Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Tarea 1

Introducción

Integrantes: Sebastián Herrera T.
Profesores: Felipe Inostroza Ferrari
Javier Ruiz del Solar
Auxiliar: Francisco Leiva C.
Ayudantes: Eduardo Jorquera C.
Gonzalo Olguín M.

Fecha de realización: 4 de abril de 2025
Fecha de entrega: 4 de abril de 2025
Santiago de Chile

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Índice de Códigos

1. Desplazamiento y Parámetros del Robot

Para calcular los parámetros del robot requeridos (r, l) se requieren utilizar las ecuaciones de movimiento vistas en clases, para simplicidad se utilizaron aquellas que asumen velocidad constante en el tiempo:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x + v\cos(\theta)\delta t \\ y + v\sin(\theta)\delta t \\ \theta \end{bmatrix} \tag{1}$$

Y aquellas donde $v_l = -v_r = v$:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta + 2v\delta t/l \end{bmatrix} \tag{2}$$

Luego, se asume también que el robot se encuentra en el origen del plano global, recordando a la vez que $v = \omega \cdot r$:

$$si\ v_l = v_r = v: \begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} \omega \cdot r\cos(\theta)\delta t \\ \omega \cdot r\sin(\theta)\delta t \\ \theta \end{bmatrix} \tag{3}$$

$$si\ v_l = -v_r = v: \begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ 2 \cdot \omega \cdot r\delta t/l \end{bmatrix} \tag{4}$$

Ahora, se extraerán las ecuaciones para x' y θ' con sus respectivas condiciones, así se arma el siguiente sistema de ecuaciones:

$$\begin{aligned} x' &= \omega \cdot r\cos(\theta)\delta t \\ \theta' &= 2 \cdot \omega \cdot r\delta t/l \end{aligned} \tag{5}$$

Por último, se asumen x', θ', ω valores conocidos, así se podrá resolver el sistema para r, l. Para esto se definió $\omega = 1[\frac{Rad}{s}]$ mientras que para x' y θ' se realizaron 10 movimientos con parámetros $\delta t = 4[s]$ y ω definido anteriormente, así para cada variable se logra obtener un valor más realista.

Tabla 1: Resultados de desplazamiento en x.

Nº	$\Delta x[m]$
1	0.2
2	0.05
3	0.2
4	0.2
5	0.19
6	0.07
7	0.2
8	0.1
9	0.2
10	0.13

Tabla 2: Resultados de desplazamiento en θ.

Nº	$\Delta\theta[Rad]$
1	1.25
2	1.25
3	1.27
4	1.25
5	1.24
6	1.26
7	1.25
8	0.49
9	1.25
10	1.25

Como se aprecia en las tablas anteriores, los desplazamientos normalmente tienden a un valor y algunas veces se alejan de la mediana. En un comienzo se pensó que podría ser el controlador de ejemplo entregado, pero posteriormente a cambiar este controlador se obtuvieron resultados similares, después de más experimentación el problema de esta variabilidad es debida al comportamiento de *Docker* y *Gazebo*, ya que *Gazebo* funciona con tiempo real, y *Docker* no estaba siendo capaz de mantener sincronizado el tiempo del contenedor con el del sistema operativo ocacionando estos desfases entre comunicaciones de *ROS2*. Se menciona esto porque fue un problema que **No pudo ser solucionado en su mayoría**, por lo que hay cierto margen de error en lo que resta de la tarea.

Entonces, según las medianas de cada tabla, se utilizarán los valores $x' = 0.2$ y $\theta' = 1.25$. Reemplazando en el sistema de ecuaciones (5) se obtiene finalmente:

$$\begin{aligned} r &= 0.05[m] \\ l &= 0.32[m] \end{aligned} \tag{6}$$

2. Cinemática Inversa

Para la cinemática inversa, se basará en las **coordenadas locales** del robot, por lo que se utilizaran principalmente las ecuaciones obtenidas en (3) y (4). En el caso de la cinemática inversa se realizan casi los mismos supuestos, la diferencia es que conociendo los valores de r y l, ahora se puede despejar δt[1]. De esta forma, la cinemática inversa calculará el tiempo que necesita moverse el robot a velocidad constante para alcanzar la pose deseada. También el orden de las acciones estará dictaminada, en parte, por las convenciones de movimiento de robot, en orden:

1. Rotar en dirección al punto de destino (θ' calculado apartir de $atan2(y, x)$).
2. Avanzar en linea recta al punto.
3. Rotar hasta llegar a la pose deseada.

