Tecnológico de Monterrey

Instituto Tecnológico de Estudios Superiores Monterrey

CAMPUS PUEBLA

Integración de Robótica y Sistemas Inteligentes TE3003B

Mini challenge II

Integrantes

José Jezarel Sánchez Mijares A01735226

Antonio Silva Martínez A01173663

Dana Marian Rivera Oropeza A00830027

Profesor

Rigoberto Cerino Jiménez

Fecha: 19 de abril de 2024

Contenidos

Resumen

En este trabajo, abordamos el desarrollo de nuestra solución para el mini challenge dos, que implicó varios aspectos clave. En primer lugar, desarrollamos sistemas dinámicos para simular el comportamiento del Puzzlebot de manera precisa y realista. Utilizamos archivos URDF (Unified Robot Description Format) para la visualización de nuestras simulaciones en RVIZ, lo que permitió una representación visual detallada y comprensible del Puzzlebot en movimiento.
Detallaremos en este informe los pasos seguidos en el desarrollo de nuestra solución, los desafíos enfrentados, los resultados obtenidos y las lecciones aprendidas durante este proceso. Nuestra solución proporciona una base sólida para futuros desarrollos en el campo de la robótica y la simulación de sistemas dinámicos.

Objetivos

Los objetivos de este Challenge se centran en la generación de una simulación en RVIZ donde un modelo del turtlebot siga una trayectoria definida, con la alimentación de diversos nodos para generar el movimiento del robot. Los objetivos específicos incluyen:

- Simulación en RVIZ
- Comunicación entre los nodos usando los tipos de datos especificados
- Desarrollo del sistema dinámico
- Localización del robot dentro del espacio
- Creación del controlador
- Creación de nodos:
    - Controlador
    - Set point
    - Movimiento
    - Localización
    - Publicador - joint state y robot state
    - Transformación de coordenadas

Introducción

En este trabajo, nos enfocamos en el desarrollo y simulación en RVIZ de un Puzzlebot, centrándonos en un modelo cinemático y la implementación de algoritmos de localización y control. El Puzzlebot es una plataforma robótica modular, diseñada para aplicaciones que requieren maniobrabilidad y precisión. Este desafío se divide en tres partes:

Simulación Cinemática: Desarrollamos una simulación cinemática utilizando el modelo cinemático de un robot no holonómico (Puzzlebot). Esta simulación permite modelar el movimiento del Puzzlebot con precisión y eficiencia, lo que facilita el desarrollo y prueba de algoritmos de control.
Localización por Odometría: Implementamos un nodo de localización que utiliza la odometría para calcular la posición y orientación del Puzzlebot en el espacio. La odometría es fundamental para permitir que el Puzzlebot se mueva de manera autónoma y precisa en su entorno.
Control de Posición: Desarrollamos un nodo de control que permite establecer la posición del Puzzlebot en un punto deseado. Este nodo de control es esencial para la aplicación práctica del Puzzlebot, ya que permite a los usuarios especificar la ubicación exacta a la que desean que el Puzzlebot se mueva.
Detallaremos los pasos seguidos en cada parte del desafío, así como los resultados obtenidos y las conclusiones derivadas de la implementación de los algoritmos. Este trabajo proporciona una base sólida para futuras investigaciones en robótica y control de sistemas autónomos.

Solución del challenge

En la primera parte hemos realizado un nodo en ROS que simula un sistema dinámico, específicamente un robot no holonómico que en este caso es el Puzzlebot. Para ello, hemos utilizado el modelo cinemático del robot, el cual se describe mediante las siguientes ecuaciones diferenciales:

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = \omega \end{cases}$$

Donde representamos las coordenadas del robot en el plano XY siendo θ es el ángulo de orientación del robot respecto al eje Z, mientras que v es la velocidad lineal del robot y ω es la velocidad angular del robot. Además, hemos publicado la velocidad de cada una de las ruedas del robot utilizando el mensaje "Float32". Los temas para las ruedas derecha e izquierda se han denominado "/wr" y "/wl" respectivamente.

Para el cálculo de la velocidad lineal y angular del Puzzlebot, hemos utilizado las siguientes fórmulas:

$$v = \frac{v_R + v_L}{2} = r\frac{\omega_R + \omega_L}{2}$$

$$v = \frac{v_R + v_L}{l} = r\frac{\omega_R + \omega_L}{l}$$

Donde v_R y $v_L$son las velocidades de las ruedas derechas e izquierdas respectivamente, debemos de tomar en cuenta que en este modelado las medidas utilizadas para las llantas es de radio de la rueda de 5 cm y la distancia entre ruedas es de 19 cm.

