Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Puebla

Implementación de robótica inteligente (Gpo 501)

Reporte: Reto semanal 2 Manchester Robotics

Alumno

José Diego Tomé Guardado A01733345
Pamela Hernández Montero A01736368
Victor Manuel Vázquez Morales A01736352
Fernando Estrada Silva A01736094

Fecha de entrega

Martes 23 de Abril de 2024

RESUMEN

Se desarrolló e implementó un código usando ROS para controlar la respuesta del Puzzebot, para ello se estableció la capacidad de definir una velocidad y un tiempo, cumpliendo con una trayectoria precodificada en ROS, específicamente una navegación multipunto con control de lazo abierto. Como parte de la documentación, se incluyó un video demostrativo de la ejecución en tiempo real.

OBJETIVOS

El objetivo principal de este reto es poder implementar un controlador para generar el trazo de trayectorias teniendo en consideración el uso del lazo abierto que en este caso debe ser robusto y poder implementar estrategias para lograrlo donde el usuario pueda seleccionar la velocidad o el tiempo para finalizar el recorrido. Nuestros objetivos particulares son los siguientes:

1. Investigar e implementar los mensajes Twist para poder mandar los valores de las velocidades al Puzzlebot.
2. Entender cómo funciona la cinemática de un robot diferencial y poder tener el uso de un controlador de lazo abierto para el robot móvil donde tengamos un controlador auto-sintonizado.
3. Experimentar con el controlador que debe estimar las velocidades, la aceleración o el tiempo requerido.
4. Lograr qué en el segundo recorrido el nodo pueda estimar las fuerzas lineales y rotacionales, informar al usuario si el punto es alcanzable según
   al comportamiento dinámico del robot móvil y a los parámetros que fueron ingresados por el usuario ($v1$, $\omega 1$)/t.

INTRODUCCIÓN

La navegación en el contexto de los robots móviles [1] se refiere al proceso mediante el cual un robot se desplaza de manera autónoma en un entorno, evitando obstáculos y alcanzando destinos específicos de manera eficiente y segura. Este proceso implica la planificación de trayectorias y el control del movimiento del robot para lograr sus objetivos.

La navegación multipunto (Enrutamiento multipunto) [2] es una extensión de este concepto, donde el robot no solo se mueve desde un punto inicial a un punto final, sino que también puede visitar múltiples puntos intermedios en su camino. En ROS el control de la navegación en robots móviles se realiza comúnmente utilizando mensajes Twist [3], parte del paquete geometry_msgs. Estos mensajes expresan la velocidad en el espacio libre, dividiéndola en sus componentes lineal y angular a través de los vectores de tipo Vector3. La utilización de estos

mensajes permite especificar con precisión el movimiento deseado del robot en términos de velocidad lineal y angular, fundamental para su desplazamiento efectivo y seguro en entornos dinámicos.

En este caso, es crucial comprender la cinemática de nuestro robot diferencial, donde el movimiento se logra mediante la diferencia de velocidades entre las dos ruedas montadas en un mismo eje. Esta configuración proporciona una amplia maniobrabilidad y versatilidad en la navegación. Sin embargo, el control de la navegación presenta desafíos, especialmente al adoptar el enfoque de lazo abierto [4], comúnmente utilizado. Este método implica enviar comandos de velocidad al robot sin considerar su estado actual, lo que puede resultar en desviaciones significativas en la trayectoria debido a la falta de retroalimentación sobre la posición y el estado del robot.

Para el cálculo de la distancia y el ángulo recorrido, es fundamental tener un conocimiento preciso de la posición y orientación del robot en todo momento. Esto se logra mediante la integración sistemas de odometría, que proporcionan información sobre la distancia recorrida por cada rueda y los ángulos de giro. Con esta información de posición, podemos calcular la distancia total recorrida por el robot utilizando métodos de integración numérica, como la suma de las distancias recorridas en cada intervalo de tiempo. Asimismo, el ángulo recorrido puede determinarse mediante la diferencia entre las orientaciones inicial y final del robot.

