Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Puebla

Fundamentación de Robótica (Gpo 101)

Cálculo del error de un robot móvil diferencial

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

Fecha de entrega
Jueves 25 de Abril de 2024

RESUMEN

Se ha desarrollado un código en ROS2 para gestionar la odometría de un robot móvil diferencial (Puzzlebot). Inicialmente, se establecen las coordenadas de la posición del robot, lo que permite calcular en tiempo real el error de posición y orientación (*ed* y *eθ*) con respecto a una ubicación deseada.

OBJETIVOS

El objetivo general es desarrollar un nodo en ROS2 que publique el cálculo de los errores *ed* y *eθ* del robot móvil diferencial, permitiendo su seguimiento y control en tiempo real. Dentro de sus objetivos particulares se encuentran los siguientes:

1. Crear un nodo en ROS2 que suscriba los datos de la odometría del robot y establezca un objetivo deseado.
2. Calcular y publicar los errores *ed* y *eθ* con respecto al objetivo establecido.
3. Realizar pruebas de funcionamiento para verificar que los valores de *ed* y *eθ* se actualicen correctamente mientras el robot se mueve hacia el objetivo.
4. Garantizar que los ángulos calculados se encuentren dentro del intervalo de un círculo completo (de 0 a 2π radianes o de 0 a 360 grados), manteniendo consistencia en el sistema de referencia utilizado.

INTRODUCCIÓN

Un robot móvil diferencial se caracteriza por su sistema de tracción, el cual consiste en dos ruedas independientes, cada una conectada a su propio motor. La locomoción de este tipo de robot se logra mediante la variación de las velocidades entre estas dos ruedas, situadas en un mismo eje. En este caso, el Puzzlebot tiene una rueda libre en la parte trasera, que gira pasivamente para proporcionar estabilidad al robot durante su desplazamiento.

El diseño del Puzzlebot se adecua al inciso A de la imagen 1; algunas otras configuraciones pueden ser como el conjunto de dos ruedas activas y dos pasivas (b) o cuatro ruedas activas conectadas por un sistema de correas (c).

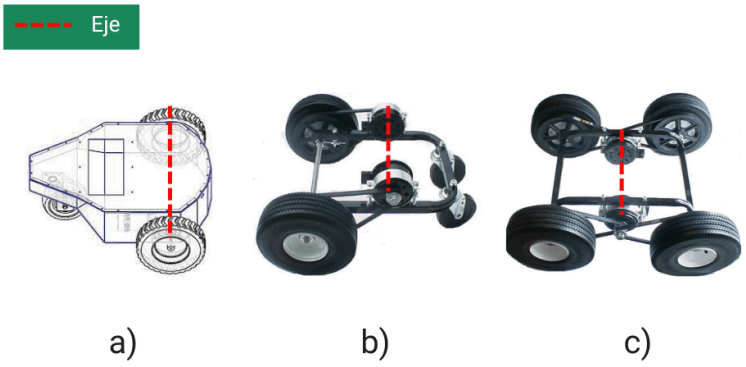

Figura 1. Diseños de un robot diferencial

El funcionamiento de un robot diferencial se basa en la interacción entre las velocidades y direcciones de sus ruedas. Sus configuraciones se pueden observar en la figura 2, cuando ambas ruedas giran en la misma dirección y velocidad, el robot avanza en línea recta hacia adelante. Al cambiar el sentido de giro manteniendo la velocidad, el vehículo se desplaza hacia atrás. Si las ruedas giran en la misma dirección pero con diferentes velocidades, el robot se alejará del motor más rápido, lo que causa un movimiento lateral. Por último, si las ruedas giran a la misma velocidad pero en direcciones opuestas, el robot girará en su propio eje, ya sea en sentido horario o antihorario. Este principio de funcionamiento de las ruedas en un robot diferencial es crucial para la odometría, ya que permite calcular la posición y orientación del robot mediante el seguimiento de los movimientos de las ruedas.

