sirve cada nodo a partir de esa información. Ignore todos los temas (topics) que empiecen con el prefijo rcl_.

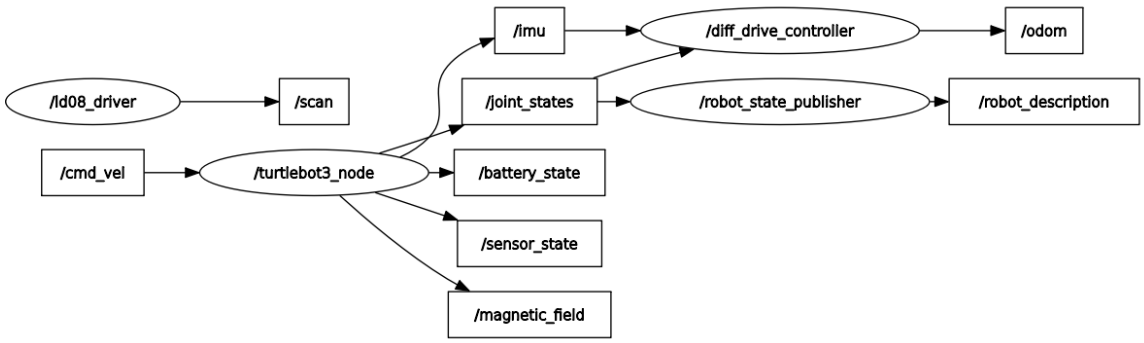

Figura 8: Grafo rqt de las relaciones del robot

En la figura descubrimos los nodos que operan el turtlebot y muchos de los topics correspondientes, entre los que cabe recalcar aquellos que usaremos sobre todo en esta práctica.

Estos son el topic $/scan$, donde el publisher es $/ld08\_driver$ que es el sensor del LiDAR, luego tenemos el $/turtlebot3\_node$ que está suscrito a $/cmd\_vel$, por donde le pasaremos la velocidad lineal en el eje x y la velocidad angular en el eje z. Él se encargará de dar la información necesaria a los motores para que se cumplan estas velocidades en el robot.

También vemos que tenemos varios topics más que pública este nodo como puede ser $/magnetic\_field$, $/battery\_state$, $/imu$ entre muchos otros.

A continuación exploramos un poco la estructura de los topics y el contenido de los mensajes:

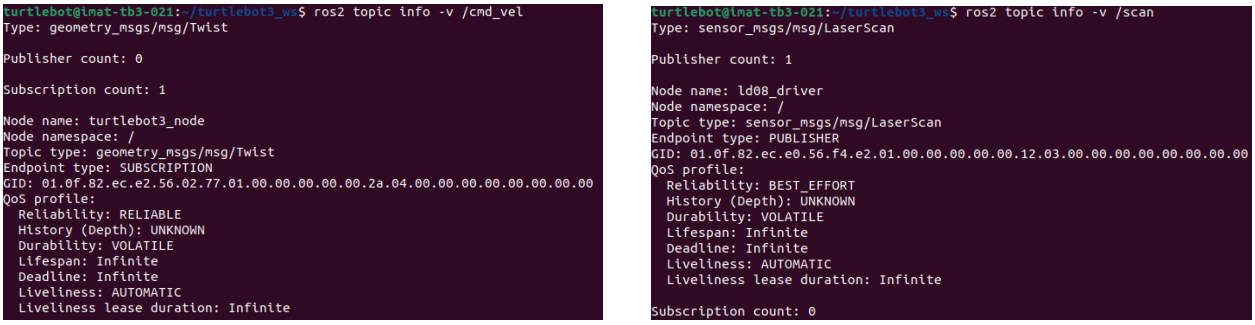

Figura 9: Información sobre topics del robot

Aquí vemos información de los 2 topics más importantes que usaremos en la práctica, que son los siguientes:

Por un lado está el topic de */cmd_vel*, en donde vemos que a través de él se pasa un mensaje de tipo Twist, y nosotros sabemos que tiene de argumentos linear y angular, pues cada uno de ellos tiene dentro las componentes (x,y,z). Además confirmamos que el endpoint es de subscripción, cosa que habiamos confirmado antes con el gráfico. También podríamos ver su qos (calidad de servicio) por si queremos ajustar el de nuestro publisher o lo necesitamos para cualquier cosa.

