GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

IMPLANTACIÓN DE ALGORITMOS DE SLAM EN EL ENTORNO DE SIMULACIÓN V-REP USANDO ROS 2

Autor: Ignacio Ampuero Gonzalez

Director: Jaime Boal Martín-Larrauri

Madrid

Agosto de 2020

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Implantación de algoritmos de SLAM en el entorno de simulación V-REP usando ROS2

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2019/20 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.: Ignacio Ampuero Gonzalez Fecha: 22/08/2020

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Firmado digitalmente por
BOAL MARTIN LARRAURI
JAIME - 05304600H
Fecha: 2020.08.23
11:08:03 +02'00'

Fdo.: Jaime Boal Martín-Larrauri Fecha: 23/08/2020

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

IMPLANTACIÓN DE ALGORITMOS DE SLAM EN EL ENTORNO DE SIMULACIÓN V-REP USANDO ROS 2

Autor: Ignacio Ampuero Gonzalez

Director: Jaime Boal Martín-Larrauri

Madrid

Agosto de 2020

Agradecimientos

Primero y principal, me gustaría agradecer a mi director de proyecto Jaime Boal Martín-Larrauri. La idea de este proyecto ha sido suya, y gracias a él he podido realizar un proyecto a tiempo debido a todos los imprevistos que han ocurrido debido al COVID-19. No solo ha aportado la idea del proyecto, si no que ha sido una ayuda fundamental para llevarlo a cabo. Jaime me ha ayudado a salir de cada bache que me he encontrado en el proyecto en cualquier momento sin importar el día, el momento o su propio tiempo. Estoy muy agradecido.

Por otro lado, agradecer a la Universidad Pontificia de Comillas y a mi coordinador de proyecto Francisco Javier Herraiz, ya que me ha proporcionado una solución y los recursos necesarios para poder realizar este proyecto.

Me gustaría agradecer también a mis amigos, los viejos y los hechos este año, que sin duda has sido un gran apoyo para seguir adelante con este proyecto y de los cuales he apoyo durante todo el proceso. Especialmente agradecer a mi compañero Jose Antonio Leiva, que ha sido mi mano derecha este año en todo momento, y sin él no estaría en este punto ahora mismo.

Por último, quiero dedicar este proyecto a mis padres, Esther e Ignacio, he llegado a este punto por vosotros y no podría estar más agradecido.

Sé que dejo a mucha gente fuera que no he mencionado, a ellos, muchas gracias, donde quiera que estéis.

IMPLANTACIÓN DE ALGORITMOS DE SLAM EN EL ENTORNO DE SIMULACIÓN V-REP USANDO ROS 2

Autor: Ampuero Gonzalez, Ignacio
Director: Boal Martín-Larrauri, Jaime
Entidad Colaboradora: ICAI

RESUMEN DEL PROYECTO

El objetivo de este proyecto ha sido conseguir que un robot móvil sea capaz de construir un mapa de un entorno desconocido a medida que lo explora. Para ello se han comparado distintos algoritmos de localización y mapeado simultáneos (SLAM, por sus siglas en inglés) disponibles en ROS 2, y se ha integrado el más adecuado usando en el proceso el entorno de simulación V-REP.

Palabras clave: SLAM, ROS, VREP

1. Introducción

La Industria 4.0 implica la promesa de una nueva revolución, cuyo cambio se basa en la adopción de las nuevas tecnologías como la robótica colaborativa. El *framework* ROS es la plataforma idónea para el desarrollo de aplicaciones para la robótica colaborativa. ROS nos da la posibilidad de implantar tecnologías como SLAM la cual está irrumpiendo en la robótica industrial y comercial, impulsando una nueva generación de robots móviles autónomos, capaces de desenvolverse solos en las fábricas y en otros entornos.

2. Robot Operating System (ROS)

Robot Operating System (ROS) es una colección de *frameworks* para el desarrollo de software de robots. ROS es el estándar más popular adoptado por la industria y existen diversas distribuciones de este. En este proyecto se ha trabajado con ROS2, que apareció en 2016, en concreto con la versión Eloquent Elusor (2019). La principal diferencia con ROS1 es, que en ROS2 desaparece la arquitectura maestra/esclavo, la cual imperaba en ROS1.

Los nodos son las piezas fundamentales de ROS. Un nodo ROS es cualquier pieza de software de este sistema. ROS está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos. Los nodos pueden pasar mensajes entre sí mediante canales lógicos llamados *topics* y se comunican entre ellos mediante un mecanismo del tipo *Publish*/*Subscribe* (Publicación/Suscripción).

ROS cuenta con varios complementos que han sido utilizadas en este proyecto. Estos han sido RViz, rqt_graph y tf (transformadas), los cuales han servido para visualizar los resultados y entender el funcionamiento de ROS. Todas estas herramientas están disponibles en ROS2.

3. SLAM

La robótica móvil es la robótica que mayoritariamente usa ROS y su objetivo es desarrollar robots autónomos. La capacidad de navegar en ambientes desconocidos, es decir, sin un mapa del entorno, forma una parte importante de lo que significa un robot autónomo. Cuando no se dispone de mapa para la navegación, se dice que se está haciendo una navegación tipo SLAM (Localización y mapeo simultaneo, es español). La

navegación SLAM plantea si es posible situar un robot móvil en un entorno desconocido y que sea capaz de construir un mapa consistente de su entorno mientras determina su posición en ese mapa simultáneamente. Podemos decir que este problema es como el problema del huevo y la gallina: “El robot necesita saber su posición para construir in mapa y necesita un mapa para determinar su posición”. Para solucionar este problema, el robot solo dispone de la información proporcionada por las medidas obtenidas de los sensores y la noción de movimiento propio.

La mayoría de los modelos usados para resolver el problema del mapeado, están basados en probabilidades y construidos sobre las reglas de Bayes. Estos son los filtros extendidos de Kalman (EKF), filtros de partículas (PF), algoritmos basados en grafos (*graph* SLAM). El algoritmo SLAM puede ser implementado de muchas maneras y para ello esencial disponer de un robot móvil y un dispositivo capaz de detectar el entorno (comúnmente un sensor láser o LIDAR)

4. Entorno de trabajo e implementación del algoritmo

El entorno inicial consistía en un escenario en el entorno de simulación V-REP (simulador de robot 3D), compuestos por un robot con cinemática diferencial (Pioneer P3DX) y un conjunto de paredes que forman caminos. Este robot está controlado por 4 scripts en Python de forma externa, los cuales inicializan el robot y lo desplazan por el escenario siguiendo una pared y evitando que colisione.

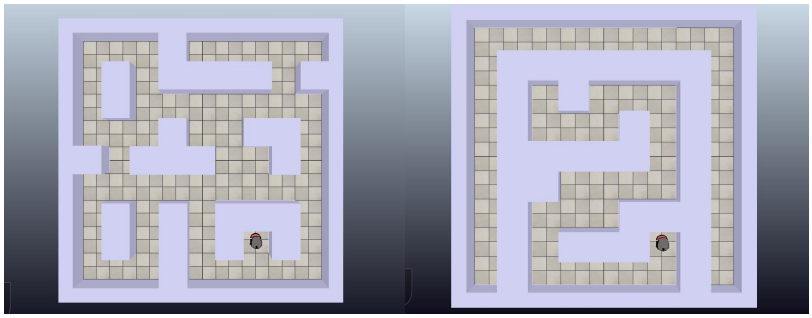


Figura 1.1 Escenarios de trabajo

Para la implementación del proyecto se han seguido una serie de pasos. El primer paso ha sido seleccionar el mejor algoritmo SLAM para ROS2. SLAM Toolbox es el paquete que se ha decidido implementar ya que es el estándar de SLAM para ROS2. Este paquete recibe mensajes de un sensor laser y va generando un mapa del entorno. El segundo paso es la búsqueda y selección de un sensor laser para acoplarlo al robot. El sensor *Hokuyo URG-04LXUG01_Fast* (disponible en V-REP) fue el elegido y se acoplo encima del robot. El siguiente paso consiste obtener los datos del sensor laser (con el robot en movimiento) y publicarlos en un mensaje de ROS. Tras esto, para tener noción del movimiento de robot, se ha implementado la odometría del robot, de manera que se supiese como se movía el robot lineal y angularmente con respecto a su punto inicial. Sin este paso, el mapa no se podía construir, ya que sin odometría, SLAM Toolbox considera que el robot esta estático.

5. Resultados

El proyecto se ha desarrollado e integrado con el entorno de simulación V-REP, donde se han simulado 3 entornos realistas. Dos de estos tres entornos realistas se trataban de entornos sencillos, mientras que el ultimo se trataba de un entorno más complejo. Para mapear estos entornos se han utilizado dos modos de mapeo disponibles, el modo offline y el modo online asíncrono. Para los entornos pequeños, ambos modos han dado los resultados esperado, mientras que el modo online asíncrono ha sido necesario para mapear el ultimo entorno, ya que el modo offline está limitado a espacios más pequeños. Los resultados finales son los siguientes:

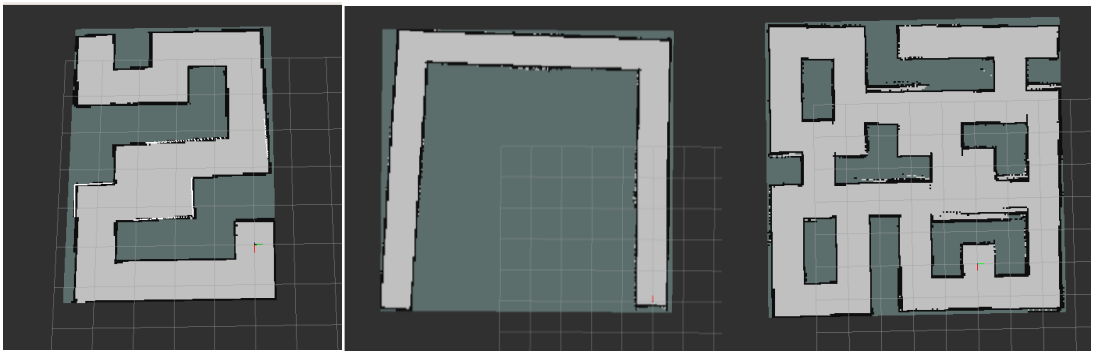


Figura 1.2 Mapas finales

6. Conclusiones

La finalidad de este proyecto era conseguir que un robot móvil sea capaz de construir un mapa de un entorno desconocido a medida que lo explora, mediante un algoritmo de localización y mapeado simultaneo (SLAM). Para ello se han comparado distintos algoritmos SLAM disponibles en ROS 2, y se ha integrado el más adecuado. La implementación del algoritmo ha sido satisfactoria y los resultados son los esperados.

Por último, quiero añadir que quedan pendientes trabajos futuros para continuar con el desarrollo de este proyecto o para utilizar sus resultados o su funcionalidad en futuros proyectos.

7. Referencias

[1] J. M. Santos, D. Portugal y R. P. Rocha, «An Evaluation of 2D SLAM Techniques Available, » 2013.

[2] S. Riisgaard y M. R. Blas, «A Tutorial Approach to Simultaneous Localization and Mapping, ». https://dspace.mit.edu/bitstream/handle/1721.1/119149/16-412j-spring2005/contents/projects/1aslam_blas_repo.pdf.

[3] S. Macenski, «SLAM Toolbox,» 2020. https://github.com/SteveMacenski/slam_toolbox.

IMPLEMENTATION OF SLAM ALGORITHMS IN THE V-REP SIMULATION ENVIRONMENT USING ROS 2

Author: Ampuero Gonzalez, Ignacio
Supervisor: Boal Martín-Larrauri, Jaime
Collaborating Entity: ICAI

ABSTRACT

The objective of this project has been to make a mobile robot capable of constructing a map of an unknown environment as it explores it. To achieve this, different simultaneous location and mapping algorithms (SLAM) available in ROS 2 have been compared, and the most suitable one has been integrated using the V-REP simulation environment in the process.

Keywords: SLAM, ROS, V-REP

1. Introduction

Industry 4.0 implies the promise of a new revolution, whose change is based on the adoption of new technologies such as collaborative robotics. The ROS framework is the ideal platform for developing applications for collaborative robotics. ROS gives us the possibility to implement technologies such as SLAM, which is breaking into industrial and commercial robotics, promoting a new generation of autonomous mobile robots, capable of operating alone in factories and in other environments.

2. Robot Operating System (ROS)

Robot Operating System (ROS) is a collection of frameworks for robot software development. ROS is the most popular standard adopted by the industry and there are several distributions of it. In this project we have worked with ROS2, which appeared in 2016, specifically with the Eloquent Elusor version (2019). The main difference with ROS1 is that in ROS2 the master/slave architecture, which prevailed in ROS1, disappears.

The nodes are the fundamental pieces of ROS. A ROS node is any piece of software in this system. ROS is based on a network architecture where the processing takes place in the nodes. The nodes can pass messages to each other through logical channels called topics and communicate with each other through a Publish/Subscribe mechanism.

ROS has several add-ons that have been used in this project. These have been RViz, rqt_graph and tf (transform), which have served to visualize the results and understand the functioning of ROS. All these tools are available in ROS2.