Las ecuaciones quedarían:

$$\begin{array}{c} para\ desplazamiento\ \delta t = \frac{x'}{\omega \cdot r \cdot cos(\theta)}\ o\ \frac{y'}{\omega \cdot r \cdot sin(\theta)}\ según\ corresponda \\ para\ rotar\ \delta t = \frac{\theta' \cdot l}{2 \cdot \omega \cdot r} \end{array} \tag{7}$$

Durante los testeos de la cinemática inversa, se observaron diferencias entre lo deseado y lo obtenido, esta diferencia posiblemente sea debida a las aproximaciones que realiza *Gazebo*, aproximaciones y supuestos que se realizaron en la sección anterior al calcular r y l. No obstante, para intentar corregir esto, se agrego un ponderador constante que multiplica directamente las ecuaciones. Obviamente, hay un ponderador k_f para el desplazamiento y otro k_r para las rotaciones.

Estos valores fueron calculados en un principio como el ponderado que le faltó (o sobró) a un movimiento determinado en llegar a su objetivo deseado, posteriormente se hizo fine tunning a este ponderador, aún así, no se aleja tanto de lo calculado inicialmente. Para el desplazamiento se realizó un movimiento a $(x = 1.0, y = 0.0)$ obteniendo que en el simulador el robot se desplazó a $(x = 0.99, y = 0.0)$, entonces, al valor real le faltó un 1 % para alcanzar el valor deseado, por lo tanto $k_f = 1.01$, que posterior al finetunning resultó $k_f = 1.00802$. Para la rotación se realizó lo análogo, se desea una rotación de $\pi[Rad]$ y se obtuvo $3.22[Rad]$, por lo que el valor real se sobrepasó un 2.5 % obteniendo así que $k_r = 0.975$, que finalmente después del finetunning quedó en $k_r = 0.97595$.

Finalmente las ecuaciones quedan:

1. Rotar en dirección al punto de destino (θ' calculado apartir de $atan2(y, x)$).
2. Avanzar en linea recta al punto.
3. Rotar hasta llegar a la pose deseada.

Las ecuaciones quedarían:

$$\begin{array}{c} para\ desplazamiento\ \delta t = \frac{x' \cdot k_f}{\omega \cdot r \cdot cos(\theta)}\ o\ \frac{y' \cdot k_f}{\omega \cdot r \cdot sin(\theta)}\ según\ corresponda \\ para\ rotar\ \delta t = \frac{\theta' \cdot l \cdot k_r}{2 \cdot \omega \cdot r} \end{array} \tag{8}$$

En código:

Código 1: Cinemática Inversa

```python
def inverse_kinematics(self, x: float = 0.0, y: float = 0.0, theta: float = 0.0, fwd_vel: float = 3.0,
    ↪ rt_vel: float = 0.5):
   # Por convención: 1) Rotar hacia el punto de destino, 2) Traslación al punto, 3) Rotar hacia la
    ↪ pose
   def rotation_time(theta):
      k_r = 0.97595
      self.get_logger().info(f"Tiempo estimado de rotación: {(theta*self.l)*k_r/(2*rt_vel*self.r)}")
      return (theta*self.l)*k_r/(2*rt_vel*self.r)
   def forward_time(x, theta, y_axis: bool = False):
      k_f = 1.00802
      self.get_logger().info(f"Tiempo estimado de traslación: {x*k_f/(fwd_vel*self.r*math.cos(theta)
    ↪ ) if not y_axis \
                                                  else (x*k_f)/(fwd_vel*self.r*math.sin(theta))}")
      return k_f*(x)/(fwd_vel*self.r*math.cos(theta)) if not y_axis \
            else k_f*(x)/(fwd_vel*self.r*math.sin(theta))

   # Verificar si existe un punto (x,y) de lo contrario, no tendría sentido realizar los pasos 1) y 2)
   if (x != 0.0) or (y != 0.0):
      # 1)
      temp_theta = math.atan2(y, x) # Ángulo temporal para mirar en dirección al punto deseado
      self.rotate(abs(rotation_time(abs(temp_theta))), True if temp_theta > 0 else False)
      # 2)
      self.move_forward(abs(forward_time(abs(x), abs(temp_theta))), 3.0) if x!=0.0 \
      else self.move_forward(abs(forward_time(abs(y), abs(temp_theta), True)), 3.0)
   # 3)
   if theta != 0.0:
      self.rotate(rotation_time(abs(theta)), True if theta > 0 else False)
```