Dentro del apartado del código, el nodo simulado ha sido incluido en el paquete denominado "puzzlebot_sim". Para la publicación de la posición del robot, se ha utilizado el mensaje "PoseStamped", el cual es publicado en el tema "/pose". Para el control del robot, se ha utilizado el mensaje "Twist", el cual es suscrito en el tema "/cmd_vel". Mientras que la simulación hemos tomando en cuenta las ecuaciones antes mencionadas y se explicará su funcionamiento en pseudocódigo a continuación

```
Inicio de la simulación:
    Inicializar la posición y orientación del robot
    Inicializar las variables de tiempo, velocidad y posición

    Mientras no se interrumpa la ejecución por ROS:
    Obtener el tiempo actual
    Si es la primera iteración:
    Guardar el tiempo actual como tiempo anterior
    Marcar la primera iteración como completa
```

Sino:
Calcular el diferencial de tiempo desde la última iteración
Actualizar el tiempo anterior al tiempo actual

Calcular las velocidades de las ruedas derecha e izquierda del robot:
wr_1 = (2.0 * x_l + wheelbase * z_a) / (2.0 * radius)
wl_1 = (2.0 * x_l - wheelbase * z_a) / (2.0 * radius)

Publicar las velocidades de las ruedas en los respectivos temas:

Publicar wr_1 en el tema "/wr"
Publicar wl_1 en el tema "/wl"

Calcular la velocidad angular del robot:
theta_dot = wrap_to_Pi(radius * ((wr_1 - wl_1) / wheelbase))

Integrar la velocidad angular para obtener la orientación del robot:
theta = theta + theta_dot * dt

Calcular la velocidad lineal del robot en el eje x:
x_dot = radius * ((wr_1 + wl_1) / 2) * cos(theta)

Calcular la velocidad lineal del robot en el eje y:
y_dot = radius * ((wr_1 + wl_1) / 2) * sin(theta)

Integrar las velocidades lineales para obtener la posición del robot:
x = x + x_dot * dt
y = y + y_dot * dt

Publicar la posición del robot en el tema "/pose":
Publicar la posición x, y y la orientación theta del robot

Publicar la posición de las articulaciones del robot en el tema "/joint_states":
Publicar la posición x de la articulación de la rueda1
Publicar la posición x de la articulación de la rueda2

En la segunda etapa del challenge, se llevó a cabo el desarrollo de un archivo URDF para describir el Puzzlebot y se configuró su visualización en RVIZ utilizando los archivos ".stl" proporcionados por MCR2. Asimismo, se creó un nodo de ROS que actúa como un publicador de articulaciones personalizado para el Puzzlebot, se estableció un marco inercial llamado "odom", y se realizó una transformación entre los marcos "odom" y "base_link".
El desarrollo del archivo URDF incluyó la definición de los enlaces y articulaciones necesarios para representar la estructura del Puzzlebot. Se crearon los siguientes enlaces:

- "base_link": Representa la posición 2D del robot.
- "chassis": Representa el chasis del Puzzlebot.
- "wl_link" y "wr_link": Representan las ruedas izquierda y derecha respectivamente.

Para las articulaciones, se establecieron:

- "chassis_joint": Articulación estática que conecta "base_link" con "chassis".
- "rightWheel" y "leftWheel": Articulaciones continuas que conectan "chassis" con "wr_link" y "wl_link" respectivamente.

La visualización en RVIZ se logró utilizando el archivo URDF y los archivos ".stl" proporcionados, lo que permitió representar el Puzzlebot como un robot en 3D.

El nodo de ROS creado como publicador de articulaciones personalizado para el Puzzlebot lee el mensaje de odometría y publica la información de las articulaciones en el robot simulado. Se estableció un marco inercial llamado "odom" para el Puzzlebot, y se creó una transformación entre los marcos "odom" y "base_link".