3. SLAM

Mobile robotics is the robotics that mostly uses ROS and its goal is to develop autonomous robots. The ability to navigate in unfamiliar environments, i.e. without a map of the environment, is an important part of what an autonomous robot means. When there is no map for navigation, it is said that a SLAM type navigation is being done (Localization and simultaneous mapping). SLAM navigation asks if it is possible for a mobile robot to build a consistent map of its environment while determining its position on that map simultaneously when it has been placed in an unknown environment. We

can say that this problem is like the chicken and egg problem: "The robot needs to know its position to build in map and it needs a map to determine its position". To solve this problem, the robot only has the information provided by the measurements obtained from the sensors and the notion of its own movement.

Most of the models used to solve the mapping problem are based on probabilities and built on Bayes' rules. These are extended Kalman filters (EKF), particle filters (PF), graph SLAM algorithms. The SLAM algorithm can be implemented in many ways and for this it is essential to have a mobile robot and a device capable of detecting the environment (commonly a laser or LIDAR sensor)

4. Work environment and algorithm implementation

The initial environment consisted of a scenario in the V-REP (3D robot simulator) simulation environment, composed of a robot with differential kinematics (Pioneer P3DX) and a set of walls that form paths. This robot is controlled by 4 external Python scripts which function was to initialize the robot, move it through the scenario following a wall and avoid collisions.

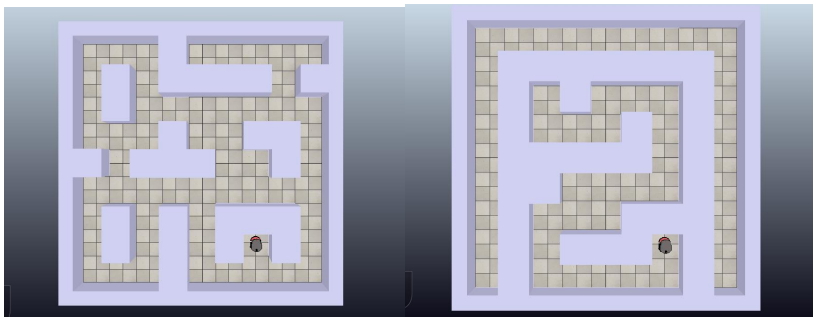


Figure 1.3 Simulated environments

A series of steps have been followed to implement the project. The first step has been to select the best SLAM algorithm for ROS2. SLAM Toolbox is the package that has been implemented as it is the SLAM standard for ROS2. This package receives messages from a laser sensor and generates a map of the environment. The second step is the search and selection of a laser sensor to be coupled to the robot. The Hokuyo URG-04LXUG01_Fast sensor (available in V-REP) was chosen and it was attached on top of the robot. The next step is to obtain the data from the laser sensor (with the robot in motion) and publish it in a ROS message. After that, to have a notion of the robot's movement, the odometry of the robot has been implemented. This allowed us to know how the robot moved linearly and angularly from its starting point. Without this step, the map could not be built, because without odometry, SLAM Toolbox considers the robot as static.

5. Results

The project has been developed and integrated with the V-REP simulation environment, where 3 realistic environments have been simulated. Two of these three realistic environments were simple environments, while the last one was a more complex environment. Two available mapping modes have been used to map these environments,

the offline mode, and the online asynchronous mode. For the small environments, both modes have given the expected results, while the asynchronous online mode has been necessary to map the last environment, since the offline mode is limited to smaller spaces. The results are as follows:

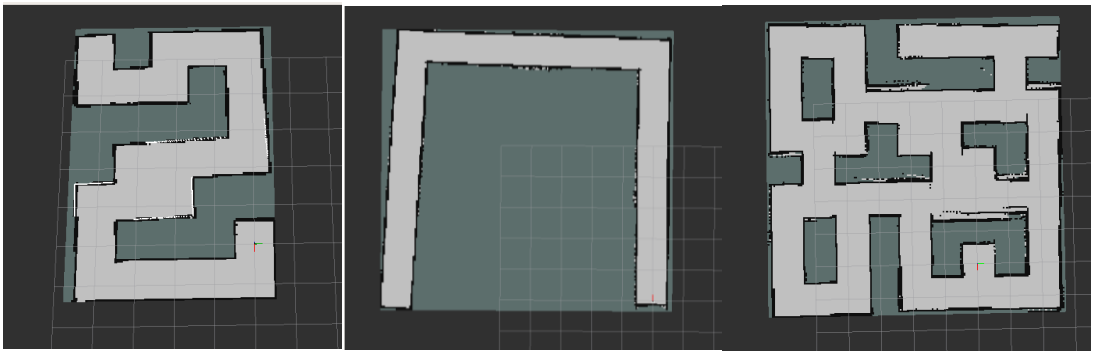


Figure 1.4 Final maps

6. Conclusions

The aim of this project was to make a mobile robot capable of building a map of an unknown environment as it explores it, using a simultaneous location and mapping algorithm (SLAM). For this purpose, different SLAM algorithms available in ROS 2 have been compared, and the most suitable one has been integrated. The implementation of the algorithm has been satisfactory, and the results are as expected.

Finally, we would like to add that future work is pending to continue with the development of this project or to use its results or functionality in future projects.

7. References

[1] J. M. Santos, D. Portugal y R. P. Rocha, «An Evaluation of 2D SLAM Techniques Available, » 2013.

[2] S. Riisgaard y M. R. Blas, «A Tutorial Approach to Simultaneous Localization and Mapping, ». https://dspace.mit.edu/bitstream/handle/1721.1/119149/16-412j-spring2005/contents/projects/1aslam_blas_repo.pdf.

[3] S. Macenski, «SLAM Toolbox,» 2020. https://github.com/SteveMacenski/slam_toolbox.

Índice de la memoria

Índice de figuras

Capítulo 1. INTRODUCCIÓN

1.1 MOTIVACIÓN DEL PROYECTO

La adopción de la tecnología digital nos ha llevado a un punto el cual estamos listos para otro cambio radical, una nueva revolución de la industria o lo que se denomina industria 4.0 [1]. La Industria 4.0 implica la promesa de una nueva revolución que combina técnicas avanzadas de producción y operaciones con tecnologías inteligentes que se integrarán en las organizaciones, las personas y los activos.

Este cambio está basado en la adopción de las nuevas tecnologías para aumentar la automatización del proceso productivo. Hablamos de tecnologías innovadoras cuya aplicación a la industria se desarrollará día a día. Algunas de estas tecnologías son: robótica colaborativa, fabricación aditiva, visión artificial, realidad virtual, herramientas de planificación de la producción, simulación de procesos, inteligencia operacional, IoT, y las denominadas KET, por su acrónimo inglés (*Key Enabling Technologies*) [1].

La revolución de la Industria 4.0 tiene su origen en las fábricas inteligentes. Actualmente, los grandes fabricantes emplean sus esfuerzos en I+D+i en dar forma y poner a punto estas nuevas tecnologías adaptándolas a sus líneas de producción y a los productos que salen de ellas. Así, el concepto de Robótica Colaborativa es central en esta revolución, donde ya no se trata de que las máquinas sustituyan a las personas, sino de que, de forma separada, cada parte se centre en las tareas que mejor hace y aportan más valor al producto:

El *framework* ROS (Robot Operating System) es la plataforma ideal para el desarrollo de aplicaciones de robótica colaborativa, donde diferentes dispositivos tienen que comunicarse para crear un entorno de producción flexible y escalable. Esta plataforma nos da la posibilidad de implantar tecnologías como SLAM (Software de Localización y Mapeo Simultáneos), la cual está irrumpiendo en la robótica industrial y comercial, dando pie a una

nueva generación de robots móviles autónomos, capaces de desenvolverse solos tanto en las fábricas como en otros entornos.

La tecnología SLAM permite a una máquina mapear el entorno en el que se encuentra para entender sus condiciones, localizarse en este entorno y deducir cómo moverse por él. Mediante el uso de diferentes tecnologías de sensores y posicionamiento, este software está consiguiendo cambiar los tradicionales vehículos guiados automatizados (AGV) a robots móviles autónomos (AMR). Estos vehículos están en auge en entornos industriales ya que, debido a su autonomía, permiten un nivel de automatización superior en estos. [2].

Como ejemplo de esto podemos tomar la reciente incorporación de esta tecnología en la fábrica Seat de Martorell.

El fabricante automovilístico Seat ha incorporado en su planta de Martorell vehículos de guiado automático en zonas exteriores, primero en Europa que cuenta con estos dispositivos [3].

Estos vehículos tienen navegación SLAM, conexión 4G y carga de baterías eléctricas por inducción. “A día de hoy, ocho AGV ya circulan fuera de los talleres de producción de la planta de Martorell para automatizar el transporte de piezas. Los nuevos vehículos se suman a los 200 AGV que ya suministran en la actualidad piezas en el interior de los talleres de montaje de la fábrica de la marca en Martorell y de Barcelona” [3].

“Gracias a la navegación SLAM, estos vehículos no circulan guiados por cinta magnética ni filoguiados, por lo que se reducen los costes de mantenimiento, son más versátiles para adaptarse a nuevas rutas y su instalación no necesita obra civil” [3].

1.2 OBJETIVOS

El objetivo de este proyecto es conseguir que un robot móvil sea capaz de construir un mapa de un entorno desconocido a medida que lo explora. Para ello se han comparado distintos algoritmos de localización y mapeado simultáneos (SLAM, por sus siglas en inglés) disponibles en ROS2, y se ha integrado el más adecuado usando en el proceso el entorno de simulación V-REP.

1.3 ESTRUCTURA DEL PROYECTO

Este trabajo está compuesto de 6 capítulos, incluido este capítulo introductorio. Para empezar, este capítulo sirve para dar una pequeña introducción al proyecto, para contar por qué de este proyecto y pone en contexto al lector del proyecto a realizar.

El Capítulo 2. y el Capítulo 3. , proporcionan una introducción y explicación del funcionamiento de los dos pilares fundamentales sobre los que se basa este proyecto, ROS Y SLAM.

En el Capítulo 2. se explica todo lo necesario sobre el *framework* ROS. El capítulo, empieza con una pequeña introducción para poner en contexto al lector y continua con una explicación sobre las bases sobre las que se establece ROS y las versiones que existen de él. El capítulo finaliza con una amplia explicación sobre el funcionamiento de ROS, y sobre el repertorio de herramientas, incluidas en ROS, que ha sido utilizadas en este proyecto.

El Capítulo 3. se encarga de explicar todo sobre el algoritmo de mapeo y localización simultaneo (SLAM), desde sus bases hasta las diferentes opciones que existen para implementarlo; concluyendo con una explicación más detallada de una de estas opciones.

Es en el Capítulo 4. donde se introduce todo el entorno de trabajo y el trabajo realizado. Este capítulo empieza con la explicación del entorno donde se ha desarrollado el trabajo, la manera en que se relacionan y conectan los componentes del proyecto entre sí y concluye con la explicación de como se ha implementación del algoritmo en el entorno proporcionado.

El Capítulo 5. es donde se muestran los resultados finales obtenidos tras haber conseguido implementar y optimizar el proyecto para conseguir los resultados esperados.

Y finalmente en el Capítulo 6. se establecen las conclusiones del proyecto. En estas se habla de todo el proceso de investigación e implementación del trabajo y se comentan brevemente sobre los posibles futuros trabajos que pueden ser aplicados a este proyecto.

Capítulo 2. ROBOT OPERATING SYSTEM (ROS)

2.1 INTRODUCCIÓN

Robot Operating System (ROS) es un *middleware* robótico, es decir, una colección de *frameworks* para el desarrollo de software de robots [4]. ROS es el estándar más popular adoptado por la industria de los robots y dispone de herramientas y librerías que permiten crear tus propias aplicaciones.

A pesar de no ser un sistema operativo, ROS provee los servicios estándar de uno de estos tales como la abstracción del hardware, el control de dispositivos de bajo nivel, la implementación de funcionalidad de uso común, el paso de mensajes entre procesos y el mantenimiento de paquetes [4].

ROS provee módulos de software para poder realizar las distintas funciones utilizadas en un robot, como la planificación de trayectorias, reconocimiento de objetos, y manipulación de objetos entre otras. Esto significa, que no se tendrá que empezar desde cero, sino que se pueden incorporar módulos previamente probados a cualquier proyecto [5]. Estos módulos funcionan independientemente, y pueden funcionar, conectarse o desconectarse sin afectar el funcionamiento general, pues no están controlados de manera central. ROS permite a un robot, ser controlado por cualquier cantidad de dispositivos; por ejemplo, un ordenador puede encargarse de la navegación, un servidor, del reconocimiento y análisis de imágenes y tener un móvil que reciba instrucciones de alto nivel, todo en tiempo real [5].

2.2 FILOSOFÍA DE ROS

La filosofía de ROS se puede resumir en los siguientes cinco principios [6]:

- Punto-a-Punto: Una arquitectura punto-a-punto permite los nodos que componen el robot puedan dialogar directamente con cualquier otro de forma síncrona o asincrónica, según sea necesario.
- Multilenguaje: ROS se puede programar en varios lenguajes de programación. Lo importante es que las clases desarrolladas permitan la generación de mensajes basados en el Lenguaje de Definición de Interfaz (IDL). C ++, Python, Java, C #, JavaScript, Ruby, …
- Basado en herramientas: En lugar de un entorno de tiempo de ejecución monolítico, ROS adoptó un diseño de microkernel, que utiliza una gran cantidad de pequeñas herramientas para construir y ejecutar sus diversos nodos.
- Ligero: Los algoritmos se empaquetan en ejecutables que son independientes, reutilizables y de tamaño reducido.
- Gratuito y de código abierto: Es soportado por una gran comunidad y se ha convertido un estándar en la industria de la robótica.