Cabe mencionar que el código posee lógica adicional para manejar la mayoría de casos especiales.

3. Odometría

Para implementar la odometría, se desarrolló un nodo llamado *OdomPublisher* que realiza las siguientes funciones:

1. **Subscripción a datos de las ruedas**: Se subscribe al tópico */joint_states* para obtener las velocidades angulares de las ruedas izquierda y derecha.
2. **Cálculo de odometría**: Utiliza las velocidades de las ruedas para calcular la velocidad lineal y angular del robot mediante las ecuaciones del modelo de cinemática directa:
   - Velocidad lineal: $v = \frac{v_r + v_l}{2}$
   - Velocidad angular: $\omega = \frac{v_r - v_l}{l}$
3. **Actualización de la pose**: Integra las velocidades para actualizar la posición y orientación del robot:

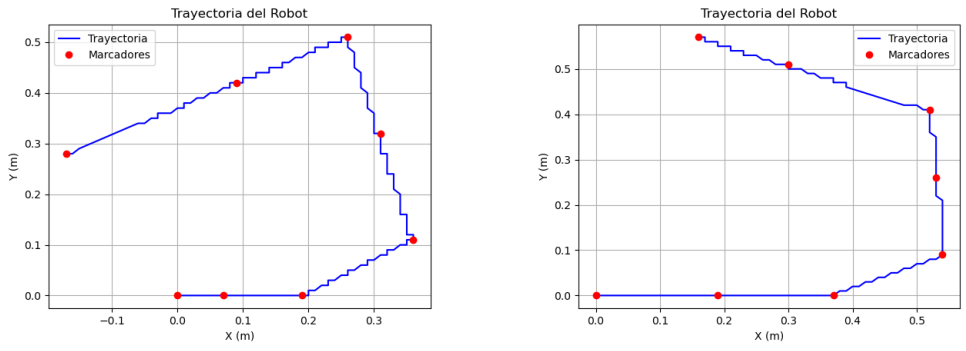
Código 2: Integración de velocidades

```
    delta_x = v * math.cos(self.theta) * dt
    delta_y = v * math.sin(self.theta) * dt
    delta_theta = omega * dt

```

4. **Publicación de transformaciones**: Utiliza *TransformBroadcaster* para publicar la transformación entre los marcos */odom* y */base_link*, lo que permite visualizar la trayectoria del robot.
5. **Publicación de mensajes de odometría**: Además de la transformación, el nodo publica mensajes de tipo *Odometry* en el tópico */odom* con la información completa de pose y velocidad.

Para la implementación se utilizaron los parámetros r y l estimados en la parte 1 de esta tarea. Obteniendo así los siguientes resultados:



(a) (b)

Figura 1: Trayectoría registrada por la Odometría (Con fallos)

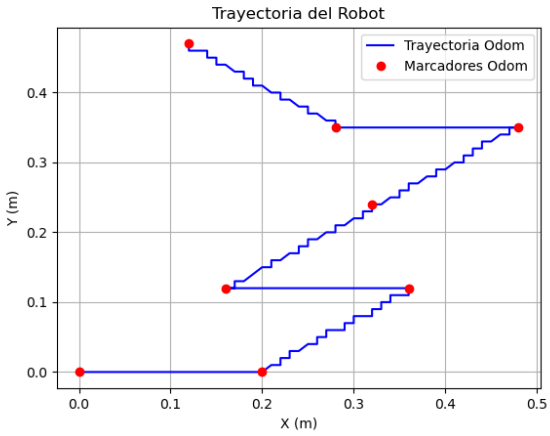


Figura 2: Trayectoría registrada por la Odometría a tiempo real.

