COMILLAS

UNIVERSIDAD PONTIFICIA

ICAI | ICADE | CIHS

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

Trabajo Fin de Master

DESARROLLO DE UN GEMELO DIGITAL PARA EL ROBOT MÓVIL TURTLEBOT3 USANDO ROS 2

Autor: Ana Berjón Valles

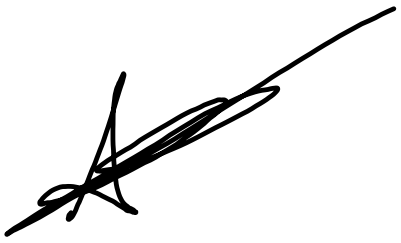
Director: Jaime Boal Martín-Larrauri

MADRID

Febrero 2022

Este trabajo se ha maquetado con LaTeX y se ha compilado en TeXmaker empleando la distribución MacTeX-2013. Las familias de fuentes empleadas son Bitstream Charter, Utopia, Bookman, and Computer Modern. A menos que se indique lo contrario, todas las figuras fueron creadas por el autor utilizando Microsoft Visio®, Adobe Illustrator®, y MATLAB®.

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título ”Desarrollo de un gemelo digital para el robot móvil Turtlebot3 usando ROS 2” en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2021/2022 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.



Fdo.: Ana Berjón Valles Fecha: 07 / 02 / 22

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Firmado digitalmente por BOAL MARTIN LARRAURI JAIME - 05304600H
Fecha: 2022.02.07 23:49:30 +01'00'

Fdo.: Jaime Boal Martín-Larrauri Fecha: ……/ ……/ ……

COMILLAS

UNIVERSIDAD PONTIFICIA

ICAI | ICADE | CIHS

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

Trabajo Fin de Master

DESARROLLO DE UN GEMELO DIGITAL PARA EL ROBOT MÓVIL TURTLEBOT3 USANDO ROS 2

Autor: Ana Berjón Valles

Director: Jaime Boal Martín-Larrauri

MADRID

Febrero 2022

Este trabajo se ha maquetado con LaTeX y se ha compilado en TeXmaker empleando la distribución MacTeX-2013. Las familias de fuentes empleadas son Bitstream Charter, Utopia, Bookman, and Computer Modern. A menos que se indique lo contrario, todas las figuras fueron creadas por el autor utilizando Microsoft Visio®, Adobe Illustrator®, y MATLAB®.

Yera distintu en 1932...
o 1933

Índice de figuras

Índice de tablas

Resumen

DESARROLLO DE UN GEMELO DIGITAL PARA EL ROBOT MÓVIL TURTLEBOT3 USANDO ROS 2

Autor: Berjón Valles, Ana.
Director: Boal Martín-Larrauri, Jaime.
Entidad colaboradora: ICAI – Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

Los robot móviles están cada vez más presentes en el ámbito industrial. En este contexto de automatización, se desarrolla el presente proyecto que busca simular un sistema de navegación de forma que un robot sea capaz de realizar el mapa y navegar autónomamente por el entorno. Para llevar a cabo este desarrollo se ha empleado un entorno virtualizado, que permite realizar pruebas de algoritmos sin riesgo y de forma mucho más eficiente. El sistema propuesto logra su objetivo siempre que se cuente con los recursos necesarios para la ejecución de la simulación.

Palabras clave: Gemelo digital, Robots móviles, ROS 2 Foxy, Gazebo, SLAM

1. Introducción

La Cuarta Revolución Industrial se caracteriza por haber traído un mayor grado de automatización a las empresas, haciendo que aparezcan nuevos elementos en las plantas industriales. Entre estos dispositivos, destacan los robots móviles, concretamente los AMR (Autonomous Mobile Robot), capaces de procesar su entorno, reconocerlo y desplazarse hasta el punto de destino calculando la mejor ruta y evitando obstáculos.

2. Metodología

En este contexto se ha elaborado el presente proyecto que busca simular el comportamiento de un robot AMR. Estos robots, a diferencia de los AGV (*Automated Guided Vehicle*), son capaces de analizar su entorno y reconsiderar la ruta a seguir en caso de encontrar algún obstáculo inesperado [**1**]. Para determinar el robot a emplear se ha llevado a cabo un estudio de mercado, con la intención de seleccionar un modelo adecuado para el ámbito docente que sea lo más parecido a lo empleado en entornos industriales. En la siguiente tabla se recoge un resumen de la comparativa realizada, donde resultó elegido el modelo Turtlebot3 Burger.

La simulación se ha llevado a cabo en un entorno virtualizado, utilizando una imagen de Ubuntu 20.04 sobre VirtualBox. En esta máquina virtual está instalado ROS 2, [**2**] un conjunto de *frameworks open-source* para el desarrollo de *software* para robots. La versión empleada es ROS 2 Foxy, lanzada en junio de 2020 [**3**]. El simulador, Gazebo es uno de los más utilizados en la industria.

	Dingo-O	Dingo-D	4WD Pro	Khepera IV	Turtlebot3 Waffle Pi	Turtlebot3 Buger	Osoyoo
Cinemática	4 ruedas suecas	4 ruedas	- 4 ruedas - 2 ruedas + 2 ruedas locas - Ruedas de oruga con aletas	2 ruedas + 2 ruedas esféricas	2 ruedas + 2 ruedas esféricas	2 ruedas + rueda esférica	4 ruedas suecas
Peso	16 kg	20 kg	11 kg	540 g	1,8 kg	1 kg	1,9 kg
Máx. Carga	20 kg.	25 kg.	68 kg.	2 kg.	30 kg.	15 kg.	-
Baterías	Ion-Litio	Ion-Litio	Ion-Litio	LiPo	LiPo	LiPo	Pilas de tipo 18650
Sensores Integrados	-	-	Cámara VGA Micrófono	Infrarrojos Ultrasónicos Micrófonos	360 LDS-01 Giróscopo Acelerómetro Magnetómetro	360 LDS-01 Giróscopo Acelerómetro Magnetómetro	Ultrasonidos
ROS	Sí	Sí	No	Sí	Sí	Sí	No
Precio	15789.47 €	10105.26 €	7640.9 €	3240 €	1199 €	499 €	136 . €

Tabla 1. Tabla comparativa de los robots analizados.

3. Resultados

En este proyecto se ha desarrollado una simulación del robot Turtlebot3 Burger en entornos diseñados a partir de unos modelos para CoppeliaSim y desarrollados utilizando el modo constructor de Gazebo. Se han desarrollado dos laberintos distintos, uno de ellos con dos posibles caminos. La siguiente imagen muestra los resultados.

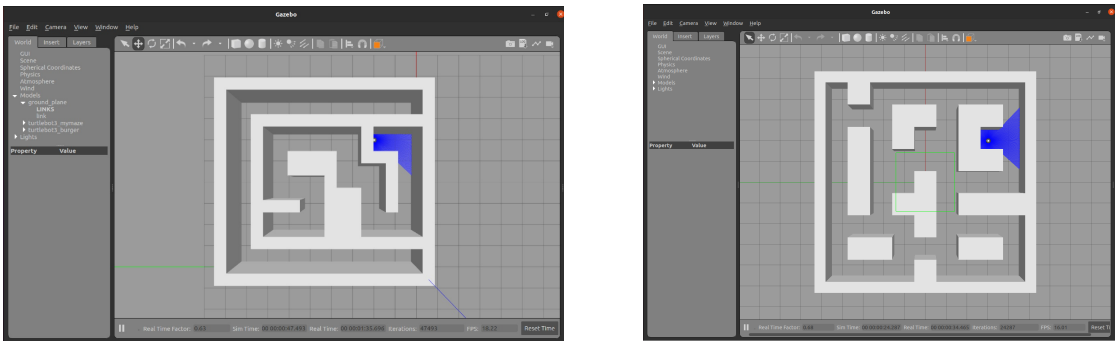


Figura 1. Laberintos desarrollados para llevar a cabo las simulaciones.

Tras generar los mundos e introducir el modelo del robot, el siguiente paso es conseguir que este sea capaz de generar un mapa de su entorno. Se emplearon algoritmos SLAM (*Simultaneous Location and Mapping*) [**4**] [**5**]. Para ello se comenzó utilizando la librería Cartographer, que presentaba buenos resultados con los trazados más sencillos, pero no era capaz de generar el mapa en entornos más complejos.

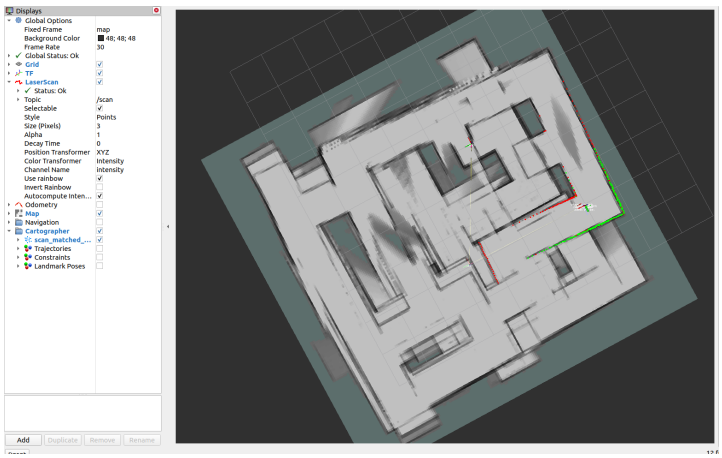


Figura 2. Resultado erróneo con la librería Cartographer.

Debido a estos problemas, se empleó la herramienta *slam_toolbox*. Con este paquete, combinado con el módulo Navigation2, se consiguieron muy buenos resultados. Se logró elaborar los tres mapas con una única vuelta al trazado. En el proceso de creación del mapa, intervienen diversos nodos, pero el usuario solo operará con tres pantallas distintas: Gazebo corriendo la simulación, RViz mostrando el mapa que se va generando y la terminal del `teleop` desde el que se controlará el movimiento del robot. La figura de la siguiente página muestra el proceso de creación del mapa para uno de los entornos creados.

Los resultados finales, mostrados en la siguiente figura, presentan el trazado de los tres circuitos con gran definición y exactitud.

(a) **(b)** **(c)**

Figura 4. Mapa de los tres entornos desarrollados.

Tras generar los mapas, se buscó conseguir la navegación autónoma del robot en estos entornos. Para lograrlo, se cargó el mapa del trazado sobre el simulador RViz, de forma que es necesario indicar el punto de inicio y el de destino y el algoritmo calculará la ruta a seguir. Este proceso es iterativo, por lo que el sistema irá recalculando el camino a medida que avanza y de esta forma podrá evitar obstáculos. La imagen mostrada a continuación representa el camino diseñado por el Turtlebot3 para alcanzar como punto de destino el interior de la pieza con forma de “C”.

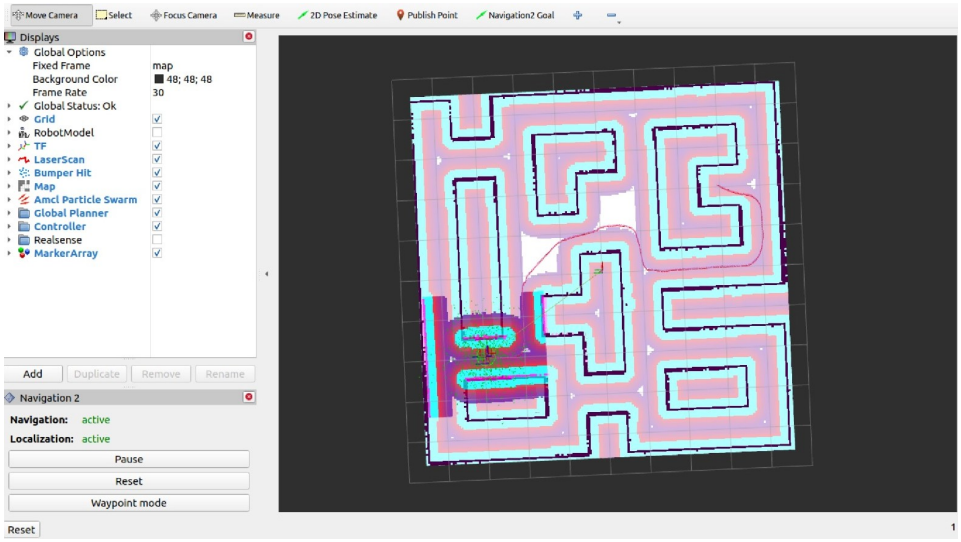


Figura 5. Ruta elaborada por el Turtlebot3 para llegar al punto de destino.

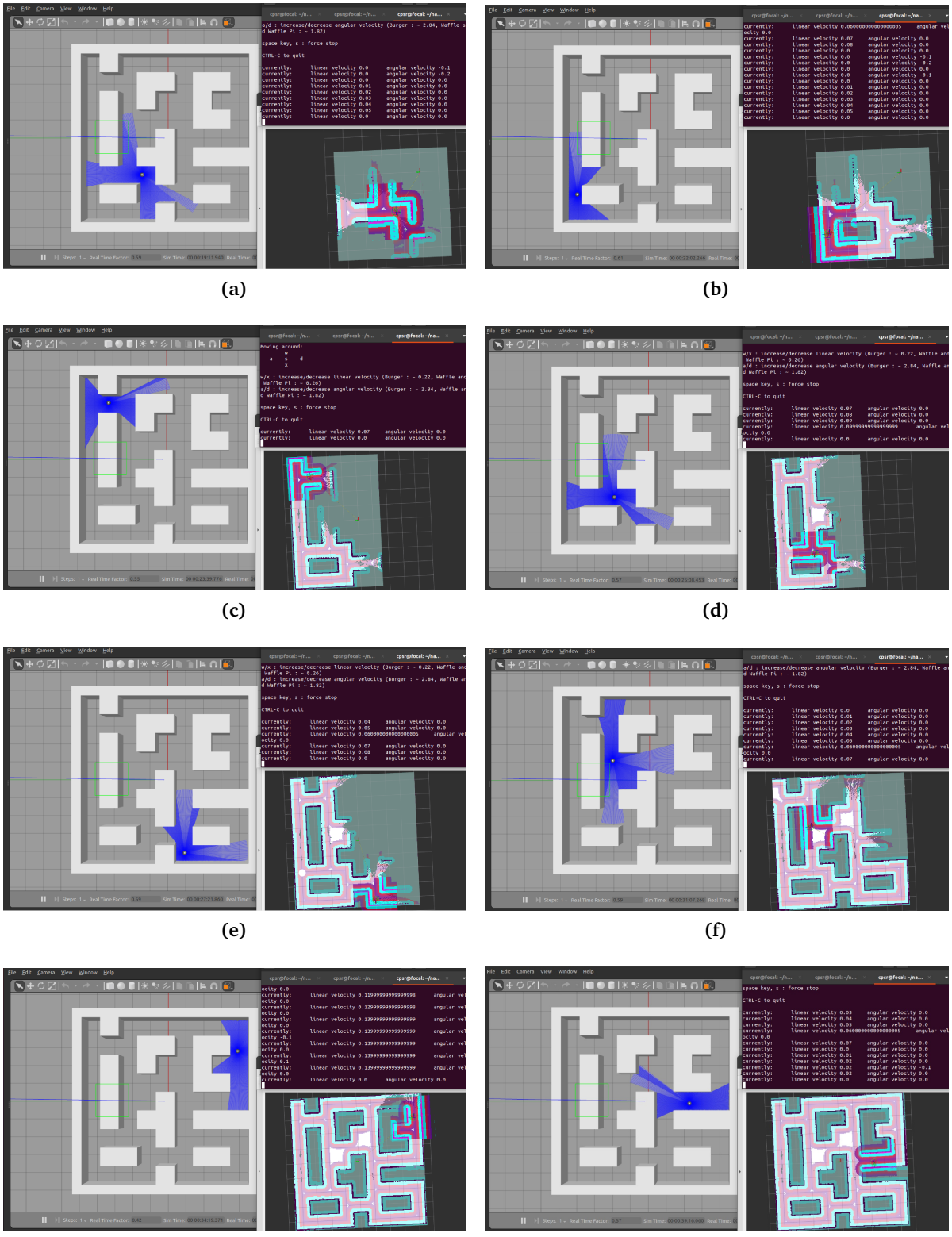


Figura 3. Proceso de creación del mapa del laberinto.

Adicionalmente, se ha generado un escenario modificado del laberinto en el que se introducen obstáculos no recogidos en la elaboración del mapa. Con esta situación se ha comprobado la capacidad del modelo para recalcular la ruta a seguir evitando estos nuevos elementos.

Para mejorar la experiencia del usuario, se ha generado un paquete de ROS con todos los archivos involucrados en el lanzamiento de los distintos escenarios. De esta forma, es posible levantar todos los paquetes con la ejecución de un simple comando.

5. Conclusiones

Se ha diseñado un sistema capaz de simular un modelo digital del robot Turtlebot3 Burger en distintos escenarios utilizando ROS 2. El modelo es capaz de generar un mapa de su entorno empleando técnicas de SLAM y navegar por dicho entorno utilizándolo.

Este proyecto consigue satisfacer los objetivos propuestos, pero existen puntos de mejora. Entre ellos, destaca la necesidad de encontrar una solución al problema generado por el alto consumo de recursos de Gazebo. Para ello, se propone adaptar el modelo .urdf diseñado en este proyecto para que sea capaz de publicar la información recibida en los *topics* correspondientes a partir de su ejecución sobre CoppeliaSim.

6. Referencias

[1] EDS Robotics, «AGV vs AMR ¿Qué les diferencia? ¿Cuál es mejor?,» 14-01-2021. [En línea]. Available: https://www.edsrobotics.com/blog/agv-vs-amr-diferencias/. [Último acceso: 15-11-2021].

[2] A. Dattalo, «ROS/Introducción,» OpenRobotics, 08-08-2018. [En línea]. Available: https://wiki.ros.org/ROS/Introduction. [Último acceso: 29-01-2022].

[3] Open Robotics, «ROS 2 Documentation,» 2021. [En línea]. Available: https://docs.ros.org/en/foxy/index.html. [Último acceso: 01-11-2021].

[4] H. Durrant-Whyte y T. Bailey, «Simultaneous localisation and mappint (SLAM): Part I,» IEEE Robotics and Automation MAgazine, vol. 13, nº 2, pp. 99-110, 2006.

[5] T. Bailey y H. Durrant-Whyte, «Simultaneous localisation and mapping (SLAM): Part II,» IEEE Robotics and Automation Magazine, vol. 13, nº 3, pp. 108-117, 2006.

Abstract

DIGITAL TWIN DEVELOPMENT FOR THE TURTLEBOT3 MOBILE ROBOT USING ROS 2

Author: Berjón Valles, Ana
Supervisors: Boal Martín-Larrauri, Jaime
Collaborating Entity: ICAI – Universidad Pontificia Comillas

ABSTRACT

Mobile robots are increasingly used in the industrial sector. In this context of automation, this project aims to simulate a navigation system in which a robot is able to map and navigate autonomously through the environment. This project has been developed in a virtualized setup, allowing safer and better algorithm testing. The proposed solution achieves its goal provided the necessary resources for the execution of the simulation.

Keywords: Digital Twin, Mobile Robots, ROS 2 Foxy, Gazebo, SLAM

1. **Introduction**
   The Fourth Industrial Revolution is characterized by having brought a higher degree of automation to companies, causing new elements to appear in industrial plants. Among these elements, mobile robots stand out, specifically the AMR (Autonomous Mobile Robot), capable of processing their environment, recognizing it and going to the destination point, calculating the best route and avoiding obstacles.

2. **Methodology**
   In this context, the present project has been developed to simulate the behaviour of an AMR robot. These robots, unlike AGV (Automated Guided Vehhicle) are able to recognize the environment and reconsider the predesigned route in case an unexpected obstacle came out [**1**]. In order to determine the robot to be used, a market research has been conducted. The intention is to select a suitable model for the educational environment that is as similar as possible to the ones used in the industrial world. The below table includes the results of this market research in which Turtlebot3 Burger was chosen.

   The simulation has been performed in a virtualized environment using an Ubuntu 20.04 on VirtualBox. In this virtual machine, ROS 2 is installed, which is a set of open-source frameworks for the development of robotic software [**2**]. The version used is ROS 2 Foxy, released in June 2020 [**3**]. The simulator ussed is Gazebo, one of the standards of the industry.

3. **Results**
   In this project a Turtlebot3 Burger simulation has been developed. The environments to run

	Dingo-O	Dingo-D	4WD Pro	Khepera IV	Turtlebot3 Waffle Pi	Turtlebot3 Buger	Osoyoo
Kinematics	4 swedish wheels	4 wheels	- 4 wheels - 2 wheels + 2 idlers - endless tracks with fins	2 wheels + 2 spherical wheels	2 wheels + 2 spherical wheels	2 wheels + spherical wheel	4 swedish wheels
Weight	16 kg	20 kg	11 kg	540 g	1,8 kg	1 kg	1,9 kg
Max. Load	20 kg	25 kg	68 kg	2 kg	30 kg	15 kg	-
Batteries	Ion-lithium	Ion-lithium	Ion-lithium	LiPo	LiPo	LiPo	Batteries 18650
Integrated Sensors	-	-	VGA Camera Microphone	Infrared Ultrasonic Microphones	360 LDS-01 Giroscope Accelerometer Magnetometer	360 LDS-01 Giroscope Accelerometer Magnetometer	Ultrasound
ROS	Yes	Sí	No	Yes	Yes	Yes	No
Price	15789.47€	10105.26€	7640.9€	3240€	1199€	499€	136 .€

Table 2. Comparative table of the characteristics of the analyzed robots.

it have been designed from models for CoppeliaSim, and developed using Gazebo's builder mode. Two different mazes have been designed, one of them having two different paths. The followind image shows the resulting mazes.

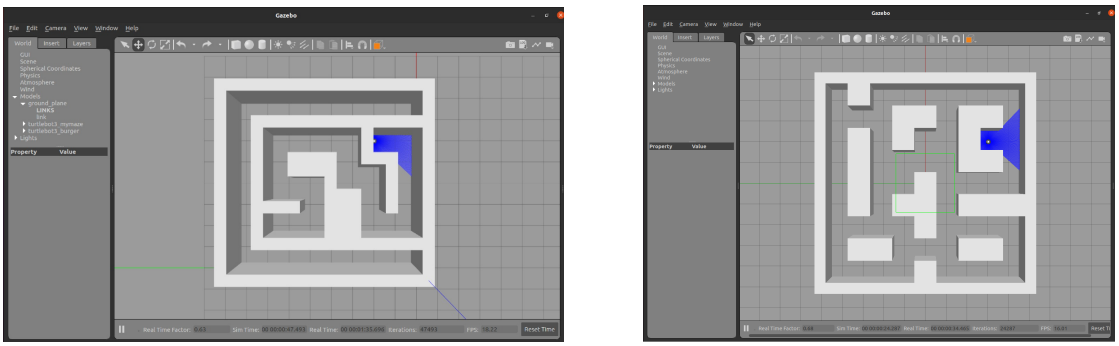


Figure 6. Mazes designed for the simulation.

After generating the worlds and combining them with the virtual model of the robot, the following step is to make the robot capable of generating a map of the surroundings. SLAM algorithms were used. To do this, we started using Cartographer, a library that performed well with the simplest layouts, but was not able to generate a map in more complex environments.

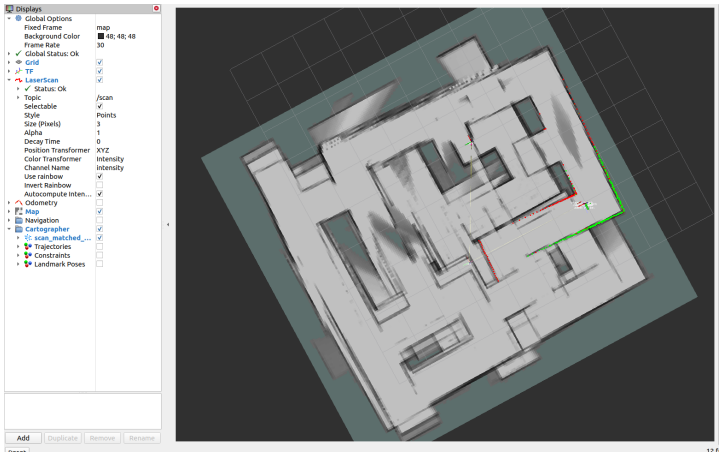
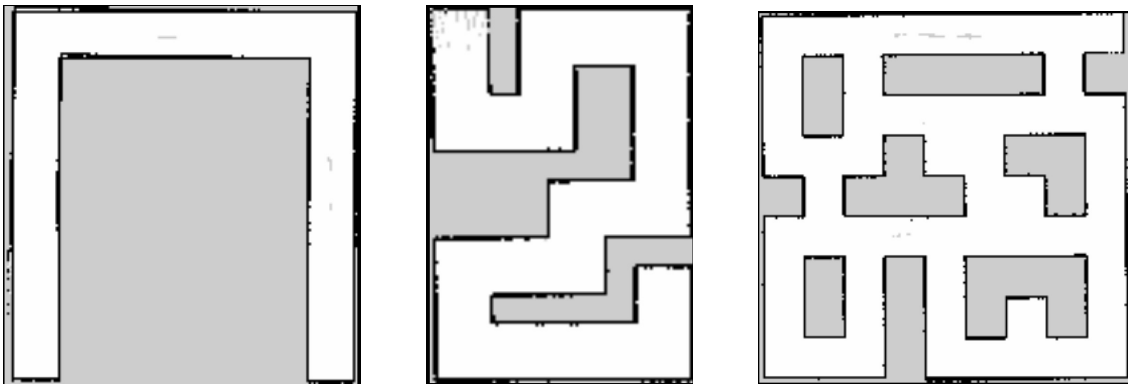


Figure 7. Wrong result using Cartographer library.

Due to these performance problems, slam_toolbox was used. With this package, in combination with Navigation2, the quality of the results achieved was very high. All three maps were produced with a single round-trip. In the process of creating the map, several

nodes are involved, but the user will only operate with three different interfaces: Gazebo running hte simulation, RViz showing the map in progress and the terminal of the `teleop` node from which the robot is controlled. The figure on next page shows the process of creating the map for one of the environments.

The final results, shown in the figure below, present the layout of the three circuits with high definition and accuracy.



(a) **(b)** **(c)**

Figure 9. Maps created for the different predesigned paths.

After generating the maps, the final goal was to achieve autonomous navigation through these environments. Maps wereloaded on the RViz simulator, the user should indicate the starting point and the destination, and with this information, the algorithm will calculate the route. This process is iterative, tehrefore, the system will recalculate the path as it progresses, making possible to avoid unexpected obstacles. The image shown below represents the designated path to reach its destination: the inner part of the C-shaped element.

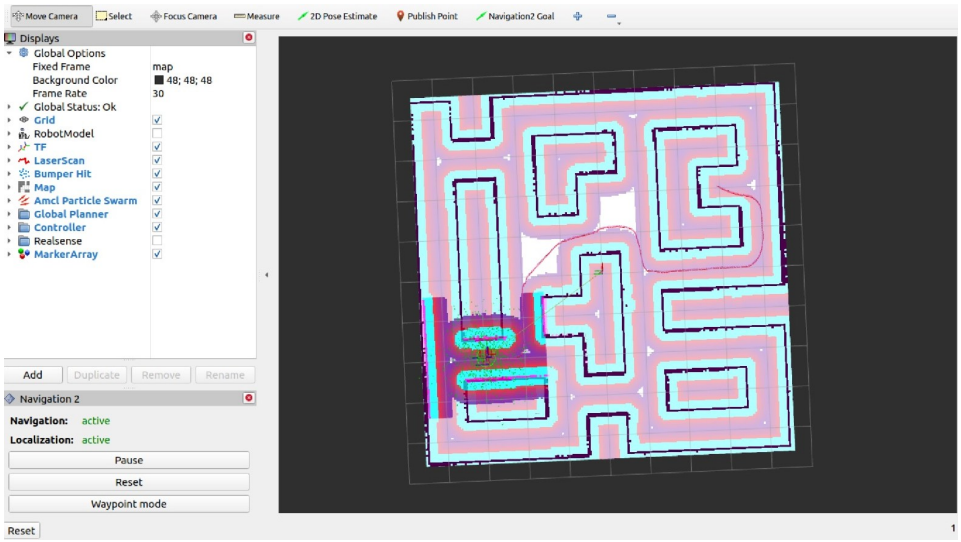


Figure 10. Best route to reach the destination point.

Additionally, a modified scenario of the maze has been generated in which obstacles not included in the elaboration of the map are introduced. With this situation, the ability of the model to recalculate the route has been successfuly tested.

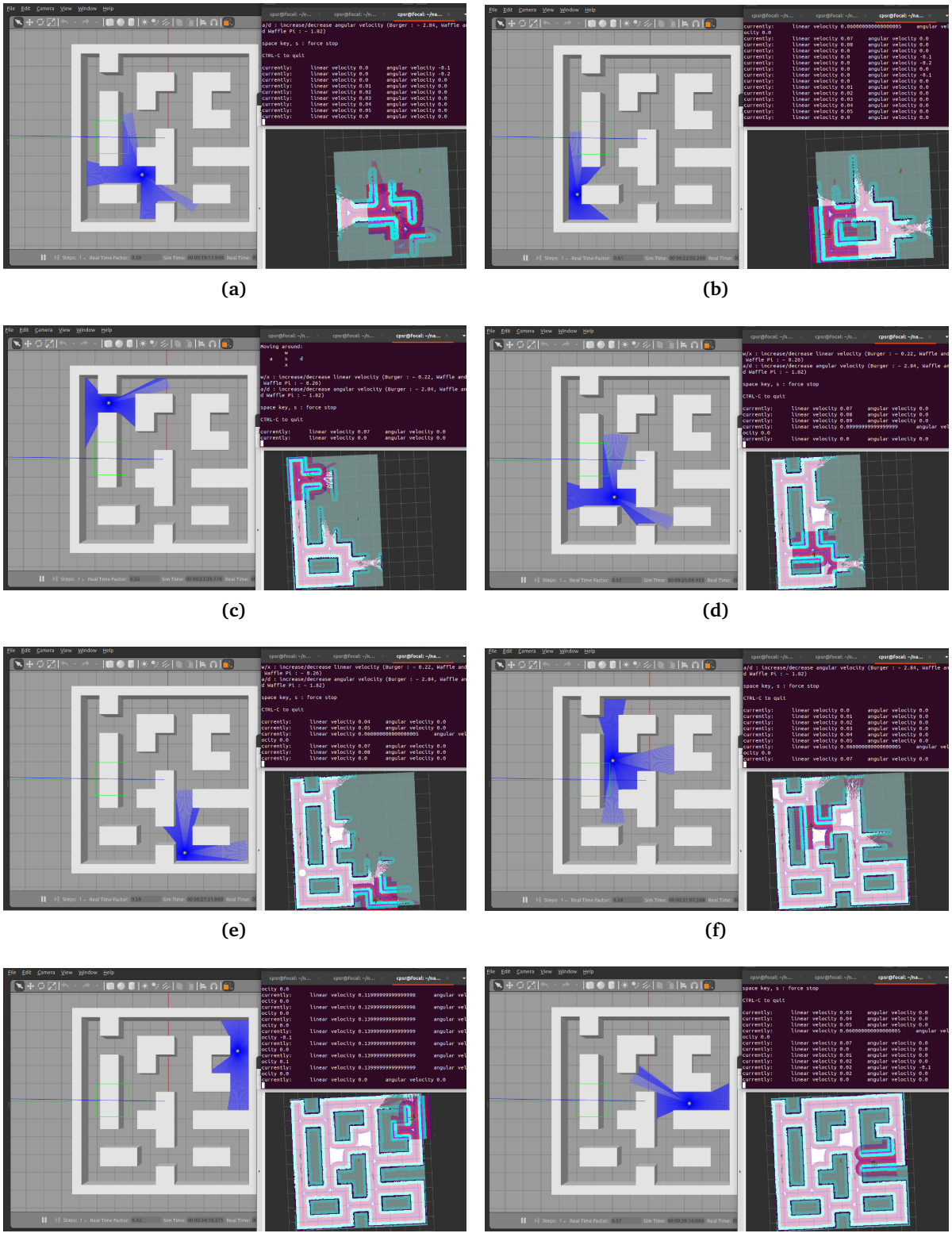


Figure 8. Mapping process.