2.3 DISTRIBUCIONES DE ROS

Las primeras piezas de lo que sería ROS en un futuro surgieron en el Laboratorio de Inteligencia Artificial de Stanford, por el año 2007, para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford (STAIR2). Desde 2008, el desarrollo continuó principalmente en Willow Garage, un instituto de investigación robótico. Willow Garage empezó a desarrollar el robot PR2, continuación de su primera versión PR1, y ROS sería el software que lo controlaría. Grupos de más de veinte instituciones colaboraron en ROS, haciendo de ROS una plataforma colaborativa desde el principio [4].

Una distribución ROS es un conjunto versionado de paquetes ROS. Son similares a las distribuciones de Linux (por ejemplo, Ubuntu). El propósito de las distribuciones ROS es

permitir que los desarrolladores trabajen contra una base de código relativamente estable hasta que estén listos para avanzar todo. Por lo tanto, una vez que se lanza una distribución, se trata de limitar los cambios que surgen al corregir errores y las mejoras continuas para los paquetes principales. Para los paquetes de nivel "superior", las reglas son menos estrictas y los encargados del mantenimiento de un paquete son los responsables de evitar cambios importantes [7].

2.3.1 ROS2

Desde agosto de 2015, se lleva trabajando en la segunda versión de ROS, también conocida como ROS2. Sus inicios se dan lanzando su versión *alpha* que se mantiene hasta finales de 2016. A partir de ese año, se suceden 3 betas que acaban en septiembre de 2017 y tres meses después saldría la que sería la primera *release* estable de ROS2 denominada *Ardent Apalone*. Mas tarde llegaría una nueva versión denominada *Bouncy Bolson* [8]. En 2019, llegaría la versión denominada *Eloquent Elusor*. La última versión llegó este 2020 en junio con el nombre *Foxy Fitzroy*.

La seguridad y los sistemas en tiempo real impulsaron la creación de la segunda versión de ROS. A continuación, se muestran algunas de las diferencias más significativas entre ambas generaciones [8]:

- Lenguajes: Mientras que ROS trabaja con Python 2.7 y C++11, ROS2 da un salto importante permitiendo trabajar en Python 3.5 y en versiones superiores de C++ como la 14 y la 17.
- Sistemas operativos: ROS ya podía instalarse en Linux y, con alguna dificultad en macOs. ROS2 da el salto a Windows 10, abriéndose la puerta a un sistema operativo nada habitual por estos campos, pero muy común en otros campos y muy familiar para la gran mayoría del mundo.
- Centralizado vs Distribuido: Si hay una diferencia que destacar es sin duda la desaparición del *master* en ROS2. En ROS imperaba el modelo centralizado, con una arquitectura maestro/esclavo. Aunque el maestro (*Master*) solo se usaba para establecer la comunicación entre nodos, este era un requisito fundamental a la hora

de levantar nuestro sistema. En ROS2 se prescinde de éste haciendo que sean los propios nodos los que comuniquen al resto de nodos, de su aparición o marcha de la red.

2.4 FUNCIONAMIENTO DE ROS

Para poder explicar cómo funciona ROS, hay que entender lo que es un nodo ROS. Un nodo ROS (ROS *node*) es la representación de un proceso ROS, es decir, de cualquier pieza de software del sistema (desde un controlador para un motor hasta un algoritmo ya implementado). ROS está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos (recibir, mandar y multiplexar mensajes de sensores, control, estados, entre otros). Estos se representan en una estructura gráfica, conectados por bordes llamados ROS *topics* o temas [4].

Los *topics* son buses sobre los que los nodos intercambian mensajes. Los nodos se comunican entre ellos mediante un mecanismo del tipo *Publish*/*Subscribe* (Publicación/Suscripción). Los nodos que quieren compartir una información publicaran en el *topic* o *topics* apropiados. Un nodo que quiere recibir información se suscribirá a uno o varios *topics* en los cuales esté interesado. De manera que el nodo que quiere recibir información no tiene que realizar ninguna petición, y cada vez que el *topic* al que está suscrito sea actualizado, este nodo recibirá la información.

Los nodos se comunican entre sí mediante la publicación de mensajes a los *topics*. Un mensaje es una estructura de datos simple, que comprende campos escritos. Los tipos primitivos estándar (entero, booleano, etc.) son compatibles, al igual que las matrices de tipos primitivos. Los mensajes pueden incluir estructuras y matrices anidadas arbitrariamente (al igual que las estructuras C) [9]. Existen muchos tipos de mensajes y cada mensaje tiene una estructura única.

Los nodos ROS pueden pasar mensajes entre sí a través de los ROS *topics*, realizar llamadas de servicio a otros nodos, proporcionar un servicio para otros nodos o establecer o recuperar datos compartidos de una base de datos común llamada servidor de parámetros. En ROS1, "un proceso llamado ROS *master* (Figura 2.1) hace que todo esto sea posible al registrar nodos para sí mismo, configurar la comunicación de nodo a nodo para los temas y controlar las actualizaciones del servidor de parámetros. Los mensajes y las llamadas de servicio no pasan por el *master*, sino que el *master* establece la comunicación entre pares y entre todos los procesos de nodo y después de que estos se registren con el *master*" [4].

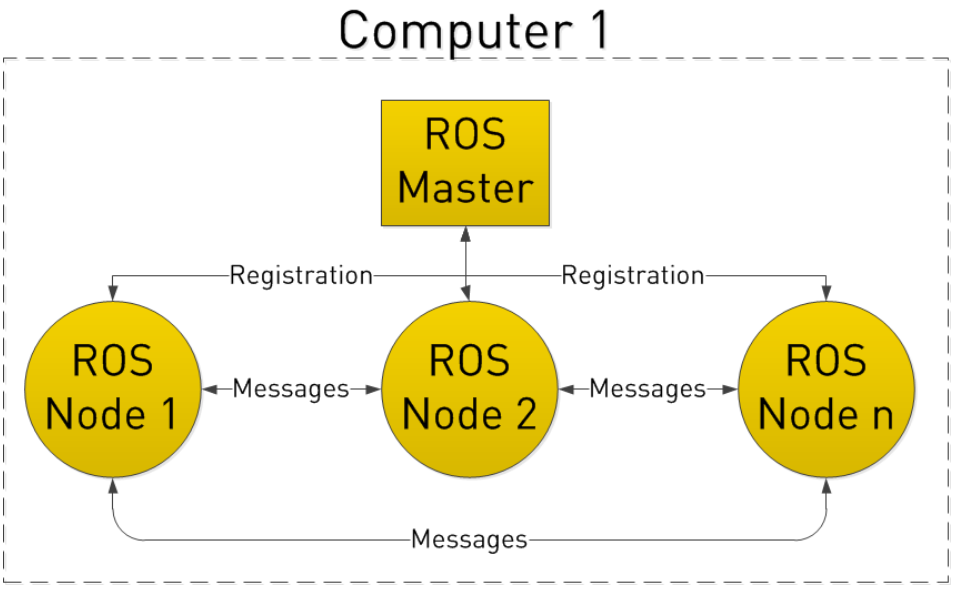


Figura 2.1 Nodos y Master ROS [10]

"Esta arquitectura descentralizada se presta bien a los robots, que a menudo consisten en un subconjunto de hardware informático en red, y pueden comunicarse con computadoras externas para realizar cálculos pesados" [4].

Pongamos un ejemplo. Digamos que tenemos una cámara en nuestro robot y queremos poder ver las imágenes de la cámara, tanto en el robot como en otro ordenador portátil (Figura 2.2).

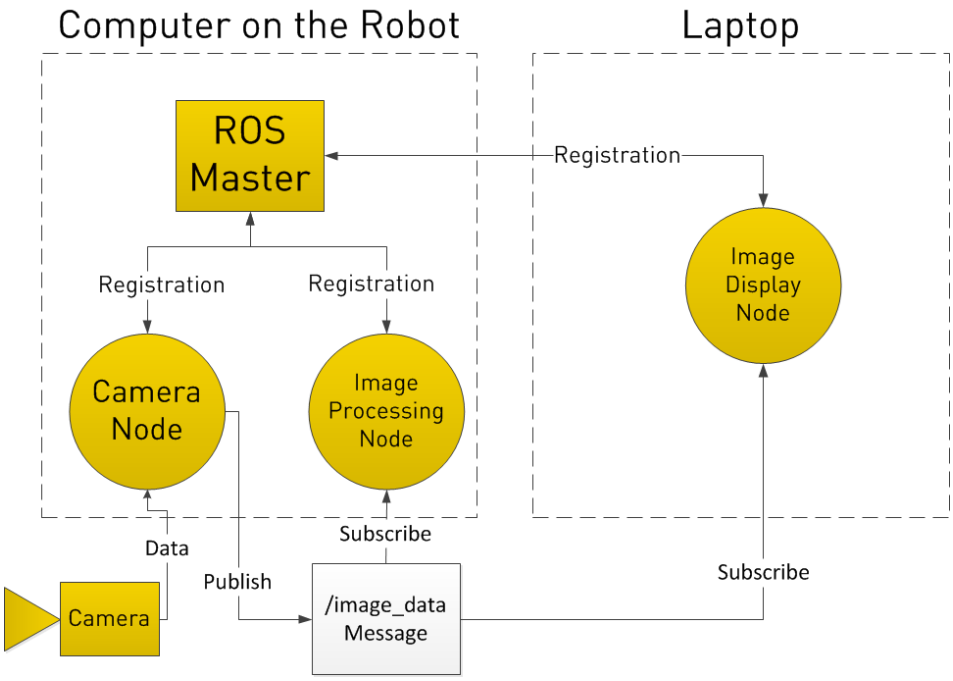


Figura 2.2 Funcionamiento ROS [10]

En la Figura 2.2, tenemos un nodo de la cámara que se encarga de la comunicación con la cámara, un nodo de procesamiento de imágenes en el robot, que procesa los datos de las imágenes, y un nodo de visualización de imágenes que muestra imágenes en una pantalla. Para empezar, todos los nodos se han registrado en el ROS *master*.

Al registrarse con el ROS *master*, el nodo de la cámara indica que publicará un *topic* llamado `/image_data` (por ejemplo). Los otros dos nodos registran que están suscritos al tema `/image_data`.

Por lo tanto, una vez que el nodo de la cámara recibe algún dato de la cámara, envía el mensaje `/image_data` directamente a los otros dos nodos.

Para concluir con este apartado, aclarar que en este proyecto se ha usado ROS2, más en concreto la versión *Eloquent Elusor*, cuyo funcionamiento es idéntico a ROS1, a excepción de que en ROS2 se prescinde del ROS *Master*, como se ha explicado anteriormente (2.3.1).

2.4.1 RQT_GRAPH

ROS cuenta con múltiples herramientas para entender mejor su funcionamiento. Una de estas herramientas es *rqt_graph*, la cual nos permite representar de manera gráfica el conjunto de nodos y *topics* que están en funcionamiento, así como sus relaciones (Figura 2.3)

Figura 2.3 Ejemplo de herramienta rqt_graph [11]

En la figura (Figura 2.3) podemos observar dos nodos ROS (azul y verde) y un *topic* (rojo). Ambos nodos se están comunicando mediante el *topic* llamado `/turtle1/command_velocity`. El nodo azul publica sobre el *topic* y el nodo verde esta suscrito a él. Un ejemplo más real de esta herramienta sería el siguiente.

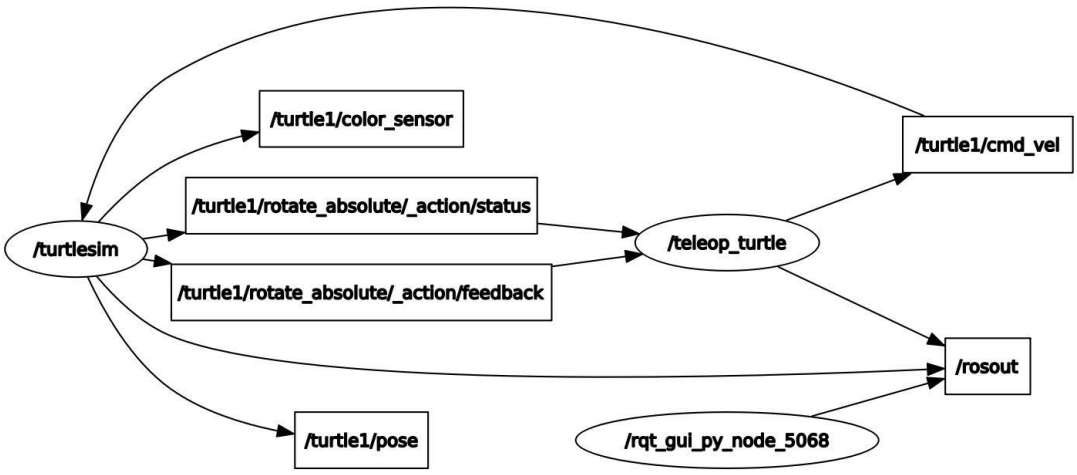


Figura 2.4 rqt_graph en ROS2

Rqt_graph representa los nodos rodeados mediante un círculo, y los *topics* encerrados en un rectángulo (Figura 2.4). Para representar el intercambio de mensajes utiliza flechas. Si la

flecha sale de un nodo y apunta a un *topic* significa el que el nodo publica en ese *topic*, el caso contrario, significa que el nodo esta suscrito a este *topic*.