Por otro lado nos fijamos también en el topic de */scan*, ya que nos interesa a la hora de ver la información del LiDAR para poder hacer que el robot se pare en caso de ir a chocarse con alguna superficie u obstáculo. Vemos que tiene un Endpoint de tipo publisher desde el node *ld*08*_driver*, como habiamos confirmado nuevamente en el gráfico. El mensaje que se manda por este topic es de tipo *LaserScan*, en donde lo que más nos interesa es un atributo que tiene denominado ranges, donde nos saldrán las distancias que se van detectando por el LiDAR e irán ordenadas en función del índice.

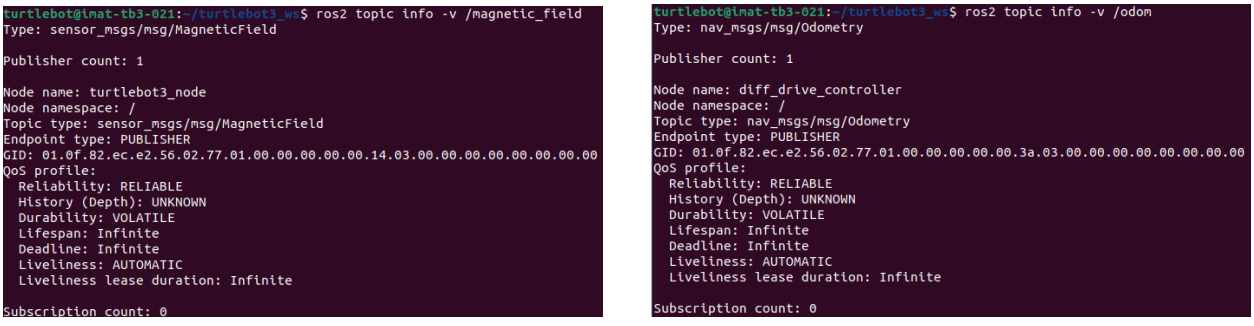

Figura 10: Información sobre topics del robot secundarios

Adicionalmente, también hemos querido buscar información extra sobre otros topics que no usaremos pero que son interesantes, como pueden ser los de */magnetic_field* y */odom*.

El topic `/magnetic_field` publica las medidas del magnetómetro en forma de vector (componentes x, y, z), representando el campo magnético detectado por el robot. Esta información puede usarse como referencia de orientación (tipo brújula) o para depuración/diagnóstico del estado de los sensores.

El topic `/odom` proporciona la odometría estimada del robot (posición y orientación) junto con velocidades lineales y angulares. Es un dato clave para estimar el movimiento en el entorno y para tareas como navegación o seguimiento de trayectoria.

Pregunta #11: ¿Qué criterio de signos utiliza el TurtleBot3 para la velocidad angular w por defecto? ¿Coincide con el habitual?

Código 5.0: Lanzar visualización y teleoperación

```
# Abrir RViz con configuración del TurtleBot3
ros2 launch turtlebot3_bringup rviz2.launch.py

# Ejecutar teleoperación por teclado
ros2 run turtlebot3_teleop teleop_keyboard
```