To improve user experience, a ROS package has been generated including all files involved in launching different scenarios. This allows the user to raise all packages involved with a single command execution.

5. Conclusions

A system has been designed to simulate a digital twin of the Turtlebot3 Burger robot in different scenarios using ROS 2. The module is able to generate a map of its environment using SLAM techniques and then navigate through it.

This project has achieved the proposed objectives, however there is room for improvement. There is a need to find a solution to the problem generated by Gazebo's high consumption of resources. For this, it is proposed to adapt the .urdf model designed in this project to be able to publish the information received in the corresponding topics from its execution on CoppeliaSim.

6. References

[1] EDS Robotics, «AGV vs AMR ¿Qué les diferencia? ¿Cuál es mejor?,» 14-01-2021. [Online] Available: https://www.edsrobotics.com/blog/agv-vs-amr-diferencias/. [Last Access: 15-11-2021].

[2] A. Dattalo, «ROS/Introducción,» OpenRobotics, 08-08-2018. [Online]. Available: https://wiki.ros.org/ROS/Introduction. [Last Access: 29-01-2022].

[3] Open Robotics, «ROS 2 Documentation,» 2021. [Online]. Available: https://docs.ros.org/en/foxy/index.html. [Last Access: 01-11-2021].

[4] H. Durrant-Whyte y T. Bailey, «Simultaneous localisation and mappint (SLAM): Part I,» IEEE Robotics and Automation MAgazine, vol. 13, nº 2, pp. 99-110, 2006.

[5] T. Bailey y H. Durrant-Whyte, «Simultaneous localisation and mapping (SLAM): Part II,» IEEE Robotics and Automation Magazine, vol. 13, nº 3, pp. 108-117, 2006.

1

Introducción

El primer capítulo de esta memoria recoge la motivación tras este proyecto, su objetivo y las herramientas necesarias para llevarlo a cabo. En la última sección se detalla la organización del documento para facilitar su lectura.

La Cuarta Revolución Industrial (Industria 4.0) se estima que comenzó en torno a 2011, aunque el término no se popularizó hasta 2016 [**1**]. Esta revolución se apoya en las tecnologías de la transformación anterior para llevar a cabo una fusión de tecnologías que consiguen difuminar la diferencia entre lo físico, lo digital y los elementos biológicos. A pesar de ser similar a la tercera, esta se distingue por su rapidez de dispersión, su alcance y los tipos de sistemas afectados.

Esta revolución incluye la posibilidad de que miles de millones de personas se encuentren conectadas a través de dispositivos móviles, la incorporación de técnicas de inteligencia artificial, robótica, IoT (Internet of Things), impresión 3D... Todos estos términos, y muchos otros que forman parte de esta revolución, a día de hoy ya son conocidos por buena parte de la población. Sin embargo, todavía no se puede considerar que estén totalmente integrados en todas las empresas.

Esta situación denota que, si bien no se puede considerar que haya finalizado la implementación de la Industria 4.0, su inserción se encuentra ya en un punto muy avanzado. Esto coincide con las perspectivas de [**2**] que en 2017 pronosticaba que esta transformación tendría lugar en los 10 o 15 años posteriores a su comienzo, lo que ubica el final de esta renovación entre 2021 y 2026.

Algunos expertos consideran que se empieza a atisbar una quinta revolución (Industria 5.0) [**3**]. Esta transformación presenta el concepto de los *cobots*: *collaborative robots* que buscarán satisfacer las necesidades de las empresas para generar artículos personalizados. Para ello, se llevarán a cabo fuertes avances en inteligencia artificial, la velocidad de conexión de las empresas (5G y 6G) y los CPS (*Cyber-Physical Systems*) que son sistemas colaborativos que están en contacto con el mundo físico que les rodea y los procesos que tienen lugar en él.

En esta época de revoluciones constantes, puede resultar complicado distinguir a cuál de ellas pertenece cada elemento, pero queda claro que los robots forman parte fundamental de ellas. Respecto a los robots móviles, existen dos grandes grupos: AGV (*Automatic Guided Vehicle*), que son los más abundantes y AMR (*Autonomous Mobile Robot*), grupo en el que se centrará este proyecto. En la Capítulo 2 se ampliará la información acerca ambos tipos y se darán algunos ejemplos industriales.

1.1. Motivation

La motivación para este trabajo surge de la necesidad de acercar las nuevas tecnologías de hoy a los profesionales de mañana a través de las aulas. Por ello, se pretende realizar un proyecto apoyándose en las tecnologías más disruptivas del mercado con la intención de que pueda ser utilizado posteriormente en el ámbito educativo para la enseñanza de aspectos de robótica industrial.

Para ello, se emplearán los últimos estándares de la materia: ROS 2 con las versiones más actualizadas. A partir de este proyecto, se podrán desarrollar otros que engloben la materia a exponer en la asignatura del plan de estudios de nueva implantación de la universidad: Grado en Ingeniería Matemática e Inteligencia Artificial.

1.2. Objetivos

El objetivo del proyecto es desarrollar una solución completa de navegación y SLAM en simulación para un robot móvil Turtlebot 3. Con esta reproducción virtual o gemelo digital se pretende acelerar el desarrollo de aplicaciones que utilicen el robot físico real tanto en docencia como en investigación. Además, el hecho de disponer de una réplica virtual permite generar mucha más experiencia sintética en menos tiempo con la que alimentar técnicas avanzadas de inteligencia artificial.

1.3. Herramientas

Las herramientas empleadas en este trabajo se dividen en dos grandes grupos: *software* y *hardware*. El desarrollo *software* y su simulación se ha llevado a cabo en una máquina virtual con Linux 20.04, en el que se encontraba instalado ROS y varios simuladores, entre ellos Gazebo y CoppeliaSim.

1.3.1. ROS2

En este proyecto se empleará ROS2 (Robot Operating System 2), que, a pesar del nombre, no es un sistema operativo, sino un conjunto de *frameworks open-source* para el desarrollo de *software* para robots. Es el más utilizado por la comunidad investigadora y educativa, y su uso se está comenzando a extender a la industria con la versión ROS-Industrial [**4**].

ROS (*Robot Operating System*) surge en 2007 en la Universidad de Stanford con el objetivo de facilitar el desarrollo colaborativo de *software* para robots. Sus creadores, dos estudiantes de doctorado, se dieron cuenta de la necesidad de crear una base sobre la que poder construir distintas aplicaciones, sin necesidad de tener que desenvolverse perfectamente en todos los aspectos que engloba la creación de un robot. De esta forma se crearía esta plataforma con una

gran oferta de herramientas y librerías fácilmente integrables con el núcleo, que gestiona su agregación [**5**].

El último gran cambio en esta arquitectura fue el cambio de ROS a ROS2. Esto ha actualizado las versiones de C++ y de Python sobre las que se trabaja y ha supuesto una transformación en la arquitectura general, ya que se ha pasado de un protocolo de transporte adaptado que proporcionaba un formato serializado a una interfaz abstracta que hace las veces de *middleware* y que proporciona la serialización y el transporte. Esto garantiza que ROS 2 puede cumplir determinadas políticas de calidad de servicio, asegurando la comunicación entre redes de distinta índole [**6**].

En el Capítulo 3 se señala el funcionamiento de ROS, señalando sus elementos principales y el modo de trabajo.

1.3.2. Gazebo

Gazebo [**7**] es un simulador dinámico 3D de código abierto capaz de simular de forma precisa diversos modelos de robots en diferentes entornos, tanto exteriores como interiores. Para ello ofrece gran calidad en las simulaciones a nivel físico, un amplio rango de sensores e interfaces. Entre sus usos habituales se encuentra el diseño de robots, la realización de pruebas de algoritmos y distintos tipos de exámenes sobre escenarios realistas.

La última versión disponible es Gazebo 11, lanzada en enero de 2020. Sin embargo, debido a la aparición de IGN Gazebo, no habrá nuevas versiones de este simulador y finalizará su soporte en 2025. A pesar de ello, es uno de los simuladores más establecidos en la comunidad, por lo que sigue mereciendo la pena emplearlo. Además, su sustituto, que es IGN Gazebo mencionado en la (Subección 1.3.3, todavía se encuentra en proceso de desarrollo continuo.

1.3.3. IGN Gazebo

Ignition Gazebo (IGN Gazebo) [**8**] es un simulador de robótica de código abierto. Su desarrollo comienza a partir de Gazebo, por lo que incluye todas sus virtudes desarrolladas a lo largo de más de 16 años. A todas estas características se añaden las ventajas de Ignition, que proporciona gran cantidad de librerías y servicios para hacer las simulaciones más detalladas, incluyendo aspectos que sería imposible considerar utilizando Gazebo.

1.3.4. CoppeliaSim

Este simulador está desarrollado por la empresa Coppelia Robotics y antes se conocía como V-REP [**9**]. Ofrece la posibilidad de controlar las simulaciones utilizando seis métodos diferentes, entre los que se incluyen nodos ROS, *scripts* embebidos o clientes API. A pesar de que el lenguaje por defecto para escribir estas aplicaciones es LUA, admite muchos otros, como C, C++, Python, Java, Matlab u Octave.

Está recomendado para el desarrollo de algoritmos, la simulación de mecanismos de automatización, temas educativos y gemelos digitales. La última versión disponible es la 4.2.0, cuya fecha de lanzamiento fue abril de 2021.

1.4. Organización del documento

La memoria está organizada en diez capítulos, siendo el presente el primero de ellos, que introduce el problema del proyecto. El siguiente contiene una revisión del mercado actual,

incluye una comparativa de algunos de los modelos de AMR más populares. En capítulos sucesivos se describe ROS como marco de trabajo y desde ahí, el desarrollo del proyecto como tal: el diseño de los entornos, el proceso de SLAM y navegación y la organización de los archivos junto a su modo de ejecución. Los tres últimos capítulos recogen los resultados, las conclusiones y los futuros desarrollos del proyecto.

Además, se incluyen tres anexos. El primero de ellos incluye la adecuación del proyecto a los Objetivos de Desarrollo Sostenible, el segundo describe el procedimiento para adaptar el modelo del robot a formato URDF y el tercero recoge los diagramas completos de las distintas arquitecturas.

2

Estado de la cuestión

A lo largo de este capítulo se llevará a cabo una revisión literaria proponiendo ejemplos industriales de distintos tipos de robots y describiendo sus particularidades, permitiendo elaborar una comparativa entre las distintas características existentes y determinando su grado de adaptación a los variados problemas que puedan surgir.

El mundo de la robótica destaca, entre otras cosas, por ser muy amplio y variado. De esta forma se pueden encontrar diversos tipos de robots que se pueden organizar y clasificar atendiendo a distintos parámetros. Una de las divisiones más fundamentales se realiza atendiendo a su movilidad de forma que surgen dos tipos fundamentales: estáticos y dinámicos. El foco de este proyecto se ubica sobre los dinámicos que, a su vez, se dividen en dos grandes tipos: AGV (*Automated Guided Vehicle*) y AMR (*Autonomous Mobile Robot*). Ambos pueden llevar a cabo el mismo tipo de funciones: transportar objetos de un punto a otro, pero presentan algunas diferencias [**10**].

Los AGV son robots móviles que se desplazan siguiendo un camino prefijado es decir, se mueven de forma autónoma pero guiada. Esta guía se puede llevar a cabo de tres formas distintas: cableada, inercial o por láser. La cableada se implementa incrustando cables en el suelo de la planta, de forma que el vehículo los detecte inductivamente para determinar su posición lateral. En el caso de la inercial, se emplean giróscopos para precisar la posición de la rueda y así conocer su situación. La guía se lleva a cabo mediante imanes estratégicamente colocados para resetear el sistema, evitando así la pérdida de exactitud que presentaría a la larga. Finalmente, la última emplea emisores-receptores láser para detectar marcas reflectoras ubicadas a lo largo de la instalación para triangular su posición.

En la categoría AMR entra cualquier tipo de robot que entiende el entorno por el que se mueve, de forma que puede desplazarse de manera autónoma sin peligro de chocarse contra obstáculos que puedan aparecer en su trayectoria de forma imprevista. Para ser consciente del entorno se necesitan distintos tipos de sensores, los más comunes son escáneres de área, láseres difusos y cámaras. Cuando se lleva a cabo su instalación inicial, se le permite al robot recorrer la planta con todos sus sensores activados y así se crea el mapa virtual que le permitirá por una parte orientarse y por otra poder detectar obstáculos y calcular la mejor ruta para esquivarlos.

Para poder llevar a cabo todo este proceso se utilizan los algoritmos denominados como SLAM (*Simultaneous Localization And Mapping*) [**11**] [**12**].

Ambos tipos tienen sus ventajas y desventajas. En los AMR todo el procesamiento acerca de la gestión de navegación, es decir, esquivar obstáculos, se produce en el *software*[1] del propio robot mientras que en los AGV esta decisión viene impuesta. Esta imposición implica que hay que realizar modificaciones en la planta para permitir su circulación, lo que reduce considerablemente la flexibilidad que tienen los AGV ante cambios en la estructura. Además, los AMR son capacees de esquivar obstáculos, lo que les hace mucho más versátiles y eficientes para la colaboración con humanos.

Sin embargo, debido a su simplicidad, los AGV tienden a ser mucho más fiables, ya que se tiene la certeza de que va a discurrir por el camino preestablecido y por último, esta sencillez hace que los robots sean mucho más baratos, lo que hace que las empresas finalmente se decanten por este tipo de modelos. Por este motivo, hay muchos más robots de tipo AGV que de tipo AMR en la industria.

Para una mejor comprensión de los modelos AMR que hay en el mercado es necesario conocer las características de dichos productos. Entre estas propiedades destacan la cinemática o los sensores necesarios para navegación.

2.1. Sensores de navegación

Los AMR se caracterizan por su capacidad de desplazarse de una posición a otra calculando su propia ruta y evitando obstáculos. Para que todo esto sea posible, deberán ser capaces de reconocer el entorno que les rodea, reconocerlo y actuar en consecuencia. Esta sección estudia algunos de los sensores más comunes a la hora de reconocer el entorno.

La navegación se puede distinguir en dos tipos fundamentales: global y local. La orientación global exige un conocimiento previo del entorno, de forma que se puedan utilizar métodos *Off-line*. Entre estos métodos destacan algoritmos que buscan calcular la trayectoria óptima. Entre estos destacan el algoritmo de Dijkstra [**13**], el algoritmo A* (*A star*) [**14**], grafos de visibilidad, métodos de campo de potencial artificial (APF) [**15**] y métodos de descomposición por celdas. Respecto a la navegación global, también conocida como *On-line*, el robot decidirá su propia posición, orientación y dirección. En este caso, los métodos más comunes son los sensores infrarrojos, ultrasónicos, láser y sensores de visión (cámaras). A partir de los datos recibidos por estos sensores y aplicando diversos algoritmos con redes neuronales, lógica difusa, sistemas neuronales difusos o algoritmos genéticos entre otros [**16**].

El LiDAR es uno de los sensores más comunes en este tipo de robots a la hora de controlar la orientación y la posición. Su funcionamiento permite que opere de manera independiente, pudiendo mapear por sí mismo todo el entorno. Sin embargo, al combinarlo con otros sensores, como GPS o sistemas de navegación inercial, presenta unos resultados excelentes. Un caso muy común es combinarlo con una cámara, de forma que se convierte en un potente sensor de posicionamiento. Esto se debe a que se complementan muy bien, supliendo las ineficiencias de cada uno de los componentes y con la ventaja de que el LiDAR es especialmente barato, por lo que no tendrá un sobre coste adicional.

[1] Actualmente existen algunas corrientes que defienden un modelo híbrido, en el que intervengan tanto el *software* como el *hardware*.

2.2. Comparativa

Esta sección compara cinco modelos de robots industriales para asegurar que el robot elegido para desarrollar este proyecto sea asequible para un entorno docente garantizando su similitud con aquellos de uso industrial. Por ello, todos los modelos considerados están diseñados para trabajar en entornos industriales interiores, aunque algunos estén preparados para ir por exteriores en caso de que sea necesario. Se presentan cinco robots, de distintas gamas y con distintas funcionalidades, cada uno ellos en su propia subsección, donde se detallarán puntos de su diseño y sus características.

2.2.1. Dingo indoor robotic platform

Dingo [**17**] es un robot diseñado por Clearpath Robotics. Destaca por ser pequeño y modular. Existen dos versiones: *Dingo-O* y *Dingo-D*, mostradas en la Figura 2.1. El primero tiene un sistema cinemático diferencial, pesa 16 kg y soporta una carga máxima de 20 kg El segundo, cuenta con un sistema omnidireccional diseñado con ruedas suecas, pesa 20 kg y puede llevar hasta 25 kg. A pesar de estas diferencias en su método de desplazamiento, ambos pueden alcanzar una velocidad máxima de $1{,}3\ ^{m}/_{s}$.

(a) **(b)**

Figura 2.1. Imágenes de ambos modelos de Dingo (Fuente: https://clearpathrobotics.com/dingo-indoor-mobile-robot/). (a) Dingo-D y (b) Dingo-O.

Para el resto de especificaciones, tienen las mismas características. Tienen un panel en la parte trasera que muestra el estado de la batería y su conexión y cuenta con un botón para apagar y encender. Ambos disponen de dos antenas: una para conexión Wi-Fi y otra para Bluetooth. También ofrecen la posibilidad de establecer conexión cableada a través de Ethernet, USB 3.0 y RS 232.

El robot presenta un diseño sencillo, con un carril central sobre el que acoplar distintos sensores, como lidars o cámaras, o actuadores. En la Figura 2.2 se puede observar cómo se acoplarían un lidar y una cámara al robot. En su interior, cuenta con espacios modulares, cuatro en el caso del Dingo-O y dos con el Dingo-D. En estos se pueden colocar distintos bloques de baterías, ion-litio o de plomo libre de mantenimiento, u ordenadores de diferentes capacidades para llevar a cabo el procesamiento de datos.

Bajo el raíl central se encuentra la unidad MCU (*Main Control Unit*), una placa con el microcontrolador responsable de ejecutar el firmware del robot y de la gestión de la

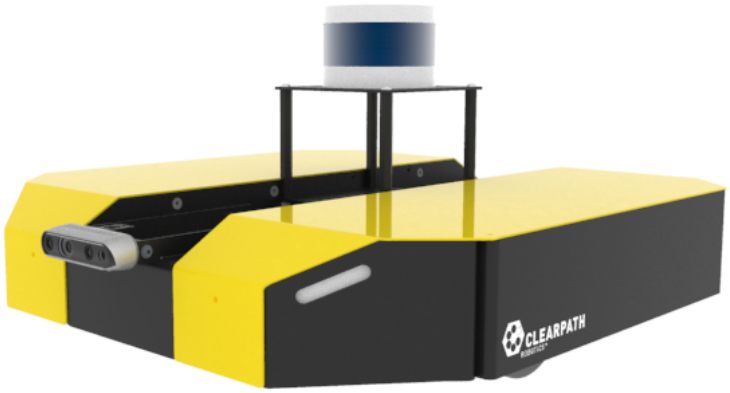


Figura 2.2. Ejemplo de un Dingo-D con un lidar y una cámara acoplados. (Fuente: https://clearpathrobotics.com/dingo-indoor-mobile-robot/).

batería, permitiendo seleccionar el voltaje al que alimentar los sensores y actuadores. En cuanto al controlador, admite ROS Melodic, Gazebo y MoveIt! y cuenta con gran cantidad de documentación y tutoriales.

Respecto a los precios, el Dingo-O estaría disponible por unos $15\ 789{,}47$€ [**18**], mientras que el Dingo-D se sitúa en unos $10\ 105{,}26$€ cada unidad sin considerar módulos de baterías ni de procesamiento [**17**].

2.2.2. 4WD Rover Pro

Rover Robotics [**19**] es la empresa responsable de este producto. Este robot, diseñado en titanio, cuenta con la protección IP67. Esto garantiza que puede operar bajo cualquier condición meteorológica y puede operar en un muy amplio rango de temperaturas, desde $-17°C$ hasta 49° C. Lo que lo convierte en el robot idóneo para operar sobre cualquier tipo de terreno, interior y exterior.

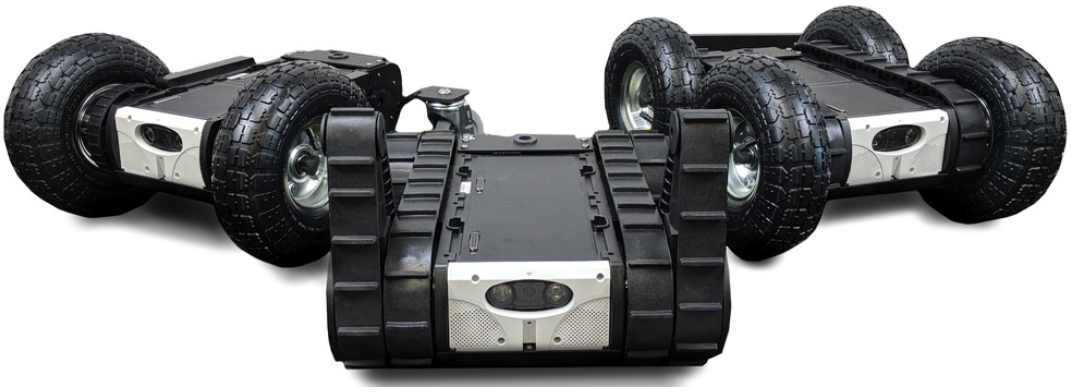


Figura 2.3. 4WD Rover pro con ruedas en los laterales y aletas en el central (Fuente: https://roverrobotics.com/products/rover-pro).

Respecto a la cinemática, cuenta con dos módulos distintos. Por una parte, se le pueden acoplar cuatro ruedas grandes, que proporcionan mucha estabilidad (Figura 2.3, laterales) y en caso de operar por interiores, las traseras se pueden sustituir por un par de ruedas locas. También ofrece la posibilidad de colocarle una tracción por ruedas de oruga, con aletas en los

extremos (Figura 2.3, central). Estas aletas permiten que pueda subir pendientes de hasta 60° y le dan gran movilidad para recuperarse en caso de volcar.

El robot alcanza velocidades de hasta $1{,}56\ ^m/_s$ utilizando las aletas y $3{,}57\ ^m/_s$ y puede soportar cargas útiles de hasta 68 Kg El robot tiene un peso base de 9 Kg valor sube hasta los 11 Kg si se tienen en cuenta las baterías, que son de ion-litio de última generación. La duración media estimada de las baterías con un uso normal es de entre 4 y 5 horas.

La edición pro incluye un procesador Intel Atom, una cámara VGA, un altavoz y micrófono. También dispone de un LED infrarrojo y otro visible para la solución de telepresencia. En la página de compra están disponibles los modelos CAD y los archivos STL optimizados para simulaciones, pero no se especifica nada acerca del sistema operativo del robot, o si soporta ROS.

Cada unidad de este robot está disponible a partir de los $7\ 028{,}95$€ sin incluir baterías, que se pueden adquirir por $540{,}95$€ ni el puesto de carga, que se puede adquirir por $399{,}95$€. El precio inicial incluye únicamente el chasis del robot.

2.2.3. Khepera IV

La empresa que comercializa este modelo es K-Team, compañía suiza que desarrolla, construye y comercializa distintos tipos de robots. Concretamente este está diseñado para operar por interiores y es muy modular. Esto se traduce en que puede se puede personalizar la configuración de muchas maneras, empleando su bus interno. Su diseño compacto se muestra en la Figura 2.4. El modelo pesa 540 g y soporta una carga de hasta 2 kg [**20**].

Figura 2.4. Diseño del Khepera IV. (Fuente: https://www.k-team.com/khepera-iv)

El robot viene equipado con gran cantidad de sensores, entre los que se incluyen acelerómetro, giróscopo o cámara RGB. También está equipado con otros destinados a la detección de obstáculos: 8 detectores infrarrojos y 5 ultrasónicos para larga distancia. Posee dos micrófonos integrados y un altavoz. Todos ellos recogen gran cantidad de información que puede ser empleada de muy diversas formas. No obstante, en comparación con modelos anteriores presenta la desventaja de estar limitado a trabajar con los sensores que trae integrados de serie, perdiendo así un alto grado de personalización.

Respecto a la cinemática, cuenta con dos ruedas y se controla de forma diferencial. La estabilidad la consigue gracias a dos ruedas esféricas situadas en el eje perpendicular al formado

por las ruedas de tracción. Esto se puede observar en la Figura 2.5, que muestra la parte inferior del robot. Alcanza velocidades máximas de $1^m/_s$

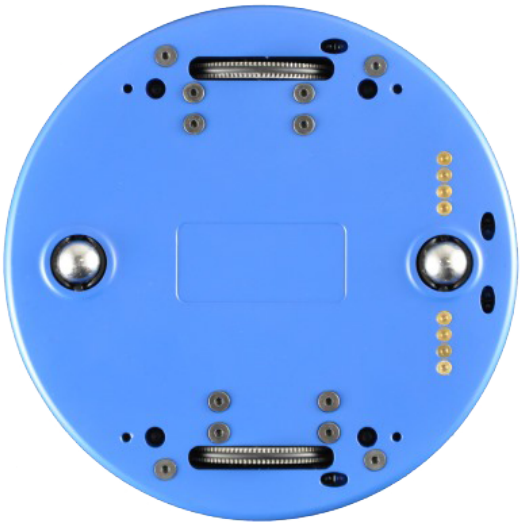


Figura 2.5. Diseño de la parte inferior del Khepera IV. (Fuente: https://www.k-team.com/khepera-iv)

El núcleo de Khepera IV incorpora un sistema operativo Linux estándar, eficiente y muy estable, que incluye una librería específica para gestionar las conexiones con los periféricos. Además, proporciona un entorno de desarrollo estándar para el desarrollo de aplicaciones C/C++, aunque también soporta Python 2.7.9. Se puede importar y utilizar prácticamente cualquier librería existente, lo que facilita el desarrollo de algoritmos y aplicaciones. En cuanto a la comunicación, integra antenas WiFi y Bluetooth internas.

Según la web oficial de K-Team, el precio recomendado del modelo sin considerar gastos de envío ni impuestos locales sería de $2\ 514{,}6$€, lo que quedaría en un total de $3\ 240$€ considerando estos factores [**21**].

2.2.4. Turtlebot 3

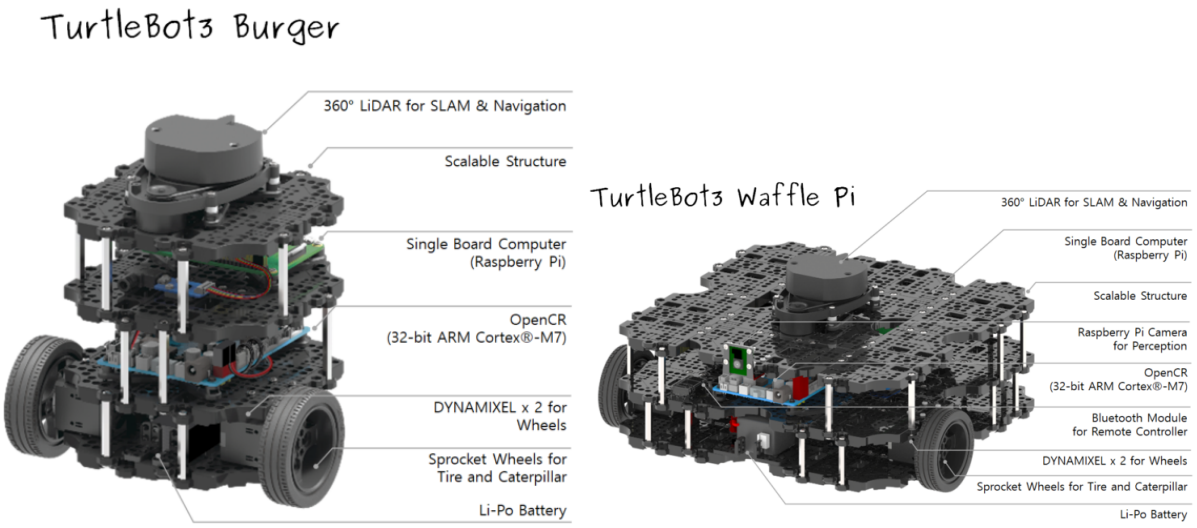
Turtlebot [**22**] es un kit de robótica de bajo coste que emplea *software open-source* muy utilizado en educación, investigación o para el prototipado de productos. Su objetivo es lograr reducir el tamaño de la plataforma y reducir los precios sin sacrificar ni funcionalidad ni calidad.

En la tercera edición de Turtlebot [**23**], se han presentado dos modelos distintos, el Burger y el Waffle Pi, mostrados en la Figura 2.6. Ambos tienen especificaciones muy distintas, y se podría dedicar una sección a cada uno de ellos. Sin embargo, este proyecto se desarrolla empleando el Turtlebot 3 Burger y por ello esta sección se centrará en sus características.

El Burger pesa en total, incluyendo baterías y sensores, 1 kg y soporta una carga máxima de 15 kg. El movimiento del robot se realiza mediante una cinemática diferencial, apoyado sobre una rueda esférica. Alcanza velocidades máximas de $0{,}22^m/_s$, muy inferior a lo logrado por los modelos anteriores, que tenían un fin más industrial. Las dimensiones del modelo se muestran en la Figura 2.7.

El modelo incluye un giróscopo, un acelerómetro y un magnetómetro integrados, pero el sensor que permite llevar a cabo todas las operaciones de SLAM es el 360 LDS-01 (*Laser Distance*

Sensor) que es un escáner láser de 360° . También se proporcionan distintos conectores de alimentación y pines de Arduino para poder conectar otro tipo de sensores y se incluyen varias interfaces para conexiones periféricas, entre las que se encuentran UART, CAN, SPI e I2C entre otras.



(a) **(b)**

Figura 2.6. Modelos de Turtlebot 3 (Fuente:https://emanual.robotis.com/docs/en/platform/turtlebot3/features/). (a) Burger y (b) Waffle pi.

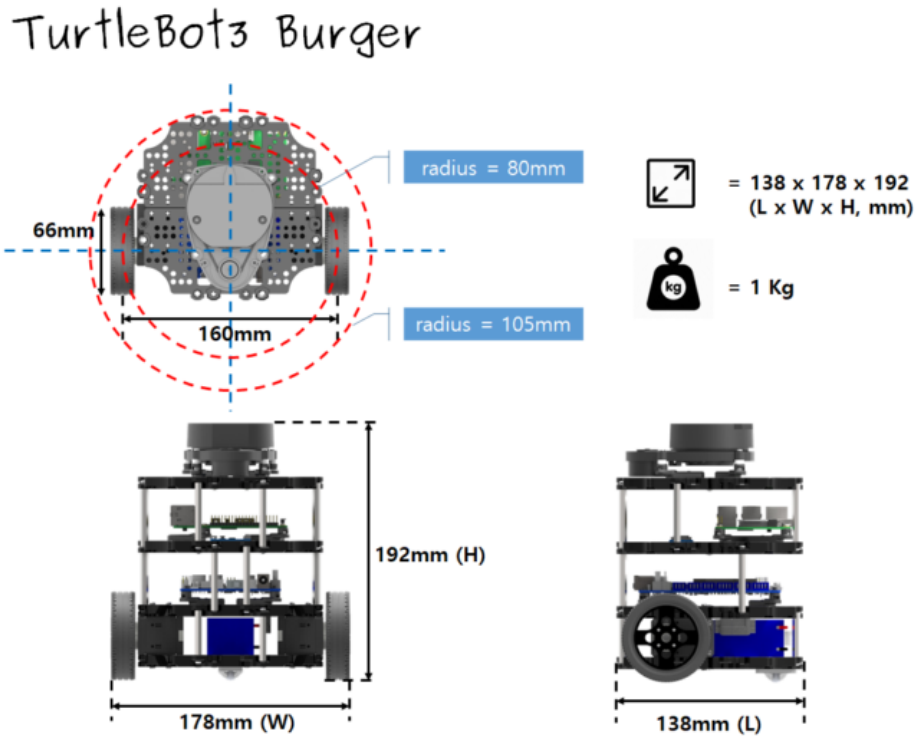


Figura 2.7. Dimensiones del Turtlebot3 Burger (Fuente: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/).