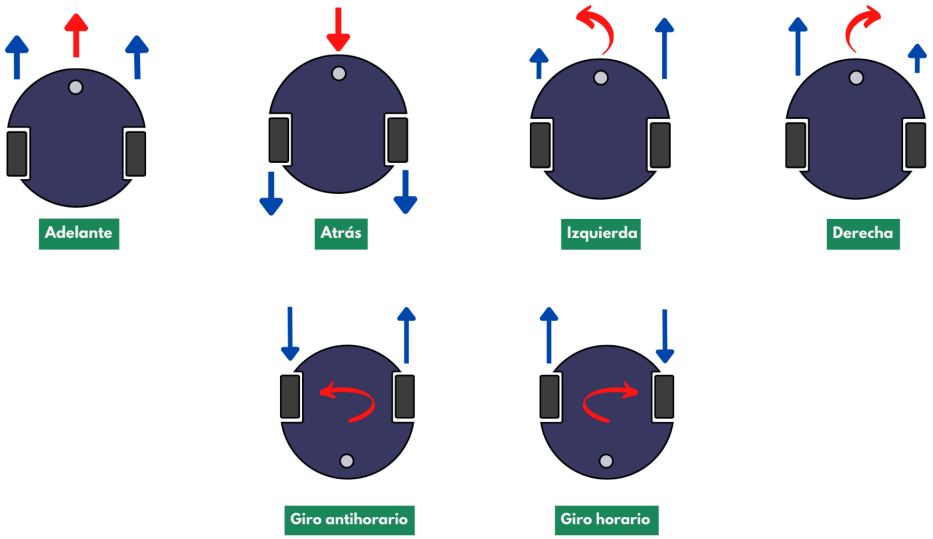

Figura 2. Configuraciones de un robot diferencial.

La odometría desempeña un papel fundamental en la robótica móvil al proporcionar datos precisos sobre la posición y orientación de un robot en su entorno [2]. En el contexto específico de un robot móvil diferencial, la odometría se refiere a la técnica utilizada para calcular los cambios en la posición de un robot móvil basándose en el seguimiento y la medición de la rotación de sus ruedas. Este método permite al robot determinar su desplazamiento relativo en relación con un punto de referencia inicial, en donde, para determinar la localización del móvil en el plano cartesiano, se utiliza su modelo cinemático diferencial directo que relaciona las velocidades del punto de interés con las velocidades de los actuadores considerando al vehículo como una masa puntual.

Por otro lado, el cálculo del error en un robot móvil diferencial es un aspecto crítico para garantizar su navegación eficiente y precisa en su entorno [4]. Este error se refiere a las discrepancias entre la posición y orientación deseada del robot y su posición y orientación real, y puede surgir por diversas razones. Una de las principales fuentes de error es la imprecisión en las medidas de odometría, que pueden deberse a factores como el deslizamiento de las ruedas, irregularidades en la superficie del terreno o desgaste de los

componentes del robot. Estas imprecisiones pueden acumularse a lo largo del tiempo y afectar significativamente la estimación de la posición del robot [3].

Además, la presencia de obstáculos en el entorno puede dificultar la navegación del robot y aumentar el margen de error. Los sensores utilizados para detectar obstáculos, como cámaras o sensores de ultrasonido, también pueden presentar errores en la estimación de la posición si no están correctamente calibrados o si su precisión es limitada. La gestión adecuada de este error es esencial para mejorar la capacidad de navegación y autonomía del robot en diferentes entornos y situaciones.

El error en la odometría, y pueden tener un impacto significativo en la capacidad de navegación del robot, especialmente en entornos complejos o dinámicos. Por ejemplo, un pequeño error en la estimación de la posición puede hacer que el robot evita obstáculos de manera incorrecta o se desvíe de su trayectoria planificada.

Por lo tanto, comprender y gestionar el error en la odometría es fundamental para garantizar la precisión y fiabilidad de la navegación del robot móvil diferencial. Esto puede implicar la implementación de técnicas de corrección de errores, como la fusión de datos de múltiples sensores (como sensores de odometría, sistemas de posicionamiento global, sistemas de visión, etc.) o el uso de algoritmos de estimación del estado del robot que puedan compensar los errores de la odometría y mejorar la precisión de la posición y orientación estimadas.

