SLAM y Navegación en ROS2

Eugenio Ivorra

En esta práctica, se abordarán las bases y aplicaciones avanzadas de SLAM (Simultaneous Localization and Mapping) y navegación autónoma usando ROS 2 y Gazebo, enfocándonos en la implementación práctica y programática de estas técnicas. El dominio de estas herramientas es esencial para el desarrollo de competencias en robótica autónoma y la realización de tareas de mapeo y navegación en entornos simulados. Los objetivos específicos de este módulo son:

1. Comprender los fundamentos teóricos y prácticos detrás del SLAM y su importancia en la robótica autónoma.

2. Utilizar RViz para visualizar y entender la información sensorial y los datos de estado del robot en un contexto de SLAM y navegación.

3. Familiarizarse con la configuración y las capacidades del TurtleBot3 como plataforma estándar en la simulación robótica.

4. Implementar y configurar slam_toolbox en ROS 2 para realizar tareas de mapeo y localización en tiempo real.

5. Integrar y manipular el stack de navegación NAV2 en ROS 2 para la planificación y ejecución de rutas autónomas.

6. Iniciar y gestionar simulaciones de robots en Gazebo, facilitando el desarrollo de competencias en el entorno virtual.

7. Ejecutar Nav2 y ROS2slam_toolbox simultáneamente, aprendiendo a manejar su interacción y configuración.

8. Generar y almacenar mapas de entornos utilizando slam_toolbox, comprendiendo la estructura y contenido de los mapas generados.

9. Desarrollar competencias en navegación autónoma del robot utilizando mapas y la suite de navegación NAV2.

10. Aprender a programar la navegación del robot de manera programática utilizando Python, estableciendo poses iniciales y enviando objetivos y waypoints para la navegación.

Índice general

1 Conceptos y herramientas de SLAM y Navegación autónoma

1.1 Introducción

La robótica móvil ha traspasado las barreras de la ciencia ficción para anclar su presencia en el corazón del desarrollo tecnológico y la innovación contemporánea. Un área de especial interés es la de los sistemas autónomos móviles, donde la navegación y la comprensión del entorno son cruciales para el avance y la implementación efectiva de robots en el mundo real.

En el centro de esta revolución robótica está el problema conocido como SLAM (Simultaneous Localization and Mapping) por sus siglas en inglés, o Localización y Mapeo Simultáneos. SLAM se refiere a la capacidad de un robot para construir un mapa de un entorno desconocido mientras simultáneamente calcula su ubicación dentro de este mapa. Este desafío no es trivial; exige una amalgama de técnicas provenientes de campos tan variados como la visión por computadora, la fusión sensorial, la inteligencia artificial y la geometría computacional. La esencia del SLAM es su capacidad para realizar correlaciones entre la percepción sensorial y la movilidad, permitiendo al robot reconocer y posicionarse dentro de su entorno con un grado de incertidumbre minimizado.

La navegación autónoma, por otro lado, es el proceso mediante el cual un robot móvil se dirige de un punto a otro de manera eficiente y segura, a menudo en entornos dinámicos y en presencia de obstáculos imprevistos. Esto requiere no solo la percepción precisa del entorno proporcionada por SLAM sino también la planificación de trayectorias, el control de movimiento, y la toma de decisiones en tiempo real. La navegación autónoma es el arte y la ciencia de la movilidad independiente y se sitúa en la intersección del conocimiento espacial y la ejecución de tareas.

La confluencia de SLAM y la navegación autónoma define el campo de la robótica móvil y establece los cimientos sobre los cuales los robots pueden interactuar y adaptarse a sus entornos de manera inteligente. Los robots que dominan estas técnicas se convierten en entidades capaces de asistir, explorar, y operar con una impresionante autonomía, desde los pasillos de nuestras casas hasta los confines más remotos del espacio o las profundidades de los océanos.

Con la llegada de ROS2 las posibilidades de SLAM y navegación autónoma han alcanzado un nuevo horizonte. ROS2 no solo facilita la implementación de estos algoritmos complejos sino que también permite la simulación y el despliegue eficiente a través de su amplia gama de herramientas gráficas y algoritmos de código abierto. Herramientas como Gazebo para la simulación del mundo real y RVIZ para la visualización en tiempo real, proporcionan una plataforma inigualable para desarrollar y probar algoritmos de robótica en un entorno controlado.

Dentro del ecosistema de ROS2, destaca Slam Toolbox y Cartographer como poderosas soluciones SLAM de código abierto. Slam Toolbox es una colección de herramientas versátiles y eficientes para el mapeo 2D y 3D, mientras que Cartographer ofrece capacidades de mapeo en tiempo real para aplicaciones que requieren los más altos niveles de fidelidad y robustez. En esta práctica, nos enfocaremos en el uso de Slam Toolbox como nuestra herramienta principal para aprender las capacidades del SLAM en la robótica móvil autónoma usando una simulación de Gazebo y el famoso robot Turtlebot3.

1.2 RViz: El Visualizador de ROS

RViz (Robot Visualization) es una herramienta gráfica 3D incluida con el Robot Operating System (ROS) que permite visualizar estados, sensores y otra información de un robot. Es ampliamente utilizado en la comunidad de robótica para depurar algoritmos y comprender la operación de robots en tiempo real o mediante datos registrados previamente. En la imagen figura 1.1 se muestra un ejemplo de esta interfaz.

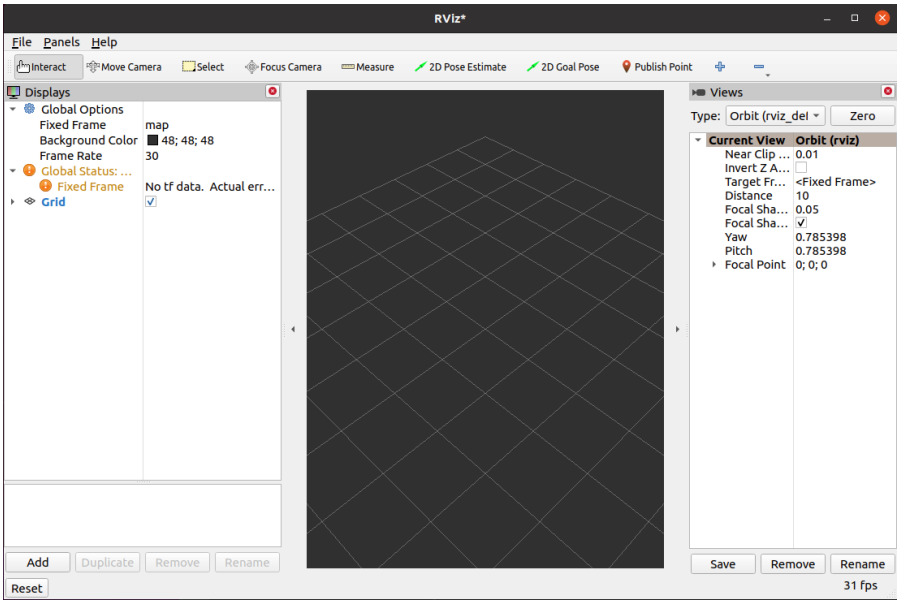

Figura 1.1: Visualización de RViz

1.2.1 Características Principales

- **Interfaz Gráfica Intuitiva**: RViz proporciona una ventana principal donde se pueden visualizar datos en un contexto 3D, y paneles adicionales para configuración y control.
- **Visualización de Datos**: Es capaz de mostrar una amplia variedad de datos, desde imágenes de cámaras y nubes de puntos LIDAR, hasta mapas, rutas y transformaciones entre marcos de referencia.
- **Plugins Extensibles**: RViz está diseñado para ser extensible, permitiendo a los usuarios crear y agregar sus propios visualizadores de datos.
- **Interactividad**: Los usuarios pueden interactuar con la visualización, moviendo la vista, seleccionando objetos y manipulando marcadores interactivos.