La conexión al robot se debe realizar vía USB. La MCU tiene un procesador Cortex-M7 de 32-bit ARM con una FPU (*Floating Point Unit*, unidad de coma flotante) con una frecuencia de cálculo de 216 MHz y una capacidad de gestión de procesos de 416 DMIPS. Respecto al

procesamiento de los datos capturados por los distintos sensores, se lleva a cabo a través de una Raspberry Pi con ROS integrado, lo que permite crear diversos tipos de programas y aplicaciones. Las baterías son de polímero litio (LiPo - *Lithium polymer*) y tienen una duración media estimada para un uso normal de 2 horas y media, igual que su periodo estimado de carga.

Cada unidad de Turtlebot3 Burger tiene un precio de 499€ [**24**], lo que lo hace mucho más barato que el resto de robots mencionados hasta el momento en esta comparativa.

2.2.5. Osoyoo

El último modelo de esta sección es muy diferente a todos los demás vistos hasta el momento. Su funcionalidad es meramente educativa, y está dirigido a un público joven cuyo único objetivo es el de introducirse en el mundo de la robótica. Osoyoo es, realmente, el nombre de la empresa que lo comercializa que está especializada en estos kits educativos [**25**]. El robot descrito en esta sección no tiene nombre como tal y se describe como un coche robótico mecánico omnidireccional para Arduino [**26**]. Se muestra en la Figura 2.8.

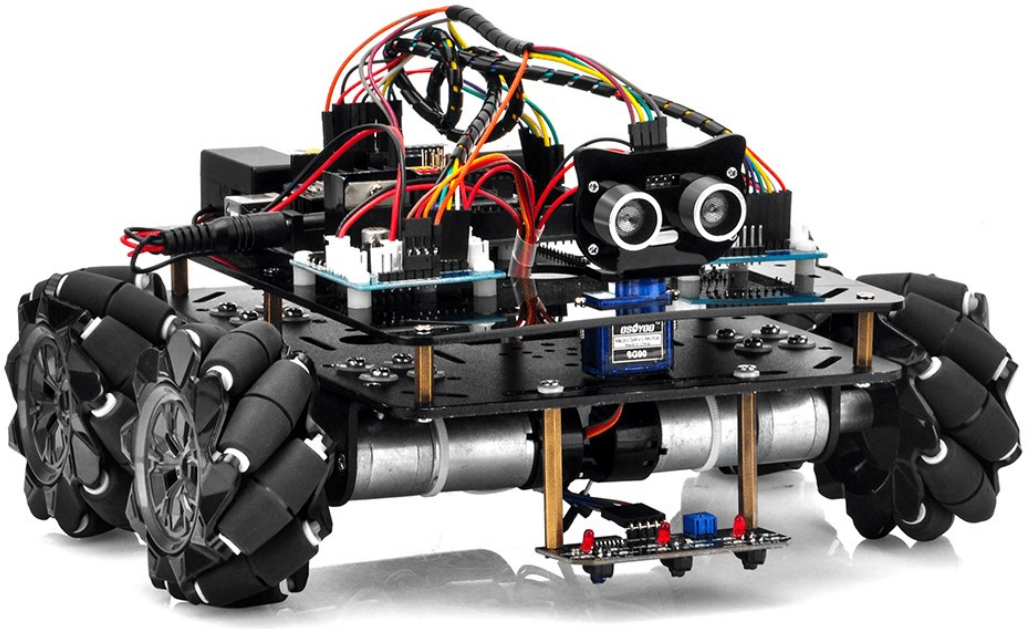


Figura 2.8. Imagen del robot de Osoyoo descrito (Fuente: https://osoyoo.com/2019/11/08/omni-direction-mecanum-wheel-robotic-kit-v1/).

Dado que el objetivo perseguido por la empresa no es industrial, las especificaciones están menos definidas, ya que su intención no es desempeñar grandes tareas, sino aprender mediante su construcción y programación. Sin embargo, cabe destacar que cuenta con un sensor de ultrasonido para la detección de obstáculos. De cara a la conexión, cuenta con conexión cableada por USB y para la inalámbrica, módulo Bluetooth y WiFi.

El procesamiento de todos los datos recibidos, así como el control del robot y su programación se realizarán a través de un Arduino, incluido también en el paquete de venta. Su peso total es de 1.9 Kg y no se considera que pueda admitir carga como tal. Para poder realizar el aprendizaje, existen tutoriales, documentación y lecciones disponibles en la web.

El precio del pack completo es de $139{,}00$€ [**25**]. El descenso del precio en relación con el resto de robots mencionados es notable, pero también lo son sus especificaciones. La finalidad es muy diferente, y ello hace que tenga que ser accesible a todos los públicos. Por otra parte, una vez montado, no se espera que desempeñe ninguna otra función, mientras que el resto tiene que llevar a cabo tareas de una cierta precisión durante un periodo prolongado de tiempo.

2.2.6. Tabla comparativa

En la tabla de la Tabla 2.1 se muestra un resumen con la comparativa general de los elementos de esta sección. Se recogen los detalles más importantes de cada robot de forma clara y ordenada, incluyendo la cinemática, el peso, la carga máxima, baterías, los sensores integrados, si soporta ROS y el precio. En la tabla están ordenados en orden descendiente de precios.

	Dingo-O	**Dingo-D**	**4WD Pro**	**Khepera IV**	**Turtlebot3 Waffle Pi**	**Turtlebot3 Buger**	**Osoyoo**
Cinemática	4 ruedas suecas	4 ruedas	- 4 ruedas - 2 ruedas + 2 ruedas locas - Ruedas de oruga con aletas	2 ruedas + 2 ruedas esféricas	2 ruedas + 2 ruedas esféricas	2 ruedas + rueda esférica	4 ruedas suecas
Peso	16 kg	20 kg	11 kg	540 g	1,8 kg	1 kg	1,9 kg
Máx. Carga	20 kg.	25 kg.	68 kg.	2 kg.	30 kg.	15 kg.	-
Baterías	Ion-Litio	Ion-Litio	Ion-Litio	LiPo	LiPo	LiPo	Pilas de tipo 18650
Sensores Integrados	-	-	Cámara VGA Micrófono	Infrarrojos Ultrasónicos Micrófonos	360 LDS-01 Giróscopo Acelerómetro Magnetómetro	360 LDS-01 Giróscopo Acelerómetro Magnetómetro	Ultrasonidos
ROS	Sí	Sí	No	Sí	Sí	Sí	No
Precio	15789.47€	10105.26€	7640.9€	3240€	1199€	499€	136 .€

Tabla 2.1. Tabla comparativa de los robots mencionados.

3

Robotic Operating System (ROS)

Se comienza realizando una descripción más exhaustiva de ROS 2, indicando sus características principales sobre las que se basa el proyecto. Se lleva a cabo un repaso de su método de funcionamiento de manera más exhaustiva que en la Subección 1.3.1.

Como se indica en [**27**], ROS (Robotic Operating System) es un meta sistema operativo de código abierto para robots, que proporciona las funcionalidades propias de un sistema operativo incluyendo la abstracción del hardware, control de dispositivos de bajo nivel o, entre otros, gestión de paquetes. También proporciona diversas herramientas para crear, compilar y ejecutar código a través de diversos ordenadores.

ROS es un conglomerado de diversos elementos, un ecosistema que engloba gran cantidad de paquetes y distribuciones con sus respectivos tutoriales y documentación. No busca ser un *framework* que aglutine el máximo número de funcionalidades distintas, sino que intenta facilitar la reutilización y el intercambio de código en entornos de investigación y desarrollo de robótica.

Se caracteriza por ser un marco de trabajo de procesos distribuidos, conocidos como nodos. Cada nodo desempeña una tarea concreta, un ejemplo sería la adquisición de datos de un sensor, otro podría gestionar los motores de las ruedas o calcular la localización. Con este diseño a bajo nivel se permite que los ejecutables puedan ser independientes y que se vayan acoplando en el momento de ejecución, lo que aporta flexibilidad, mayor tolerancia a fallos y reduce la complejidad.

Los nodos realizan tareas sencillas e independientes, por lo que deben intercambiar información entre sí para lograr alcanzar el objetivo final del desarrollo. Esta comunicación se realiza a través de un protocolo de publicación-suscripción basado en los protocolos TCP/IP. El nodo que envía la información la publicará en un *topic* al que estarán suscritos los nodos interesados en recibirla. Los nodos pueden estar interesados en esa información por diversos motivos, desde procesarla hasta mostrarla al usuario o guardarla.

En ROS existía un responsable de gestionar los nodos y los *topics*, el denominado ROS Master. Al inicializar un nodo, este se debía registrar en el maestro, indicando si va a

publicar información y en qué *topic*. También debía señalar a qué *topic* se iba a suscribir, en caso de hacerlo. El maestro proporcionaba la información necesaria para permitir estas conexiones.

No obstante, la situación es distinta en ROS 2, ya que se permiten nodos en tiempo real. Por ello, en esta ocasión se emplea un DDS (*Data Distribution Service*, un estándar de la industria a la hora de conseguir interoperabilidad entre proveedores, muy probado en múltiples situaciones y con grandes garantías [**28**]. Los *topics* deben ser nombrados según ciertas indicaciones para así poder obtener la FQN (*Fully Qualified Name* que permita obtener el nombre equivalente en el DDS [**29**].

A lo largo de este proyecto se han incluido distintos gráficos que recogen esta información para las distintas soluciones planteadas. Estas imágenes se han obtenido con la herramienta `rqt_graph`, que identifica los nodos y los *topics* activos y los representa. En estos diagramas, los nodos aparecen representados con un óvalo, mientras que los topics estarán representados por flechas que comienzan en el publicador y señalan a los nodos suscritos.

La Figura 3.1 muestra un diagrama sencillo, que sirve de ejemplo para explicar estas imágenes. Los nodos empleados para este ejemplo son, `/turtlesim` y `/teleop_turtle`. Ambos aparecen representados en la Figura 3.2, donde `/turtlesim` es la representación de la tortuga y `/teleop_turtle` se encuentra en la terminal, esperando las instrucciones del usuario.

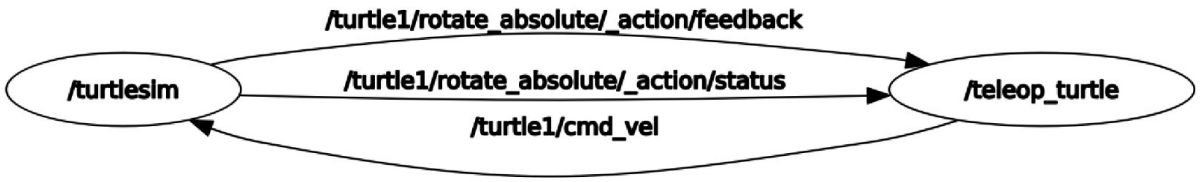


Figura 3.1. Ejemplo sencillo de diagrama de nodos.

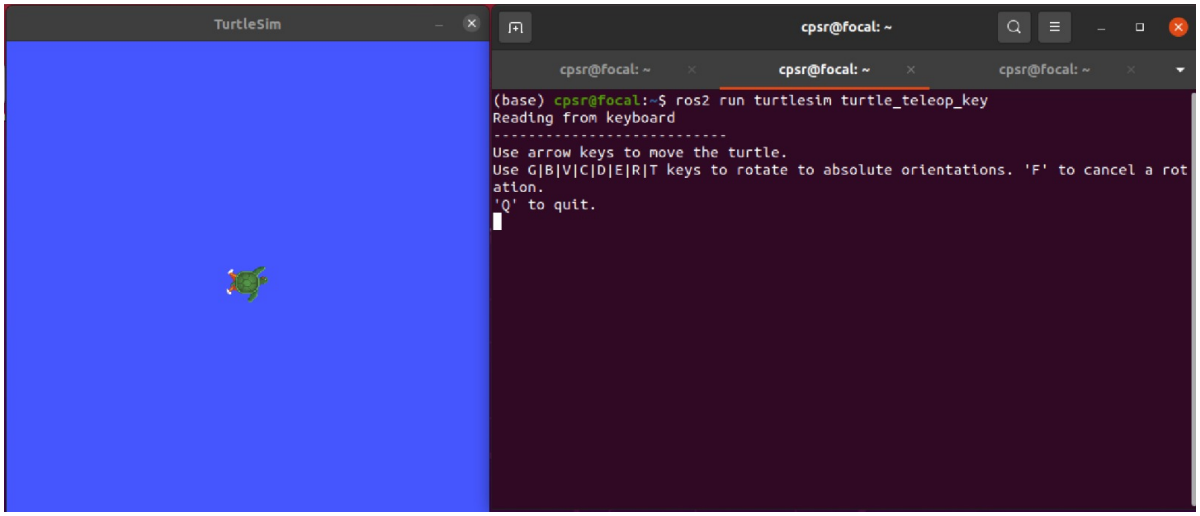


Figura 3.2. Visualización de los nodos presentes en el ejemplo.

En la Figura 3.1 también se observan tres flechas entre los nodos, que representan los *topics*. Los dos superiores son publicados por `/turtlesim` y contienen información acerca de la posición y el estado de la tortuga mientras que el tercero está publicado por `/teleop_turtle`. Este *topic*,

`/cmd_vel` contiene la información introducida por el usuario y necesaria para que la tortuga se pueda desplazar por el entorno.

Este ejemplo es muy sencillo, pero a medida que se van buscando soluciones más completas se van complicando los diagramas. En muchas ocasiones surgen nodos que deberán trabajar en conjunto para lograr un objetivo, y resultaría poco práctico tener que ir levantando los nodos manualmente. Por ello se organizan en paquetes, que son unidades de software de ROS. De esta forma es posible reutilizar el código de manera sencilla.

Un paquete contiene distintos tipos de librerías, nodos, conjuntos de datos, ficheros de configuración... En el Capítulo 7 se describe el paquete generado para la solución de este documento, y dan más detalles acerca de su estructura y de los archivos que definen sus características.

4

Diseño del entorno

En este capítulo se describen los pasos necesarios para generar un mundo para Turtlebot3 utilizando el modo constructor de Gazebo. Se detallan los archivos obtenidos y su incorporación a los archivos para poder ejecutarlos.

El desarrollo de la funcionalidad planteada se llevará a cabo utilizando las diversas herramientas de ROS2. Sin embargo, para probar su funcionamiento será necesario conocer el entorno sobre el que se simulará. Este entorno debe cumplir una serie de características. Por ello, se ha decidido diseñar un conjunto de laberintos conocidos sobre los que ejecutar las tareas.

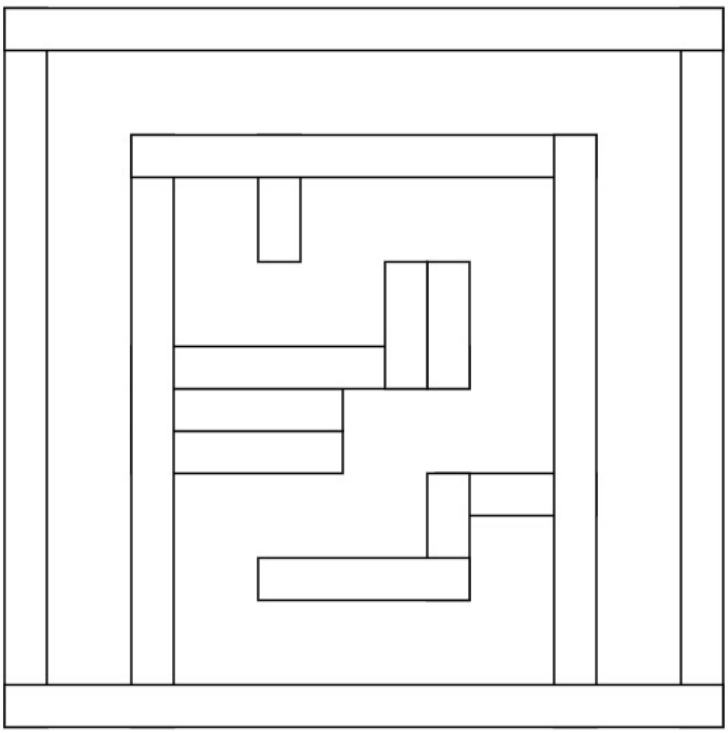


Figura 4.1. Esquema del primer laberinto.

Se han utilizado dos diseños diferentes de laberintos, inspirados en unos modelos para CoppeliaSim. El primero de ellos está representado en la Figura 4.1. Las paredes tienen un grosor de medio metro, siendo posible que haya algunas zonas en las que se encuentren anchuras superiores. En él se distinguen dos caminos: el exterior y el interior.

El camino exterior es una sencilla U, y será el utilizado para probar en una primera instancia los algoritmos implementados. El trazado interior es ligeramente más complicado en cuanto a la forma. Sin embargo, no se puede considerar un laberinto como tal, ya que solo cuenta con un trazado para poder desplazarse, de forma que a la hora de realizar una navegación, no hay distintas alternativas.

El segundo laberinto es ligeramente más complejo y tiene un tamaño de $10\,x\,10\,m$. Las paredes siguen contando con una anchura mínima de medio metro, pero el trazado es mucho más complicado. En esta ocasión, no hay un camino único que permita llegar a cada punto del mapa, lo que permite comprobar el funcionamiento de los algoritmos de navegación. El trazado del segundo laberinto se muestra en la Figura 4.2. Este diseño se realizó buscando una ligera simetría entre las partes del laberinto, lo que dificulta en gran medida la realización de determinadas acciones.

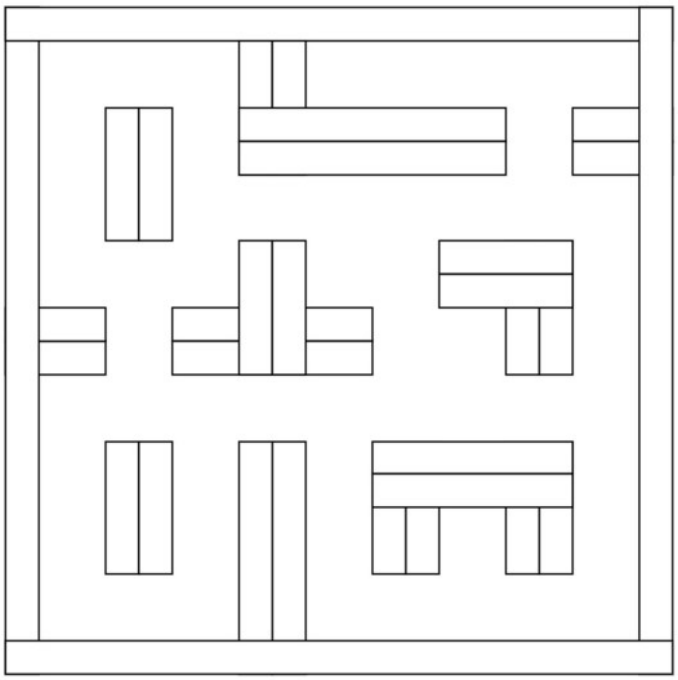


Figura 4.2. Esquema del segundo laberinto.

El primer paso fue decidir qué herramienta utilizar para desarrollar los laberintos[1]. Se plantearon varias opciones, incluyendo la transformación de los archivos ya existentes en formato .ttt, programando directamente el archivo .sdf o incluso empleando herramientas de modelado como Blender. Finalmente, se optó por utilizar el modo constructor de Gazebo [**30**].

Para acceder, se debe lanzar el simulador, que por defecto se abrirá con un mundo vacío. Sobre este, se abrirá el modo constructor pulsando la combinación de teclas “Ctrl+B”. En

[1]Ambos laberintos se han desarrollado siguiendo el mismo procedimiento.

esta nueva interfaz, mostrada en la Figura 4.3, se distinguen dos partes principales. La parte superior representará a modo esquemático la construcción que se está realizando, mientras que la inferior muestra el resultado final en tres dimensiones. El panel de la izquierda ofrece opciones para incluir elementos en el mundo.

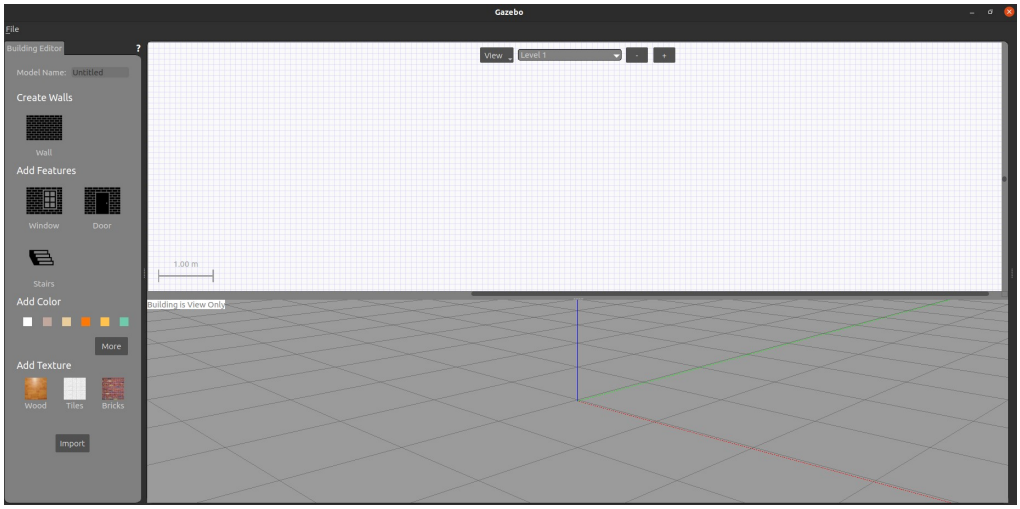


Figura 4.3. Interfaz del modo constructor de Gazebo.

El simulador permite añadir una imagen esquemática que sirva de guía para ir añadiendo los elementos en el lugar correcto. No obstante, debido al grosor de las paredes, se optó por prescindir de esta ayuda. El grosor fue un problema a la hora de desarrollar el laberinto ya que al modificar esta medida, el programa cambiaba la longitud de la pared. Por este motivo en las Figura 4.1 y Figura 4.2, las paredes están compuestas de uno o varios segmentos de $0{,}5\,m$ de ancho. Esta situación se puede observar con gran claridad en la Figura 4.4, en la figura con forma de "U" invertida.

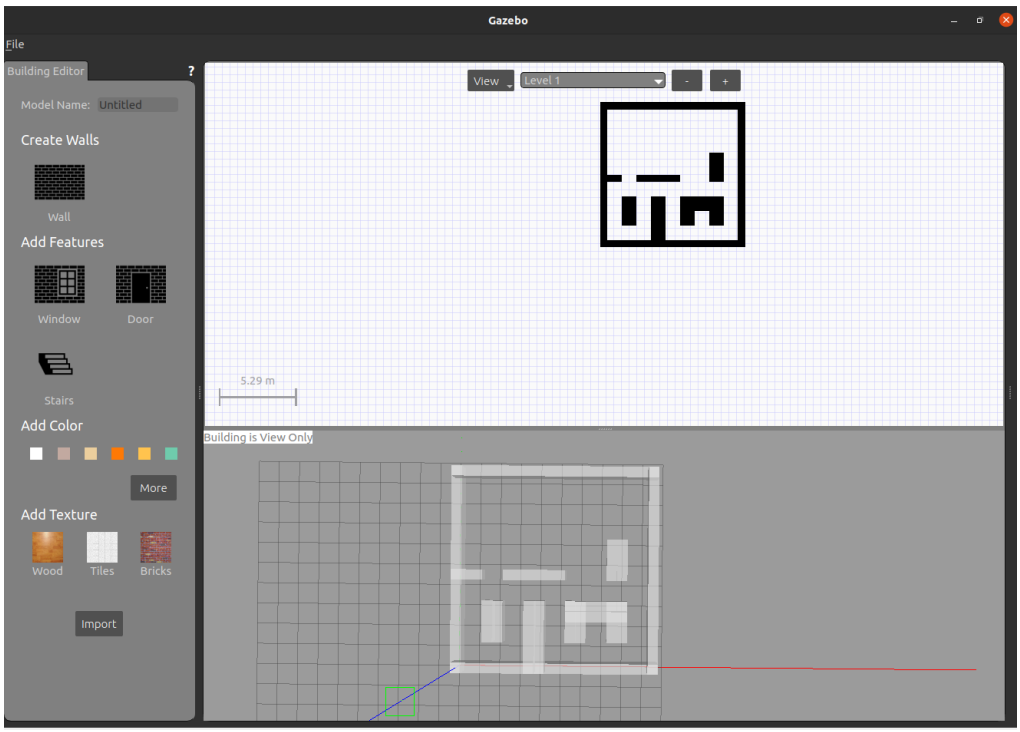
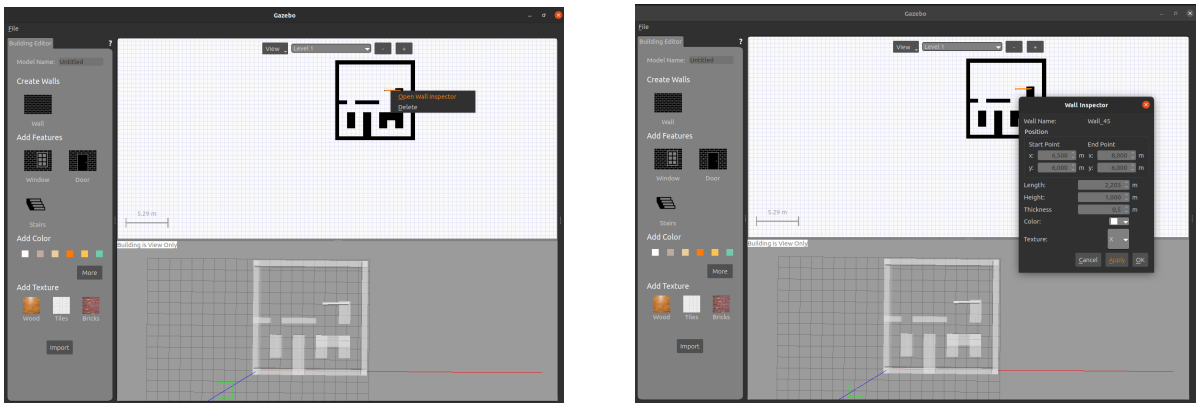


Figura 4.4. Ejemplo de visualización de la interfaz durante el desarrollo del laberinto.

En el panel de la izquierda se ofrecen diversas opciones para generar el mundo. Dado que el aspecto del modelo empleado es muy sencillo, para diseñar los laberintos solo fue preciso utilizar las paredes. También es posible añadir varios pisos a un modelo, introduciendo distintas capas y construyendo sobre ellas, pero dado que los laberintos solo se desarrollan en un plano, no fue necesario utilizarlo.

El primer paso es introducir una pared. Aunque es posible incluir varias de una vez, se recomienda hacerlo de forma secuencial. Esto se debe a que hay muchos ajustes que cambiar en cada una de ellas. Primero la altura, que por defecto es de $2{,}5$ metros. Se podría dejar así y no habría ningún problema, pero puesto que el Turtlebot3 Burger no llega a los 20 cm de altura, es recomendable disminuirla para facilitar el seguimiento del robot. Por ello se debe reducir hasta 1 metro de altura. La segunda modificación que se debe hacer es el grosor de la pared. Para satisfacer el diseño descrito, se debe establecer en $0{,}5$ metros, siendo el valor por defecto $0{,}150$ m. El cuadro de diálogo y el acceso al mismo se muestran en la Figura 4.5.



(a) **(b)**

Figura 4.5. Pasos para la configuración de una pared tras su inserción inical. (a) Cómo abrir el cuadro de diálogo y (b) Modificación de los valores.

El último paso es indicar el punto de inicio y el de fin de la pared. Para conseguir las medidas deseadas, se debe tener en cuenta el grosor de la pared, y establecer el punto de finalización medio metro antes de lo que realmente se espera. Este ajuste es muy importante, ya que de no tenerlo en cuenta, no se obtendrá el diseño esperado.

Una vez terminado el diseño del laberinto, se debe guardar para poder utilizarlo posteriormente. Se guarda el archivo en la ruta deseada y Gazebo genera una carpeta con dos archivos:

- .config: archivo que incluye la información básica del modelo, autor, versión y otros factores.
- .sdf: archivo que utilizará ROS y Gazebo para poder lanzar el mundo. Contiene el diseño realizado con etiquetas XML.

El siguiente paso es encontrar la forma de introducir este mundo en la simulación con el robot. Para ello se estudió el sistema de archivos del proyecto. En el momento de lanzar la simulación, el comando utilizado es: `ros2 launch turtlebot3_gazebo [archivo].launch.py`. El primer archivo a estudiar fue, por ende, el de tipo .launch.py.

Almacenados en la ruta: tutlebtot3_ws/src/turtlebot3_simulations/turtlebot3_gazebo/launch. En él se indica el nombre de la carpeta en la que se encuentra el archivo .world, y habrá que modificarlo para apuntarlo al directorio en el que se ubicará el mundo diseñado.

A continuación, se accedió al directorio en el que se almacenan los archivos world: tutlebtot3_ws/src/turtlebot3_simulations/turtlebot3_gazebo/worlds/ y se generó la carpeta indicada en el archivo .launch.py. En este nuevo directorio se introduce el archivo burger.model, que es igual que el contenido para otras simulaciones pero modificando el valor del mundo. Es importante tener en cuenta el valor de la etiqueta <pose> que señala la posición en el eje de coordenadas en la que aparecerán tanto el mundo como el robot. En cuanto al mundo, es importante mencionar que el sistema considera el punto central del mismo para ubicar en la localización elegida.

Finalmente, en la ruta tutlebtot3_ws/src/turtlebot3_simulations/turtlebot3_gazebo/models/ se generará otra carpeta, con el nombre asignado en el archivo de la carpeta worlds. En esta ubicación se colocarán los archivos .config y .sdf generados por Gazebo.

Antes de ejecutar la simulación es importante compilar el entorno de trabajo para garantizar que se tienen en cuenta todos los cambios realizados. Al ejecutar el nuevo archivo .launch.py generado, el resultado visual en gazebo se muestra en la Figura 4.6.