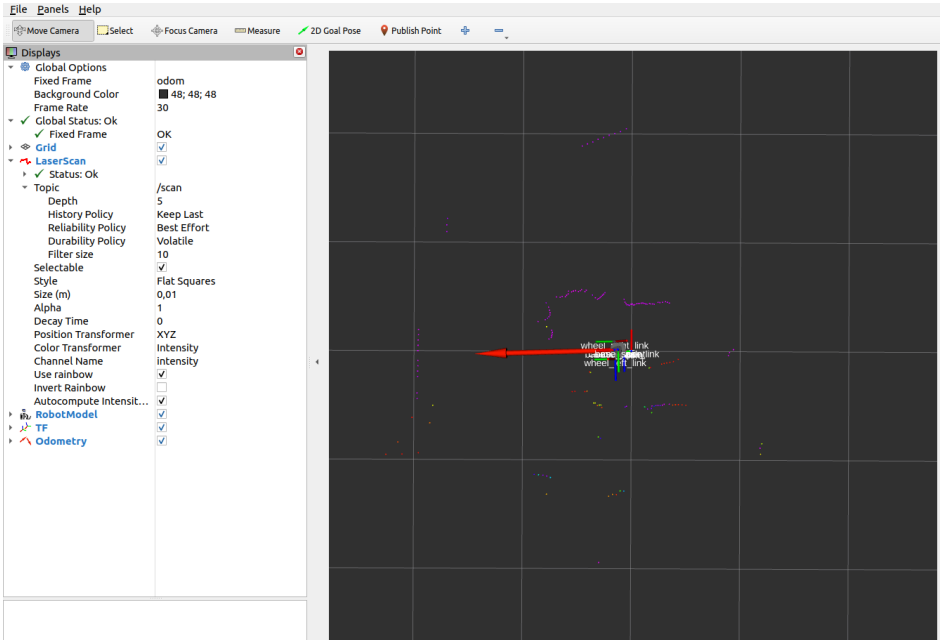

Figura 11: rvz view

En nuestro TurtleBot3, el signo de la velocidad angular del eje z, está invertido respecto a lo esperado. Al presionar la tecla 'd', esperamos un giro hacia la derecha pero vemos que el robot va hacia la izquierda, y con la tecla 'a' esperamos que gire hacia la izquierda pero va hacia la derecha.

Esto nos ha pasado igualmente a la hora de probar nuestro robot, ya que modificábamos la velocidad (mediante un diccionario que asocia teclas a valores de velocidad lineal en x y velocidad angular en z) de manera que teníamos:

```
"a": (0.0, 1.0),
"d": (0.0, -1.0),
```

Figura 12: Modificación de velocidad linea y angular en función de teclas 'a' y 'd'

Al darle a la 'a' sumábamos positivo, esperando movimiento a la izquierda y con la 'd' restábamos esperando un movimiento de giro hacia la derecha, por tanto nos encontramos con el siguiente criterio de signos:

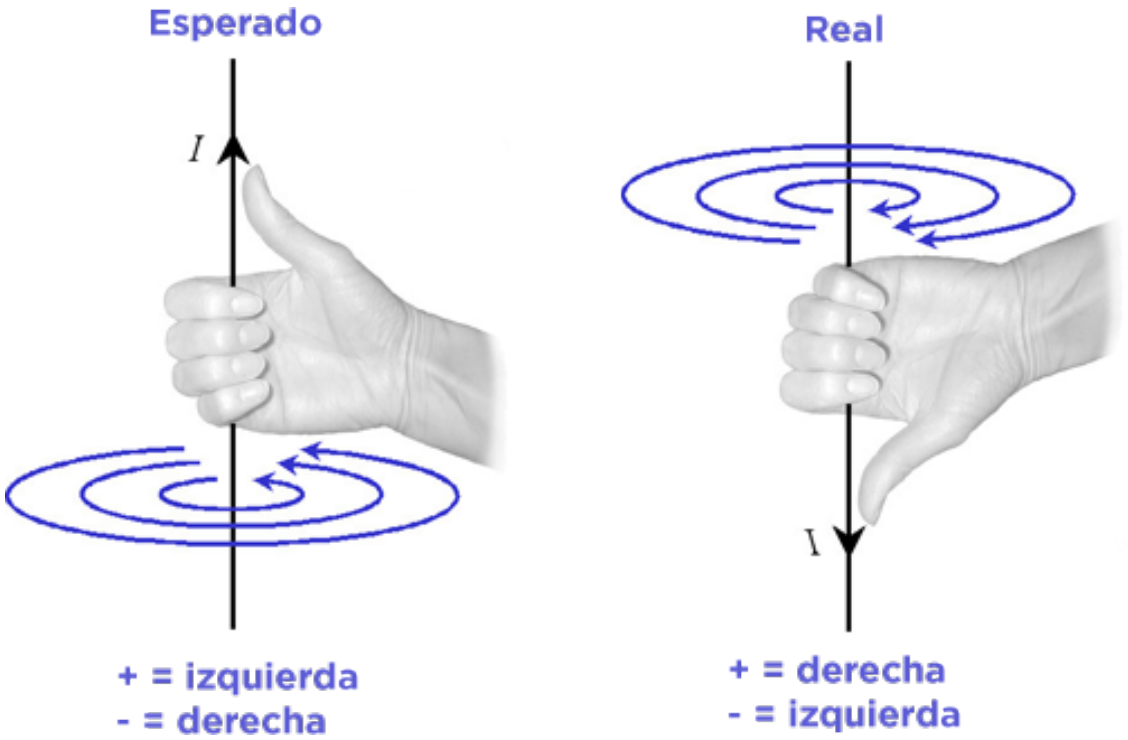

Figura 13: Criterio de signos de la velocidad angular

Al darnos cuenta de este criterio de signos, hemos modificado nuestro código de forma que funcione coherentemente e intuitivamente para el usuario.

Pregunta #12 (Apartado 5.4 - Sistema de teleoperación supervisada)

Implementación del paquete `amr_msgs`

Para la comunicación entre los nodos que ya hemos ido comentando, se ha creado el paquete `amr_msgs`, que contiene un mensaje personalizado llamado `KeyPressed`. Este mensaje tiene una estructura muy simple, conteniendo únicamente un campo de tipo `String` llamado `key`, que almacena el carácter de la tecla pulsada:

Código 5.1: Definición del mensaje KeyPressed.msg

```
string key
```

Este mensaje personalizado permite transmitir información entre el nodo que captura las pulsaciones del teclado (keyboard_node) y el nodo que las traduce en comandos de velocidad para el robot (teleoperation_node). La estructura del paquete `amr_msgs` sigue la organización estándar de ROS 2, con la definición del mensaje en el directorio `msg/` y las configuraciones necesarias en `package.xml` y `CMakeLists.txt`.

Nodo de captura del teclado: `keyboard_node.py`

El primer componente del sistema de teleoperación es el nodo `keyboard_node.py`, que se encarga de capturar las pulsaciones del teclado mediante la librería `sshkeyboard`, y publicarlas como mensajes personalizados (`KeyPressed`) a través del topic `listen_key`:

Código 5.2: Nodo keyboard_node.py

```python
import rclpy
from rclpy.node import Node
from amr_msgs.msg import KeyPressed
from sshkeyboard import listen_keyboard


class KeyPublisher(Node):
    """
    ROS2 node that listens for keyboard input and publishes pressed keys
    as KeyPressed messages on a topic.

    This node uses the sshkeyboard library to capture key presses from
    the terminal and forwards them into the ROS ecosystem.
    """

    def __init__(self):
        """
```

```
        Initializes the KeyPublisher node.

        Creates a ROS publisher that sends KeyPressed messages on the
        'listen_key' topic with a queue size of 10.

        Args:
            None

        Returns:
            None
        """
        # Initialize the ROS2 node with the name "websocket_key"
        super().__init__("websocket_key")

        # Create a publisher for KeyPressed messages on the "listen_key"
        ↪  topic
        self.publisher = self.create_publisher(
            KeyPressed,
            "listen_key",
            10,
        )

        # Internal flag that could be used to control execution
        self._running = True

    def on_key_press(self, key):
        """
        Callback function executed whenever a key is pressed.

        It creates a KeyPressed message, fills it with the pressed key,
        publishes it to the ROS topic, and logs the event.

        Args:
            key (str): The key detected by the keyboard listener.

        Returns:
            None
        """
        # Create a new ROS message
        msg = KeyPressed()

        # Store the pressed key in the message
        msg.key = key

        # Publish the message to the topic
        self.publisher.publish(msg)
```

```
        # Print information in the ROS logger
        self.get_logger().info(f"Publishing key: {msg.key}")


def main(args=None):
    """
    Main entry point for the ROS2 node.

    Initializes ROS, creates the KeyPublisher node, starts the keyboard
    listener, and spins the node manually until ROS is shut down.

    Args:
        args (list, optional): Command-line arguments passed to ROS.

    Returns:
        None
    """
    # Initialize the ROS2 communication layer
    rclpy.init(args=args)

    # Create the node instance
    node = KeyPublisher()

    # Start listening to the keyboard and bind key presses to the callback
    listen_keyboard(on_press=node.on_key_press)

    # Manual ROS loop that processes callbacks while ROS is running
    while rclpy.ok():
        rclpy.spin_once(node, timeout_sec=0.1)

    # Cleanly destroy the node before exiting
    node.destroy_node()

    # Shut down ROS
    rclpy.shutdown()


if __name__ == "__main__":
    # Run the main function only if the script is executed directly
    main()
```