Para verificar el funcionamiento correcto de la simulación, se realizaron pruebas bajo diferentes condiciones, incluyendo diferentes velocidades. Se definió un tiempo de muestreo adecuado para la simulación para garantizar una representación precisa del comportamiento del Puzzlebot. Además, se utilizó métodos numéricos para resolver las ecuaciones diferenciales del Puzzlebot, lo que permitió simular su movimiento de manera precisa en el entorno de ROS.

```
Inicio del método calculate_odometry():
    Si self.first es Verdadero:
    self.previous_time = obtener el tiempo actual de ROS
    self.first = Falso
    Sino:
    current_time = obtener el tiempo actual de ROS
    dt = (current_time - self.base_time) convertido a segundos
    self.base_time = current_time

    wr = obtener la velocidad de la rueda derecha
    wl = obtener la velocidad de la rueda izquierda

    v = radio * (wr + wl) / 2.0
    w = radio * (wr - wl) / wheelbase

    Actualizar la posición x del robot:
    self.pose.pose.position.x += v * cos(theta) * dt
    Actualizar la posición y del robot:
    self.pose.pose.position.y += v * sin(theta) * dt
    Actualizar el ángulo theta del robot:
    self.theta = wrap_to_Pi(self.theta + w * dt)
    Obtener el cuaternion a partir de los ángulos de Euler (0, 0, theta):
    quaternion = quaternion_from_euler(0, 0, self.theta)
    Actualizar la orientación del robot:
    self.pose.pose.orientation.x = quaternion[0]
    self.pose.pose.orientation.y = quaternion[1]
    self.pose.pose.orientation.z = quaternion[2]
    self.pose.pose.orientation.w = quaternion[3]
```

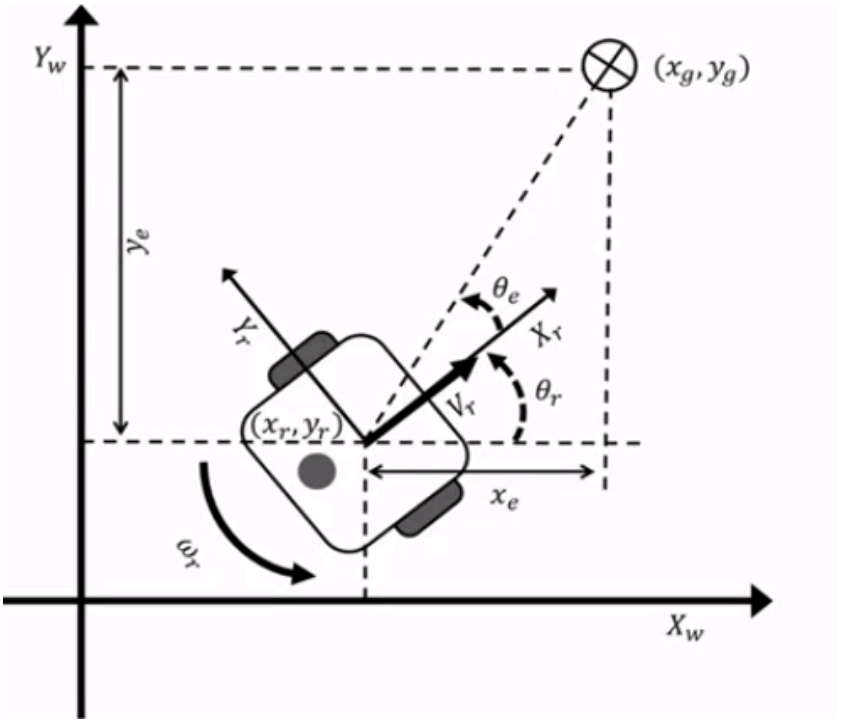
```
        Crear un mensaje Odometry y publicarlo:
        odom_msg = Odometry()
        odom_msg.header.stamp = current_time
        odom_msg.header.frame_id = "odom"
        odom_msg.child_frame_id = "base_link"
        odom_msg.pose.pose = self.pose.pose
        self.odom_pub.publish(odom_msg)

        Publicar la transformación de odometry a base_link:
        self.odom_broadcaster.sendTransform(
        (self.pose.pose.position.x,                    self.pose.pose.position.y,
self.pose.pose.position.z),
        (quaternion[0], quaternion[1], quaternion[2], quaternion[3]),
        current_time,
        "base_link",
        "odom"
        )
```

En la fase final de este proyecto, se implementó un nodo denominado "control" con el objetivo de gestionar el movimiento del robot simulado basado en la información de odometría recibida a través del tópico "/odom". Este nodo se encargó de enviar comandos al robot para dirigirlo a lo largo de una trayectoria específica, mientras se verifica la precisión y eficacia del controlador mediante la visualización en RVIZ.

La implementación del nodo de control fue fundamental para el funcionamiento autónomo del robot simulado en el entorno de ROS. Utilizando la información de odometría, el nodo calculó y envió comandos adecuados al robot para seguir una trayectoria predefinida. En este caso, se estableció una trayectoria en forma de cuadrado para probar el algoritmo del controlador, aunque se alentó a los usuarios a explorar diferentes formas de trayectorias, como un pentágono u otras figuras geométricas, para evaluar aún más la eficacia del controlador.