SOLUCIÓN DEL PROBLEMA

Cómo ya se mencionó, para realizar el trazado de figuras de manera eficiente es importante conocer en todo momento la posición del robot, ya que esto nos permitirá posteriormente realizar el cálculo correspondiente para calcular la distancia entre la posición actual del robot y un punto que se desea alcanzar (error). Dicho esto, para realizar el cálculo del error dentro de ROS creamos los nodos descritos a continuación.

Nodo de Odometría

Tal y cómo hemos venido insistiendo en los últimos entregables, la odometría es la rama encargada de calcular o estimar la posición de un robot en un espacio determinado, lo que resulta fundamental en la navegación autónoma de vehículos y robots.

Existe una gran variedad de algoritmos y métodos para realizar la estimación de la posición un vehículo, pero uno de las más eficientes y simples se basa en la lectura de la velocidad de las llantas del robot, la cual se puede estimar a partir de un encoder.

En la siguiente imagen en la figura 3, donde podemos observar un pequeño diagrama del modelo cinemático de un robot diferencial, en el cuál se puede observar ciertos elementos de importancia que nos servirán para calcular la posición en todo momento del robot:

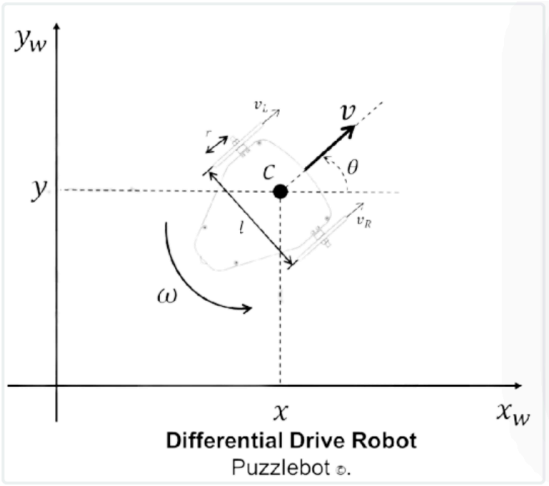

Figura 3. Diagrama del modelo cinemático del Puzzlebot

Por medio de un análisis, es posible construir ciertas fórmulas que podremos plasmar dentro de nuestro código para realizar la implementación de la odometría:

Tal y cómo podemos observar, las fórmulas que se obtienen a partir del análisis son las de velocidad lineal y angular, pero podemos obtener los datos de posición realizando una integral.

$$d = v \cdot dt = r\left(\frac{\omega_r + \omega_l}{2}\right) \cdot dt \quad [1]$$

$$\theta = \omega \cdot dt = r\left(\frac{\omega_r - \omega_l}{l}\right) \cdot dt \quad [2]$$

Ahora que tenemos claro lo que debemos implementar en código, procederemos a realizar la programación en ROS2. Comenzaremos por crear el paquete correspondiente en el que estaremos trabajando:

```
ros2 pkg create --build-type ament_python --license Apache-2.0 closed_loop --node-name odometry --dependencies std_msgs geometry_msgs
```

***Nota:** Nombramos a nuestro paquete como "closed_loop" ya que lo usaremos más adelante para realizar la implementación de trazado de trayectorias con un controlador de lazo cerrado.*

Una vez que hemos creado nuestro paquete y nuestro nodo, procederemos a realizar la programación del nodo.

Funcionamiento del nodo:

1. Importamos las librerías correspondientes para el manejo de nuestro nodo

```
import rclpy
import numpy as np
from rclpy.node import Node
```

```
from geometry_msgs.msg import Pose2D
import rclpy.qos
from std_msgs.msg import Float32, Bool
```

2. Posteriormente, inicializamos el nodo bajo el nombre ‘odometry_node’ y nos suscribimos a los tópico ‘VelocityEncL’ y ‘VelocityEncR’, los cuáles contienen la velocidad de las llantas y nos servirá para calcular la posición más adelante:

```
class My_publisher(Node):
 def __init__(self):
  super().__init__('odometry_node') #Inicializamos el nodo

  #Nos suscribimos a los nodos de las velocidades
  self.left_speed_subscriber = self.create_subscription(Float32,
  'VelocityEncL', self.read_left,rclpy.qos.qos_profile_sensor_data)

  self.right_speed_subscriber = self.create_subscription(Float32,
  'VelocityEncR', self.read_right,rclpy.qos.qos_profile_sensor_data)
```

