**ROBOTS MÓVILES AUTÓNOMOS**

Ingeniería Matemática e Inteligencia Artificial

# Práctica 2: Exploración: Seguimiento de Pared

**Informe de Laboratorio**

**Autores:**
Liam Esgueva González
Jorge Carnicero Príncipe
Andrés Gil Vicente

**Fecha de entrega:**
23 de febrero de 2026

# Índice

# 1. Se mueve como Jagger

En esta primera sección implementamos la cinemática inversa del TurtleBot3 Burger para controlar sus movimientos. El objetivo es completar el método `move` en el archivo `robot_turtlebot3_burger.py`, que debe convertir las velocidades lineal ($v$) y angular ($\omega$) del robot en velocidades angulares para las ruedas izquierda y derecha.

Para ello, utilizamos las ecuaciones de la cinemática diferencial. Dado que el robot tiene tracción diferencial con dos ruedas motorizadas, las relaciones entre las velocidades del centro del robot y las velocidades angulares de las ruedas vienen dadas por:

$$v = \frac{\dot{\phi}_r + \dot{\phi}_l}{2} r \tag{1}$$

$$\omega = \frac{\dot{\phi}_r - \dot{\phi}_l}{2b} r \tag{2}$$

donde $\dot{\phi}_r$ y $\dot{\phi}_l$ son las velocidades angulares de las ruedas derecha e izquierda respectivamente, $r$ es el radio de las ruedas y $b$ es la mitad de la distancia entre ruedas (track/2).

Resolviendo el sistema de ecuaciones para obtener las velocidades angulares de las ruedas (cinemática inversa) obtenemos las ecuaciones que hemos visto en las clases de teoría:

$$\dot{\phi}_l = \frac{1}{r}(v - b\omega) \tag{3}$$

$$\dot{\phi}_r = \frac{1}{r}(v + b\omega) \tag{4}$$

Nuestra implementación también garantiza que las velocidades calculadas no superen los límites físicos del robot. Si cualquiera de las ruedas excede la velocidad angular máxima definida en `WHEEL_SPEED_MAX`, mantenemos las velocidades del periodo de muestreo anterior en lugar de aplicar comandos inválidos.

**Código 1.0: Implementación de la cinemática inversa - TODO 2.1**

```
def move(self, v: float, w: float) -> None:
    # TODO: 2.1. Complete the function body with your code (i.e., replace
    the pass statement).

    # The distance between the center of the robot and the wheel is track/2
    b = self._track / 2

    # We compute the angular vel of the wheels (inverse diff): dot_X_R = v,
    dot_theta_R = w
    left_angular_vel = (1/self._wheel_radius) * (v-b*w)
```

```
        right_angular_vel = (1/self._wheel_radius) * (v+b*w)

        # We check if velocities are valid before sending them
        if (abs(left_angular_vel) <= self.WHEEL_SPEED_MAX and abs(
    right_angular_vel) <= self.WHEEL_SPEED_MAX):

            self._last_left_angular_vel = left_angular_vel
            self._last_right_angular_vel = right_angular_vel

        self._sim.setJointTargetVelocity(self._motors["left"],  self.
    _last_left_angular_vel)
        self._sim.setJointTargetVelocity(self._motors["right"], self.
    _last_right_angular_vel)
```

Como se observa en el código, primero calculamos $b$ como la mitad de la distancia entre ruedas. Luego aplicamos las fórmulas de la cinemática inversa para obtener las velocidades angulares de cada rueda. Antes de enviar los comandos a los motores, verificamos que ambas velocidades estén dentro de los límites permitidos. Solo si ambas son válidas, actualizamos los valores almacenados en los atributos de clase. Finalmente, enviamos las velocidades (ya sean las recién calculadas o las del periodo anterior) a los motores mediante la API de CoppeliaSim, con el comando que se nos proporciona en el enunciado de la práctica.

Esta estrategia conservadora de mantener las velocidades anteriores cuando se superan los límites garantiza que el robot nunca reciba comandos inválidos, lo cual es fundamental para mantener un comportamiento predecible y seguro durante la navegación.

## 2. Volviendo sobre sus pasos como Pulgarcito

La odometría es esencial para conocer el estado del robot en cada instante. En esta sección implementamos la cinemática directa para estimar las velocidades lineal y angular reales del robot a partir de las lecturas de los encoders de las ruedas.

Los encoders nos proporcionan el incremento de posición angular de cada rueda durante el último periodo de muestreo $\Delta t$. Para obtener las velocidades angulares de las ruedas, simplemente dividimos estos incrementos entre el periodo de muestreo (derivada discreta):

$$\dot{\phi}_l = \frac{\Delta\phi_l}{\Delta t} \tag{5}$$

$$\dot{\phi}_r = \frac{\Delta\phi_r}{\Delta t} \tag{6}$$

Una vez tenemos las velocidades angulares de las ruedas, aplicamos las ecuaciones de la cinemática directa que relacionan estas velocidades con las velocidades lineal y angular del centro del robot:

$$v = \frac{r}{2}(\dot{\phi}_l + \dot{\phi}_r) \tag{7}$$

$$\omega = \frac{r}{2b}(\dot{\phi}_r - \dot{\phi}_l) \tag{8}$$

donde $r$ es el radio de las ruedas y $b$ es la mitad de la distancia entre ruedas.

**Código 2.0: Implementación de la cinemática directa - TODO 2.2 y 2.3**

```
# TODO: 2.2. Compute the derivatives of the angular positions to obtain
    velocities [rad/s].

# The distance between the center of the robot and the wheel is track/2
b = self._track / 2

# We compute the vel of the wheels using the encoders (dividing by time dt )
left_wheel_vel = encoders["left"] / self._dt
right_wheel_vel = encoders["right"] / self._dt

# TODO: 2.3. Solve forward differential kinematics (i.e., calculate z_v and
    z_w).
# We compute the angular and linear vels of the robot (direct diff)
z_v = (self._wheel_radius/2) * (left_wheel_vel + right_wheel_vel)
z_w = (self._wheel_radius/(2*b)) * (right_wheel_vel - left_wheel_vel)

return z_v, z_w
```

En nuestra implementación, primero leemos los incrementos de posición angular de ambos encoders proporcionados por el simulador. Luego calculamos las velocidades angulares de las ruedas dividiendo estos incrementos entre el periodo de muestreo $\Delta t$. Finalmente, aplicamos las fórmulas de la cinemática directa para obtener la velocidad lineal $z_v$ y angular $z_w$ del centro del robot.