1.2.2 Funcionamiento

La función principal de RViz es subscribirse a los temas de ROS que contienen datos sensoriales y de estado, y visualizar esos datos en tiempo real. Su funcionamiento se basa en los siguientes componentes:

- **Displays**: Son los componentes que interpretan y muestran datos de un tipo específico. Por ejemplo, un 'Display' podría visualizar datos LIDAR como una nube de puntos en 3D.
- **Views**: Definen cómo se presenta el mundo 3D al usuario. Puede ser en perspectiva, ortográfico, entre otros.
- **Tools**: Permiten la interacción con la visualización. Esto incluye mover la vista, seleccionar datos, medir distancias, entre otras tareas.
- **Fixed Frame**: Es el marco de referencia que RViz usa para interpretar todas las transformaciones y datos. Es esencial elegir un marco de referencia fijo adecuado para que las visualizaciones tengan sentido.

RVIZ se utiliza principalmente para:

- **Depuración**: RViz es invaluable para depurar problemas con robots, permitiendo a los desarrolladores ver exactamente lo que el robot está "viendo" y "pensando".
- **Diseño y Planificación**: Al simular entornos y rutas, RViz puede ser utilizado para planificar cómo un robot navegará en un espacio.
- **Educación**: RViz es una herramienta educativa poderosa, ayudando a los estudiantes a visualizar conceptos complejos de robótica.

RViz es una herramienta esencial en el arsenal de cualquier desarrollador de ROS. Con su capacidad para visualizar una amplia variedad de datos en un contexto significativo, juega un papel clave en el desarrollo, depuración y operación de robots complejos.

1.3 TurtleBot3

El TurtleBot3 es uno de los robots móviles más populares en la comunidad de robótica, particularmente para aquellos que trabajan con ROS (Robot Operating System). Desarrollado por ROBOTIS, el TurtleBot3 es conocido por su simplicidad, modularidad y accesibilidad.

- **Diseño Modular**: TurtleBot3 ha sido diseñado con un enfoque modular, lo que permite a los usuarios y desarrolladores personalizar y adaptar el robot según sus necesidades específicas.
- **Diferentes Modelos**: Está disponible en varias versiones, como el 'Burger', 'Waffle' y 'Waffle Pi', que varían en tamaño, capacidad y funcionalidad.
- **Integración con ROS**: TurtleBot3 es completamente compatible con ROS, lo que facilita la implementación de diversos algoritmos de robótica y navegación.

- **Costo-Efectivo**: A diferencia de muchos otros robots de investigación, TurtleBot3 es relativamente asequible, lo que lo hace ideal para la educación y la investigación.

- **Sensores Incorporados**: Equipado con sensores como Lidar, cámaras y sensores IMU, lo que lo hace capaz de realizar tareas como la navegación autónoma, el seguimiento de objetos y la detección de obstáculos.

Debido a su versatilidad y accesibilidad, TurtleBot3 es utilizado en una amplia variedad de aplicaciones:

- **Educación**: Es una herramienta ideal para enseñar conceptos de robótica y programación a estudiantes.

- **Investigación**: Muchos investigadores lo utilizan como plataforma para probar nuevos algoritmos y soluciones en robótica.

- **Desarrollo de Software**: Con su integración completa con ROS, es una plataforma excelente para el desarrollo y prueba de software robótico.

- **Robot de Servicio**: Puede ser adaptado para tareas específicas, como la entrega de objetos en entornos interiores.

Existen tres versiones principales del TurtleBot3 figura 1.2, cada una nombrada metafóricamente para reflejar su tamaño y capacidades: Burger, Waffle y Waffle Pi. Estas variantes ofrecen diferentes niveles de funcionalidad y rendimiento, adaptándose a una variedad de necesidades educativas y de investigación en robótica.

El TurtleBot3 ha establecido un estándar en la comunidad de robótica, proporcionando una plataforma robusta y versátil para la educación, investigación y desarrollo en robótica. Su diseño modular y la integración con herramientas de software líderes en la industria, como ROS, lo hacen invaluable para aquellos que buscan aprender y avanzar en el campo de la robótica.

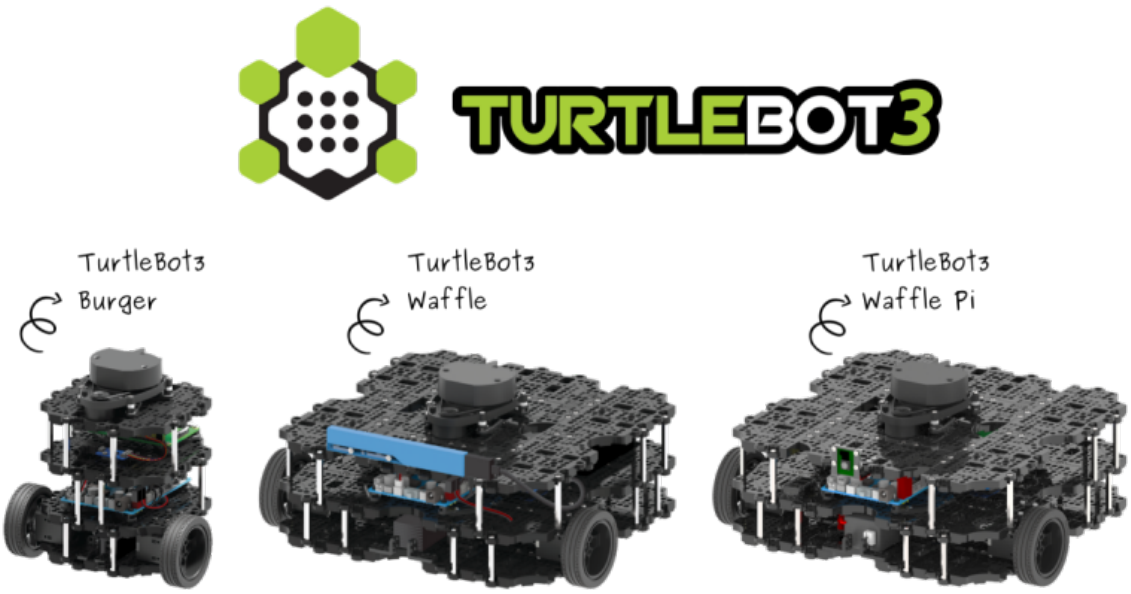

Figura 1.2: Turtlebot3

1.4 Slam_ toolbox

El paquete *slam_ toolbox* es una herramienta avanzada en el ecosistema ROS (Robot Operating System) que permite a los robots construir un mapa del entorno mientras mantienen su localización dentro de ese mapa. El término "SLAM" proviene de las siglas en inglés para "Simultaneous Localization and Mapping", que se traduce como "Localización y Mapeo Simultáneos".