3. De igual forma, creamos un publisher que se encargará de mandar un dato de tipo Pose2D al tópico ‘/Position’, a una frecuencia de 100 hz (periodo de 0.01 s):

```
  #Creamos el publisher para la posición del robot:
  self.position_publisher = self.create_publisher(Pose2D, 'Position', 10)
  #Declaramos el timer asociado al publisher
  self.timer_period = 0.01 #Declaramos el periodo del publsher como 100 HZ
  self.timer = self.create_timer(self.timer_period, self.position_callback)
```

4. Recordemos que dentro de las fórmulas tenemos ciertas variables propias del robot, como es la distancia entre los ejes *l* y el radio de las ruedas *r:*

```
  #Declaramos las variables del robot
  self.l = 0.19 #Distancia entre ruedas
  self.r = 0.05 #Diametro de las ruedas
```

5. Declaramos otras variables que nos servirán para ir realizando los cálculos respectivos de la odometría:

```
  self.right = Float32() #Variable para velocidad derecha
  self.left = Float32() #Variable para velocidad izquierda

  #Variables para la velocidad
  self.xp = 0.0
  self.yp = 0.0
  self.thetap = 0.0

  #Variables para la posición
  self.x = 0.0
  self.y = 0.0
  self.theta = 0.0

  #Variables del robot
  self.position = Pose2D()
```

6. Agregamos un mensaje que nos informe sobre la inicialización correcta del nodo:

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

7. Si volvemos atrás en la explicación del código de este nodo podremos notar que al suscribirnos a las velocidades asociamos cada suscripción a distintas funciones o callbacks (*read_left* y *read_right*). Dentro de estas funciones, lo único que realizamos es realizar una copia de la información recibida a las variables "locales" de nuestro nodo:

```
 def read_left(self, msg):
  #Copiamos la información a la variable del nodo
  self.left = msg

 def read_right(self, msg):
  #Copiamos la información a la variable del nodo
  self.right = msg
```

7. Procedemos con la explicación de nuestra función *position_callback*. Dentro de esta función, comenzamos calculando la velocidad lineal en *x* y *y,* así como el cálculo de la velocidad angular:

```
 def position_callback(self):

  #Calculamos la velocidad para x, y y theta
  self.xp = self.r*((self.right.data+self.left.data)/2)*np.cos(self.theta)
  self.yp = self.r*((self.right.data+self.left.data)/2)*np.sin(self.theta)
  self.thetap = self.r*(self.right.data - self.left.data)/self.l
```

8. Ahora bien, tal y cómo mencionamos anteriormente, para obtener la posición simplemente debemos realizar la integral de la velocidad lineal y angular. Para lograr esto, debemos ir sumando a nuestras variables de posición la velocidad multiplicada el periodo de muestreo ($d = vt$):

```
  #Integramos para obtener la posición
  self.x += self.xp * self.timer_period
  self.y += self.yp * self.timer_period
  self.theta += self.thetap * self.timer_period
```

9. Una vez que hemos calculado las coordenadas de nuestro robot, procederemos a copiar dichos datos a nuestra variable tipo Pose2D para finalmente publicarla a través del topico */Position*:

```
  #Copiamos la información al dato tipo Pose2D
  self.position._x = self.x
  self.position._y = self.y
  self.position._theta = self.theta*180/np.pi

  #Publicamos la posición
  self.position_publisher.publish(self.position)
```

10. Finalmente, lo que debemos realizar es crear el nodo dentro de nuestro main:

```
#Inicialización y creación del nodo
def main(args=None):
 rclpy.init(args=args)
 m_p = My_publisher()
 rclpy.spin(m_p)
 m_p.destroy_node()8
```

```
 rclpy.shutdown()

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

Nodo de error

Este segundo nodo tendrá la función de calcular el error de la posición obtenido a partir de una trayectoria determinada. En tiempo real, la diferencia entre el valor deseado de la posición con el actual obtenido por el nodo de odometría.

Para dar inicio, dentro de la carpeta de src del paquete *closed_loop*, se ha incluido un nuevo nodo para representar este error. Sin embargo deben tomarse ciertas consideraciones:

1. Dada la robustez del algoritmo, se han implementado varios nodos dentro el mismo paquete para realizar suscripciones y publicaciones simultáneamente.
2. Para poder compilar varios nodos, es necesario modificar nuestro archivo setup.py, el cual se encuentra dentro de la misma carpeta de *src*.

Dentro de estas modificaciones se incluye indicar el nuevo nodo que se crea y el paquete al que pertenece, almacenado en una nueva etiqueta:

```
entry_points={

'console_scripts': [
'control = closed_loop.control:main',
'odometry = closed_loop.odometry:main',
'error = closed_loop.error:main'
],
},
```

Custom message: Error2D

Dentro de nuestros paquetes, consideramos bastante útil el crear nuestro propio mensaje personalizado, el cuál contendría el error de distancia y de theta. Para realizar este mensaje, dentro de nuestra carpeta src creamos un nuevo paquete nombrado *srv_int*:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 srv_int
```

Posteriormente, nos movemos a la carpeta del nuevo paquete y creamos un nueva carpeta *msg* en dónde agregaremos nuestro mensaje personalizado *Error2D*:

```
cd srv_int
mkdir msg
touch msg/Error2D.msg
```

Dentro de nuestro archivo *Error2D* agregaremos los datos que contendrá este mensaje:

```
float64 distance_error
float64 theta_error
```

Para verificar que la creación de nuestro mensaje se haya realizado correctamente, realizaremos la ejecución de los siguientes comandos en terminal:

```
colcon build
source install/setup.bash
ros2 interface show srv_int/msg/Error2D
```

Al ejecutar estos comandos, deberemos obtener en terminal información sobre los datos que tenemos dentro de este archivo:

```
float64 distance_error
float64 theta_error
```

Posterior a esto, para poder hacer uso de este mensaje deberemos agregar las siguientes líneas al archivo CMakeLists.txt

```
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
"msg/Error2D.msg" )
```

Además de lo anterior, debemos agregar las siguientes líneas al archivo package.xml:

```
<buildtool_depend>rosidl_default_runtime</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Por último, para hacer uso de este mensaje en el nodo de error deberemos agregar la dependencia a este paquete dentro del archivo package.xml:

```
<depend>srv_int</depend>
```

Nota: *Es importante poner atención y no confundirse entre los archivos package.xml de los distintos paquetes.*

A continuación, explicaremos el funcionamiento del nodo de error.

Funcionamiento del Nodo

1. Como inicio, se incluyen las librerías que se han estado utilizando con regularidad, además de un nuevo tipo de mensaje proveniente de srv_int.msg que utiliza *Error2D*. En el siguiente apartado se describe con más detalle el funcionamiento de este custom message.

```
from std_msgs.msg import Float32
from srv_int.msg import Error2D
```

2. A continuación inicia la clase *My_Publsiher* la cual contiene el funcionamiento general del programa. El programa nombra como 'error node' a este nodo dentro del constructor. Dado que el error depende de las mediciones realizadas con la odometría, se realiza la suscripción a este tópico y se utilizará el callback get_error para manipular estas nuevas cifras. Posteriormente se crea un publisher para publicar sobre un nuevo tópico 'Error' los resultados obtenidos. Para ello entonces se utilizará un tipo de dado *Error2D*. También se incluye la creación de un timer el cual define el periodo que utilizará el publisher, siendo de 10 hz.

```
class My_publisher(Node):
 def __init__(self):
  super().__init__('error_node') #Inicializamos el nodo
  #Nos subscribimos a los nodos de las velocidades
  self.position_subscriber = self.create_subscription(Pose2D, '/Position',
  self.get_error, rclpy.qos.qos_profile_sensor_data)

  self.error_publisher = self.create_publisher(Error2D, '/Error', 10)
  self.timer_period = 0.01 #Declaramos el periodo del publsher como 10 HZ
  self.timer = self.create_timer(self.timer_period, self.position_callback)
```

3. Se definen variables propias de la clase que se utilizarán para almacenar las lecturas del robot, cada una con su respectivo tipo de mensaje. Una vez que el nodo ha sido inicializado, se imprime un mensaje de confirmación.