Estas medidas odométricas son fundamentales para el sistema de control, ya que nos permiten conocer con precisión el estado de movimiento real del robot en cada instante, lo cual es necesario para implementar el algoritmo de control en lazo cerrado que desarrollamos en otro apartado posterior, y para posibles futuros sistemas de localización que quisiéramos implementar.

# 3. El teléfono escacharrado

En esta sección establecemos la infraestructura de comunicación entre el nodo del simulador y el nodo de seguimiento de pared utilizando topics de ROS 2. Esta arquitectura modular permite separar claramente las responsabilidades: el simulador se encarga de la física del robot y los sensores, mientras que el nodo de control implementa la lógica de navegación.

## 3.1. Mensaje en una botella

El primer paso es crear los publicadores en el nodo del simulador para transmitir las medidas de los sensores. Implementamos dos topics: `/odometry` para publicar las velocidades del robot y `/scan` para las medidas del LiDAR.

Para el topic `/odometry` utilizamos el tipo de mensaje estándar `Odometry`, que incluye información sobre la posición, orientación y velocidades del robot. En nuestro caso, nos centramos en rellenar los campos de velocidad lineal y angular dentro de la estructura `twist.twist`.

Para el topic `/scan` utilizamos el tipo de mensaje `LaserScan`, específicamente diseñado para datos de escáneres láser. Este mensaje contiene un array de distancias medidas por cada rayo del LiDAR.

**Código 3.0: Publicadores en el nodo del simulador - TODO 2.4**

```
# LiDAR quality of service
qos_lidar_profile = QoSProfile(
    history=QoSHistoryPolicy.KEEP_LAST,
    depth=10,
    reliability=QoSReliabilityPolicy.BEST_EFFORT,
    durability=QoSDurabilityPolicy.VOLATILE,
)

# Publishers
# TODO: 2.4. Create the /odometry (Odometry message) and /scan (LaserScan)
   publishers.
self._odometry_publisher = self.create_publisher(
    msg_type=Odometry,
    topic = "odometry",
    qos_profile = 10  # we may need: ros2 topic info odometry -v
)

self._laserScan_publisher = self.create_publisher(
    msg_type=LaserScan,
    topic = "scan",
    qos_profile=qos_lidar_profile
)
```

Es importante resaltar que para el suscriptor del LiDAR hemos configurado un perfil de calidad de servicio (QoS) específico con las políticas apropiadas para sensores donde la pérdida ocasional de datos es aceptable, dado que llegan nuevas medidas con alta frecuencia.

Las funciones de publicación extraen los datos de los sensores y los empaquetan en los mensajes apropiados:

**Código 3.1: Funciones de publicación de medidas - TODO 2.5 y 2.6**

```
def _publish_odometry(self, z_v: float, z_w: float) -> None:
    # TODO: 2.5. Complete the function body with your code (i.e., replace
    the pass statement).

    msg = Odometry()
    msg.header.stamp = self.get_clock().now().to_msg()

    msg.twist.twist.linear.x = z_v
    msg.twist.twist.angular.z = z_w

    self._odometry_publisher.publish(msg=msg)

def _publish_scan(self, z_scan: list[float]) -> None:
    # TODO: 2.6. Complete the function body with your code (i.e., replace
    the pass statement).

    msg = LaserScan()
    msg.header.stamp = self.get_clock().now().to_msg()

    msg.angle_min = 0.0
    msg.angle_max = 2.0 * math.pi
    msg.angle_increment = 2.0 * math.pi / len(z_scan)
    msg.ranges = z_scan

    self._laserScan_publisher.publish(msg=msg)
```

## 3.2. Puede retrasarse, pero el tiempo no lo hará

El nodo de seguimiento de pared necesita recibir las medidas de ambos sensores para calcular los comandos de navegación apropiados. Sin embargo, no tiene sentido que ejecutemos el algoritmo de control con información incompleta. Por tanto, necesitamos sincronizar la recepción de mensajes de múltiples topics, para que se ejecute el callback que queremos pero sólo cuando tengamos la información de todos los topics que necesitamos.

Para resolver este problema utilizamos `message_filters` con un sincronizador de tiempo aproximado (`ApproximateTimeSynchronizer`). Este objeto espera hasta recibir un conjunto

completo de mensajes de todos los topics de interés con marcas de tiempo cercanas antes de invocar el callback.

**Código 3.2: Sincronización de suscriptores - TODO 2.7**

```
# Subscribers
# TODO: 2.7. Synchronize _compute_commands_callback with /odometry and /scan
    .

# We define an empty list of suscribers
self._subscribers: list[ message_filters.Subscriber ] = []

# We append the odometry suscriber
self._subscribers.append(
    message_filters.Subscriber(
        self,
        Odometry,
        "odometry",
        qos_profile = 10
    )
)

# We append the laser scan suscriber
self._subscribers.append(
    message_filters.Subscriber(
        self,
        LaserScan,
        "scan",
        qos_profile = qos_lidar_profile
    )
)

# Wait until receive all measurements, then invoke the callback
ts = message_filters.ApproximateTimeSynchronizer(
    self._subscribers,
    queue_size=10,  # number of messages of each until we are "completed"
    slop=10.0  # max delay in seconds to consider that 2 messages are able
    to be synchronized
)

# We register the callback that we want to execute once the measurements are
     received
ts.registerCallback(self._compute_commands_callback)
```

El parámetro `queue_size` especifica cuántos mensajes de cada topic se deben almacenar mientras se espera a completar el conjunto. El parámetro `slop` define el retardo máximo en segundos para considerar que dos mensajes pueden sincronizarse. Inicialmente utilizamos un valor alto (10 segundos) para garantizar el funcionamiento en simulación, aunque este valor lo reduciremos cuando transfiramos el código al robot físico, como mostraremos más

adelante.