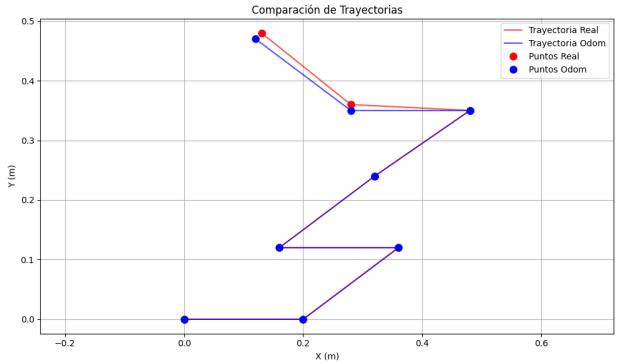


Figura 3: Trayectoría registrada por la Odometría vs Trayectoría real.

El propósito de mostrar la Figura 1 es para demostar lo dicho en la sección anterior con respecto a los fallos producidos por la mala sincronización entre *Docker* y *Gazebo*, obteniendo resultados muy distintos a pesar de haber realizado los mismos movimientos.

Volviendo a los resultados en un ambiente más controlado (Figura 3), se aprecia que la odometría basada únicamente en la cinemática y las velocidades de las ruedas acumula error con el tiempo debido a posibles deslizamientos, aproximaciones y otras imprecisiones del modelo. Sin embargo, proporciona una estimación razonable de la pose del robot para trayectorias cortas, es muy probable que este error aumente exponencialmente mientas más comandos de movimiento se entreguen en una misma ejecución.

Esta implementación de odometría basada en encoders (o velocidades de ruedas en este caso) es una buena alternativa para la navegación de robots móviles, proporcionando una estimación de la posición relativa al punto de inicio. Esta técnica tiene limitaciones obvias debido a la acumulación de errores, pero sirve como base para sistemas más complejos que podrían incorporar sensores adicionales (como LiDARs) para corrección de pose.

El método de integración utilizado (Euler) es simple pero efectivo para frecuencias de actualización altas. En aplicaciones más críticas, podría considerarse el uso de métodos de integración más precisos.

Cabe mencionar que la normalización del ángulo en el rango [-π, π] se realizó porque gazebo funciona con ese rango de ángulos.

4. Rodear las cajas

Para la última misión de rodear 10 veces las cajas utilizando la cinemática inversa, se utilizaron 4 comandos para completar una vuelta, por lo que sólo bastó con repetir esta secuencia 10 veces.

Código 3: Secuencia de comandos para rodear las cajas

```
controller.inverse_kinematics(x=0.0, y=3.75)
controller.inverse_kinematics(x=0.0, y=2.9)
controller.inverse_kinematics(x=0.0, y=3.75)
controller.inverse_kinematics(x=0.0, y=2.9)
```

Cómo la cinemática inversa funciona con el sistema local de coordenadas del robot, no es necesario utilizar el otro eje (x), ya que como se mostró en el item 2, el robot rotará automáticamente en dirección al punto de destino, y al estar en el eje y, este rotará 45º en sentido antihorario quedando de frente con el eje y, permitiendo un movimiento directo.

De esta forma se obtuvieron los siguientes resultados:

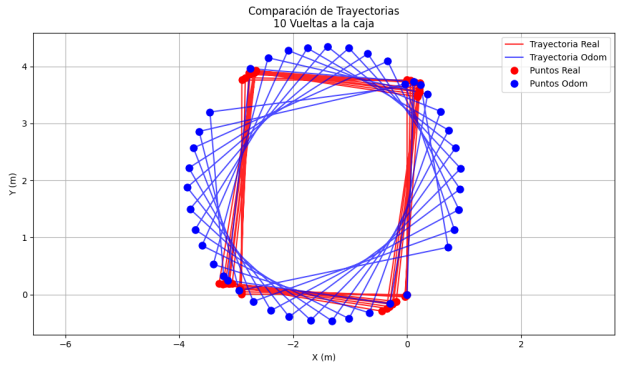


Figura 4: Trayectoría Odom vs Real para las 10 vueltas a las cajas.