```
  self.point = Point()
  self.point._x = 2.0 #Setpoint de prueba
  self.point._y = 2.0 #Setpoint de prueba
  self.error = Error2D()
  self.get_logger().info('Error messaage succesfully initialized !')
```

Dentro de esta sección de código contamos con una variable *point* la cual nos será útil para realizar pruebas sobre nuestro nodo error.

4. Método get_error: En esta función se calcula el error tanto para la magnitud de la posición en x e y, como para el ángulo de rotación en z.
En primer lugar, el error en la posición puede considerarse como la relación de distancia que existe entre 2 puntos, donde:

[3] $$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Siguiendo la sintaxis de python, la resta de variables se da a partir de la creación de un objeto *point*, con un atributo *_x,* lo mismo para y. A este valor se le resta el mensaje obtenido de las mediciones del robot dentro del nodo odometry, siendo *msg._x*. Finalmente en una variable local de la función, se almacena el error.

```
  x_distance = self.point._x - msg._x
```

```
y_distance = self.point._y - msg._y
distance = np.sqrt((x_distance**2)-((y_distance**2)))
```

El mismo procedimiento se realiza para obtener el error para el ángulo, a partir de razones trigonométricas.

$$[4] \quad e_\theta = \theta_T - \theta_R = \text{atan2}(x_T, y_T) - \theta_R$$

```
#se calcula el error del angulo
angle = np.arctan(x_distance/y_distance)
angle_degrees = np.degrees(angle)
error = angle_degrees- msg._theta
```

Finalmente, los errores se almacenan dentro de las variables declaradas en msg y se publican.

```
self.error.distance_error = distance
self.error.theta_error = angle_degrees
self.error_publisher.publish(self.error)
```

RESULTADOS

Video del funcionamiento:
https://drive.google.com/file/d/1IE8rWSuIjuORBP-47-_CbHFw_meSwm_a/view?usp=sharing

Primera prueba de trayectoria con un punto

Para la primera prueba que nosotros propusimos fue tener un punto marcado con un ángulo declarado también para poder observar cómo se iba acercando hacía ese punto para tener un error mínimo cercano a 0. Por lo que debemos de proponer un punto en nuestro *error_node* donde vamos a poder definir las coordenadas del punto qué para este caso se consideró un 1.0 en x y también 1.0 en y, así como también el ángulo que se obtendra a partir del punto que para nuestro caso ya haciendo pruebas lo tendriamos de 45 grados al que debe de tener.