Este nodo funciona como un puente entre la entrada del usuario por teclado y el sistema ROS 2. La función `listen_keyboard` de la librería `sshkeyboard` captura cada tecla presionada y ejecuta el callback `on_key_press`, que se encarga de empaquetar la información en un mensaje de tipo `KeyPressed` y publicarlo en el topic `listen_key` correspondiente.

Nodo de teleoperación supervisada: `teleoperation_node.py`

El segundo componente del sistema es el nodo `teleoperation_node.py`, que implementa la lógica de control del robot. Este nodo se suscribe tanto al topic de pulsaciones del teclado como al topic del LiDAR, y publica comandos de velocidad en `cmd_vel` para hacer que el robot se mueva según se indique por teclado:

Código 5.3: Nodo teleoperation_node.py - Parte 1: Configuración e inicialización

```
import math
import rclpy
from rclpy.node import Node
import numpy as np
from amr_msgs.msg import KeyPressed
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

from rclpy.qos import (
    QoSDurabilityPolicy,
    QoSHistoryPolicy,
    QoSProfile,
    QoSReliabilityPolicy,
)



# Maximum allowed angular velocity (rad/s)
MAX_ANGULAR_VEL = 2.81

# Maximum allowed linear velocity (m/s)
MAX_LINEAR_VEL = 0.22

# Distance in meters considered dangerous for obstacles
OBSTACLE_THRESHOLD = 0.25


# QoS profile for LiDAR sensor data:
# BEST_EFFORT is common for sensors because occasional drops are acceptable.
qos_lidar_profile = QoSProfile(
    history=QoSHistoryPolicy.KEEP_LAST,
    depth=10,
    reliability=QoSReliabilityPolicy.BEST_EFFORT,
    durability=QoSDurabilityPolicy.VOLATILE,
)

# QoS profile for velocity commands:
# RELIABLE ensures commands are delivered.
```

```
qos_cmdvel_profile = QoSProfile(
    history=QoSHistoryPolicy.UNKNOWN,
    reliability=QoSReliabilityPolicy.RELIABLE,
    durability=QoSDurabilityPolicy.VOLATILE,
)


class NodeMapper(Node):
    """
    ROS2 node that converts keyboard input into velocity commands and
    stops the robot if an obstacle is detected using LiDAR data.

    The node subscribes to:
        - 'listen_key' (KeyPressed): keyboard commands.
        - 'scan' (LaserScan): LiDAR ranges.

    And publishes:
        - 'cmd_vel' (Twist): velocity commands for the robot.
    """

    def __init__(self) -> None:
        """
        Initializes the NodeMapper node.

        Creates publishers and subscribers, defines the keyboard mapping,
        and initializes velocity state variables.

        Args:
            None

        Returns:
            None
        """
        # Initialize the ROS2 node with name "node_mapper"
        super().__init__("node_mapper")

        # Mapping of keyboard keys to linear and angular velocity increments
        # Format: key -> (linear_increment, angular_increment)
        self.map = {
            "w": (1.0, 0.0),
            "s": (-1.0, 0.0),
            "a": (0.0, -1.0),
            "d": (0.0, 1.0),
            "space": (0.0, 0.0),
        }

        # Current velocity state
```

```
        self.vel_lin = 0.0
        self.vel_ang = 0.0

        # Publisher for robot velocity commands
        self._publisher = self.create_publisher(
            msg_type=Twist,
            topic="cmd_vel",
            qos_profile=10,
        )

        # Subscriber for keyboard messages
        self._subscriber = self.create_subscription(
            msg_type=KeyPressed,
            topic="listen_key",
            callback=self.key_callback,
            qos_profile=10,
        )

        # Subscriber for LiDAR scan messages
        self.teleoperationLiDARSubs = self.create_subscription(
            msg_type=LaserScan,
            topic="scan",
            callback=self.AnalyzeLiDAR,
            qos_profile=qos_lidar_profile,
        )
```