2.4.2 RVIZ

RViz es una herramienta de visualización en 3D para aplicaciones de ROS. Proporciona una vista del modelo de robot, captura la información de los sensores del robot y reproduce los datos capturados. Puede mostrar datos de cámara, láseres y dispositivos 3D y 2D, como imágenes y nubes de puntos, entre otras cosas. RViz nos permite visualizar la mayoría de los tipos de mensajes pertenecientes a ROS en tiempo real. Para realizar las tareas que se indican a continuación, RViz debe estar abierto y conectado a un trabajo de simulación en ejecución

En este proyecto ha sido una herramienta esencial que ha servido para comprobar a cada paso si los pasos seguidos se estaban ejecutando correctamente a la vez que ha permitido visualizar el resultado final. Hay que mencionar que debido a que se ha utilizado ROS2, también se estaba usando RViz2, que es un simple *port* de la herramienta original a ROS2 (Figura 2.5).

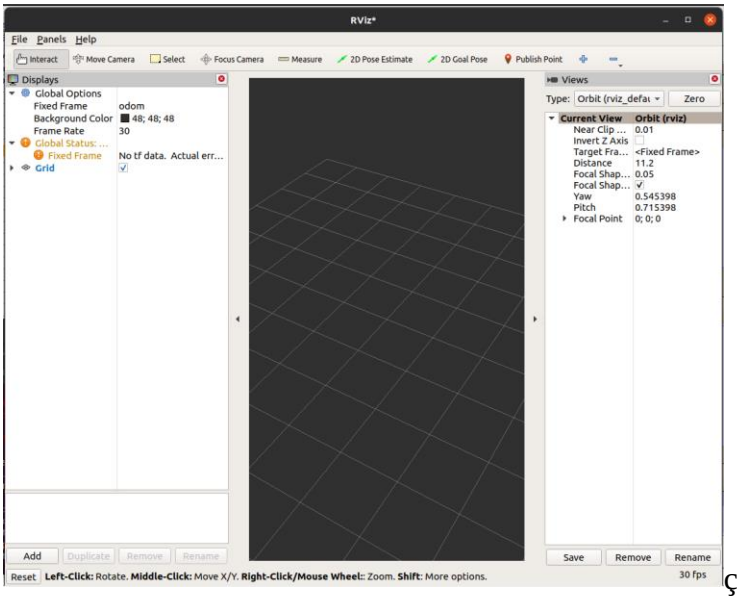


Figura 2.5 RVIZ2

2.4.3 ROS TF

Un sistema robótico generalmente tiene muchos marcos de coordenadas 3D que cambian con el tiempo, como un marco de la base del robot, un marco de mapa, etc.

tf, que es una abreviación de transformaciones o transformadas, es un paquete que permite al usuario realizar un seguimiento de múltiples marcos de coordenadas a lo largo del tiempo. tf mantiene la relación entre los marcos de coordenadas en una estructura de árbol almacenada en el tiempo y permite al usuario transformar puntos, vectores, etc. entre dos cuadros de coordenadas en cualquier momento. En ROS, existen marcos de coordenadas comunes, tales como la base del robot, la pose (posición y orientación) de un robot en referencia a la inicial, o el que pertenece a un mapa (Figura 2.6). Este paquete realiza un seguimiento de todos estos marcos con el tiempo y le permite hacer preguntas como:

- ¿Dónde estaba el marco de la cabeza en relación con el marco del mundo, hace 5 segundos?
- ¿Cuál es la pose actual de la base del robot con respecto al marco del mapa?

tf puede operar en un sistema distribuido. Esto significa que toda la información sobre los marcos de coordenadas de un robot está disponible para todos los componentes de ROS en cualquier dispositivo del sistema.

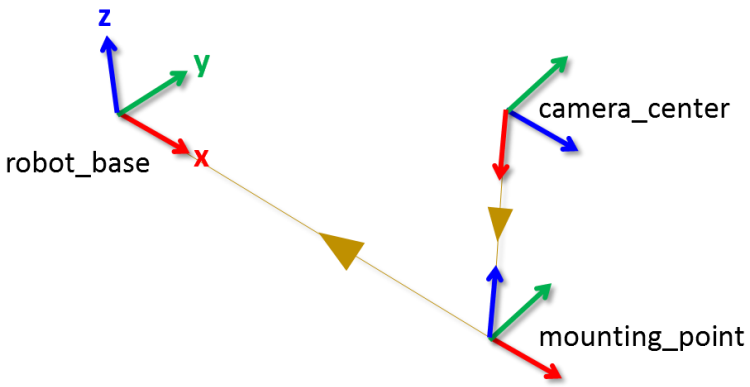


Figura 2.6 Ejemplo de marcos de coordenadas [12]

En este proyecto se ha usado tf2, que es simplemente la segunda generación de la biblioteca de transformación. Para que todos estos marcos de referencia estén disponibles para el sistema tienen que ser emitidos. Básicamente, hay dos tareas por las que cualquier usuario usaría tf2, escuchar transformaciones y transmitir transformaciones [13].

- Escuchar transformaciones: se recibe y guarde en el búfer todos los cuadros de coordenadas que se emiten en el sistema, para poder consultar las transformaciones específicas entre cuadros.
- Difundir transformaciones: se envía la pose relativa de los marcos de coordenadas al resto del sistema. Un sistema puede tener muchas partes que transmiten transformadas, las cuales proporcionan información sobre las diferentes partes del robot.

Capítulo 3. NAVEGACIÓN Y ALGORITMO SLAM

3.1 INTRODUCCIÓN

Una vez establecido lo que es ROS, tenemos que hablar de la robótica que lo usa, en concreto la robótica móvil. El objetivo de la robótica móvil es desarrollar entidades completamente autónomas capaces de llevar a cabo tareas difíciles, sin la necesidad de intervención humana durante largos periodos de tiempo. Estamos hablando de robots autónomos.

Para definir bien lo que es un robot autónomo podemos basarnos en las siguientes definiciones:

- Funcionar autónomamente implica que un robot puede operar, autocontenido, en variadas condiciones y sin necesidad de supervisión humana.
- Un sistema es autónomo en la medida en que su comportamiento está determinado por su propia experiencia.

De estas definiciones se desprende que la autonomía de un robot es necesaria para la solución de problemas planteados por la comunidad científica y la industria como el producto comercial Roomba [14], el coche autónomo de Google [15] y el desafío DARPA [16]. De estas definiciones se desprende también que la capacidad de navegar en ambientes desconocidos forma una parte importante de lo que significa un robot autónomo.

3.2 SISTEMA DE POSICIONAMIENTO ABSOLUTO Y RELATIVO

A la hora de hablar de navegación se tiene que establecer un sistema de referencia para el robot. Un sistema de coordenadas es un conjunto de valores y puntos que permiten definir unívocamente la posición de cualquier punto de un espacio. En robótica móvil se usan habitualmente robots terrestres por lo que se usan sistemas de coordenadas cartesianos de dos dimensiones; ya que estos robots se mueven en un plano. [17].

Entonces, se puede definir un sistema de referencia con un sistema de coordenadas y un punto de referencia; y dependiendo de ese punto de referencia el sistema de referencia será distinto.

Para transformar un punto de un sistema de referencia a otro sistema de referencia, hay que tomar las coordenadas de ese punto, multiplicarlas por una matriz de rotación y a esto sumarle una matriz de traslación.

La ecuación para transformar las coordenadas de un punto de un sistema de referencia local a las de un sistema global es la siguiente:

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} \cos\varphi & -sen\varphi \\ sen\varphi & \cos\varphi \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \end{pmatrix} + \begin{pmatrix} Xo \\ Yo \end{pmatrix}$$

Transformada entre sistemas de referencia [17]

Siendo:

- (X, Y) coordenadas en el SR global del punto que estamos transformando.
- (Xo, Yo) coordenadas en el SR global del centro del SR local.
- (x', y') coordenadas en el SR local del punto que estamos transformando.
- φ ángulo diferencia entre los dos sistemas de referencia.

En la Figura 3.1 se muestra la diferencia entre los dos sistemas de referencia que se han mencionado.

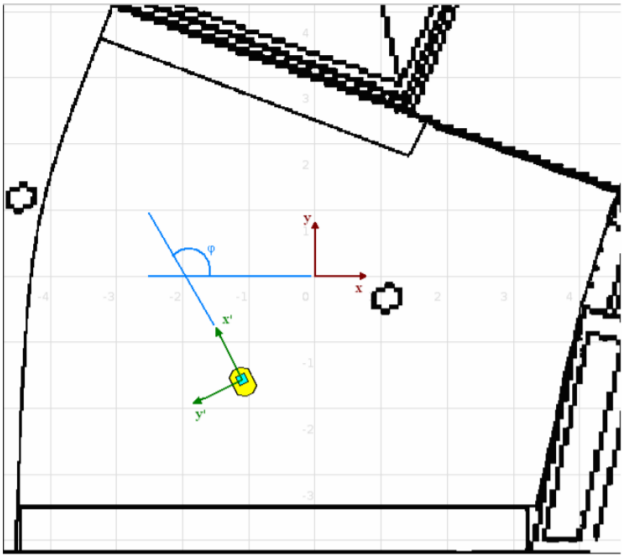


Figura 3.1Sistemas de referencia global (rojo) y local (verde) [17]

En robótica móvil, cuando se toma como punto de referencia el centro del robot, se habla de un sistema de referencia local. Estos sistemas de referencias se utilizan para hacer navegación reactiva. Si el punto de referencia es un punto externo al robot, se habla de un sistema de referencia global. Estos otros sistemas se utilizan para hacer navegación global [17].

3.2.1 NAVEGACIÓN CON POSICIONAMIENTO RELATIVO

La utilización de un sistema de posicionamiento relativo para navegar se conoce como navegación local. Debido a que la navegación local está basada en un sistema de posicionamiento relativo, las posiciones que toma el robot a lo largo de sus movimientos tienen como referencia, la posición inicial que tenía el robot al principio de la aplicación/simulación que se esté realizando. Para hallar las posiciones del robot relativas al inicio, normalmente se usan las ecuaciones correspondientes a la odometría del robot, que tienen en cuenta las dimensiones y las características de éste. Este sistema de odometría se basa en sumar incrementos de posición a la posición inicial del robot, esto es lo que se conoce como *Dead-Reckoning*. Por esto, el gran problema de este método es la acumulación de errores a la hora de estimar la posición del robot, la cual crece con el tiempo y la distancia recorrida [17].

Al trabajar sin un mapa del entorno se considera que este tipo de navegación es reactiva, es decir, el robot se desplaza de un punto a otro del plano, utiliza sus sensores externos para detectar obstáculos, y sigue algún protocolo de actuación para evitarlos. Luego el robot necesita algún tipo de sensor externo para detectar el mundo que lo rodea.

3.2.2 NAVEGACIÓN CON POSICIONAMIENTO ABSOLUTO

La utilización de un sistema de posicionamiento absoluto para navegación se conoce como navegación global. Al basarse en un sistema de posicionamiento absoluto, las posiciones que toma el robot a lo largo de sus movimientos están referidas a un punto externo a él. Este tipo de navegación se puede realizar con o sin mapa [17].

Cuando no se dispone de mapa para la navegación, se dice que se está haciendo una navegación tipo SLAM (Simultaneous Localization And Mapping). Básicamente, esto consiste en construir un mapa de un entorno desconocido en el que se encuentra el robot, a la vez que mantiene información de su trayectoria dentro de dicho entorno. Este tipo de navegación es más robusta que la mencionada en el aparatado anterior, ya que los errores de posicionamiento asociados al *Dead-Reckoning* se van corrigiendo mediante un método

Predictor-Corrector. En los apartados siguientes se explicará con más detalle esta navegación.

De todo esto podemos extraer que el mapa del entorno es una parte fundamental de este problema, tanto a la hora de construirlo, como usarlo para navegación global. Dos de los tipos de mapas más usados en navegación son:

- Un mapa topológico se define por un conjunto de lugares diferentes y otro conjunto que caracterizan las relaciones entre estos. Este se modela como un grafo. En los mapas topológicos se representan mediante nodos los lugares del entorno con alguna característica concreta, y con arcos se representan las conexiones de dichos nodos. A pesar de que estos mapas requieren poco en cuanto a capacidad de cómputo, su principal problema es el *perceptual aliasing*, es decir, que siempre hay un riesgo de que dos localizaciones sean consideradas como la misma por el robot [18].
- Un mapa métrico representa el entorno como un conjunto de coordenadas de objetos y obstáculos con la ayuda de datos en bruto y características geométricas. En los mapas métricos se representa el espacio libre y el espacio ocupado (obstáculos) de forma geométrica. Estos tipos de mapas a pesar de proporcionar una representación muy detallada del entorno, requiere una capacidad muy grande de cómputo [18].
- Un mapa hibrido es una combinación de los dos previos. Estos intentan compensar los errores que presentan ambos mapas combinando ambos, de manera que usa un mapa topológico para moverse entre zonas y mapas métricos detallados en cada una de esas zonas [18].