Una vez recibidos todos los mensajes sincronizados que necesitamos, el callback que hemos configurado extrae la información relevante de cada uno de los mensajes recibidos, y publica los comandos de velocidad apoyándose en el método de `compute_commands` del nodo `WallFollower`, que permite calcular correctamente dichos comandos:

**Código 3.3: Extracción de datos de los mensajes - TODO 2.8 y 2.9**

```
def _compute_commands_callback(
        self, odom_msg: Odometry, scan_msg: LaserScan, pose_msg: PoseStamped
     = PoseStamped()
    ):

    if not pose_msg.localized:
        # TODO: 2.8. Parse the odometry from the Odometry message (i.e.,
    read z_v and z_w).

        # We need to extract the info from the messages inside the message
        z_v: float = odom_msg.twist.twist.linear.x  # linear vel from the
    robot in x axis
        z_w: float = odom_msg.twist.twist.angular.z  # angular vel from the
    robot in z axis

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

        # Execute wall follower
        v, w = self._wall_follower.compute_commands(z_scan, z_v, z_w)
        self.get_logger().info(f"Commands: v = {v:.3f} m/s, w = {w:+.3f} rad
    /s")

        # Publish
        self._publish_velocity_commands(v, w)
```

## 3.3. ¡Que empiece el baile!

Nuestro nodo de seguimiento de pared debe poder enviar comandos de velocidad al robot. Para ello creamos un publicador en el topic `/cmd_vel` que utiliza mensajes de tipo `TwistStamped`.

El tipo de mensaje `TwistStamped` es una variante de `Twist` que incluye una cabecera con marca de tiempo. Esto facilita la sincronización del flujo de información con el simulador y mejora la trazabilidad del sistema. Sin embargo, debemos tener en cuenta que al transferir

el código al robot físico lo cambiaremos a `Twist` estándar.

**Código 3.4: Publicador de comandos de velocidad - TODO 2.10 y 2.11**

```
# Publishers
# TODO: 2.10. Create the /cmd_vel velocity commands publisher (TwistStamped
    message).

# Publisher will be modified when transfer the code to the physical robot
self._commands_publisher = self.create_publisher(
    msg_type=TwistStamped,
    topic = "cmd_vel",
    qos_profile=10
)

def _publish_velocity_commands(self, v: float, w: float) -> None:

    # TODO: 2.11. Complete the function body with your code (i.e., replace
    the pass statement).

    # We create a TwistStamped() messages and introduce the info of v and w
    msg = TwistStamped()
    msg.header.stamp = self.get_clock().now().to_msg()
    msg.twist.linear.x = v
    msg.twist.angular.z = w

    # We publish the message through the commands publisher
    self._commands_publisher.publish(msg)
```

## 3.4. Solo déjate llevar

Finalmente, el nodo del simulador debe recibir los comandos de velocidad publicados por el nodo de control y aplicarlos al robot simulado. Para ello creamos un suscriptor al topic `/cmd_vel` en el nodo del simulador:

**Código 3.5: Suscriptor a comandos de velocidad - TODO 2.12**

```
# Subscribers
# TODO: 2.12. Subscribe to /cmd_vel. Point to _next_step_callback.
self._cmd_vel_suscriber = self.create_subscription(
    msg_type=TwistStamped,
    topic="cmd_vel",
    callback=self._next_step_callback,
    qos_profile=10
```

```
)
```

El callback `_next_step_callback` extrae las velocidades del mensaje y se las pasa al robot simulado:

**Código 3.6: Procesamiento de comandos de velocidad TODO - 2.13**

```
def _next_step_callback(self, cmd_vel_msg: TwistStamped, pose_msg:
    PoseStamped = PoseStamped()):

    # Check estimated pose
    self._check_estimated_pose(pose_msg)

    # 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

    # Execute simulation step
    self._robot.move(v, w)
    self._coppeliasim.next_step()
    z_scan, z_v, z_w = self._robot.sense()

    self.get_logger().info(f"Odometry: z_v = {z_v:.3f} m/s, w = {z_w:+.3f}
    rad/s")

    # Check goal
    if self._check_goal():
        return

    # Publish
    self._publish_odometry(z_v, z_w)
    self._publish_scan(z_scan)
```

Con esta implementación completamos el ciclo de comunicación: el simulador publica las medidas de los sensores, el nodo de control las recibe sincronizadas, calcula los comandos apropiados y los publica. Finalmente, el simulador recibe estos comandos y actualiza el estado del robot.

Al ejecutar el archivo de lanzamiento con la escena de CoppeliaSim abierta, verificamos que el robot se mueve correctamente con una velocidad lineal constante de 0.15 m/s, que es el comportamiento por defecto implementado en la plantilla que se nos proporciona, antes de que programemos el algoritmo de seguimiento de pared.

## 3.5. Una imagen vale más que mil líneas de código

**Pregunta #1:** Incluya en el informe un diagrama de rqt_graph que permita entender el flujo de información entre los dos nodos que ha programado en la práctica y explíquelo brevemente.

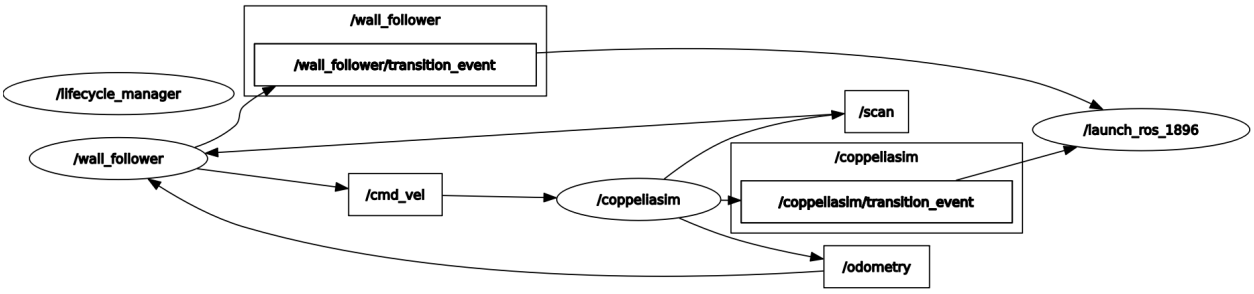


Figura 1: Diagrama rqt_graph mostrando la comunicación entre nodos

El diagrama de `rqt_graph` muestra la arquitectura de comunicación que tenemos implementada. Podemos observar dos nodos principales:

