# 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)

# 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

# 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)

# 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

# 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)

# 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.

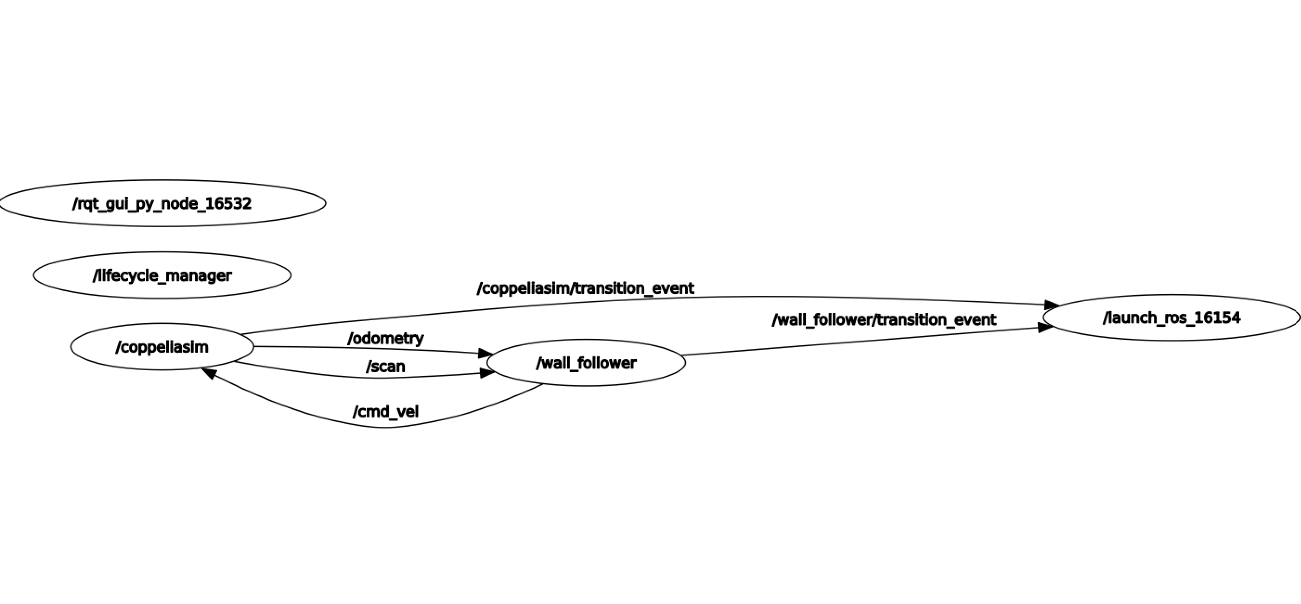
# TODO: 2.13. Parse the velocities from the TwistStamped message (i.e., read v and w).

        v: float = cmd\_vel\_msg.twist.linear.x

        w: float = cmd\_vel\_msg.twist.angular.z

# 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.



# 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

# 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\_parameter\_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()

# 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”.

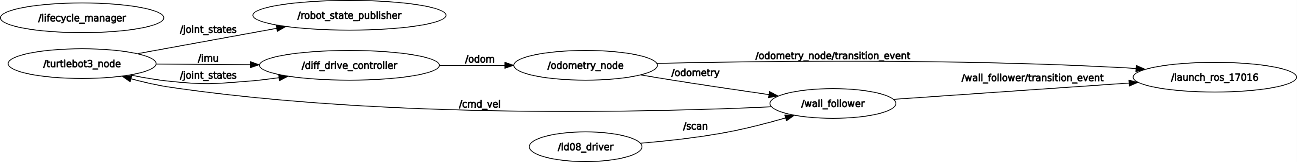
También habrá que crear el nodo odometry\_node que se ha explicado en el punto anterior.

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.