1.4.1 Características de slam_ toolbox

- **Modos de operación**: Puede funcionar en un modo en línea para mapeo en tiempo real o en un modo fuera de línea para procesar grandes conjuntos de datos de bolsas.
- **Mapeo continuo**: A diferencia de otros paquetes SLAM que requieren que el robot siempre empiece desde un punto conocido, *slam_ toolbox* puede reanudar el mapeo a partir de cualquier lugar.
- **Optimización del grafo**: Permite la corrección de bucle y relocalización para mejorar la precisión del mapa.
- **Flexibilidad**: Proporciona una amplia variedad de parámetros que se pueden ajustar según las necesidades del usuario y el hardware del robot.

1.4.2 ¿Cómo funciona?

El proceso básico de SLAM implica recopilar datos del entorno mediante sensores, como LIDAR o cámaras RGB-D, y luego usar estos datos para construir un mapa y localizar el robot dentro de ese mapa simultáneamente. *slam_ toolbox* hace esto de manera eficiente al mantener un grafo de poses del robot y características del entorno, y optimizando este grafo para minimizar el error total.

Para utilizar *slam_ toolbox* con un robot, es esencial que el robot esté equipado con sensores adecuados, como un LIDAR. Una vez que el robot está configurado, el paquete se puede lanzar y configurar mediante archivos de parámetros específicos. A medida que el robot se mueve y recopila datos, el toolbox procesará estos datos y construirá un mapa en tiempo real. En la navegación robótica, los sistemas de planificación son esenciales para garantizar que un robot pueda desplazarse de un punto a otro de manera efectiva y segura. Estos sistemas se basan en diversos algoritmos para determinar las rutas óptimas y reaccionar ante obstáculos. En este documento, se describen dos tipos principales de planificadores: el global y el local, así como sus respectivos roles y algoritmos fundamentales.

1.4.3 Planificador Global

El planificador global es responsable de proporcionar una estrategia de alto nivel para alcanzar un objetivo determinado desde un punto de partida. Se basa en algoritmos de búsqueda en grafos, como el A* o Dijkstra, para determinar el camino óptimo.

Características principales:

- **Ruta de Alto Nivel**: Establece un camino en base a un mapa previamente conocido, sin considerar obstáculos dinámicos.
- **Basado en Mapa**: Opera sobre un mapa estático del entorno.
- **No Reactivo**: Una vez establecida la ruta, no cambia a menos que se requiera una nueva planificación.

1.4.4 Planificador Local

El planificador local, en contraste con el global, se encarga de la navegación reactiva, ajustándose dinámicamente a los obstáculos. Se fundamenta en algoritmos como el Campo Potencial y el Vector Field Histogram (VFH).

Características principales:

- **Reacción a Obstáculos**: Capacidad para esquivar obstáculos en tiempo real.
- **Navegación a Corto Plazo**: Se enfoca en el movimiento inmediato del robot, adaptándose a cambios en el entorno cercano.
- **Independencia del Mapa**: Puede operar sin un mapa, adaptándose a entornos desconocidos o cambiantes.

Para una navegación eficiente, ambos planificadores operan conjuntamente. El planificador global provee una estrategia general, mientras que el local ajusta la ruta en tiempo real. Es análogo a un sistema GPS en un vehículo: el GPS proporciona la ruta, pero el conductor (planificador local) realiza ajustes basados en condiciones del tráfico o bloqueos inesperados en la carretera.

La combinación de planificadores global y local es esencial para garantizar una navegación robótica eficiente y segura. Mientras el primero ofrece una visión general del trayecto, el segundo se encarga de los desafíos inmediatos, asegurando que el robot se adapte a las condiciones cambiantes del entorno.

1.4.5 Conclusiones

Mientras que *slam_ toolbox* es una herramienta poderosa, la calidad del mapa resultante depende en gran medida de la calidad y precisión de los sensores del robot, así como de la configuración y parámetros proporcionados al toolbox. Por lo tanto, es crucial comprender bien las capacidades y limitaciones de su hardware y software específicos al utilizar esta herramienta.

1.5 NAV2 en Robótica Autónoma

NAV2 Macenski et al. 2020 es el sucesor espiritual del ROS Navigation Stack, contando con el respaldo profesional y desplegando tipos de tecnología similares a los que impulsan los Vehículos Autónomos. Esta tecnología ha sido adaptada, optimizada y reestructurada para la robótica móvil y de superficie. El proyecto busca encontrar una manera segura de que un robot móvil se mueva para completar tareas complejas a través de muchos tipos de entornos y clases de cinemática robótica. NAV2 no solo puede moverse del punto A al punto B, sino que también puede tener poses intermedias y representar otros tipos de tareas como el seguimiento de objetos y más. NAV2 es un marco de navegación de alta calidad, utilizado en más de 50 empresas a nivel mundial.

Proporciona percepción, planificación, control, localización, visualización y mucho más para construir sistemas autónomos altamente fiables. Esto completará la modelización ambiental a partir de datos sensoriales, la planificación de caminos dinámicos, el cálculo de velocidades para motores, la evasión de obstáculos, la representación de regiones semánticas y objetos, y la estructuración de comportamientos robóticos de alto nivel.

NAV2 utiliza árboles de comportamiento para crear un comportamiento de navegación inteligente y personalizado mediante la orquestación de muchos servidores modulares independientes. Un servidor de tareas se puede utilizar para calcular una ruta, el esfuerzo de control, la recuperación o cualquier otra tarea relacionada con la navegación. Estos servidores independientes se comunican con el árbol de comportamiento (BT) a través de una interfaz ROS como un servidor de acción o servicio. Un robot puede utilizar potencialmente muchos árboles de comportamiento diferentes que le permitan realizar muchos tipos de tareas únicas.

El diagrama de figura 1.3 muestra una buena primera impresión de la estructura de NAV2. Las entradas esperadas a NAV2 son transformaciones TF conforme a REP-105, una fuente de mapa si se utiliza la Capa de Costmap Estático, un archivo XML de BT y cualquier fuente de datos sensoriales relevante. Luego, proporcionará comandos de velocidad válidos para los motores de un robot holonómico o no holonómico a seguir. Actualmente admite todos los principales tipos de robots: tipos de base holonómicos, de accionamiento diferencial, con patas y Ackermann (tipo automóvil).