Tabla 3: Poses de inicio y final en para las 10 vueltas.

Pose	x_{real}	y_{real}	θ_{real}	x_{odom}	y_{odom}	θ_{odom}
Inicial	0.0	0.0	0.0	0.0	0.0	0.0
Final	-0.44	-0.29	-0.31	-3.22	0.33	1.43

Si se observa la figura 4 y la tabla 3, se observa que ninguno llegó a la pose original, al verificar la trayectoría obtenida por la odometría, ésta tuvo un error bastante significante en comparación a la trayectoria real. Esto tiene varias causantes, además de las descritas en las secciones anteriores

(acumulación de errores por aproximaciones, suposiciones, etc.) se debe considerar que la cinemática inversa tenía incluidos los ponderadores k_f y k_r, los cuales no fueron incluidos en los calculos de la odometría. Analizando un poco más las trayectorías, se observa que el error tiene un crecimiento lineal, ya que se observa como cada vuelta se desplaza la misma distancia en cada iteración.

Para mejorar el rendimiento en este caso, se podría implementar que el controlador se suscriba a la odometría y así utilizar coordenadas globales en vez de locales (o ambas, inclusive). De esta forma se tendría más precisión sin tener que añadir sensores adicionales.

Por último, si se compara lo obtenido con la cinemática inversa y se piensa realizar lo mismo pero con cinemática directa, se notará que la cinemática directa podría llegar a ser mucho más tediosa que la inversa, ya que sólo se entrega ángulos para cada motor, lo que podría derivar en muchos más comandos en comparación a la cinemática inversa, en la cual sólo fueron 4 comandos que se repitieron 10 veces consecutivamente.

Estos resultados permiten comprobar que a pesar de que utilizar la cinemática directa para calcular la posición actual del robot es un buen comienzo, al suponer y aproximar se produce una acumulación de errores que crece enormemente mientras más acciones realice el robot. Es por esto que se necesita algo más para apoyar la odometría, ahí es donde entran en juego distintos tipos de sensores, normalmente en estos tipos de proyectos se suelen usar LiDARs, Sonares e incluso algoritmos visuales.

Referencias

[1] Dudek, G. y Jenkin, M., "Computational principles of mobile robotics," 2010.