SOLUCIÓN DEL PROBLEMA

Para dar solución al problema previamente planteado y para recorrer las trayectorias previamente descritas, haremos uso del nodo *odometry* creado en la actividad previa, ya que gracias a este nodo podremos aproximar la distancia recorrida y los ángulos de giro. Dentro de una nueva carpeta *Week2* crearemos un nuevo paquete *paths*, en donde crearemos una copia del nodo *odometry*. A continuación, describiremos a detalle cómo implementamos las dos trayectorias solicitadas por la actividad:

Primera trayectoria: Cuadrado

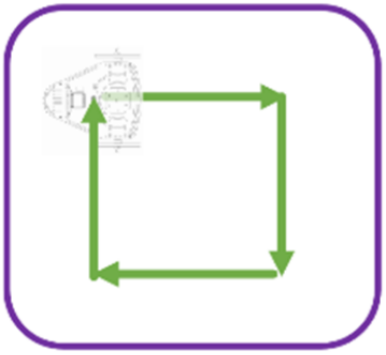

Figura 1. Diagrama de la primera trayectoria.

Comenzaremos por crear nuestro controlador de lazo abierto para trazar la trayectoria de un cuadrado como se observa en la *figura 1*, para el cuál tendremos en cuenta las siguientes consideraciones:

- ➔ Cada lado del cuadrado tiene una distancia de 1 m
- ➔ El usuario tendrá control parcial sobre el recorrido de la trayectoria, pudiendo configurar ya sea la velocidad angular y lineal del robot o configurando el tiempo en el que esperaría que el recorrido sea finalizado.

Para este caso decidimos que el usuario pudiera ajustar las velocidades del robot.

Ahora bien, existían varias formas o enfoques en las que se podía implementar una solución para esta trayectoria. En nuestro caso, después de realizar distintas pruebas y analizar los distintos enfoques decidimos aprovechar el nodo de odometría previamente creado para de esta forma tener noción de la distancia recorrida por nuestro robot.

Implementación en ROS2

Comenzamos por importar las librerías correspondientes para el manejo de este nodo, resaltando especialmente el uso de un geometry_msgs de tipo Twist, el cual nos serviría para mover el robot a través del tópico */cmd _vel.*

Para trazar esta trayectoria creamos un nodo llamado *square_generator* y, además, declaramos los parámetros asociados con la velocidad que servirán para que el usuario pueda configurar parcialmente el recorrido realizado por el robot:

```
class My_publisher(Node):
    def __init__(self):
        super().__init__('square_generator')

        self.declare_parameters(
            namespace='',
            parameters=[
                ('angular_speed', rclpy.Parameter.Type.DOUBLE),
                ('linear_speed', rclpy.Parameter.Type.DOUBLE)
            ]
        )
```

De igual forma, nos suscribimos al nodo *path_selection*, por el cuál indicaremos si deseamos trazar la primer o segunda trayectoria. Además, crearemos otra subscripción al nodo

/Position, generado por el nodo de odometría, el cuál nos servirá para saber la distancia recorrida por nuestro robot.

```
        self.left_speed_subscriber = self.create_subscription(String, 'path_selection', self.path_initialization, rclpy.qos.qos_profile_sensor_data)
        self.positon_subscriber = self.create_subscription(Pose2D, '/Position', self.set_position, rclpy.qos.qos_profile_sensor_data)
```

Posteriormente, crearemos publishers, uno para mandar los datos de la velocidad deseada y otro para resetear los datos del nodo de odometría (un pequeño truco que nos ayudará más adelante a trazar la trayectoria):

```
self.speed_publisher = self.create_publisher(Twist, '/cmd_vel',10)
self.reset_odometry = self.create_publisher(Bool, '/reset_data',10)
```