Cuenta con herramientas para:

- Cargar, servir y almacenar mapas (*Map Server*).
- Localizar el robot en el mapa (*AMCL*).
- Planificar una ruta de A a B evitando obstáculos (*Nav2 Planner*).
- Controlar el robot mientras sigue la ruta (*Nav2 Controller*).

- Suavizar los planes de ruta para que sean más continuos y factibles (*Nav2 Smoother*).
- Convertir los datos del sensor en una representación del mundo de costmap (*Nav2 Costmap 2D*).
- Construir comportamientos robóticos complicados utilizando árboles de comportamiento (*Nav2 Behavior Trees y BT Navigator*).
- Calcular comportamientos de recuperación en caso de fallo (*Nav2 Recoveries*).
- Seguir waypoints secuenciales (*Nav2 Waypoint Follower*).
- Gestionar el ciclo de vida y el watchdog para los servidores (*Nav2 Lifecycle Manager*).
- Plugins para habilitar sus propios algoritmos y comportamientos personalizados (*Nav2 Core*).
- Monitorear datos de sensores crudos para una colisión inminente o situación peligrosa (*Collision Monitor*).
- API de Python3 para interactuar con NAV2 de manera pythonica (*Simple Commander*).
- Un suavizador en las velocidades de salida para garantizar la factibilidad dinámica de los comandos (*Velocity Smoother*).

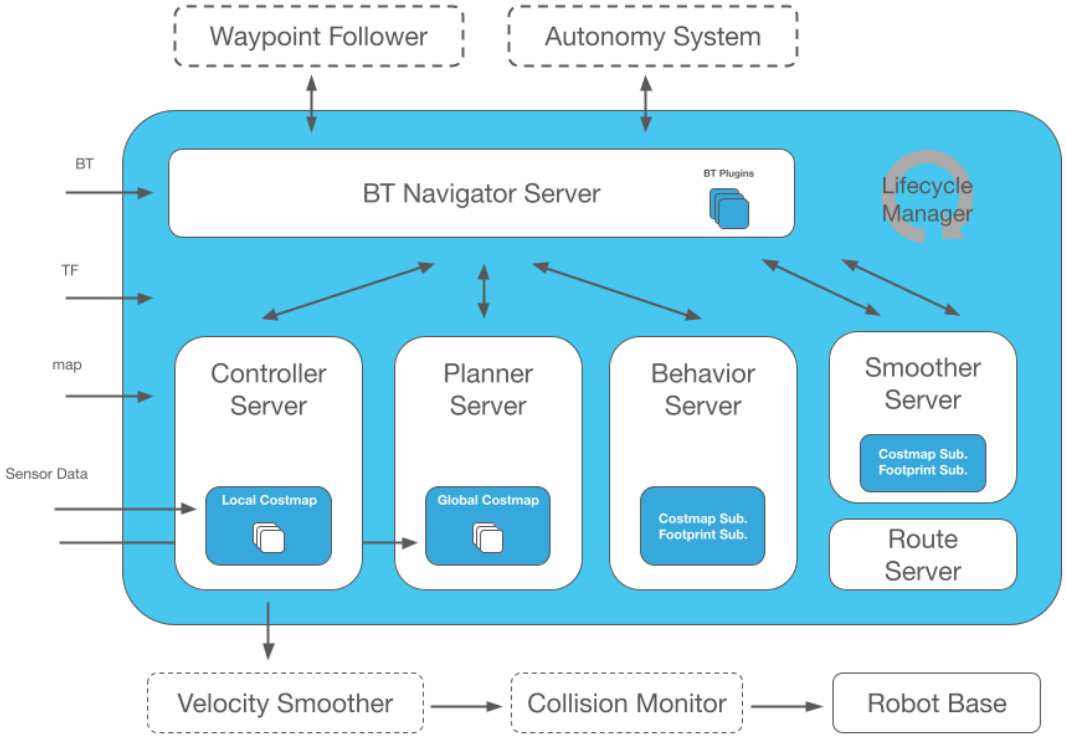

Figura 1.3: Diagrama de Bloques de Navigation2

2 SLAM usando RVIZ y Gazebo

En esta práctica usaremos el Turtlebot3 para obtener un mapa de un entorno simulado en GAZEBO usando SLAM. Siguiendo esta metodología se podría aplicar al robot real e incluso a otros tipos de robots. Cuando desees usar tu propio robot, asegúrate de poder iniciarlo, controlarlo (con teclado, joystick, etc.) y tener un mensaje LaserScan publicado en un topic /scan. Estos son los requisitos principales para que funcione el *slam_ toolbox*.

2.1 Iniciar la simulación con el robot

Para iniciar el stack de Turtlebot3, primero debemos exportar una variable de entorno para especificar qué versión deseamos lanzar (burger, waffle, waffle pi). Por defecto en el .bashrc deberíamos tener definido que usaremos el burguer pero si quieras cambiarlo podemos hacerlo con este comando:

```
export TURTLEBOT3_MODEL=burger
```

Para iniciar la simulanción en Gazebo lanzamos este comando:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

El resultado debería de ser como el de la figura 2.1

Ahora hay que asegurarse de que exista un topic llamado /scan listando los topics:

```
ros2 topic list
```

Observamos que hay un topic /scan. Y si verificamos el tipo:

```
ros2 topic info /scan
ros2 interface show sensor_msgs/msg/LaserScan
```

¡Perfecto! Hemos comprobado que el tipo de mensaje que se envía en el topic /scan es un LaserScan de los paquetes sensor_msgs. Este mensaje contiene la información del Lidar del robot que usaremos para reconstruir el mapa 2D. Concretamente en el campo float32[] ranges tenemos las medidas de distancia que mide el Lidar.