- **/coppeliasim**: Actúa como nodo del simulador. Publica en los topics `/odometry` y `/scan`, proporcionando las medidas de velocidad del robot y las lecturas del LiDAR respectivamente. Además, está suscrito al topic `/cmd_vel` para recibir los comandos de velocidad.
- **/wall_follower**: Implementa el algoritmo de control. Está suscrito a `/odometry` y `/scan` para recibir las medidas de los sensores de forma sincronizada. Publica en el topic `/cmd_vel` los comandos de velocidad calculados por el algoritmo de seguimiento de pared.
- **/launch_ros_1896**: Este nodo es generado automáticamente por el sistema de lanzamiento de ROS 2. En el grafo, se observa que está suscrito a los topics `/transition_event` tanto del simulador como del controlador. Esto indica que el sistema de lanzamiento está monitorizando los cambios de estado de los nodos para gestionar la secuencia de arranque de la aplicación.
- **/lifecycle_manager**: Es el nodo encargado de orquestar el ciclo de vida. Aunque no se ven las conexiones, su función es enviar los cambios de estado a `/coppeliasim` y `/wall_follower`, asegurando que el robot no empiece a moverse hasta que los sensores y el simulador estén listos.

# 4. El corredor del laberinto

El desafío principal de esta práctica es desarrollar un algoritmo de seguimiento de pared que permita al robot explorar entornos desconocidos sin colisionar con obstáculos. Para ello, hemos implementado una solución basada en una máquina de estados combinada con un control proporcional-derivativo (PD) para mantener una distancia constante a la pared derecha o izquierda en función de la situación del robot y de los giros que esté haciendo en cada instante. Por tanto, a partir de ahora nos referiremos a "seguir la pared", a seguir cualquiera de las dos paredes.

## 4.1. Inicialización

Algunas de las variables y constantes que utilizaremos en nuestro algoritmo de control de comandos, son los siguientes:

**Código 4.0: Constantes: TODO - 2.14**

```
class WallFollower:
    def __init__(self, dt: float, logger=None, simulation: bool = False) ->
    None:
        self._dt: float = dt
        self._logger = logger
        self._simulation: bool = simulation
        self._front_distance_threshold = 0.23
        self.expected_turning_distance = self._front_distance_threshold * np
    .sqrt(2) * 1.3
        self._x_vel = 0.15
        self._w_vel = 0.0
        self._front_dist = 0
        self._right_dist = 0
        self._left_dist = 0
        self._last_right_error = 0
        self._last_left_error = 0
        self._right_dist_error = 0
        self._left_dist_error = 0
        self._front_dist_error = 0
        self._wall_dist_target= 0.2
        self._whole_path_width = 0.4
        self._Kp = 4
        self._Kd = 5
        self._followed_wall = "right"
        self._state = states.Front
```

## 4.2. Diseño del algoritmo

Nuestra estrategia de seguimiento de paredes se basa en tres estados principales que determinan el comportamiento del robot:

- **Front**: El robot avanza siguiendo la pared que le corresponda, mientras trata de mantener una distancia de seguridad respecto de dicha pared, lo cual se consigue mediante el control en lazo cerrado, y combinándose con estados de giro.
- **Turn_Right**: El robot gira a la derecha cuando encuentra una esquina/intersección o necesita ajustar su orientación, para situarse a la distancia correcta de la pared.
- **Turn_Left**: El robot gira a la derecha cuando encuentra una esquina/intersección o necesita ajustar su orientación, para situarse a la distancia correcta de la pared.

Para detectar la distancia del robot a las paredes, utilizamos rangos específicos del array de medidas del LiDAR. Para las funcionalidades que necesitamos, hemos definido tres zonas de interés:

**Código 4.1: Extracción de distancias relevantes del LiDAR TODO - 2.14**

```
def compute_commands(self, z_scan: list[float], z_v: float, z_w: float) ->
    tuple[float, float]:
    # TODO: 2.14. Complete function body with your code (compute v and w).

    self._front_dist = self._safe_min(list(z_scan[0:5]) + list(z_scan[-5:]))
    self._right_dist = self._safe_min(z_scan[(3 * len(z_scan) // 4) - 5 : (3
     * len(z_scan) // 4) + 5])
    self._left_dist = self._safe_min(z_scan[(1 * len(z_scan) // 4) - 5 : (1
    * len(z_scan) // 4) + 5])
```

## 4.3. Control proporcional-derivativo

En el estado `Front`, implementamos un control PD que ajusta la velocidad angular para mantener una distancia objetivo a la pared que esté siguiendo el robot. Este control calcula el error entre la distancia deseada y la distancia medida, y aplica una corrección proporcional al error y a su derivada:

**Código 4.2: Control PD para seguimiento de pared TODO - 2.14**

```
def _handle_front_move(self):
    if self._left_dist + self._right_dist >= self._whole_path_width:
        if self._right_dist <= self._left_dist:
```

```
            self._followed_wall = "right"
        else:
            self._followed_wall = "left"
    if self._front_dist <= self._front_distance_threshold:
        self._handle_turn()
        return 0.0, 0.0
```

Los parámetros del controlador fueron ajustados empíricamente:

- $K_p = 4$: Ganancia proporcional que determina la intensidad de la corrección
- $K_d = 5$: Ganancia derivativa que amortigua las oscilaciones
- Distancia objetivo: 0.23 m de la pared derecha

Tenemos la función get_w_vel(), para en función de si estamos siguiendo la pared derecha o izquierda:

**Código 4.3: Control PD para seguimiento de pared TODO - 2.14**

```
def get_w_vel(self):
        if self._followed_wall == "right":
            error, last_error = self._right_dist_error, self.
    _last_right_error
        elif self._followed_wall == "left":
            error, last_error = self._left_dist_error, self._last_left_error
        else:
            return 0.0
        return self._Kp * error + self._Kd * (
            (error - last_error) / self._dt
        )
```

El término derivativo se calcula como la diferencia entre el error actual y el anterior dividida por el periodo de muestreo, lo que proporciona una estimación de la velocidad de cambio del error.

