1. Calcular velocidad de cada rueda (TODO 2.1)

La siguiente función calcula la velocidad de ambas ruedas del Turtlebot para conseguir la velocidad lineal y angular deseadas. Para ello utilizamos las fórmulas de cinemática diferencial inversa. Si alguna de las velocidades supera el límite de velocidad para una rueda, se escala la velocidad de ambas ruedas acordemente.

```
def move(self, v: float, w: float) -> None:
        """Solve inverse differential kinematics and send commands
to the motors.

        If the angular speed of any of the wheels is larger than
the maximum admissible,
        sets the larger value to the maximum speed and
proportionately scales the other.

        Args:
            v: Linear velocity of the robot center [m/s].
            w: Angular velocity of the robot center [rad/s].

        """
      # TODO: 2.1. Complete the function body with your code (i.e.,
replace the pass statement).
        left_speed = (v - self.TRACK * w / 2) / self.WHEEL_RADIUS
        right_speed = (v + self.TRACK * w / 2) / self.WHEEL_RADIUS

        if right_speed > left_speed and right_speed >
self.WHEEL_SPEED_MAX:
            left_speed *= self.WHEEL_SPEED_MAX / right_speed
            right_speed = self.WHEEL_SPEED_MAX
        elif left_speed > right_speed and left_speed >
self.WHEEL_SPEED_MAX:
            right_speed *= self.WHEEL_SPEED_MAX / left_speed
            left_speed = self.WHEEL_SPEED_MAX

        self._sim.setJointTargetVelocity(self._motors["left"],
left_speed)
        self._sim.setJointTargetVelocity(self._motors["right"],
right_speed)
```

2. Velocidad medida por el encoder (TODO 2.2 y 2.3)

La siguiente función calcula en primer lugar la velocidad angular de cada rueda derivando las posiciones angulares medidas por el encoder. En segundo lugar, utilizamos las ecuaciones de la cinemática diferencial directa para calcular la velocidad lineal y angular del robot y devolverlas.

```
def _sense_encoders(self) -> tuple[float, float]:
        """Solve forward differential kinematics from encoder
readings.

        Returns:
            z_v: Linear velocity of the robot center [m/s].
            z_w: Angular velocity of the robot center [rad/s].

        """
        # Read the angular position increment in the last sampling
period [rad]
        encoders: dict[str, float] = {}

        encoders["left"] =
self._sim.getFloatProperty(self._sim.handle_scene,
"signal.leftEncoder")
        encoders["right"] = self._sim.getFloatProperty(
            self._sim.handle_scene, "signal.rightEncoder"
        )

        # TODO: 2.2. Compute the derivatives of the angular
positions to obtain velocities [rad/s].
        left_wheel_velocity = encoders["left"] / self._dt
        right_wheel_velocity = encoders["right"] / self._dt

        # TODO: 2.3. Solve forward differential kinematics (i.e.,
calculate z_v and z_w).
        z_v = self.WHEEL_RADIUS * (left_wheel_velocity +
right_wheel_velocity) / 2
        z_w = self.WHEEL_RADIUS * (right_wheel_velocity -
left_wheel_velocity) / (self.TRACK)

        return z_v, z_w
```

3. Publicar temas desde coppeliasim_node (TODO 2.4, 2.5 y 2.6)

El siguiente código crea un publicador para mensajes de tipo Odometry en el tema "/odometry" y un publicador para mensajes de tipo LaserScan en el tema "/scan". Para poder publicar el mensaje del LiDAR, tenemos que definir una calidad de servicio compatible con la que utiliza el LiDAR.

```
# TODO: 2.4. Create the /odometry (Odometry message) and /scan
(LaserScan) publishers.
            qos_profile = QoSProfile(
                reliability=QoSReliabilityPolicy.BEST_EFFORT,
                durability=QoSDurabilityPolicy.VOLATILE,
                depth=10,
                history=QoSHistoryPolicy.KEEP_LAST,
            )
            self.publisher_odometry =
self.create_publisher(Odometry, "/odometry", 10)
            self.publisher_scan = self.create_publisher(LaserScan,
"/scan", qos_profile)
```

