Universidad Zaragoza
1542

Trabajo Fin de Grado

Navegación de robots manipuladores en el entorno de ROS
Navigation of mobile manipulators in the ROS environment

Autor/es

Jorge Playán Garai

Director/es

Gonzalo López
Rosario Aragües

Escuela de ingeniería y arquitectura

2019

Resumen

La robótica se ha extendido en las últimas décadas en entornos industriales, y en la actualidad genera grandes expectativas en multitud de campos. En este Trabajo de Fin de Grado se pretende que el alumno se familiarice con la interacción entre "Robot Operating System" (ROS) y dos robots diferentes (uno comercial, Turtlebot, y otro en desarrollo, Campero). Los objetivos incluyen comprender la herramienta ROS aplicada a la navegación de robots, desarrollar capacidades para la instalación del software ROS en un ordenador con el sistema operativo Ubuntu, aplicar el software a los robots Turtlebot y Campero realizando simulaciones de navegación y analizar críticamente los resultados obtenidos. De esta manera, se puede usar la experiencia ganada en Turtlebot para manejar Campero y hacer algunas contribuciones al mismo.

Se estudió el software del robot Turtlebot Waffle Pi, un pequeño robot con cuatro ruedas, dos de ellas motrices, que incluye un paquete de navegación. Se realizaron simulaciones de mapeo de entorno y de navegación autónoma sorteando obstáculos. Los archivos de navegación de Turtlebot se modificaron y aplicaron a la navegación del robot Campero. Este robot es mucho más grande que Turtlebot, dispone de cuatro ruedas y un brazo robótico. Se realizaron simulaciones de navegación en dos entornos desarrollados para Campero, usando los dos tipos de ruedas que incluye el robot: diferenciales y omnidireccionales. Se usó la rutina SLAM para realizar mapas del entorno y después poder emplearlos para una navegación más eficiente.

El Software de Turtlebot para ROS está disponible como código abierto, y ha permitido realizar simulaciones complejas esquivando obstáculos y actualizando la ruta en tiempo real. A pesar del rápido progreso actual del software para Campero, este robot se halla en una fase inicial de su desarrollo. Puesto que Campero no tiene rutinas propias de navegación, se aplicaron las usadas en Turtlebot realizando las modificaciones y empleando los parámetros propios de Campero. Esto permitió hacer una variedad de experimentos de simulación de navegación en los que se alcanzaron los puntos propuestos.

Se han encontrado problemas con la navegación de Campero, que parecen relacionados con su parametrización, con el uso de sus sensores y con la especificación del objetivo de la navegación. Estos son aspectos que deben de recibir atención en el futuro para poder hacer de Campero un robot fiable y operativo en entornos reales.

Índice

1. Introducción 4
1.1. Contexto y Motivación 4
1.2. Objetivos 5
1.3. Alcance y entorno de trabajo 5
2. ROS, Robot Operating System 8
2.1. Introducción a ROS 8
2.2. Instalación de Linux y ROS 10
2.2.1. Instalación del sistema operativo Linux 10
2.2.2. Instalación de ROS 11
2.2.3. Instalación de Turtlebot 3 11
2.3. Conceptos básicos de ROS 11
2.4. Herramientas incluidas en ROS 13
2.5. Un problema de navegación para resolver con ROS 13
2.6. Navegación del robot Turtlebot 13
3. Archivos de navegación 21
3.1. Adaptative Monte-Carlo Localization (AMCL) 21
3.2. Move_Base 23
3.3. Global launch file 25
4. Robot Campero 27
4.1. Versión del software 28
4.2. Archivos proporcionados 28
4.3. Instalación y ejecución 30
5. Navegación sobre el robot Campero 34
5.1. Preparación Inicial 34
5.1.1. Modelo inicial 34

5.1.2. Versión con navegación 35
5.2. Archivos finales 35
5.2.1. AMCL 35
5.2.2. Move_base 36
5.2.3. One_robot 36
5.2.4. Global 37
6. Experimentación con el robot Campero 39
6.1. Reconocimiento del entorno 39
6.2. Navegación en exterior con ruedas diferenciales 41
6.3. Navegación en interior con ruedas omnidireccionales 45
7. Discusión general 49
7.1. Simulación con el robot Turtlebot 49
7.2. Simulación con el robot Campero 49
8. Conclusiones 51
9. Recomendaciones para trabajos futuros con el robot Campero 52
10. Anexos 53
10.1. Archivo de especificaciones del robot Waffle Pi por la empresa Robotis 53
10.2. Archivo de código `amcl.launch` para el robot Campero 55
10.3. Archivo de código `move_base.launch` para el robot Campero 58
10.4. Archivo de código `campero_one_robot_nav.launch` para el robot Campero 61
10.5. Archivo de código `campero_nav.launch` para el robot Campero 63
10.6. Archivo de mapa del entorno exterior `mymap.pgm` 66
10.7. Archivo de mapa del entorno interior mapaInt.pgm 67
11. Bibliografía 68

1. Introducción

1.1. Contexto y Motivación

El intenso desarrollo actual de la robótica está basado en el progreso de áreas de conocimiento complementarias como la informática, la mecánica, la electrónica, los sensores o la inteligencia artificial. La popularización de la robótica en los entornos industriales e incluso en la vida diaria está generando grandes expectativas en la sociedad. Los ingenieros electrónicos y automáticos (junto a otros profesionales) tienen una responsabilidad directa sobre el desarrollo de la robótica.

En España son muchos los ámbitos en los que la robótica puede en las próximas décadas ayudar a que la sociedad sea sostenible. Un país muy envejecido, con la población concentrándose en las ciudades y con una baja densidad de población en la mayoría del territorio presenta retos muy importantes para la robótica. Entre ellos, la fabricación, la asistencia a personas dependientes, la seguridad, las ciudades inteligentes, el cuidado del medio ambiente o la agricultura.

En este contexto, la robótica es un campo de trabajo de clara importancia tanto en el presente como en el futuro. Este Trabajo Fin de Grado (TFG) trata de un aspecto horizontal de la robótica: el software que gobierna los distintos aspectos de un robot. El software usado es ROS (Open Source Robotics Foundation, 2019a), una plataforma de código abierto, genérica (cualquier robot puede ser diseñado para usarlo) y de desarrollo cooperativo. Esta plataforma de software se aplica a aspectos como el mapeado del entorno del robot, el uso de mapas y sensores para la navegación terrestre, la manipulación con brazos robóticos o la interacción con el robot físico.

En este TFG el software se aplica al robot comercial Turtlebot waffle Pi (Robotis, 2019) y al robot Campero, que está actualmente en sus fases iniciales de desarrollo por la empresa Robotnik. De todas las capacidades del software ROS, el TFG se orienta a la navegación de los robots. Esta aplicación es muy interesante para comprender cómo se programa esta navegación en dos robots diferentes y en dos momentos muy diferentes de su desarrollo.

El TFG se enmarca dentro de las actividades del proyecto COMMANDIA, Robótica móvil colaborativa de objetos deformables en aplicaciones industriales (COMMANDIA, 2019). Este proyecto está cofinanciado por el Programa Interreg Sudoe y por el Fondo

Europeo de Desarrollo Regional (FEDER). El proyecto está orientado a la manipulación y el procesamiento de objetos deformables tales como alimentos, ropa, juguetes o artículos de cuero. El objetivo es "mejorar la competitividad y las condiciones de trabajo de las industrias donde los objetos deformables deben ser manipulados directamente por los operadores humanos para controlar su forma durante la producción". El proyecto cuenta con la participación de socios de Francia, Portugal y España, entre los que se cuenta la Universidad de Zaragoza.

1.2. Objetivos

El objetivo docente de este TFG es comprender la interacción entre ROS y dos robots diferentes, analizando sus posibilidades y sus limitaciones actuales. Este objetivo docente se concreta en cuatro objetivos específicos:

1. Comprender la herramienta ROS, particularmente su aplicación a la navegación autónoma de robots.
2. Desarrollar las capacidades para la instalación y puesta a punto del software ROS en un ordenador, y para su aplicación a los robots Turtlebot y Campero mediante el uso del software específico de los robots.
3. Contribuir al desarrollo de software para la navegación del robot Campero.
4. Probar y analizar la navegación de los dos robots en los aspectos de mapeo del territorio y de simulación de la navegación a tiempo real.

1.3. Alcance y entorno de trabajo

Al tratarse ROS de una plataforma de software genérica, el alcance de este TFG es global: cualquier robot puede ser diseñado para utilizar ROS. En este documento se analizan dos robots en particular, pero el software es adaptable a otros robots.

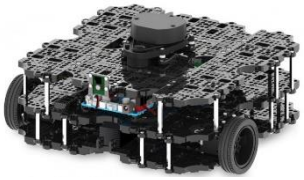
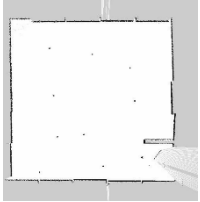
Es preciso recordar en este punto que el alcance del TFG no incluye la manipulación de los robots a nivel físico, ni su control por ROS. Se han utilizado modelos informáticos y simulaciones de los dos robots, que han sido controlados por ROS. En una fase posterior a este trabajo, se podría analizar el resultado del control del robot físico por ROS.

El uso de un modelo informático de los robots permite acelerar el desarrollo de nuevas utilidades de control o la explotación de nuevos sensores en los robots. Las simulaciones

realizadas con ROS permiten identificar problemas de diseño o de compatibilidad de datos.

Finalmente, los elementos utilizados en este trabajo fin de grado se especifican a continuación (Tabla 1.1).

Tabla 1.1 Listado de los elementos del trabajo con una descripción e ilustración.

NOMBRE	DESCRIPCIÓN	ILUSTRACIÓN
ROS	Sistema operativo gratuito de código libre que proporciona herramientas relacionadas con robótica.	
CAMPERO	Robot de gran envergadura sobre el que se trabaja la navegación.	
TURTLEBOT	Robot de código libre con un sistema de navegación integrado.	
SENSORES	Elementos del robot para recibir información del entorno. Varían con el robot empleado.	
ENTORNO GAZEBO	Escenario dónde se realizan las simulaciones del robot.	
MAPA RVIZ	Mapa del escenario para ayudar con el guiado del robot durante la navegación.	

Se añade un diagrama de alto que muestra el funcionamiento conjunto de estos elementos (Figura 1.1).

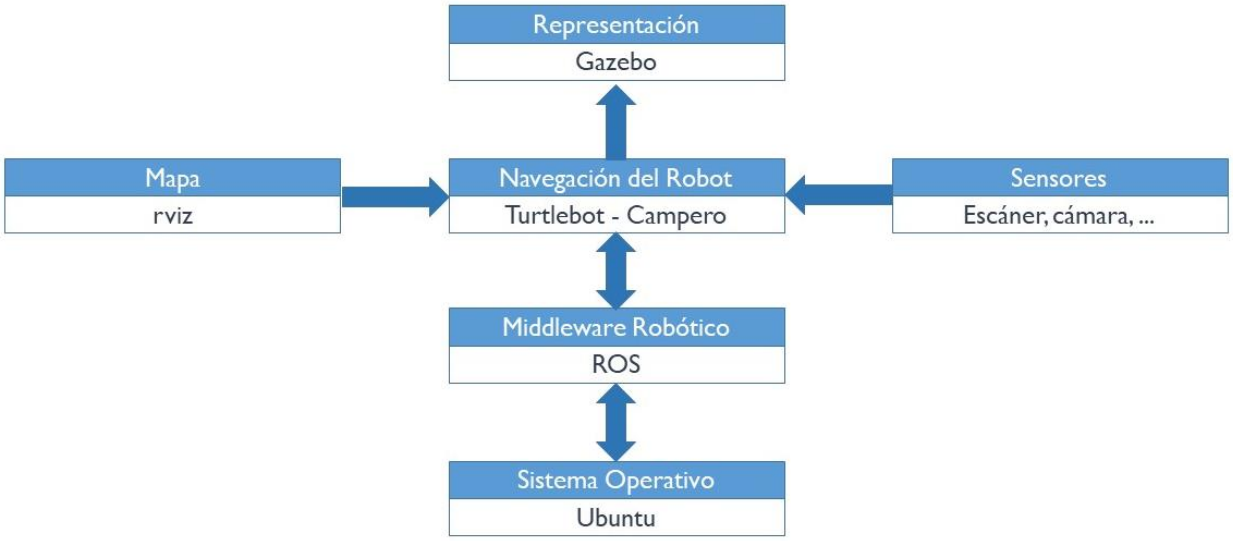


Figura 1.1. Diagrama de alto nivel que conecta a nivel lógico los elementos que componen este trabajo.

2. ROS, Robot Operating System

2.1. Introducción a ROS

Robot Operating System (ROS) es un sistema operativo gratuito de código libre[1] que proporciona funciones flexibles para el manejo y control de robots como la abstracción de hardware, control de dispositivos, comunicación entre procesos y manejo de paquetes, entre otros. Los desarrolladores de ROS lo definen como "una colección de herramientas, librerías y convenciones que simplifican la tarea de crear comportamientos robóticos complejos y robustos en una variedad de plataformas robóticas" (Open Source Robotics Foundation, 2019a). ROS incluye herramientas y librerías para obtener, construir, escribir y ejecutar código en uno o varios ordenadores simultáneamente. Desde el punto de vista informático, ROS es un middleware (Wikipedia contributors, 2019a), situándose entre el sistema operativo y las aplicaciones que ejecuta. Por lo tanto, facilita y acelera el trabajo de los desarrolladores. En definitiva, ROS no es un sistema operativo en sentido estricto, sino que se ejecuta sobre uno: Linux.

Uno de los aspectos más importantes de ROS es su filosofía de "desarrollo colaborativo de software robótico" (Open Source Robotics Foundation, 2019a). El objetivo es facilitar que laboratorios especializados en distintos aspectos de la robótica puedan colaborar y apoyarse entre ellos para construir una herramienta de gran valor añadido. De esta manera, la web de ROS actualmente cita a 79 desarrolladores de distintas organizaciones. A pesar de este esfuerzo colaborativo, ROS tiene dos autores principales, Eric Berger y Keenan Wyrobek, que en torno a 2007 comenzaron a trabajar en el proyecto mientras hacían la tesis doctoral en la Universidad de Stanford (Wyrobek, 2017). En este artículo se presenta un comic (Figura 2.1) en el que se explica la necesidad de ROS, como una forma de evitar la pérdida de tiempo de los desarrolladores.

[1] ROS se distribuye bajo una licencia del tipo BSD, que se caracteriza por imponer restricciones mínimas en su uso y distribución.

Figura 2.1. Comic explicando la necesidad de ROS o la pérdida de tiempo reinventando la rueda en la robótica. Autoría: (Wyrobek, 2017).

El comic muestra como alguien hace un desarrollo innovador en robótica. Otro laboratorio quiere seguir esa línea, pero no tiene los detalles que hacen falta. Finalmente, decide hacer un programa para que otros puedan usarlo. Se consigue algo que funciona, lo publica, otros lo leen y la rueda comienza de nuevo. ROS se creó para romper este círculo, evitando la pérdida de tiempo, integrando esfuerzos y asegurando una continuidad en el desarrollo. Los autores iniciales consiguieron dinero para el desarrollo, implicaron a otros profesionales y trabajaron en un garaje. Diez años después, muchos jóvenes se han formado en el programa y muchas empresas han comenzado a usarlo

para sus nuevos productos y negocios. Al mismo tiempo, la *Open Source Robotics Foundation* se hizo cargo del desarrollo de ROS en 2013 (Wyrobek, 2017).

En 2012 comenzó el desarrollo de ROS-Industrial, otro proyecto de código abierto con licencia BSD que tiene el objetivo de ampliar ROS al mundo de la fabricación automática y robótica (Wikipedia Contributors, 2019b). Esta rama tiene tres nodos activos: uno en Texas (EE.UU.), otro en Alemania (Europa) y el último en Singapur (Asia). Se ocupa de la interacción entre la robótica y los equipos de fabricación, con aplicaciones industriales y actividades formativas.

A lo largo de estos doce años de desarrollo, se han ido creando en ROS capacidades para un buen número de aplicaciones. Algunas de las muchas capacidades actualmente disponibles en ROS son:

- Detectar las características del medio mediante el uso de sensores. Se puede usar una variedad de ellos, siendo típicos los sensores LIDAR[2] y las cámaras de visión óptica.
- Navegar en un espacio que puede estar registrado de antemano en un mapa. No es obligatorio que exista este mapa, pero ayuda con la navegación. El Propio ROS puede fabricar estos mapas usando los sensores del robot, y algoritmos SLAM[3].
- Gestionar entornos multi-robot. De esta manera, una flota de robots puede colaborar para hacer una tarea conjuntamente.
- Manejar brazos robóticos para tareas de manipulación o fabricación.

2.2. Instalación de Linux y ROS

Actualmente, ROS únicamente está disponible, de forma estable, en Linux. ROS está disponible de forma experimental en MacOS y Windows 10. Este trabajo se ha realizado sobre Linux debido a su estabilidad.

2.2.1. Instalación del sistema operativo Linux

Para la realización de este trabajo se ha empleado el sistema operativo Linux, en particular la distribución Ubuntu 16.04.6 LTS Xenial Xerus. En esta instalación se comenzó por la descarga de la versión siguiendo las instrucciones de la página oficial (Ubuntu releases, 2019). Dependiendo del ordenador que se usará se necesitará la

[2] Light Detection and Ranging o Laser Imaging Detection and Ranging.
[3] Simultaneous localization and mapping.

versión de 32 bits o la de 64 bits. Ya con la descarga completada se procede a la instalación del software Rufus (Batard, 2019), que permite colocar esta imagen del sistema en un dispositivo USB para instalarlo desde éste. Una vez la imagen del sistema está lista, es necesario hacer una partición del sistema para tener espacio sobre el que alojar Ubuntu.

Cuando todo está preparado, se reinicia el ordenador sobre el que queramos realizar la instalación para acceder a la BIOS y ejecutar el USB. Se siguen los pasos expuestos en la pantalla para finalizar la instalación.

2.2.2. Instalación de ROS

Una vez tenemos Ubuntu 16.04.6 Xenial Xerus instalado procedemos a la instalación de ROS. En este TFG se ha empleado la versión de ROS "Kinetic Kame", la cual esta específicamente diseñada para la versión de Ubuntu 16.04.6 LTS Xenial Xerus. Para ello se sigue el tutorial que se presenta en la propia página de ROS (Open Source Robotics Foundation, 2019d).

2.2.3. Instalación de Turtlebot 3

Una vez instalados Ubuntu y ROS, pasamos a la instalación del robot Turtlebot y todos los paquetes necesarios para su funcionamiento. Estas instrucciones están disponibles en la página web del robot Turtlebot (Robotis, 2019b).

2.3. Conceptos básicos de ROS

El funcionamiento de ROS consiste en un modelo gráfico de procesos. Los procesos generados son llamados nodos, como los nodos de un diagrama, y están conectados por tópicos. Estos se representan como líneas que conectan nodos. Los tópicos permiten a los nodos pasarse mensajes con información. Todos los nodos dependen de un maestro que los registra y ordena, `roscore`. Este maestro establece comunicación "peer-to-peer" entre los nodos, por lo cual los mensajes o llamadas entre nodos no necesitan pasar por el maestro.

A continuación, se proporcionan definiciones más completas de los elementos que componen ROS.

- **Nodos**: Los nodos son el centro de la programación en ROS, ya que la mayoría del código ejecuta nodos, comunicación entre nodos u otras acciones relacionadas con nodos.

- **Tópicos**: Para poder enviar mensajes a un tópico, primero el nodo debe publicar en dicho tópico y si queremos que un nodo reciba la información enviada deberá subscribirse al tópico. Ningún nodo puede saber quién está publicando en qué tópico ni quién lo está recibiendo: es completamente anónimo.
- **Servicios**: Un servicio es una acción que un nodo puede ejecutar con un solo resultado. Los nodos publican sus servicios y otros nodos pueden requerirlos.