## 4.4. Lógica de decisión en esquinas

Cuando el robot detecta un obstáculo frontal (distancia menor al umbral), debe decidir hacia qué lado girar. Implementamos una lógica que compara las distancias laterales:

**Código 4.4: Decisión de dirección de giro TODO - 2.14**

```
def _handle_turn(self):
    # Reset error
    self._right_dist_error = 0
    self._left_dist_error = 0
    diff = self._right_dist - self._left_dist  # Here maybe put a threshold
    if abs(diff) <= 0.05:
        self._state = random.choice([states.Turn_Left, states.Turn_Right])
    elif diff <= 0:  # The wall is at the right
        self._state = states.Turn_Left
        self._followed_wall = "right"
    elif diff > 0:  # The wall is at the left
        self._state = states.Turn_Right
        self._followed_wall = "left"
```

Si las distancias laterales son similares (diferencia menor a 5 cm), interpretamos que estamos en una intersección o callejón sin salida, y elegimos aleatoriamente la dirección de giro. En caso contrario, giramos hacia el lado que tiene la pared más cercana, lo que permite al robot seguir consistentemente la misma pared.

## 4.5. Ejecución de los giros

Los giros se implementan deteniendo el avance lineal y aplicando una velocidad angular constante:

**Código 4.5: Implementación de giros TODO - 2.14**

```
def _handle_turn_right(self):
    if self._state == states.Turn_Right:
        return 0.0, -0.5

def _handle_turn_left(self):
    if self._state == states.Turn_Left:
        return 0.0, 0.5
```

El robot permanece en el estado de giro hasta que la distancia frontal supera un valor calculado como $\sqrt{2} * 1.3$ veces el umbral frontal, lo que indica que ha completado el giro y puede volver a avanzar, entonces cuando se da esta condición cambiamos al estado Fixed_Front y sacamos el momento temporal en el que ha pasado:

**Código 4.6: Condición de salida del estado de giro TODO - 2.14**

```
elif self._state == states.Turn_Right:
    self._x_vel, self._w_vel = self._handle_turn_right()
    if self._front_dist >= self.expected_turning_distance and self.
    _front_dist <= self.expected_turning_distance_upper:
        self._fixed_time = time.perf_counter()
        self._state = states.Fixed_Front

elif self._state == states.Turn_Left:
    self._x_vel, self._w_vel = self._handle_turn_left()
    if self._front_dist >= self.expected_turning_distance and self.
    _front_dist <= self.expected_turning_distance_upper:
        self._fixed_time = time.perf_counter()
        self._state = states.Fixed_Front
```

## 4.6. Bucle principal del algoritmo TODO - 2.14

El método principal `compute_commands` orquesta toda la lógica:

**Código 4.7: Bucle principal del algoritmo de seguimiento**

```
def compute_commands(self, z_scan: list[float], z_v: float, z_w: float) ->
    tuple[float, float]:
    # TODO: 2.14. Complete the function body with your code (i.e., compute v
     and w).

    self._front_dist = self._safe_min(list(z_scan[0:5]) + list(z_scan[-5:]))

    self._right_dist = self._safe_min(z_scan[(3 * len(z_scan) // 4) - 5 : (3
     * len(z_scan) // 4) + 5])

    self._left_dist = self._safe_min(z_scan[(1 * len(z_scan) // 4) - 5 : (1
    * len(z_scan) // 4) + 5])

    try:
        self._right_dist_error = self._wall_dist_target - self._right_dist
        self._left_dist_error = - self._wall_dist_target + self._left_dist

    except Exception:
        self._right_dist_error = 0
        self._left_dist_error = 0

    if self._state == states.Front:
```

```
        self._x_vel, self._w_vel = self._handle_front_move()

    elif self._state == states.Turn_Right:
        self._x_vel, self._w_vel = self._handle_turn_right()

        if self._front_dist >= self.expected_turning_distance:

            self._state = states.Front

    elif self._state == states.Turn_Left:
        self._x_vel, self._w_vel = self._handle_turn_left()

        if self._front_dist >= self.expected_turning_distance:

            self._state = states.Front

    self._last_right_error = self._right_dist_error
    self._last_left_error = self._left_dist_error

    return self._x_vel, self._w_vel
```

Para el cálculo de las distancias nos apoyamos en la función *_safe_min* la cual nos permitirá controlar los valores faltantes y seleccionar el mínimo del haz que recibe:

**Código 4.8: Función de control de distancias TODO - 2.14**

```
def _safe_min(self, values, default=8.0):
    vals = [v for v in values if v is not None and not math.isinf(v) and not
     math.isnan(v)]
    return min(vals) if vals else default
```

## 4.7. Resultados en simulación

Probamos el algoritmo en dos escenarios de complejidad creciente:

### 4.7.1. El correpasillos

En el primer escenario (pasillo exterior simple), el robot demostró capacidad para seguir la pared de forma suave y consistente. El control PD mantuvo una distancia estable de aproximadamente 0.2 m de la pared, con oscilaciones mínimas gracias al término derivativo. El robot completó el recorrido sin colisiones.

### 4.7.2. Aprendiendo a volver a casa

Al llegar al final del pasillo, el robot detectó correctamente el obstáculo frontal y decidió girar. La lógica de comparación de distancias laterales que hemos implementado permitió al robot identificar que se trataba de un callejón sin salida y ejecutar un giro hasta encontrar camino libre. El robot completó exitosamente el recorrido de ida y vuelta.

### 4.7.3. Me estoy mareando

En el entorno interior más complejo, nuestro algoritmo demostró robustez en situaciones variadas: esquinas interiores, esquinas exteriores e intersecciones. El control PD se adaptó correctamente a los cambios de configuración del entorno, y la máquina de estados gestionó adecuadamente todas las transiciones. El robot exploró continuamente el laberinto en ambos sentidos sin colisionar.

# 5. La jungla de cartón

Una vez comprobado que el robot funciona correctamente en simulación, el siguiente paso es transferir el código al TurtleBot3 Burger real e intentar que resuelva los mismos laberintos, montados en el laboratorio. Para ello, copiamos al espacio de trabajo `tb3_ws` todos los paquetes utilizados en simulación salvo `amr_simulation`, que es sustituido por los nodos del robot físico.