La verificación de los resultados se llevó a cabo en RVIZ, donde se observó el movimiento del robot a lo largo de la trayectoria establecida. Esta verificación permitió confirmar que el controlador funcionaba correctamente y que el robot seguía la trayectoria deseada de manera precisa y consistente.

Cabe destacar que el tipo de controlador utilizado, tanto para la velocidad angular como para la lineal, es un controlador de tipo PID, tuneado, es decir que fue calibrado específicamente para cada velocidad, este control permite mejorar la disminución de errores y controlar mejor el sistema a través del tiempo. Para hacer este controlador primero empezamos calculando los errores como se aprecia en la siguiente figura



Podemos observar que el error es la distancia entre el punto deseado y la posición en la que se encuentra nuestro robot. Esta relación se puede expresar matemáticamente a través de las siguientes ecuaciones

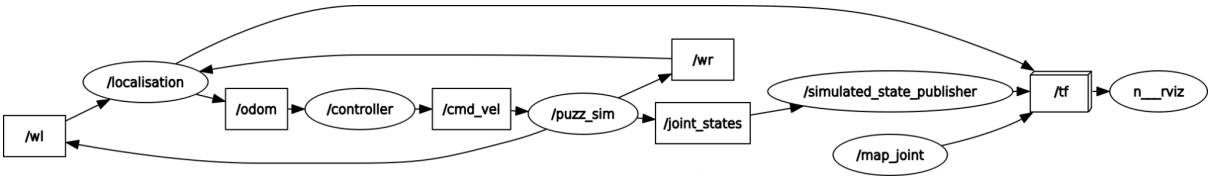
$$ed = \sqrt{e^2{}_x + e^2{}_y}$$

$$e\theta = arctan2(e_y - e_X) - \theta$$

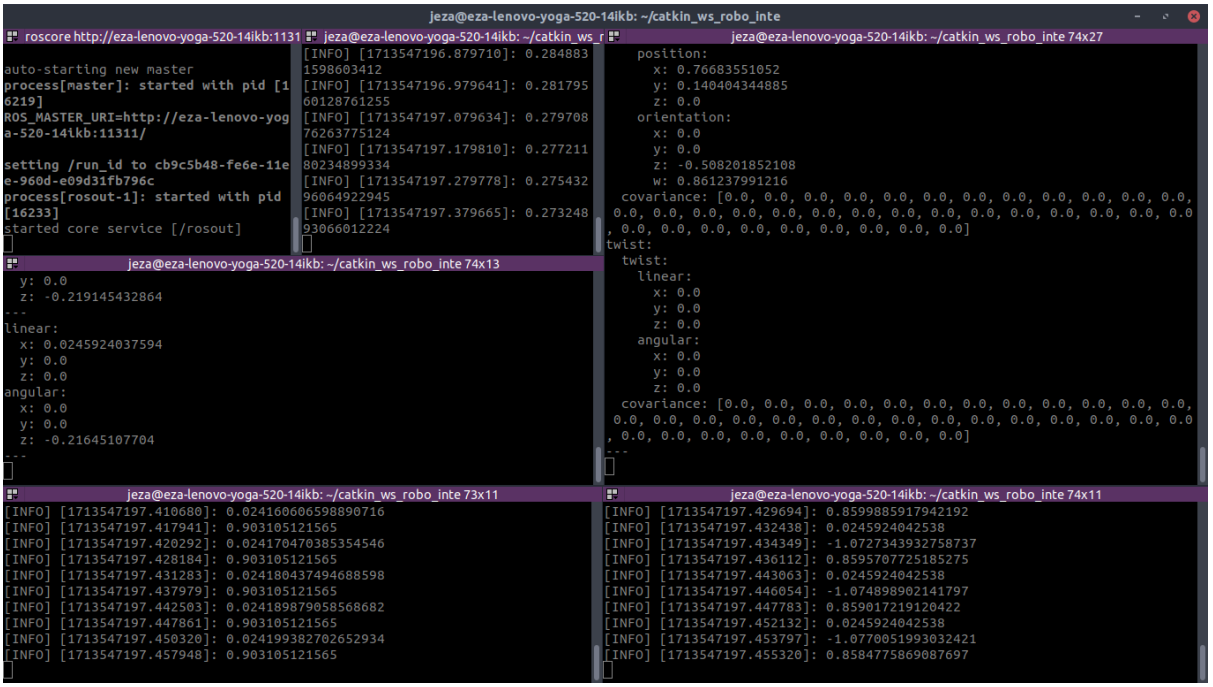
estas ecuaciones nos permiten modelar el controlador angular y lineal comparando los errore dados con cada modificación de lo parámetros de K(PID)