En los siguientes párrafos se presentan ejemplos de cómo establecer un entorno de trabajo y cómo crear sobre éste un paquete.

El entorno más simple sobre el que se puede trabajar consiste en crear un paquete con su carpeta *source* (*src*). Esto se hace de la siguiente forma:

```
mkdir -p ~/catkin_ws/src

cd ~/catkin_ws

catkin_make
```

Al hacer este último paso habremos compilado el paquete y se deberían haber creado las carpetas `devel` y `build` junto a la `src` previamente mencionada.

Se finaliza la creación del paquete de la siguiente forma:

```
source devel/setup.bash
```

Esto permite utilizar el espacio de trabajo como nuestro entorno actual.

Ahora procedemos a la creación de un paquete en el interior de este espacio. La forma más sencilla sería:

```
catkin_create_pkg <nombre_paquete> [dependencia1]
  [dependencia2]
```

Por ejemplo, podría ser así:

```
catkin_create_pkg paquete_prueba std_msgs rospy roscpp
```

Esto crea un paquete llamado `paquete_prueba` y dice que este paquete va a depender de *std_msgs, rospy, y roscpp* (librerías incluidas en ROS para comunicación, código Python y código c++)

Una vez creados, volvemos a compilar con:

```
cd ~/catkin_ws/
```

```
catkin_make

source devel/setup.bash
```

2.4. Herramientas incluidas en ROS

La funcionalidad de ROS se ve altamente potenciada por la inclusión de herramientas que ayudan a los desarrolladores a visualizar y guardar información, navegar la estructura de paquetes de ROS y crear "scripts" para automatizar o configurar procesos. Entre ellas cabe destacar:

- **`rviz`** (Open Source Robotics Foundation, 2019e): Visualizador 3D que forma parte de ROS y que se usa para representar robots, los escenarios en los que trabajan y los datos de los sensores. Este programa es altamente configurable con diferentes formas de visualizar y extensiones.
- **`Gazebo`** (Open Soure Robotics Foundation, 2019f): Aunque no está incluido en la instalación, sí que se incluyen paquetes para poder ejecutarlo junto con ROS y poder emplearlo para obtener simulaciones de nuestros robots y de escenarios en 3D.
- **`Roslaunch`**: Es una herramienta clave en ROS, que ejecuta varios nodos de forma local o remota y les proporciona parámetros para ejecutarse correctamente.

2.5. Un problema de navegación para resolver con ROS

Tal como se ha visto antes, ROS puede resolver una gran cantidad de problemas generales sobre el uso de robots. Estos problemas van desde el análisis del medio que rodea al robot mediante sensores, el movimiento del robot, el uso de un brazo robótico o la fabricación en entornos industriales. Sin embargo, este Trabajo Fin de Grado se refiere a aplicaciones de navegación de un solo robot con ruedas en entornos mapeados apoyado por sensores que caracterizan el medio. Se quiere que el robot sea capaz de navegar de un punto inicial a un punto destino sorteando los obstáculos presentes. En este trabajo, ROS resuelve este problema en una simulación en el ordenador, que podría posteriormente trasladarse a un robot físico que navega en un entorno físico.

2.6. Navegación del robot Turtlebot

En estas simulaciones vamos a trabajar con el robot comercial Turtlebot Waffle Pi (Robotis, 2019) (Anexo 10.1). Como se aprecia en el anexo, este robot es ligero y tiene reducidas dimensiones (1,8 kg, 281 x 306 x 141 mm). Además, viene con paquetes de

navegación (entre otras funcionalidades) que lo hacen muy adecuado para este trabajo. El robot se muestra en la figura 2.2.

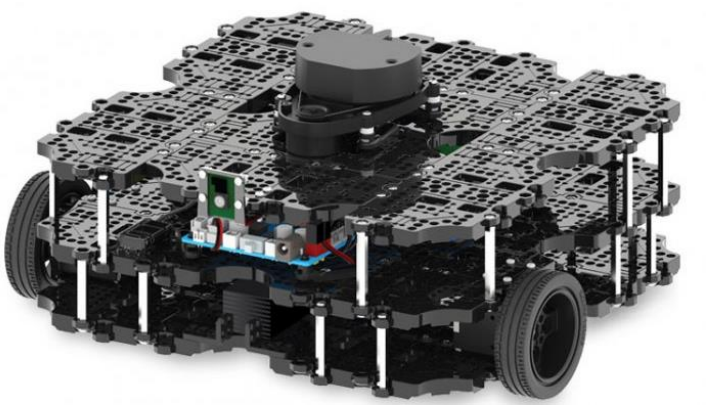


Figura 2.2. El robot Turtlebot3 Waffle Pi. Autoría: (Robotis, 2019).

La instalación de Ubuntu con ROS y con todos los paquetes necesarios deja el sistema preparado para el uso del robot Turtlebot. Los paquetes disponibles incluyen utilidades de navegación, entorno multirobot, gmapping, y SLAM, entre otros. Para poder comprender cómo funciona la navegación ya incluida en este robot y poder aplicarla posteriormente al robot Campero, es necesario examinar el funcionamiento en el Turtlebot y analizar los archivos que lo forman.

Los archivos del Turtlebot están diseñados de tal forma que ROS comprueba con qué versión de éste está trabajando (Burger, Waffle o Waffle Pi). En función de la versión se aplican unas físicas y unos parámetros diferentes.

Una vez instalado el robot Turtlebot y su paquete de navegación, se pasa a comprobar el funcionamiento de éste. Para esto se abre en una terminal un maestro:

```
roscore
```

Se abre entonces otra terminal sobre la que se carga el mapa del entorno. En principio debería ejecutarlo el archivo de navegación del Turtlebot, pero en las condiciones de trabajo descritas provoca errores y no consigue cargarlo bien. Es preciso pues ejecutarlo con antelación para un correcto funcionamiento.

```
rosrun map_server map_server
    catkin_ws/src/turtlebot3/turtlebot3_navigation/maps/map.yaml
```

Ahora podemos ejecutar el código con normalidad. Sobre otra terminal, elegimos el modelo de Turtlebot de Robotis que deseamos utilizar. En este ejemplo se emplea el waffle_pi (Robotis, 2019),

Este robot incorpora un "360 Laser Distance Sensor LDS-01" se trata de un escáner LIDAR 2D que puede medir en 360° y recoge datos alrededor del robot para usarlos para SLAM.

Procedemos a ejecutar el mundo sobre el que simular el robot elegido:

```
export TURTLEBOT3_MODEL=waffle_pi

roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Finalmente, en una última terminal, entramos en el directorio en el que se aloja el fichero a ejecutar y lo activamos. Para poder disponer de una guía del mapa en rviz es necesario completar el comando con el nombre y localización del mapa a emplear.

```
cd catkin_ws/src/turtlebot3/turtlebot3_navigation

roslaunch turtlebot3_navigation
  turtlebot3_navigation.launch map_file:=maps/map.yaml
```

Una vez completado este proceso deberíamos de tener una ventana de Gazebo y otra de rviz que deberían de tener el aspecto que se muestra en las figuras 2.3 y 2.4.

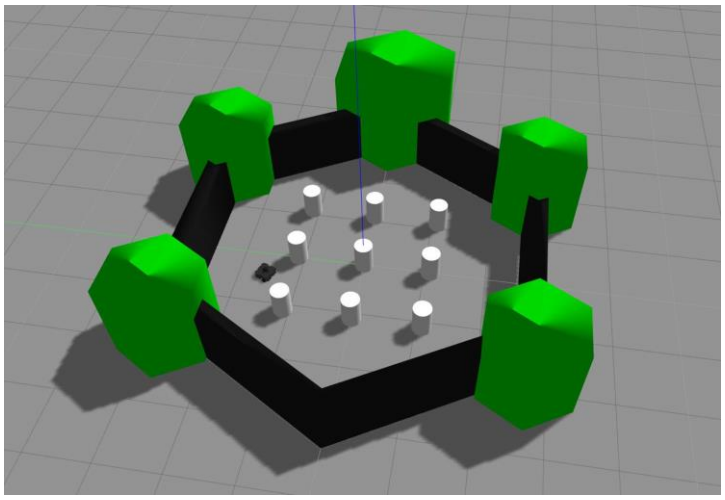


Figura 2.3. Contenido gráfico de la ventana de Gazebo para el caso de estudio del robot Turtlebot.

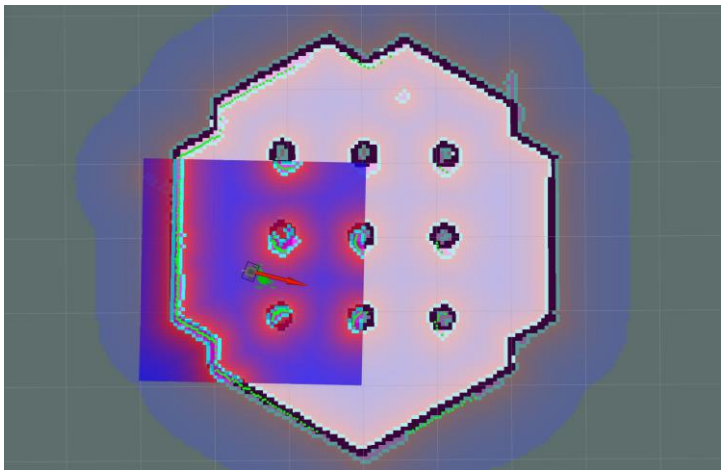


Figura 2.4. Contenido gráfico de la ventana de rviz para el caso de estudio del robot Turtlebot.

La ventana de Gazebo es únicamente una visualización en 3D de nuestro robot en el entorno simulado. En este caso, el entorno está compuesto por un recinto hexagonal con nueve obstáculos cilíndricos en su interior. El robot se representa por el pequeño rectángulo negro dentro del hexágono.

La ventana de rviz es de mucho mayor interés puesto que presenta los valores de todos los sensores del robot. En la imagen vemos el mapa de fondo del entorno. Superpuesto a éste, se ve la imagen del escáner incluido en el robot que detecta los obstáculos del entorno y el entorno en sí (las columnas y el recinto). Originalmente, al activar este proceso, en la ventana de rviz el mapa y el radar de nuestro robot no deberían de solaparse: debería de haber un desplazamiento. Esto se debe a que el origen del robot en el mapa no se corresponde con el origen del robot en Gazebo. Este desplazamiento puede modificarse manualmente. En la barra superior al mapa de rviz se pueden encontrar varios comandos útiles para la simulación, tal como se muestra en la figura 2.5:

Figura 2.5. Barra de navegación de rviz, con comandos útiles para la navegación del robot.

Los dos comandos más importantes para esta aplicación son “2D Pose Estimate” y “2D Nav Goal”. El primero de estos permite solucionar el problema en cuestión. Seleccionándolo y seleccionando posteriormente la localización aproximada de nuestro

robot (visible en Gazebo) sobre el mapa, el escáner y el mapa deberían solaparse. Este proceso se puede repetir hasta conseguir un solapamiento satisfactorio.

Una vez tenemos el robot listo y en posición podemos simular la navegación. Seleccionamos la segunda opción mencionada, "2D Nav Goal", que permite seleccionar un destino para nuestro robot. A modo de demostración seleccionamos un destino que obligue al robot a esquivar algunos pilares. Partiendo de la posición vista en las figuras 2.3 y 2.4, marcamos un destino en la esquina superior derecha (figura 2.6). Si mantenemos pulsado el ratón se puede además indicar la orientación del robot en el punto de destino. Al finalizar la selección el robot traza una primera ruta, que se puede ver sobre el mapa, y empieza a avanzar. La ruta se recalcula en cada momento de la trayectoria en función de lo que detecten los sensores. Esto puede dar lugar a cambios en el recorrido.

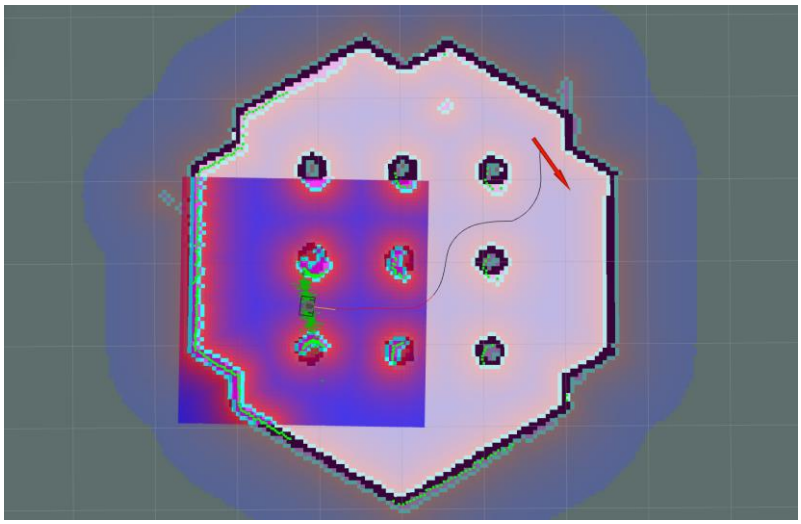


Figura 2.6. Especificación del destino de la navegación (la flecha roja indica el punto y la orientación final del robot). ROS ha calculado la primera ruta a seguir por el robot, que se muestra como una línea.

En la figura 2.7 se aprecia que, a pesar de que el recorrido inicial indicaba un movimiento entre los pilares de la fila central, el recorrido final ha preferido rodear los pilares por el exterior.

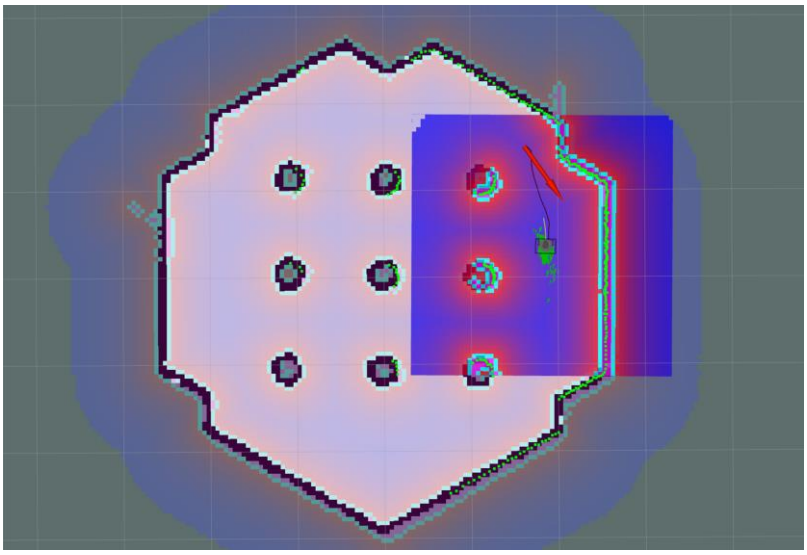


Figura 2.7. Robot llegando al punto de destino, mostrando una ruta diferente de la primera.

Rviz no es la única forma de indicar un destino para nuestro robot. Podemos indicar un destino directamente sobre el tópico correspondiente en la navegación. Para esto aplicamos el siguiente comando sobre una terminal:

```
rostopic pub /move_base_simple/goal
  geometry_msgs/PoseStamped '{header: {stamp: now,
  frame_id: "map"}, pose: {position: {x: 1.0, y: 0.5, z:
  0.0}, orientation: {w: 1.0}}}'
```

El comando "`rostopic pub (nombre del tópico) (elemento a publicar)`" nos permite publicar sobre el tópico que deseemos. En este caso, "`move_base_simple/goal`" es un tópico que indica el destino del robot. Es muy importante publicar en un formato que sea compatible con el del tópico. Este tópico acepta el destino en el formato `PoseStamped`, pero esto no ocurre para todos los tópicos, sino que cada uno requiere un formato determinado (Open Source Robotics Foundation, 2019g). Una vez indicado el formato se procede a escribir el mensaje a transmitir siguiendo la sintaxis adecuada. En nuestro caso, las partes más importantes son "`map`" que indica el tópico sobre el que está el mapa sobre el que indicaremos el destino, "`position`" y "`orientation`".

La posición en coordenadas (x, y, z) podemos obtenerla de rviz. Sobre rviz empleamos la herramienta "Publish Point" (Figura 2.5). Es posible que esta herramienta no esté visible en el menú al principio de la ejecución. Para verla necesitamos hacer click en el

símbolo "+" de la barra de menús de rviz y añadirlo desde ahí. Una vez seleccionada esta herramienta, al mover el cursor por el mapa sin necesidad de hacer click, en la esquina inferior izquierda de la ventana se pueden ver las tres coordenadas (x, y, z). Aunque la coordenada z no esté exactamente en cero, podemos mantenerla en cero para la navegación. Una vez localizado el punto al que queremos mover el robot, escribimos las coordenadas en el comando anterior. Con esto transmitimos al robot la posición. Para la orientación se usa un cuaternio:

$$\text{Cuaternio} = \{\cos\left(\frac{\theta}{2}\right), X * \sin\left(\frac{\theta}{2}\right), Y * \sin\left(\frac{\theta}{2}\right), Z * \sin\left(\frac{\theta}{2}\right)\}$$

En esta ecuación, X, Y, Z representan el eje unitario sobre el que se realiza el giro y θ el giro en sí. En el comando establecemos el primer valor a `1,` indicando que se trata de un giro de cero radianes (No presenta giro).

Para comprender el funcionamiento del paquete de navegación, se aplica una herramienta proporcionada por ROS que permite observar la dinámica interna. Mientras el algoritmo de navegación está en simulación, la herramienta ejecuta un comando que permite ver las comunicaciones entre nodos, quién está publicando y quién está suscrito a qué tópicos.

Esta herramienta consiste en el comando `rqtgraph`, que devuelve un diagrama con los tópicos y nodos actualmente en ejecución (figura 2.8).

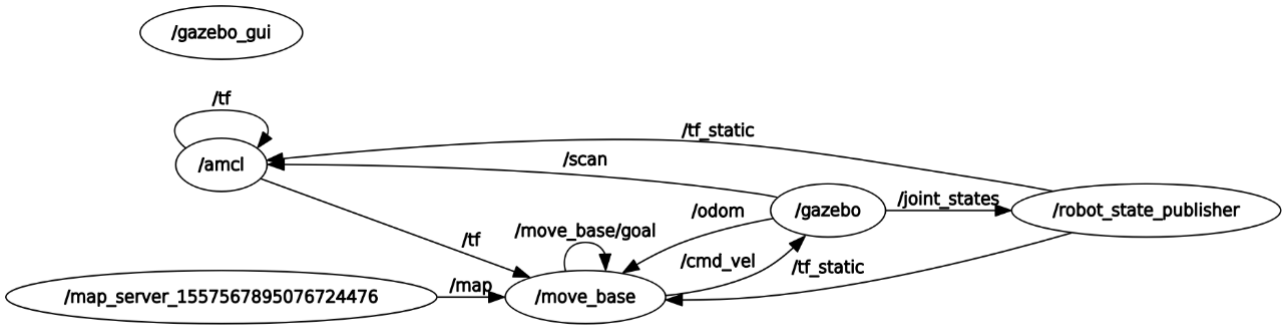


Figura 2.8. Diagrama del módulo de navegación Turtlebot generado con `rqtgraph`. Los nodos están indicados por óvalos, mientras que los tópicos están escritos sobre las flechas. Autoría: (Robotis, 2019) Robotis (2019).

Sobre este diagrama se observa que el nodo `move_base` es el principal encargado de la navegación, ya que es la piedra angular del diagrama que recibe todos los datos y los

procesa y como salida envía la velocidad necesaria. También se aprecia que está utilizando el nodo AMCL como localización, junto con la ejecución del mapa por map_server y la ejecución virtual del robot (nodo gazebo).

3. Archivos de navegación

Procedemos a examinar los archivos incluidos en la navegación de ROS y a comprender su funcionamiento. Estos archivos son usados por `roslaunch` para poder ejecutar los nodos necesarios, que están escritos en formato `.launch`. Los archivos que componen la navegación de un robot en ROS son: Adaptative Monte-Carlo Localization (AMCL), Move-Base y el archivo global de ejecución.

3.1. Adaptative Monte-Carlo Localization (AMCL)

AMCL es un filtro de partículas que permite a los robots localizarse en un entorno. Consiste en encajar las lecturas de los escáneres del robot con su posición en el mapa, en un proceso paralelo a la localización por odometría. Esto se ve claramente en la figura 3.1.

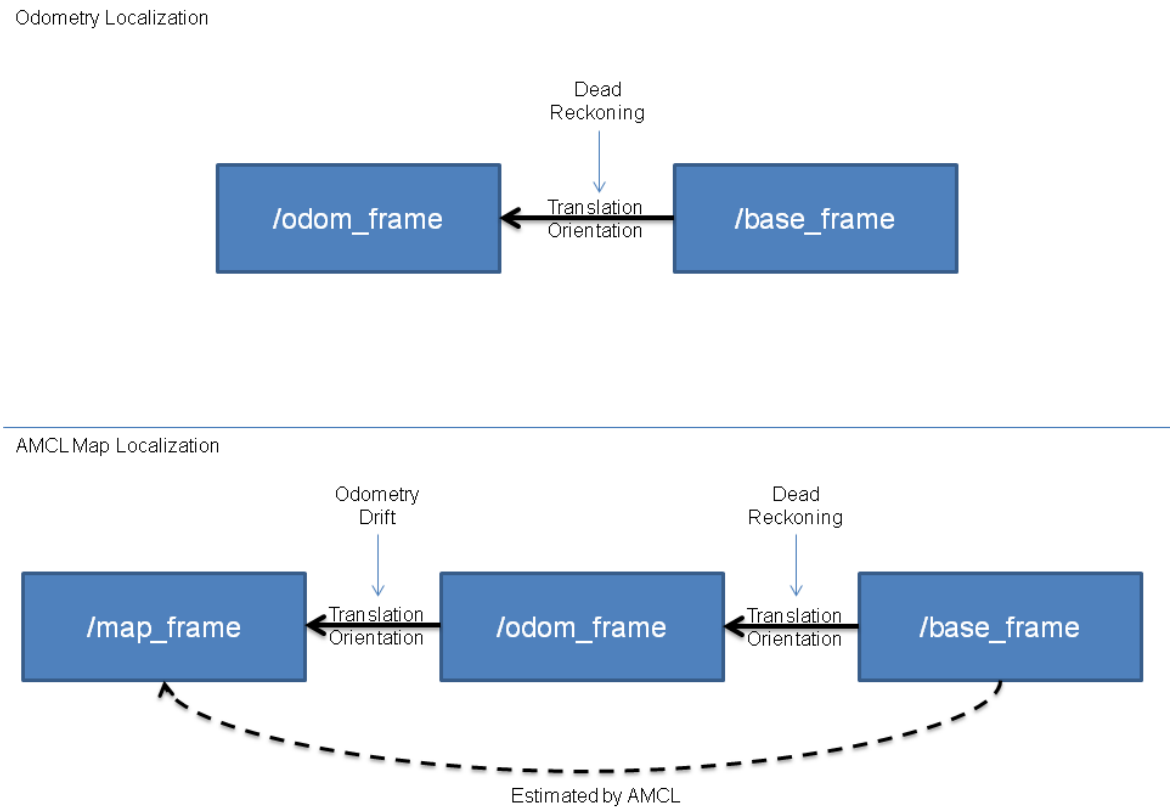


Figura 3.1. Imagen que relaciona ambas formas de localización (AMCL y odometría). Autoría: ROS.

Desde un nivel técnico, AMCL se encarga de proporcionar unas posiciones estimadas que pueden verse afectadas con el tiempo por ruido en el sistema de odometría. Tras el movimiento del robot las posibles variaciones de posición se ven amplificadas, por lo

que el algoritmo compara las posiciones con el cambio en la medición del sensor y busca cuales se ajustan más. Esto mejora la localización del robot y consigue una mayor precisión a lo largo del tiempo.

En ROS ya viene implementado un nodo que, configurándolo adecuadamente, permite aplicar AMCL a nuestro robot. El nodo que viene instalado en el Turtlebot se presenta en la Figura 3.1, en la que se reproduce código del archivo `amcl.launch`.