## 5.1. Derivando, derivando, la velocidad vamos estimando

El TurtleBot3 Burger publica un topic denominado `/odom` que proporciona únicamente la odometría en posición (pose del robot), pero no incluye la odometría en velocidad que necesita nuestro algoritmo de seguimiento de pared. Para resolver este inconveniente, creamos un nuevo paquete `amr_turtlebot3` que contiene un nodo `odometry_node` encargado de estimar las velocidades lineal y angular derivando a partir de medidas consecutivas de posición.

El nodo se suscribe al topic `/odom`, extrae la posición $(x, y)$ y la orientación $\theta$ del robot en cada instante, y calcula las velocidades. La orientación se obtiene convirtiendo los cuaterniones del mensaje a ángulos de Euler mediante la función `quat2euler` de la librería `transforms3d`. La información calculada se incorpora al mensaje original de odometría y se publica en el topic `/odometry`.

**Código 5.0: Nodo de odometría para el robot real - Sección 5.1**

```
class OdometryPubSub(Node):

  def __init__(self):
```

```
        super().__init__('odometry_pub_sub')

        self._prev_x: float | None = None
        self._prev_y: float | None = None
        self._prev_theta: float | None = None
        self._prev_time: float | None = None

        self._odom_publisher = self.create_publisher(Odometry, 'odometry', 10)
        self._odometry_subscriber = self.create_subscription(
            Odometry, "odom", self.add_vel_odometry, 10
        )

    def add_vel_odometry(self, msg):
        # Extract position
        x = msg.pose.pose.position.x
        y = msg.pose.pose.position.y

        # Extract orientation (quaternion -> euler yaw)
        qw = msg.pose.pose.orientation.w
        qx = msg.pose.pose.orientation.x
        qy = msg.pose.pose.orientation.y
        qz = msg.pose.pose.orientation.z
        _, _, theta = quat2euler((qw, qx, qy, qz))

        # Current time in seconds
        t = msg.header.stamp.sec + msg.header.stamp.nanosec * 1e-9

        if self._prev_time is not None:
            dt = t - self._prev_time

            if dt > 0.0:
                # Linear velocity: project displacement onto robot heading
                dx = x - self._prev_x
                dy = y - self._prev_y
                avg_theta = (self._prev_theta + theta) / 2.0
                v = (dx * math.cos(avg_theta) + dy * math.sin(avg_theta)) / dt

                # Angular velocity: derivative of orientation
                dtheta = math.atan2(
                    math.sin(theta - self._prev_theta),
                    math.cos(theta - self._prev_theta),
                )
                w = dtheta / dt

                # Fill in the twist fields of the original message
                msg.twist.twist.linear.x = v
```

```
                msg.twist.twist.angular.z = w

                # Publish on /odometry
                self._odom_publisher.publish(msg)

        # Store for next iteration
        self._prev_x = x
        self._prev_y = y
        self._prev_theta = theta
        self._prev_time = t
```

### Calculando la velocidad lineal

Para saber a qué velocidad se mueve el robot, podríamos simplemente dividir la distancia recorrida por el tiempo. Pero hay un problema: cuando el robot está girando mientras avanza, necesitamos considerar qué parte de ese movimiento es realmente avance hacia adelante.

La solución es tomar la dirección "promedio" en la que apuntaba el robot durante ese pequeño intervalo de tiempo. Usamos $\bar{\theta} = \frac{\theta_{k-1}+\theta_k}{2}$, que básicamente es el punto medio entre su orientación anterior y la actual. Luego proyectamos su desplazamiento sobre esa dirección:

$$v = \frac{\Delta x \cos\bar{\theta} + \Delta y \sin\bar{\theta}}{\Delta t} \tag{9}$$

Esto nos da una medida más precisa de su verdadera velocidad de avance.

### Calculando la velocidad angular

Para saber qué tan rápido está girando el robot, necesitamos medir cuánto cambió su orientación. Aquí debemos tener muy en cuenta que no podemos simplemente restar los ángulos directamente, porque los ángulos "dan la vuelta" (cuando pasamos de 179° a -179°, en realidad solo giraste 2°, no 358°).

Para evitar este problema, usamos una función matemática que "entiende" cómo funcionan los ángulos y siempre nos da la diferencia más corta:

$$\omega = \frac{\text{atan2}(\sin(\theta_k - \theta_{k-1}),\ \cos(\theta_k - \theta_{k-1}))}{\Delta t} \tag{10}$$

Esto nos asegura que el resultado siempre esté entre $-\pi$ y $\pi$, sin saltos bruscos.

**Código 5.1: Archivo de lanzamiento para el robot real - lab02.launch.py**

```
def generate_launch_description():
    simulation = False  # boolean to switch from sim to physical robot
    dt = 0.05  # sampling period
```

```
    # start = (1.0, -1.0, 0.5 * math.pi)  # Outer corridor
    # start = (0.6, -0.6, 1.5 * math.pi)  # Inner corridor

    wall_follower_node = LifecycleNode(
        package="amr_control",
        executable="wall_follower",
        name="wall_follower",
        namespace="",
        output="screen",
        arguments=["--ros-args", "--log-level", "INFO"],
        parameters=[{"simulation": simulation, "dt": dt}],
    )

    lifecycle_manager_node = Node(
        package="amr_bringup",
        executable="lifecycle_manager",
        output="screen",
        arguments=["--ros-args", "--log-level", "WARN"],
        parameters=[
            {
                "node_startup_order": (
                    "wall_follower",
                )
            }
        ],
    )

    odometry_node = Node(
        package="amr_turtlebot3",
        executable="odometry_node",
        output="screen",

    )

    return LaunchDescription(
        [
            wall_follower_node,
            odometry_node,
            lifecycle_manager_node,  # Must be launched last
        ])
```

Además cambiamos el archivo de lanzamiento de simulación, aquí no incluimos el nodo de CoppeliaSim y añadimos en su lugar el nodo de odometría. El parámetro `simulation` se establece a `False`.

## 5.2. Saliendo de TRON: De la simulación al mundo real

Como estamos viendo, transferir el código funcional en simulación al robot físico requiere abordar múltiples modificaciones. A continuación se comentan algunas de ellas.