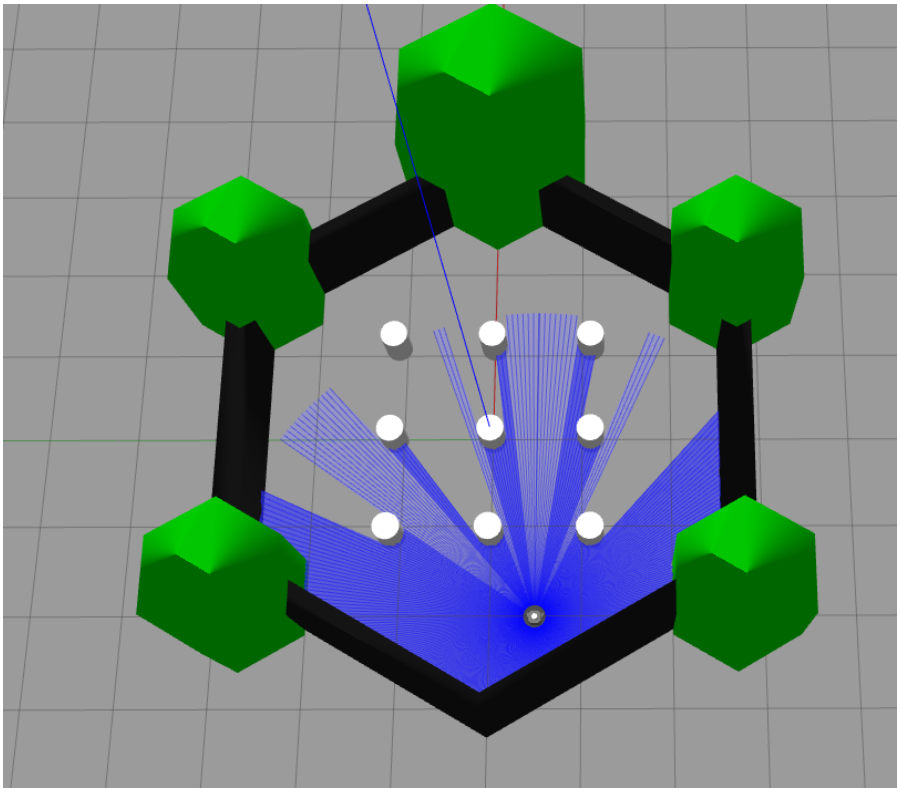

Figura 2.1: Simulación de Gazebo de turtlebot3

2.2 Iniciar Nav2 y ROS2 slam_toolbox

Para esto, necesitaremos 4 terminales.

En el primer terminal, ya habrás iniciado la simulación de Gazebo de tu robot usando el ejemplo de Turtlebot3.

En un segundo terminal, inicia el stack Navigation 2. Nota: al estar usando Gazebo, agrega "use_sim_time:=True" para usar el tiempo de simulación. Si estás usando un robot real hay que omitir este argumento.

```
ros2 launch nav2_bringup navigation_launch.py use_sim_time:=True
```

En el tercer terminal, inicia el *slam_toolbox*.

```
ros2 launch slam_toolbox online_async_launch.py use_sim_time:=True
```

Todo se ha lanzado correctamente. Ahora, lo que verás será una serie de registros en diferentes terminales. En el cuarto terminal iniciaremos RViz para visualizar lo que está sucediendo.

Para facilitar las cosas, podemos comenzar con una configuración RViz ya existente para Nav2.

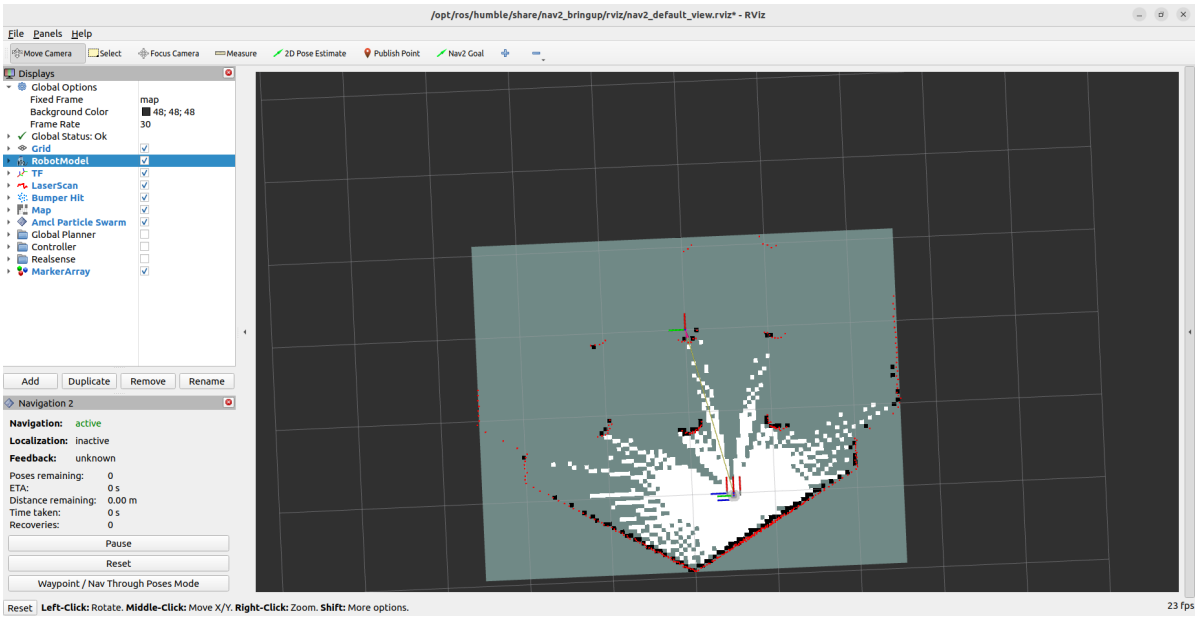

Figura 2.2: ROS2 Nav2 RViz2 vista predeterminada para SLAM

```
ros2 run rviz2 rviz2 -d
 ↪  /opt/ros/humble/share/nav2_bringup/rviz/nav2_default_view.rviz
```

Con esto, deberías ver una ventana como esta figura 2.2.

Como puedes observar en el menú de la izquierda, muchas cosas ya están configuradas: TF, LaserScan, AMCL, planificador global y local, etc.

Para obtener una mejor vista para SLAM, deseleccionaré el "Global Planner" (planificador global) y el "Controller" (planificador local) y marcaré el RobotModel. También puedes desactivar opcionalmente todos los "TF" excepto base_link para saber a donde apunta el robot.

2.3 Generar un mapa con slam_ toolbox

Como se puede ver en la captura anterior (figura 2.2), tenemos 3 posibles colores para cualquier píxel en el mapa:

- Blanco: espacio libre.
- Negro: obstáculo.
- Gris: desconocido.

Ahora, todo lo que necesitamos hacer es mover el robot en el mapa para que los píxeles grises (desconocidos) se vuelvan blancos o negros. Cuando tengamos un resultado que nos satisfaga, podemos guardar el mapa.

Para hacer que el robot se mueva, necesitas iniciar el nodo para mover tu robot, ya sea con un teclado, joystick, etc.

Para hacer que el Turtlebot3 se mueva usando el teclado, abre otro terminal y ejecuta:

```
ros2 run turtlebot3_teleop teleop_keyboard
```

Ahora haz que el robot se mueva por el mapa hasta que estés satisfecho con la reconstrucción.

El mapa que obtuve después de 20-30 segundos se ve así figura 2.3.