```
<launch>
    <!-- Arguments -->
    <arg name="scan_topic"     default="scan"/>
    <arg name="initial_pose_x" default="0.0"/>
    <arg name="initial_pose_y" default="0.0"/>
    <arg name="initial_pose_a" default="0.0"/>
    <!-- AMCL -->
    <node pkg="amcl" type="amcl" name="amcl">
        <param name="min_particles"             value="500"/>
        <param name="max_particles"             value="3000"/>
        <param name="kld_err"                   value="0.02"/>
        <param name="update_min_d"              value="0.20"/>
        <param name="update_min_a"              value="0.20"/>
        <param name="resample_interval"         value="1"/>
        <param name="transform_tolerance"       value="0.5"/>
        <param name="recovery_alpha_slow"       value="0.00"/>
        <param name="recovery_alpha_fast"       value="0.00"/>
        <param name="initial_pose_x"            value="$(arg initial_pose_x)"/>
        <param name="initial_pose_y"            value="$(arg initial_pose_y)"/>
        <param name="initial_pose_a"            value="$(arg initial_pose_a)"/>
        <param name="gui_publish_rate"          value="50.0"/>
        <remap from="scan"                      to="$(arg scan_topic)"/>
        <param name="laser_max_range"           value="3.5"/>
        <param name="laser_max_beams"           value="180"/>
        <param name="laser_z_hit"               value="0.5"/>
        <param name="laser_z_short"             value="0.05"/>
        <param name="laser_z_max"               value="0.05"/>
        <param name="laser_z_rand"              value="0.5"/>
        <param name="laser_sigma_hit"           value="0.2"/>
        <param name="laser_lambda_short"        value="0.1"/>
        <param name="laser_likelihood_max_dist" value="2.0"/>
        <param name="laser_model_type"          value="likelihood_field"/>
        <param name="odom_model_type"           value="diff"/>
        <param name="odom_alpha1"               value="0.1"/>
        <param name="odom_alpha2"               value="0.1"/>
        <param name="odom_alpha3"               value="0.1"/>
        <param name="odom_alpha4"               value="0.1"/>
        <param name="odom_frame_id"             value="odom"/>
        <param name="base_frame_id"             value="base_footprint"/>
    </node>
</launch>
```

Figura 3.1. Líneas de código del archivo `amcl.launch` para el robot Turtlebot. Autoría: (Robotis, 2019b).

Sobre este código diferenciamos dos partes: por un lado, los argumentos de entrada al fichero en sí, y por otro, la ejecución del nodo AMCL. Este nodo ya viene implementado en ROS junto a Turtlebot, en el paquete del mismo nombre, y permite modificar los

valores de los parámetros para adaptarlo a las necesidades y permitir su funcionamiento en una amplia gama de robots.

Entre los argumentos de entrada se encuentran el nombre del escáner que permite la localización y la posición inicial del robot. Los valores por defecto que se muestran en la figura 4.1 están establecidos para poder comunicarse con el sensor del Turtlebot.

Para poder ejecutar el nodo AMCL es necesario además pasarle multitud de parámetros para que se ajuste correctamente a nuestro robot. Entre todos los parámetros necesarios, los más importantes son los relacionados con la odometría y con el sensor del robot. Los parámetros que hacen referencia al láser en el nombre están directamente relacionados con el escáner. Es necesario conocer las especificaciones de nuestro escáner para poder establecer valores como el número máximo de haces empleado o la distancia máxima del láser, entre otros. La odometría también es fundamental para permitir una correcta localización del robot. El parámetro más importante es el relacionado con el tipo de movimiento en función de las ruedas (`odom_model_type`), que puede variar en función de unas ruedas diferenciales[4] u omnidireccionales. En el caso del Turtlebot, únicamente tiene ruedas diferenciales, de ahí que este esté establecido cómo valor por defecto.

3.2. Move_Base

El archivo `move_base.launch`, que ejecuta el nodo del mismo nombre, es fundamental para la navegación del robot. Ejecutar este nodo en un robot correctamente configurado da lugar a que el robot intente conseguir llegar a su posición de destino con la base del robot dentro de una tolerancia especificada o hasta que indique un error al intentar llegar. Si el robot detecta que se ha atascado, este nodo también intentará recuperarlo. En la figura 3.2 se presentan las líneas de código de este archivo.

[4] Diferenciales: Ruedas diferenciales son aquellas que incluyen un mecanismo diferencial. Este permite que, al girar hacia un lado con ruedas en paralelo, giren a diferentes velocidades (La del interior de la curva más lento que la del exterior).

```
<launch>
    <!-- Arguments -->
    <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle,
        waffle_pi]"/>
    <arg name="cmd_vel_topic" default="/cmd_vel" />
    <arg name="odom_topic" default="odom" />
    <arg name="move_forward_only" default="false"/>
    <!-- move_base -->
    <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
        <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS" />
        <rosparam file="$(find turtlebot3_navigation)/param/costmap_common_params_$(arg model
            ).yaml" command="load" ns="global_costmap" />
        <rosparam file="$(find turtlebot3_navigation)/param/costmap_common_params_$(arg model
            ).yaml" command="load" ns="local_costmap" />
        <rosparam file="$(find turtlebot3_navigation)/param/local_costmap_params.yaml" command
            ="load" />
        <rosparam file="$(find turtlebot3_navigation)/param/global_costmap_params.yaml" command
            ="load" />
        <rosparam file="$(find turtlebot3_navigation)/param/move_base_params.yaml" command="load"
            />
        <rosparam file="$(find turtlebot3_navigation)/param/dwa_local_planner_params_$(arg model
            ).yaml" command="load" />
        <remap from="cmd_vel" to="$(arg cmd_vel_topic)"/>
        <remap from="odom" to="$(arg odom_topic)"/>
        <param name="DWAPlannerROS/min_vel_x" value="0.0" if="$(arg move_forward_only)" />
    </node>
</launch>
```

Figura 3.2. Líneas de código del archivo `move_base.launch` para el robot Turtlebot. Autoría: (Robotis, 2019b).

Al igual que en el AMCL, aquí también hay dos partes claras, los argumentos de entrada al fichero y la ejecución del nodo.

Los parámetros necesarios son: el nombre de los tópicos necesarios para navegación, `cmd_vel`, y `odom` (velocidad y odometría del robot); y el tipo de robot Turtlebot. Estos valores son después empleados como parámetros de entrada sobre el nodo para ajustarlo a los mensajes que emite el robot.

Para la ejecución del nodo es necesario proporcionar varios parámetros. Entre ellos se encuentran la localización en carpetas de los mapas de características tanto global como local. Estos archivos ya estaban disponibles en la segunda versión de Campero. En la sección 4.1 se explica el progreso de las versiones de Campero. Anteriormente a esta segunda versión, se empleaban los archivos pertenecientes a Turtlebot con ligeras modificaciones. Finalmente se utiliza el comando `remap` para cambiar el nombre de los tópicos a algo que encaje con la nomenclatura empleada por los otros nodos.

3.3. Global launch file

Este archivo, `Turtlebot3_navigation.launch`, ejecuta todos los elementos que componen la navegación, incluidos los archivos de las secciones 3.1 y 3.2,, el mapa y el software necesario (figura 3.3).

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="map_file" default="$(find turtlebot3_navigation)/maps/map.yaml"/>
  <arg name="open_rviz" default="true"/>
  <arg name="move_forward_only" default="false"/>

  <!-- Turtlebot3 -->
  <include file="$(find turtlebot3_bringup)/launch/turtlebot3_remote.launch">
    <arg name="model" value="$(arg model)" />
  </include>

  <!-- Map server -->
  <node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)"/>

  <!-- AMCL -->
  <include file="$(find turtlebot3_navigation)/launch/amcl.launch"/>

  <!-- move_base -->
  <include file="$(find turtlebot3_navigation)/launch/move_base.launch">
    <arg name="model" value="$(arg model)" />
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
  </include>

  <!-- rviz -->
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find turtlebot3_navigation)/rviz/turtlebot3_navigation.rviz"/>
  </group>
</launch>
```

Figura 3.3. Líneas de código del archivo `Turtlebot3_navigation.launch` para el robot Turtlebot. Autoría: (Robotis, 2019b).

Este documento está estructurado en varias partes para una comprensión clara del mismo. Primero se especifican los argumentos de entrada. En este caso, los más importantes son el modelo de robot Turtlebot y la localización del mapa a abrir.

Posteriormente se procede a la ejecución del robot. El archivo `Turtlebot3_remote.launch` ejecuta únicamente el robot de forma virtual. No ejecuta ningún otro software de visualización o de control, como podrían ser procesos de navegación o software como Gazebo o rviz.

El nodo `map_server` es específico para cargar un mapa y trabajar sobre él en entornos virtuales, en este caso, colocar el robot en dicho escenario.

Ahora, desde dentro de este archivo, se llama a los otros dos ficheros previamente descritos `amcl.launch` y `move_base.launch`, que se ejecutarán con sus valores por defecto.

Finalmente, para poder ver el desarrollo de la simulación realizada con el software, es necesario ejecutar alguna forma de visualización. Para ello ejecutamos el archivo `Turtlebot3_navigation.rviz`.

Estos archivos que hemos descrito son los encargados de las ejecuciones realizadas en el apartado 2.6. En la siguiente sección se aplican estos conceptos a la navegación de nuestro robot móvil manipulador.

4. Robot Campero

En este Trabajo Fin de Grado se requería trabajar sobre el modelo de robot Campero. Este robot está siendo diseñado por la empresa Robotnik[5].

Campero es un robot de cuatro ruedas que tiene un brazo robótico de seis ejes. El brazo robótico está situado en la superficie del robot, y se puede programar para dos o tres dedos, dependiendo de la aplicación que se desee ejecutar (figura 4.1).

Figura 4.1. Primer plano de Campero en el entorno de simulación Gazebo.

El robot cuenta con dos escáneres láser, uno frontal y uno trasero. Cada uno es capaz de observar 270°, y están colocados en esquinas opuestas del vehículo. El robot también incluye una cámara óptica frontal. También se dispone de dos tipos de ruedas, omnidireccionales y direccionales.

Para este robot no está disponible una hoja de especificaciones como en el caso de Turtlebot. Sin embargo, el software incluye algunas medidas del robot, que dan cuenta de su magnitud. Así, el robot emplea ruedas de aproximadamente 40 cm de diámetro, que contrastan claramente con las ruedas de 6,6 cm de diámetro presentes en Turtlebot Waffle Pi. Se trata de un robot mucho más grande que el Turtlebot Waffle Pi, que parece estar diseñado para utilidades en las que se precisa fuerza y capacidad de transporte.

[5] Robotnik: Empresa de robótica española. C/ Ciudad de Barcelona, 3-A, Valencia (España)

4.1. Versión del software

Al estar el proyecto Campero en desarrollo durante la realización de este TFG, se han ido actualizando los archivos proporcionados por la empresa. Todas las versiones recibidas están firmadas como la versión 0.0.0, lo que determina que todavía es un proyecto relativamente nuevo.

La versión original de Campero fue proporcionada el 10 de abril de 2019. Ésta incluía una primera versión del robot que simulaba su manejo en dos entornos. En esta versión no había ninguna mención sobre la navegación del robot, por lo que se desarrollaron archivos de código específicos para este uso.

Posteriormente se recibió una actualización el 22 de mayo de 2019 que incluye, entre otras modificaciones, un paquete de navegación no operativo en el interior del software del robot. A pesar de no estar operativo incluye archivos de configuración sobre el robot Campero necesarios para la navegación. Después de probar los cambios incluidos en el robot, se instalaron los archivos generados hasta el momento en el TFG en el nuevo destino especificado por el fabricante.

Finalmente se recibió una última versión el 5 de agosto de 2019 que contenía gran cantidad de paquetes, pero no aportaba ninguna mejora significativa sobre el paquete de navegación. Tras probar esta versión se decidió continuar con la versión anterior.

4.2. Archivos proporcionados

Tal como se ha descrito. el sistema operativo ROS está basado en un funcionamiento por carpetas. Se comienza por establecer un espacio de trabajo (*work space*) en el que se generan las carpetas necesarias de nuestro robot. En los siguientes párrafos se describe el esquema de las carpetas de Campero, principalmente el primer nivel y cuando sea relevante el segundo.

`campero_ws`: Esta es la carpeta madre de definición de Campero, que contiene a todas las demás:

- **`Build`**: Carpeta obtenida al compilar el *work space* en nuestro espacio de trabajo.
- **`Devel`**: Carpeta obtenida al compilar el *work space* en nuestro espacio de trabajo.
- **`src/campero`**: Carpeta *source* y subcarpeta campero, que tiene los siguientes componentes:

- **`Campero_common`**: Contiene los paquetes desarrollados o en desarrollo disponibles para ejecutar por el robot.
  - **`Campero_control`**: Paquete encargado del movimiento y control del robot Campero. Una vez proporcionados los parámetros principales del robot (tipo de ruedas, tipo de brazo…), se encarga del control de los ejes del robot y de establecer el control de la base del robot.
  - **`Campero_description`**: Proporciona las distintas configuraciones que puede tener el robot (Ruedas omnidireccionales o direccionales y brazo con tres dedos o dos… cuatro posibles combinaciones en total). También contiene las descripciones de las estructuras dentro del robot, como las ruedas o los sensores.
  - **`Campero_localization`**: Paquete fundamental de localización del robot. Contiene el fichero AMCL y mapas de prueba sobre los que ejecutar el robot.
- **`Campero_robot`**: Archivos de configuración del robot. Fundamentalmente para el inicio y el control.
- **`Campero_sim`**: Ficheros para la simulación del robot y sus escenarios.
  - **`Campero_gazebo`**: Archivos principales de ejecución. Contiene la ejecución del software Gazebo (solo, junto con el robot campero, o junto a rviz). También incluye los escenarios sobre los que ejecutar Gazebo.
  - **`Campero_sim_bringup`**: Contiene el archivo de ejecución principal. Está diseñado para ejecutar el robot campero en un escenario exterior junto con Gazebo y rviz para conseguir una primera toma de contacto con el robot y manejarlo libremente por la escena.
- **`Openrave_catkin`**: Paquete de instalación necesario para el funcionamiento del robot. Para poder ejecutarlo correctamente es necesario instalarlo desde la terminal de Linux, al instalarse con ROS. OpenRAVE es un software de simulación de robots para desarrollo, simulación y aplicación de algoritmos de *motion planning*[6]

6 Motion planning: Se emplea para encontrar una secuencia de movimientos válidos que muevan el robot desde su posición de origen hasta su destino evitando obstáculos. (Wikipedia Contributors, 2019c)

- **`Robotnik_msgs`**: Paquete de mensajes para los robots de la empresa Robotnik.
- **`Robotnik_sensors`**: Paquete de los sensores de robots de la empresa Robotnik.

4.3. Instalación y ejecución

A continuación, se explica la instalación del robot Campero a partir de los ficheros pertinentes.