En los últimos años los métodos topológicos se han quedado un poco obsoletos debido a que la capacidad de cálculo y almacenamiento ha aumentado, permitiendo mantener mapas métricos más grandes. En este proyecto se trabaja con navegación con mapa métrico.

Hay que mencionar también que es posible hacer navegación global con un mapa, construido previamente usando SLAM. Haciendo navegación global con mapa, aparte de necesitar el mapa, se necesita un sistema de posicionamiento absoluto o global que es el que

da la posición absoluta del robot con el que se trabaja, y un navegador local para que realice la evasión de obstáculos.

3.3 EL PROBLEMA DE SLAM

El problema de SLAM plantea si es posible situar un robot móvil en un entorno desconocido y que sea capaz de construir un mapa consistente de su entorno mientras determina su posición en ese mapa simultáneamente. Podemos decir que este problema es como el problema del huevo y la gallina: "El robot necesita saber su posición para construir in mapa y necesita un mapa para determinar su posición" [19].

Para solucionar este problema, el robot solo dispone de la información proporcionada por los datos obtenidos de los sensores y la noción de movimiento propio. En el contexto del SLAM existen diversas fuentes de incertidumbre, es decir factores que hacen más difícil la estimación de la posición del robot y la creación del mapa del entorno. Como Federico Andrade y Martin Llofriu explican en [20], estas fuentes de incertidumbre son:

- Ruido de los sensores: Los sensores utilizados en un robot presentan usualmente ruido en los datos que proporcionan.
- Desplazamiento impreciso del robot: El resultado de un movimiento del robot es de naturaleza no determinista. Esto se debe a que, por ejemplo, las ruedas de un robot pueden resbalar sobre el suelo. Como resultado, no es posible conocer a ciencia cierta como fue el movimiento real del robot como resultado de una orden de movimiento o en base a información de odometría.
- Simetrías en el entorno: El entorno sobre el que opera el robot puede presentar simetrías que dificultan la determinación de la posición actual del robot en el mapa confeccionado.
- Observabilidad parcial: La ausencia de un mecanismo de visión global del entorno dificulta la estimación de la posición y la confección del mapa.

- Entorno dinámico: Si se trabaja en un entorno dinámico, los cambios en el entorno dificultarán el proceso de estimación de la posición debido a que el mapa confeccionado puede estar desactualizado.

Para solucionar esto, la gran parte de las soluciones de SLAM plantea la solución de la estimación de la posición del robot y el mapa del entorno como distribuciones de probabilidades.

Existen dos grandes enfoques para la solución del problema SLAM, bioinspirados y probabilísticos. Estos enfoques difieren principalmente en la forma en que procesan la información de entrada de forma de encontrar una buena estimación de la ubicación del robot y mapa del entorno. Actualmente, la mayoría de los modelos usados para resolver SLAM están basados en el enfoque probabilístico, como veremos más adelante.

Los métodos probabilísticos se concentran en encontrar la distribución de probabilidad de la posición del robot y del mapa del entorno a través del tiempo. Las soluciones al problema probabilístico SLAM implican encontrar una representación apropiada tanto para el modelo de observación como para el modelo de movimiento que permita el cálculo eficiente y consistente de las distribuciones anteriores y posteriores.

3.4 IMPLEMENTACIÓN DE SLAM

Para empezar, hay que decir que SLAM se puede implementar de muchas maneras. En primer lugar, hay una gran cantidad de hardware diferente que se puede utilizar. En segundo lugar, SLAM se parece más a un concepto que a un algoritmo único. Hay muchos pasos involucrados en SLAM y estos pasos pueden implementarse usando algoritmos diferentes. Debido a que se está trabajando con un robot terrestre, se utilizara la alternativa de SLAM aplicada a movimiento 2D, aunque también se dispone de la alternativa 3D.

SLAM es un proceso complejo que consta de múltiples partes: Extracción de *landmarks* o puntos de referencia, asociación de datos, estimación de estado, etc. Hay muchas formas de resolver cada una de las partes más pequeñas, lo que también significa que algunas de las

partes pueden reemplazarse por una nueva forma de hacerlo [21]. Para poder implementar SLAM, es esencial tener un robot móvil y un dispositivo capaz de detectar lo que tiene alrededor.

3.4.1 ROBOT

A la hora de elegir que robot utilizar, hay que tener en cuenta una serie de parámetros. Entre ellos está la facilidad de uso, el rendimiento de la odometría y su precio. El rendimiento de la odometría mide lo bien que el robot puede estimar su propia posición, sólo por la rotación de las ruedas.

3.4.2 DISPOSITIVO DE MEDICIÓN DE RANGOS

El dispositivo de medición de rangos/alcances utilizado hoy en día, suele ser un escáner láser o LIDAR. Estos son muy precisos, eficientes y su lectura (*output*) no requiere mucha computación para procesar. Los problemas con los escáneres de láser son con ciertas superficies, incluyendo el vidrio, donde pueden dar muy malas lecturas. Además, los escáneres de láser no pueden ser usados bajo el agua ya que el agua interrumpe la luz y el alcance se reduce drásticamente [21].

En segundo lugar, está la opción del sonar. El sónar se usó intensamente hace algunos años. Son muy baratos comparados con los escáneres láser. Sus mediciones no son muy buenas comparadas con los escáneres láser y a menudo dan malas lecturas. Bajo el agua, sin embargo, son la mejor opción y se asemejan a la forma de navegar de los delfines [21].

La tercera opción es usar la visión. Tradicionalmente ha sido muy intensivo en computación para usar la visión y también propenso a errores debido a los cambios en la luz. Dada una habitación sin luz, un sistema de visión ciertamente no funcionará. En los últimos años, sin embargo, ha habido algunos avances interesantes dentro de este campo. El uso de la visión se asemeja a la forma en que los humanos miran el mundo y por lo tanto puede ser más intuitivo que el láser o el sonar. También hay mucha más información en una imagen comparada con los escaneos de láser y sonar. Esto solía ser el cuello de botella, ya que todos

estos datos necesitaban ser procesados, pero con los avances en los algoritmos y la potencia de cálculo esto se está convirtiendo en un problema menor [21].

3.4.3 MODELOS PARA RESOLVER SLAM

Como se mencionó al principio del capítulo, actualmente todos los algoritmos reconocidos para el mapeo mediante robots tienen una característica común: están basados en probabilidades. La ventaja de aplicar probabilidades es la solidez a la hora de la medición de ruido y la capacidad de representar formalmente la incertidumbre en el proceso de medición y estimación. La mayoría de los modelos usados para resolver el problema del mapeado están construidos sobre las reglas de Bayes [22]:

- Los filtros Kalman (KF) son una de las implementaciones más populares de los filtros Bayes. El KF tiene dos fases distintas: predicción y actualización. La fase de predicción estima el espacio de estado (anterior) a partir de una iteración anterior, mientras que, en la fase de actualización, el estado estimado se combina con las observaciones proporcionadas por los sensores. El resultado de la fase de actualización se llama posterior. A partir del desarrollo previo de la KF, surgió el Filtro Kalman Extendido (EKF), el cual resuelve el problema de la no linealidad en el modelo de pose del robot.
- Los filtros de partículas (PF) son otra aplicación de los filtros Bayes. La probabilidad posterior está representada por un conjunto de partículas ponderadas y cada partícula tiene un factor de importancia. Este supone que el siguiente estado depende solo del actual (aproximación de Markov). Los PF tienen la ventaja de representar la incertidumbre a través de distribuciones multimodales y tratar el ruido no gaussiano. Montemerlo et al. [23] propuso un nuevo enfoque llamado FastSLAM. Utiliza un FP modificado para estimar el posterior. Luego, cada partícula posee un numero K de filtros Kalman los cuales estiman las K ubicaciones de los puntos de referencia (landmarks). Se demostró que el esfuerzo computacional para ejecutar este algoritmo es menor que los enfoques EKF. También, el enfoque se ocupa de un gran número

de puntos de referencia, incluso con pequeños conjuntos de partículas y los resultados siguen siendo apropiados.

- Igualmente, importantes son los algoritmos SLAM basados en gráficos (graph SLAM), ya que cubren algunas debilidades de las técnicas PF y EKF. En estos algoritmos SLAM, los datos extraídos se utilizan para construir un gráfico. El gráfico está compuesto por nodos y arcos. Cada arco en el gráfico representa una restricción entre poses sucesivas, que puede ser un evento de movimiento o un evento de medición. Una vez que se construye dicho gráfico, el problema crucial es encontrar una configuración de los nodos que sea los más consistente posible con las medidas Esto implica resolver un gran problema de minimización. Thrun et al. [24] presentó el algoritmo GraphSLAM, que se basa en estos algoritmos, y evaluó su comportamiento en entornos urbanos a gran escala.

3.4.4 EL PROCESO SLAM

Tras haber visto los modelos existentes para resolver SLAM, se explicará a grandes rasgos en qué consiste un proceso de mapeo y localización simultanea empleando un filtro extendido de Kalman (EKF).

Como hemos podido ver el proceso SLAM consiste en una serie de pasos. Su objetivo principal es utilizar los datos del entorno para actualizar la posición del robot. Estos datos sueles ser proporcionados por un sensor, generalmente un sensor laser (LIDAR). Esto es así, debido a que la odometría del robot (que proporciona la posición del robot) a menudo es errónea, y va acumulando un error a la hora de estimar la posición, como se mencionó, por lo que no podemos confiar directamente en ella. Es por esto por lo que se usan escaneos láser del entorno para corregir la posición del robot. Esto se logra extrayendo características del entorno y la re-obesrvacion cuando el robot se mueve. El responsable de actualizar la estimación de la posición del robot es el EKF, y este se basa en esas características. Estas características se denominan comúnmente puntos de referencia o *landmarks*.

Cuando los datos de odometría cambian debido a que el robot se mueve, la incertidumbre relativa a la nueva posición del robot se actualiza en el EKF mediante una actualización de

odometría. Es entonces cuando se extraen del entorno las *landmarks* de la nueva posición. Tras esto, el robot intenta asociar estas *landmarks* a observaciones de *landmarks* previamente hechas. Las *landmarks* re-observadas se utilizan para actualizar la posición del robot en el EKF, mientras que las nuevas *landmarks*, es decir que nunca han sido observadas, se agregan al EKF como nuevas observaciones, para que puedan volver a observarse más tarde [21]. En las siguientes figuras se explicará el proceso en más detalle:

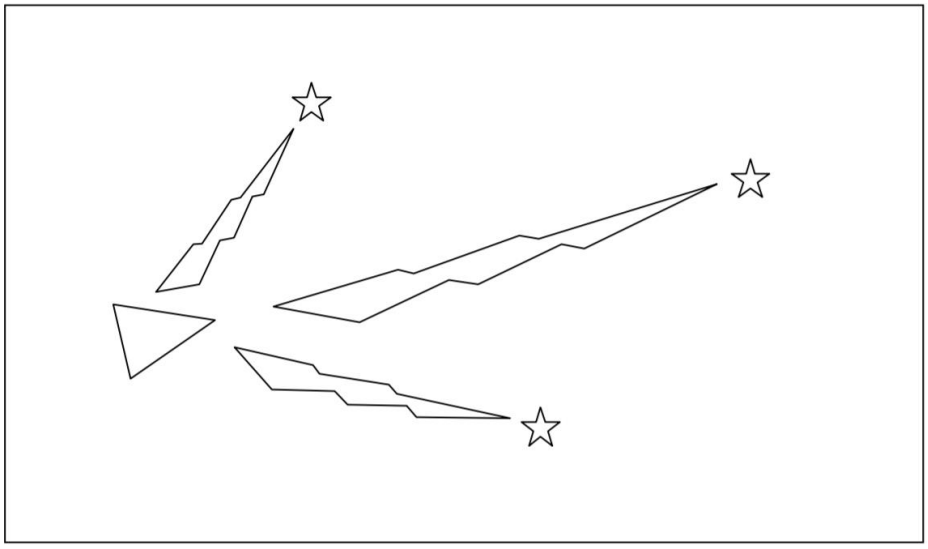


Figura 3.2 El robot está representado por el triángulo. Las estrellas representan landmarks. Las landmarks son detectadas inicialmente por el sensor láser. (las medidas del sensor vienen representadas por rayos) [21]