Código 5.4: Nodo teleoperation_node.py - Parte 2: Análisis del LiDAR

```
def AnalyzeLiDAR(self, msg: LaserScan):

    """

    Process incoming LaserScan data to detect obstacles in front of and
    ↪  behind the robot.

    This callback estimates the nearest obstacle distance in two directions
    ↪  by taking the
    minimum valid range within an angular window around:
        - 0 radians (front direction)
        - pi radians (rear direction)

    Using a window (instead of a single ray) makes the detection more robust
    ↪  to noise,
    missing/invalid measurements (inf/nan), and small misalignments of the
    ↪  sensor.
```

```
        If the minimum distance in either window is below OBSTACLE_THRESHOLD,
        ↪ the node
        publishes a zero-velocity Twist command to stop the robot (only once
        ↪ when entering
        the blocked state) and sets internal flags to block further
        ↪ forward/backward commands.

        Args:
            msg (sensor_msgs.msg.LaserScan): LiDAR scan message containing
            ↪ ranges and scan geometry.

        Returns:
            None

        """

        # Number of range measurements in the scan (one per angle)
        N = len(msg.ranges)

        # Edge case: if there are no measurements or the angular increment is
        ↪ zero,
        if N == 0 or msg.angle_increment == 0.0:
            self.get_logger().warn("Empty scan or angle_increment=0")
            return

        # Helper function: returns the minimum valid distance within an angular
        ↪ window
        def min_range_around(target_angle_rad: float, half_window_deg: float =
        ↪ 10.0) -> float:
            half_window = math.radians(half_window_deg)

            # Index of the ray closest to the target angle:
            i_center = int((target_angle_rad - msg.angle_min) /
            ↪ msg.angle_increment)
            i_delta = max(1, int(half_window / msg.angle_increment))

            i0 = max(0, i_center - i_delta)
            i1 = min(N - 1, i_center + i_delta)

            # Search for the minimum valid range within the window
            best = float("inf")
            for i in range(i0, i1 + 1):
                r = msg.ranges[i]

                # Only consider finite readings within the sensor limits
                if math.isfinite(r) and (msg.range_min <= r <= msg.range_max):
                    if r < best:
```

```
                best = r

        # If no valid reading was found, this will return infinity
        return best

    # Distances estimated in front and behind as the minimum within a ±10°
    ↪  window
    dist_front = min_range_around(0.0, half_window_deg=10.0)
    dist_back  = min_range_around(math.pi, half_window_deg=10.0)

    # Store distances for possible use by other methods or for debugging
    self.distance_front = dist_front
    self.distance_back = dist_back

    # Informative log to observe the measured distances
    self.get_logger().info(f"Front: {dist_front:.3f} m | Back:
    ↪  {dist_back:.3f} m")

    # Determine whether there is danger by comparing with the safety
    ↪  threshold
    danger_front = dist_front < OBSTACLE_THRESHOLD
    danger_back  = dist_back  < OBSTACLE_THRESHOLD

    # Informative log to observe the velocities
    self.get_logger().info(f"Angular vel: {self.vel_ang}")
    self.get_logger().info(f"Linear vel: {self.vel_lin}")

    # If an obstacle is detected in front or behind
    if danger_front or danger_back:

        # If this is the first time entering a blocked state, publish a stop
        ↪  command
        if not self.obstacle_stop_front and not self.obstacle_stop_back:

            # Force commanded velocities to zero
            self.vel_lin = 0.0
            self.vel_ang = 0.0

            # Publish a zero-velocity Twist message to stop the robot
            stop_msg = Twist()
            stop_msg.linear.x = self.vel_lin
            stop_msg.angular.z = self.vel_ang
            self._publisher.publish(stop_msg)

        # Update blocking flags depending on where the obstacle is
        if danger_front:
            self.obstacle_stop_front = True
```

```
        else:
            self.obstacle_stop_back = True

    # If no obstacle is detected, clear both blocking flags
    else:
        self.obstacle_stop_front, self.obstacle_stop_back = False, False
```