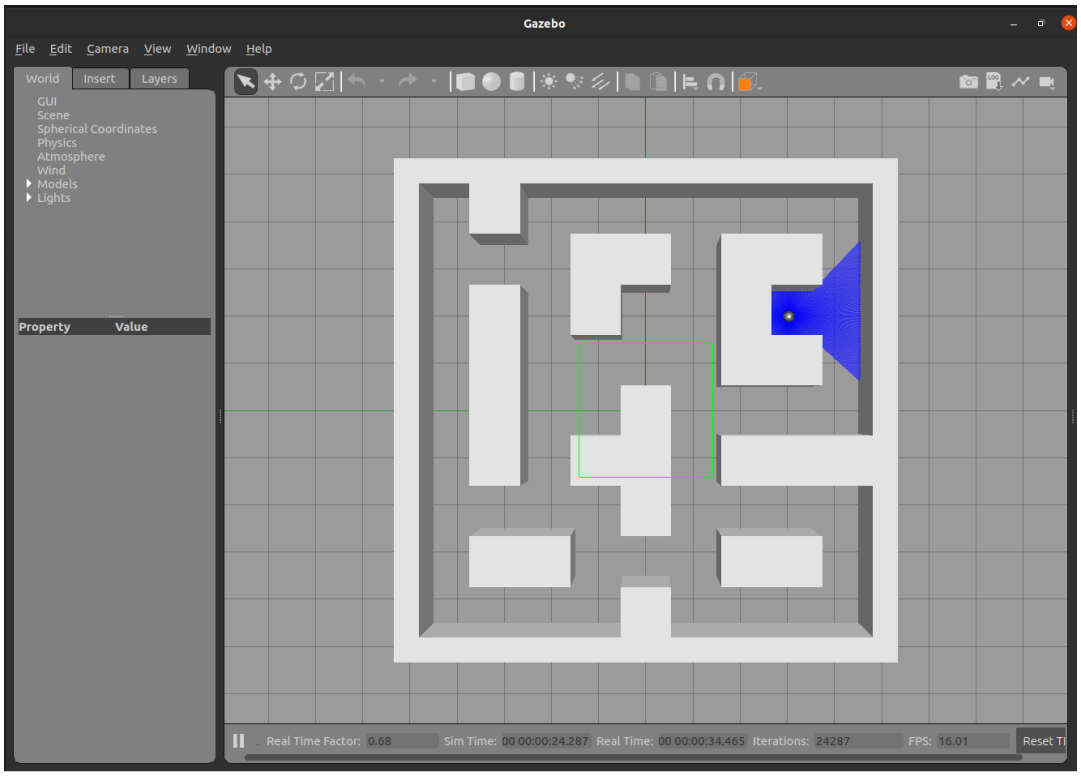


Figura 4.6. Ejemplo de visualización en Gazebo tras incluir en el archivo de ejecución el nuevo mundo y el Turtlebot3 Burger.

5

Localización y mapeo simultáneos (SLAM)

Este capítulo describe el proceso seguido y las herramientas empleadas para conseguir generar un mapa fiable que luego pueda ser utilizado para navegación. Se estudian dos de las herramientas principales en este campo analizando las ventajas y desventajas que presenta cada una.

Tras conseguir incluir el laberinto en la simulación, el siguiente paso consiste en hacer que el robot recorra el laberinto y elabore un mapa que represente el entorno en el que se encuentra para poder navegar después por él. En una primera iteración se buscó conseguir la creación del mapa dirigiendo el movimiento del robot con el teclado.

Para lograr este fin se emplean algoritmos SLAM (Simultaneous Location And Mapping). Utilizan la distintos sensores o dispositivos para obtener datos de posicionamiento. Con esta información, se va creando un mapa del entorno, calculando la posición relativa del robot respecto a estos elementos. Uno de los momentos fundamentales es el *loop closure* o cierre de lazos, en el que el sistema debe identificar que ya ha pasado por ese punto.

5.1. Cartographer

La primera herramienta empleada fue el Cartographer, recomendada por la página oficial de Robotis [**31**]. La puesta en marcha inicial fue muy sencilla, ya que gestionaba automáticamente los *topics* a los que se debía subscribir para obtener la información que necesitaba para construiir un mapa.

Para conseguir que funcione, se debe lanzar primero la simulación de Gazebo con el mundo deseado. A continuación, se debe ejecutar el nodo *teleop_keyboard*, utilizado para mover el robot con el teclado. El último módulo que se debe ejecutar es el Cartographer, que abrirá el programa RViz. Este permite ir generando el mapa a partir de los valores devueltos por el robot. El resultado final de nodos activos se muestra en la Figura 5.1.

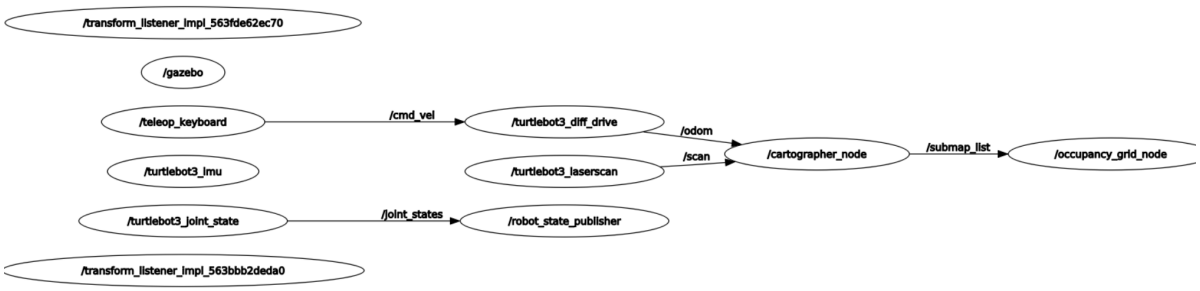


Figura 5.1. Nodos activos al ejecutar el módulo Cartographer.

En esta figura se observan, dentro de los óvalos, los nodos activos en este proceso. Destacan algunos de ellos. El nodo `/teleop_keyboard` envía instrucciones al Turtlebot a través del *topic* `/cmd_vel`. A esta información está suscrito el nodo `/turtlebot3_diff_drive` que a su vez publicará la información acerca de la odometría (`/odom`).

El nodo del cartographer, llamado `/cartographer_node` recoge esta información y la proveniente del `/turtlebot3_laserscan`, que publicada en el *topic* `/scan`, contiene la información del sensor. Con estos elementos generará el mapa, y publicará de nuevo información para que RViz sea capaz de representarlo.

Tras entender su funcionamiento, se comprobó su funcionamiento con la parte exterior del laberinto sencillo y en una primera pasada consigue captar el mapa del entorno, como se observa en la Figura 5.2. Sin embargo, analizando con un poco de detenimiento la imagen, se observa que los ángulos no son del todo rectos.

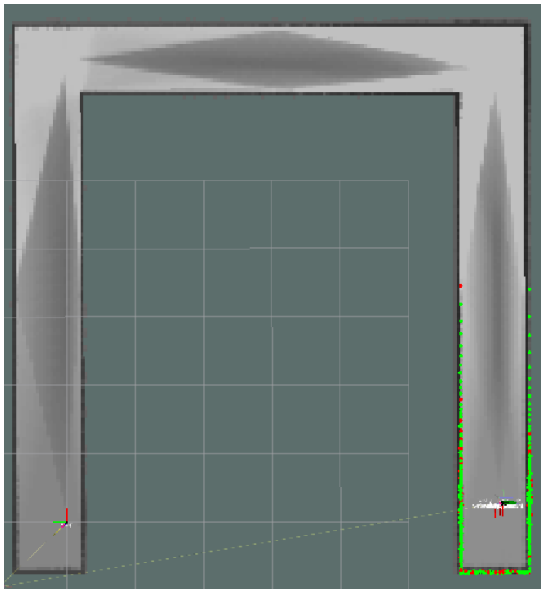


Figura 5.2. Elaboración del mapa más sencillo utilizando Cartographer.

Si se intenta llevar a cabo el mapeado del entorno en el laberinto abierto, el resultado es muy deficiente. Al recorrer un único elemento del laberinto consigue hacerlo bien, identificando correctamente el trazado realizado aunque continúa mostrando zonas de sombra en partes del trazado que deberían estar completas. En la Figura 5.3 se observa el progreso realizado hasta ese punto.

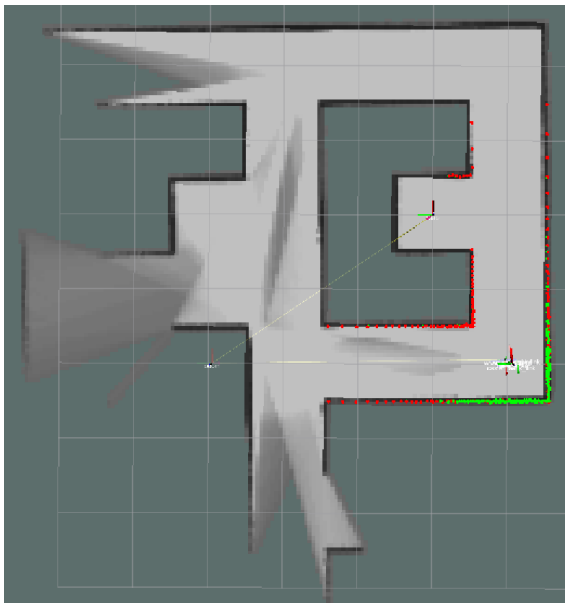


Figura 5.3. Trazado del mapa bien identificado por Cartographer.

No obstante, al intentar ir un poco más allá comienza a distorsionar el dibujo hasta el punto de no tener nada reconocible. Inicialmente la distorsión es muy pequeña y puede responder a un pequeño fallo de cálculos. No obstante, una vez ocurre esto, ya no hay forma de volver a corregir el mapa. Los primeros indicios de fallo se muestran en la Figura 5.4, donde se aprecia que algunos bordes del circuito están borrosos y parecen duplicados. Esto aparece de manera repentina durante la ejecución y se debe a que no está siendo capaz de cerrar el lazo y se está fiando de las medidas obtenidas y la odometría.

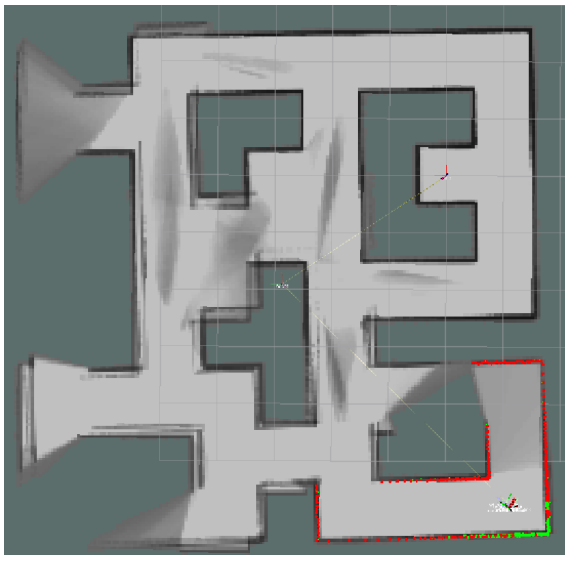


Figura 5.4. Fallo en el Cartographer.

Para mejorar esto se intentó dar más vueltas al circuito, pero solo empeoraba la situación. El resultado final obtenido para el laberinto *open* se observa en la Figura 5.5.

Por ello, se intentó modificar algún parámetro relacionado con el rango del sensor, pero las opciones que permitía modificar esta herramienta, ubicadas en el archivo "turtlebot3_lds_2d.lua", son muy limitadas, y no mejoraron los resultados. Debido a esta situación, se optó por

cambiar de herramienta para utilizar el paquete *Navigation2*, que emplea el módulo de ROS2 *slam_toolbox*.

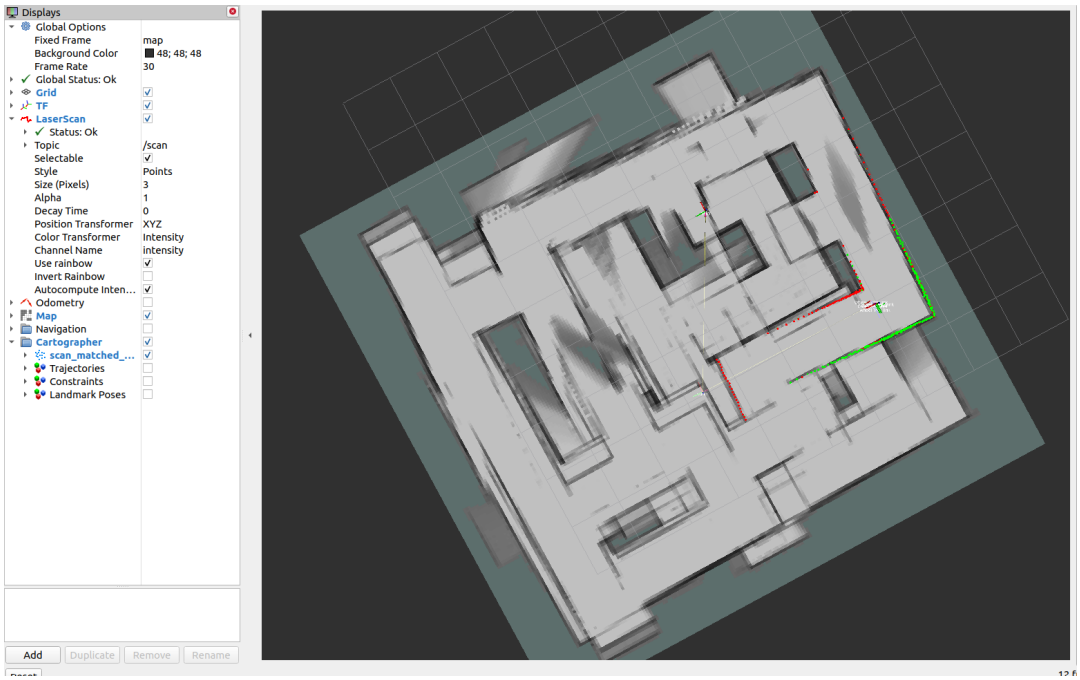


Figura 5.5. Resultado del mapeo del laberinto utilizando Cartographer.

5.2. Navigation2

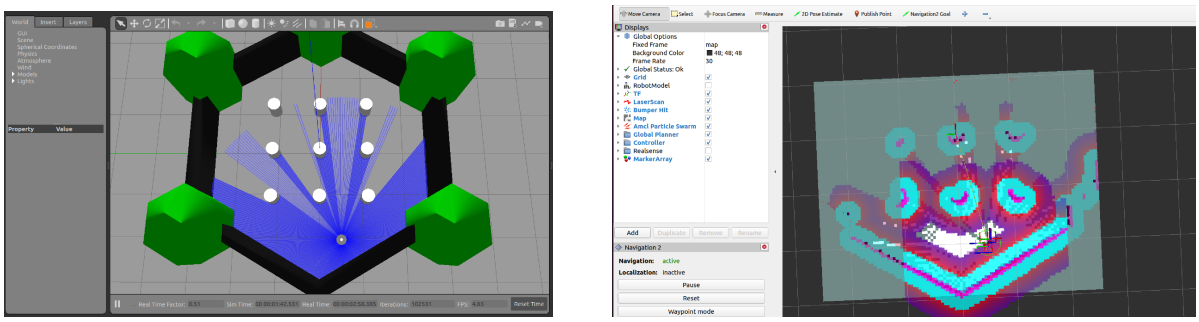
Este módulo es un paquete de ROS2 formado por un conjunto de herramientas para realizar SLAM en 2D. Se instala con el comando *sudo apt install ros-foxy-navigation2 ros-foxy-nav2-bringup*. Para utilizar la parte de SLAM, además se debe añadir el slam_toolbox [**32**], clonando el repositorio oficial[1]. Para ello hay que corregir el comando de la documentación, sustituyendo la URL indicada por la dirección real del repositorio.

Para poder utilizar este módulo se deben ejecutar distintos nodos. En primer lugar, es necesario activar la simulación del Turtlebot3 en Gazebo, con el mundo que se quiera mapear. De esta forma se empezará a publicar los datos recopilados por el sensor. A continuación se debe inicializar el módulo Navigation2 [**33**], cuyo arranque iniciará el RViz. Finalmente quedaría arrancar el `/teleop`, para poder mover el robot y el paquete del *slam_toolbox* para comenzar a crear el mapa.

Se incluye un archivo .launch.py que permite ejecutar directamente todos estos nodos y realizar la navegación, pero ambas acciones no son compatibles simultáneamente. Para seleccionar que se desea utilizar el complemento SLAM, se debe incluir el parámetro `slam:=True` al ejecutar el *script*.

El archivo de prueba, permite correr el SLAM sobre el mundo del Turtlebot3. Al lanzarlo, se abren las terminales mostradas en la Figura 5.6. En Gazebo se observa la simulación del robot, mientras que RViz recoge la información recibida por el sensor, que se publica en el *topic* `/LaserScan` y lo muestra en pantalla. Al comenzar a desplazarse, va almacenando todos estos datos hasta generar el mapa.

[1] https://github.com/stevemacenski/slam_toolbox



(a) **(b)**

Figura 5.6. Interfaces abiertas al ejecutar el .launch.py del Nav2 en modo SLAM. (a) Gazebo. (b) RViz.

El objetivo en este caso fue generar un archivo a través del cual se pudieran desempeñar estas mismas funciones utilizando los mundos construidos en el Capítulo 4. Para ello se ha decidido tomar el archivo de demostración como base y por eso el primer paso es determinar donde se ubica ese archivo y conocer su estructura.

5.2.1. Archivo launch

En el caso de la simulación del mapa en Gazebo, el archivo .launch.py se encontraba dentro del *workspace*, ya que la instalación del paquete *simulations* se realizó clonando el repositorio oficial de Github[2]. No obstante, aquí se ha llevado a cabo a través del comando de instalación de Linux.

Los paquetes de ROS se instalan por defecto en el directorio /opt/ros/foxy/share/[3] donde cada uno de los paquetes instalados tiene una carpeta propia. Entre estos paquetes se encuentran todos aquellos instalados utilizando el comando *apt install*. Entre ellos se encontraría, por ejemplo, el Cartographer, empleado en la Sección 5.1.

Estas carpetas cuentan con una distribución muy similar: todos (o casi) cuentan con una carpeta para archivos launch, otra para *worlds, maps* y otros elementos que pudieran ser necesarios.

Por tanto, el documento se ubica en nav2_bringup/launch y para garantizar que siempre va a haber un documento operativo, se hizo una copia del mismo, evitando así trabajar con el archivo original. El esquema del ejecutable es muy sencillo. Tras realizar los *imports*, se declaran las variables de configuración y se les asigna un valor. Se debe definir la transformación de coordenadas, realizado automáticamente por el nodo `/tf`.

Quedaría por definir los argumentos de lanzamiento del *script*, indicando parámetros a los que se les asigna un valor por defecto. Entre estos parámetros se encuentra el archivo de parámetros del Navigation, si utilizar SLAM... Estos valores se encuentran recogidos en la Tabla 5.1.

[2] https://github.com/ROBOTIS-GIT/turtlebot3_simulations

[3] A lo largo de este capítulo y el próximo (Navegación) cada vez que se haga referencia a un paquete se asume la ruta local a partir de este directorio a menos que se especifique lo contrario.

Declaración	Explicación
namespace_cmd	Nombre del “espacio de nombres”.
use_namespace_cmd	Si utilizar el *namespace* para la navegación.
slam_cmd	Si aplicar SLAM o directamente navegación.
map_yaml_cmd	Ruta completa al archivo del mapa.
use_sim_time_cmd	Si emplear el reloj de simulación de Gazebo.
params_file_cmd	Ruta a los archivos que parametrizan los nodos ROS2 a ejecutar.
bt_xml_cmd	Ruta al .xml que indica el comportamiento del paquete.
autostart_cmd	Si comenzar automáticamente el *stack* del Nav2.
rviz_config_file_cmd	Ruta completa al archivo de configuración de RViz.
use_simulator_cmd	Define si utilizar el simulador.
use_robot_state_pub_cmd	Si se debe inicializar el publicador de estado del robot.
use_rviz_cmd	Si se debe inicializar RViz.
simulator_cmd	Declara si es preciso inicializar el *gzclient*.
world_cmd	Establece la ruta completa al archivo *world*.

Tabla 5.1. Argumentos de lanzamiento presentes en el archivo .launch.py.

También hay que definir los procesos que se iniciarán y los nodos que habrá que arrancar, para lo que se emplearán todos los parámetros definidos, ya sea como condición o como ruta para encontrar el archivo necesario. La parte final del código añade las definiciones, acciones y nodos necesarios para el lanzamiento del programa.

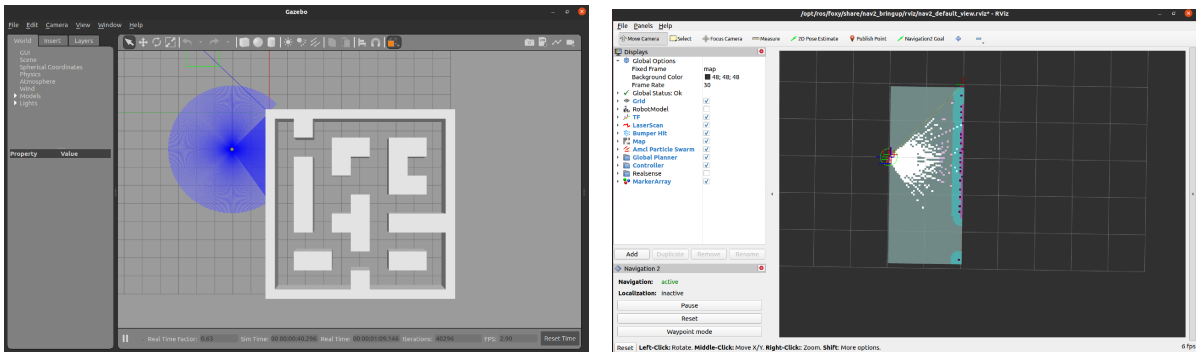
5.2.2. Ejecución

Tras estudiar la estructura del documento, se comprobó que habría que modificar varíos parámetros. Entre ellos:

- world_cmd: la ruta del mundo se debe modificar para que apunte al directorio en el que se ubica. Se pueden seguir diversos enfoques, ya que la ruta es absoluta y el archivo podría estar situado en cualquier directorio. Sin embargo, en este caso se ha optado por mantener la estructura del paquete y colocarlo dentro de la carpeta *worlds* del directorio nav2_bringup. Este archivo es el mismo que se utilizó en el Capítulo 4.
- slam_cmd: el valor por defecto de este parámetro está fijado a False. Esto quiere decir que por defecto no se ejecutará en modo SLAM sino en modo navegación. Se puede modificar, pero se ha optado por dejarlo como está ya que se entiende que será más común tener que realizar una navegación que un mapeado.

Tras modificar todos estos argumentos, se ejecuta el comando[4], y al inicializarse, los resultados de las interfaces son los mostrados en la Figura 5.7. Se observa que ha habido un pequeño problema con las cordenadas iniciales en las que se ubica el robot, pero el sistema completo está funcionando bien: RViz está suscrito a los *topics* necesarios y procesa adecuadamente la información.

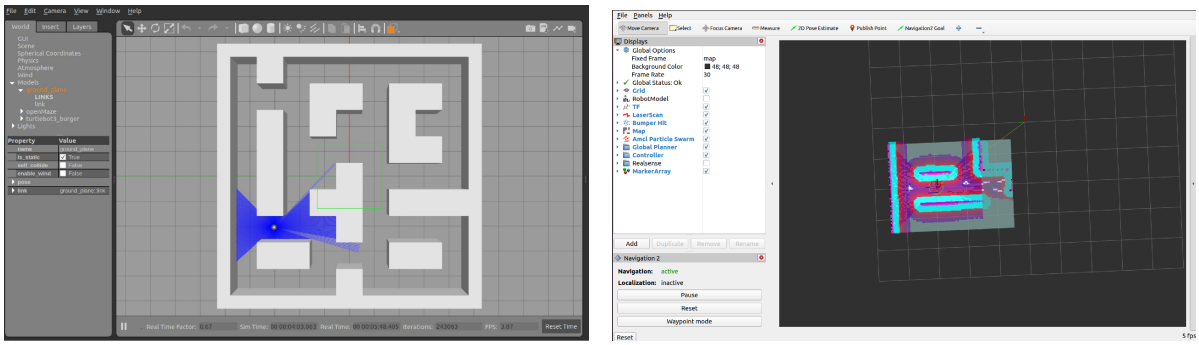
[4]A lo largo de esta sección siempre se incluirá el parámetro necesario para inicializar el SLAM.



(a) (b)

Figura 5.7. Interfaces abiertas al ejecutar el .launch.py del Nav2 en modo SLAM tras seleccionar el mundo. (a) Gazebo. (b) RViz.

Tras unas pequeñas modificaciones en las coordenadas de posición del Trutlebot3, se consiguió situar dentro del laberinto, como se muestra en la Figura 5.8. La situación inicial del robot es arbitraria, ya que la generación del mapa debería ser la misma independientemente de dónde se inicie el trayecto.



(a) (b)

Figura 5.8. Interfaces abiertas al ejecutar el .launch.py del Nav2 en modo SLAM tras modificar las coordenadas. (a) Gazebo. (b) RViz.

En este punto, se inicia el nodo de teleoperación, utilizado para mover el robot por el laberinto. Al ir desplazándose, el sensor recopilará los datos de su entorno y con ellos traza el mapa del laberinto. El proceso por el que se va trazando se muestra en la Figura 5.9.

En estas imágenes, la mitad izquierda de la pantalla representa la simulación de Gazebo, y cómo se va desplazando el robot. En la mitad derecha, la parte superior presenta la terminal desde donde se controla el Turtlebot3, mediante la ejecución del nodo de teleoperación. Finalmente, la parte inferior derecha es el visualizador RViz, donde se va creando el mapa a medida que va avanzando.

Cabe destacar que, a diferencia de lo que ocurría en la Sección 5.1 con el módulo Cartographer, en esta ocasión no ha habido ningún tipo de problema de superposición de trazados, y ha reconocido la situación en los casos en los que se producía un *loop closure*, lo que demuestra que el algoritmo es mucho mejor en el paquete slam_toolbox.

Este paquete ofrece distintos tipos de algoritmos: el SLAM síncrono y el asíncrono. El primero se caracteriza por procesar todas las medidas obtenidas ya que va almacenando los datos que

recibe. Esto que puede causar problemas si hay poca capacidad de cómputo. Puede realizarse *online* u *offline*, aunque generalmente la opción *offline* obtiene mejores resultados.

Por otra parte, el procesamiento asíncrono sigue una aproximación *best-effort*, procesando únicamente aquellos datos que es capaz de gestionar. Es decir, renuncia a toda aquella información que no es capaz de incorporar a su computación en el momento en el que llega. En este caso solo existe método *online*, ya que, al no almacenar los datos, no sería posible realizar el mapa de forma *offline*.

En la elaboración de este mapa se ha utilizado la ejecución síncrona *online*. De acuerdo con lo indicado en la documentación, este método es el que más problemas puede acarrear en caso de encontrarse ante un espacio abierto muy grande o de tener poca capacidad de procesamiento. Sin embargo, los laberintos diseñados no se caracterizan por tener amplias extensiones sin obstáculos y aunque en el Capítulo 6 si habrá incidentes relacionados con los recursos, en esta ocasión no se detectó ningún problema.

Los resultados fueron óptimos, por lo que ni siquiera se planteó la posibilidad de utilizar algún otro tipo de SLAM. Sin embargo, en caso de querer aplicarlo a algún otro mundo, es un factor a considerar en caso de no obtener los resultados esperados a lo largo de la realización del mapa.

Los pasos para poder cambiar el método de SLAM empleado son complejos. Actualmente, el archivo *launch* utilizado invoca a otro archivo responsable de la ejecución del paquete: *nav2_bringup/launch/slam_launch.py*. Desde allí ya se invoca directamente al archivo del *slam_toolbox*, que posee un archivo de lanzamiento para cada uno de los modos mencionados. En la Capítulo 7 se estudia este problema y se presenta una solución.

Una vez conseguido el mapa, es necesario guardarlo. Para ello existe un comando que permite almacenarlo en una ubicación conocida. Se decidió crear una nueva carpeta en el directorio ~/Documents/ y allí se dejaron los archivos. Al guardarlo, se crearon dos documentos distintos:

- map.yaml: contiene la información del documento, incluyendo la ubicación de la imagen, el punto de origen del laberinto o la resolución.
- map.pgm: imagen real del circuito.

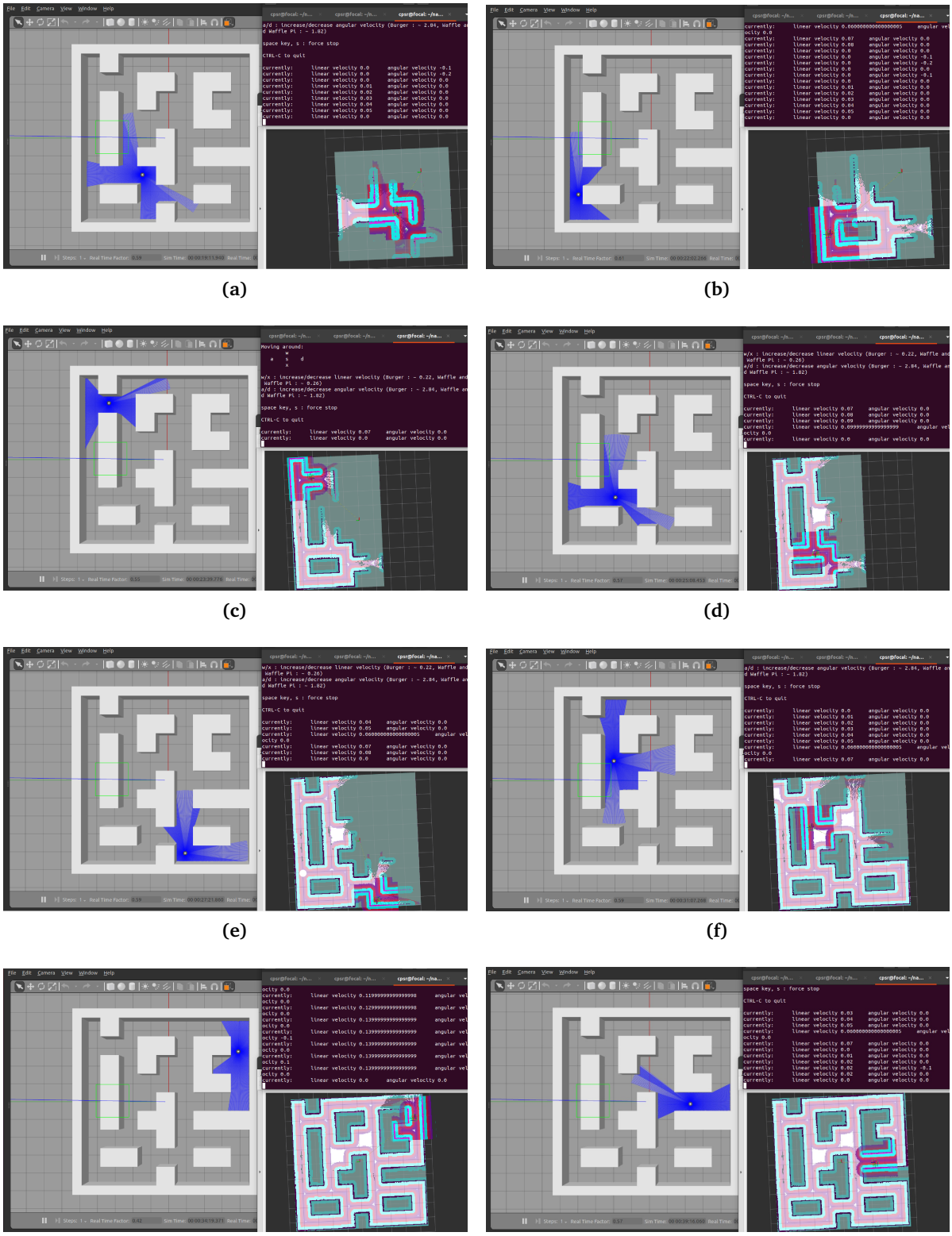


Figura 5.9. Proceso de creación del mapa del laberinto.