Ahora bien, crearemos una variable propia del nodo de tipo Twist sobre la cual estaremos modificando la velocidad lineal y angular:

```
 self.speed_configuration = Twist()

 self.speed_configuration.angular.z = 0.0
 self.speed_configuration.linear.x = 0.0
```

Por último, creamos las variables relacionadas con la trayectoria y, además, imprimimos un mensaje indicando la inicialización del nodo:

```
 self.l_square = 1
 self.theta = 90*np.pi/180

 self.v = 0.0
 self.w = 0.0

 self.position = Pose2D()

 self.move = False
 self.sides_recorred = 0
 self.get_logger().info('Square generator node succesfully initialized !')
```

De igual forma, dentro de este nodo tenemos una función asociada a la subscripción al tópico *path_selection*, el cual recibe un dato de tipo String. Dentro de esta función, comparamos que

el dato recibido sea ‘Square’ y, de ser el caso, indicamos que el robot comience la trayectoria al asignar las velocidades y al modificar la variable move de False a True:

```
def path_initialization(self,msg):
        if msg.data == 'Square':
        self.v                                                                  =
self.get_parameter('linear_speed').get_parameter_value().double_value
        self.w                                                                  =
self.get_parameter('angular_speed').get_parameter_value().double_value
        self.move = True
```

Por último, tenemos la función callback asociada al tópico *Position*, Esta función, en resumen, se ejecuta a una alta frecuencia (cada que recibe un dato del nodo de odometry) y realiza el trazado de la trayectoria únicamente cuando la variable move esta en True.

En caso de que la variable move indique el movimiento, moveremos al robot en línea recta a la velocidad indicada por el usuario durante un metro y después lo rotaremos en 90 grados, comprobando el recorrido apoyándonos del tópico de *Position.* Ahora bien, para simplificar el ćodigo, decidimos modificar el nodo de odometría de tal forma que pudiéramos resetear los datos de tal forma que únicamente nos preocupemos pro un recorrido de 1m en x y una rotación de 90 grados, repitiendo este patrón por 4 veces (los lados del cuadrado).

```
def set_position(self,msg):
        #self.get_logger().info("Recibio dato ")
        self.position = msg


        if self.move:
        #Iniciamos con la velocidad lineal
        if(self.position.x<1.0):
                self.get_logger().info('Linea recta')
                self.speed_configuration.angular.z =  0.0
                self.speed_configuration.linear.x = self.v
                self.speed_publisher.publish(self.speed_configuration)
        #Rotamos el bot
        elif(self.position.theta<90.0):
                self.get_logger().info('Rotar')
                self.speed_configuration.angular.z = self.w
                self.speed_configuration.linear.x = 0.0
                self.speed_publisher.publish(self.speed_configuration)
```

```
#Para facilitar el programa, reseteamos los datos del nodo odometria
else:
        self.get_logger().info('Reseting data...')
        reset = Bool()
        reset.data = True
        self.reset_odometry.publish(reset)
        self.sides_recorred +=1
        if(self.sides_recorred>3):
        self.sides_recorred = 0
        self.move = False
        self.get_logger().info('STOP')
        self.speed_configuration.angular.z =  0.0
        self.speed_configuration.linear.x =  0.0
        self.speed_publisher.publish(self.speed_configuration)
```

Archivo YAML: square_path.yaml

Tal y como se pudo observar a lo largo de la explicación del código, para esta trayectoria se utilizó un archivo config .yaml, el cuál contiene los datos de las velocidades angulares y lineales para esta trayectoria:

```
square_generator:
  ros__parameters:
        angular_speed: 0.2
        linear_speed: 0.2
```