El primer paso consiste en colocar el espacio de trabajo `campero_ws` junto al espacio `catkin_ws` instalado por defecto con Turtlebot. Una vez colocado debemos eliminar los archivos `devel` y `build` para poder compilarlo en el ordenador de trabajo. Sobre la terminal de Linux se ejecutan los siguientes comandos:

```
cd campero_ws/
catkin_make
```

Esto compilará el espacio de trabajo para poder operar en él. Es posible que en este punto aparezcan mensajes de error o de advertencia diciendo que falta algún paquete necesario de ROS. Algunos estos paquetes son:

```
ros-kinetic-mavros
ros-kinetic-localization
ros-kinetic-costmap-prohibition-layer
ros-kinetic-teb-local-planner
ros-kinetic-twist-mux
ros-kinetic-teleop-tools
ros-kinetic-openrave
ros-kinetic-universal-robot
```

Para poder instalar los paquetes es necesario ejecutar comandos de este tipo:

```
sudo apt-get install [nombre del paquete]
```

Al tratarse de un comando "sudo", el sistema pedirá la contraseña de administrador. Una vez instalados todos los paquetes necesarios y compilado el espacio debemos añadir un plugin de teleoperación:

```
cd campero_ws/src
git clone https://github.com/RobotnikAutomation/teleop_panel.git
cd..
catkin_make
```

Una vez añadido este plugin y compilado todo el espacio adecuadamente procedemos a declaralo entorno de trabajo:

```
rosws init campero_ws
source devel/setup.bash
```

Ahora debería estar correctamente instalado el robot Campero. Para comprobar su correcto funcionamiento probamos a ejecutar un entorno de prueba proporcionado por Robotnik para el manejo de Campero:

```
roslaunch campero_sim_bringup campero_complete.launch
```

Una vez ejecutado correctamente y estando además Gazebo y rviz en ejecución, se puede probar a manejar el robot desde la pantalla de rviz.

Con estos pasos el robot debería proporcionar un correcto funcionamiento y deberíamos observar las siguientes pantallas de Gazebo (Figura 4.2) y rviz (Figura 4.3).

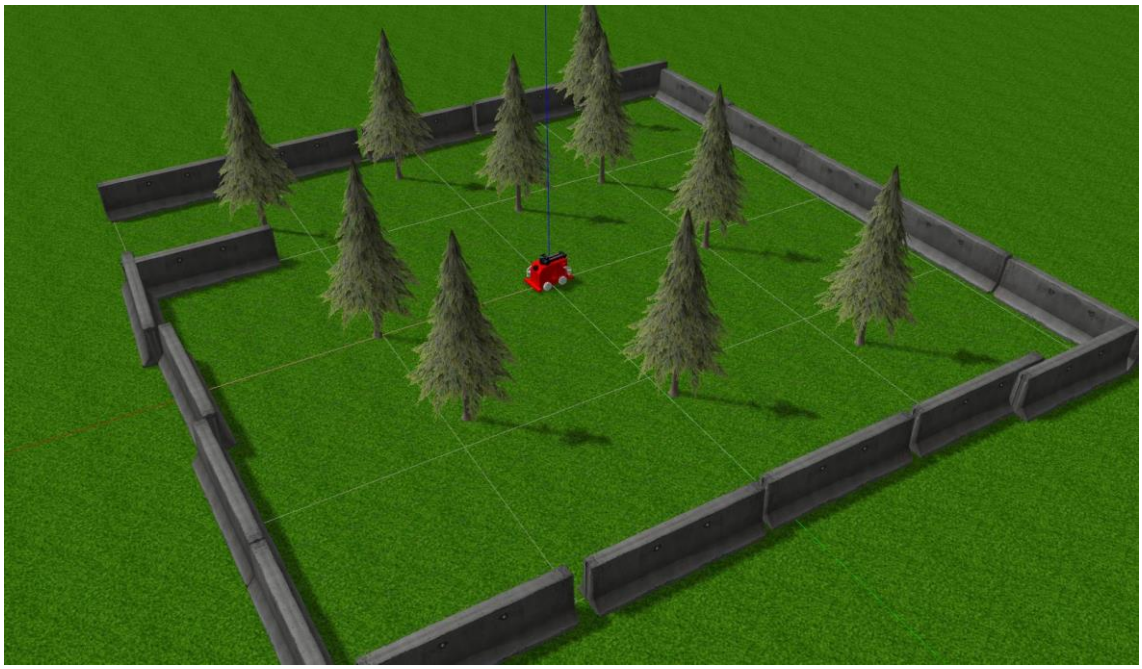


Figura 4.2. Simulación del robot Campero en el entorno Gazebo, en el escenario por defecto exterior.

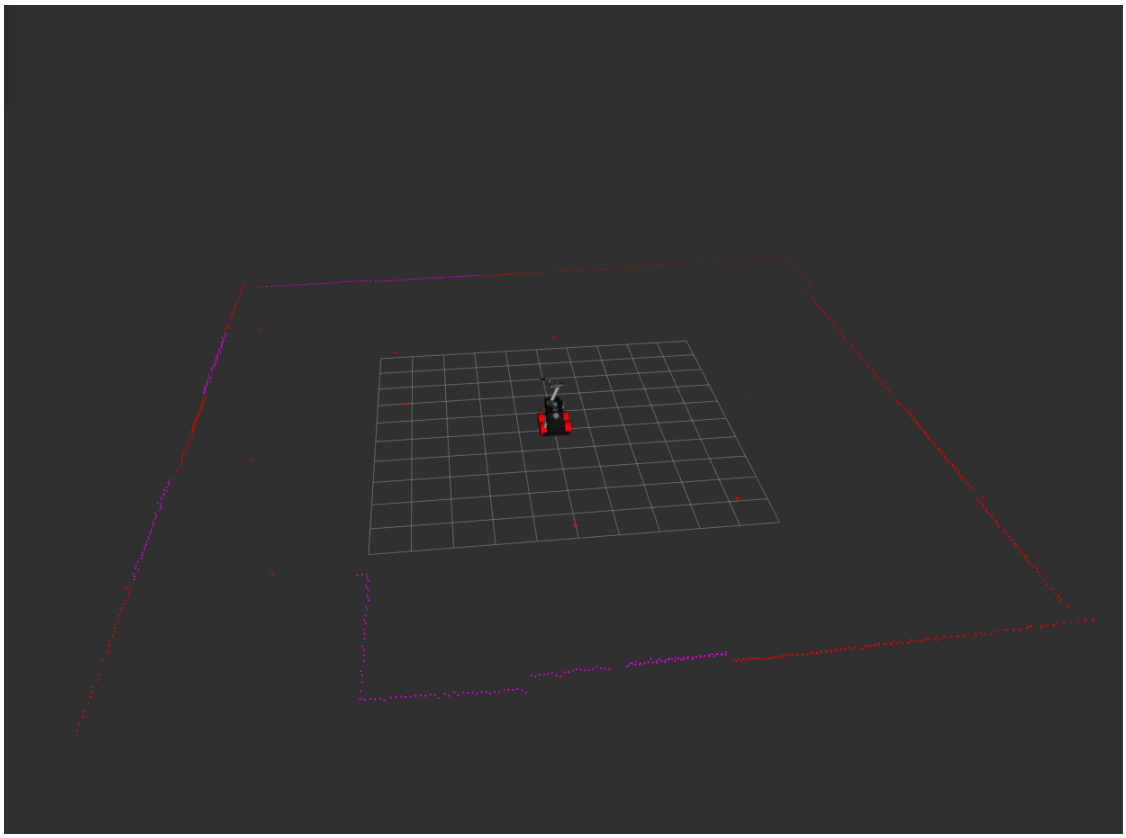


Figura 4.3. Simulación del robot Campero en el entorno rviz, en el escenario por defecto exterior.

5. Navegación sobre el robot Campero

Una vez descrito el funcionamiento del paquete de navegación del robot Turtlebot y documentados los archivos de especificaciones del robot Campero, se describen los aspectos principales de la navegación de Campero.

5.1. Preparación Inicial

Para comenzar, es preciso especificar dónde se quiere aplicar el paquete de navegación. En ROS se puede aportar capacidades de navegación a un robot de distintas formas. Las dos formas que se plantearon en este trabajo fueron: 1) la creación de un nuevo espacio de trabajo que se encargue de la comunicación y que aloje los ficheros de navegación, y 2) el desarrollo de un nuevo paquete, dentro de los ficheros del robot Campero, que se encargue de la navegación.

5.1.1. Modelo inicial

La opción de crear un nuevo espacio de trabajo y comunicar el nuevo espacio de trabajo con el espacio de trabajo `catkin_ws`, se vio afectada por problemas. Esta opción únicamente se puede aplicar si el robot está configurado de una forma específica, compatible con el entorno `catkin_ws`. Para conseguir esto sería necesario reestructurar los archivos del robot y renombrar nodos y tópicos para que se adaptaran a lo preestablecido por ejemplo en el paquete de navegación del robot Turtlebot.

Este problema hizo que se desestimara la primera opción y se optó por el desarrollo de un paquete de navegación propio para el robot Campero. Para ello se empieza creando un paquete dentro del espacio de trabajo `campero_ws`:

```
cd campero_ws/src/campero

catkin_create_pkg campero_navigation std_msgs rospy roscpp
```

Con esto creamos el paquete en la localización deseada, pero todavía falta compilarlo:

```
cd

cd campero_ws

catkin_make
```

Este proceso debería de haber creado el paquete correctamente con los archivos `packahe.xml` y `cmakelists.txt`.

5.1.2. Versión con navegación

Conforme avanzaba con el proyecto e iban llegando nuevas versiones del robot Campero, llegó una versión que dejaba claro que se estaba trabajando en el desarrollo de la navegación del robot.

Se observa que se había creado un paquete con el mismo nombre en el sistema, pero en un nivel inferior, en la subcarpeta `campero_common`. A pesar de estar creado, el paquete estaba casi vacío: sólo contenía archivos básicos del paquete. Entre ellos estaban los archivos de configuración del robot necesarios para poder aplicar la navegación, archivos que es necesario incluir como argumentos para lanzar el nodo de navegación del robot. Se tomó la decisión de adaptarse a la nueva localización del paquete y se movieron los archivos de navegación previamente desarrollados a esta localización, usando los datos de configuración recibidos en esta versión del software de campero.

5.2. Archivos finales

En esta sección se explican los nuevos ficheros de Campero con detalle. Los ficheros necesarios para este sistema de navegación son:

- `AMCL`, encargado de la localización del robot en cada momento.
- `move_base`, encargado principal de la navegación.
- `one_robot`, encargado de ejecutar el robot sin ningún software extra (una función similar a la del fichero `Turtlebot3_remote.launch`).
- Un fichero de ejecución global.

5.2.1. AMCL

Para poder aplicar el código AMCL sobre nuestro robot es preciso identificar un conjunto de valores correctos para los parámetros necesarios. Los parámetros más importantes para el correcto funcionamiento de este algoritmo son los nombres de los tópicos sobre los que publica nuestro robot. Es necesario localizar los tópicos responsables para poder incluirlos en el código:

```
Scanner: front_laser/scan

Mapa/global frame: campero_map

Odometría: campero_odom

Base frame: campero_base_footprint
```

También es necesario saber qué tipo de rueda emplear para poder aplicarlo como parámetro en la categoría "odom_model_type".

El código completo empleado, que incluye comentarios sobre cada parámetro, se presenta en el Anexo 10.2.

5.2.2. Move_base

Como ya se ha comentado, este archivo es el principal encargado de la navegación de nuestro robot. En estos archivos se empieza por asignar valores a los parámetros necesarios para la ejecución, entre ellos los nombres de los tópicos. Los tópicos más importantes a destacar son el de velocidad de navegación y el que hace referencia a la odometría de navegación, ambos diferentes a la odometría y velocidad normales. Estos tópicos de navegación se usan exclusivamente para enviar las velocidades calculadas en navegación, mientras que los empleados por defecto son de uso exclusivo para un movimiento manual del robot.

Una vez pasamos los parámetros comienza la ejecución del nodo `move_base`. Entre los parámetros necesarios para su ejecución están los parámetros de los `costmaps`[7] necesarios. Cómo nuestro robot dispone de dos tipos de ruedas, ha sido necesario colocar una cláusula condicional que permita diferenciar sobre cual estamos trabajando.

Ahora encontramos uno de los problemas que enfrentaremos más adelante. El nodo `move_base` requiere un tópico cómo el escáner que vamos a emplear, pero nuestro robot no dispone de un único escáner, sino que dispone de dos para cubrir los 360°. Por esta razón hemos tenido que elegir cual de ambos le proporcionamos al sistema, que en este caso fue el frontal.

El código completo empleado está en el Anexo 10.3.

5.2.3. One_robot

El código de este archivo se basa en la ejecución de una instancia del robot sin ningún otro *add on* para, posteriormente, ejecutar el software de navegación.

Se empieza estableciendo los argumentos de entrada requeridos en la ejecución. En primer lugar, se requiere el nombre del robot, ya que podría darse la situación de que

[7] Costmap: Mapa de características que indica la posición de obstáculos en el mundo. Aplicado principalmente en navegación se actualiza conforme recibe información de los sensores. (Open Source Robotics Foundation, 2019b).

se quisieran representar varios robots en un entorno para simular tareas cooperativas o entornos multi-robot. A continuación, se especifica la posición inicial del robot. Esta información no tiene mucha relevancia ya que se puede modificar posteriormente en el software de visualización y el propio algoritmo puede obtenerla de forma detallada. El archivo `xacro` permite identificar la configuración en la que se encuentra el robot (tipo de ruedas y de brazo). Este es un archivo XML en un formato que lo hace más corto y legible. Lo mismo sucede con el mapa que queremos representar en rviz, en el caso de que no lo haya creado el software del robot en sí (SLAM). El robot también necesita otros parámetros dependientes del tipo de ruedas o del tipo de localización empleado (odometría o AMCL).

El siguiente paso es ejecutar el nodo que conlleva la simulación del robot en Gazebo, siguiendo el esquema tanto del robot Turtlebot como de los ficheros propios del robot Campero. También es necesario ejecutar el nodo que permita el control del mismo y que simule las uniones de sus partes, lo que se especifica en la carpeta `campero_control`.

Habiendo cubierto estos apartados, la ejecución de este archivo debería crear un robot en el entorno de Gazebo para permitir aplicarle el algoritmo deseado, en nuestro caso la navegación del robot.

El código completo empleado está en el Anexo 10.4.

5.2.4. Global

Finalmente, fue preciso desarrollar un archivo de ejecución global, que deberá ejecutarse para poner en marcha todo el programa. Esto implica ejecutar los archivos anteriores, ejecutar el nodo de mapeado y el software empleado, rviz y Gazebo. Empezamos, como en todos los casos, con los argumentos de entrada para la ejecución del archivo.

Entre estos argumentos, al igual que en el modelo del Turtlebot, están presentes el nombre del robot, la posición inicial, el `xacro` y el mapa, y el tipo de movimiento y brazo.

El argumento del prefijo también debe especificarse en este archivo. Viendo otros archivos del robot Campero, se pudo observar que el nombre de los tópicos varía en función del nombre del robot, lo que está preparado para evitar superposiciones en la emisión de información por tópicos. El nombre del robot se posiciona al principio de

todos los tópicos en los que publican los nodos pertinentes al mismo. Por esta razón se incluye un argumento con ID del robot que pasa a ser el prefijo usado para localizar los tópicos y acceder a ellos.

Posteriormente, necesitamos ejecutar el algoritmo descrito anteriormente para posicionar un robot en nuestro entorno de simulación. Para pasarle los parámetros necesarios basta con referirse a los que le hemos puesto a la entrada, los cuales deberían ser suficientes para ejecutar correctamente el archivo.

Como dato importante se puede apreciar que, antes de ejecutar el robot, se coloca un indicador que se encarga de abrir un grupo. Esto dictamina que todos los elementos en su interior están bajo el mismo nombre. Este paso es fundamental para el funcionamiento, ya que de otra manera sería imposible localizar los tópicos fácilmente y no se podrían ejecutar entornos multi-robot en un futuro.

El archivo `map_server` se encarga de cargar el mapa sobre el que observar la simulación. En su interior contiene el nodo del mismo nombre que empieza el proceso. La última parte de este código cierra el grupo, ya que se han ejecutado todos los archivos necesarios en su interior.

A continuación, de forma opcional se ejecuta un fichero que simula la acción de la gravedad. Sólo tiene utilidad en el caso de un posible choque, para que el robot en la simulación vuelva a la posición de reposo. Si este PID[8] no estuviera implementado, en el momento en el que el robot se chocara podría quedar levitando y sería preciso reiniciar la simulación.

Como parte final de este código se ejecutan Gazebo y rviz para poder observar la simulación en un visualizador.

El código completo empleado está en el Anexo 10.5.

[8] Control Proporcional, Integral y Derivativo

6. Experimentación con el robot Campero

En esta sección se explica el funcionamiento del código presentado en la sección 5 y se aplica a casos de estudio.

6.1. Reconocimiento del entorno

Previo a la ejecución de la navegación es necesario realizar un mapa del entorno en el caso de que no dispongamos de uno de antemano. Para esto es necesario emplear la funcionalidad de Gmapping y SLAM introducida en el archivo global de ejecución. Ejecutamos el siguiente código en una terminal:

```
Roslauch campero_common campero_navigation
        campero_nav.launch gmapping:=true
```

Con esto deberíamos de ejecutar la navegación del robot junto a un mapeado del entorno. En rviz debería mostrarse el robot empezando a mapear el entorno, tal como se aprecia en la figura 6.1.

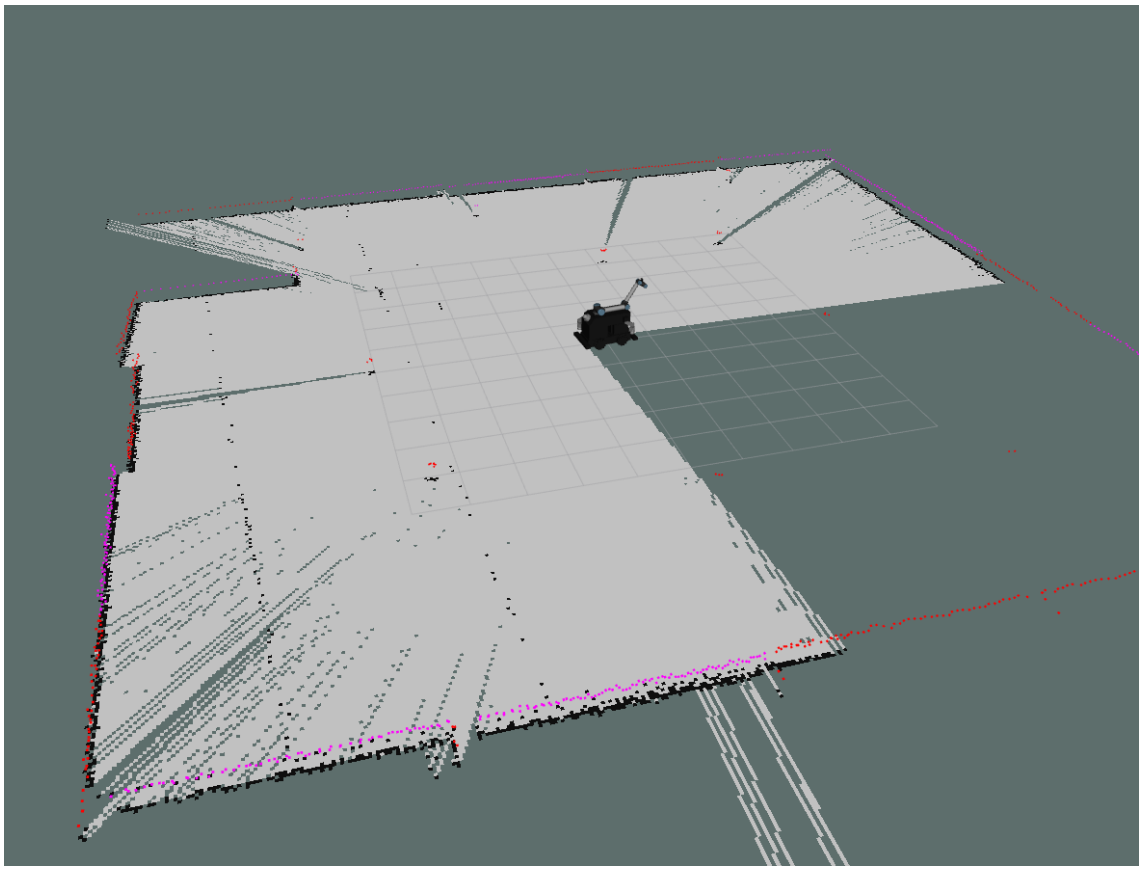


Figura 6.1. Robot Campero sobre un entorno sin mapear.