La siguiente función crea un mensaje de tipo Odometry para contener los datos sobre velocidad lineal y angular del robot y publicarlos en el tema "/odometry". También incluimos un header con la marca de tiempo.

```
def _publish_odometry(self, z_v: float, z_w: float) -> None:
        """Publishes odometry measurements in a
nav_msgs.msg.Odometry message.

        Args:
            z_v: Linear velocity of the robot center [m/s].
            z_w: Angular velocity of the robot center [rad/s].

        """
        # TODO: 2.5. Complete the function body with your code
(i.e., replace the pass statement).
        odometry = Odometry()
        odometry.header.stamp = self.get_clock().now().to_msg()
        odometry.twist.twist.linear.x = z_v
        odometry.twist.twist.angular.z = z_w

        self.publisher_odometry.publish(odometry)
```

La siguiente función crea un mensaje de tipo LaserScan conteniendo el rango de valores obtenido por el LiDAR y la marca de tiempo, y lo publica en el tema "/scan".

```
def _publish_scan(self, z_scan: list[float]) -> None:
        """Publishes LiDAR measurements in a
sensor_msgs.msg.LaserScan message.

        Args:
            z_scan: Distance from every ray to the closest obstacle
in counterclockwise order [m].

        """
        # TODO: 2.6. Complete the function body with your code
(i.e., replace the pass statement).
        laser_scan = LaserScan()
        laser_scan.header.stamp = self.get_clock().now().to_msg()
        laser_scan.ranges = z_scan

        self.publisher_scan.publish(laser_scan)
```

4. Suscribirse en wall_follower_node (TODO 2.7, 2.8 y 2.9)

El siguiente código crea los suscriptores a los temas de "/odometry" y "/scan". Para asegurarse de que no se ejecuta el algortimo de navegación hasta que se hallan recibido todos los datos, utilizamos el objeto ApproximateTimeSynchronizer que llama a la

función de _compute_commands_callback solo cuando ha recibido un conjunto completo de mensajes.

```
qos_profile = QoSProfile(
reliability=QoSReliabilityPolicy.BEST_EFFORT,
durability=QoSDurabilityPolicy.VOLATILE,
depth=10,
history=QoSHistoryPolicy.KEEP_LAST,
)
self._subscribers: list[message_filters.Subscriber] = []
# Append as many topics as needed
self._subscribers.append(message_filters.Subscriber(
      self, Odometry, "/odometry", qos_profile=10)
)
self._subscribers.append(message_filters.Subscriber(
      self, LaserScan, "/scan", qos_profile=qos_profile)
)
ts = message_filters.ApproximateTimeSynchronizer(
      self._subscribers, queue_size=10, slop=9
)
ts.registerCallback(
      self._compute_commands_callback
)
```

A continuación, tenemos el código que parsea los datos del mensaje Odometry y el LaserScan.

```
# TODO: 2.8. Parse the odometry from the Odometry message (i.e.,
read z_v and z_w).
z_v: float = odom_msg.twist.twist.linear.x
z_w: float = odom_msg.twist.twist.angular.z

# TODO: 2.9. Parse LiDAR measurements from the LaserScan message
(i.e., read z_scan).
z_scan: list[float] = scan_msg.ranges
```

5. Publicar en “/cmd_vel” desde wall_follower_node (TODO 2.10 y 2.11)

El siguiente código crea un publicador para mensajes de tipo TwistStamped en el tema “/cmd_vel”.

```
# TODO: 2.10. Create the /cmd_vel velocity commands publisher
(TwistStamped message).
self.publisher = self.create_publisher(TwistStamped, "/cmd_vel",
10)
```

La siguiente función crea un mensaje de tipo TwistStamped que contiene la velocidad lineal y angular, así como una marca de tiempo. Luego publica el mensaje en el tema "/cmd_vel".

```
def _publish_velocity_commands(self, v: float, w: float) -> None:
        """Publishes velocity commands in a
geometry_msgs.msg.TwistStamped message.

        Args:
            v: Linear velocity command [m/s].
            w: Angular velocity command [rad/s].

        """
        # TODO: 2.11. Complete the function body with your code
(i.e., replace the pass statement).
        msg = TwistStamped()
        msg.twist.linear.x = v
        msg.twist.angular.z = w
        msg.header.stamp = self.get_clock().now().to_msg()
        self.publisher.publish(msg)
```

6. Suscribirse a "/cmd_vel" desde coppeliasim_node

El siguiente código crea el suscriptor al tema "/cmd_vel".