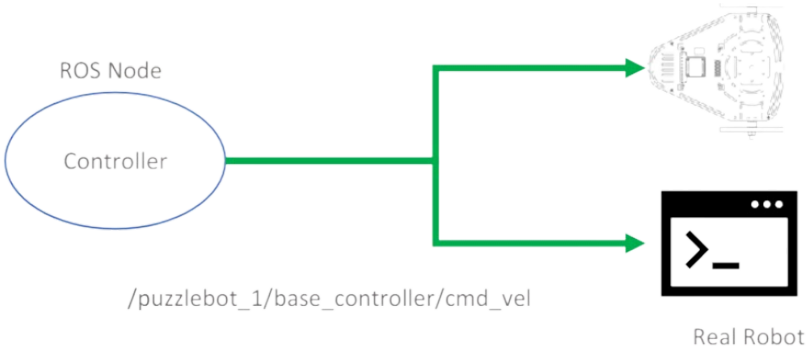

Figura 2. Primer Diagrama de nodos y tópicos

Segunda trayectoria

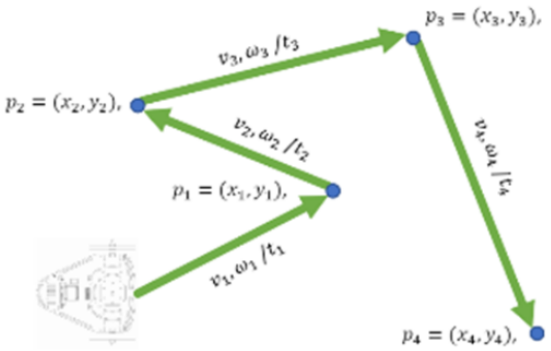

Figura 3. Diagrama de la segunda trayectoria.

Para la segunda trayectoria necesitaremos de poder definir un nodo en el cual pueda generar diferentes trayectorias dependiendo de puntos *pk* = (*xk*, *yk*), velocidades o tiempo ($v1$, $\omega 1$)/*t* dependiendo del usuario para que pueda seguir una trayectoria como la de la *figura 3* como ejemplo . Para cada punto, el nodo necesitará estimar la velocidad lineal y angular en caso de que el usuario le de el tiempo o estimar el tiempo en caso de que las velocidades sean las que haya dado, finalmente el nodo debe de indicar al usuario si es que el punto es alcanzable de acuerdo a cómo se comporta el movimiento del robot y los parámetros que el usuario le ponga ($v1$, $\omega 1$)/*t*.

Implementación en ROS2

Para esta segunda trayectoria se ha implementado un nodo custom path, el cual permite trazar cualquier tipo de trayectoria para el robot. Dentro de un archivo de parámetros, se enlistan los valores de velocidad lineal y angular a partir de los cuales se construye la nueva trayectoria.

Dada la longitud de la trayectoria es como se determina el número de paŕametros que se utilizan, asignándose también un tiempo determinado para ejecutar a la misma.

```
class My_publisher(Node):
  def __init__(self):
        super().__init__('custom_path')
        #Parametros del nodo
        self.declare_parameters(
        namespace='',
        parameters=[
                ('v1', rclpy.Parameter.Type.DOUBLE), ('w1',
rclpy.Parameter.Type.DOUBLE),
                ('v2', rclpy.Parameter.Type.DOUBLE), ('w2',
rclpy.Parameter.Type.DOUBLE),
```

```
                ('v3', rclpy.Parameter.Type.DOUBLE), ('w3',
rclpy.Parameter.Type.DOUBLE),
                ('v4', rclpy.Parameter.Type.DOUBLE), ('w4',
rclpy.Parameter.Type.DOUBLE),
                ('t', rclpy.Parameter.Type.DOUBLE)
        ]
        )
```

A continuación se inician las suscripciones y publicaciones en distintos tópicos del paquete. La primera suscripción se encarga de seleccionar el tipo de trayectoria que se desea elaborar, dado que recibe el nombre de esta, se espera un tipo de dato string. Por otro lado, para poder mover el robot como tal, se realiza una publicación sobre el tópico cmd_vel para poder ingresar una velocidad, utilizando el tipo de dato Twist.