Código 5.5: Nodo teleoperation_node.py - Parte 3: Procesamiento de teclas

```
    def key_callback(self, msg: KeyPressed) -> None:
        """
        Callback executed when a KeyPressed message is received.

        Extracts the pressed key and forwards it to the key processing
        ↪  logic.

        Args:
            msg (KeyPressed): Incoming keyboard message.

        Returns:
            None
        """
        key = msg.key

        # Allow backward motion only if there is no obstacle behind
        if not self.obstacle_stop_back and key == "s":
            self.process_key(key)

        # Allow forward motion only if there is no obstacle in front
        elif not self.obstacle_stop_front and key == "w":
            self.process_key(key)

        # Any other key (like turning) is always allowed
        elif key not in ["s", "w"]:
            self.process_key(key)


    def process_key(self, key):
        """
        Processes a keyboard key and updates the robot velocity.

        Applies incremental changes to the linear and angular velocities,
        enforces maximum limits, and publishes Twist commands.

        Args:
```

```
            key (str): Pressed keyboard key.

        Returns:
            None
        """
        if key in self.map:

            # Immediate stop command
            if key == "space":

                # No velocity to stop
                self.vel_lin = 0.0
                self.vel_ang = 0.0

                self.get_logger().info("Stopping...")

                # Publish velocity command
                t = Twist()
                t.linear.x = self.vel_lin
                t.angular.z = self.vel_ang

                self._publisher.publish(t)

            else:
                lin, ang = self.map.get(key)

                # Increment velocities
                self.vel_lin += lin * 0.05
                self.vel_ang += ang * 0.3

                # Safety checks for velocity limits
                if abs(self.vel_ang) >= MAX_ANGULAR_VEL:
                    self.get_logger().info("Max angular velocity reached")

                elif abs(self.vel_lin) >= MAX_LINEAR_VEL:
                    self.get_logger().info("Max linear velocity reached")

                else:
                    # Publish velocity command
                    t = Twist()
                    t.linear.x = self.vel_lin
                    t.angular.z = self.vel_ang

                    self.get_logger().info(f"Transmiting key: {key}")
                    self._publisher.publish(t)

        else:
```

```
            self.get_logger().info(f"Not a valid key: {key}")

def main(args=None) -> None:
    """
    Entry point for the NodeMapper ROS2 node.

    Initializes ROS2, creates the node, spins it to process callbacks,
    and performs cleanup on shutdown.

    Args:
        args (list, optional): Command-line arguments passed to ROS2.

    Returns:
        None
    """
    # Initialize ROS2
    rclpy.init(args=args)

    # Create the node instance
    mapping_node = NodeMapper()

    # Keep the node alive and processing callbacks
    rclpy.spin(mapping_node)

    # Explicitly destroy the node before exit
    mapping_node.destroy_node()

    # Shutdown ROS2
    rclpy.shutdown()


# Script entry point
if __name__ == "__main__":
    main()
```

Lógica de funcionamiento del sistema

El sistema de teleoperación supervisada funciona mediante la coordinación de dos nodos que se comunican a través de topics. El nodo `keyboard_node` captura las pulsaciones del teclado (W, A, S, D, espacio) y las publica como mensajes `KeyPressed` en el topic `/listen_key`, ambos creados de forma personalizada.