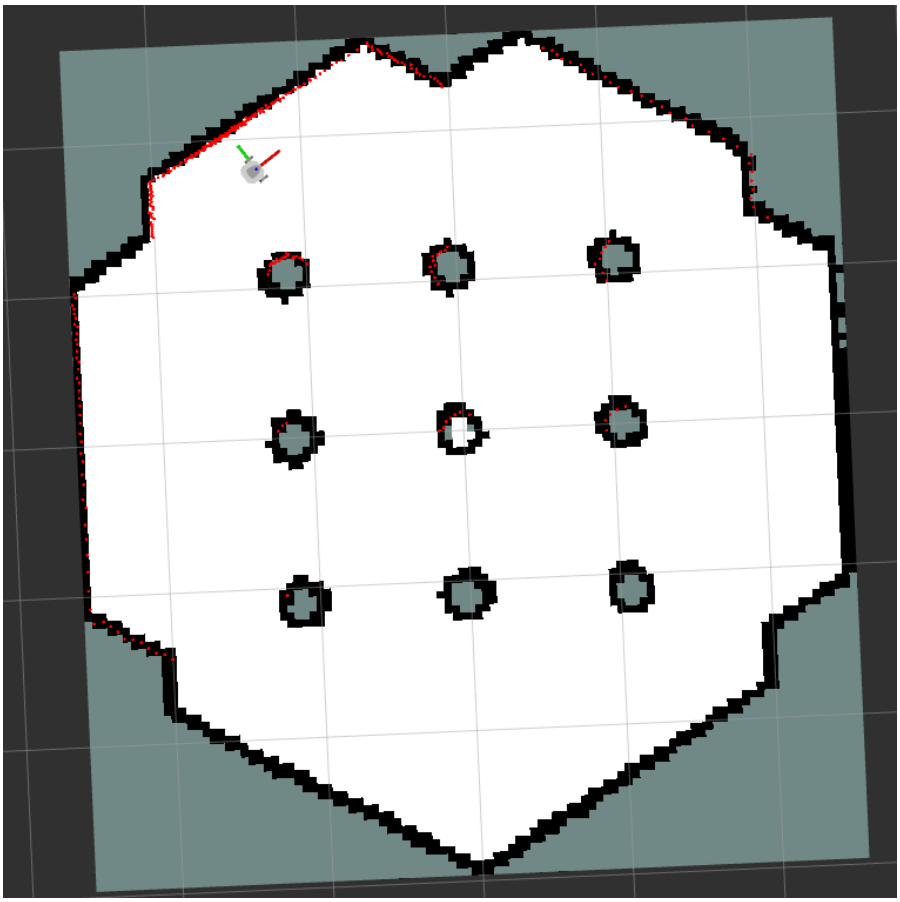

Figura 2.3: Mapa en Rviz2 generado por slam_toolbox

2.4 Guardar el mapa

Una vez que tengas un mapa que te parezca lo suficientemente bueno, puedes guardarlo.

Y antes de guardar el mapa, ¡asegúrate de no detener ninguno de los primeros 4 terminales, de lo contrario tendrías que comenzar todo de nuevo!

Para guardar el mapa:

```
ros2 run nav2_map_server map_saver_cli -f my_map
```

Ahora deberías tener 2 nuevos archivos en tu directorio llamados *my_ map.yaml* y *my_ map.pgm*.

De esta forma has generado exitosamente un mapa con el paquete ROS2 *slam_ toolbox*. Este paquete suele funcionar bastante bien y es bastante rápido para generar el mapa.

2.5 Contenido y estructura del mapa

El archivo `my_map.pgm` es una representación gráfica del mapa. En él, los píxeles blancos indican un espacio libre, los negros representan obstáculos y los grises denotan áreas desconocidas o no exploradas.

Por otro lado, el archivo `my_map.yaml` actúa como una guía de metadatos para el mapa y es esencial para el funcionamiento del stack de Navegación. Su estructura es la siguiente:

```yaml
image: my_map.pgm
mode: trinary
resolution: 0.05
origin: [-2.97, -2.57, 0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.25
```

A continuación, se ofrece una explicación detallada de algunos campos clave:

- **image**: Define la ruta relativa del archivo PGM asociado al mapa.
- **resolution**: Especifica la resolución del mapa en términos de metros por píxel. En este caso, un píxel es equivalente a 5 cm, indicando la precisión del mapa.
- **origin**: Señala las coordenadas iniciales del robot en relación con el punto inferior izquierdo del mapa.
- **negate**: Si se establece en 1, los espacios libres y ocupados en el mapa se invertirán.
- **occupied_thresh** y **free_thresh**: Estos valores determinan las probabilidades para considerar un espacio como ocupado o libre, respectivamente. Si, por ejemplo, un píxel tiene una probabilidad superior al 65% de tener un obstáculo, se considerará como espacio ocupado.

2.6 Navegación del Robot con el Mapa y Nav2

Con tu mapa ya creado y la configuración adecuada, es posible que el robot navegue de manera autónoma y esquive obstáculos.

2.6.1 Puesta en marcha de la Navegación 2 para el robot

Para evitar problemas potenciales con RViz y Gazebo, es conveniente empezar en un entorno limpio. Por ello, detén todas las ejecuciones, cierra y abre nuevamente todas las terminales.

A continuación, inicia el robot:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

En una terminal diferente, activa el stack de Navegación, suministrando el mapa como un argumento:

```
ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True
↪  map:=path/to/my_map.yaml
```

Si el mapa no aparece en RViz, busca y modifica las opciones del tópic `map` en el panel izquierdo, cambiando de `volatile` a `transient local`. Si el problema persiste, considera reiniciar todas las terminales o incluso el computador.

Una vez visualices el mapa en RViz, probablemente no veas la representación del robot. Esto se debe a que Nav2 requiere una estimación inicial de la pose en 2D del robot para operar correctamente.

2.6.2 Establecimiento de la pose en 2D y definición de objetivos de navegación

En RViz, selecciona la herramienta `2D pose estimate` y marca en el mapa la ubicación y orientación del robot, tal y como se refleja en Gazebo. Para ello selecciona en la barra superior de RVIZ 2D Pose Estimate y marca en el mapa la posición y orientación del robot. Intenta que sea lo más aproximada posible para que funcione bien, aunque luego el algoritmo la refina y corrige.

Una vez hayas establecido la pose, RViz debería mostrarte una representación adecuada del robot en el mapa (figura 2.4).

Ahora, estás listo para enviar comandos de navegación. Utiliza la herramienta `Nav2 Goal` en RViz (en la barra superior), elige un punto y orientación en el mapa, y observa cómo el robot inicia su movimiento hacia esa posición. Puedes corroborar el desplazamiento del robot en Gazebo.

Experimenta enviando el robot a diferentes puntos, tanto alcanzables como aquellos que representen un desafío para su navegación.