Como se ha mencionado en entregas anteriores, el robot solo cuenta con movimiento a lo largo del eje x y rotación en z, y para evitar cualquier tipo de datos erróneos al momento de medir, se inicializan estas variables en 0.

Una vez que se crea correctamente el nodo, se despliega un mensaje indicando la inicialización.

```
        #Nos subscribimos al nodo
        self.left_speed_subscriber = self.create_subscription(String,
'path_selection', self.path_initialization, rclpy.qos.qos_profile_sensor_data)
        self.speed_publisher = self.create_publisher(Twist, '/cmd_vel',10)
        self.speed_configuration = Twist()
        #self.speed_configuration.angular.x = 0.0
        #self.speed_configuration.angular.y = 0.0
        self.speed_configuration.angular.z = 0.0
        self.speed_configuration.linear.x = 0.0
        #self.speed_configuration.linear.y = 0.0
        #self.speed_configuration.linear.z = 0.0s

        self.get_logger().info('Custom_path node succesfully initialized !')
```

Seguido de esto se describe un método encargado de inicializar la trayectoria deseada. En caso de que el mensaje recibido sea el string ‘Custom Path’, el robot comenzará la trayectoria determinada.

En primera instancia, el tiempo total de la trayectoria se divide entre 8 secciones iguales para que cada movimiento tenga la misma duración. Las variables de *v_string* y *w_string* son las

cadenas para acceder a los parámetros de velocidad (v1, w1, v2, w2, v3, w3, v4, w4) definidos en los parámetros del nodo.

Un mensaje se imprime dentro de la terminal para indicar los instantes en que el robot se mueve en línea recta o hace un giro. Ambos resultados se publican en el tópico cmd_vel.

Una vez que la trayectoria ha sido finalizada, se publica directamente en el tópico velocidad angular y lineal de 0 para hacer que el robot se detenga. También se imprime un mensaje de STOP.

```
  def path_initialization(self,msg):
        self.get_logger().info('Mensaje recibido')
        #Comprobamos la selección
        if msg.data == 'Custom path':
        #Dividimos el tiempo en partes igaules
        t = self.get_parameter('t').get_parameter_value().double_value
        t = t/8
        for i in range(4):
                print(i)
                #Seleccionamos la velocidad correspondiente
                v_string = f'v{i+1}'
                w_string = f'w{i+1}'

                #Movemos el bot
                self.get_logger().info("Recta")
                self.speed_configuration.linear.x = 0.0
                self.speed_configuration.angular.z =
self.get_parameter(w_string).get_parameter_value().double_value
                self.speed_publisher.publish(self.speed_configuration)
                time.sleep(t)
                self.get_logger().info("Rotar")
                self.speed_configuration.linear.x =
self.get_parameter(v_string).get_parameter_value().double_value
                self.speed_configuration.angular.z = 0.0
                self.speed_publisher.publish(self.speed_configuration)
                time.sleep(t)


        self.get_logger().info('STOP')
        self.speed_configuration.linear.x = 0.0
```

```
        self.speed_configuration.angular.z = 0.0
        self.speed_publisher.publish(self.speed_configuration)
```

Archivo YAML: custom_path.yaml

Como se ha mencionado, este formato tiene el propósito de almacenar los parámetros con los que funcionará el programa. En este caso se almacenan los valores de las velocidades lineales y angulares para trazar la trayectoria deseada. En este caso se mantiene una velocidad lineal constante para asegurar el funcionamiento correcto del robot, mientras que se varían velocidades angulares (en cuanto a magnitud y sentido) con el fin de trazar la trayectoria de zigzag.

```
custom_path:
 ros__parameters:
   v1: 0.2
   w1: 0.15
   v2: 0.4
   w2: -0.2
   v3: 0.4
   w3: 0.15
   v4: 0.4
   w4: -0.15
   t: 10.0
```