Resultados

Presentamos la estructura de nuestro sistema y como los nodos y los tópicos interactúan entre sí

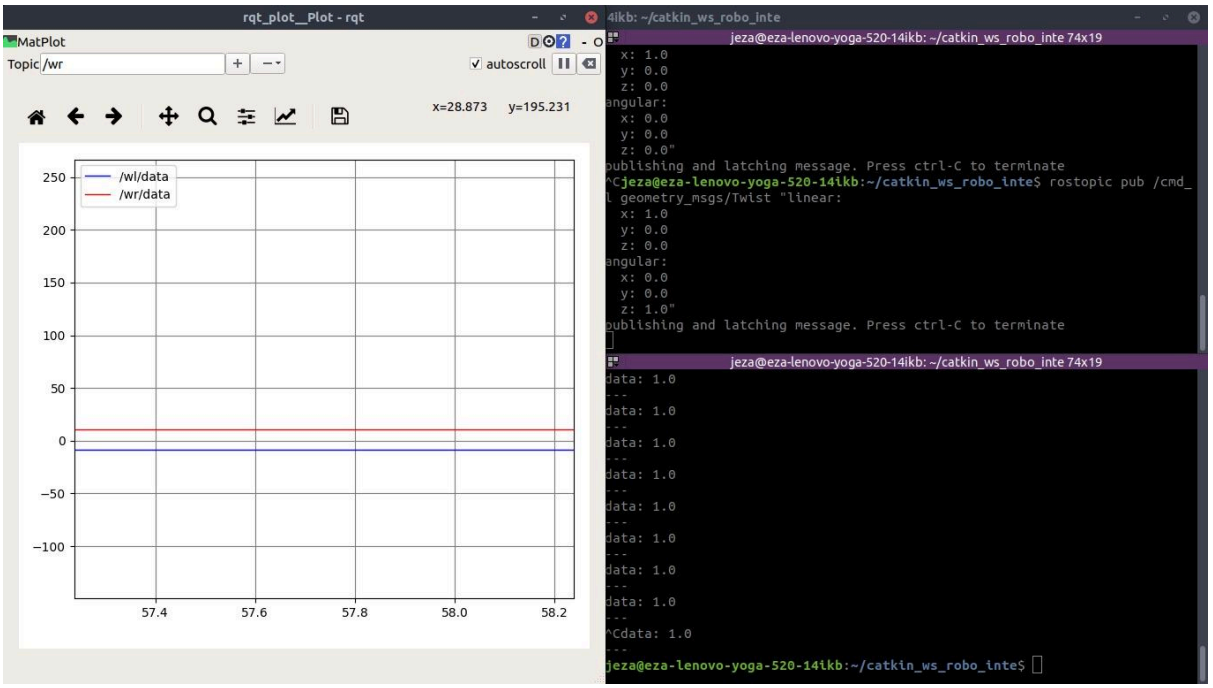


Para comenzar a calibrar controladores, así como para verificar que los mensajes que se estuvieran enviando fueran correctos, se procedió a ejecutar cada nodo por separado, y realizando un rostopic echo a cada tópico activo



Esto acción permitió tener un mayo control sobre el sistema, porque se tenían conocimiento del comportamiento de las variables durante todo el tiempo de ejecución lo que facilitó la detección de errores y ajustes que se podían realizar