Aquí se evidencia la problemática de emplear un único escáner: el robot presenta un punto ciego tanto en navegación cómo en mapeado que le impide ver obstáculos en 90° de su visión. Ahora podemos mover el robot manualmente por el entorno o aplicar la navegación para conseguir un mejor resultado en el mapa. Tras enviar el robot por el mapa deberíamos tener una imagen similar a la figura 6.2.

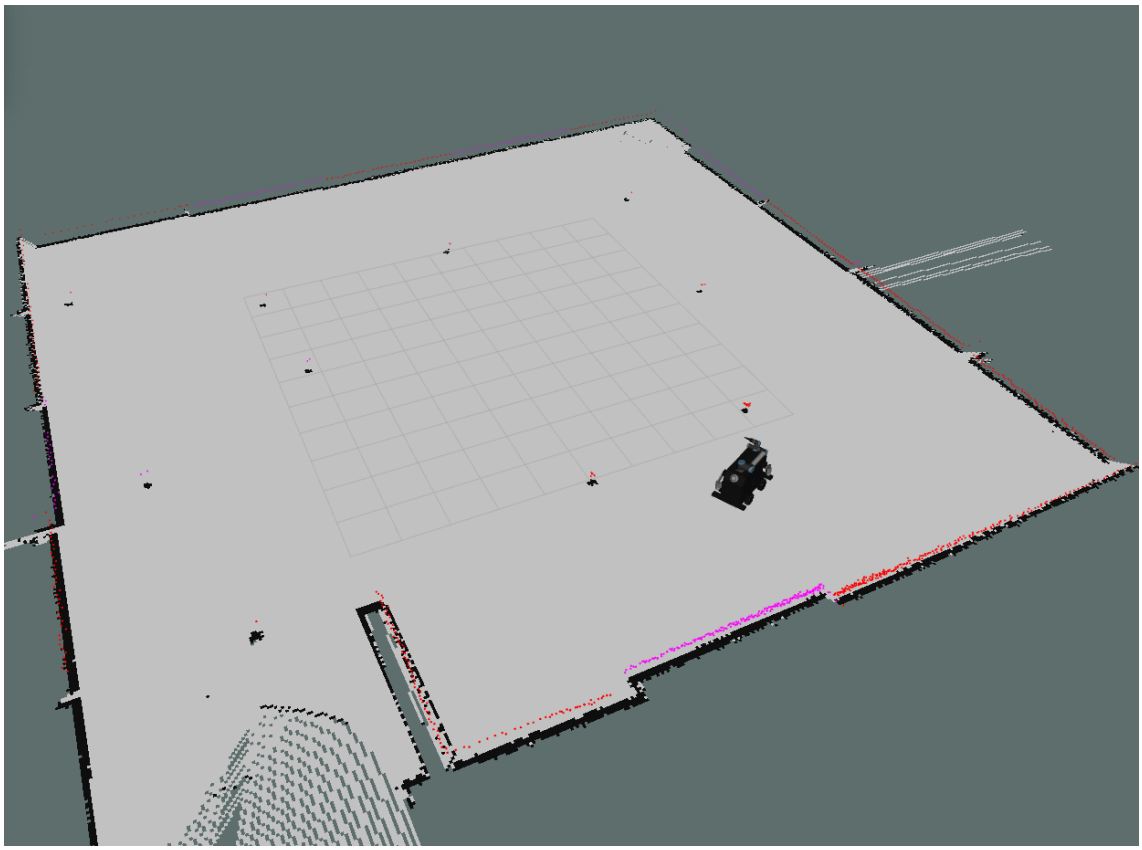


Figura 6.2. Robot Campero sobre un entorno mapeado.

Una vez explorado el entorno debemos guardar el mapa producido. Sin cerrar la ejecución anterior abrimos otra terminal:

```
rosrun map_server map_saver map:=/campero/map -f mymap
```

Este comando creará dos archivos: `mymap.pgm` y `mymap.yaml`.

El archivo `mymap.pgm` se presenta en el Anexo 10.6.

Ahora es preciso colocar estos archivos en la propia ejecución de la simulación:

```
Roslaunch campero_common campero_navigation
    campero_nav.launch map:=
    [Directorio en el que guardemos el mapa]
```

Tras esta ejecución se puede proceder con la navegación.

6.2. Navegación en exterior con ruedas diferenciales

A continuación, se muestra un caso de estudio en el entorno exterior con las ruedas por defecto de Campero: diferenciales. Antes de ejecutar la navegación, como el entorno es conocido de antemano, ejecutamos la línea siguiente haciendo referencia a la localización del mapa del entorno:

```
Roslaunch campero_common campero_navigation
        campero_nav.launch map:=
        [Directorio en el que guardemos el mapa]
```

Una vez ejecutado este comando deberíamos ver que se empiezan a ejecutar Gazebo y rviz para la visualización de nuestra navegación. Cabe destacar que no es necesario ejecutar un maestro antes (`roscore`) debido a que `roslaunch` ya se encarga de ejecutarlo por sí mismo.

Al ejecutar este algoritmo deberíamos ver imágenes similares a las que se presentan en la figura 6.3 (Gazebo) y 6.4 (rviz).

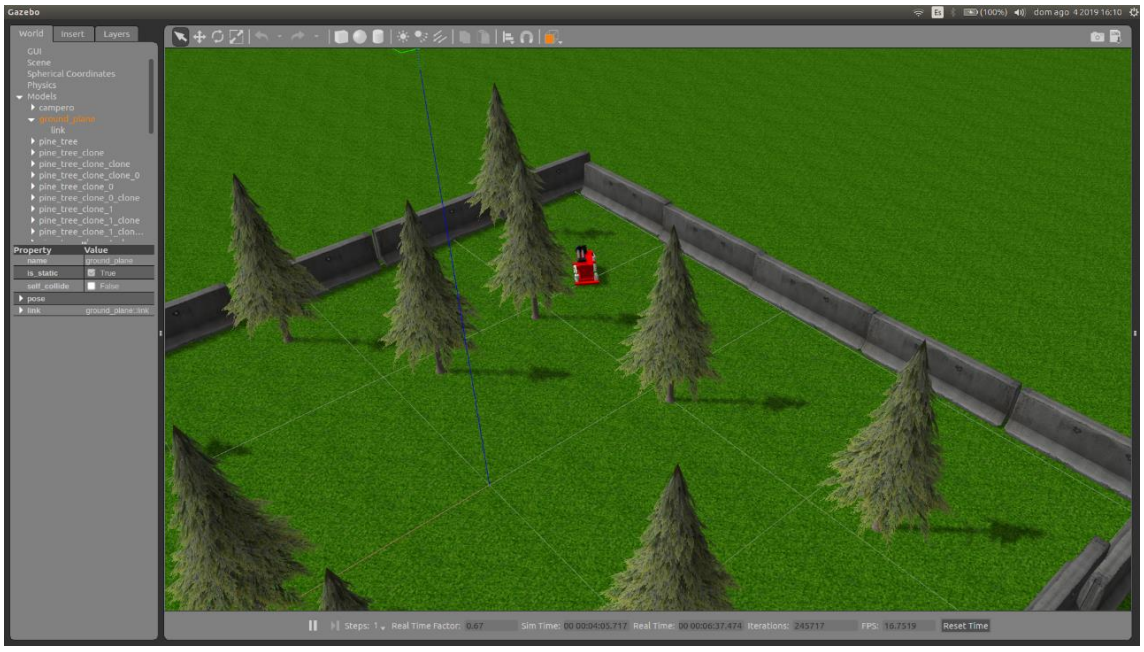


Figura 6.3. Representación en Gazebo de una simulación con el robot Campero usando ruedas diferenciales en el entorno exterior.

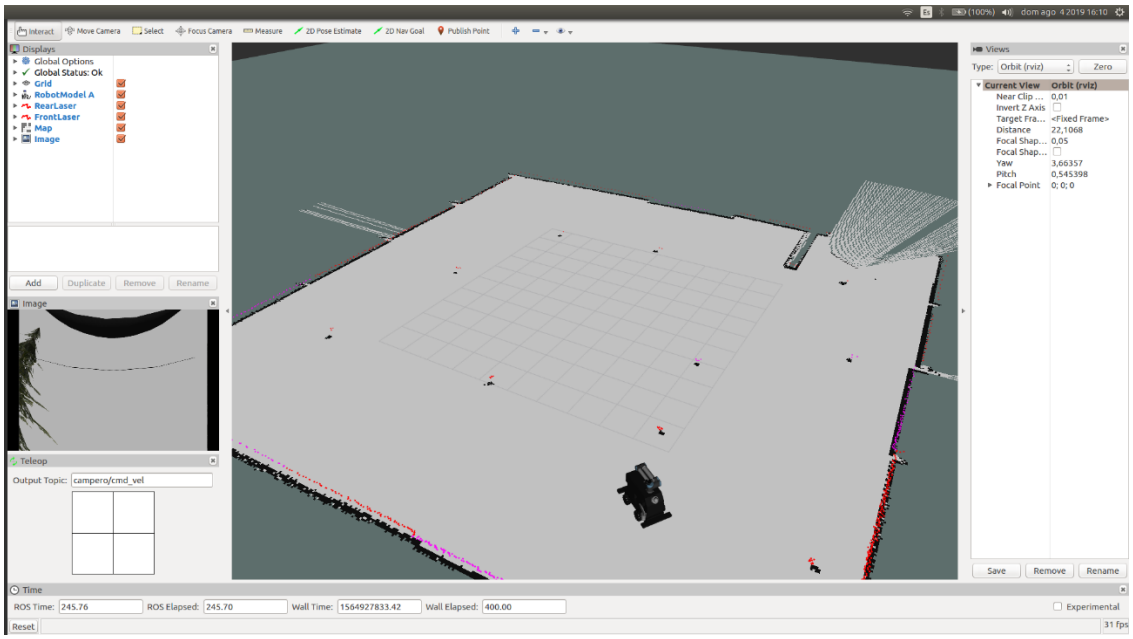


Figura 6.4. Representación de una simulación exitosa de Campero en rviz.

De esta manera se puede apreciar en Gazebo que se ha simulado correctamente el robot junto con el entorno de simulación. Por otro lado, en rviz vemos el robot y los sensores disponibles, tanto la cámara como el sensor frontal.

Para probar el sistema de navegación, en rviz debemos establecer un objetivo a alcanzar por el robot.

De la misma forma que en el caso del Turtlebot, es posible indicar a Campero el destino de la navegación por medio de una terminal. Para ello es preciso realizar una serie de cambios para adaptar el comando usado en Turtlebot, principalmente sobre el nombre de los tópicos. El tópico a modificar es el del destino, pero Campero trabaja con varios tópicos que interactúan con el destino, principalmente:

`/campero/move_base/goal`

`/campero/move_base_simple/goal`

Tenemos que seleccionar cuál de estos tópicos emplear para comunicarle el destino de navegación a nuestro robot en función de las ventajas e inconvenientes que incluyan.

El primero consiste en una forma de interacción que prepara el destino para la introducción de una cadena de diferentes destinos, dándole un ID a cada uno y posteriormente las coordenadas. Este tópico requiere la introducción de datos en formato "`MoveBaseActionGoal`" (Open Source Robotics Foundation, 2019h).

El segundo es el más similar al usado en Turtlebot. Requiere el mismo formato de introducción de datos, "`PoseStamped`", y tiene un funcionamiento similar, por lo que es el que emplearemos.

Una vez elegido el tópico sobre el que publicar, necesitamos cambiar el nombre del tópico empleado para el mapa. Campero emplea "`campero_map`". Una vez modificado se aplica la posición y la orientación de la misma forma que Turtlebot.

El comando que obtenemos es:

```
rostopic pub /campero/move_base_simple/goal
  geometry_msgs/PoseStamped '{header: {stamp: now,
  frame_id: "campero_map"}, pose: {position: {x: 3.0, y:
  0.0, z: 0.0}, orientation: {w: 1.0}}}'
```

Para este trabajo vamos a emplear la opción de menú de rviz "2D Nav Goal", pero tanto esta opción como el comando anterior dan los mismos resultados. Hacemos la selección que se presenta en la figura 6.5.

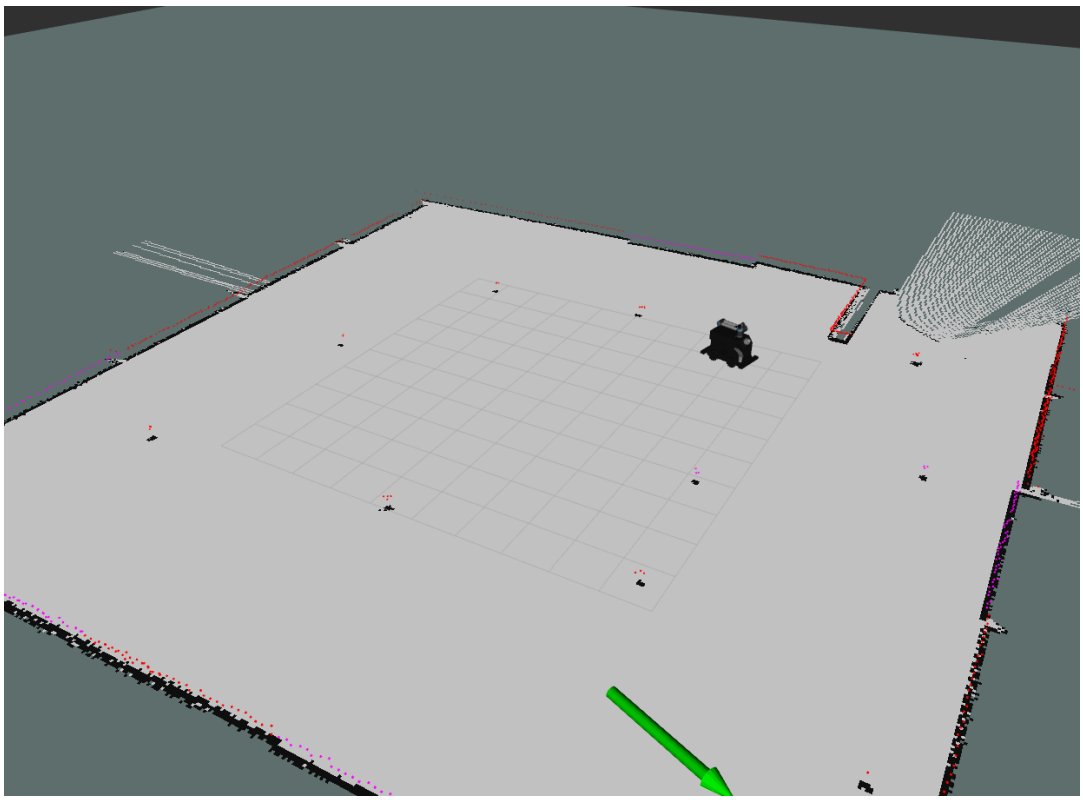


Figura 6.5. Representación en rviz del destino de la navegación del robot Campero. Robot Campero en la posición inicial del recorrido.

Intencionadamente hemos seleccionado un camino que requiere esquivar obstáculos, árboles en este caso. Si el robot siguiera un camino recto colisionaría con estos. Tal como se aprecia en la figura 6.6, el robot localiza el obstáculo y recalcula una ruta para llegar al destino.

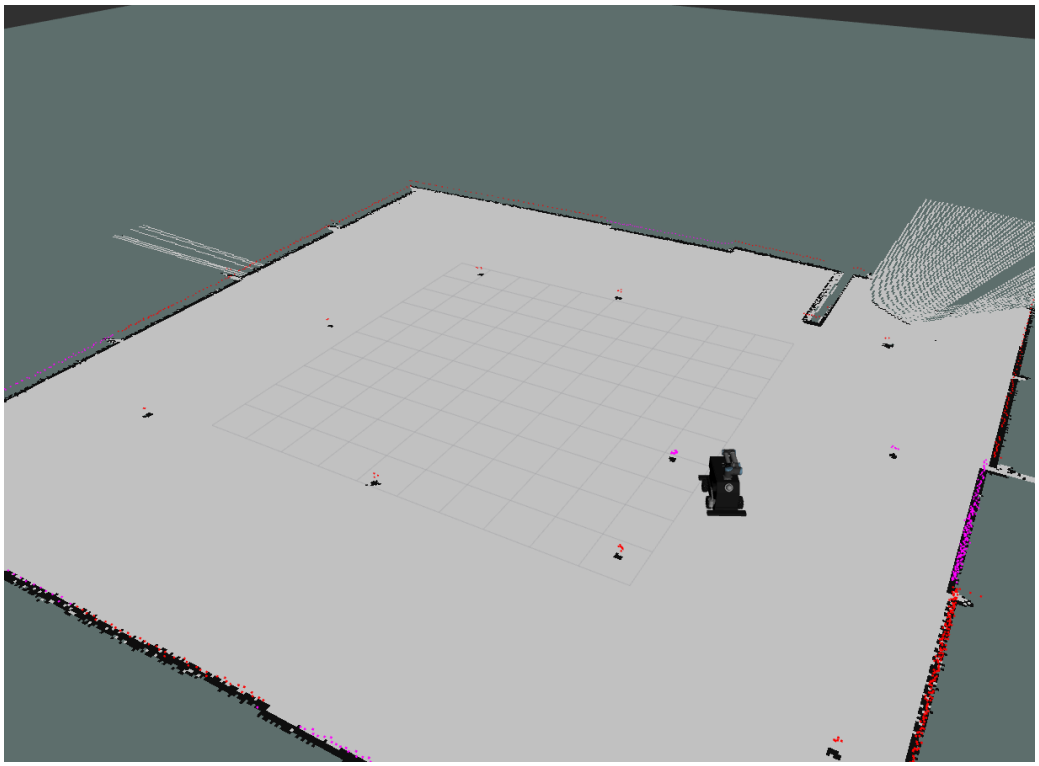


Figura 6.6. El robot Campero durante la navegación, tras esquivar los obstáculos. Robot Campero en un punto intermedio del recorrido.

Se aprecia cómo el software de navegación y de evitación de obstáculos se ha acoplado correctamente a Campero y ha sido capaz de esquivar el obstáculo introducido para llegar al destino. (figura 6.7).

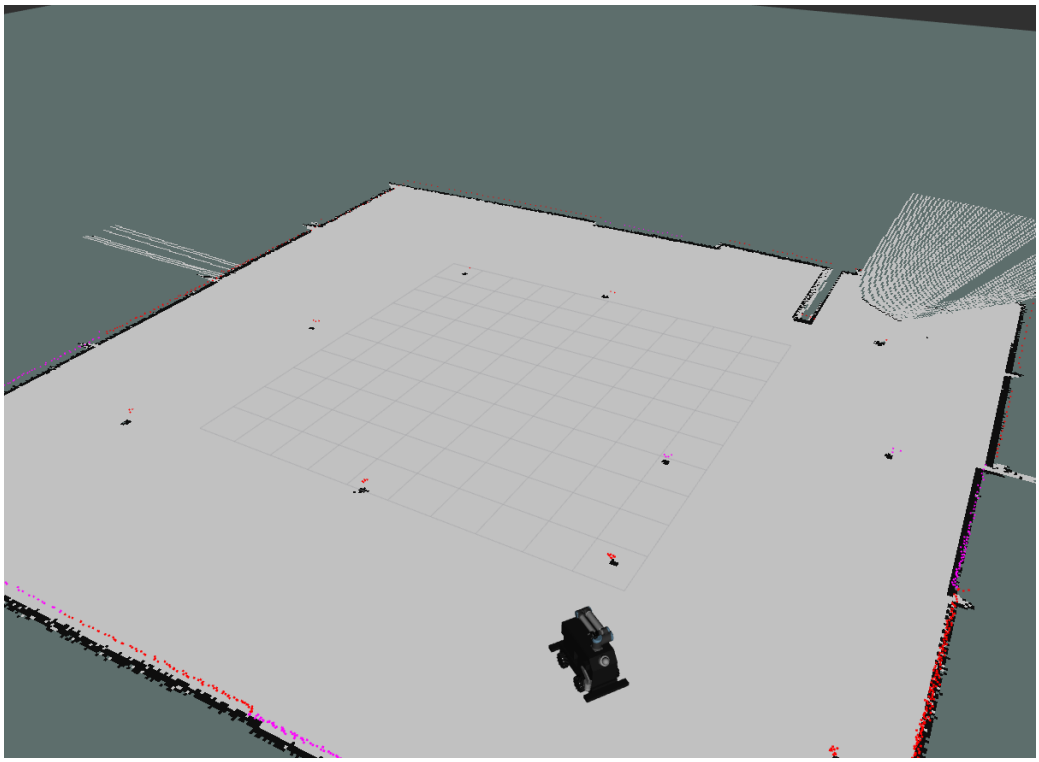


Figura 6.7. El robot Campero ha alcanzado su destino.