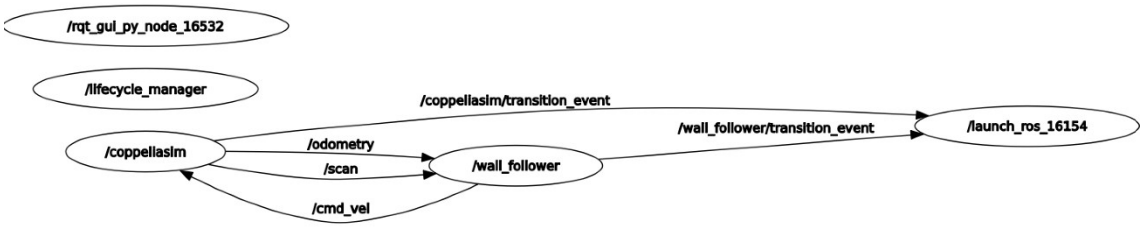
```
# TODO: 2.12. Subscribe to /cmd_vel. Connect it with with
_next_step_callback.
            self.subscriber = self.create_subscription(
                TwistStamped, "/cmd_vel",
callback=self._next_step_callback, qos_profile=10
```

El siguiente código parsea la velocidad lineal y angular del mensaje recibido.

```
# TODO: 2.13. Parse the velocities from the TwistStamped message
(i.e., read v and w).
        v: float = cmd_vel_msg.twist.linear.x
        w: float = cmd_vel_msg.twist.angular.z
```

7. RQT_GRAPH (Pregunta 1)

En el siguiente gráfico podemos ver los dos nodos que hemos estado programando, el coppeliasim y el wall_follower. Podemos ver que el coppeliasim publica en el tema "/odometry" la información sobre la odometría del robot, y en el tema "/scan" los datos del LiDAR. Mientras, el wall_follower se suscribe a ambos temas para recibir la información del robot y publica en el tema "/cmd_vel" los comandos de velocidad generados. El coppeliasim se suscribe a este tema y aplica los comandos de velocidad al robot simulado.



8. Seguimiento de pared

El sistema se basa en una máquina de estados finitos con dos modos principales: "Avanzar" y "Giro". En el estado de avance, el robot sigue una pared manteniendo una distancia deseada, mientras que en el estado de giro, corrige su trayectoria al detectar un obstáculo frontal.

El control del movimiento se gestiona mediante un controlador PD que ajusta la velocidad angular en función del error de distancia respecto a la pared.

Se segmenta la información del LiDAR en regiones frontal, izquierda y derecha. Cuando la distancia frontal cae por debajo de un umbral, el robot cambia a modo "Giro", ajustando su orientación hasta que tenga espacio delante y volver a "Avanzar".

```
import math
class WallFollower:
    """Class to safely explore an environment (without crashing)
when the pose is unknown."""

    def __init__(self, dt: float) -> None:
        """Wall following class initializer.

        Args:
            dt: Sampling period [s].

        """
        self._dt: float = dt
        self.state = "AVANZAR"

        self.Kp = 150.0
        self.Kd = 15.0
        self.prev_error = 0.0

        self.stop_distance = 0.2
        self.follow_distance = 0.2
```

```
        self.v = 2.0
        self.w_control = 0.2
        self.w_giro = 3.0

        self.w_actual = 0.0

    def compute_commands(self, z_scan: list[float], z_v: float,
z_w: float) -> tuple[float, float]:
        """Wall following exploration algorithm.

        Args:
            z_scan: Distance from every LiDAR ray to the closest
obstacle [m].
            z_v: Odometric estimate of the linear velocity of the
robot center [m/s].
            z_w: Odometric estimate of the angular velocity of the
robot center [rad/s].

        Returns:
            v: Linear velocity [m/s].
            w: Angular velocity [rad/s].

        """
        # TODO: 2.14. Complete the function body with your code
(i.e., compute v and w).

        # Selección de sectores del LiDAR
        front = list(z_scan[0:20]) + list(z_scan[-20:])
        left = list(z_scan[40:60])
        right = list(z_scan[-60:-40])

        # # Ignorar valores nan o negativos
        valid_front = [d for d in front if d > 0.0]
        valid_left = [d for d in left if d > 0.0]
        valid_right = [d for d in right if d > 0.0]

        # Cálculo de distancias mínimas con detección de pared muy
cercana
        d_front = min(valid_front) if valid_front else 0.0
        d_left = min(valid_left) if valid_left else 0.0
        d_right = min(valid_right) if valid_right else 0.0


        # Estado: AVANZAR
        if self.state == "AVANZAR":
            error = self.follow_distance - d_right

            # Control PD
            P = self.Kp * error
            D = self.Kd * (error - self.prev_error) / self._dt
            w = (P + D) * 0.8 + z_w * 0.2 # Suavizamos con la
velocidad angular actual
            w = min(w, self.w_control) if w > 0 else max(w, -
self.w_control)

            self.prev_error = error
```

```
            v = min(self.v, z_v + 0.02)  # Aceleramos
progresivamente

            # Si la pared desaparece, vuelve a AVANZAR
            if d_front < self.stop_distance:
                self.state = "GIRO"
                v = 0.0
                w = self.w_giro if d_left > d_right else -
self.w_giro
                self.w_actual = w

        # Estado: GIRO
        elif self.state == "GIRO":
            if d_front > self.stop_distance + 0.15:
                self.state = "AVANZAR"
                v = self.v
                w = -self.w_actual*0.5
            else:
                v = 0.0
                w = self.w_actual


        return v, w
```