5. Anexo

5.1. new_controller.py

```python
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from std_msgs.msg import Float64
from collections import deque
from std_srvs.srv import Empty

import math

class WheelController(Node):
    def __init__(self):
        super().__init__('wheel_controller')
        self.node = rclpy.create_node('wheel_controller')
        self.mark_position_client = self.node.create_client(Empty, '/mark_position')

        self.left_pub = self.create_publisher(Float64, 'left_wheel_cmd_vel', 10)
        self.right_pub = self.create_publisher(Float64, 'right_wheel_cmd_vel', 10)

        self.get_logger().info("Controller initialized. Ready for commands.")

        # Sistema de cola de acciones
        self.action_queue = deque()
        self.current_action = None
        self.action_timer = None

        # Parámetros del robot según lo calculado en la parte 1)
        self.r = 0.05
        self.l = 0.32

        # Timer
        self.process_timer = self.create_timer(0.1, self.process_queue)

    # Avanzar en linea recta
    def move_forward(self, duration: float, speed: float = 1.0):
        self.action_queue.append(('FORWARD', duration, speed))
        self.get_logger().info(f"Added FORWARD action: {speed} m/s for {duration}s")

    # Rotar
    def rotate(self, duration: float, clockwise: bool = True):
        self.action_queue.append(('ROTATE', duration, clockwise))
        direction = "clockwise" if clockwise else "counter-clockwise"
```

```
        self.get_logger().info(f"Added ROTATION action: {direction} for {duration}s")

    # Cinemática inversa, asume vel angular fija y cte = |3.0| para desplazamientos y vel =
    ↪  0.5 para rotaciones
    # Calcula el tiempo necesario para llegar a los puntos necesarios.
    # Se basa en el sistema de coordenadas del ROBOT.
    def inverse_kinematics(self, x: float = 0.0, y: float = 0.0, theta: float = 0.0, fwd_vel: float
    ↪  = 3.0, rt_vel: float = 0.5):
        # Por convención: 1) Rotar hacia el punto de destino, 2) Traslación al punto, 3) Rotar
        ↪  hacia la pose
        def rotation_time(theta):
            k_r = 0.97595
            self.get_logger().info(f"Tiempo estimado de rotación:
            ↪  {(theta*self.l)*k_r/(2*rt_vel*self.r)}")
            return (theta*self.l)*k_r/(2*rt_vel*self.r)
        def forward_time(x, theta, y_axis: bool = False):
            k_f = 1.00802
            self.get_logger().info(f"Tiempo estimado de traslación:
            ↪  {x*k_f/(fwd_vel*self.r*math.cos(theta)) if not y_axis \
                                            else (x*k_f)/(fwd_vel*self.r*math.sin(theta))}")
            return k_f*(x)/(fwd_vel*self.r*math.cos(theta)) if not y_axis \
                  else k_f*(x)/(fwd_vel*self.r*math.sin(theta))

        # Verificar si existe un punto (x,y) de lo contrario, no tendría sentido realizar los pasos
        ↪  1) y 2)
        if (x != 0.0) or (y != 0.0):
            # 1)
            temp_theta = math.atan2(y, x) # Ángulo temporal para mirar en dirección al
            ↪  punto deseado
            self.rotate(abs(rotation_time(abs(temp_theta))), True if temp_theta > 0 else False)
            # 2)
            self.move_forward(abs(forward_time(abs(x), abs(temp_theta))), 3.0) if x!=0.0 \
            else self.move_forward(abs(forward_time(abs(y), abs(temp_theta), True)), 3.0)
        # 3)
        if theta != 0.0:
            self.rotate(rotation_time(abs(theta)), True if theta > 0 else False)


    def process_queue(self):
        if self.current_action is None and self.action_queue:
            # Comenzar nueva acción
            self.current_action = self.action_queue.popleft()
            action_type, duration, param = self.current_action

            if action_type == 'FORWARD':
                speed = param
                msg = Float64()
```

```
            msg.data = speed
            self.left_pub.publish(msg)
            self.right_pub.publish(msg)
            self.get_logger().info(f"Executing FORWARD: {speed} m/s")

        elif action_type == 'ROTATE':
            clockwise = param
            base_speed = 0.5
            left_msg = Float64()
            right_msg = Float64()
            left_msg.data = -base_speed if clockwise else base_speed
            right_msg.data = base_speed if clockwise else -base_speed

            self.left_pub.publish(left_msg)
            self.right_pub.publish(right_msg)
            direction = "clockwise" if clockwise else "counter-clockwise"
            self.get_logger().info(f"Executing ROTATE: {direction}")

        # Timer para la acción
        self.action_timer = self.create_timer(duration, self.complete_current_action)

    def complete_current_action(self):
        if self.action_timer:
            self.action_timer.cancel()

        stop_msg = Float64()
        stop_msg.data = 0.0
        self.left_pub.publish(stop_msg)
        self.right_pub.publish(stop_msg)

        self.get_logger().info(f"Completed action: {self.current_action[0]}")
        if self.current_action[0] == "FORWARD":
            self.call_mark_position_service()
        self.current_action = None
        self.action_timer = None

    def clean_shutdown(self):
        self.process_timer.cancel()
        if self.action_timer:
            self.action_timer.cancel()

        # Detener cualquier movimiento
        stop_msg = Float64()
        stop_msg.data = 0.0
        self.left_pub.publish(stop_msg)
        self.right_pub.publish(stop_msg)
```

```
        self.destroy_node()
        rclpy.shutdown()

    # No aporta funcionalidad al controlador, sólo ayuda a graficar mejor
    def call_mark_position_service(self):
        if self.mark_position_client.wait_for_service(timeout_sec=1.0):
            request = Empty.Request()
            self.mark_position_client.call_async(request)
        else:
            self.node.get_logger().warn('Servicio /mark_position no disponible')

def main(args=None):
    rclpy.init(args=args)
    controller = WheelController()

    """
    Movimientos para el gráfico de la Parte 3)
    """
    # controller.move_forward(duration=4.0, speed=1.0)
    # controller.rotate(duration=4.0, clockwise=True)
    # controller.move_forward(duration=4.0, speed=1.0)
    # controller.rotate(duration=4.0, clockwise=False)
    # controller.move_forward(duration=4.0, speed=-1.0)
    # controller.rotate(duration=4.0, clockwise=True)
    # controller.move_forward(duration=4.0, speed=1.0)
    # controller.move_forward(duration=4.0, speed=1.0)
    # controller.rotate(duration=4.0, clockwise=False)
    # controller.move_forward(duration=4.0, speed=-1.0)
    # controller.rotate(duration=4.0, clockwise=False)
    # controller.move_forward(duration=4.0, speed=-1.0)

    """
    Movimientos para la Parte 4) (ejecutar script 10 veces o agregar ciclo for)
    """
    controller.inverse_kinematics(x=0.0, y=3.75)
    controller.inverse_kinematics(x=0.0, y=2.9)
    controller.inverse_kinematics(x=0.0, y=3.75)
    controller.inverse_kinematics(x=0.0, y=2.9)

    try:
        rclpy.spin(controller)
    except KeyboardInterrupt:
        pass
    finally:
        controller.clean_shutdown()
        rclpy.shutdown()
```