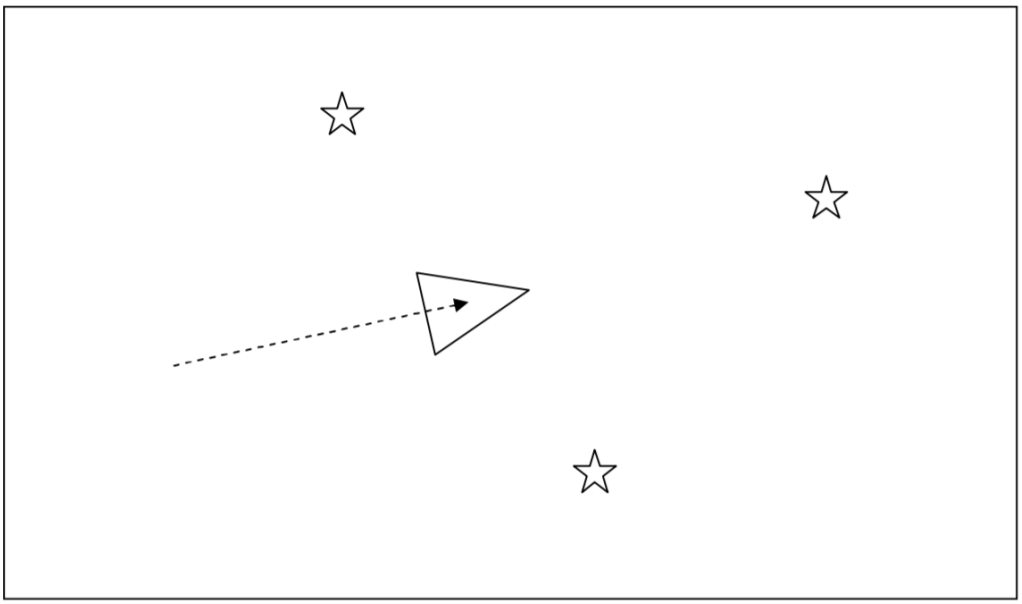


Figura 3.3 El robot se mueve y piensa que está aquí, según los datos aportados por los datos de odometría [21]

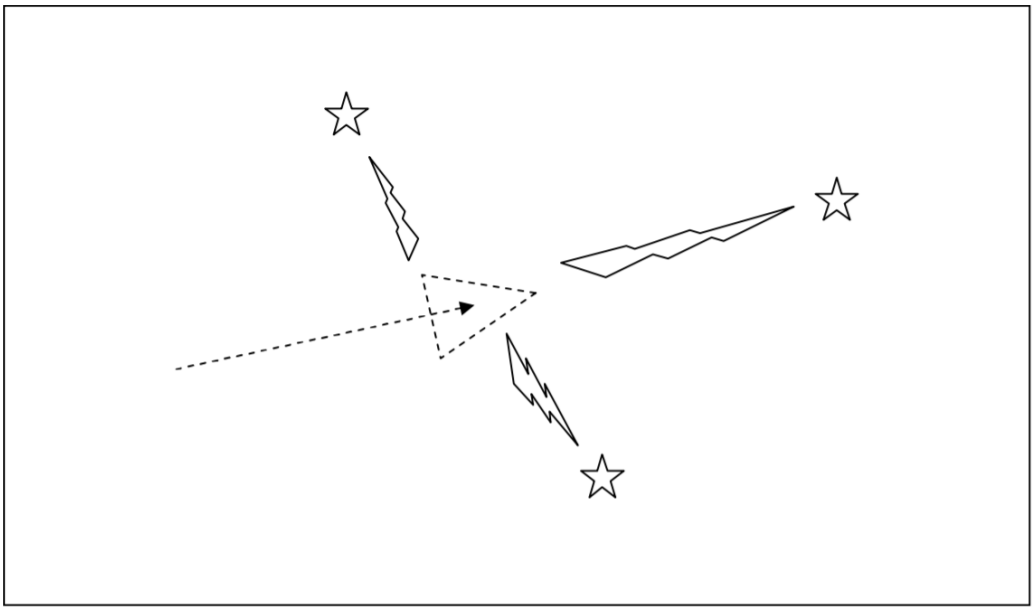


Figura 3.4 3 El robot mide de nuevo la localización de las landmarks usando el sensor láser, pero resulta que no coincide con donde el robot piensa que debería estar (dada la posición estimada según los sensores de odometría). La posición estimada es errónea [21]

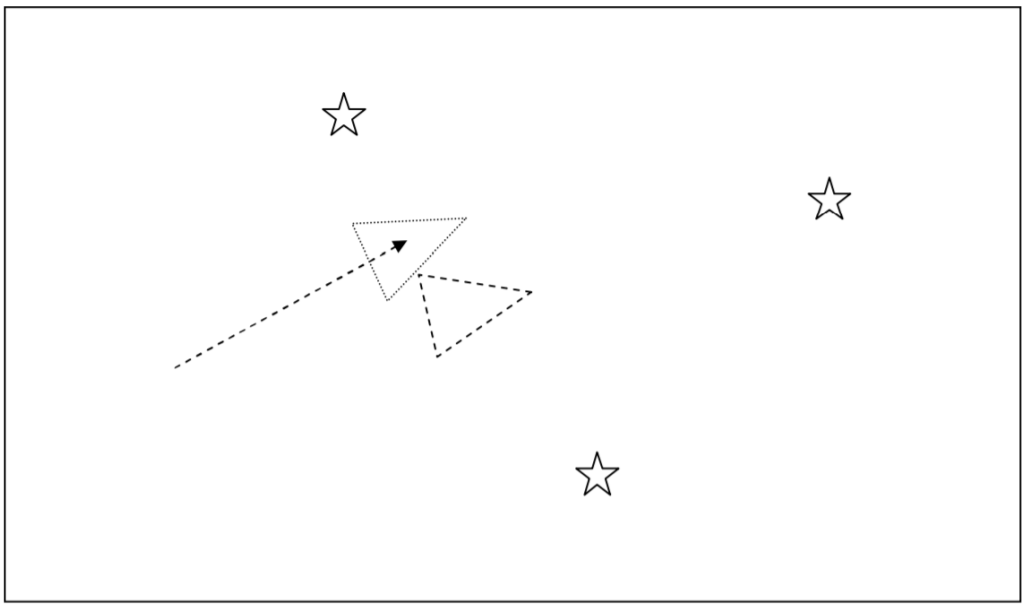


Figura 3.5 Como el robot se fía más de sus sensores que de la edometría, la nueva información acerca de la posición real de las landmarks es usada para corregir la posición estimada subsanando así una parte importante del error de posición acumulado (El triángulo a trazos representa la posición estimada inicial, y el punteado la posición estimada después de la corrección) [21]

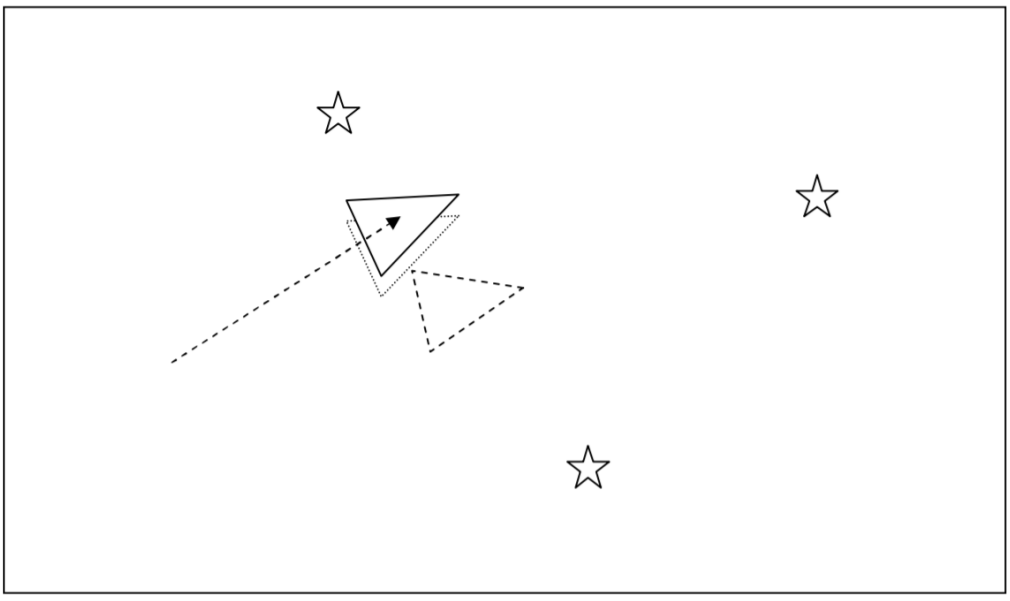


Figura 3.6 El robot está realmente donde el triángulo de trazado continuio, por lo que realmente la posición estimada corregida (triangulo punteado) no se corresponde al 100% con la posición real. No obstante, la nueva estimación es mejor que la inicial utilizando solamente la odometría

3.4.5 LANDMARKS

Se hará una breve explicación de lo que son las citadas *landmarks* y los métodos que se utilizan para extraerlas.

Las *landmarks* son características que se pueden volver a observar y distinguir fácilmente del entorno. El robot los utiliza para averiguar dónde está (para ubicarse). Una forma de imaginar cómo funciona esto para el robot es imaginarse con los ojos vendados. Si te mueves con los ojos vendados en una casa, puedes alcanzar y tocar objetos o abrazar paredes para que no te pierdas. Cosas características como la que se siente al tocar el marco de una puerta pueden ayudarlo a establecer una estimación de dónde se encuentra. Las sondas y los escáneres láser serian el tacto de los robots [21].

Las *landmarks* deben ser re-observables permitiéndoles, por ejemplo, ser vistos (detectado) desde diferentes posiciones y, por lo tanto, desde diferentes ángulos. Estas deben ser lo suficientemente únicas para que puedan identificarse fácilmente desde uno paso de tiempo a otro sin mezclarlas.

Los puntos clave sobre puntos de referencia adecuados son los siguientes:

- Las *landmarks* deben ser fácilmente re-observables.
- Las *landmarks* individuales deben distinguirse unas de otras.
- Las *landmarks* deben ser abundantes en el entorno.
- Las *landmarks* deben ser estacionarias.

Las dos formas más comunes de extracción de *landmarks* usando un scanner laser son las llamadas *spike landmarks* (*landmarks* punta) y RANSAC (*Random Sampling Consensus*) [21].

Las *spike landmarks* encuentran valores en el escaneo láser que difieran entre sí en más de cierta cantidad; esto suele reflejar, por ejemplo, cuando una parte del barrido detecta una pared con un huevo o cuando detecta una esquina entre dos paredes.

RANSAC es un método que se basa en extraer líneas rectas del escaneo láser. Estas líneas pueden ser usadas como *landmarks*. Este método, usa una aproximación por mínimos cuadrados sobre muestras obtenidas por las mediciones del láser, encontrando la mejor recta que aproxima dichas medidas. Una vez hecho esto se comprueba cuantas medidas del láser coinciden mejor con esta recta, y si este número supera cierta cantidad se puede asumir con seguridad que hemos visto una recta (que suele coincidir con un segmento de pared).

Capítulo 4. ENTORNO DE TRABAJO E IMPLEMENTACIÓN DEL ALGORITMO

4.1 INTRODUCCIÓN

En este apartado se procederá a explicar los componentes de este proyecto y la implementación que se ha llevado a cabo en él. Es decir, puesta en funcionamiento del sensor láser, redacción de código en Python para almacenar las medidas laser y para implementar la odometría y por último implementación del algoritmo de SLAM.

Hay que mencionar que la fase inicial del proyecto se compone de 4 scripts programados en Python, los cuales se encargan de inicializar y desplazar a un robot con cinemática diferencial a lo largo de un escenario en un entorno de simulación.

4.2 ANÁLISIS DEL SISTEMA

4.2.1 ENTORNO DE SIMULACIÓN V-REP

V-REP es un simulador de robot 3D basado en una arquitectura de control distribuida: los programas de control (o scripts) pueden conectarse directamente a los objetos de escena y ejecutarse de forma simultánea con o sin hilos. Esto hace que V-REP sea muy versátil e ideal para aplicaciones multi robot, y permite a los usuarios modelar sistemas robóticos de forma similar a como lo hacen en la realidad, donde el control también se distribuye la mayor parte del tiempo.

El elemento principal de V-REP son las escenas. Cuando se inicializa el simulador se abre una escena nueva (Figura 4.1), que es por así decirlo el “lienzo” donde se podrá llevar a cabo el proyecto deseado.

Los elementos principales en V-REP que se usan para construir una escena de simulación son los objetos de escena (en resumen, objetos). Los objetos son visibles en la jerarquía de escena y en la vista de escena. En la vista de escena, los objetos tienen una representación tridimensional. El simulador, en el lateral izquierdo tiene un menú de todos los objetos disponibles que puedes incluir en el escenario (robots móviles e inmóviles, sensores, paredes…). Hay un total de 14 tipos de objetos, los cuales puedes combinar entre ellos y formar sistemas complejos con módulos de cálculo y mecanismos de control. Para incluir cualquier objeto en tu escena, basta con arrastrar el objeto que desees a la escena y ya formara parte de ella.

Cada objeto tiene una posición y orientación dentro de la escena de simulación. Nos referimos a la posición y orientación de un objeto como configuración del objeto. Los objetos se pueden unir a otros objetos (o construirse uno encima del otro). Si el objeto A está construido sobre el objeto B, entonces el objeto B es el padre y el objeto A es el hijo. Para crear una relación padre-hijo entre el objeto B y el objeto A, se selecciona el objeto A, luego se selecciona el objeto B (el orden de selección es importante) [25].