9. Nuevo odometry_node

El nodo odometry_node se trata de un LifecycleNode. Al configurarse crea un suscriptor al tema "/odom" y un publicador al tema "/odometry". Lo que hará este nodo es derivar la pose en x, en y, y en ángulo para obtener las velocidades lineal y angular del robot. Para lograr esto guardamos los últimos valores en cada iteración y calculamos la derivada como el valor actual menos el anterior entre el tiempo que ha pasado entre ambas iteraciones consecutivas. También será importante convertir los cuaterniones a ángulos de Euler antes de todo, utilizando la función quat2euler. Luego guardamos los valores en el mensaje Odometry que hemos recibido de "/odom" para conservar todos los datos del mensaje original, y publicamos el mensaje con los nuevos datos en el tema "/odometry".

```
class OdometryNode(LifecycleNode):
    def __init__(self):
        super().__init__('odometry_node')
        # self.subscription = self.create_subscription(
        #     Odometry,
        #     '/odom',
        #     self.odom_callback,
        #     10
        # )
        # self.publisher = self.create_publisher(Odometry,
'/odometry', 10)

        # Variables para almacenar la última posición y tiempo
        self.last_x = None
```

```
        self.last_y = None
        self.last_theta = None
        self.last_time = None

        # Parameters
        self.declare_parameter("dt", 0.05)
        self.declare_parameter("enable_localization", False)

    def odom_callback(self, msg):
        # Extraer posición actual del mensaje de odometría
        current_x = msg.pose.pose.position.x
        current_y = msg.pose.pose.position.y

        # Convertir cuaternión a ángulo de Euler (yaw)
        orientation_q = msg.pose.pose.orientation
        quaternion = [orientation_q.w, orientation_q.x,
orientation_q.y, orientation_q.z]
        _, _, current_theta = quat2euler(quaternion,
axes='sxyz')  # Convención estándar ZYX

        # Obtener el tiempo actual del mensaje
        current_time = msg.header.stamp.sec +
msg.header.stamp.nanosec * 1e-9

        # Si no hay datos previos, inicializar y salir
        if self.last_time is None:
            self.last_x = current_x
            self.last_y = current_y
            self.last_theta = current_theta
            self.last_time = current_time
            return

        # Calcular diferencias en posición, orientación y tiempo
        delta_x = current_x - self.last_x
        delta_y = current_y - self.last_y
        delta_theta = current_theta - self.last_theta

        # Normalizar delta_theta para que esté entre -pi y pi
        delta_theta = math.atan2(math.sin(delta_theta),
math.cos(delta_theta))

        delta_time = current_time - self.last_time

        if delta_time > 0:
            # Calcular velocidades lineales y angulares
            linear_velocity = math.sqrt(delta_x**2 + delta_y**2) /
delta_time
            angular_velocity = delta_theta / delta_time

            # Publicar las velocidades calculadas en el tópico
/odometry
            msg.twist.twist.linear.x = linear_velocity
            msg.twist.twist.angular.z = angular_velocity
            self.publisher.publish(msg)

        # Actualizar los valores previos para la próxima iteración
        self.last_x = current_x
        self.last_y = current_y
```

```
        self.last_theta = current_theta
        self.last_time = current_time

    def on_configure(self, state: LifecycleState) ->
TransitionCallbackReturn:
        """Handles a configuring transition.

        Args:
            state: Current lifecycle state.

        """
        self.get_logger().info(f"Transitioning from '{state.label}'
to 'inactive' state.")