### 5.2.1. Cambio del tipo de mensaje en `/cmd_vel`

En simulación, el publicador de comandos de velocidad utiliza mensajes de tipo `TwistStamped`, que incluyen una cabecera con marca de tiempo para facilitar la sincronización con el simulador. Sin embargo, el TurtleBot3 real espera mensajes de tipo `Twist` estándar. Además, el criterio de signos para la velocidad angular es dextrógiro (positivo en sentido horario), contrario al criterio anterior. Para manejar ambas situaciones de forma limpia, utilizamos el parámetro booleano `simulation` que ya hemos introducido antes:

**Código 5.2: Publicador adaptado para simulación y robot real**

```
# TODO: 2.10. Create the /cmd_vel velocity commands publisher
if self._simulation:
    self._commands_publisher = self.create_publisher(
        msg_type=TwistStamped,
        topic = "cmd_vel",
        qos_profile=10
    )

else:
    self._commands_publisher = self.create_publisher(
        msg_type=Twist,
        topic = "cmd_vel",
        qos_profile=10
    )
```

La función de publicación también se adapta de igual forma según el modo de ejecución:

**Código 5.3: Publicación de comandos adaptada**

```
def _publish_velocity_commands(self, v: float, w: float) -> None:
  # TODO: 2.11.
    if self._simulation:
        # We create a TwistStamped() messages and add the v and w info
        msg = TwistStamped()
        msg.header.stamp = self.get_clock().now().to_msg()
        msg.twist.linear.x = v
        msg.twist.angular.z = w
```

```
        # We publish the message through the commands publisher
        self._commands_publisher.publish(msg)

    else:

        # We create a Twist() messages and introduce the info of v and w
        msg = Twist()
        msg.linear.x = v
        msg.angular.z = -w  # Positive sign in clockwise

        # We publish the message through the commands publisher
        self._commands_publisher.publish(msg)
```

### 5.2.2. Arquitectura asíncrona con temporizador

En simulación, el tiempo avanza de forma discreta: cada vez que el simulador recibe un comando de velocidad, avanza exactamente un periodo de muestreo $\Delta t$. Esto significa que si el cálculo de los comandos se retrasa, nuestro robot simplemente espera. En el robot real, sin embargo, el tiempo avanza de forma continua y el robot sigue moviéndose aunque no reciba comandos nuevos.

Para garantizar un control periódico en el robot real, implementamos una arquitectura con dos callbacks separados como nos indica el enunciado:

- Un callback de almacenamiento que captura las medidas del LiDAR y la odometría de forma sincronizada y las guarda en atributos, sobreescribiendo los valores anteriores.
- Un callback asociado a un temporizador que ejecuta el algoritmo de control cada $\Delta t$ utilizando las medidas más recientes disponibles.

**Código 5.4: Arquitectura asíncrona para el robot real**

```
# We register the callback depending on simulation/real robot behavior
if not self._simulation:
    # Store latest sensor data
    self._latest_odom_msg = None
    self._latest_scan_msg = None
    self._latest_pose_msg = PoseStamped()

    # Change sync callback to just store data
    ts.registerCallback(self._store_measurements_callback)

    # Timer to run control at dt rate
```

```
    self._timer = self.create_timer(dt, self._timer_callback)
else:
    ts.registerCallback(self._compute_commands_callback)

def _store_measurements_callback(self, odom_msg, scan_msg, pose_msg=
    PoseStamped()):
  """Stores the latest sensor measurements."""
  self._latest_odom_msg = odom_msg
  self._latest_scan_msg = scan_msg
  self._latest_pose_msg = pose_msg

def _timer_callback(self):
  """Timer callback. Runs control with latest measurements."""
  if self._latest_odom_msg is not None and self._latest_scan_msg is not None
    :
      self._compute_commands_callback(
          self._latest_odom_msg, self._latest_scan_msg, self.
    _latest_pose_msg
      )
```

Esta separación garantiza que el algoritmo de control se ejecute a una frecuencia constante, independientemente de cuándo lleguen las medidas de los sensores.

### 5.2.3. Ajuste del sincronizador de mensajes

En simulación, habíamos fijado el parámetro `slop` del sincronizador en un valor de 10 segundos, garantizar el funcionamiento sin problemas. En el robot físico, donde los sensores publican datos con temporización real, reducimos este valor a 0.15 segundos para garantizar que las medidas sincronizadas correspondan efectivamente al mismo instante:

**Código 5.5: Sincronizador ajustado para el robot real**

```
# Wait until receive all measurements and then we invoke the callback
ts = message_filters.ApproximateTimeSynchronizer(
    self._subscribers,
    queue_size=10,  # number of messages of each topic we need to receive
    until we are "completed"
    slop=0.15 # max delay in seconds to consider that 2 messages are able to
     be syncronized
)
```

### 5.2.4. Adaptaciones en el algoritmo de seguimiento de pared

El paso de simulación a robot real requirió también de ajustes significativos en el algoritmo de control que habíamos implementado inicialmente, motivados tanto por las diferencias físicas del hardware como por las perturbaciones del entorno real. A continuación describimos los cambios principales que hemos tenido que hacer:

**Nuevo estado: `Fixed_Front`.** En simulación, nuestra máquina de estados tenía tres estados (`Front`, `Turn_Right`, `Turn_Left`). En el robot real añadimos un cuarto estado `Fixed_Front` que actúa como periodo de estabilización tras completar un giro. El robot avanza en línea recta a 0.1 m/s durante 0.35 segundos antes de retomar el seguimiento de pared con el control PD. Este estado evita que el controlador reaccione bruscamente a las medidas del LiDAR inmediatamente después de un giro, momento en el cual las lecturas pueden ser ruidosas. Los valores tanto de velocidad como de tiempo usados en este nuevo estado, han sido seleccionados tras hacer numerosas pruebas y encontrar un funcionamiento adecuado por parte del robot.