El nodo `teleoperation_node` actúa como el cerebro del sistema, suscribiéndose simultáneamente a dos fuentes de información: las pulsaciones del teclado a través de `/listen_key` y los datos del LiDAR a través del topic `/scan`. Cada tecla se mapea a un incremento específico de velocidad: 'W' y 'S' modifican la velocidad lineal (avance y retroceso), 'A' y 'D' modifican

la velocidad angular (giros), y la barra espaciadora detiene completamente el robot.

La supervisión mediante LiDAR se implementa en el método `AnalyzeLiDAR`, que analiza continuamente las lecturas del sensor en dos direcciones críticas: frente (0 radianes) y atrás (π radianes). El método calcula los índices correspondientes en el array de rangos del LiDAR usando la siguiente fórmula:

$$\text{índice} = \frac{\text{ángulo} - \text{ángulo\_mínimo}}{\text{incremento\_angular}}$$

Si alguna de estas lecturas detecta un obstáculo más cerca de 25 cm (umbral que se puede modificar mediante la variable `OBSTACLE_THRESHOLD`), el sistema publica inmediatamente un comando de velocidad cero, deteniendo el robot antes de la colisión.

Configuración de perfiles QoS

Los perfiles de Calidad de Servicio (QoS) se configuraron de forma diferenciada según las necesidades de cada topic. Para el LiDAR se utilizó el perfil `BEST_EFFORT`, que es apropiado para sensores donde la pérdida ocasional de mensajes es aceptable, ya que los datos llegan con alta frecuencia y siempre se dispone de una lectura reciente. Este perfil reduce la sobrecarga de comunicación al no requerir confirmaciones de recepción.

Para los comandos de velocidad en `cmd_vel` se utilizó el perfil `RELIABLE`, que garantiza la entrega de cada comando al robot. Esto es fundamental porque la pérdida de comandos de velocidad podría resultar en comportamientos impredecibles del robot. La configuración también especifica `VOLATILE` en durabilidad, lo que significa que solo los suscriptores activos recibirán los mensajes, sin almacenamiento histórico.

Resultados de las pruebas

Durante las pruebas en el laboratorio, el sistema de teleoperación supervisada demostró un funcionamiento robusto y confiable. El robot respondió correctamente a todos los comandos de teclado, con incrementos suaves de velocidad que permitieron un control preciso del movimiento. La supervisión mediante LiDAR funcionó según lo esperado, deteniendo el robot automáticamente cuando se detectaron obstáculos a menos de 25 cm tanto en la parte frontal como trasera.

Se observó que el criterio de signos invertido de la velocidad angular (mencionado en la Pregunta #11) se compensó adecuadamente en el mapeo de teclas, resultando en un comportamiento intuitivo para el operador. Además, cabe resaltar también que el sistema mantuvo límites de seguridad en todo momento, impidiendo que el robot excediera las velocidades máximas configuradas de 0.22 m/s lineal y 2.81 rad/s angular.

6. Uso auxiliar de Inteligencia Artificial

A lo largo de la elaboración de esta memoria se ha empleado inteligencia artificial como herramienta de apoyo, sin sustituir en ningún caso el trabajo de comprensión, experimentación y redacción realizado por los autores. En concreto, su uso se ha limitado a tareas auxiliares orientadas a mejorar la claridad y la presentación del documento:

- **Corrección lingüística:** se ha utilizado para detectar y corregir faltas de ortografía, erratas y pequeños problemas de concordancia o puntuación.
- **Mejora del estilo y uniformidad:** se ha empleado para sugerir ajustes de redacción (frases más claras, eliminación de repeticiones y homogeneización del tono).
- **Consulta de ejemplos de licencias de software:** se ha utilizado para recopilar y verificar ejemplos habituales de licencias de software existentes.