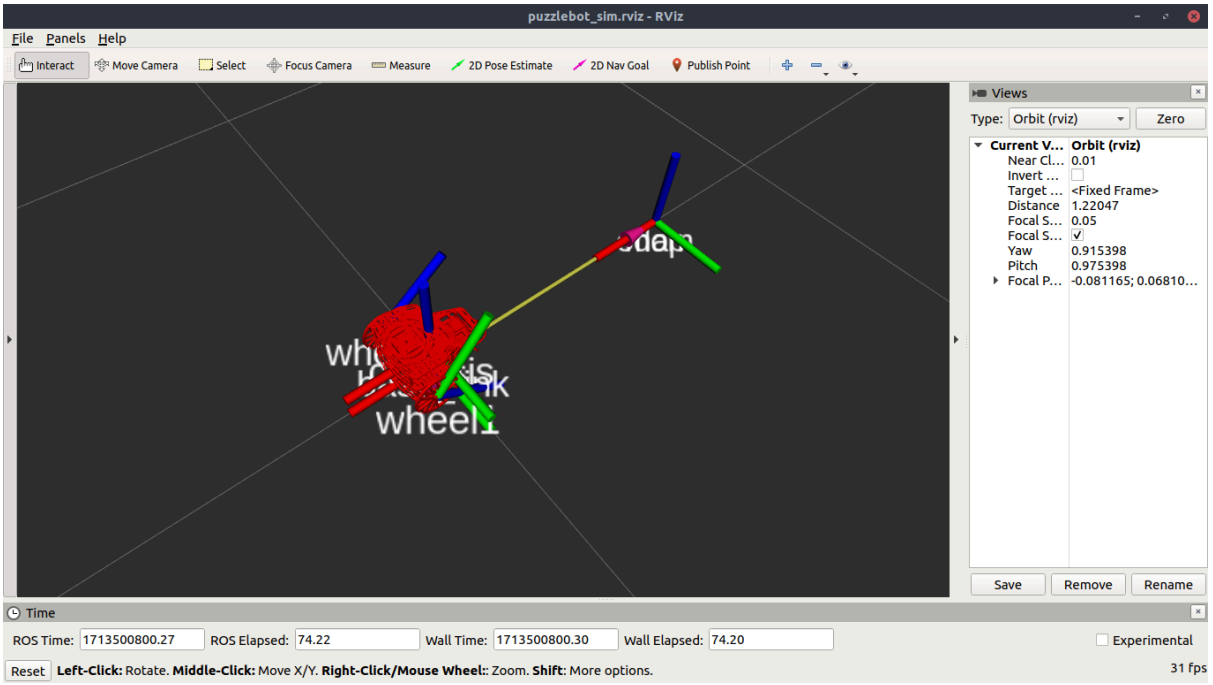
Tambíen se obtuvieron gráficas para ver el comportamiento de algunos tópicos como el de las ruedas, donde se observa que para velocidades angulares, es decir para rotar a un lado, una rueda gira en sentido contrario, negativo, y la otra en sentido positivo



Posteriormente se realizó la visualización con rviz donde observamos el comportamiento del robot, podemos observar que el controlador es un poco mejor cuando no se aplica la simulación esto puede deberse a que aumentan la cantidad de recursos necesarios para ejecutar el sistema y los cálculos que se hacen empezar a disminuir su precisión. También

vemos que el controlador angular es el que afecta principalmente a la efectividad de nuestro Non-homolimic robot ya que genera demasiados errores al sistema conforme a los errores van pasando.
Una demostración más amplia de los resultados es presentada en el siguiente video:

https://youtu.be/F9iJZc6aEsw?feature=shared



Conclusiones

En este proyecto, se logró desarrollar e implementar con éxito un sistema de simulación en ROS para el Puzzlebot, abordando diferentes aspectos clave como la simulación cinemática, la localización por odometría y el control de posición. A través de este proyecto, se pudo aplicar y consolidar los conocimientos teóricos adquiridos en el curso, así como desarrollar habilidades prácticas en el diseño, implementación y prueba de algoritmos de control para robots autónomos.

En la primera parte del proyecto, se creó un simulador cinemático para el Puzzlebot, lo que permitió modelar y simular su movimiento de manera precisa y eficiente. Posteriormente, se implementó un nodo de localización por odometría que calcula la posición y orientación del Puzzlebot en el espacio, lo que es fundamental para su capacidad de movimiento autónomo.

Finalmente, se desarrolló un nodo de control que permitió establecer la posición del Puzzlebot en un punto deseado, lo que completó el desafío planteado. A través de la integración de estos componentes, se logró controlar con éxito el movimiento del Puzzlebot a lo largo de una trayectoria predefinida, verificando los resultados utilizando RVIZ.

Referencias

ROS Wiki. (n.d.). geometry_msgs/PoseStamped. Retrieved from http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/PoseStamped.html

Valencia, J. A., V., O, A. M., & Rios, L. H. (2009). MODELO CINEMÁTICO DE UN ROBOT MÓVIL TIPO DIFERENCIAL y NAVEGACIÓN a PARTIR DE LA ESTIMACIÓN ODOMÉTRICA. Redalyc.org. https://www.redalyc.org/articulo.oa?id=84916680034

ROS Wiki. (n.d.). URDF. Retrieved from http://wiki.ros.org/urdf