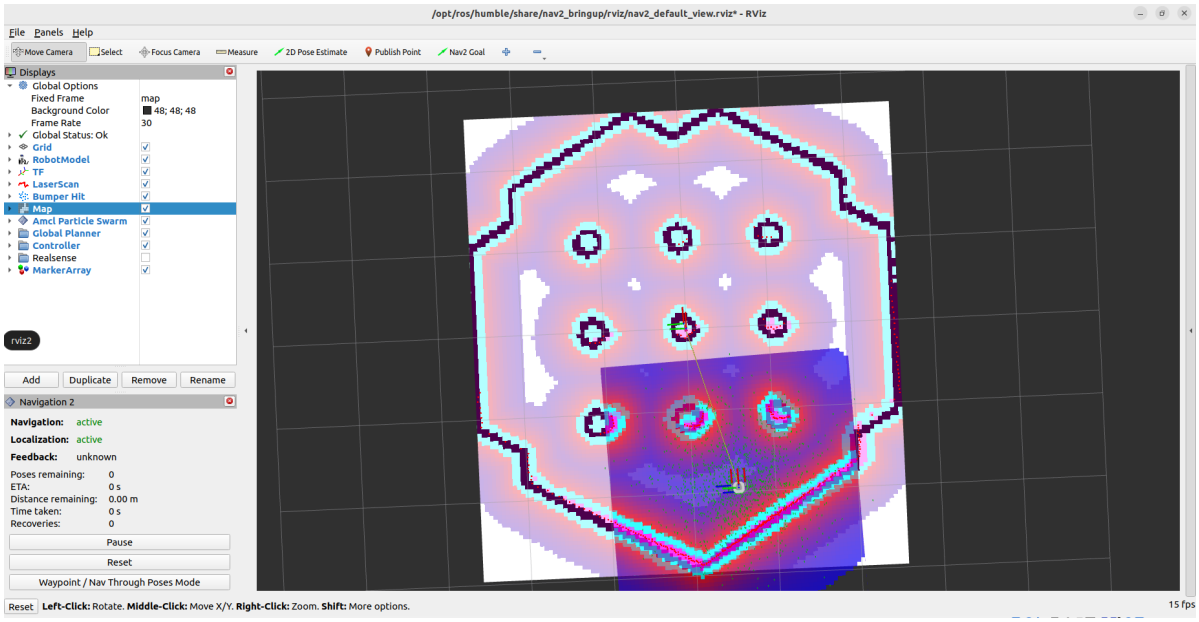

Figura 2.4: Nav2 - Navegación a partir del mapa generado

También se pueden establecer Waypoints para que el robot siga una trayectoria más compleja que atraviese los puntos que tu definas. Para ello activa en el panel izquierdo de RVIZ el botón Startup luego Waypoint /Nav Through Poses mode y ves dándole poses objetivo con Nav2 Goal. Cuando ya estés satisfecho puedes activar Start Waypoint Following para que el robot siga esos puntos en el orden que le has indicado.

3 Navegación de forma programática en Python

En este capítulo, aprenderemos cómo realizar una navegación de forma programáticamente de un mapa previamente creado. Para ello escribiremos un código Python que interactúe con el stack de navegación de ROS2.

3.1 Establecer la pose inicial

A continuación, crea un script de Python en tu paquete de ROS2. Evita usar el nombre nav2_simple_commander.py para evitar conflictos con la biblioteca.

```
touch nav2_test.py
chmod +x nav2_test.py
```

Ahora, vamos a escribir nuestro programa en Python.

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

# Importar las bibliotecas necesarias
import rclpy
from nav2_simple_commander.robot_navigator import BasicNavigator
from geometry_msgs.msg import PoseStamped
import tf_transformations

def main():
    # Iniciar la comunicación con ROS 2
    rclpy.init()

    # Crear una instancia del BasicNavigator para gestionar la navegación
    nav = BasicNavigator()

    # Crear y establecer la pose inicial del robot en el mapa
    initial_pose = PoseStamped()
    initial_pose.header.frame_id = "map"
    initial_pose.header.stamp = nav.get_clock().now().to_msg()
    initial_pose.pose.position.x = -2.0
    initial_pose.pose.position.y = -0.5
```

```
    initial_pose.pose.position.z = 0.0

    # Convertir los ángulos de Euler a cuaternión y establecer la orientación
    ↪  inicial
    qx, qy, qz, qw = tf_transformations.quaternion_from_euler(0.0, 0.0, 0.0)
    initial_pose.pose.orientation.x = qx
    initial_pose.pose.orientation.y = qy
    initial_pose.pose.orientation.z = qz
    initial_pose.pose.orientation.w = qw

    # Configurar la pose inicial en el navegante
    nav.setInitialPose(initial_pose)

    # Esperar hasta que el sistema de navegación esté completamente activo
    nav.waitUntilNav2Active()

    # Finalizar la comunicación con ROS 2
    rclpy.shutdown()

# Ejecutar la función principal si el script se ejecuta como el programa
↪  principal
if __name__ == "__main__":
    main()
```

Este código inicializa ROS2, configura la API de navegación y establece una posición inicial y orientación incial para el robot [1].

Para poder ejecutarlo antes tienes que lanzar igual que antes la simulación y el stack de Nav2 con RVIZ con estos comandos:

```
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

En una terminal diferente, activa el stack de Navegación, suministrando el mapa como un argumento:

```
ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True
↪  map:=path/to/my_map.yaml
```

A la hora de compilar el nuevo script añade la dependencia nueva al `package.xml`.

```
<depend>nav2_simple_commander</depend>
```

[1] Si el paquete `tf_transformations` no es reconocido, instalar utilizando el comando: `sudo apt install ros-humble-tf-transformations` y `sudo apt install python3-transforms3d`.

Y crea el punto de ejecucción en `setup.py`.

Ahora tienes una forma programática de establecer la posición inicial de tu robot utilizando ROS2 y Python. Comprueba que en RVIZ se inicializa la posición inicial y se podría empezar a navegar. Si te fijas el mapa no se ha inicializado bien la posición y orientación inicial y el mapa sale desplazado. Para que funcione bien cambia en el script anterior la posición teniendo en cuanta que un cuadrado es un metro, que la orientación está bien y que el sistema de coordenadas está en el centro siendo rojo el eje X, verde el eje y y azul el eje z. Comprueba que al poner bien la posición el mapa sale bien y ya se podría empezar a navegar.

A continuación expandiremos este código para incluir más funcionalidades como enviar objetivos de navegación o interactuar con otros nodos en tu sistema ROS2.

3.2 Enviar objetivo de navegación

Antes de enviar una pose de navegación, es esencial establecer una pose inicial para el robot. Una vez hecho esto, esperamos a que la navegación esté lista para recibir comandos.

Dado que ya hemos configurado la pose inicial y esperado a que la navegación esté lista, podemos proceder a enviar la pose de navegación.

```
# Código anterior para inicializar la pose

# Definimos y configuramos la pose objetivo.
goal_pose = PoseStamped()
goal_pose.header.frame_id = "map"
goal_pose.header.stamp = nav.get_clock().now().to_msg()
goal_pose.pose.position.x = 3.5
goal_pose.pose.position.y = 1.0
goal_pose.pose.position.z = 0.0

# Configuramos la orientación del robot utilizando cuaterniones.
yaw = 1.57  # 90 grados en radianes
qx, qy, qz, qw = tf_transformations.quaternion_from_euler(0,0,yaw)
goal_pose.pose.orientation.x = qx
goal_pose.pose.orientation.y = qy
goal_pose.pose.orientation.z = qz
goal_pose.pose.orientation.w = qw

# Enviamos el comando para que el robot se dirija a la pose objetivo.
nav.goToPose(goal_pose)

# Monitoreamos el progreso del robot hacia la pose objetivo y mostramos
↪  actualizaciones.
while not nav.isTaskComplete():
    feedback = nav.getFeedback()
    print(feedback)

# Mostramos el resultado final de la navegación.
result = nav.getResult()
```

```
    print(result)

    # Finalizamos el nodo ROS2.
    rclpy.shutdown()


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

Con el código anterior, hemos configurado y enviado una pose a la que el robot debe navegar. Es importante recordar configurar la pose inicial solo una vez, a menos que se requiera corregirla. También se puede obtener retroalimentación mientras el robot está navegando y un resultado al final de la tarea, lo cual es útil para saber si la navegación fue exitosa o si se abortó.

3.3 Enviar Waypoints

En esta sección, explicaremos cómo enviar una serie de waypoints a un robot en ROS2 utilizando Python. ROS2 ofrece una API sencilla para establecer la posición inicial del robot y enviarlo a un objetivo de navegación. Sin embargo, en lugar de enviar un solo objetivo, también podemos enviar una serie de waypoints, lo que le indica al robot que navegue de un punto a otro en secuencia.

Primero, refactorizamos el código para facilitar la definición de waypoints. Creamos una función que devuelve un `pose_stamped` dado una posición y orientación. Esta función facilitará la creación de múltiples waypoints sin tener que duplicar el código.