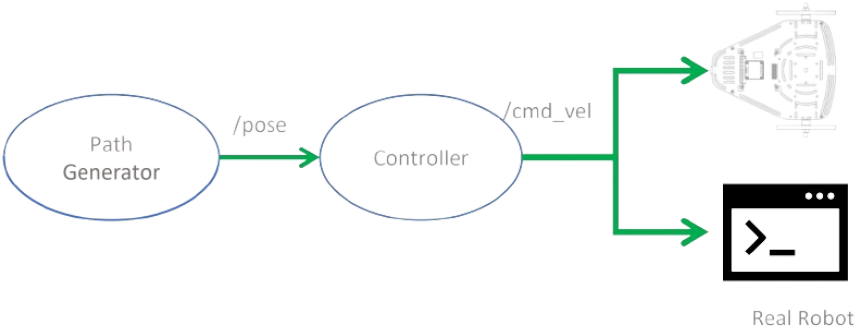

Figura 4. Diagrama de nodos y tópicos

La inicialización de parámetros de los nodos se realizó usando un launch file

RESULTADOS

Video de funcionamiento:
https://drive.google.com/file/d/15j6DQHOB-aTWpt2Ov9dHmatGmqAQzUPl/view?usp=sharing

- **Primera trayectoria: Cuadrado**

Para la primera trayectoria podemos observar en el video que estamos correctamente comenzando a formar el cuadrado, haciendo el recorrido lado por lado hasta llegar al final donde en cada parte se hace el giro de 90 grados del puzzlebot para poder avanzar a la siguiente esquina. Tenemos una velocidad tanto lineal como angular de 0.2 lo cual nos ayuda a poder visualizar de mejor manera el comportamiento del puzzlebot para cada parte del cuadrado. Con el uso del yaml podemos nosotros poder definir las velocidades tanto angulares como lineales para formar la trayectorias:

```
square_generator:
ros__parameters:
    angular_speed: 0.2
    linear_speed: 0.2
```

Teniendo en cuenta que en el nodo de square_generator tendremos declarado la longitud de cada lado y la al ángulo para dar cada vuelta:

```
#Declaramos las variables del robot
        self.l_square = 1
        self.theta = 90*np.pi/180
```

- **Segunda trayectoria: Control de velocidades**

Para la segunda trayectoria podemos observar que tuvimos un comportamiento de forma en zigzag debido a qué nosotros en los parametros de yaml propusimos valores para las velocidades lineales y angulares y un tiempo para trazarlo, de esta forma pudimos tener ese comportamiento que se observa en el video del funcionamiento. Los parámetros definidos para la primera prueba fueron los siguientes:

```
custom_path:
 ros__parameters:
   v1: 0.2
   w1: 0.15
   v2: 0.2
   w2: -0.2
```

```
  v3: 0.2
  w3: 0.15
  v4: 0.2
  w4: -0.15
  t: 10.0
```

En este caso para este nodo tendremos las variables de *v_string* y *w_string* van a ser las cadenas para acceder a los parámetros de velocidad (v1, w1, v2, w2, v3, w3, v4, w4) definidos en los parámetros del nodo.

```
#Parámetros del nodo
        self.declare_parameters(
            namespace='',
            parameters=[
                ('v1', rclpy.Parameter.Type.DOUBLE), ('w1',
        rclpy.Parameter.Type.DOUBLE),
                ('v2', rclpy.Parameter.Type.DOUBLE), ('w2',
        rclpy.Parameter.Type.DOUBLE),
                ('v3', rclpy.Parameter.Type.DOUBLE), ('w3',
        rclpy.Parameter.Type.DOUBLE),
                ('v4', rclpy.Parameter.Type.DOUBLE), ('w4',
        rclpy.Parameter.Type.DOUBLE),
                ('t', rclpy.Parameter.Type.DOUBLE)
            ]
         )
```

Podemos ver que el tiempo total de la trayectoria se divide entre 8 secciones iguales para que cada movimiento tenga la misma duración.

```
#Dividimos el tiempo en partes iguales
t = self.get_parameter('t').get_parameter_value().double_value
t = t/8
```