```
self.point = Point()
self.point._x = 1.0
self.point._y = 1.0
self.error = Error2D()
```

Ya teniendo definido nuestro punto, es hora de comenzar con el uso del teleop, primeramente tenemos que correr nuestros nodos de la siguiente manera tanto el de odometria como el del error para posteriormente observar los tópicos en la terminal:

ros2 run closed_loop odometry
ros2 run closed_loop error

Ahora ya hacemos un echo de nuestro tópico de posición de odometria: *ros2 topic echo /Position* para poder conocer constantemente la posición mediante los movimientos que vaya

ejecutando el robot y también el topico para el error: *ros2 topic echo /error* que nos ayuda a observar el error constante a los movimientos del puzzlebot.

En esta primera prueba lo quisimos probar de esta forma manualmente para poder observar de mejor manera y más a detalle cómo poco a poco iba variando el error hasta llegar a nuestro punto propuesto y tener un error cercano a cero. Por lo que en el video se muestra que comenzamos en las coordenadas de origen (0,0) y posteriormente a medida que vamos metiendo los movimientos para acercarnos al punto vemos como el error está constantemente siendo calculado y publicándose, del lado izquierdo se observa el error que se va teniendo en la distancia junto con el error de theta que es el ángulo que igual constantemente va disminuyendo a medida que más nos acercamos al punto.

Finalmente cuando acaba la trayectoria vemos en la terminal del lado derecho vemos que correctamente se tuvo un aproximado al ajuste de nuestra coordenada (1,1) con un aproximado y también de nuestro ángulo llegó a un aproximado de 44.68 grados y del lado izquierdo mostramos que el error llega a un aproximado de 0 para la distancia y para el ángulo por que ya nos encontramos demasiado cercanos al punto, casi exactamente en la posición del punto propuesto.

Segunda prueba trayectoria del cuadrado

Para la segunda prueba consideramos ahora la trayectoria de un cuadrado, de esta forma en que pudiera ya tener la trayectoria definida para hacerlo de forma automática y comprobar también el error que se esté generando. Para trazar la trayectoria del cuadrado, decidimos reutilizar el código del entregable dos, en el cual indicamos al bot trazar una trayectoria cuadrada. Ahora bien, es importante mencionar que realizamos una modificación a los nombres de los nodos para evitar confusiones dentro de ROS.

En nuestro nodo encargado de trazar la trayectoria cuadrada agregamos un publisher a un nuevo tópico que si bien no publicaba un dato de importancia, nos servirá para indicar al nodo de error que debe realizar el cálculo de dicha distancia en relación a otro punto. De igual forma, creamos un vector de puntos para x y y que contiene los vértices del cuadrado.

Ahora bien, nuestro código se encarga de calcular el error para cada uno de los puntos. En nuestro video demostrativo, podemos observar que el error de distancia va disminuyendo a lo largo del movimiento del robot en cada uno de los lados del cuadrado, decrementando poco a poco y volviendo a un valor alto al realizar el cambio de vértice con el cual se calcula el error.

Por su parte, el error relacionado con el ángulo si que es mucho más estable, ya que de momento nuestro código no considera el ángulo en relación al último vértice de referencia en el que está nuestro robot, si no que lo considera a partir del origen. Esto provoca que exista un ruido o error en este valor.

A pesar de todo, consideramos que la implementación del error es totalmente exitosa, pues en términos generales podemos observar que el cálculo es correcto, aunque es necesario realizar algunos ajustes pero esto lo realizaremos más adelante considerando ya el enfoque que tendremos para realizar nuestro controlador de lazo cerrado.

CONCLUSIONES

En conclusión, este trabajo nos permitió alcanzar importantes logros; a través de la interacción directa con el Puzzlebot, pudimos conocer en todo momento las coordenadas de su posición, gracias a la implementación de la odometría y el cálculo de errores en ROS2. Esta experiencia nos brindó una comprensión más profunda de los desafíos y complejidades asociadas con la navegación de robots móviles diferenciales.

En cuanto al cumplimiento de los objetivos, podemos afirmar que se lograron en gran medida. La implementación y prueba de nuestro código demostraron la eficacia del sistema en la gestión de errores y el seguimiento preciso del robot hacia un objetivo deseado. El hecho de que el error se aproximara a cero a medida que el Puzzlebot se acercaba al punto objetivo establecido es un claro indicador del éxito de nuestra metodología.

Sin embargo, es importante reconocer que siempre hay margen para mejoras. En este sentido, podríamos optimizar el algoritmo de control, considerar la presencia de obstáculos dinámicos y realizar una calibración más precisa de los sensores para reducir aún más el error y mejorar la eficiencia del sistema. Asimismo, es fundamental continuar investigando y desarrollando estrategias para mejorar la precisión de la odometría y mitigar los errores, lo que permitirá al robot operar de manera aún más autónoma y eficiente en una variedad de entornos y situaciones.

REFERENCIAS

1. Edisonsasig. (Enero del 2021).¿Qué es un robot móvil diferencial?. Roboticoss. https://roboticoss.com/modelo-cinematico-y-simulacion-con-python-robot-movil-diferencial/
2. Suárez-Gómez, A. D., & Pérez-Holguín, W. J. (2022). Sistema de odometría basado en codiseño H/S para un robot móvil diferencial. In IV Congreso Internacional de Ciencias Básicas e Ingeniería CICI2022 (pp. 1-4).
3. Calderon Garcia, J. E. (2016). Evaluación de los métodos umbmark y triangular path para la estimación y corrección de errores sistemáticos en odometría.https://e-archivo.uc3m.es/error_de_odometria_55sas5f
4. Canencia, C., Nicolay, M., Castillo, P. A., & Geovanny, W. Reconocimiento y seguimiento de plataformas para el aterrizaje automático de un vehículo aéreo no tripulado basado en inteligencia artificial y odometría visual.