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

5.2. odom_node.py

```
#!/usr/bin/env python3

import math
import rclpy
from rclpy.node import Node
from tf2_ros import TransformBroadcaster
from geometry_msgs.msg import TransformStamped
from sensor_msgs.msg import JointState
from nav_msgs.msg import Odometry


class OdomPublisher(Node):

    def __init__(self):
        super().__init__('example_controller')
        # Subscriber a las velocidades de las ruedas
        self.joint_state_sub = self.create_subscription(
            JointState,
            '/joint_states',
            self.joint_state_callback,
            10)
        # TF
        self.tf_broadcaster = TransformBroadcaster(self)
        self.timer = self.create_timer(0.05, self.update_odometry) # Timer para la TF
        # Odom
        self.odom_pub = self.create_publisher(Odometry, '/odom', 10)

        # Parámetros
        self.r = 0.05 # Radio de las ruedas
        self.l = 0.32 # Longitud del eje de las ruedas

        # Estado Inicial
        self.x = 0.0
        self.y = 0.0
        self.theta = 0.0
        self.last_time = self.get_clock().now()

        # Velocidades de las ruedas
        self.left_vel = 0.0
        self.right_vel = 0.0

    def joint_state_callback(self, msg):
```

```
        left_idx = msg.name.index('left_wheel_joint') # Obtenido al hacer echo al tópico
        ↪  /joint_states
        right_idx = msg.name.index('right_wheel_joint')
        self.left_wheel_vel = msg.velocity[left_idx]
        self.right_wheel_vel = msg.velocity[right_idx]

    def update_odometry(self):
        current_time = self.get_clock().now()
        dt = (current_time - self.last_time).nanoseconds / 1e9
        self.last_time = current_time

        if dt <= 0:
            return

        # Cinemática directa
        v_left = self.left_wheel_vel * self.r
        v_right = self.right_wheel_vel * self.r

        # Vel. lineal y angular
        v = (v_right + v_left) / 2.0
        omega = (v_right - v_left) / self.l

        # Actualizar la posición y orientación
        delta_x = v * math.cos(self.theta) * dt
        delta_y = v * math.sin(self.theta) * dt
        delta_theta = omega * dt

        self.x += delta_x
        self.y += delta_y
        self.theta += delta_theta

        # Normalizar el ángulo entre -pi y pi
        self.theta = math.atan2(math.sin(self.theta), math.cos(self.theta))

        # Publicar la transformación odom->base_link
        self.publish_odom_tf()

        # Opcional: publicar mensaje de odometría
        self.publish_odom_msg(current_time)

    def publish_odom_tf(self):
        t = TransformStamped()

        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'odom'
        t.child_frame_id = 'base_link'
```

```
        t.transform.translation.x = self.x
        t.transform.translation.y = self.y
        t.transform.translation.z = 0.0

        q = self.quaternion_from_euler(0, 0, self.theta)
        t.transform.rotation.x = q[0]
        t.transform.rotation.y = q[1]
        t.transform.rotation.z = q[2]
        t.transform.rotation.w = q[3]

        self.tf_broadcaster.sendTransform(t)

    def publish_odom_msg(self, current_time):
        odom = Odometry()
        odom.header.stamp = current_time.to_msg()
        odom.header.frame_id = 'odom'
        odom.child_frame_id = 'base_link'

        odom.pose.pose.position.x = self.x
        odom.pose.pose.position.y = self.y
        odom.pose.pose.position.z = 0.0

        q = self.quaternion_from_euler(0, 0, self.theta)
        odom.pose.pose.orientation.x = q[0]
        odom.pose.pose.orientation.y = q[1]
        odom.pose.pose.orientation.z = q[2]
        odom.pose.pose.orientation.w = q[3]

        self.odom_pub.publish(odom)

    def quaternion_from_euler(self, roll, pitch, yaw):
        cy = math.cos(yaw * 0.5)
        sy = math.sin(yaw * 0.5)
        cp = math.cos(pitch * 0.5)
        sp = math.sin(pitch * 0.5)
        cr = math.cos(roll * 0.5)
        sr = math.sin(roll * 0.5)

        q = [
            sr * cp * cy - cr * sp * sy,
            cr * sp * cy + sr * cp * sy,
            cr * cp * sy - sr * sp * cy,
            cr * cp * cy + sr * sp * sy
        ]

        return q
```

```
def main(args=None):
    rclpy.init(args=args)
    node = OdomPublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

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

5.3. odom_plotter.py