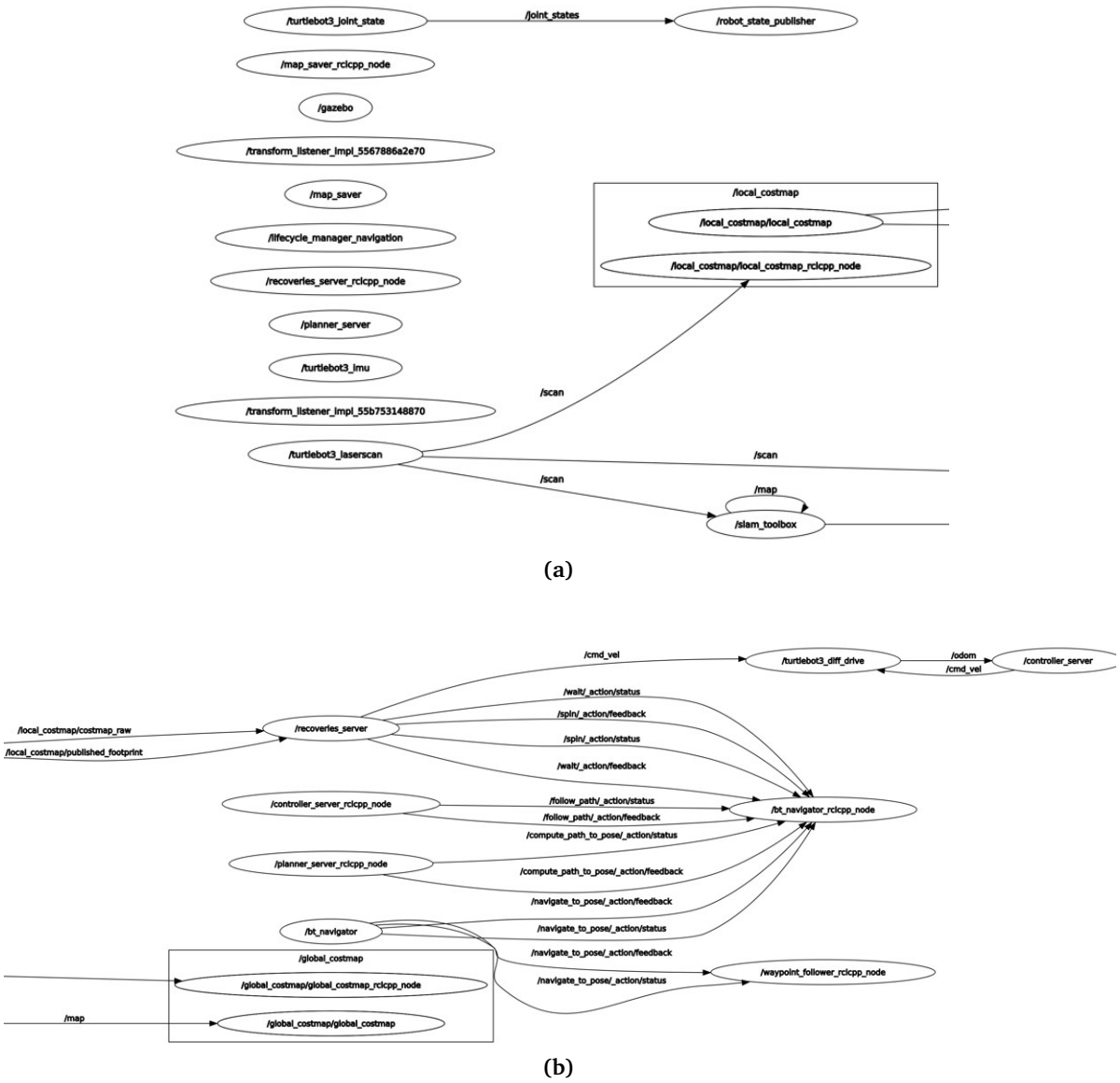


Figura 5.10. Nodos activos y conectados durante la elaboración del mapa.

En la creación de este mapa intervienen muchos nodos y por ello el esquema completo se encuentra en el Apéndice C. En esta sección se incluye una porción de ese diagrama, en la Figura 5.10. La imagen está dividida en dos partes, aunque la parte inferior (parte b) es continuación de la superior (parte a).

En la parte superior destacan los nodos `/turtlebot3_joint_state` y `/robot_state_publisher` que son los encargados de gestionar el estado y la ubicación de las distintas partes móviles del robot.

El otro nodo importante es el `/turtlebot_laserscan`, que como se comentará con mayor detalle en el Apéndice B, es el responsable de publicar los datos recibidos por el sensor. A este topic, llamado `/scan`, se encuentran suscritos varios elementos. Entre ellos el `/slam_toolbox`, responsable de la generación del mapa y los distintos mapas.

El *topic* `/map` almacena la información del mapa que, tras finalizar la navegación deberá almacenar el nodo `/map_saver`.

6

Navegación

En este capítulo se detalla el trabajo realizado en cuanto a la navegación del robot, tras haber generado el mapa necesario para desplazarse por el entorno diseñado. Se describe el paquete utilizado, los motivos de esta elección y los resultados obtenidos.

Siguiendo el mismo procedimiento que en capítulos anteriores, se ha comenzado por ejecutar el tutorial. En esta ocasión, se optó por no seguir las indicaciones de Robotis, ya que utilizan un código propio para llevar a cabo la navegación. En su lugar, se han utilizado las instrucciones del módulo Nav2 [**33**], ya usado para conseguir el mapa en el Capítulo 5.

El funcionamiento de este módulo es ligeramente diferente a lo explicado hasta el momento. Por su naturaleza, las acciones que debe realizar no son inmediatas, ya que la navegación, por ejemplo, requiere tiempo para poder llevarse a cabo. Esto provoca que sea necesario trabajar con métodos asíncronos, incluyendo futuros y *threads*. Estos elementos, llamados servidores de acción permiten recuperar estados y resultados de manera síncrona empleando funciones *callback* o asíncrona mediante futuros.

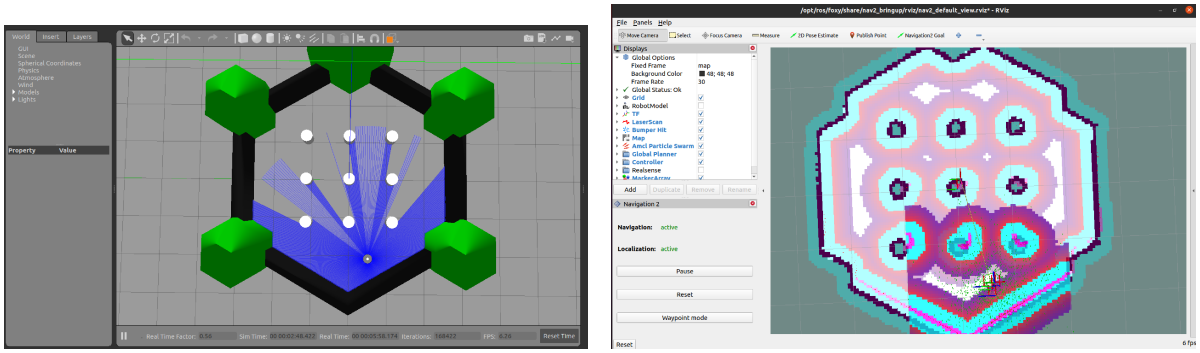
En estos servidores de acción surgen nodos para controlar el ciclo de vida (*lifecycle nodes*), que contienen la máquina de estados del servidor. El otro elemento distintivo son los BT (*behavior trees*) que son cada vez más comunes en tareas robóticas complejas. Generan marcos de trabajo más escalables y entendibles para los humanos, a través de los cuales se definen fácilmente aplicaciones con muchos pasos y estados. Nav2 está diseñado de esta manera para facilitar que se pueda acoplar a proyectos mayores de forma sencilla.

A nivel práctico, para poder ejecutar la navegación se deben fijar dos variables de entorno: el modelo del robot que se va a utilizar y la ruta en la que se encuentran los modelos de Gazebo. A continuación se lanza el paquete *nav2_bringup* con el mismo archivo que se utilizaba en el tutorial anterior pero sin incluir el parámetro *slam*. Automáticamente se cargan las pantallas mostradas en la Figura 6.1.

La imagen de RViz no es exactamente la primera que aparece al arrancar la simulación. En el momento inical, el robot carga el mapa que tiene guardado, pero no tiene forma de saber en qué punto del mismo se encuentra. Por ello, inicialmente aparece un mapa en blanco, sin

ubicación del robot. Para conseguir que empiece a funcionar y que aparezca lo mostrado en la imagen, es preciso fijar la ubicación inicial con el botón “2D Pose Estimate”.

A pesar de lo sencillo que puede parecer, se tardó en conseguir tener esto en funcionamiento. El motivo es que las instrucciones oficiales indican que es necesario presionar el botón “Startup” para poder trabajar con la simulación, pero inicialmente el botón aparece deshabilitado. Esto se debe a que, en el tutorial, asumen que el RViz se arranca en un punto previo, en el que ni siquiera aparece el mapa. Este detalle no aparece representado en ninguna captura de pantalla y tampoco queda claramente especificado en el texto.



(a) **(b)**

Figura 6.1. Interfaz del tutorial de navegación. (a) Gazebo. (b) RViz.

En el RViz, se puede observar el mapa general de fondo y resaltado en otro color aquellos datos que está recibiendo el sensor y que le permiten conocer la orientación en la que se encuentra. A continuación, si se presiona en el botón “Navigation2 Goal” se le puede asignar un punto de destino. En caso de dejar presionado en ese lugar, aparece una flecha verde que permite indicar la dirección en la que quedará orientado.

Al hacer eso, el sistema calcula el camino que debe seguir para llegar hasta allí y lo indica con una fina línea. Finalmente, el robot deberá desplazarse hasta llegar al punto indicado. No obstante, esto no ocurría cada vez que se ejecutaba. Al revisar el estado de la máquina, se comprobó que, asignando un único núcleo a la máquina, este estaría al 100 % de su capacidad.

Cabe recordar, que la ejecución de todas las simulaciones se realiza en una máquina virtual sobre VirtualBox, con un sistema operativo Ubuntu 20.04. La gestión de recursos de la máquina virtual se realiza a través de este programa, pero al tratarse de un elemento virtualizado, las capacidades se ven reducidas respecto a las que ofrecería un ordenador real. Contando con estas limitaciones, se aumentó a tres el número de núcleos disponibles.En esta ocasión, el robot se desplazó una única vez hasta el punto de destino, pero la utilización de recursos seguía por encima del 90 % (Figura 6.2). Para intentar mejorar la situación se añadió un núcleo más a la máquina.

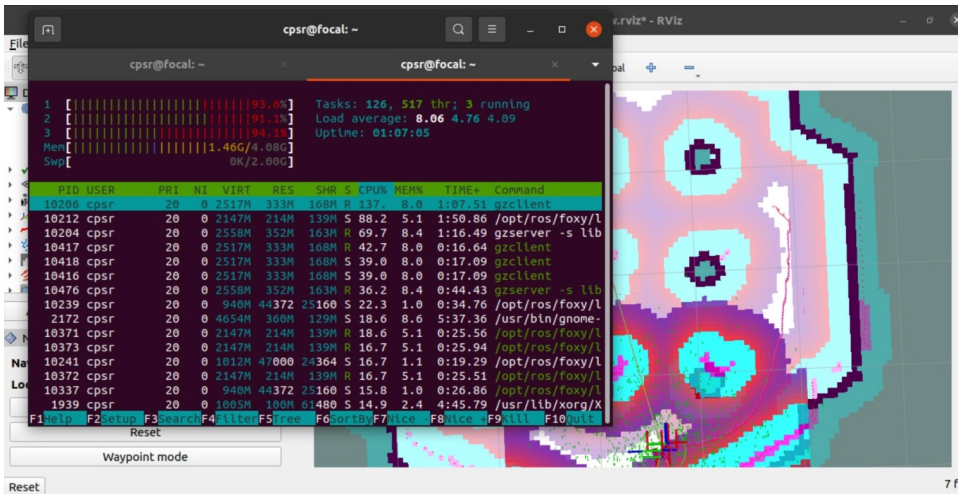


Figura 6.2. Consumo de la máquina al ejecutar la navegación.

Tras hacer funcionar este *script* inicial, se pasó a ejecutar el entorno deseado. Se ha optado por utilizar el mismo archivo generado en el capítulo anterior, siguiendo el mismo procedimiento que el ejemplo. Para ello, hubo que modificar un parámetro de los mostrados en la Tabla 5.1: map_yaml_cmd. En este parámetro se debe incluir la ruta al mapa del laberinto.

El parámetro bt_xml_cmd determina algunas de las características que debe seguir el proceso al ser ejecutado. El árbol de comportamiento que se utiliza por defecto recalcula el camino cada metro recorrido. Sin embargo, existen otros archivos ya preparados que recalculan en función del tiempo o de la velocidad.

Una vez conseguido el mapa del laberinto, se almacenó en una carpeta en el directorio "Documentos" de la máquina virtual. Se podría dejar ahí, para asegurar que el documento está localizado. No obstante, como se mencionó anteriormente, se ha intentado seguir la estructura del paquete, por lo que se movieron ambos archivos a su directorio correspondiente: */opt/ros/foxy/share/nav2_bringup/maps* y se hizo referencia a esta ruta en el *launch*.

Al ejecutarlo, Gazebo se inicializaba correctamente, sin embargo, el RViz no mostraba ningún mapa. De nuevo, no había un mensaje de error que inidcara qué estaba sucediendo. Se comprobó que la ruta al mapa desde el archivo launch estaba bien, ya que si se eliminaban los archivos de esa carpeta, sí aparecía un error claro indicando que no se podía encontrar el archivo.

Puesto que el error también podía estar en el mapa, se intentó abrir el archivo .pgm utilizando otras aplicaciones para comprobar que el mapa estaba bien creado, pero no resultó sencillo. Las siglas PGM hacen referencia a mapa portátil en grises (*Portable Gray Map*) por lo que se abre utilizando editores de texto. Este formato contiene imágenes codificadas en un formato legible para el ser humano. El archivo se divide en dos partes fundamentales: la cabecera, que contiene una visión general del contenido de la imagen y el cuerpo, que incluye la información de cada pixel de la imagen en escala de grises. La Figura 6.3 muestra la imagen del laberinto, abierta con un visualizador de imágenes en lugar de con un editor de texto.

En última instancia, se leyó el archivo .yaml ya que todo lo demás parecía estar correcto. En este archivo aparece una ruta absoluta a la imagen (Figura 6.4). Se modificó para que apuntara directamente a la misma carpeta en la que estaba el archivo .yaml y así pudiera encontrar fácilmente el .pgm.

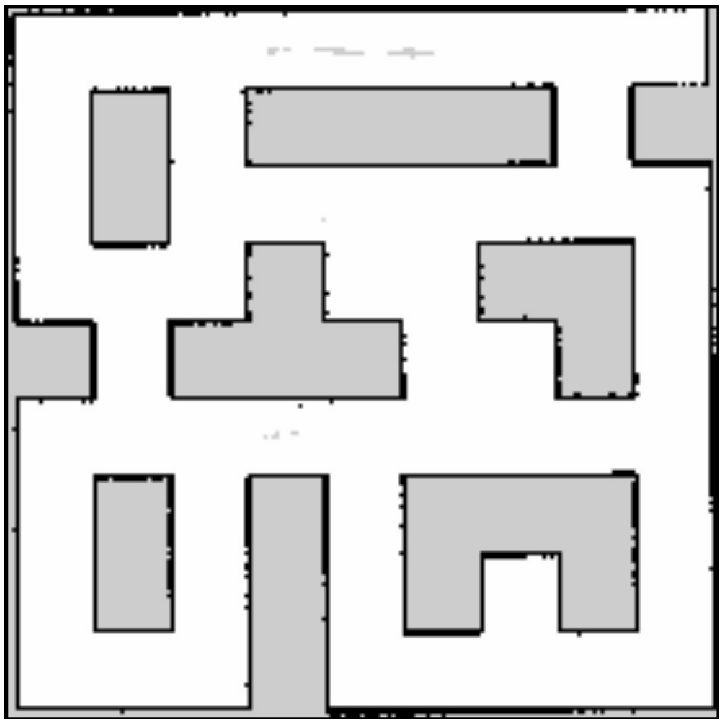


Figura 6.3. Diseño del mapa generado a través del SLAM.

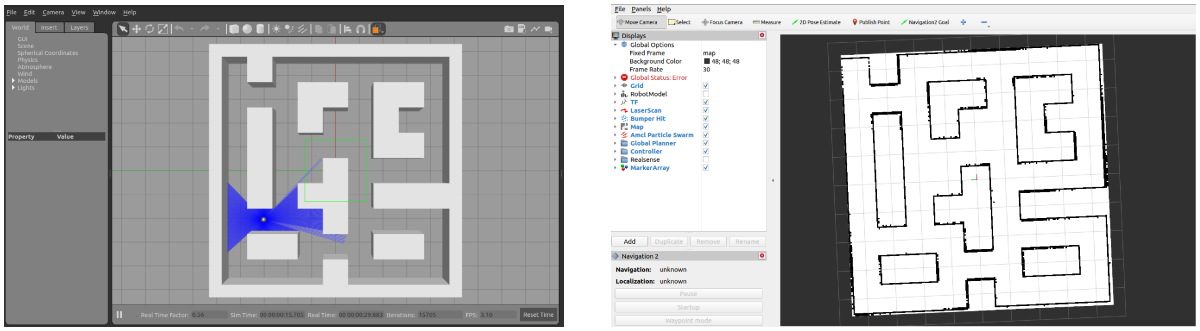
map.yaml

opt > ros > foxy > share > nav2_bringup > maps > openMaze > map.yaml

```
1  image: /opt/ros/foxy/share/nav2_bringup/maps/openMaze/map.pgm
2  mode: trinary
3  resolution: 0.05
4  origin: [-4.59, -4.59, 0]
5  negate: 0
6  occupied_thresh: 0.65
7  free_thresh: 0.25
```

Figura 6.4. Contenido del archivo map.yaml, resaltando el campo de la ruta a la imagen del mapa.

Tras hacer esta modificación, se volvió a ejecutar el archivo *launch* y en esta ocasión sí funcionó correctamente. Las interfaces iniciadas con esta ejecución son mostradas en la Figura 6.5. En este caso se aprecia cómo el botón “Startup” aparece deshabilitado, pero igual que en el caso del tutorial, no es necesario presionarlo.



(a) **(b)**

Figura 6.5. Interfaces mostradas al inicializar la navegación con el entorno del laberinto abierto.

Se le debe indicar a la simulación de RViz la posición aproximada en la que se encuentra, y la pantalla pasará a representar al robot en su posición, remarcando sobre el mapa los valores percibidos por el sensor. Esto se puede observar en la Figura 6.6.

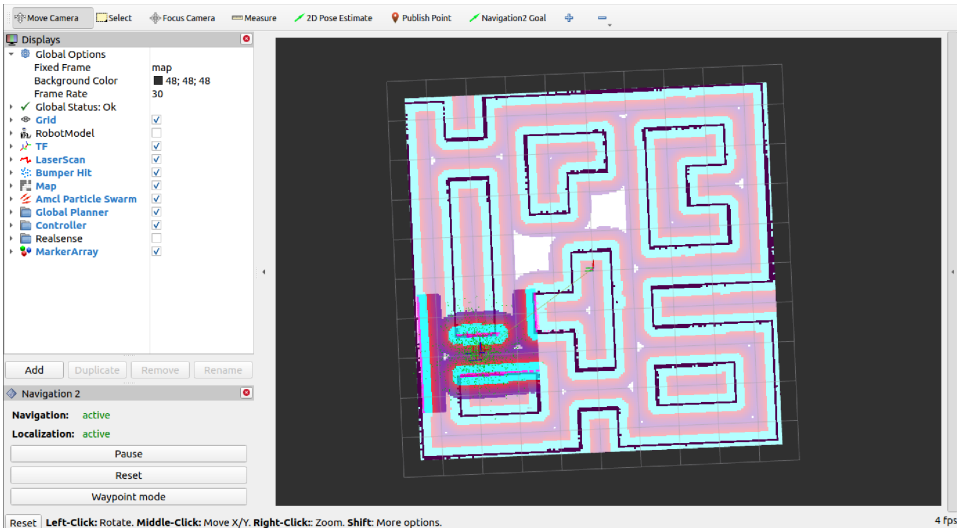


Figura 6.6. Interfaz de RViz tras proporcionar la ubicación inicial del robot.

Para ejecutar la navegación, solo quedaría establecerle un punto destino en el laberinto. Automáticamente calculará la ruta que deberá seguir para llegar hasta allí y debería dirigirse a su destino. En la Figura 6.7 se muestra el camino diseñado para ir desde su punto origen hasta dentro de la "U".

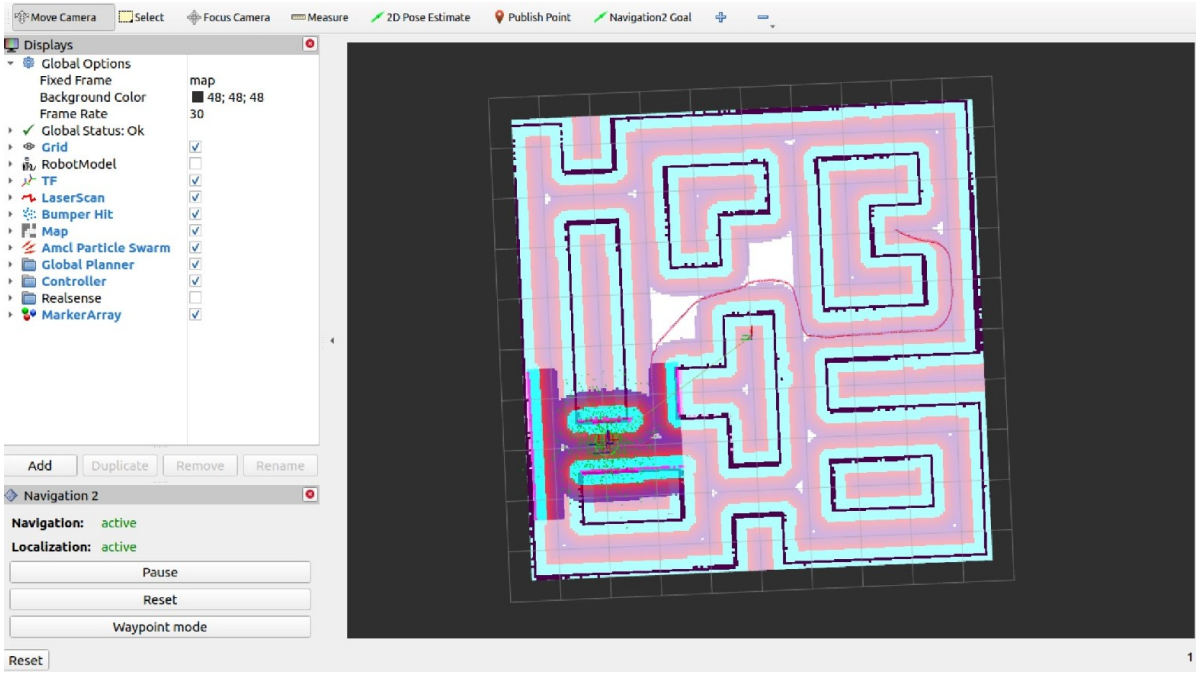


Figura 6.7. Ruta elaborada por el Turtlebot3 para llegar al punto de destino.

No obstante, debido al problema de asignación de recursos, es frecuente que no llegue a recorrer la ruta diseñada. En la Figura 6.2 se muestran los niveles de ocupación de la máquina al llevar a cabo la navegación en el entorno por defecto con tres núcleos asignados a la máquina virtual. Puesto que el mundo sobre el que se va a ejecutar la navegación en esta ocasión es

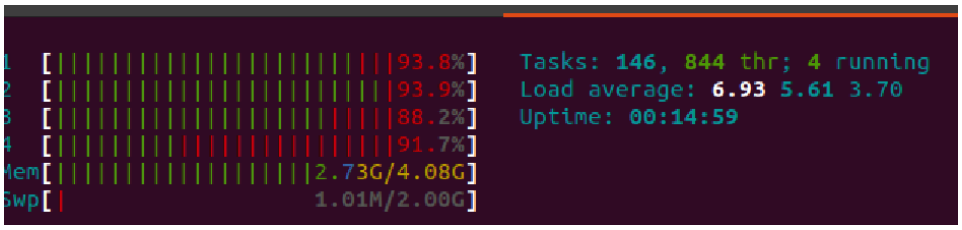
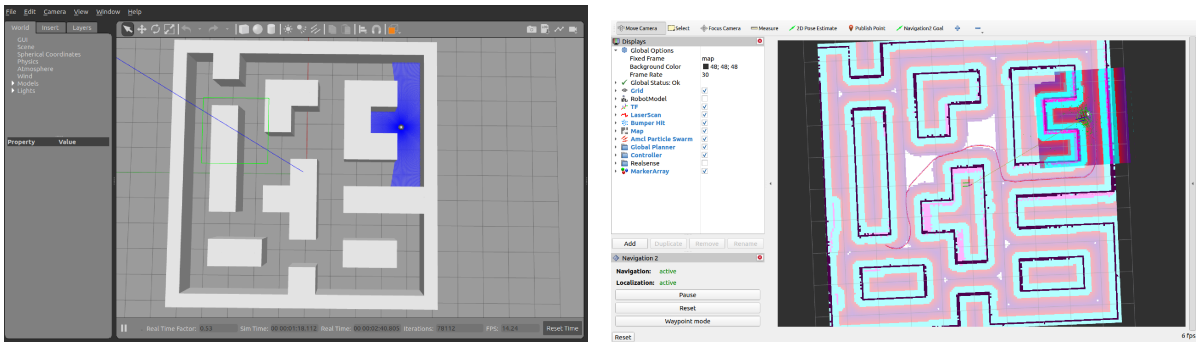


Figura 6.8. Consumo de la máquina tras ejecutar la navegación en el entorno *open*.

más pesado, se decidió asignar un cuarto núcleo, llegando así al máximo permitido por el portátil.

Con estas condiciones, al ejecutar la simulación, los recursos continúan estando en condiciones críticas, estando la utilización de los cuatro por encima del 90 % la mayor parte del tiempo. A pesar de esta situación, en ocasiones sí llega a completar la tarea, recorriendo el camino diseñado hasta alcanzar el objetivo.



(a) **(b)**

Figura 6.9. Interfaces mostradas tras realizar la navegación.

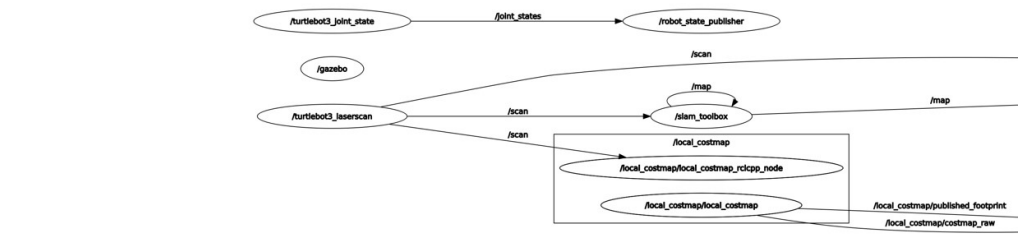
Existe un parámetro en la configuración de la navegación que permite determinar cada cuánto tiempo o espacio recorrido se debe recalcular la ruta. Al reducir este parámetro se consiguió mejorar ligeramente el estado de los núcleos pero la diferencia no fue significativa ya que el principal problema es el simulador Gazebo.

Igual que ocurría en la Subección 5.2.2, el diagrama de nodos es demasiado grande como para incluirlo, por lo que se puede revisar en el Apéndice C. En esta ocasión se ha tomado la parte más relevante del diagrama, Figura 6.10 y se ha dividido en dos partes, siendo la inferior (parte b) continuación de la superior (parte a).

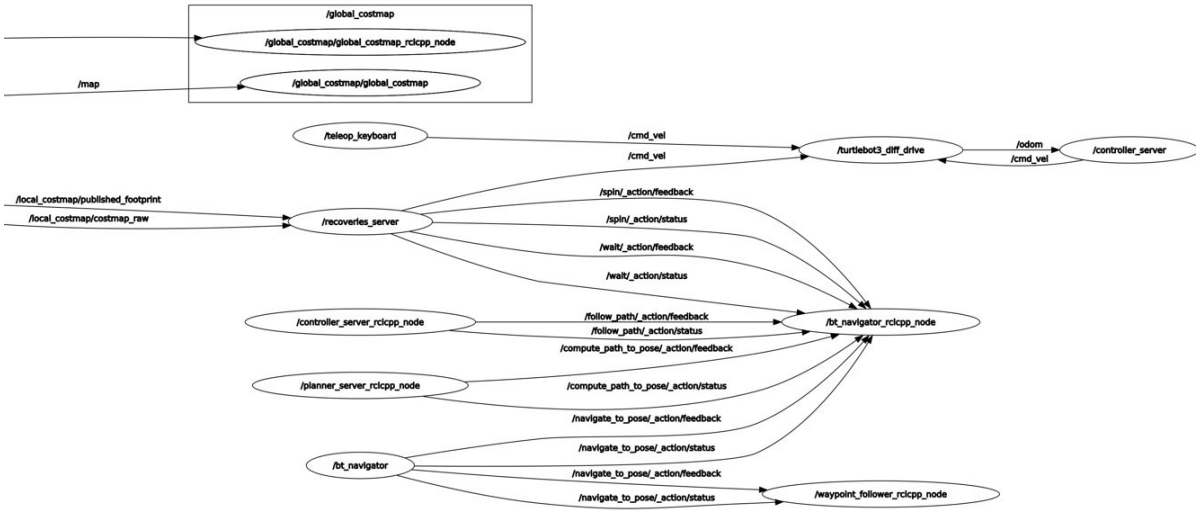
En esta ocasión vuleven a aparecer los nodos `/turtlebot3_joint_state` y `/robot_state_publisher` que continúan cumpliendo la misma función.

Por otra parte, la información recogida por el laser y procesada por el mapa local, llega al nodo `/recoveries_server`, quien enviará las instrucciones de movimiento al `/turtlebo3_diff_drive`, responsable del movimiento del robot. En este caso el nodo `/teleop_keyboard` se encuentra activo, pero no está realizando ninguna función.

Se pueden observar también los nodos responsables de la retroalimentación del sistema, que son los encargados de recalcular la ruta en caso de que surgiera algún obstáculo imprevisto.



(a)



(b)

Figura 6.10. Nodos activos y conectados durante la elaboración del mapa.

6.1. Reacción ante obstáculos

Como se comentó en el Capítulo 2, los robots AMR son capaces de detectar obstáculos inesperados y esquivarlos. Además, se sabe que existe este parámetro responsable de determinar cada cuanto tiempo se comprueba la ruta, lo que confirma que este paquete permite simular este tipo de operaciones.

Para comprobar su funcionamiento, se decidió hacer una última prueba: incorporar un nuevo obstáculo a la navegación del robot que no estuviera contemplado en el mapa inicial. Dado que no se pueden añadir elementos en el momento de la simulación, se creó una segunda versión del laberinto abierto con modificaciones respecto al trazado original. La nueva versión del laberinto está representada en la Figura 6.11, donde aparecen en verde los bloques que no figuraban en el original.

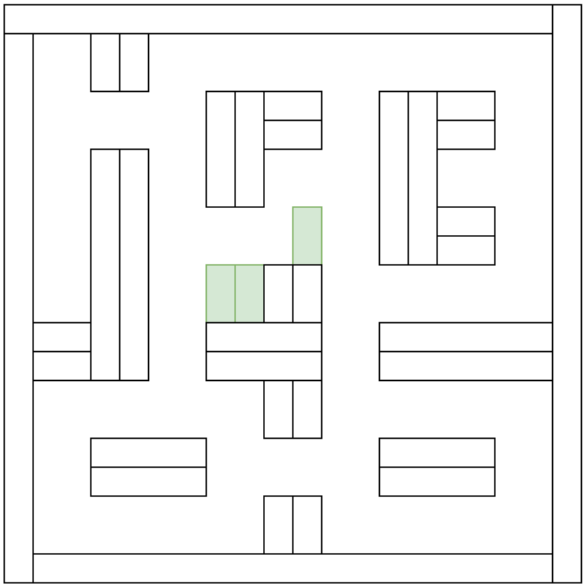


Figura 6.11. Diseño del laberinto con obstáculos.

Para conseguir el nuevo trazado, se intentó modificar el archivo .sdf original empleando el editor gráfico de Gazebo con el que se construyó (Capítulo 4). No obstante, este modo constructor solo permite comenzar modelos de cero, por lo que hubo que explorar otros métodos. Finalmente se optó por modificar el archivo a mano, generando nuevas paredes idénticas a las que forman el modelo y editando únicamente su tamaño, posición y color.