Al final mostramos una última prueba para probar las velocidades lineales en esta trayectoria por lo que dejamos las velocidades angulares con el mismo valor para que girará como la prueba anterior solo que aumentamos las velocidades lineales a partir de la v2 para que el puzzlebot hiciera el recorrido más rápido. De esta manera lo observamos en el video donde se ve que cuando el robot hace el recorrido más rápido hasta detenerse, los parámetros definidos en el yaml quedaron de la siguiente forma donde solo tuvimos un aumento de 0.2 más a partir de la velocidad declarada como v2:

```
custom_path:
```

```
ros__parameters:
  v1: 0.2
  w1: 0.15
  v2: 0.4
  w2: -0.2
  v3: 0.4
  w3: 0.15
  v4: 0.4
  w4: -0.15
  t: 10.0
```

CONCLUSIONES

A lo largo del reto pudimos experimentar de forma correcta con el uso de un controlador de lazo abierto como controlar de mejor manera las trayectorias del Puzzlebot a partir de las velocidades lineales y angulares. Es importante poder decir que los objetivos qué planteamos se pudieron lograr en gran medida ya qué tuvimos una buena implementación del controlador para generar trayectos con el enfoque del lazo abierto permitiendo que el usuario pudiera definir las velocidades en nuestro caso y cumpliendo así con la funcionalidad, además de poder utilizarlo para tener un buen comportamiento para las trayectorias sobre todo para la del cuadrado que logrará hacer la figura visualmente correcta.

El controlador de lazo abierto funcionó mediante los comandos en ROS para poder considerar las velocidades, determinandolas para que siguiera una trayectoria predeterminada sin recibir retroalimentación sobre la posición. Teniendo los parámetros a partir de la ejecución de un archivo YAML, también el nodo donde podíamos observar implicaba la división del tiempo total de la trayectoria en secciones iguales para que el Puzzlebot tuviera el mismo tiempo en cada movimiento qué hacía, avanzar o girar.

Aunque se logró una buena implementación del controlador de lazo abierto al no tener una retroalimentación sobre la posición provocaba que en ocasiones se tuvieran algunas desviaciones pequeñas en la trayectoria, lo que podría considerarse como una limitación del enfoque del lazo abierto, por lo que podemos tener una posible mejora el implementar una metodología con un sistema de lazo cerrado debido a que es más precios, ya qué cuenta con la información necesaria para permitir corregir cualquier error en el proceso y además tiene una mejor capacidad para hacer las modificaciones de las desviaciones en la trayectoria en tiempo real. Lo qué permite tener un mejor ajuste y una mejor precisión del controlador, además de poder tener un control más avanzado con mayor robustez para hacer trayectorias más complejas como otras figuras geométricas con muchos lados y giros.

A pesar de algunas complicaciones que tuvimos pudimos diseñar y experimentar con nuestro controlador de lazo abierto logrando avances significativos en la navegación del Puzzlebot

sobre todo proporcionandonos una base sólida para comprender de mejor manera y sentar bases para mejoras y ayuda a la precisión de trayectorias.

REFERENCIAS

1. Fernández Lancha. (2010). Planificación de trayectorias para un robot móvil. Universidad Complutense Madrid Facultad de Informática. https://docta.ucm.es/rest/api/core/bitstreams/67e3b70f-a68a-4f74-b779-2c05658f211b/content
2. Hema lab's. Enrutamiento multipunto. Recuperado 23 de Abril del 2024 de http://help.hema-labs.com/HX-1_Guide/Drive/Routing/multipoint_routing.htm
3. ROS org. Twist Message. Recuperado 23 de Abril del 2024 de https://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Twist.html
4. Mirano Suilera (2023). Sistemas de control de lazo abierto y cerrado https://blog.suileraltamirano.com/contenido06-sistemas-de-control-en-lazo-abierto-y-cerrado/