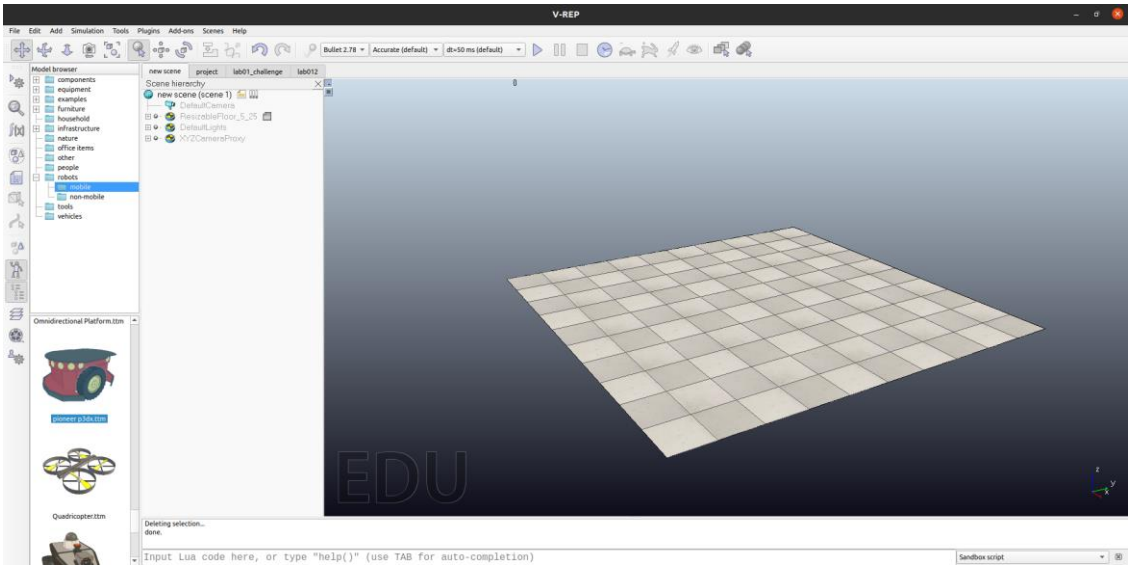


Figura 4.1 Interfaz de VREP (nueva escena)

Cada aspecto de una simulación puede ser personalizado. Además, el simulador mismo puede ser personalizado y adaptado para que se comporte exactamente como se desea. Esto se permite a través de una elaborada Interfaz de Programación de Aplicaciones (API). Se admiten seis enfoques diferentes de programación o codificación, cada uno de los cuales tiene ventajas particulares (y obviamente también desventajas) sobre los demás, pero los seis son mutuamente compatibles. Estas opciones son un script en el propio simulador, un plugin, un nodo ROS o BlueZero, un cliente API remoto o un *add on*. Esto hace que V-REP sea muy versátil e ideal para aplicaciones multi robot. Todos estos controladores pueden ser escritos en C/C++, Python, Java, Lua, Matlab u Octave [25].

Los enfoques que se han utilizado en este proyecto son:

- Sript incorporado: este método, que consiste en escribir scripts de Lua, es muy fácil y flexible, con una compatibilidad garantizada con todas las demás instalaciones predeterminadas de VREP. Este método permite personalizar una simulación particular, una escena de simulación, y hasta cierto punto el propio simulador. Es el enfoque de programación más fácil y utilizado. Un script principal es un script de simulación. Por defecto, cada escena en V-REP tendrá un script principal. Contiene el código básico que permite ejecutar una simulación. Sin el script principal, una simulación en ejecución no hará nada. A los scripts asociados a los objetos se les suele llamar *child* scripts (scripts hijos).
- Cliente API remoto: este método permite a una aplicación externa (por ejemplo, ubicada en un robot, otra máquina, etc.) conectarse a V-REP de forma muy sencilla, utilizando comandos API remotos. El API remoto forma parte del marco del API de V-REP. Permite la comunicación entre V-REP y una aplicación externa (es decir, una aplicación que se ejecuta en un proceso diferente, o en una máquina diferente), es multiplataforma y soporta llamadas de servicio y el flujo de datos bidireccional. En este proyecto se ha usado un cliente API remoto escrito en Python.
- Nodo ROS: este método permite que una aplicación externa (por ejemplo, ubicada en un robot, otra máquina, etc.) se conecte a V-REP a través de ROS.

4.2.2 DISEÑO INICIAL

En lo que respecta a este proyecto, el entorno inicial consistía en un conjunto de paredes que crean distintos recorridos dentro de la escena, y de un robot autónomo con cinemática diferencial.

4.2.2.1 Pioneer P3-DX

Pioneer P3-DX (Figura 4.2) es el modelo del robot que se ha implementado en el proyecto. Consiste en un pequeño robot de dos ruedas con cinemática diferencial, muy utilizado en el ámbito académico y de la investigación.

La mayoría de los robots autónomos llevan algún tipo de sensor que les permita recoger información de su entorno. Un tipo de sensor muy habitual son los sensores de distancia, ya sean por ultrasonidos o infrarrojos. El modelo del robot Pioneer P3-DX lleva dieciséis sensores de ultrasonidos en los laterales, que le permite medir la distancia. Por otro lado, el Pioneer P3-DX cuenta con una serie de conexiones y articulaciones, que permiten colocar distintos objetos en esas conexiones, para realizar distintas tareas.

El modelo Pioneer P3-DX viene precargado con un script interno (*child* script) en Lua que le permite esquivar obstáculos. Sin embargo, el script incorporado en el modelo de este proyecto era distinto al predeterminado. En este caso el script hace que los datos de los encoders del robot estén disponibles para su uso externo.

Figura 4.2Modelo de Pioneer P3-DX en VREP

4.2.2.2 Escenarios de trabajo

Todos los escenarios empleados en el proyecto estaban compuestos por los mismos elementos, un conjunto de paredes que conforman un camino y el robot Pioneer P3-DX.

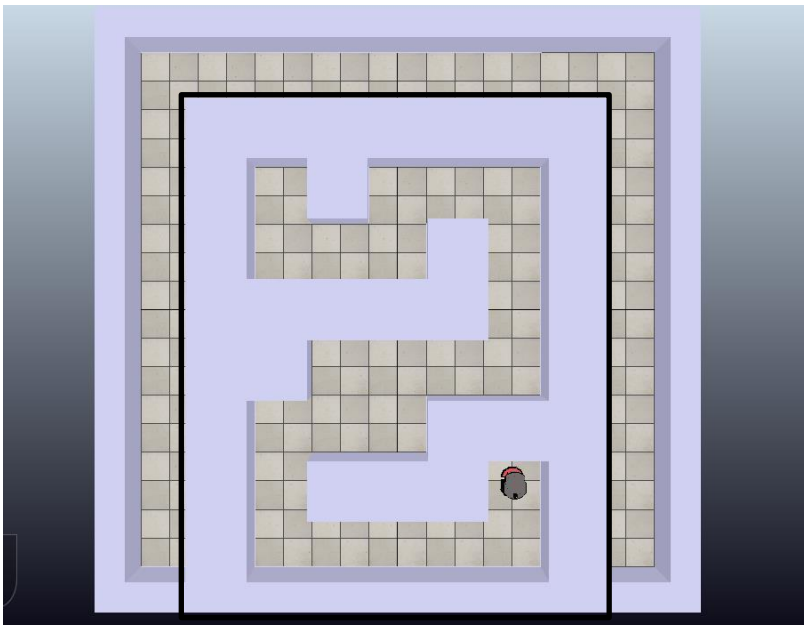


Figura 4.3 Escenario 1 (parte interior)

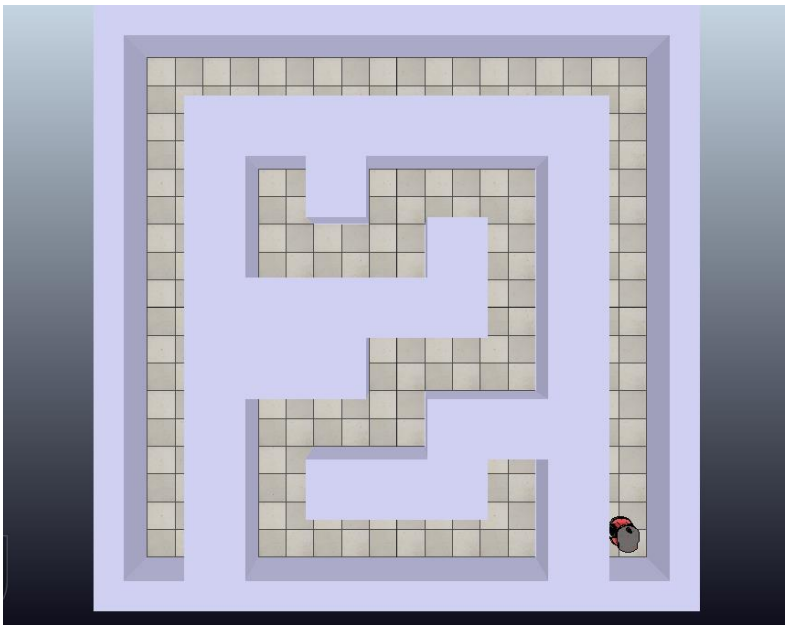


Figura 4.4 Escenario 1 (parte exterior)

Figura 4.5 Escenario 2

4.2.3 COMO SE RELACIONA ROS CON V-REP

Como se ha mencionado anteriormente, el simulador V-REP puede ser controlado mediante un cliente API externo, y este cliente externo se puede tratar de un nodo ROS, que escrito en un lenguaje compatible con las APIs disponibles para V-REP, pueden implementar las funciones para el manejo de VREP de forma sencilla.

En el entorno inicial del proyecto, disponía de dos nodos ROS y dos scripts, escritos en Python; `node_simulator.py`, `robot_p3dx.py`, `node_navigator.py` y `navigation.py`. Estos scripts, los podemos diferenciar en dos partes basándonos en su funcionalidad. Una parte se encarga de conectarse con el simulador, controlar el robot y recoger todos los datos que este puede proporcionar, y la otra parte que se encarga de la navegación del robot.

4.2.3.1 node_simulator y robot_p3dx

robot_p3dx.py se trata de una clase escrita en Python la cual dispone de diversas funciones relacionadas con el robot Pioneer-P3DX. Esta clase se encarga de inicializar el robot (ruedas,

sensores, encoders), de mover el robot mediante una función que recibe velocidad linear y angular, leer los sensores de proximidad de este y los encoders (velocidad de cada rueda).

`node_simulator.py` se trata de un nodo ROS escrito en Python que hace uso de la clase `robot_p3dx.py`. Este nodo esta suscrito al *topic* `/cmd_vel`, y cada vez que recibe datos de ese *topic*, este se encarga de publicar en distintos *topics* los datos de los sensores de ultrasonidos del robot la velocidad del robot en ese momento y de usar la función de mover al robot con la velocidad publicada en `/cmd_vel`, es decir, el robot se mueve de acuerdo con la velocidad que se publica en este *topic*.

4.2.3.2 node_navigator y navigation

`Navigation.py` se trata de una clase en Python que tiene dos funciones, explorar y seguir una pared. Esta clase calcula, mediante los datos proporcionados por la clase `node_navigator.py` (datos de los sensores de ultrasonidos del robot y la velocidad de los encoders), que velocidad (lineal y angular) debe tener el robot en cada momento para seguir una determinada pared, manteniendo una trayectoria recta y siempre a la misma distancia de esta pared. Cuando se encuentra con un obstáculo de frente, evita la colisión y cambia de pared a la que seguir.

`Node_navigator.py` es un nodo ROS escrito en Python el cual está suscrito, a los *topics* que publica `node_simulator.py`, es decir, a los datos de los sensores de ultrasonidos del robot y la velocidad del robot, proporcionada por los encoders; y que, cada vez que recibe datos de cualquier *topic*, utiliza la clase navigation.py para calcular que velocidad debe tener el robot y publicarla en el *topic* `/cmd_vel`.

En la Figura 4.6, se puede observar el grafico generado por rqt_graph donde se puede observar gráficamente lo explicado.

Figura 4.6 rqt_graph (node simulator y node navigator)

Como se ha mencionado a la hora de explicar el funcionamiento de `node_navigator.py`, el robot se mueve cuando se publican datos en el *topic* `/cmd_vel`, es decir, que si tuviésemos un nodo ROS sencillo que solo publicase datos en este *topic*, seriamos capaces de operar manualmente el robot. Por suerte, este nodo existe y se llama *Teleop twist keyboard*.

4.2.3.3 Teleop twist keyboard

Teleop twist keyboard se trata de un nodo ROS, que nos permite mover el robot manualmente, usando el teclado. Este nodo, mediante el uso de ciertas teclas, publica en el *topic* `/cmd_vel` (Figura 4.7), un mensaje de tipo Twist (velocidad linear y angular). Y dado que el robot solo necesita esto para desplazarse, este nodo nos da la posibilidad de operar el robot sin la necesidad de usar el nodo *node_navigator*.