```
def create_pose_stamped(navigator, position_x, position_y, rotation_z):
    q_x, q_y, q_z, q_w = tf_transformations.quaternion_from_euler(0.0, 0.0,
    ↪  rotation_z)
    goal_pose = PoseStamped()
    goal_pose.header.frame_id = 'map'
    goal_pose.header.stamp = navigator.get_clock().now().to_msg()
    goal_pose.pose.position.x = position_x
    goal_pose.pose.position.y = position_y
    goal_pose.pose.position.z = 0.0
    goal_pose.pose.orientation.x = q_x
    goal_pose.pose.orientation.y = q_y
    goal_pose.pose.orientation.z = q_z
    goal_pose.pose.orientation.w = q_w
    return goal_pose
```

Una vez que tengamos la función anterior, podemos crear fácilmente múltiples waypoints y enviarlos al robot.

```
#!/usr/bin/env python3
import rclpy
from nav2_simple_commander.robot_navigator import BasicNavigator
from geometry_msgs.msg import PoseStamped
```

```python
import tf_transformations

def create_pose_stamped(navigator, position_x, position_y, rotation_z):
    #Código anterior

def main():
    # Inicializar ROS2 y la API BasicNavigator
    rclpy.init()
    nav = BasicNavigator()

    # Establecer la posición inicial del robot
    # Si la posición inicial ya está establecida, comentar la siguiente línea
    initial_pose = create_pose_stamped(nav, -2.5, -0.5, 0.0)
    nav.setInitialPose(initial_pose)

    # Esperar hasta que Nav2 esté activo
    nav.waitUntilNav2Active()

    # Definir metas para la navegación
    goal_pose1 = create_pose_stamped(nav, 1.5, 0.5, 1.57)
    goal_pose2 = create_pose_stamped(nav, 0.0, 1.5, 3.14)
    goal_pose3 = create_pose_stamped(nav, -1.5, 0.5, 0.0)

    # Navegar hacia la primera meta
    nav.goToPose(goal_pose1)
    while not nav.isTaskComplete():
        feedback = nav.getFeedback()
        # Descomentar la siguiente línea para imprimir feedback
        # print(feedback)

    # Seguir una serie de waypoints
    waypoints = [goal_pose1, goal_pose2, goal_pose3]
    for _ in range(3):
        nav.followWaypoints(waypoints)

        while not nav.isTaskComplete():
            feedback = nav.getFeedback()
            # Descomentar la siguiente línea para imprimir feedback
            # print(feedback)

    # Mostrar el resultado de la navegación
    print(nav.getResult())

    # Finalizar comunicaciones de ROS2
    rclpy.shutdown()

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

Con la ayuda de la función `create_pose_stamped`, es fácil definir y enviar múltiples waypoints al robot en ROS2 usando Python. Este enfoque no solo simplifica el código, sino que también

permite al robot navegar a través de múltiples puntos de manera secuencial. Compila y prueba que el robot navega de forma correcta.

4 Actividad de Navegación Autónoma

Esta actividad está diseñada para evaluar la capacidad de su robot para navegar de manera autónoma en un entorno simulado. Usted guiará al robot a través de una serie de puntos predeterminados, observando su habilidad para planificar rutas y evitar obstáculos dinámicos.

4.1 Navegación a Puntos de Control

Su tarea es programar el robot para que visite una serie de posiciones, cada una con una orientación específica dentro del entorno simulado, como se muestra en la Figura 4.1. Puede determinar el orden de las posiciones según prefiera, pero asegúrese de que el robot visite todas ellas.

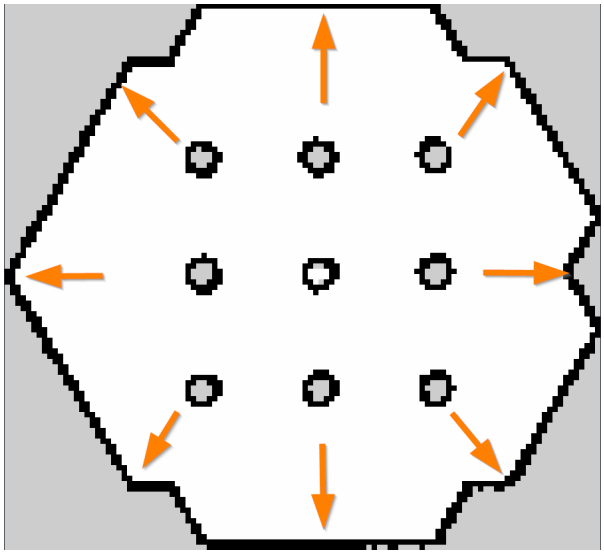

Figura 4.1: Posiciones y orientaciones específicas que el robot debe visitar en el entorno de simulación.

4.2 Prueba de Obstáculos Dinámicos

Una vez que la navegación básica esté funcionando, introduzca obstáculos en el entorno de Gazebo para desafiar al robot con situaciones no mapeadas previamente. Observe y documente cómo el robot utiliza su planificador local para detectar y evitar estos nuevos obstáculos, asegurando que aún puede alcanzar todos los puntos de control sin colisiones.

4.2.1 Instrucciones Detalladas

1. Inicie el entorno de simulación en Gazebo y cargue el mapa predefinido para la actividad.

2. Utilice el código de navegación que ha implementado para dirigir al robot a las posiciones indicadas en la Figura 4.1.

3. Añada obstáculos físicos en el entorno de simulación de Gazebo en lugares aleatorios que el robot tendría que sortear.

4. Inicie la navegación autónoma y observe el comportamiento del robot. Fíjate especialmente en las situaciones en la que el robot maneje de manera efectiva los obstáculos dinámicos.

Bibliografía

Macenski, Steve et al. (2020). «The marathon 2: A navigation system». En: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, págs. 2718-2725 (vid. pág. 7).

Web de los tutoriales de ROS (2023). Accessed: 2023-09-01.

Web de Nav2 (2023). Accessed: 2023-10-30.

Web de ROS2 Humble (2023). Accessed: 2023-09-01.