6.3. Navegación en interior con ruedas omnidireccionales

En esta sección se muestra un caso de estudio que consiste en la navegación en otro entorno proporcionado por la empresa Robotnik, que incluye ruedas omnidireccionales. La figura 6.8 muestra en Gazebo este entorno cerrado que se corresponde con el interior de una construcción. El entorno tiene forma rectangular, y tiene una pared interior que permite al robot acceder a la parte trasera por los dos lados. Delante de la pared hay tres obstáculos pegados a ella a modo de mesas. Al igual que en el caso anterior, en las primeras simulaciones es necesario realizar un mapa del entorno, que se empleará más adelante para guiar la navegación. El archivo resultante, `mapaInt.pgm` se presenta en el Anexo 10.7.

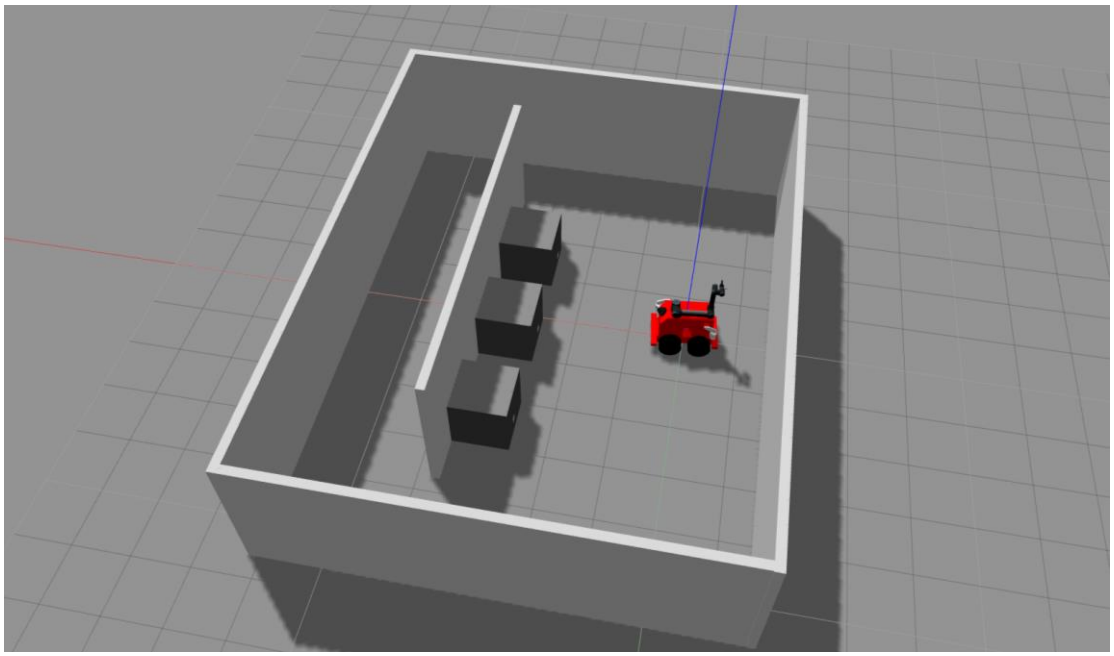


Figura 6.8. Representación en Gazebo de una simulación con el robot Campero usando ruedas omnidireccionales en el entorno interior.

La figura 6.9 muestra en rviz a campero en este mismo entorno. Se puede comprobar que el robot está equipado con ruedas omnidireccionales realizando una simulación particular: se establece como destino el mismo punto de origen y se comprueba que para ejecutarla el robot hace una rotación pura sin desplazarse. Tras hacer otras pruebas simples y comprobar que la navegación funciona adecuadamente, procedemos a ejecutar una simulación más compleja. Para ello, establecemos cómo objetivo un punto en la parte posterior de la pared interior, tal como se muestra en la figura 6.10.

Una vez establecido el objetivo, el software de navegación se encarga de esquivar el obstáculo principal, la pared interior. La figura 6.11 muestra al robot en tránsito, rodeando la pared interior. La figura 6.12 muestra al robot detenido en el punto de destino y con la orientación deseada.

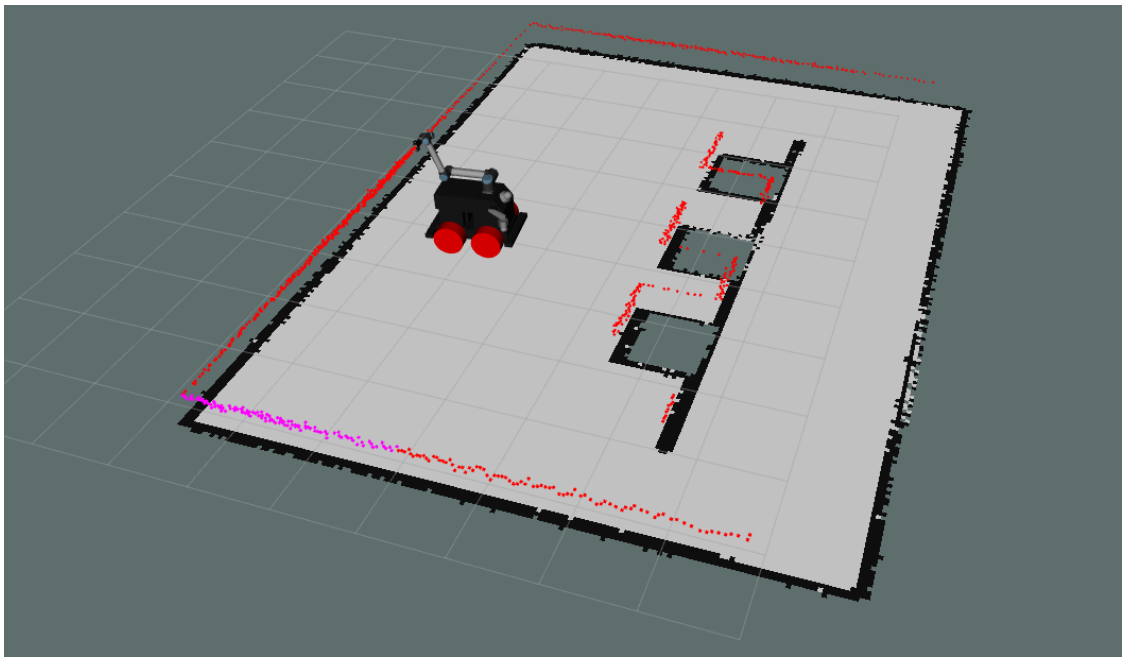


Figura 6.9. Representación en rviz del robot Campero usando ruedas omnidireccionales en el entorno interior.

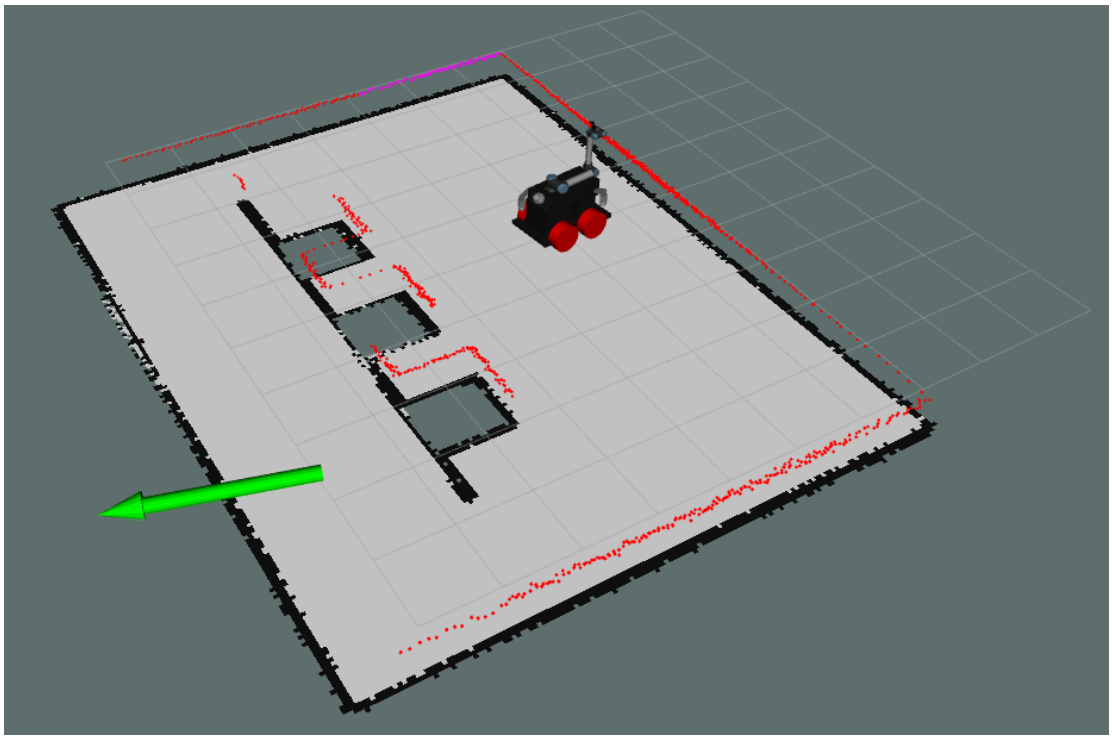


Figura 6.10. Representación en rviz del robot Campero con un objetivo de navegación establecido en un punto localizado tras la pared interior.

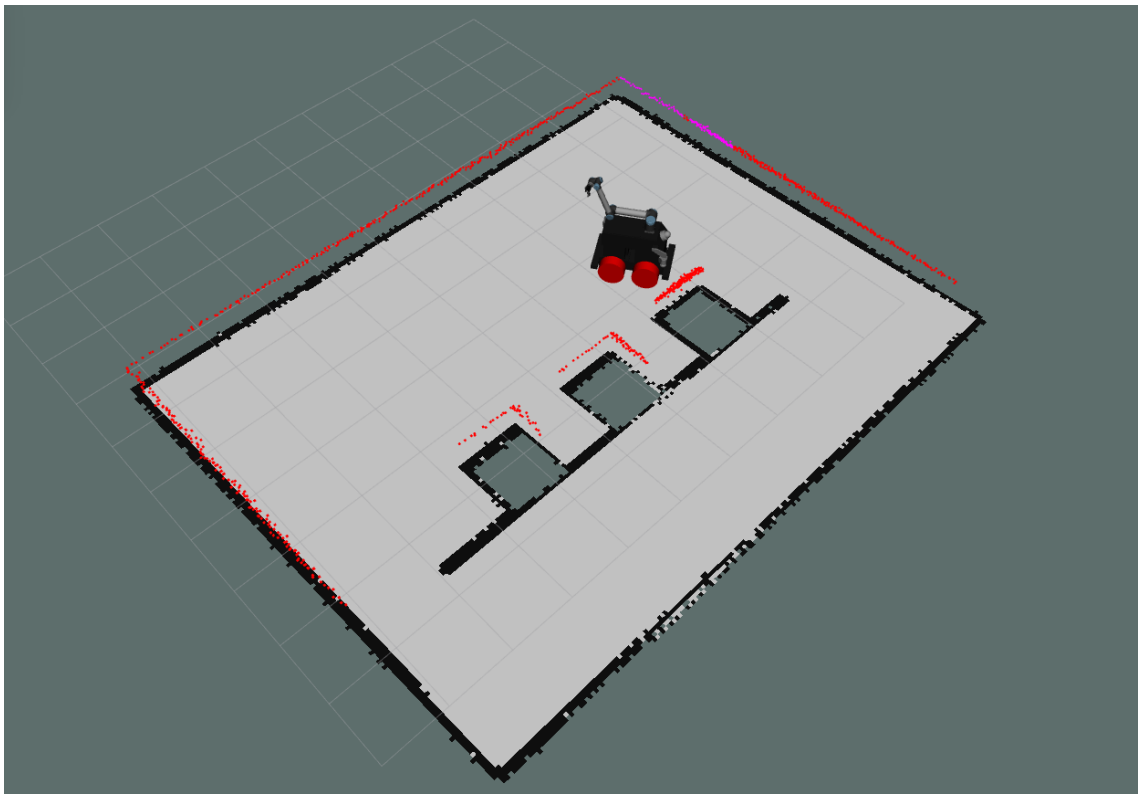


Figura 6.11. Representación en rviz del robot Campero navegando hacia el objetivo, para lo que debe rodear la pared interior.

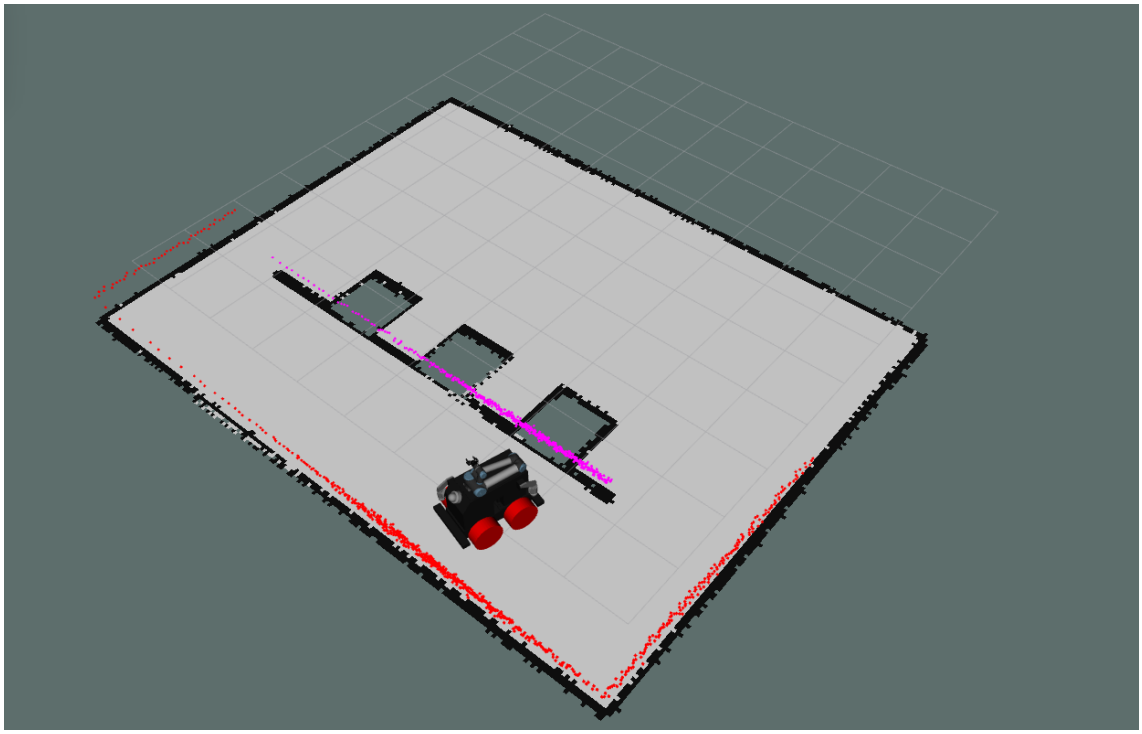


Figura 6.12. Representación en rviz del robot Campero detenido en el punto de destino y con la orientación deseada.

7. Discusión general

7.1. Simulación con el robot Turtlebot

Las simulaciones de la navegación de Turtlebot Waffle Pi han funcionado en general de forma adecuada. Todavía se aprecian algunos problemas que es necesario pulir, como errores de ejecución y algún error en el mapeado. Sin embargo, la tecnología de navegación aplicada a este robot ha sido muy eficaz. Sólo es necesario resolver algunos aspectos para conseguir una ejecución más fluida. Las simulaciones con Turtlebot me han permitido familiarizarme con los procedimientos de gmapping y SLAM, que pude luego aplicar al robot Campero con pequeñas adaptaciones.

A pesar de estar muy desarrollado, Turtlebot presentó alguna complicación en relación con la instalación y la ejecución. Fue necesario instalar algunos paquetes necesarios para la ejecución de otros. Una vez instalado, seguir los pasos del tutorial no resultó suficiente para conseguir simulaciones exitosas. El problema resultó estar asociado a la ejecución del mapa. Pude resolver este problema realizando una ejecución por partes en distintas terminales, tal como se explica en el apartado 2.6. Tras solucionar esto, la ejecución del robot seguía provocando errores. Por ello, nuevamente, fue necesario ejecutar el robot en una nueva terminal. Resueltos estos problemas, se pudo ejecutar las simulaciones. Sin embargo, en ocasiones aparecían mensajes de errores y advertencias que eran compatibles con la continuación de la ejecución del programa.

La navegación de Turtlebot en las condiciones experimentales se benefició de disponer de un solo escáner con cobertura de 360°. De esta manera, no había ángulos muertos y el robot pudo evitar los obstáculos correctamente. Aunque en este entorno el robot se haya beneficiado de este sensor único, en otros casos podría ser más práctico usar varios sensores.

7.2. Simulación con el robot Campero

La instalación de Campero no dio problemas específicos. La guía del robot y la guía proporcionada por los tutores de este trabajo permitieron progresar de forma rápida y firme.

Las simulaciones realizadas sobre Campero han sido satisfactorias, pero han puesto de manifiesto que el robot está una fase inicial de su desarrollo y que todavía necesita mucho trabajo para llegar a una madurez similar a la de Turtlebot. Las simulaciones han

permitido navegar con el robot Campero en dos entornos diferentes, aunque con dos limitaciones relevantes.

La primera limitación importante es que en ocasiones el planificador de trayectorias no consigue trazar un camino desde la localización actual hasta el objetivo. Este es el caso de simulaciones que necesitan esquivar obstáculos con trayectorias complejas. A la vista de estas situaciones en Gazebo, parece ser físicamente posible llegar, pero el software no es capaz de trazar el camino. En estos casos, el software de navegación se muestra demasiado restrictivo. Cabe pensar que haya diferencias entre la geometría real de Campero y los parámetros geométricos introducidos en `move_base.launch`. Esto podría explicar también algunas colisiones durante las maniobras. El hecho de que se usara la misma rutina de navegación en Turtlebot sin dar lugar a estos problemas me hace pensar que el error podría localizarse en la parametrización del robot Campero.

La segunda limitación de las simulaciones con Campero consiste en que el software de navegación (`move_base.launch`) sólo gestiona una entrada de señal de escáner. En el caso de Campero, como dispone de dos escáneres, se eligió usar el escáner delantero. Como consecuencia, nos encontramos con un punto ciego durante la navegación de 90° (la parte trasera izquierda). Esto es especialmente limitante para la navegación con las ruedas omnidireccionales, ya que a la hora de girar o de dirigirse hasta el destino el robot no detecta los obstáculos por su punto ciego y puede sufrir colisiones. Con ruedas direccionales el problema es menos importante porque el robot siempre se dirige de cara hacia el destino (el punto ciego está a la espalda del robot). Con las ruedas direccionales, este problema puede ser importante durante los giros que realiza el robot al posicionarse antes de moverse o después de llegar al destino.