```
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
from std_srvs.srv import Empty
import matplotlib.pyplot as plt
import numpy as np

class OdomPlotter(Node):
    def __init__(self):
        super().__init__('odom_plotter')

        # Configuración del gráfico
        plt.ion()  # Modo interactivo
        self.fig, self.ax = plt.subplots()
        self.line, = self.ax.plot([], [], 'b-', label='Trayectoria Odom')
        self.markers = self.ax.plot([], [], 'ro', label='Marcadores Odom')[0]
        self.ax.set_xlabel('X (m)')
        self.ax.set_ylabel('Y (m)')
        self.ax.set_title('Trayectoria del Robot')
        self.ax.grid(True)
        self.ax.legend()

        # Datos
        self.x_data = []
        self.y_data = []
        self.marker_x = []
        self.marker_y = []

        # Suscriptor
```

```
        self.odom_sub = self.create_subscription(
            Odometry,
            '/odom',
            self.odom_callback,
            10)

        # Servicio
        self.srv = self.create_service(
            Empty,
            'mark_position',
            self.mark_position_callback)

        # Timer para actualizar el gráfico
        self.timer = self.create_timer(0.05, self.update_plot)

        self.get_logger().info('Odom Plotter listo. Use el servicio /mark_position para marcar
        ↪  la posición actual.')

    def odom_callback(self, msg):
        self.x_data.append(round(msg.pose.pose.position.x, 2))
        self.y_data.append(round(msg.pose.pose.position.y, 2))
        self.current_pose = msg.pose.pose.position  # Guardar la posición actual

    def mark_position_callback(self, request, response):
        if hasattr(self, 'current_pose'):
            self.marker_x.append(round(self.current_pose.x, 2))
            self.marker_y.append(round(self.current_pose.y, 2))
            self.get_logger().info(f'Marcador añadido en: ({self.current_pose.x:.2f},
            ↪  {self.current_pose.y:.2f})')
        else:
            self.get_logger().warn('No hay datos de odometría disponibles aún')
        return response

    def update_plot(self):
        if len(self.x_data) > 0:
            self.line.set_data(self.x_data, self.y_data)

            # Actualizar marcadores si hay alguno
            if len(self.marker_x) > 0:
                self.markers.set_data(self.marker_x, self.marker_y)

            self.ax.relim()
            self.ax.autoscale_view()
            self.fig.canvas.draw()
            self.fig.canvas.flush_events()

def main(args=None):
```

```
    rclpy.init(args=args)
    node = OdomPlotter()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.destroy_node()
        rclpy.shutdown()

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