**Código 5.6: Estado de estabilización tras giro (solo robot real)**

```
# States: added Fixed_Front for real robot
states = enum.Enum("states", "Front Turn_Right Turn_Left Fixed_Front")

if self._state == states.Front:
    self._x_vel, self._w_vel = self._handle_front_move(z_scan)

elif self._state == states.Turn_Right:
    self._x_vel, self._w_vel = self._handle_turn_right()

    if self._front_dist >= self.expected_turning_distance and self.
   _front_dist <= self.expected_turning_distance_upper:
        self._fixed_time = time.perf_counter()
        self._state = states.Fixed_Front
        self._logger.info(f"Front distance: {self._front_dist}")
        self._logger.info(f"state: {self._state}\n")

elif self._state == states.Turn_Left:
    self._x_vel, self._w_vel = self._handle_turn_left()

    if self._front_dist >= self.expected_turning_distance and self.
   _front_dist <= self.expected_turning_distance_upper:
        self._fixed_time = time.perf_counter()
        self._state = states.Fixed_Front
        self._logger.info(f"Front distance: {self._front_dist}")
        self._logger.info(f"state: {self._state}\n")
```

```
elif self._state == states.Fixed_Front:
    self._x_vel, self._w_vel = self._handle_fixed_front()

    if (time.perf_counter() - self._fixed_time) > 0.35:
        self._state = states.Front
```

**Reajuste de parámetros del control en lazo cerrado.** Las ganancias del controlador PD que habíamos implementado inicialmente en simulación se incrementaron significativamente para compensar la dinámica más lenta del robot real y el distinto periodo de muestreo:

| Parámetro | Simulación | Robot real |
|---|---|---|
| $K_p$ | 4.0 | 7.6 |
| $K_d$ | 5.0 | 6.0 |
| Umbral frontal | 0.23 m | 0.28 m |
| Velocidad de giro | $\pm$0.5 rad/s | $\pm$0.37 rad/s |

Cuadro 1: Comparación de parámetros entre simulación y robot real

El incremento de $K_p$ de 4.0 a 7.6 se debe a que el robot real responde de forma más amortiguada que el modelo simulado, por lo que necesita una corrección proporcional más fuerte. El umbral de distancia frontal se amplió de 0.23 m a 0.28 m para dar más margen de frenado, compensando la inercia del robot físico. También se redujo la velocidad de giro para conseguir un control más suave y estable al tratar de corregir la distancia del robot a la pared seguida.

**Saturación de la velocidad angular.** En simulación, la salida del controlador PD se aplica directamente como velocidad angular. En el robot real, hemos añadido una saturación explícita a $\pm$0.34 rad/s para evitar movimientos bruscos que podrían desestabilizar el robot:

**Código 5.7: Saturación de la velocidad angular (robot real)**

```
def get_w_vel(self):
    if self._followed_wall == "right":
        error, last_error = self._right_dist_error, self._last_right_error

    elif self._followed_wall == "left":
        error, last_error = self._left_dist_error, self._last_left_error

    else:
        return 0.0
```

```
    value = self._Kp * error + self._Kd * (
        (error - last_error) / self._dt
    )

    if value < 0:
        w = max(value, -0.34)
        self._logger.info(f"Value of W: {w}")
        return w

    else:
        w = min(value, 0.34)
        self._logger.info(f"Value of W: {w}")
        return w
```

**Simplificación de la lectura del LiDAR.** En simulación, donde el LiDAR genera siempre 240 haces de forma determinista, utilizábamos ventanas de 10 rayos alrededor de las direcciones de interés y calculamos el mínimo de cada ventana mediante la función `_safe_min`. En el robot real, simplificamos la lectura utilizando un solo rayo en cada dirección así para evitar ruido, sobre todo en las situaciones de varios giros seguidos. Tras varias pruebas, hemos llegado a la conclusión de que nos funciona mejor de este modo.

**Código 5.8: Lectura del LiDAR: simulación vs. robot real**

```
# Simulation: window of 10 rays with safe minimum
self._front_dist = self._safe_min(list(z_scan[0:5]) + list(z_scan[-5:]))
self._right_dist = self._safe_min(
  z_scan[(3*len(z_scan)//4)-5 : (3*len(z_scan)//4)+5])
self._left_dist = self._safe_min(
  z_scan[(1*len(z_scan)//4)-5 : (1*len(z_scan)//4)+5])

# Real robot: single ray at each cardinal direction
self._front_dist = z_scan[-1]
self._right_dist = z_scan[3 * len(z_scan) // 4]
self._left_dist = z_scan[len(z_scan) // 4]
```

En simulación, al utilizar índices relativos al tamaño del array (`len(z_scan) // 4`, `3 * len(z_scan) // 4`), nuestro código se adapta automáticamente al número variable de haces que genera el LiDAR del robot real, ya que las direcciones de interés se calculan como fracciones del total. No obstante, como hemos dicho, en el robot físico ya no lo usamos.

**Condición de salida de giro con límite superior.** En simulación, el robot salía del estado de giro simplemente cuando la distancia frontal superaba el umbral calculado. En el robot real, hemos añadido un límite superior de 0.75 m para evitar que el robot continúe girando cuando detecta un espacio abierto demasiado amplio que no corresponde a la salida correcta del giro:

**Código 5.9: Condición de salida de giro: simulación vs. robot real**

```
# Simulation: single threshold
if self._front_dist >= self.expected_turning_distance:
  self._state = states.Front

# Real robot: bounded range with transition to Fixed_Front
self.expected_turning_distance_upper = 0.75
if (self._front_dist >= self.expected_turning_distance and
  self._front_dist <= self.expected_turning_distance_upper):
  self._fixed_time = time.perf_counter()
  self._state = states.Fixed_Front
```

## 5.3. Dificultad: Soy muy joven para chocarme

### Vídeo #1

Este vídeo se adjunta en la entrega junto con el resto de códigos y materiales pedidos. Tiene el nombre de `RobotsMovilesAutonomos_P2_G21_Physical_Outer_Corridor` .

## 5.4. Dificultad: Oye, no seas tan duro

### Vídeo #2

Este vídeo se adjunta en la entrega junto con el resto de códigos y materiales pedidos. Tiene el nombre de `RobotsMovilesAutonomos_P2_G21_Physical_Inner_Corridor` .

# 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).
- **Cálculo de la velocidad angular en un entorno real:** Nos apoyamos en la IA en este caso para hacer la conversión de ángulos a velocidades angulares ya que en un principio, en ciertas situaciones, el robot con un giro pequeño leía un giro de 360 grados y no encontrábamos con la solución a este problema. Ayudarnos de la explicación que nos proporcionó la IA nos permitió detectar y solventar dicho problema.