8. Conclusiones

La realización de este TFG ha permitido que el alumno se familiarizara con la herramienta ROS, habiendo realizado una instalación exitosa de Ubuntu, ROS, Turtlebot 3 y Campero, habiendo manejado los archivos que implementan SLAM y la navegación autónoma de ambos robots, y habiendo completado simulaciones en distintos robots, configuraciones, entornos y modos.

Además, de este TFG se derivan las siguientes conclusiones:

1. El Software de Turtlebot para ROS está ya muy desarrollado y da lugar a simulaciones de navegación exitosas en situaciones que requieren trayectorias complejas que cambian durante la propia simulación. Sin embargo, se han encontrado problemas durante la ejecución, que han necesitado la apertura simultánea de varias terminales de Ubuntu.
2. El software de Campero para ROS está en una fase de progreso rápido por parte de la empresa Robotnik, a pesar de que los archivos de código fuente todavía están en una versión inicial. Entre abril y agosto de 2019 se han recibido tres versiones con mejoras importantes en cuanto a la parametrización de la navegación y desarrollo del brazo robótico. Sin embargo, el software de Campero no contiene en este momento rutinas específicas para la navegación.
3. Las rutinas de navegación de ROS usadas en Turtlebot se han aplicado con éxito a Campero, usando los parámetros del robot recibidos en la versión de 22 de mayo de 2019. Con esta configuración, Campero ha realizado navegaciones en dos entornos y dos tipos de ruedas, alcanzando los puntos propuestos.
4. Las simulaciones realizadas con campero han puesto de manifiesto dos limitaciones del estado actual del software y su parametrización: 1) en algunos casos en que es necesario esquivar obstáculos, el robot no es capaz de trazar un camino que visualmente parece posible o bien colisiona en las maniobras; y 2) el software de navegación sólo acepta la entrada de uno de los dos sensores de Campero, lo que genera un punto ciego al elegir uno de los dos.

9. Recomendaciones para trabajos futuros con el robot Campero

En relación con las posibles mejoras a implementar en trabajos futuros con Campero, será preciso abordar la mejora de la navegación. Los parámetros geométricos usados en la versión con la que está realizado el trabajo parecen no ser completamente adecuados. El primer paso para resolver este problema sería revisar con detalle la parametrización de la geometría del robot.

Otras posibles mejoras llevarían a incluir otros paquetes que puedan ser de interés para el usuario o para mejorar las capacidades del robot. Este sería el caso de otras formas de localización, de otras rutinas de navegación o de entornos multirobot. Quizás la mejora más clara podría ser el desarrollo de un software específico para el manejo del brazo robótico localizado en la superficie del robot Campero, que actualmente no está operativo. ROS tiene una gran variedad de utilidades para el control de robots manipuladores, que podrían adaptarse a Campero.

Como ya se ha comentado, a lo largo del trabajo se han ido recibiendo nuevas versiones con actualizaciones del software de Campero. Cabe esperar que este intenso desarrollo continuará en un futuro, por lo que dejarán de ser necesarias algunas de las recomendaciones actuales y aparecerán otras que darán pie a nuevos trabajos. Las nuevas versiones del software apuntan a mejoras inminentes en los campos de navegación y de manejo del brazo robótico. Por ello, convendría trabajar en la introducción de nuevos paquetes o del desarrollo de entornos multirobot. Entre las funciones que pueden ser interesantes para el usuario, se incluyen la conducción autónoma (incluyendo, por ejemplo, paquetes similares al "Autonomous Driving" de Turtlebot) o paquetes para la manipulación de objetos en un entorno multirobot y colaborativo.

Para finalizar, Campero es todavía un proyecto joven que requiere de mucho trabajo y dedicación. Existen gran cantidad de mejoras y de correcciones a realizar.

10. Anexos

10.1. Archivo de especificaciones del robot Waffle Pi por la empresa Robotis

• Features

WORLD'S MOST POPULAR ROS PLATFORM
TurtleBot is the world's most popular open source robot for education and research.

AFFORDABLE COST
TurtleBot is the most affordable platform for educations and prototype research & developments.

SMALL SIZE
Imagine the TurtleBot in your backpack and bring it anywhere.

EXTENSIBILITY
Extend ideas beyond imagination with various SBC, sensor, motor and flexible structure.

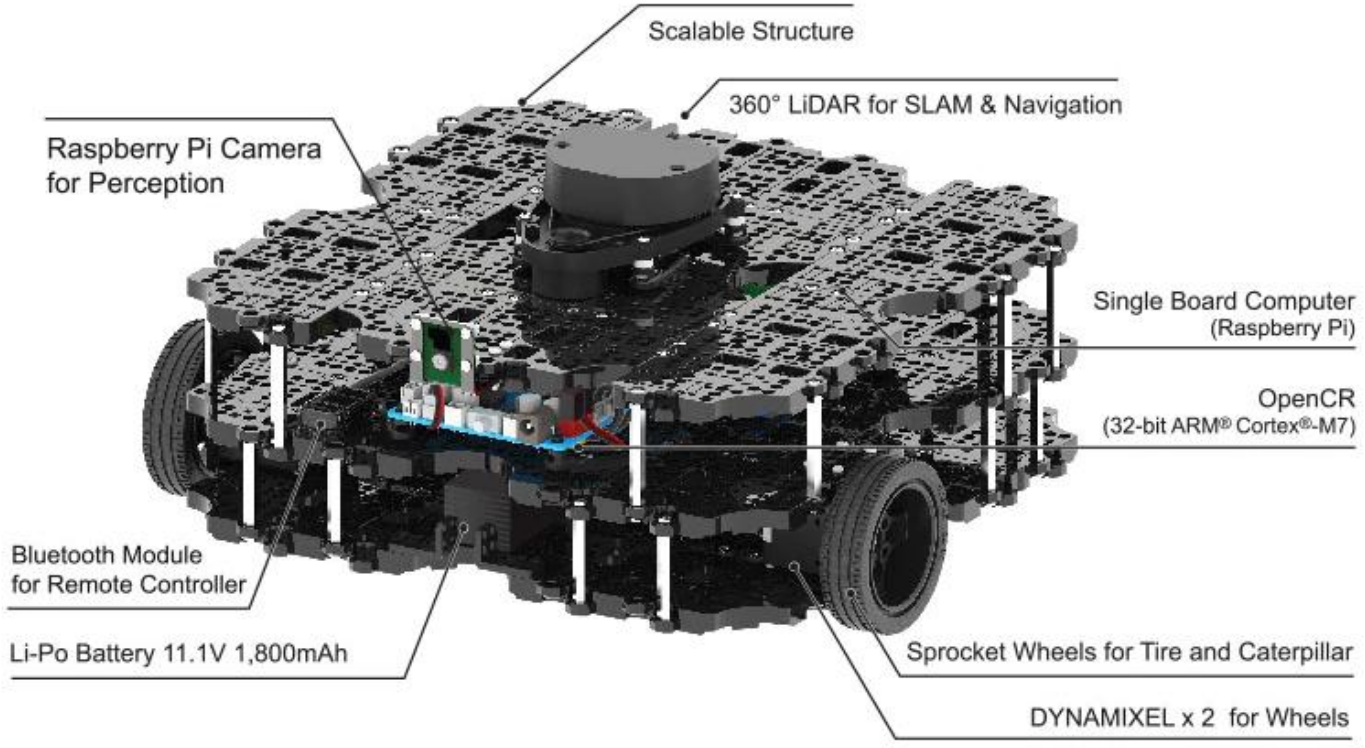
MODULAR ACTUATOR
Easy to assemble, maintain, replace and reconfigure.

OPEN SOURCE SOFTWARE
Variety of open source software for the user.
You can modify downloaded source code and share it with your friends.

OPEN SOURCE HARDWARE
Schematics, PCB Gerber, BOM and 3D CAD data are fully opened to the user.

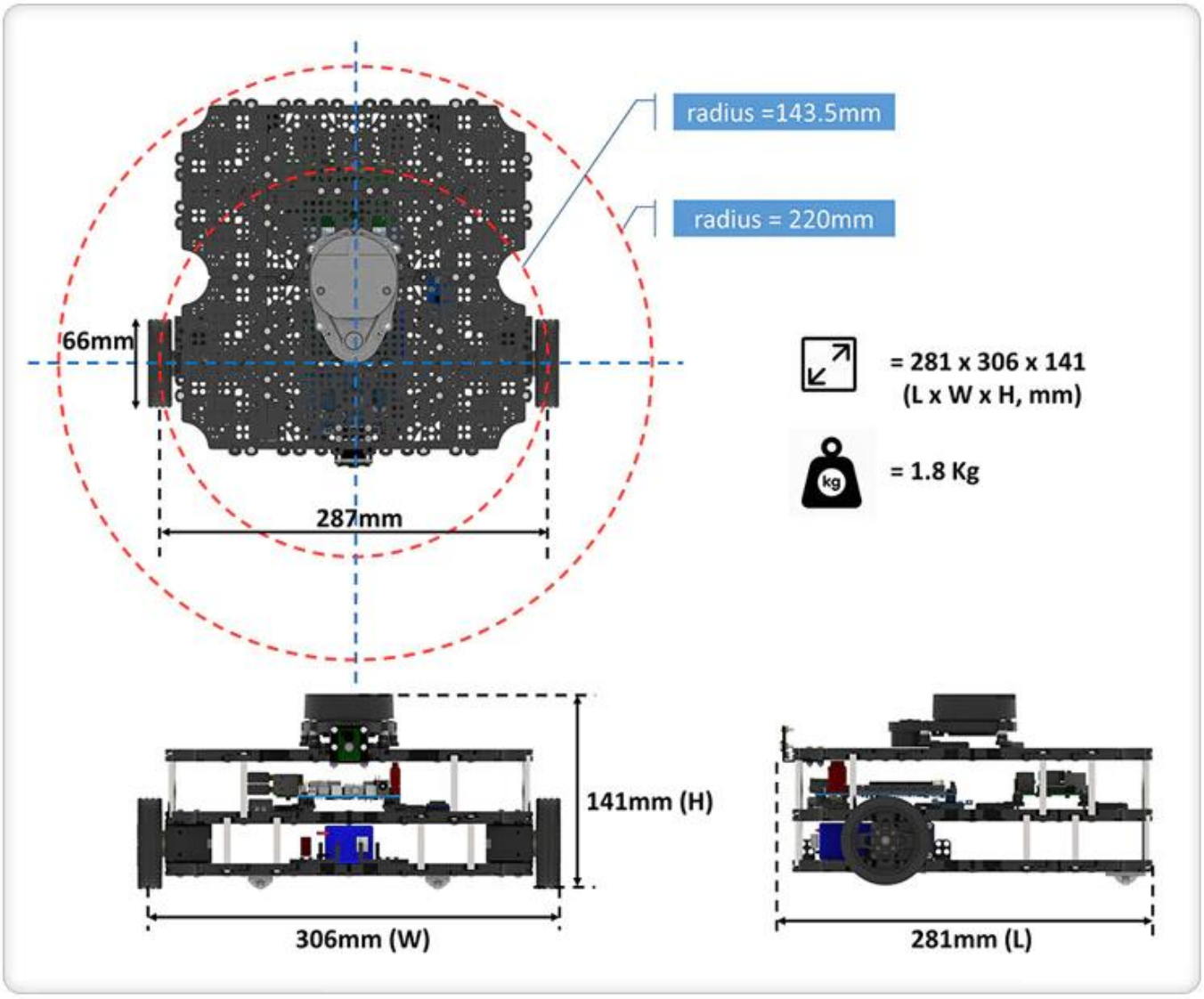
STRONG SENSOR LINEUPS
High utilized Raspberry Pi Camera, Enhanced 360° LiDAR, 9-Axis Inertial Measurement Unit and precise encoder for your robot.

• Main Components



• Specification

Items	Waffle Pi	Burger
Maximum Translational Velocity	0.26m/s	0.22m/s
Maximum Rotational Velocity	1.82rad/s (104.27 deg/s)	2.84rad/s (162.72 deg/s)
Maximum Payload	30kgs	15kgs
Size (L x W x H)	281mm × 306mm × 141mm	138mm × 178mm × 192mm
Weight (+ SBC + Battery + Sensors)	1.8kgs	1kg
Operating Time	About 2hr	About 2hr 30min
Charging Time	About 2hr 30min	About 2hr 30min
DYNAMIXEL	XM430-W210-T	XL430-W250-T
SBC	Raspberry Pi 3	Raspberry Pi 3
Embedded Controller	OpenCR (32-bit ARM® Cortex®-M7)	OpenCR (32-bit ARM® Cortex®-M7)
Sensor	Raspberry Pi Camera 360˚ LiDAR 3-Axis gyroscope 3-Axis accelerometer 3-Axis magnetometer	360˚ LiDAR 3-Axis gyroscope 3-Axis accelerometer 3-Axis magnetometer



10.2. Archivo de código `amcl.launch` para el robot Campero

Autoría: Código creado por Jorge Playán a partir del fichero de AMCL de Turtlebot. Valores empleados para los parámetros por la empresa Robotnik.

```
<?xml version="1.0"?>
<launch>

<!-- Arguments -->

    <arg name="prefix" default="campero_"/>

    <!-- Busca los topics -->
    <arg name="scan_topic"       default="front_laser/scan"/>
    <arg name="map_topic"        default="$(arg prefix)map"/>
    <arg name="global_frame"     default="$(arg prefix)map"/>
    <arg name="odom_frame"       default="$(arg prefix)odom"/>
    <arg name="base_frame"       default="$(arg prefix)base_footprint"/>

    <arg name="odom_model_type"  default="diff"/>

    <!-- Posiciones iniciales -->
    <arg name="x_init_pose"    default="0.0"/>
    <arg name="y_init_pose"    default="0.0"/>
    <arg name="z_init_pose"    default="0.0"/>
    <arg name="a_init_pose"    default="0.0"/>

<!-- AMCL -->

    <!-- Ejecutamos el nodo "amcl" con los siguientes parámetros -->
    <node pkg="amcl" type="amcl" name="amcl" output="screen">

        <!--cambiar el nombre en función del de tu sensor-->
        <remap from="scan" to="$(arg scan_topic)"/>
        <remap from="map" to="$(arg map_topic)"/>

        <param name="use_map_topic" value="false"/>

        <!-- Rango de partículas, ajustar el max
             en función de la potencia del pc -->
        <param name="min_particles"             value="500"/>
        <param name="max_particles"             value="3000"/>

        <!-- Max error entre la distribución estimada y la real -->
        <param name="kld_err" value="0.02"/>
        <param name="kld_z"                     value="0.99"/>

        <!-- Valores para un filtro? -->
```

```
    <param name="update_min_d"              value="0.20"/>
    <param name="update_min_a"              value="0.20"/>

    <!-- Intervalo de re-sampling -->
    <param name="resample_interval"         value="1"/>

    <!-- Tiempo de conversión permitido, s  -->
    <param name="transform_tolerance"       value="0.2"/>

    <!-- Index drop rate, si es 0.0 es que está desactivado -->
    <param name="recovery_alpha_slow"       value="0.00"/>
    <param name="recovery_alpha_fast"       value="0.00"/>

    <!-- Posición inicial -->

    <param name="initial_pose_x"            value="$(arg x_init_pose)"/>
    <param name="initial_pose_y"            value="$(arg y_init_pose)"/>
    <param name="initial_pose_z"            value="$(arg z_init_pose)"/>
    <param name="initial_pose_a"            value="$(arg a_init_pose)"/>

    <!-- Periodo máximo de mostrar la info del scan y el path, en Hz -->
    <param name="gui_publish_rate"          value="50.0"/>


<!-- Parámetros del sensor de distancia -->

    <!-- Valores max de distancia y de rayos usados -->
    <param name="laser_max_range"           value="3.5"/>
    <param name="laser_max_beams"           value="180"/>

    <!-- ___ mixed weight of sensor model? -->
    <param name="laser_z_hit"               value="0.5"/>
    <param name="laser_z_short"             value="0.05"/>
    <param name="laser_z_max"               value="0.05"/>
    <param name="laser_z_rand"              value="0.5"/>

    <!-- Desviación estandar -->
    <param name="laser_sigma_hit"           value="0.2"/>

    <!-- Drop rate para z_short -->
    <param name="laser_lambda_short"        value="0.1"/>
    <param name="laser_likelihood_max_dist" value="2.0"/>

    <!-- Tipo de sensor ("likelihood_field" o "beam") -->
    <param name="laser_model_type"          value="likelihood_field"/>

<!-- Parámetros de odometría -->

    <!-- Forma de conducción, "diff" o "omni" -->
    <param name="odom_model_type"           value="$(arg odom_model_type)"/>

    <!-- Valores estimados de notion noise -->
```

```
        <param name="odom_alpha1"              value="0.1"/>
        <param name="odom_alpha2"              value="0.1"/>
        <param name="odom_alpha3"              value="0.1"/>
        <param name="odom_alpha4"              value="0.1"/>
        <param name="odom_alpha5"              value="0.1"/>

        <!-- Odometry frame -->
        <param name="odom_frame_id"            value="$(arg odom_frame)"/>

        <!-- Robot base frame -->
        <param name="base_frame_id"            value="$(arg base_frame)"/>
        <param name="global_frame_id"          value="$(arg global_frame)"/>

    </node>
</launch>
```

10.3. Archivo de código `move_base.launch` para el robot Campero

Autoría: Código basado en move_base.launch de Turtlebot. Contiene llamadas a ficheros de parámetros creados por la empresa Robotnik.

```
<?xml version="1.0"?>
<launch>

<!-- Arguments -->

    <arg name="prefix" default="campero_"/>
        <!-- Busca los topics -->
    <arg name="cmd_vel_topic" default="move_base/cmd_vel"/>
        <!-- Distinto del tópico de velocidad normal -->
    <arg name="odom_topic" default="robotnik_base_control/odom"/>
    <arg name="global_frame" default="$(arg prefix)map"/>
    <arg name="odom_frame" default="$(arg prefix)odom"/>
    <arg name="base_frame" default="$(arg prefix)base_footprint"/>
    <arg name="scan_topic" default="front_laser/scan"/>
    <arg name="omni" default="true"/>

<!-- Ejecutar move_base -->
    <!-- Ejecutamos el nodo -->
    <node pkg="move_base" type="move_base" respawn="false"
          name="move_base" output="screen">

        <!-- Parámetros que le pasamos al nodo  -->
        <rosparam file=
         "$(find campero_navigation)/config/move_base_params.yaml"
         command="load" />

        <!-- Parámetros de constmaps local y global -->
        <rosparam file=
         "$(find campero_navigation)/config/costmap_common_params.yaml"
         command="load" ns="global_costmap" />
        <rosparam file=
         "$(find campero_navigation)/config/costmap_common_params.yaml"
         command="load" ns="local_costmap" />
        <rosparam file=
         "$(find campero_navigation)/config/local_costmap_params.yaml"
         command="load" />
        <rosparam file=
         "$(find campero_navigation)/config/global_costmap_params_map.yaml"
         command="load" />
```

```
<!-- If para el tipo de ruedas -->
<rosparam if="$(arg omni)"
 file=
 "$(find campero_navigation)/config/teb_local_planner_omni_params.yaml"
 command="load" />
<rosparam unless="$(arg omni)"
 file=
 "$(find campero_navigation)/config/teb_local_planner_diff_params.yaml"
 command="load" />

<!-- Parámetros relativos al sensor láser -->
<param name=
 "global_costmap/obstacle_layer/scan/sensor_frame"
 value="$(arg prefix)front_laser_link"/>
<param name=
 "local_costmap/obstacle_layer/scan/sensor_frame"
 value="$(arg prefix)front_laser_link"/>
<param name=
 "global_costmap/obstacle_layer/scan/topic"
 value="$(arg scan_topic)"/>
<param name=
 "local_costmap/obstacle_layer/scan/topic"
 value="$(arg scan_topic)"/>

<!-- Parámetros de los marcos -->
<param name=
"local_costmap/global_frame"
value="$(arg odom_frame)"/>
<param name=
"local_costmap/robot_base_frame"
value="$(arg base_frame)"/>
<param name=
"global_costmap/global_frame"
value="$(arg global_frame)"/>
<param name=
"global_costmap/robot_base_frame"
value="$(arg base_frame)"/>
<param name=
"DWAPlannerROS/global_frame_id"
value="$(arg odom_frame)"/>
<param name=
"TebLocalPlannerROS/map_frame"
value="$(arg global_frame)"/>

<!-- Específicos de navegación -->
<param name=
"base_local_planner"
value="teb_local_planner/TebLocalPlannerROS" />
<param name=
"controller_frequency"
value="5.0" />
<param name=
```

```
        "controller_patience"
        value="15.0" />

        <!-- Renombrar -->
        <remap from="cmd_vel" to="$(arg cmd_vel_topic)" />
        <remap from="odom" to="$(arg odom_topic)" />
    </node>

</launch>
```