La parte más ardua fue encontrar el eje de coordenadas y su orientación, puesto que las coordenadas que presentaban los elementos eran distintas de aquellas que se habían introducido al generar el mapa: el centro se había desplazado y rotado. Para ubicar de nuevo las posiciones, se realizaron un par de pruebas antes de intentar introducir los componentes.

Para ejecutar este nuevo escenario, se crearon los archivos pertinentes, y se indicó al archivo de lanzamiento que habría de levantar el nuevo mundo con el mapa generado para el caso del laberinto normal. Al levantar sus interfaces, el usuario observa la Figura 6.12.

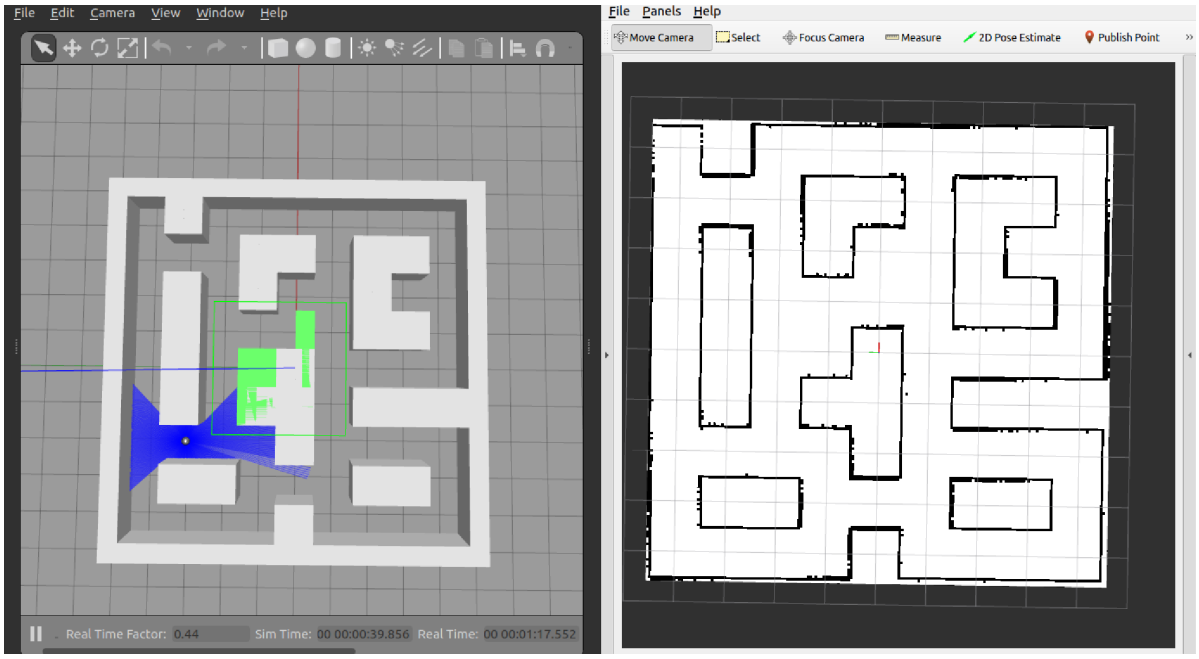


Figura 6.12. Interfaces de Gazebo y RViz al incluir el mundo con obstáculos y el mapa anterior.

Se introduce la posición inicial y el punto de destino, de forma que el robot traza la ruta con normalidad, muy parecido a lo que se muestra en la Figura 6.7[1]. La ruta inicial atraviesa la pared, de forma que solo hay dos opciones: o la corrige, o se choca. Al comenzar su movimiento, se puede observar cómo el sensor detecta que la pared continúa extendiéndose a pesar de no estar representado en el mapa (Figura 6.13).

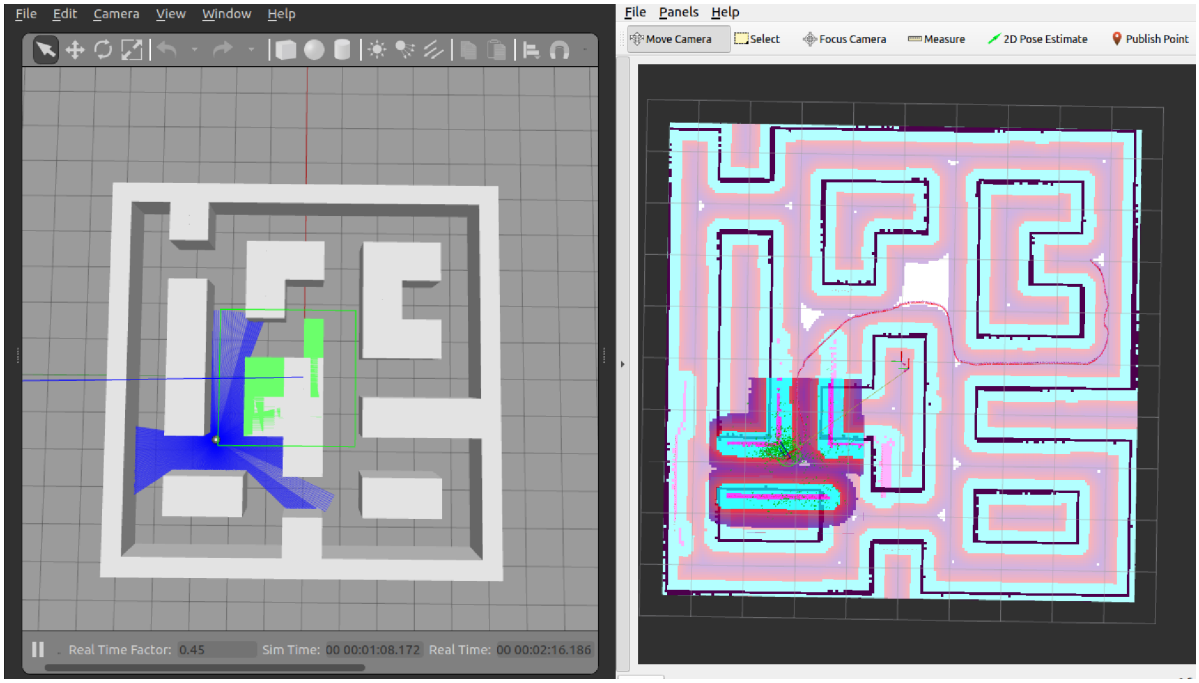


Figura 6.13. Interfaces de Gazebo y RViz al incluir el mundo con obstáculos y el mapa anterior.

Sin embargo, esta tarea no se pudo completar en local, ya que el ordenador no tiene los recursos necesarios. Para poder simularlo correctamente, el entorno virtualizado necesitaría

[1]Solo parecido y no igual porque se ha fijado un punto de destino diferente.

```
[rviz2-4] Start navigation
[rviz2-4] [ERROR] [1642782080.584817848] [rviz2]: Send goal call failed
[bt_navigator-11] [INFO] [1642782080.611419386] [bt_navigator]: Begin navigating from current location to (2.01, -3.63)
[controller_server-8] [INFO] [1642782080.646127873] [controller_server]: Received a goal, begin computing control effort.
[rviz2-4] [INFO] [1642782080.818878290] [rviz2]: Message Filter dropping message: frame 'base_scan' at time 50,674 for reason 'Unknown'
[controller_server-8] [WARN] [1642782081.064374268] [controller_server]: Control loop missed its desired rate of 20.0000Hz
[rviz2-4] [INFO] [1642782081.281646426] [rviz2]: Message Filter dropping message: frame 'base_scan' at time 50,877 for reason 'Unknown'
[rviz2-4] [INFO] [1642782081.574918714] [rviz2]: Message Filter dropping message: frame 'base_scan' at time 51,077 for reason 'Unknown'
[controller_server-8] [WARN] [1642782082.053167405] [controller_server]: Control loop missed its desired rate of 20.0000Hz
[controller_server-8] [INFO] [1642782082.117751076] [controller_server]: Passing new path to controller.
[controller_server-8] [WARN] [1642782083.063622020] [controller_server]: Control loop missed its desired rate of 20.0000Hz
[bt_navigator-11] [ERROR] [1642782083.171493985] [bt_navigator]: Action server failed while executing action callback: "send_goal failed"
[bt_navigator-11] [WARN] [1642782083.171772682] [bt_navigator]: [navigate_to_pose] [ActionServer] Aborting handle.
[controller_server-8] [INFO] [1642782083.194909722] [controller_server]: Passing new path to controller.
[controller_server-8] [ERROR] [1642782087.284678787] [DWBLocalPlanner]: No valid trajectories out of 419!
[controller_server-8] [ERROR] [1642782087.284805615] [DWBLocalPlanner]: 1.00: BaseObstacle/Trajectory Hits Obstacle.
[controller_server-8] [ERROR] [1642782087.285106697] [controller_server]: No valid trajectories out of 419!
[controller_server-8] [WARN] [1642782087.285157929] [controller_server_rclcpp_node]: [follow_path] [ActionServer] Aborting handle.
```

Figura 6.14. Error mostrado tras iniciar la navegación debido a falta de recursos.

entre cuatro y ocho núcleos, ya que el simulador, Gazebo, consume mucho. El error obtenido en local, mostrado en la Figura 6.14 ratifica que ese es el problema principal.

Para poder comprobar el funcionamiento real de la navegación con obstáculos, se recurrió a una máquina virtual mucho mayor, ubicada en un *datacenter*. Esta máquina, cuenta con 10 núcleos y un sistema operativo Ubuntu 20.04, por lo que presentaba las características necesarias para replicar el entorno. Se subió el paquete a un repositorio git y, tras instalar las dependencias necesarias, se ejecutó. La navegación tuvo lugar sin ningún tipo de percance, y el Turtlebot3 llegó a su destino.

La Figura 6.15 meustra este punto de llegada tras la ejecución. Resulta llamativa la actualización del mapa, que si bien no ha sido realmente modificado, muestra en rosa claro los obstáculos. A la hora de pedir que vuelva al punto de partida, el algoritmo modifica la ruta, ya que considerando los nuevos obstáculos, el camino más directo sería otro.

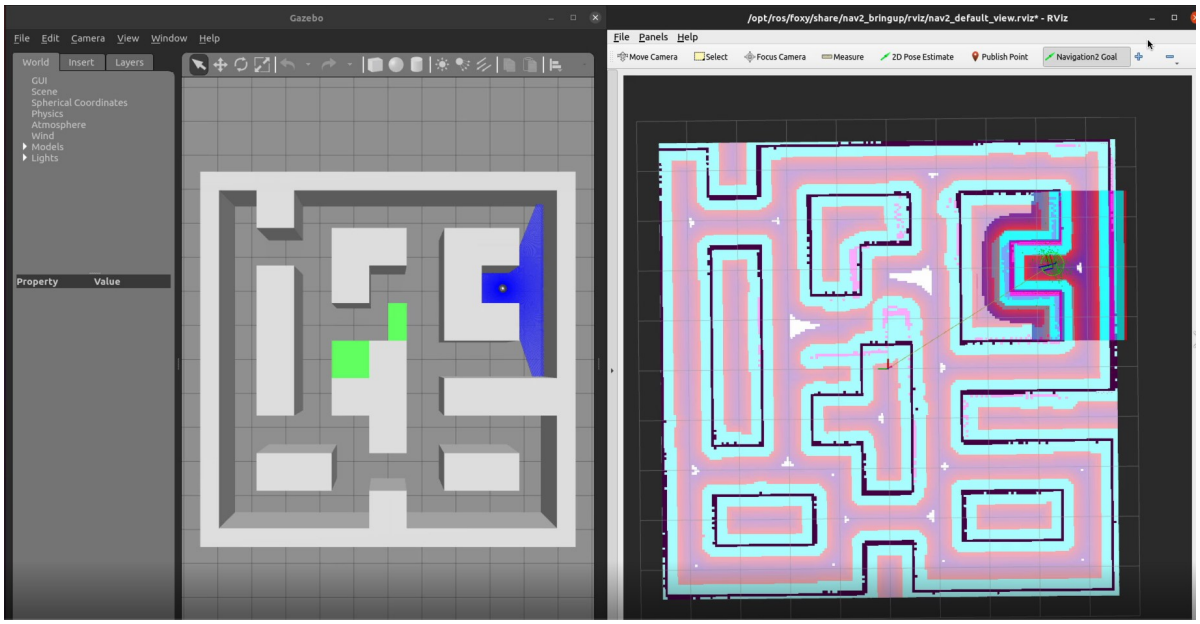


Figura 6.15. Representación de la ruta tras haber concluido la navegación con obstáculos.

7

Sistema de archivos

Este capítulo se describe los pasos a seguir por el usuario para ejecutar el sistema siguiendo las indicaciones que se han ido dando a lo largo del documento y se describe la nueva organización del sistema y su puesta en marcha.

Hasta este punto se ha ido detallando en cada capítulo cómo ejecutar cada una de las acciones a realizar. En todos los casos existen dos opciones: ejecutar los nodos uno a uno en distintas terminales o utilizar un archivo de lanzamiento que permita levantarlos todos desde una misma consola.

La primera aproximación fue ejecutar los nodos por separado. Esto proporciona mayor control sobre las ejecuciones y hace que el usuario sea realmente consciente de lo que está realizando. Esta forma de proceder puede ser adecuada a la hora de aprender cómo funciona, pero de cara a conseguir aplicaciones un poco más elevadas termina resultando poco práctico, lento y engorroso. Además, por el diseño que tienen los nodos, se exige que se fije por lo menos una variable de entorno para cada uno de ellos, lo que provoca que el usuario esté constantemente realizando esta tarea.

El segundo método estudiado es el de ejecutar un archivo que se encargue de inicializar todos los nodos necesarios a partir de unos parámetros prefijados. Este archivo, programado en Python y cuya estructura fue descrita en la Subección 5.2.1 sigue necesitando las variables de entorno para funcionar. No obstante, al lanzar todos los nodos desde un mismo ejecutable, solo será necesario incluirlas dos veces.

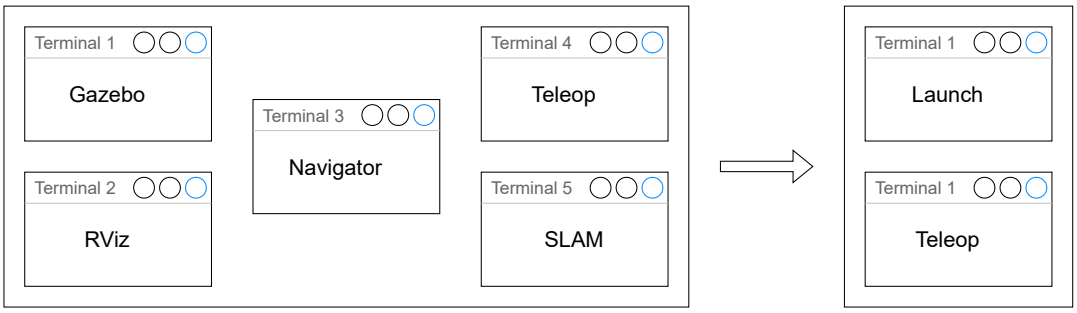


Figura 7.1. Esquema básico de un entorno de trabajo.

En la Figura 7.1 se representa de forma gráfica la mejora que supone la utilización del archivo de lanzamiento. Destaca el hecho de que, incluso tras haber mejorado la situación sigue habiendo que arrancar dos terminales distintas. El motivo de esto es que la segunda terminal final es el nodo de teleoperación, que requiere la introducción de datos por parte del usuario. Esto dificulta que se pueda arrancar en el mismo momento que todos los demás, ya que pueden producirse errores al mezclar los *logs* con los *inputs*.

La aproximación seguida hasta este momento en el proyecto, ha sido añadir archivos *launch* a paquetes preexistentes. El paquete al que se han ido añadiendo archivos es el nav2_bringup, cuya ubicación está en la carpeta */opt/ros/foxy/share*. Esta es una ruta de acceso restringido, para la que se necesita un usuario con permisos para poder modificar documentos, lo que debería dar alguna pista de que esa forma de operación no es recomendable. Además, aunque no se realicen modificaciones directas, se está cambiando la estructura de un paquete oficial y modificar algo incorrecto podría llevar a causar que dejara de funcionar.

Para evitar esta situación surgen dos alternativas. La primera es mucho más sencilla, pero menos recomendable. En esta ocasión se propone situar el archivo *launch* aislado, con referencias a la ubicación de los archivos world, a los maps y a los modelos. De esta forma no se asocia el archivo a un paquete y ROS2 no lo gestiona. Para poder ejecutarlo, será necesario incluir la ruta al archivo en el comando para asegurar que todo se lleva a cabo correctamente. Esta opción es válida puesto que no se inicializa ningún nodo propio.

La otra opción es crear un paquete para gestionarlo. Dentro de un espacio de trabajo hay cuatro carpetas principales, como se muestra en la Figura 7.2, donde la más importante de cara a la gestión manual de archivos es *src/*. Dentro de este directorio se encuentran los distintos paquetes, cada uno en una carpeta diferente.

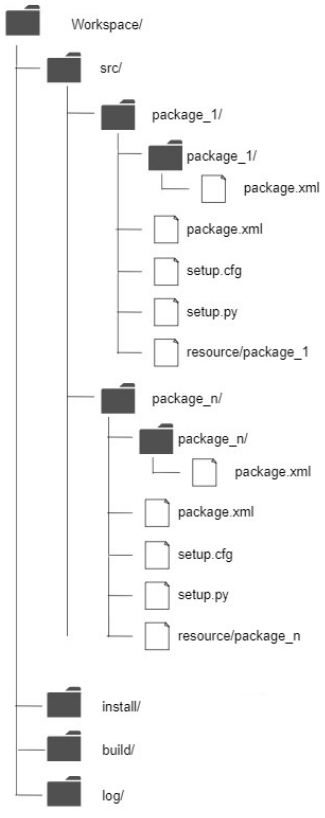


Figura 7.2. Esquema básico de un entorno de trabajo.

Los paquetes vienen definidos por tres archivos principales: package.xml, setup.cfg y setup.py. El primero contiene la información del paquete y allí se debe incluir la información acerca de las dependencias a ejecutar, y los nodos propios que se incluyan. El segundo archivo, no es necesario modificarlo, e indica la ruta donde estarán los distintos archivos. El último de todos es más importante, y hay que hacer más cambios si se pretende que ROS2 pueda acceder a todos los paquetes sin necesidad de especificar las rutas.

Para comenzar, este archivo también debe contener la misma meta información del paquete que package.xml en cuanto a versión, paquete y mantenedor. El campo que se debe modificar es el de *data_files*. En él se debe especificar que existe un directorio donde se almacenarán los archivos *launch* y se debe modificar para que sea capaz de identificarlo e invocar los archivos que se encuentren en él.

Tras llevar a cabo esta modificación, solo queda añadir los archivos necesarios para ejecutar cualquiera de los mundos. Esto implica que hay que incluir los archivos *worlds*, los mapas y el modelo .urdf de los robots que se quieran utilizar. Para ello se ha seguido el mismo formato que tienen otros paquetes oficiales de ROS2, haciendo que el esquema del paquete sea el mostrado en la Figura 7.3.

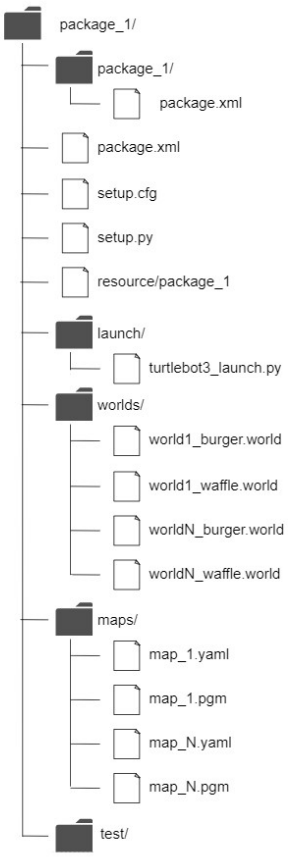


Figura 7.3. Esquema final del paquete creado.

Este paquete mayoritariamente invocará archivos de otros paquetes para poder ejecutar todas las acciones deseadas. Se podrían clonar desde el repositorio directamente al entorno de trabajo. De esta forma, estarían accesibles y se podrían realizar fácilmente modificaciones. Esto es importante de cara a trabajar en otro tipo de entornos o utilizar otros modelos de robot, ya que en ese caso puede ser necesario modificar parámetros de configuración.

Se puede comprobar que el paquete cuenta con un único archivo de lanzamiento, pero hay varios mundos y mapas a disposición en él. Para facilitar más la ejecución, se ha creado un archivo en *shell scripting* encargado de fijar las variables de entorno. Este se debe ejecutar cada vez que se abra un terminal nuevo, es decir, tanto en el caso de ejecución del *launch* general, como en el del teleop.

Con esta configuración y los archivos introducidos en el paquete, es posible ejecutar las distintas simulaciones con dos modelos de Turtlebot3 diferentes: el Waffle y el Burger. Además de las variables requeridas por los paquetes ya mencionados, se ha introducido una nueva: WORLD_NAME cuya función principal consiste en seleccionar el mundo sobre el que se va a ejecutar la simulación, ya sea de SLAM o de navegación.

Para que esta nueva característica funcione correctamente, su valor debe corresponderse con el de uno de los cuatro disponibles:

- **turtlebot3_worlds**: este valor ejecutará el mundo por defecto de Turtlebot3.
- **mymazeOut**: este término hace referencia a la parte exterior del laberinto *mymaze*, es decir, el más sencillo de los tres recorridos propuestos.
- **mymazeIn**: al seleccionar esta opción, el sistema mostrará el laberinto serpenteante de camino único, es decir, la parte interna del laberinto *mymaze*.
- **openMaze**: hace referencia al laberinto openMaze, el segundo diseñado cuya complejidad es superior a las anteriores.
- **f_openMaze**: conjunto de archivos utilizados para descubrir la reacción ante obstáculos imprevistos. Aunque este mundo también se puede mapear, no tiene mucho sentido hacerlo debido al motivo por el que se generó.

Para agregar nuevos mundos y garantizar que el sistema continúa funcionando, sería necesario crear el archivo correspondiente para referenciar al modelo que se debe cargar en Gazebo y siguiendo la nomenclatura establecida: *[nombre del mundo]_[modelo Turtlebot].world*.

Manteniendo el nombre del mundo, se deben situar los archivos del mapa en el directorio *maps*. Ambos archivos necesarios, .yaml y .pgm deben tener el mismo nombre, que deberá seguir la estructura *map_[nombre del mundo].[yaml/pgm]*. El parámetro “nombre del mundo” será el valor que tome la variable de entorno WORLD_NAME

8

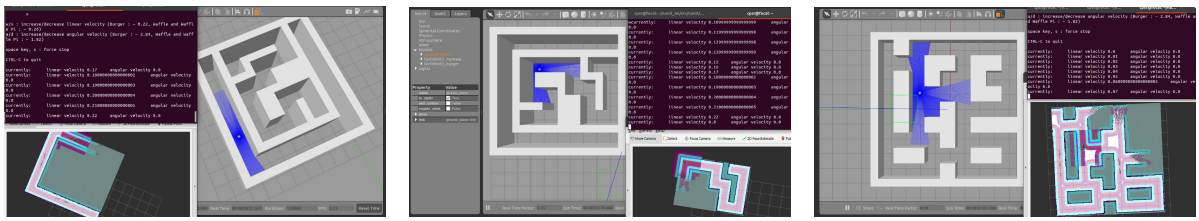
Resultados

En este capítulo se detalla la eficiencia del sistema, a nivel de velocidad de procesamiento y de resultados obtenidos. Se hace referencia a los fallos que provocan detenciones de sistema repasando, una a una, las fases del procesamiento del conjunto hasta la navegación del robot en su lugar designado.

A lo largo de este proyecto se ha conseguido obtener un gemelo digital para simular correctamente el comportamiento del Turtlebot3 Burger, modelo que, como se estudió en el Capítulo 2, tiene buenas prestaciones por un precio muy reducido. Además, destaca por la gran cantidad de documentación disponible a pesar de que, como se ha ido viendo a lo largo del documento, no siempre era correcto. En muchas ocasiones, esto ha causado más problemas que ventajas, pero se han conseguido solucionar en la mayoría de las ocasiones.

El primer paso fue obtener un modelo para simular el laberinto a partir de un diseño inicial. Para realizar este desarrollo se ha empleado el modo constructor de Gazebo. Trabajando con las medidas y las posiciones de las distintas paredes que los forman, se obtuvieron dos laberintos diferentes en formato .sdf, óptimo para simular empleando Gazebo o incluso Ign Gazebo llegado el momento. Se diseñó el archivo de manera propicia para conseguir que el robot se inicializara correctamente

Probando distintas herramientas de SLAM, se obtuvo un sistema capaz de elaborar un mapa correctamente. Para ello se utilizó el paquete Nav2 con las herramientas de *slam_toolbox*. La ejecución de los nodos de manera conjunta permite conducir el robot alrededor del circuito con el teclado.

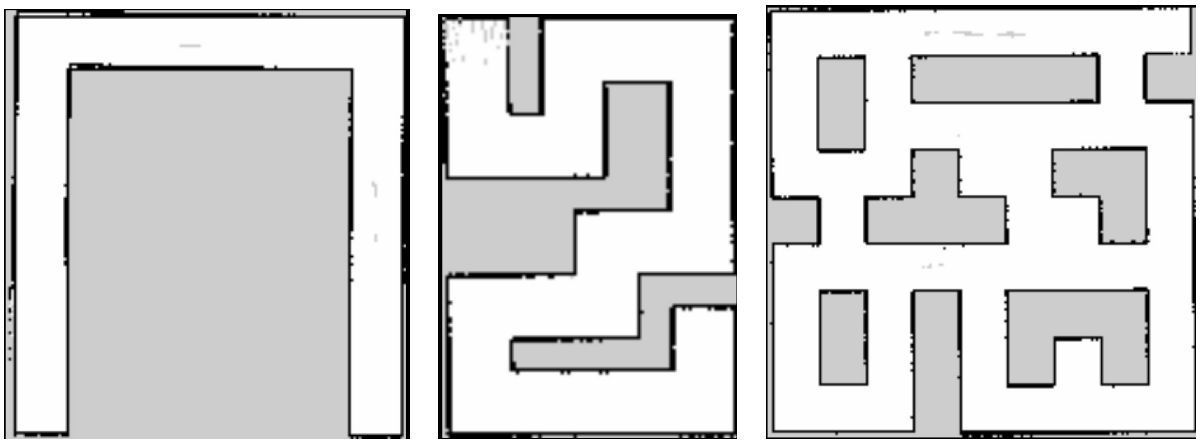


(a) **(b)** **(c)**

Figura 8.1. Mapa de los tres entornos desarrollados.

Mientras se realiza esta navegación manual, el robot publica en el topic */LaserScan* la información recogida por su sensor. El paquete *slam_toolbox*, que se encuentra suscrito, recoge esa información y va desarrollando el mapa a partir de ella. Este proceso queda representado en la Figura 8.1, donde se muestran las interfaces empleadas para construir los mapas de los tres laberintos empleados.

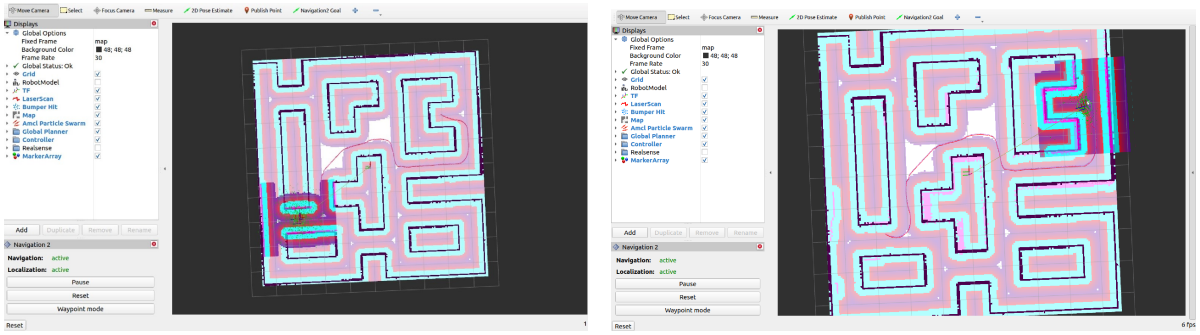
Tras finalizar este proceso y verificar la correcta creación del mapa, cuyos resultados se observan en la Figura 8.2, estos se almacenan en una ubicación conocida. De esta forma, se podrá acceder a ellos más adelante. A la hora de realizar la navegación, el robot cargará este mapa y se orientará sobre él para llevar a cabo su desplazamiento.



(a) **(b)** **(c)**

Figura 8.2. Mapa de los tres entornos desarrollados.

De nuevo, la ejecución del módulo de navegación se realiza a través de un único archivo preparado para levantar todos los nodos necesarios. El usuario observará dos interfaces distintas: una con la simulación del robot, y la otra con el mapa cargado. Tras indicar el punto inicial, y el punto de destino, el robot es capaz de ir de un lugar a otro, siempre que se disponga de suficientes recursos en la máquina donde se esté ejecutando. La Figura 8.3 muestra el principio y el final de una navegación tras haber recorrido la ruta señalada.



(a) **(b)**

Figura 8.3. Interfaz de RViz al comienzo de la navegación y tras haberla llevado a acabo.

Además, se ha generado un nuevo modelo de mundo para poder ejecutar una simulación con obstáculos, es decir, elementos que no están contemplados en el mapa que el robot tiene cargado.

Con esta simulación se ha podido comprobar que realmente el sistema está considerando su entorno constantemente, ya que fue capaz de esquivar el elemento y llegar a su punto de destino.

Para facilitar y controlar la ejecución, se ha generado un paquete propio en el que almacenar todos los archivos relativos involucrados. De esta forma se evita manipular el paquete original Navigation2 pudiendo utilizar libremente todos los *scripts* que ofrece. Esta nueva forma de operación permite tener toda la información accesible en un directorio con los documentos organizados. Otra ventaja de esta configuración es que se eliminan las rutas absolutas de los archivos, lo que facilita traspasar los archivos entre distintas máquinas.

Con la intención de agilizar todavía más la ejecución, se han generado dos archivos en *shell script* para fijar las variables de entorno necesarias: el modelo de Turtlebot utilizado y la ruta donde están ubicados los modelos de simulación. La primera variable en este proyecto siempre ha tenido el valor “burger”, sin embargo esto deja abierta la posibilidad de ejecutar todas las simulaciones con el modelo “waffle”. Adicionalmente, se ha añadido una nueva variable de entorno que permite determinar desde ahí qué mundo ejecutar.

9

Conclusiones

El presente capítulo expone las conclusiones finales del proyecto extraídas a partir de los resultados obtenidos de las diferentes pruebas realizadas.

Durante la elaboración de este proyecto se ha desarrollado una simulación completa del robot Turtlebot3 Burger, capaz de obtener el mapa de su entorno con una única vuelta por él y navegar por él desde su posición, hasta el punto de destino indicado. En caso de surgir un obstáculo inesperado no representado en el mapa, es capaz de esquivarlo, recalcular la ruta, e identificar otro camino para llegar a la posición señalada.

Respecto a los objetivos del proyecto, señalados en la Sección 1.2, se han conseguido en su totalidad, ya que se ha obtenido un gemelo digital que reproduce las características del Turtlebot3 Burger en distintos entornos de simulación.

Tomando un punto de vista más global, como ya se indicó en la Sección 1.1, la intención principal de este trabajo es acercar las tecnologías del mundo profesional a las aulas. Por ello, es muy importante la búsqueda de simplicidad a la hora de trasladar los modelos, ya que contiene toda la información en un mismo directorio, clasificada y ordenada. Además, su ejecución es rápida y sencilla gracias a la elaboración de los distintos archivos de lanzamiento y los encargados de fijar las variables de entorno.