        try:
            # Parameters
            dt =
self.get_parameter("dt").get_parameter_value().double_value
            enable_localization = (
                self.get_parameter("enable_localization").get_param
eter_value().bool_value
            )

            self.subscription = self.create_subscription(
                msg_type=Odometry,
                topic = "odom",
                callback = self.odom_callback,
                qos_profile = 10,
            )

            self.publisher = self.create_publisher(
                msg_type = Odometry,
                topic = "/odometry",
                qos_profile = 10,
            )

        except Exception:
            self.get_logger().error(f"{traceback.format_exc()}")
            return TransitionCallbackReturn.ERROR

        return super().on_activate(state)

    def on_activate(self, state: LifecycleState) ->
TransitionCallbackReturn:
        """Handles an activating transition.

        Args:
            state: Current lifecycle state.

        """
        self.get_logger().info(f"Transitioning from '{state.label}'
to 'active' state.")

        return super().on_activate(state)

def main(args=None):
    rclpy.init(args=args)
    odometry_node = OdometryNode()
```

```
    try:
        rclpy.spin(odometry_node)
    except KeyboardInterrupt:
        pass

    odometry_node.destroy_node()
    rclpy.try_shutdown()


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

10. Cambios de la simulación al robot real

En primer lugar, no se debe usar el paquete amr_simulation ya que está diseñado exclusivamente para la interacción con Coppeliasim. En su lugar, utilizaremos el paquete turtlebot3_bringup y el archivo de lanzamiento robot.launch.py.

Para trasladar los paquetes creados al robot (amr_control, amr_msgs y amr_bringup) habrá que tener especial cuidado al trasladar amr_msgs. No se deberá copiar el paquete entero sino únicamente PoseStamped.msg. Además, habrá que actualizar CMakeLists para añadir el mensaje PoseStamped y las dependencias “std_msgs” y “geometry_msgs”.

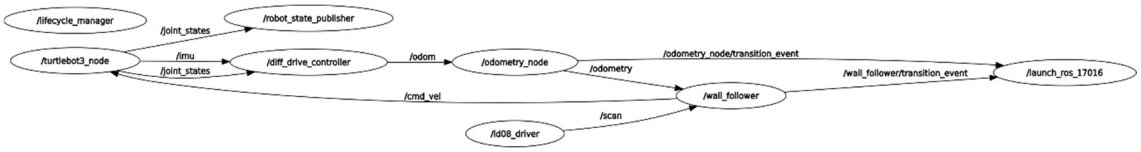
También habrá que crear el nodo odometry_node que se ha explicado en el punto anterior.

En la simulación, los comandos de velocidad se publicaban en “/cmd_vel” como mensajes de tipo TwistStamped. En el robot real utilizaremos mensajes de tipo Twist.

En la simulación sincronizábamos los mensajes recibidos por wall_follower_node se sincronizaban con un slop muy alto de 9. En el robot real utilizaremos un slop de 0.25 para minimizar el retraso entre los sensores.

En el algoritmo de wall_follower es también necesario cambiar algunos parámetros. En primer lugar, las velocidades angulares deben tener el signo contrario. Esto ocurre porque nuestro robot tiene el eje z mirando hacia abajo, lo cual es opuesto a la orientación convencional. Por tanto, al rotar sobre el eje z nuestro robot lo hará en dirección contraria a la convencional. También ajustamos los valores de las velocidades lineal y angular, así como las constantes proporcional y diferencial del control PD, para ajustarlos al robot real y que funcione correctamente.

El nuevo rqt_graph queda así:



Podemos ver que el diff_drive_controller recibe los datos sobre las articulaciones del robot y los datos del LiDar del turtlebot3_node y publica el mensaje Odometry con la información sobre la pose del robot en el tema “/odom”. El nuevo odometry_node recibe este mensaje y publica el mensaje con las velocidades lineal y angular en el tema “/odometry”. Los nodos de tipo LifecycleNode (odometry_node y wall_follower) publican sus transiciones de estados en los temas respectivos.