10.4. Archivo de código campero_one_robot_nav.launch para el robot Campero

Autoría: Jorge Playán. Basado en la ejecución de navegación del Turtlebot. Creación del robot basada en "campero_one_robot.launch" por la empresa Robotnik.

```
<?xml version="1.0"?>
<launch>

<!-- Argumentos de entrada -->
    <!-- Nombre del robot -->
    <arg name="id_robot" default="campero"/>

    <!-- Posición inicial -->
    <arg name="x_init_pose" default="0"/>
    <arg name="y_init_pose" default="0"/>
    <arg name="z_init_pose" default="0"/>

    <!-- Xacro utilizada -->
    <arg name="xacro_robot" default="campero_rubber.urdf.xacro"/>

    <!-- Mapa de rviz -->
    <arg name="map_file" default="empty/empty.yaml"/>

    <!-- Tipo de movimiento -->
    <arg name="robot_localization_mode" default="odom"/>
    <arg name="ros_planar_move_plugin" default="false"/>

    <!-- Tipo de brazo -->
    <arg name="3_finger_gripper" default="false"/>

    <!-- Empieza el robot-->

        <!-- Prefijo -->
        <arg name="prefix" value="$(arg id_robot)_"/>

        <!-- Load the URDF into the ROS Parameter Server -->
        <param name="robot_description"
        command="$(find xacro)/xacro
            '$(find campero_description)/robots/$(arg xacro_robot)'
            prefix:=$(arg prefix)
            ros_planar_move_plugin:= $(arg ros_planar_move_plugin)
            --inorder"/>

        <!-- Nodo que publica datos del robot -->
        <node name="robot_state_publisher"
            pkg="robot_state_publisher"
            type="robot_state_publisher"
            respawn="false"
```

```
        output="screen">
    <remap from="/joint_states" to="joint_states" />
        <!-- Cambio de nombre -->
    </node>

    <!-- Spawn del robot en Gazebo -->
    <node name="urdf_spawner_campero_model"
       pkg="gazebo_ros"
       type="spawn_model"
       respawn="false" output="screen"
       args="-x $(arg x_init_pose)
             -y $(arg y_init_pose)
             -z $(arg z_init_pose)
             -J campero_ur10_elbow_joint -2
             -J campero_ur10_shoulder_lift_joint -0.785
             -J campero_ur10_shoulder_pan_joint -0.785
             -J campero_front_ptz_camera_tilt_joint -2
             -J campero_front_ptz_camera_pan_joint 3.14
             -urdf
             -param robot_description
             -model $(arg id_robot)
             -unpause
             " />

    <!-- Empieza nodos de control del robot campero -->
    <include file="$(find campero_control)/launch/campero_control.launch">
        <arg name="prefix" value="$(arg prefix)"/>
        <arg name="sim" value="true"/>
        <arg if="$(arg ros_planar_move_plugin)"
         name="kinematics" value="steel_omni"/>
        <arg name="ros_planar_move_plugin"
         value="$(arg ros_planar_move_plugin)"/>
        <arg name="launch_robot_localization" value="false"/>
        <arg name="3_finger_gripper" value="$(arg 3_finger_gripper)"/>
    </include>

</launch>
```

10.5. Archivo de código campero_nav.launch para el robot Campero

Autoría: Jorge Playán. Basado en archivos de ejecución global de Turtlebot y de la empresa Robotnik para Campero.

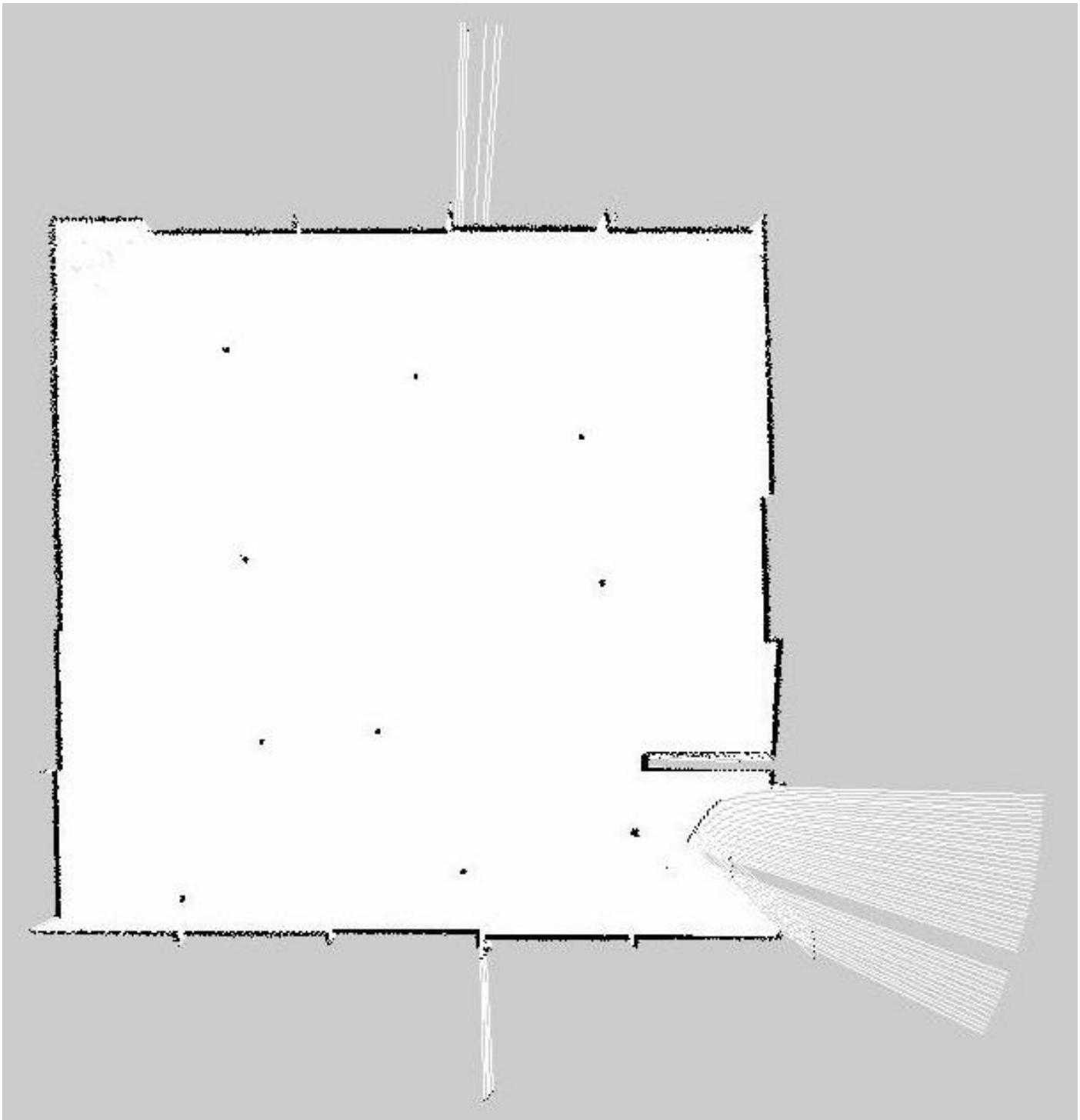
```
<?xml version="1.0"?>
<launch>

<!-- Argumentos de entrada -->
    <!-- Nombre del robot -->
    <arg name="id_robot" default="campero"/>

    <!-- Posición inicial -->
    <arg name="x_init_pose" default="0"/>
    <arg name="y_init_pose" default="0"/>
    <arg name="z_init_pose" default="0"/>

    <!-- Ejecución de Gmapping, idealmente deberíamos ejecutarlo al principio
         y manejar manualmente el robot por el entorno para realizar un mapa y,
         posteriormente, añadirlo -->
    <arg name="gmapping"        default="false"/>

    <!-- Xacro utilizada -->
    <arg name="xacro_robot" default="campero_mecanum.urdf.xacro"/>

    <!-- Mapa de rviz -->
    <arg name="map_file" default="empty/map_empty.yaml"/>

    <!-- Tipo de movimiento y localización -->
    <arg name="robot_localization_mode" default="odom"/>
    <arg name="ros_planar_move_plugin" default="true"/>

    <!-- Tipo de brazo -->
    <arg name="3_finger_gripper" default="false"/>

    <!-- Prefijo para encontrar topics -->
    <arg name="prefix" value="$(arg id_robot)_"/>

<!-- Ejecutar el robot campero -->
    <!-- Colocamos todos los nodos en el grupo del Robot -->
    <group ns="$(arg id_robot)">

        <!-- El Robot -->
        <include file=
         "$(find campero_gazebo)/launch/campero_one_robot_nav.launch">

            <!-- Argumentos -->
                <!-- Nombre -->
```

```
        <arg name="id_robot" value="$(arg id_robot)"/>

        <!-- PosiciÃ³n inicial -->
        <arg name="x_init_pose" value="$(arg x_init_pose)"/>
        <arg name="y_init_pose" value="$(arg y_init_pose)"/>
        <arg name="z_init_pose" value="$(arg z_init_pose)"/>

        <!-- Xacro -->
        <arg name="xacro_robot" value="$(arg xacro_robot)"/>

        <!-- Mapa -->
        <arg name="map_file" value="$(arg map_file)"/>

        <!-- Movimiento específico para ruedas "omni" -->
             <arg name="ros_planar_move_plugin"
             value= "$(arg ros_planar_move_plugin)"/>

        <!-- Brazo -->
             <arg name="3_finger_gripper"
             value="$(arg 3_finger_gripper)"/>
</include>

<!-- Ejecutar MapServer-->
<include file="$(find campero_localization)/launch/map_server.launch">

    <!-- Para poder buscar topics -->
    <arg name="prefix" value="$(arg prefix)"/>

    <!-- Mapa -->
    <arg name="map_file" value="$(arg map_file)"/>
</include>

<!-- Ejecutar AMCL-->
<include file="$(find campero_localization)/launch/mi_amcl.launch">

    <!-- Para poder buscar topics -->
    <arg name="prefix" value="$(arg prefix)"/>

    <!-- Tipo ruedas (diff o omni) en función
         del tipo de movimiento -->
    <arg if="$(arg ros_planar_move_plugin)"
            name="odom_model_type" value="omni"/>
    <arg unless="$(arg ros_planar_move_plugin)"
            name="odom_model_type" value="diff"/>

    <!-- Posicion inicial -->
    <arg name="x_init_pose" value="$(arg x_init_pose)"/>
    <arg name="y_init_pose" value="$(arg y_init_pose)"/>
    <arg name="z_init_pose" value="$(arg z_init_pose)"/>
</include>

<!-- Ejecutar MoveBase -->
```

```
        <include file="$(find campero_navigation)/launch/move_base.launch">

            <!-- Para poder buscar topics -->
            <arg name="prefix" value="$(arg prefix)"/>

            <!-- Tipo de ruedas -->
            <arg name="omni" value="$(arg ros_planar_move_plugin)"/>
        </include>

        <!-- Ejecutar gmapping -->
        <include if="$(arg gmapping)"
            file="$(find campero_localization)/launch/slam_gmapping.launch">

            <arg name="prefix" value="$(arg prefix)"/>
        </include>


    <!-- Fin del grupo -->
    </group>

    <!-- Activa un PID que simula la gravedad-->
    <rosparam command="load"
        file=
        "$(find campero_control)/config/gazebo/gazebo_controller_omni.yaml"/>

    <!-- Ejecutar Gazebo y rviz-->
    <include file="$(find campero_gazebo)/launch/gazebo_rviz.launch">

        <!-- Ejecutar rviz -->
        <arg name="launch_rviz" value="true"/>

        <!-- Mundo de Gazebo-->
        <arg name="world"
            value="$(find campero_gazebo)/worlds/campero_outside.world"/>

        <!-- Setting debug-->
        <arg name="debug" value="false"/>
    </include>
</launch>
```

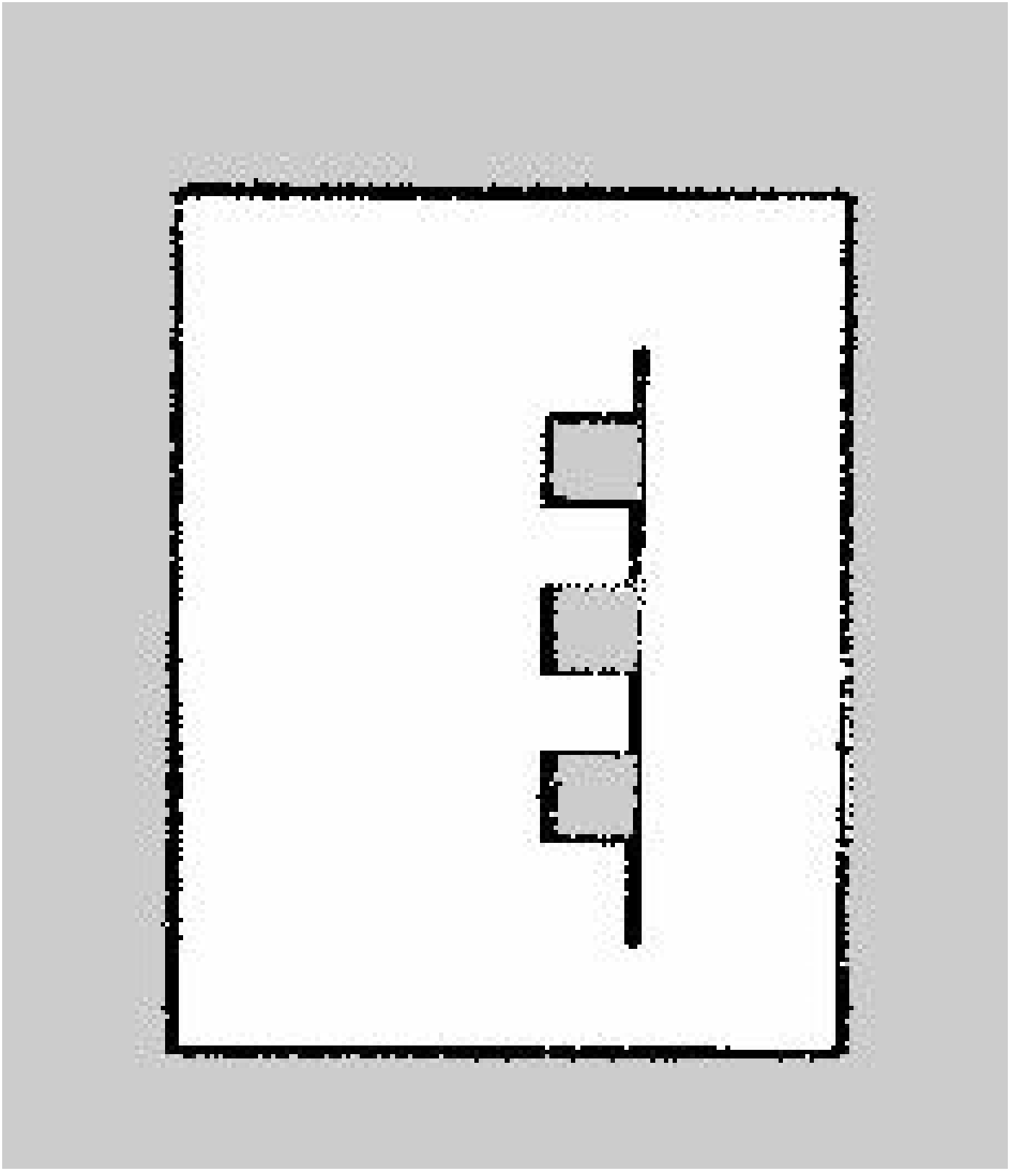
10.6. Archivo de mapa del entorno exterior `mymap.pgm`

Autoría: Jorge Playán creó el mapa. El entorno de simulación fue creado por la empresa Robotnik.



10.7. Archivo de mapa del entorno interior mapaInt.pgm

Autoría: Jorge Playán creó el mapa. El entorno de simulación fue creado por la empresa Robotnik.



11. Bibliografía

Batard, P. (5 de Agosto de 2019). *Rufus*. Obtenido de https://rufus.ie/

COMMANDIA. (15 de septiembre de 2019). *Robótica móvil colaborativa de objetos deformables en aplicaciones industriales*. Obtenido de http://commandia.unizar.es/es/lo-basico-de-commandia/

Open Source Robotics Foundation. (4 de agosto de 2019a). *ROS*. Obtenido de https://www.ros.org/

Open Source Robotics Foundation. (5 de Agosto de 2019b). *Costmap_2d*. Obtenido de http://wiki.ros.org/costmap_2d

Open Source Robotics Foundation. (5 de Agosto de 2019c). *TurtleBot*. Obtenido de http://wiki.ros.org/Robots/TurtleBot

Open Source Robotics Foundation. (5 de Agosto de 2019d). *Kinetic Installation Ubuntu*. Obtenido de http://wiki.ros.org/kinetic/Installation/Ubuntu

Open Source Robotics Foundation. (5 de agosto de 2019e). *rviz*. Obtenido de http://wiki.ros.org/rviz

Open Source Robotics Foundation. (28 de Agosto de 2019g). *geometry_msgs/PoseStamped Message*. Obtenido de http://docs.ros.org/melodic/api/geometry_msgs/html/msg/PoseStamped.html

Open Source Robotics Foundation. (28 de Agosto de 2019h). *move_base_msgs/MoveActionGoal Message*. Obtenido de http://docs.ros.org/fuerte/api/move_base_msgs/html/msg/MoveBaseActionGoal.html

Open Soure Robotics Foundation. (5 de agosto de 2019f). *Gazebo. Robot Simulation Made Easy*. Obtenido de http://gazebosim.org/

Robotis. (4 de agosto de 2019). *Turtlebot 3 Waffle Pi*. Obtenido de http://www.robotis.us/turtlebot-3-waffle-pi/

Robotis. (5 de Agosto de 2019b). *Turtlebot 3 PC Setup*. Obtenido de http://emanual.robotis.com/docs/en/platform/turtlebot3/pc_setup/#pc-setup

Ubuntu releases. (5 de Agosto de 2019). *Ubuntu 16.04.06 LTS (Xenial Xersus)*. Obtenido de http://releases.ubuntu.com/16.04/

Wikipedia contributors. (4 de agosto de 2019a). *Wikipedia, the free encyclopedia*. Obtenido de Middleware: https://en.wikipedia.org/w/index.php?title=Middleware&oldid=902648688

Wikipedia Contributors. (4 de agosto de 2019b). *Wikipedia, The Free Encyclopedia*. Obtenido de Robot Operating System: https://en.wikipedia.org/w/index.php?title=Robot_Operating_System&oldid=904579289

Wikipedia Contributors. (5 de Agosto de 2019c). *Wikipedia, The Free Encyclopedia*. Obtenido de Motion Planning: https://en.wikipedia.org/wiki/Motion_planning

Wyrobek, K. (31 de octubre de 2017). The Origin Story of ROS, the Linux of Robotics. *IEEE Spectrum*. Obtenido de https://spectrum.ieee.org/automaton/robotics/robotics-software/the-origin-story-of-ros-the-linux-of-robotics