No obstante, para poder ejecutar este sistema correctamente, será necesario disponer de ordenadores Linux potentes a disposición del usuario, ya que, como se ha visto, el simulador Gazebo consume gran cantidad de recursos. Para poder replicar las condiciones del proyecto, deberán contar con Ubuntu 20.04 y ROS2 Foxy instalados.

10
Futuros desarrollos

El último capítulo recoge una serie de mejoras que podrían realizarse en el futuro partiendo del elaborado en el presente proyecto. Se relatan las posibles mejoras y se proponen diversas maneras de llevarlas a cabo.

Este proyecto representa el inicio de una serie de estudios que se pueden conseguir a partir de aquí. En este trabajo se ha conseguido elaborar un gemelo digital del Turtlebot3 Burger, con el que se puede simular en cualquier entorno su capacidad para generar un mapa del lugar. Además, se puede simular la navegación posterior en ese entorno incluyendo obstáculos inesperados.

No obstante, se han encontrado problemas con el simulador, ya que necesita gran cantidad de recursos para poder ejecutar correctamente la navegación. Además, Gazebo es un simulador cuyo soporte finaliza en 2025. Por estos dos motivos surge la necesidad de conseguir combinar los desarrollos de SLAM y navegación con otro simulador.

Surgen dos opciones principales: IGN Gazebo y CoppeliaSim. Se ha iniciado el desarrollo en CoppeliaSim, cargando el modelo en el escenario. Para conseguir una simulación precisa, sería necesario conseguir que el sensor publicara los valores recogidos en el topic `/LaserScan` para poder operar con el resto de herramientas. Para esto, es preciso emplear la interfaz de comunicación de ROS 2 con CoppeliaSim, que admite archivos en lenguaje Lua y Python.

En IGN Gazebo habría que considerar el punto de partida. El simulador admite los archivos .sdf, por lo que se podrían reutilizar los modelos de laberinto. Además, soporta .urdf, por lo que el modelo del robot también se puede utilizar. En caso de que hubiera algún problema con el .urdf, se debería considerar utilizar el modelo obtenido en la Subección B.3.2.

Sería preciso conseguir un mundo que combinara ambos modelos en el archivo de lanzamiento y garantizar que se está publicando toda la información necesaria para el SLAM y la navegación. Finalmente, se debería generar un archivo conjunto, capaz de ejecutar todo el sistema desde una única terminal.

El otro aspecto importante que quedaría pendiente sería probar los paquetes con el robot físico. En este caso es importante crear un entorno real sencillo y replicarlo virtualmente.

Comenzar simulando el robot en ese entorno y una vez se haya comprobado que funciona correctamente, ejecutarlo con el real.

A

Objetivos de Desarrollo Sostenible

En este primer anexo se describe el efecto que este proyecto puede llegar a tener en el desarrollo del mundo. Para ello, se utilizará la agenda de desarrollo sostenible, destacando aquellos objetivos con los que el trabajo se encuentra más alineado.

Los Objetivos de Desarrollo Sostenible (ODS) son el conjunto de objetivos globales determinados por los líderes mundiales en septiembre de 2015 y cuya finalidad es asegurar la prosperidad mundial erradicando la pobreza y garantizando la protección del planeta. Esta agenda de desarrollo sostenible contiene un total de 17 objetivos, cuyas metas específicas deberán alcanzarse antes de 2030.

En esta sección se tratarán únicamente aquellos elementos de la agenda 2030 que estén directamente ligados al desarrollo del proyecto. Entre ellos se encuentran:

- **Objetivo 4**: Educación de calidad.
- **Objetivo 8**: Trabajo decente y crecimiento económico.
- **Objetivo 9**: Industria, innovación e infraestructuras.

El análisis pormenorizado de la relación entre los objetivos y el proyecto se realizará de forma ordenada, yendo del objetivo que más fuertemente relacionado está con el trabajo hasta el menos alineado. Por ello se tratarán en orden descendente, primero el 9, a continuación el 8 y finalmente el 4.

El **objetivo 9**, que lleva por título: “Industria, innovación e infraestructuras” engloba grandes y dispares objetivos. Entre ellos, se encuentran el desarrollo y modernización de infraestructuras de calidad adecuadas para potenciar el desarrollo, promover la industrialización de forma sostenible aumentando así el empleo y el PIB (Producto Interior Bruto) de la nación en la que se encuentre. Se busca, por otra parte, favorecer el acceso a pequeñas y medianas industrias/empresas a cadenas de valor y mercados. También se pretende aumentar

la investigación científica de cara a mejorar la capacidad tecnológica de diversos sectores industriales.

Es precisamente en esta última meta en la que se encuadra el presente proyecto. Si bien el trabajo no se basa en investigación, busca acercar estas tecnologías a la Academia para formar a futuros profesionales de la materia, que serán los responsables de llevar a cabo innovaciones en esta disciplina. Por otra parte, al formar profesionales en estas tecnologías y realizar trabajos y desarrollos sobre ellas, se produce un acercamiento de estos componentes a la industria. Finalmente, con el avance del tiempo, a medida que aumente su utilización, el coste de estos elementos se irá reduciendo, llegando a ser accesibles también para PYMES.

Respecto al **objetivo 8**: "Trabajo decente y crecimiento económico", se pueden identificar dos partes diferenciadas, ambas presentes en su enunciado. Como se comentó en la Capítulo 1, la llegada de una nueva revolución industrial trae consigo un cambio en las condiciones socioeconómicas ya que suelen estar asociadas a crecimientos económicos. Este trabajo está asociado a este objetivo ya que se tratan estas nuevas tecnologías que favorecen el desarrollo. Por otra parte, y aunque a más largo plazo, estas etapas se asocian a grandes cambios en las condiciones laborales de los trabajadores pero esto no es algo que se pueda evaluar a día de hoy.

Por último, el **objetivo 4**, que hace referencia, a grandes rasgos, a la escolarización de los niños y niñas de todas partes del mundo, asegurando que el mayor porcentaje posible de gente esté alfabetizada y tenga nociones básicas de aritmética. Ciertamente, este objetivo hace hincapié en no dejar de lado a aquellos niños en situaciones más vulnerables, ya sea por algún tipo de discapacidad, por una situación de pobreza o por las condiciones de su lugar de residencia.

Revisando estas condiciones, puede llegar a parecer que este objetivo no tiene ningún punto en común con este proyecto. Sin embargo, analizando las metas establecidas en esta sección de la agenda, la tercera de ellas busca asegurar el acceso igualitario a una formación técnica superior de calidad, y se incluye específicamente la formación universitaria. Como se recogió en la Sección 1.1, la motivación de este proyecto es la de traer a las aulas el último estándar en el mundo de la robótica.

B

Preparación del URDF del Turtlebot3

En este capítulo se explican los diferentes intentos seguidos para conseguir elaborar un modelo URDF del Turtlebot3. Primero se detallan todos los intentos fallidos por conseguirlo y la última sección describe cómo se consiguió finalmente.

Los archivos de simulación empleados a lo largo del proyecto son documentos con extensión .sdf, el formato por defecto empleado en Gazebo. Sin embargo, existe otro tipo de archivos, con un modelado URDF (Unified Robotic Description Format) que es el XML empleado por defecto en ROS.

Por este motivo el primer objetivo del proyecto fue intentar conseguir un modelado URDF del robot Turtlebot3 burger. Para ello, se emplearon distintas aproximaciones, y solo se consiguió generarlo empleando otro modelo similar como base. A continuación, se comentan todas las alternativas empleadas y los motivos por los que no funcionaron.

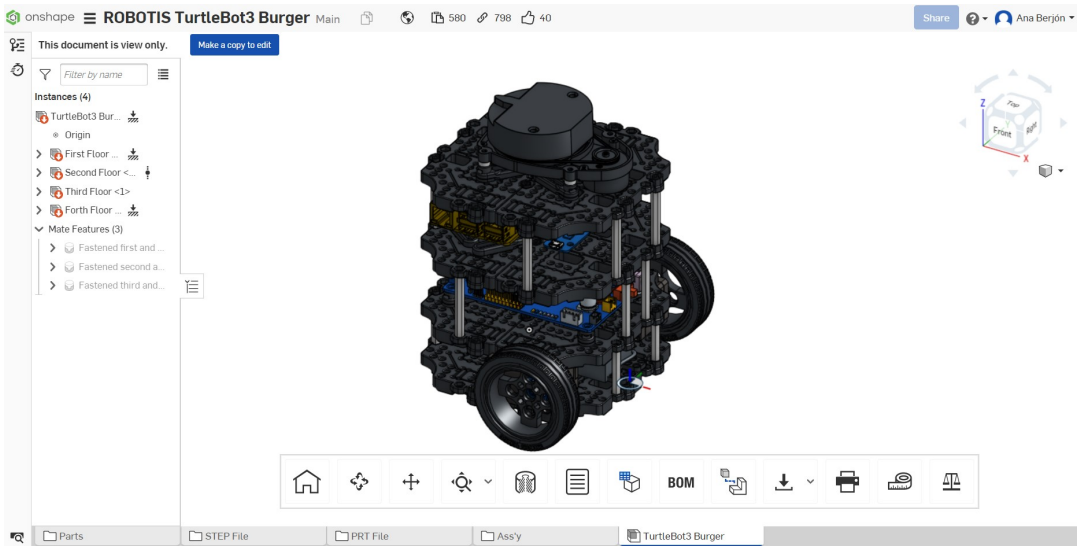


Figura B.1. Interfaz principal del modelo oficial de ROBOTIS para Turtlebot3 burger.

La mayoría de las opciones estudiadas tenían un mismo punto de partida, el modelo oficial del Turtlebot3 burger subido por ROBOTIS a la plataforma Onshape[1]. Al acceder a la plataforma se obtiene una vista como la mostrada en la Figura B.1. Puesto que se trata del modelo oficial, no se pueden realizar modificaciones directamente sobre él. No obstante, tras crearse una cuenta educativa, es posible generar una copia del robot y sobre ella se podrá modificar casi todo lo que el usuario decida.

B.1. Onshape-to-robot

La primera herramienta utilizada, y a la que más tiempo se le dedicó fue el módulo para Python3 Onshape-to-robot [**34**]. Este es un paquete de código libre con licencia MIT disponible en Github[2]. Está diseñada para pasar modelos Onshape a distintos tipos de formato, entre ellos URDF o SDF. Para poder utilizarla se necesita haber generado una clave y un secreto para la API, lo que se puede realizar desde el portal de desarrolladores de Onshape.

La clave, el secreto y la dirección de la API de Onshape se deben añadir como variables de entorno en la máquina desde la que se vaya a ejecutar la herramienta. También es preciso haber generado una carpeta, en la que se quiera guardar el robot en el nuevo formato. Inicialmente esta carpeta debe incluir un archivo llamado *config.json* que debe contener el identificador del documento (uno de los bloques de la URL) y el formato de salida.

Tras generar este archivo, solo es necesario ejecutar un comando para obtener el modelo. Se probó inicialmente con uno de los robots de muestra que contiene Onshape, y el procedimiento funcionó perfectamente. Sin embargo, a la hora de ejecutarlo con la copia del Turtlebot3, los resultados no fueron los esperados.

En una primera prueba, se obtuvo un *Warning* en la ejecución del programa, el mostrado en la Figura B.2. Este aviso indica que no hay materiales asignados al modelo. Por ello, a la hora de lanzar el archivo en el simulador, este se mantendrá en "el aire" y no descenderá puesto que no tiene peso.

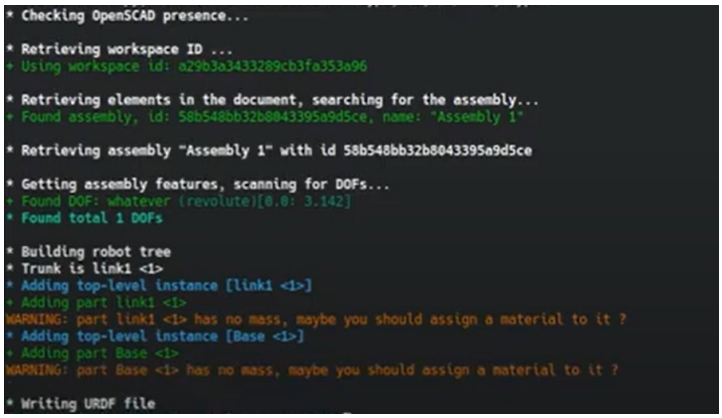


Figura B.2. Warning debido a la falta de asignación de material a las piezas.

Para solucionar este problema, es necesario ir a la interfaz de Onshape y asignar un material a cada pieza. Esta información no viene recogida en ningún documento de especificaciones, por

[1]Para acceder al modelo haga click aquí o siga este enlace https://cad.onshape.com/documents/2586c4659ef3e7078e91168b/w/14abf4cb615429a14a2732cc/e/9ae9841864e78c02c4966c5e.

[2]https://github.com/Rhoban/onshape-to-robot

lo que fue preciso escoger materiales que pudieran coincidir. En esta ocasión, los materiales elegidos fueron:

- ABS (Acrilonitrilo Butadieno Estireno): este material se aplicó a todas aquellas piezas representadas en negro en el modelo. También se denomina plástico de ingeniería ya que su elaboración es más compleja que la de otros plásticos y tiene infinidad de propiedades, entre las que se encuentra la dureza, rigidez o su estabilidad a altas temperaturas. Este es el único material que venía asignado, aunque solo a una de las piezas, el resto se tuvieron que fijar manualmente.
- Aluminio: aplicado a todas aquellas piezas de color metálico en la simulación sometidos a esfuerzos estructurales. Es el material más típico para este tipo de piezas por las funciones que suelen desempeñar. Además, es muy ligero y barato.
- Acero inoxidable A2: seleccionado para la bola esférica y las tuercas que sujetan los distintos tornillos. Se eligió para aquellos materiales representados con color metálico en la simulación, pero que deberían soportar mayores cargas que los anteriores.
- Caucho de silicona: material seleccionado para los neumáticos de las ruedas.

Tras asignar los materiales, se volvió a ejecutar el comando, en esta ocasión sin obtener ningún aviso. Al lanzar el simulador con el .urdf generado, el modelo llegó al suelo, pero lo que mostraba no era el robot, sino una rueda. Analizando el mensaje de la ejecución, quedó claro que el problema era que no estaba encontrando nada más, sencillamente cogía el ensamblaje de la rueda: llanta y neumático.

```
(base) cpsr@focal:~/Documents/onshape$ onshape-to-robot tostada/
* Checking OpenSCAD presence...
Can't run openscad -v, disabling OpenSCAD support
TIP: consider installing openscad:
sudo add-apt-repository ppa:openscad/releases
sudo apt-get update
sudo apt-get install openscad
* Checking MeshLab presence...
No /usr/bin/meshlabserver, disabling STL simplification support
TIP: consider installing meshlab:
sudo apt-get install meshlab

* Retrieving workspace ID ...
+ Using workspace id: 5f72c2526e2de02a8fe0478f

* Retrieving elements in the document, searching for the assembly...
+ Found assembly, id: 2ff6d8ccb4c4b63715e13fe3, name: "ball_caster"
+ Found assembly, id: 718cbdc206d8acc74b550154, name: "First Floor"
+ Found assembly, id: a33aeecd2931f5c2fa5f7ab2, name: "Second Floor"
+ Found assembly, id: b1c45632268b14fd84d58867, name: "Third Floor"
+ Found assembly, id: 9b212fd40ad7b34b69d109c4, name: "OPENCR"
+ Found assembly, id: b7bd5cfa1e1a4f7906cb8e46, name: "Forth Floor"
+ Found assembly, id: 4c2ff932a31d8b911a76018d, name: "TurtleBot3 Burger"
+ Found assembly, id: 3d4f8d9bdf9f9370a25cd2ad, name: "USB2LDS"
+ Found assembly, id: 32d4ec16435424b4454b14b3, name: "connected plates"
+ Found assembly, id: ba5bb2346f9e6ef19fdaf169, name: "wheel"

* Retrieving assembly "wheel" with id ba5bb2346f9e6ef19fdaf169

* Getting assembly features, scanning for DOFs...
* Found total 0 DOFs

* Building robot tree
* Trunk is PR16_A01_TIRE_2 <1>
* Adding top-level instance [PR16_A01_TIRE_2 <1>]
+ Adding part PR16_A01_TIRE_2 <1>
+ Adding part P30_ISP_A001_3D <1>

* Writing URDF file
```

Figura B.3. Mensaje en la ejecución del comando al obtener únicamente una rueda en el archivo .urdf.

Esto resultaba extraño, ya que en el modelo se encontraban todas las piezas del robot ensambladas. El *output* de la ejecución del comando se muestra en la Figura B.3 y se puede observar cómo recupera todos los ensamblajes del documento web pero luego solo construye la

rueda. En este punto se decidió emplear una segunda herramienta: CoppeliaSim, con la que se confirmó que el modelo no estaba bien generado. Esto se narra en la Subección B.3.1.

Para acotar el problema, se decidió dejar de trabajar con el robot completo, y pasar a operar únicamente con el ensamblado del primer piso (Figura B.4). Para reducir aun más la posibilidad de errores, se eliminaron todos los elementos que no pertenecieran a la placa base, las ruedas, la bola esférica y los motores.

(a) **(b)**

Figura B.4. Ensamblado del primer piso del Turtlebot3 burger. (a) Primer piso completo y (b) Elementos utilizados para la prueba.

Esta reducción del objeto de trabajo no solucionó nada en una primera iteración, pero permitió trabajar de una forma más cómoda. Tras revisar la documentación, quedó claro que el problema estaba en la definición de los *mate features*, es decir, la definición del enlace entre dos piezas del modelo. Para obtener como URDF una rueda, la pieza base que tenía que estar seleccionando por defecto es el neumático, que solo tiene un enlace: la llanta. La

Esto solo podía deberse a que este enlace estaba hecho al revés, y se había definido como padre de la unión al neumático. Se modificaron todos los enlaces de las piezas empleadas del primer piso, de los cuales se podían modificar cinco y se colocaron de forma que el padre fuera siempre la pieza que correspondía. Tras volver a ejecutar el programa, el modelo se exportaba correctamente, por lo que se incluyeron las piezas restantes del primer piso (Figura B.5).

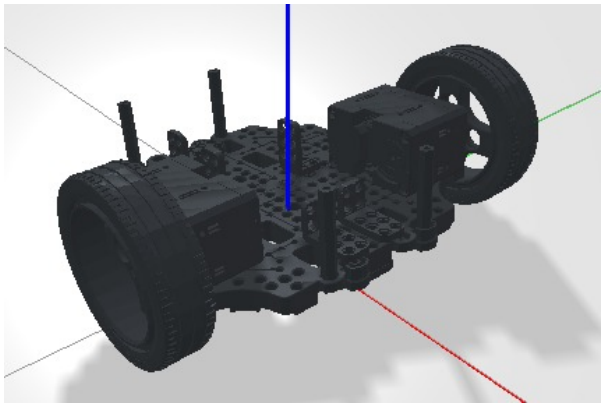


Figura B.5. Muestra del modelo URDF del primer piso ejecutado en el simulador propio de onshape-to-robot.

Sin embargo, este elemento es estático, es decir, las ruedas no giran y por lo tanto el robot nunca podrá moverse en una simulación. El siguiente paso sería conseguir que las laterales

girasen y que la rueda esférica pudiera virar en cualquier dirección para dar estabilidad al sistema. Tras analizar la documentación, se observó que para conseguirlo había que modificar los nombres de las *mate features* para que cumplan el siguiente formato: *dof_name* donde *name* suele ser una característica representativa del conector.

Tras realizar las modificaciones pertinentes y ejecutar de nuevo el comando, el .urdf no contenía lo que se esperaba, el resultado se observa en la Figura B.6. En esta ocasión, y sin ningún motivo aparente, el primer piso del robot solo mostraba una rueda. La otra era funcional, es decir, se podía mover utilizando el panel de la derecha de la Figura B.6.

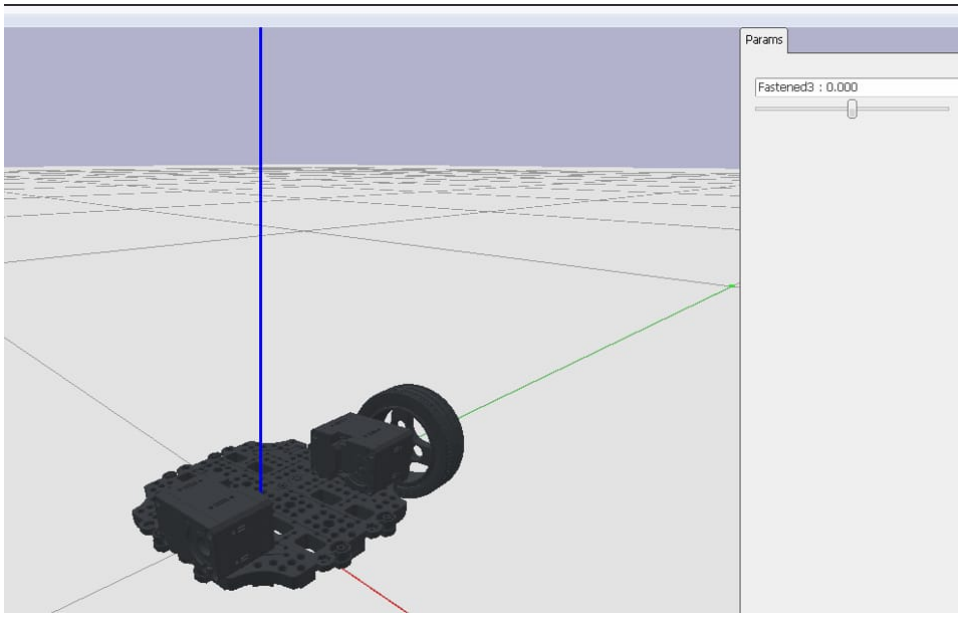


Figura B.6. Primer piso del Trutlebot3 burger con una rueda móvil y sin la otra.

Este problema no parece tener ninguna explicación razonable. El modelo desde el que se exportaba en Onshape estaba completo (Figura B.7), con los *mate features* correctamente nombrados y todas las partes ensambladas. Además, ni los logs de la herramienta ni el *output* de ejecución indicaban el motivo. Por ello se consideró que esto era una vía muerta.

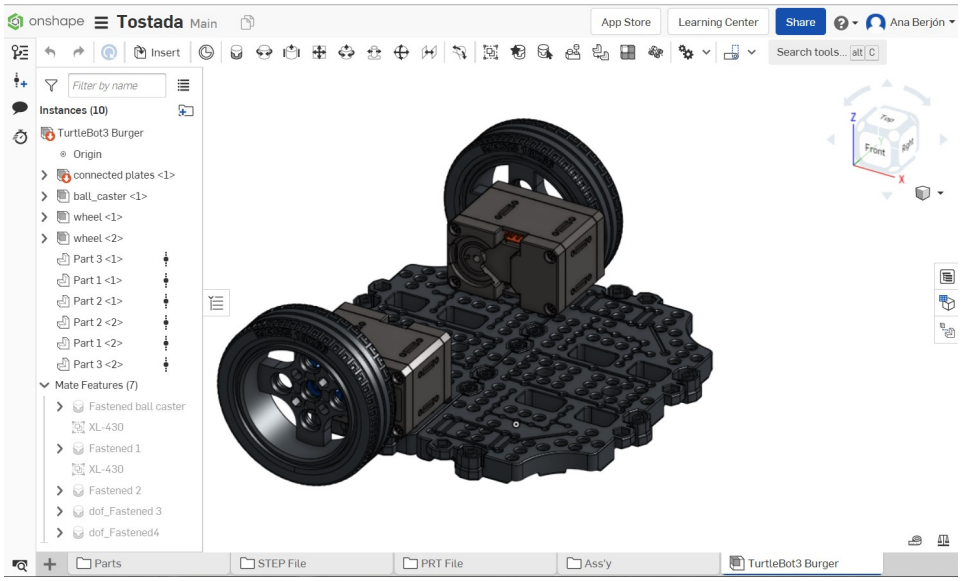


Figura B.7. Documento web desde el que se generaba el .urdf con una sola rueda.

B.2. SolidWorks

Solidworks es uno de los programas de diseño e ingeniería asistidas por ordenador más importantes del mercado. Se necesita licencia para poder utilizarlo, pero los ordenadores de la escuela lo tienen instalado. Este *software* utiliza principalmente archivos SLDASM, pero soporta y puede trabajar con muchos otros. Entre sus múltiples herramientas cuenta con un *plugin* que sirve para pasar archivos a formato .urdf. Para poder utilizarlo es necesario haberlo instalado previamente y esto, con permisos de alumno, solo es posible en los ordenadores de los laboratorios de electrónica.

Para poder seguir este enfoque, el primer paso es exportar el modelo de Onshape a un formato que pudiera interpretar el Solidworks. El primer formato con el que se intentó fue Parasolid, pero el programa no era capaz de interpretarlo. Se inspeccionarion los formatos que admitía el simulador y se decidió que el más adecuado era STEP. Por ello se exportó el archivo en ese formato y se importó. Tras ello, se guardó el modelo como step.SLDASM.

Una vez obtenido el fichero en formato .step.SLDASM se pudo abrir con el programa señalado. Sin embargo, el modelo simplificado, mostrado en la Figura B.8 tiene un problema fundamental que afecta al funcionamiento del robot: aunque las ruedas se importan al modelo, solo aparecen esos discos. Puesto que se trata de un modelo simplificado, podrían hacer las veces de ruedas y hacer que el robot rodara sobre ellas. No obstante, no llegan al suelo, así que aunque se consiguiera establecer correctamente el *link*, y giraran, el robot no se movería.

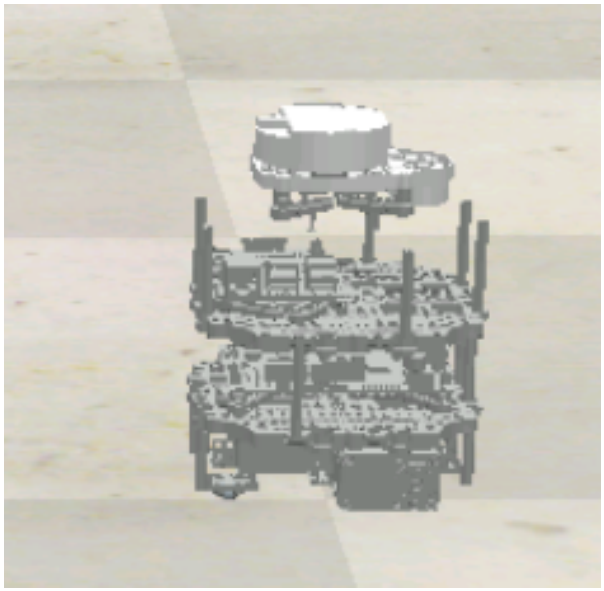


Figura B.8. Modelo simplificado importado en SolidWorks.

Para intentar solucionar este problema se llevaron a cabo diversas acciones. Lo primero que se intentó fue modificar el archivo .mesh, que contienen datos del modelo 3D. En este documento se trató de cambiar la escala a la que aparecía el modelo de las ruedas en relación con el resto del robot. Sin embargo, aunque parece que las ruedas se importan, no llegan a aparecer, por lo que esta modificación no surtió ningún efecto.

A continuación, se intentó modificar el archivo .stl, otro fichero con datos sobre el modelo. Para modificarlo, se debían emplear herramientas de modelado 3D, presentes en los ordenadores de la universidad, pero en esta ocasión no había un parámetro claro que modificar.

Finalmente, se trató de manipular el archivo directamente desde SolidWorks, buscando modificar las propiedades de las piezas y uniéndolos y desuniéndolos en distintas combinaciones de ensamblajes. Todas estas modificaciones tuvieron igual resultado que las anteriores.

B.3. Otras herramientas

La introducción del modelo en SolidWorks fue el último intento realizado para conseguir obtener el modelo URDF. En esta sección se recogen otros intentos que se realizaron antes en el tiempo, pero que no merecen una sección propia ya que no se dedicaron tantos esfuerzos para conseguir que funcionara.

B.3.1. CoppeliaSim

CoppeliaSim es el sucesor de V-Rep, uno de los principales simuladores de robótica del mercado. Este programa está integrado en la máquina virtual sobre la que se desarrolla el trabajo de este proyecto.

El *software* tiene un sistema para importar archivos .urdf, así que se decidió abrir el archivo generado por *onshape-to-robot* para intentar determinar cuál podría ser el problema. En el momento de importar la rueda, aparecían multitud de errores, entre ellos, indicaba que no existía el archivo .stl. No obstante, el archivo existía y estaba ubicado en el mismo directorio que el .urdf, por lo que realmente no debería haber habido ningún problema.

A raíz de estos contratiempos, se analizó con detenimiento el modelo de Onshape. En algunas partes del robot original había fallos del tipo "*fault topology*". Como era de esperar, en la copia también estaban estos errores puesto que estaban heredados del modelo oficial de Robotis. Con el fin de determinar el origen del problema, se descargaron los archivos en formato .step e importarlo de nuevo en el Onshape ya que debería señalar los problemas al intentar cargarlo. Sorprendentemente, no apareció señalado ningún error y los problemas de *fault topology* desaparecieron.

El nuevo modelo importado también presentaba algunos problemas, aunque distintos de los mostrados en el modelo original. En este caso, los ensamblajes tenían errores básicos de formación y era imposible manipularlos correctamente. Por este motivo, a pesar de los errores originales del modelo oficial, se optó por seguir trabajando con él.

B.3.2. Xacro

El Xacro es un lenguaje de macros XML típico de ROS1. Permite construir archivos XML más cortos y legibles utilizando una serie de macros que, en el momento de ejecución, convertirán el código en expresiones de mayor tamaño. En ROS existe una herramienta que permite convertir este tipo de ficheros a .urdf automáticamente con la simple ejecución de un comando.

El repositorio oficial de Turtlebot3 tiene este tipo de archivos subidos[3] por lo que resultaría muy sencillo descargarlos y transformarlos utilizando esta herramienta.

Para poder realizar esto, se instanció una nueva máquina virtual con Ubuntu focal 20.04 y se instaló ROS siguiendo las instrucciones oficiales. De entre todas las distribuciones disponibles,

[3] https://github.com/ROBOTIS-GIT/turtlebot3/blob/master/turtlebot3_description/urdf/turtlebot3_burger.urdf.xacro

se optó por instalar ROS *Noetic Ninjemys*, cuya fecha de lanzamiento es mayo de 2020. Esta es la distribución recomendada en ROS, y su EOL está fijado para mayo de 2025.

Tras preparar todos los paquetes y descargar los archivos necesarios, se ejecutó el comando para transformar el archivo .xacro en .urdf: *rosrun xacro xacro [directorio y nombre del archivo .urdf.xacro] >[carpeta destino]*. Se generó un archivo tal y como se esperaba que ocurriera. El archivo tenía un formato de etiquetas propio del .urdf y parecía haber obtenido el resultado correcto.

Para confirmar que el archivo se había generado correctamente, se intentó abrir usando Pybullet. Sin embargo, el programa no llegaba a arrancar, devolviendo un error que parecía indicar que no era capaz de localizar los meshes. Para solucionar este problema, se copiaron los archivos al directorio y se comprobó que la ruta del .urdf apuntaba a ellos.

Tras solucionar este problema, se volvió a ejecutar el Pybullet, pero en este caso tampoco fue capaz de abrir el archivo. Además, el mensaje de error no aporta ningún tipo de información: *pybullet.error: Cannot load URDF file*.

Para descartar que fuera algún error del programa, se intentaron buscar alternativas para abrir el archivo, utilizando otros programas. Solidworks no permite importar un .urdf y Coppeliasim daba error al abrirse, incluso sin importar el archivo. Por ello, la opción elegida fue el *plugin* de Visual Studio Code (VS Code) para visualizar en un lado de la pantalla el resultado del .urdf y en el otro el contenido del propio archivo. Al ejecutarlo, la simulación aparecía como un entorno vacío, sin ningún elemento y tampoco daba ninguna pista sobre el motivo.

En este punto, parece claro que el error está en el archivo, y dada la situación, aparenta tener algún problema de formato. Tras el análisis del documento, comparándolo con otros del mismo tipo, se descubrió que en el .urdf generado, se utilizaba un gran número de etiquetas <gazebo>que en otros archivos no aparecían. Dado el gran número de etiquetas presentes, y la imposibilidad de conocer a ciencia cierta cuáles se debían eliminar y cómo, se descartó esta vía de trabajo.

B.4. Adaptación de otro URDF

Durante el estudio del fichero *launch*, descrito en el Capítulo 6, se descubrió que el módulo Navigation2 invocaba un archivo URDF. Este archivo es el que se utiliza para invocar el nodo de publicación del estado del robot: `__state__publisher`. El último argumento que toma este nodo es siempre un URDF.

Desgraciadamente, este archivo no representaba al modelo Burger, sino al Waffle, pero los datos publicados son iguales para ambos, por lo que era irrelevante cuál utilizar. Al analizar el archivo, se descubrió que su esquema era muy parecido al del formato .urdf.xacro, y que ambos xacros eran muy similares en el caso del Burger y del Waffle.

Por ello, tomando como referencia los .xacro y realizando una copia del .urdf, se procedió a modificar todos aquellos datos en los que hubiera alguna diferencia. Los puntos que hubo que modificar fueron en la inercia y las coordenadas de los distintos elementos. También fue necesario eliminar algunos componentes, por ejemplo, la segunda rueda loca del Waffle o la cámara.

Tras guardar el archivo, se intentó importar en el simualdor Coppeliasim, que posee un módulo para importar este tipo de archivos. La importación generó numerosos errores, todos

ellos relacionados con los .stl, aquellos archivos que realmente tienen la información del diseño de las distintas partes del robot.

Tras copiar la carpeta meshes, que es en la que se almacenan estos documentos al mismo directorio que el .urdf y asegurar que las rutas a ellos estaban bien, se volvió a intentar importar. En esta ocasión, el modelo se importó correctamente, como muestra la figura

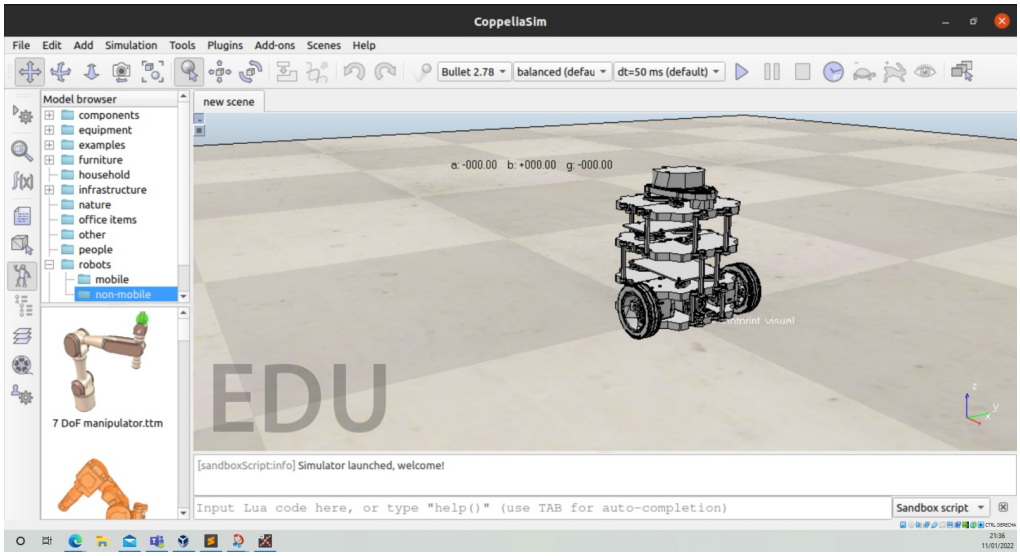


Figura B.9. Representación del modelo URDF en CoppeliaSim.

Al tratar de introducir este modelo directamente en un escenario ya construido, surgen varios problemas. El que más llama la atención a primera vista es la ubicación del robot: debajo de una de las piezas que conforman el laberinto[4] ya que por defecto se carga el modelo .urdf en el centro del eje de coordenadas. No obstante, a la hora de desplazar el robot hacia otras coordenadas aparece el segundo problema, mucho más importante.

Las partes del robot no se comportan como un conjunto, sino como elementos independientes. Esto puede acarrear gran cantidad de problemas a la hora de desplazar las piezas y, por su puesto, en sus desplazamientos. Por ello es preciso solucionar este problema antes de desplazar el robot. En la Figura B.10 se muestra lo que ocurre al desplazar el robot sin fusionarlo. Se observa el robot desde la parte superior, y se puede comprobar cómo las ruedas han quedado completamente disociadas del resto del cuerpo.

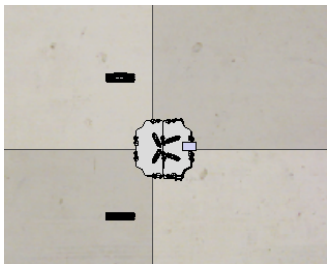
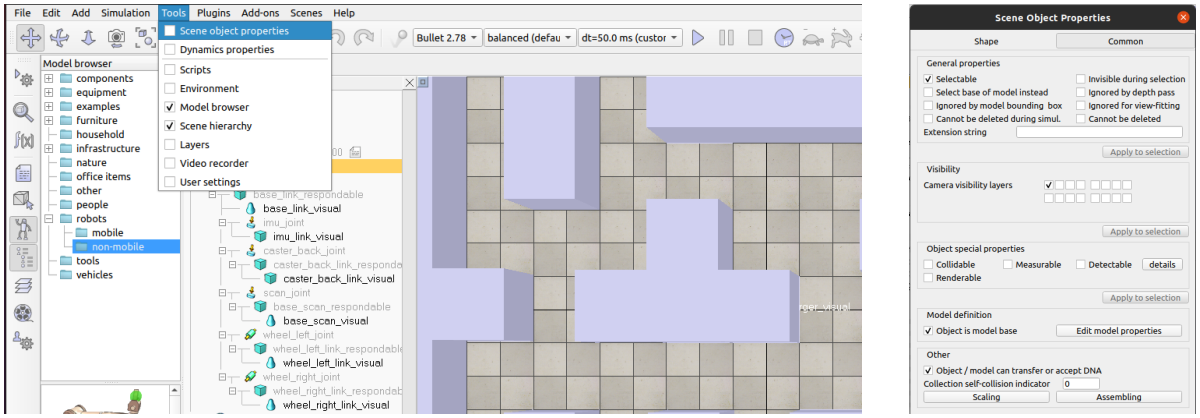


Figura B.10. Muestra de lo que ocurre al desplazar el robot antes de convertirlo en un único objeto.

[4]Este escenario se aprecia en la Figura B.11 [a], donde solo se pueden ver unas letras blancas sobresaliendo de esta pieza central. Estas letras son identificadoras del modelo, ya que etiquetan las partes del mismo.

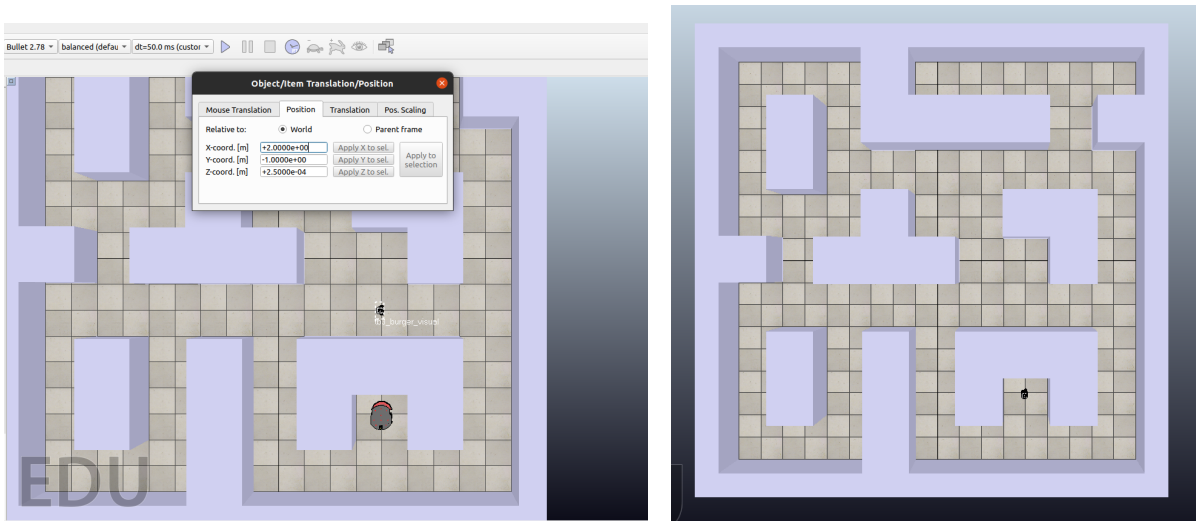
Para solventar esta situación, fue preciso seleccionar todos los elementos del robot y acceder desde la barra superior a las características del objeto (*Tools >Scene object properties*, Figura B.11 [a]). Se abrirá la ventana mostrada en la Figura B.11 [b] y en ella se deben marcar las cajas de los dos últimos campos, tal y como aparece en la figura.



(a) **(b)**

Figura B.11. Pasos para agrupar el robot en una sola entidad.

Desde este momento, se puede trabajar con el robot sin riesgo de perder las piezas por el camino. El primer paso es trasladarlo al punto de inicio, que en esta ocasión será dentro de la "U" de la parte inferior, es decir, el punto en el que se encuentra el robot de la simulación original en la Figura B.12. Para realizar este movimiento, es necesario pulsar el botón de la barra de herramientas y fijar las coordenadas en las que se querrá ubicar, en este caso, el punto (3,-2).



(a) **(b)**

Figura B.12. Movimiento del Turtlebot3 a la posición de salida.

Por último es necesario solucionar el problema del sensor. Al incorporar el modelo como .urdf ha perdido la funcionalidad, y es necesario especificarla. Se debería incorporar la funcionalidad sobre el láser, incorporando un archivo programado en .lua que especifique su comportamiento. Realizar este *script* desde cero resultaría muy complicado para alguien que no está familiarizado

con el lenguaje. Por ello, en una primera aproximación, se ha decidido incorporar uno de los modelos de ejemplo que incluye el simulador.

Entre todos los sensores que ofrece, existen tres con características similares al LDS01, todos ellos distintas versiones del modelo *Hokuyo URG-04LX-UG01*. Siguiendo los razonamientos de [**35**], se optó por la versión *Fast* ya que está diseñado específicamente para ROS. La principal diferencia entre este sensor y el LDS01 es el ángulo de escaneado, que mientras que el *Hokuyo* es de 270º, el original del Turtlebot3 escanea en todas direcciones.

Para incorporar el sensor al modelo, es preciso seleccionar la pieza sobre la que se va a acoplar y presionar el botón *assemble/dissasemble*, en la barra de herramientas del simulador. El ensamblaje se realiza de manera automática sobre la superficie, sin poder escoger orientación. Por ello es muy importante elegir correctamente dónde realizarlo. En este caso, se decidió que la pieza que se debía utilizar era *base_scan_respondable* para conseguir que el haz del sensor quede paralelo al suelo. La Figura B.13 muestra cómo queda finalmente el robot tras incorporar este elemento.

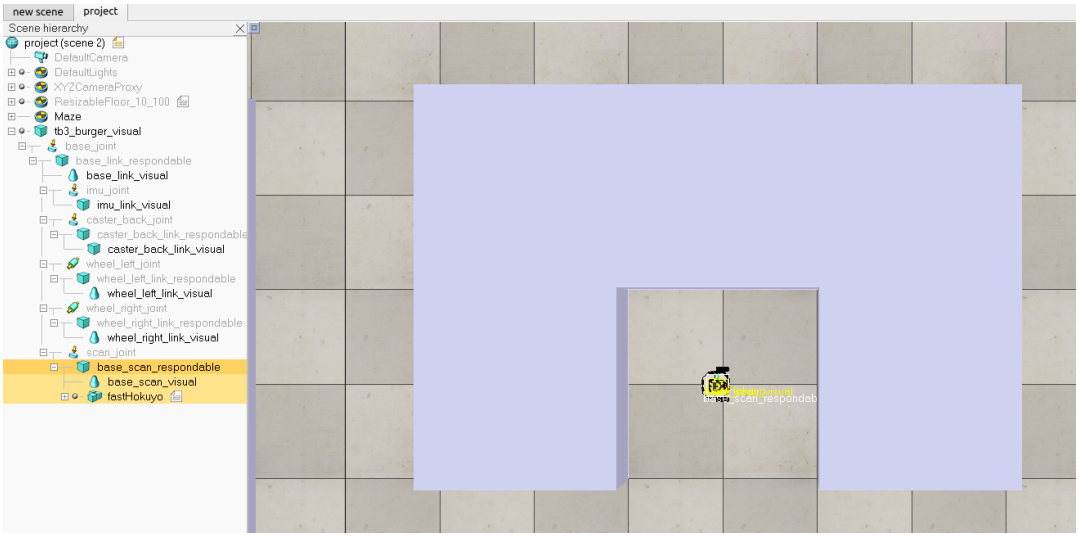


Figura B.13. Colocación del sensor sobre el Turtlebot3.

Para comprobar el funcionamiento del sensor, se debe comenzar la simulación, y, como se puede observar en la Figura B.14, el sensor está recibiendo información. Queda patente aquí la diferencia de rangos de ambos sensores, ya que en el caso de la simulación con Gazebo, el sensor cubre 360° , mientras que en este caso el rango es muy inferior.

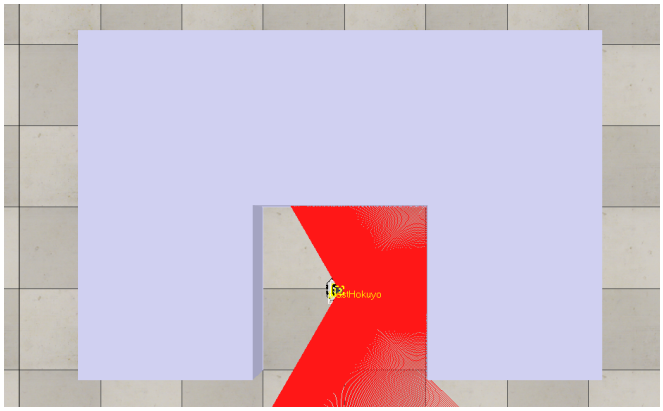


Figura B.14. Muestra del sensor activado.

Para conseguir incorporar este modelo a todos los elementos desarrollados a lo largo de este proyecto, será necesario conseguir que el modelo publique los datos obtenidos por el sensor en el *topic* `/scan` con el formato requerido por ROS 2. Esto se puede conseguir configurando el archivo de ejecución del sensor para que realice esa publicación.

C

Diagramas de nodos

En esta sección se incluyen los diagramas de nodos de ejecución que no se pudieron añadir en los capítulos anteriores debido a su enorme tamaño. La primera primera imagen, Figura C.2 incluye todos los nodos involucrados en la generación del mapa. Por su parte, la segunda recoge los nodos activos durante el proceso de navegación.

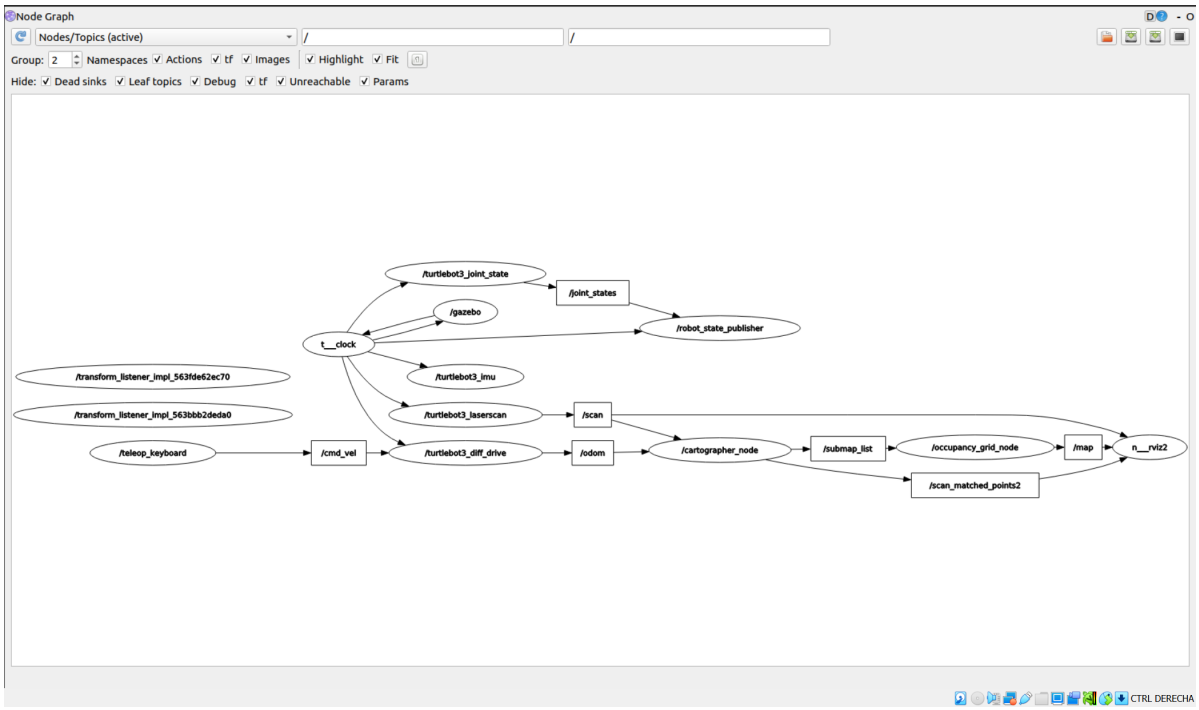


Figura C.1. Nodos involucrados en el proceso de creación del mapa utilizando Cartographer.

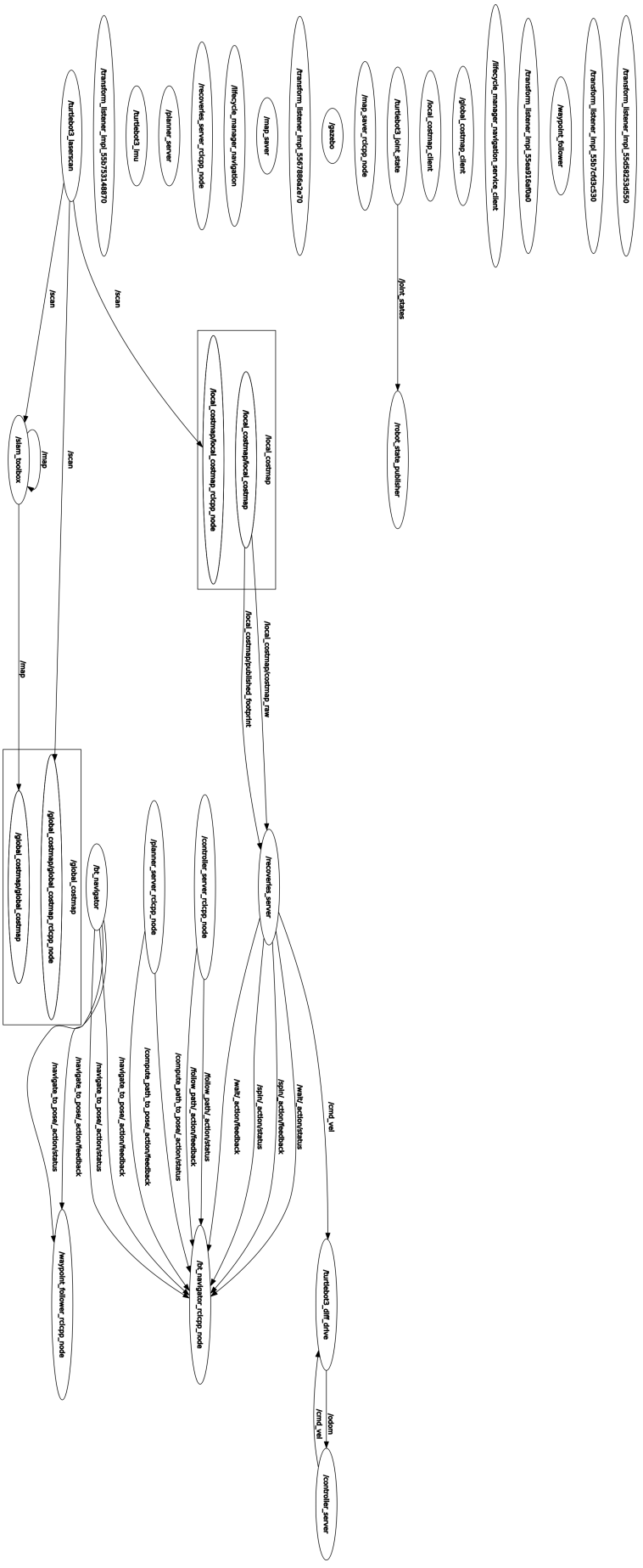


Figura C.2. Nodos involucrados en el proceso de creación del mapa utilizando Navigation2 y *slam_toolbox*.

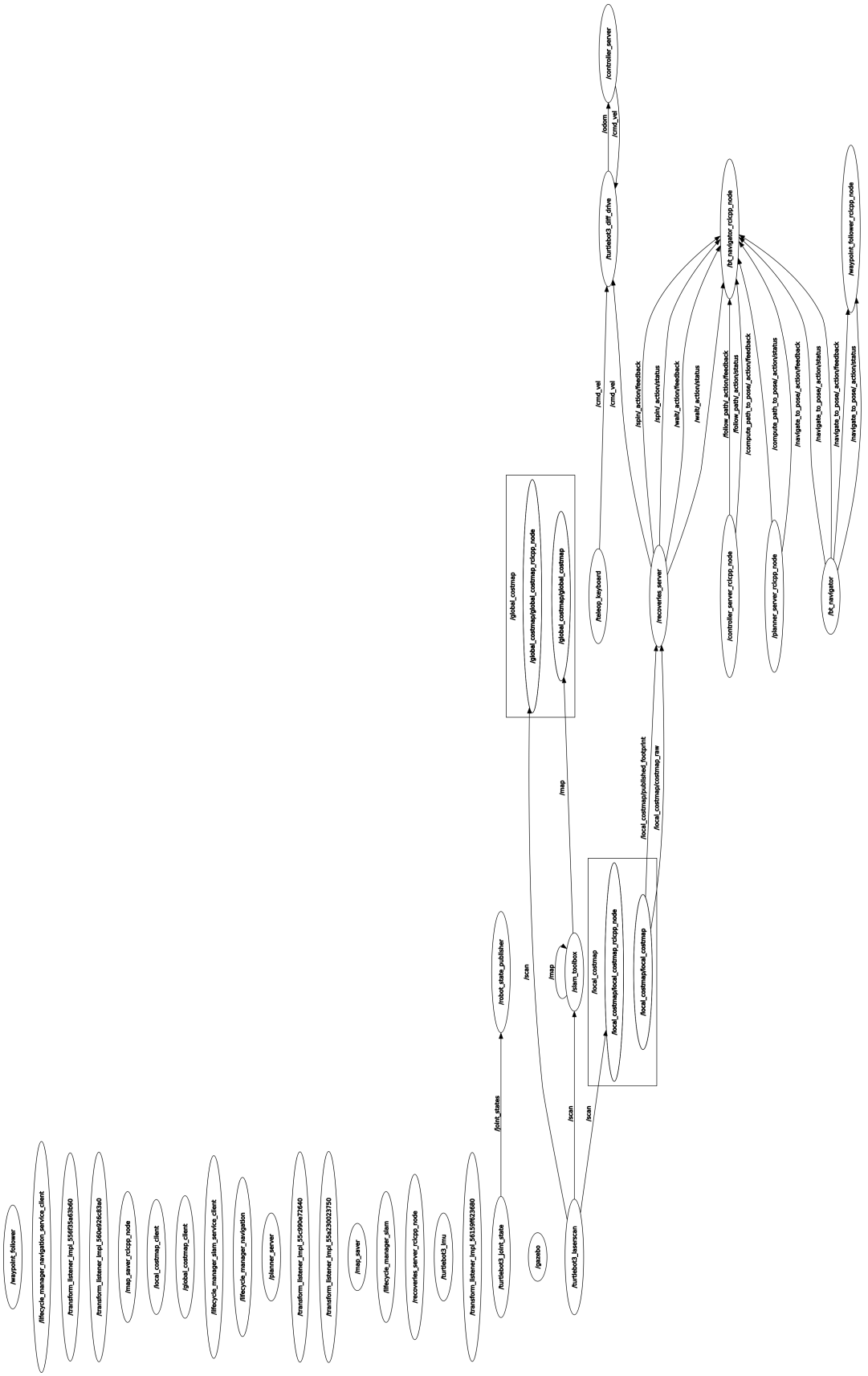


Figura C.3. Nodos involucrados en el proceso de navegación.

Bibliografía

[1] M. A. Arroyo Lazo, «The fourth Industrial Revolution,» Economía, vol. 41, nº 81, pp. 194-197, 2018.

[2] L. Probst, B. Pedersen, O.-K. Lonkeu, C. Martinez-Diaz, L. Novelle Araujo, D. Klitou, J. Conrads y M. Rasmussen, «Evidence of positive outcomes and current opportunities for EU businesses,» de Digital Transformation Scoreboard, European Commission, 2017.

[3] P. Pathak, P. Ora, P. R. Pal y M. Shrivastava, «Fifth revolution: Applied AI & human intelligence with cyber physical systems,» International Journal of Engineering and Advanced Technology, vol. 8, pp. 23-27, 2019.

[4] ROS-Industrial, «Description,» ROS-Industrial, 2012. [En línea]. Available: https://rosindustrial.org/about/description. [Último acceso: 29-10-2021].

[5] Open Robotics, «ROS 2 Documentation,» 2021. [En línea]. Available: https://docs.ros.org/en/foxy/index.html. [Último acceso: 01-11-2021].

[6] Open Source Robotics Foundation, «Changes between ROS 1 and ROS 2,» 06-2017. [En línea]. Available: http://design.ros2.org/articles/changes.html. [Último acceso: 29-10-2021].

[7] Open Source Robotics Foundation, «Gazebo,» 30-01-2020. [En línea]. Available: http://gazebosim.org/. [Último acceso: 13-11-2021].

[8] louise@openrobotics.org, «ign-gazebo,» Ignition Robotics, 8-11-2021. [En línea]. Available: https://github.com/ignitionrobotics/ign-gazebo. [Último acceso: 14-11-2021].

[9] Coppelia Robotics, «Coppelia Robotics,» 06-04-2021. [En línea]. Available: https://www.coppeliarobotics.com/features. [Último acceso: 14-11-2021].

[10] EDS Robotics, «AGV vs AMR ¿Qué les diferencia? ¿Cuál es mejor?,» 14-01-2021. [En línea]. Available: https://www.edsrobotics.com/blog/agv-vs-amr-diferencias/. [Último acceso: 15-11-2021].

[11] H. Durrant-Whyte y T. Bailey, «Simultaneous localisation and mappint (SLAM): Part I,» IEEE Robotics and Automation MAgazine, vol. 13, nº 2, pp. 99-110, 2006.

[12] T. Bailey y H. Durrant-Whyte, «Simultaneous localisation and mapping (SLAM): Part II,» IEEE Robotics and Automation Magazine, vol. 13, nº 3, pp. 108-117, 2006.

[13] A. Javaid, «Understanding Dijkstra Algorithm,» SSRN Electronic Journal, 2013.

[14] X. Liu y D. Gong, «A comparative study of A-star algorithms for search and rescue in perfect maze,» 2011.

[15] A. Ma'arif, O. Wahyunggoro y A. Imam, «Artificial Potential Field Algorithm Implementation for Quadrotor Path Planning,» International Journal of Advanced Computer Science and Applications, vol. 10, 2019.

[16] W. R. Faiza Gul, N. A. Syed Sahal y C. Kun, «A comprehensive study for robot navigation techniques,» Cogent Engineering, vol. 6, nº 1, 2019.

[17] Clearpath Robotics, «Dingo,» Clearpath Robotics, [En línea]. Available: https://clearpathrobotics.com/dingo-indoor-mobile-robot/. [Último acceso: 28-10-2021].

[18] Generation Robots, «Dingo Indoor Robotic Platform,» 2019. [En línea]. Available: https://www.generationrobots.com/en/403731-dingo-indoor-robotic-platform.html#/217-drive_type-_d_differential. [Último acceso: 28-10-2021].

[19] Rover Robotics, «4WD Rover Pro,» Rover Robotics, 2020. [En línea]. Available: https://roverrobotics.com/products/4wd-rover-pro. [Último acceso: 30-10-2021].

[20] K-Team, «Khepera IV,» K-Team, [En línea]. Available: https://www.k-team.com/khepera-iv. [Último acceso: 1-11-2021].

[21] Generation Robots, «Khepera IV mobile robot,» 2015. [En línea]. Available: https://www.generationrobots.com/en/402241-khepera-iv-mobile-robot.html. [Último acceso: 01-11-2021].

[22] Robotis, «Overview,» Robotis, 2021. [En línea]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/#overview. [Último acceso: 2-11-2021].

[23] Robotis, «Turtlebot3 Specifications,» Robotis, 2021. [En línea]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications. [Último acceso: 2-11-2021].

[24] ROS Componentes, «TurtleBot3 Burger,» ROS Componentes, [En línea]. Available: https://www.roscomponents.com/en/mobile-robots/214-turtlebot3-burger.html#/courses-no/turtlebot_3_burger_model-burger_intl_. [Último acceso: 9-11-2021].

[25] Osoyoo, «Main webpage,» Osoyoo, 2017. [En línea]. Available: https://osoyoo.com/. [Último acceso: 3-11-2021].

[26] Osoyoo, «Osoyoo Omni-directional Mecanum Wheels Robot Car Kit for Arduino,» Amazon, [En línea]. Available: https://www.amazon.es/OSOYOO-Omni-directinal-Mecanum-Controlled-Educational/dp/B082D18QVD. [Último acceso: 3-11-2021].

[27] A. Dattalo, «ROS/Introducción,» OpenRobotics, 08-08-2018. [En línea]. Available: https://wiki.ros.org/ROS/Introduction. [Último acceso: 29-01-2022].

[28] W. Woodall, «ROS on DDS,» ROS 2 Design, 07-2019. [En línea]. Available: https://design.ros2.org/articles/ros_on_dds.html. [Último acceso: 19-01-2022].

[29] W. Woodall, «Topic and Service name mapping to DDS,» 06-2018. [En línea]. Available: http://design.ros2.org/articles/topic_and_service_names.html. [Último acceso: 29-01-2022].

[30] Open Source Robotics Foundation, «Building a world,» 2018. [En línea]. Available: http://gazebosim.org/tutorials?tut=build_world. [Último acceso: 28-11-2021].

[31] Robotis, «SLAM Simulation,» 20-01-2021. [En línea]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/slam_simulation/. [Último acceso: 30-11-2021].

[32] S. Macenski, «On Use of the SLAM Toolbox: A Fresh(er) look at Mapping and Localization for the Dynamic World,» 2019. [En línea]. Available: https://www.google.es/search?q=slam+toolbox+presentation&ie=UTF-8&oe=. [Último acceso: 15-01-2022].

[33] Navigation, «Nav2,» 2020. [En línea]. Available: https://navigation.ros.org/. [Último acceso: 28-12-2021].

[34] G. Passault y M. Bestmann, «Onshape-to-robot documentation,» 08-2021. [En línea]. Available: https://onshape-to-robot.readthedocs.io/en/latest/. [Último acceso: 05-09-2021].

[35] I. Ampuero, Implantación de algoritmos de SLAM en el entorno de simulación V-REP usando ROS 2, Madrid: Universidad Pontificia Comillas - ICAI, 2020.