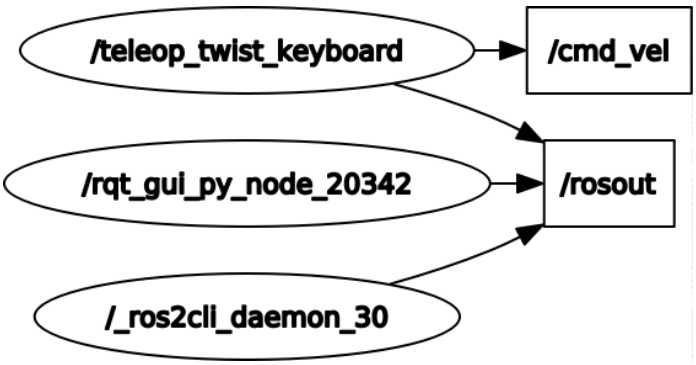


Figura 4.7 Teleop twist keyboard

4.3 IMPLEMENTACIÓN

En este apartado se explicarán todos los pasos que se han seguido para la implementación del algoritmo de SLAM en el proyecto. Podemos resumir la implementación del algoritmo en los siguientes puntos:

- Búsqueda, selección e instalación de algoritmos de SLAM disponibles para ROS2
- Búsqueda e implementación de sensor laser en V-REP
- Extracción de datos laser de V-REP y publicación de los datos provenientes del sensor
- Configuración y solución de errores en SLAM Toolbox
- Posición del robot y marcos de referencia

4.3.1 BÚSQUEDA Y SELECCIÓN DE ALGORITMOS DE SLAM DISPONIBLES PARA ROS2

El primer paso de todos fue la búsqueda y selección de algoritmos de SLAM para ROS2, más concretamente para la versión *Eloquent Elusor*.

Actualmente existen varios paquetes en ROS1 que realizan implementan el algoritmo de SLAM. Los 3 paquetes más usados por la comunidad de ROS, son 3: *gmapping* [22], *cartographer* [26] y *karto* [22]. La implementación de este algoritmo en ROS2 es prácticamente nueva, y se está desarrollando actualmente. Actualmente existen dos proyectos, que implementan el algoritmo de SLAM para ROS2, bastante desarrollados y apoyados por la comunidad de ROS2:

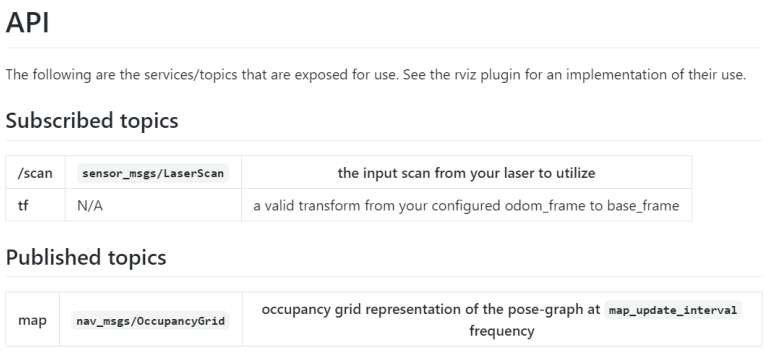
- Cartographer: Tiene mucha tracción comunitaria. Estaba siendo mantenido por Google, pero este ha dejado de trabajar en gran medida en él, aunque algunas empresas se han apoderado de él. Trabaja con LIDARS 2D y 3D, localización y serialización limitada. El problema de este paquete es que nadie estaba interesado en mantenerlo y se dejó de mantener hace unos años.

- SLAM Toolbox: Se trata de un paquete de ROS2, el cual consiste en un conjunto de herramientas y capacidades para SLAM 2D. Se construye sobre la librería open-karto con optimizaciones adicionales e incluye cosas como la serialización completa, la localización, la continuación de la cartografía, y funciona bien para espacios grandes. Steve Macenski, su creador, se comprometió a mantener el paquete y a dar la bienvenida a otros que se lo tomaran en serio para que se unan también.

Según las discusiones en el grupo de trabajo oficial de navegación ROS2, se decidió hacer SLAM Toolbox la nueva implementación predeterminada de SLAM para ROS2. Debido a esto, es este paquete el que se ha implementado en este proyecto.

SLAM Toolbox, no deja de ser un paquete de ROS2, por lo que, como cualquier paquete ROS, se puede resumir su comportamiento mediante *topics*. *Topics* a los cuales este paquete o nodo está suscrito o, por el contrario, *topics* que son publicados por este paquete.

El propio creador de SLAM toolbox, en su página oficial explica cómo, a grandes rasgos, utilizar este paquete. Observando la API del paquete (Figura 4.8), podemos obtener las siguientes conclusiones: SLAM Toolbox está suscrito (recibe) al *topic* con nombre `/scan` del tipo `sensor_msgs/LaserScan` y publica en el *topic* con nombre `/map` del tipo `nav_msgs/OccupancyGrid`. Con otras palabras, podemos decir que recibe mensajes del tipo `LaserScan`, es decir, los datos de un sensor y lo convierte en un mapa. Por otro lado, también podemos ver como necesita una transformada entre `odom_frame` y `base_frame` [27].



API

The following are the services/topics that are exposed for use. See the rviz plugin for an implementation of their use.

Subscribed topics

/scan	sensor_msgs/LaserScan	the input scan from your laser to utilize
tf	N/A	a valid transform from your configured odom_frame to base_frame

Published topics

map	nav_msgs/OccupancyGrid	occupancy grid representation of the pose-graph at map_update_interval frequency

Figura 4.8 API SLAM Toolbox [27]

Entendido lo anterior, una de las tareas principales va a ser publicar datos de un sensor laser, mediante un mensaje del tipo `LaserScan`, de manera que el paquete reciba datos del *topic* al que está suscrito. Aunque el primer paso, es instalar el paquete para poder empezar a trabajar con él. Para ello hay dos opciones:

- La primera es usar el comando *apt install ros-eloquent-slam-toolbox.* Esta opción instala en la máquina que se esté usando el paquete de SLAM Toolbox, permitiéndonos ejecutar el paquete perfectamente y trabajar con él. Sin embargo, al instalar el paquete de esta manera no se puede modificar ningún trozo de código o parámetro, y todos los parámetros del paquete serían los correspondientes a la configuración predeterminada del creador.
- La segunda opción es instalar el paquete desde la fuente. De esta manera se crea un directorio correspondiente a SLAM Toolbox, en tu entorno de trabajo, donde se encuentran todos los archivos que lo componen. Esta opción da la posibilidad de modificar cualquier línea de código que compone el paquete o cualquier parámetro del paquete.

SLAM Toolbox se trata de un paquete que puede ser implementado en infinidad de proyectos, por lo que es un paquete muy versátil. En este, existen ciertos parámetros deben ser configurados de acuerdo con tu proyecto y a las características de los sensores usados, por lo que la segunda opción es por la que se optó en este proyecto.

4.3.2 BÚSQUEDA E IMPLEMENTACIÓN DE SENSOR LASER EN VREP

Tras la instalación del paquete, el siguiente paso es encontrar el sensor laser adecuado para el proyecto. Como se mencionó al principio del capítulo, el simulador VREP dispone de una serie de sensores de manera predeterminada. Entre todas estas opciones, existe un sensor (LIDAR) llamado *Hokuyo URG 04LX UG01* (Figura 4.9). Este sensor se trata de uno de los más utilizados a la hora de implementar SLAM, de manera que en este proyecto se ha optado por usar este sensor.

Figura 4.9 Hokuyo URG-04LX-UG01

En VREP existen tres versiones de este, *Hokuyo URG 04LX UG01*, *Hokuyo URG 04LX UG01_Fast y Hokuyo URG 04LX UG01_ROS*. Este último está diseñado específicamente para su uso en ROS, de manera que en su *child* script implementa ya una función que publica directamente los datos obtenidos por el sensor en un *topic* del tipo `sensor_msgs/LaserScan`, sin necesidad de un cliente remoto. Sin embargo, este solo esta implementado para ROS y no para ROS2 por lo que no tenía utilidad en este proyecto.

Dependiendo del modelo del sensor, las mediciones se realizan de distintas formas. *Hokuyo URG 04LX UG01_Fast* se diferencia del *Hokuyo URG 04LX UG01* en que está basado en sensores de visión y no es geométricamente perfecto. *Hokuyo URG 04LX UG01* está basado en sensores de proximidad y es más preciso que en su versión *Fast*.

Debido a que no se necesitaba tanta precisión, *Hokuyo URG 04LX UG01_Fast* fue la elección fina al tratarse de una versión más rápida, aunque menos precisa. Una vez decidido el sensor que usar, hay que incluirlo en la escena y acoplarlo al robot. Para incluir el sensor en la escena, basta con arrastrarlo a ella. Para acoplarlo al robot Pioneer P3-DX disponemos de una serie de conexiones en su parte superior. Para ello hay que:

1. Seleccionar una de las conexiones del robot
2. Seleccionar el sensor mientras se mantiene pulsado la tecla *ctrl*

3. Presionar el botón `assemble/dissasemble` disponible en la parte superior de la interfaz de V-REP.

Tras esto se obtiene un resultado como el mostrado en la Figura 4.10.

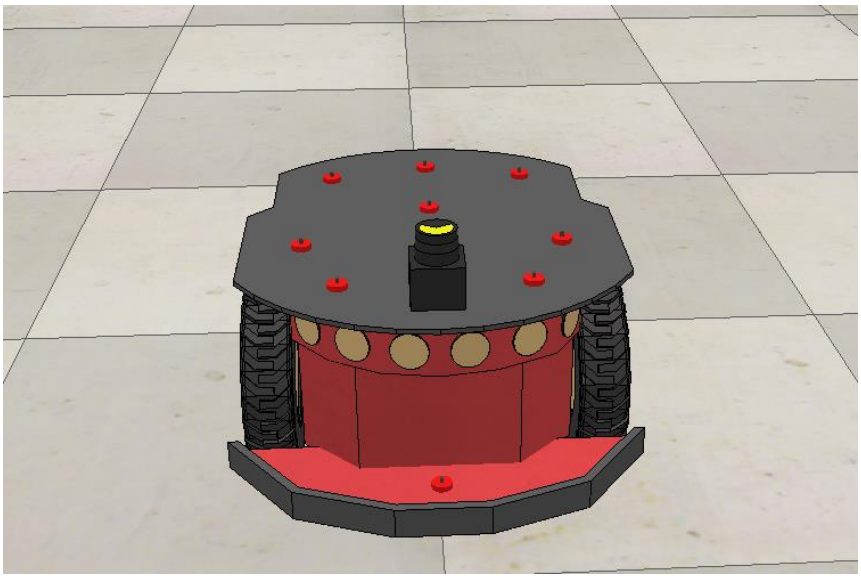


Figura 4.10 Pioneer P3-DX con sensor Hokuyo URG 04LX UG01_Fast acoplado

4.3.3 EXTRACCIÓN DE DATOS LASER DE V-REP Y PUBLICACIÓN DE LOS DATOS EXTRAÍDOS DEL SENSOR

Tras tener el robot y el láser acoplados correctamente el siguiente paso es extraer los datos de ese sensor laser y publicarlos en un *topic*.

Como se mencionó anteriormente, cada objeto del simulador V-REP tiene asociado un script en lenguaje Lua (child script), con el que se puede modificar el comportamiento de estos objetos, o manejar las variables que estos proporcionan.

Todos los objetos tienen un patrón común en estos scripts, en cuanto a funciones se refiere, y suele venir código comentado que incluye alguna funcionalidad de ese objeto. En este patrón de funciones siempre aparecen:

- la función de inicialización: sysCall_init. Esta función de devolución de llamada del sistema no es opcional. Se ejecutará una vez justo al comienzo de una simulación. El código se encarga de preparar una simulación, etc.
- la función de detección: sysCall_sensing. Esta función se ejecutará en cada paso de simulación. El código se encarga de manejar todas las funciones de detección del simulador (sensores de proximidad, detección de colisiones, etc.) de manera genérica.

El sensor laser elegido, *Hokuyo URG 04LX UG01_Fast*, cuenta en su función de detección con código comentado que aportaba la funcionalidad de externalizar mediante la función, `simxSetStringSignal()` perteneciente a la API Python de V-REP, en una variable tipo `string`, los puntos detectados por el sensor laser.

El código predeterminado en el simulador guarda en una variable llamada "*measuredData*" las coordenadas de todos los puntos que han detectado ambos sensores de visión dentro del sensor laser. Sin embargo, para poder publicar los datos del sensor laser necesarios se necesita otro tipo de medida. Esta medida se conoce como rangos y consiste en la distancia a la que esta cada punto detectado del sensor laser, más concretamente, la distancia a la esta cada punto detectado con respecto a laser que lo lanzo. Para conseguir esta variable hay añadir código en la función `sysCall_sensing()` del sensor laser (ANEXO II). Tras añadir ese código ya se dispone de los datos requeridos para poder usarlos externamente.

Una vez conseguido externalizar esos datos, se necesita obtenerlos en los nodos Python para poder trabajar con ellos. Para ello se utilizan las funciones pertenecientes a la API remota de VREP:

```
# 1, datos invalidos : Sirve para inicializar, la primera llamada
errorCode, ranges = vrep.simxGetStringSignal(clientID, 'scan ranges',
vrep.simx_opmode_streaming)
# Datos validos
errorCode, ranges = vrep.simxGetStringSignal(clientID, 'scan ranges',
vrep.simx_opmode_buffer)
```

```
 #Convertir la variable string a una lista de floats donde los valores de la
lista son los valores medidos por el sensor
ranges = vrep.simxUnpackFloats(ranges)
```

En este proyecto, estas funciones se han añadido al `node_simulator`. Es en este nodo donde se va a trabajar principalmente. Este nodo cuenta con una función de inicialización, donde se ha añadido la función `simxGetStringSignal`, en su modo de operación `opmode_streaming`. El resto de las funciones se han añadido más adelante en el script, de manera que cada vez que `node_simulator` recibe un nuevo mensaje en `/cmd_vel`, la variable `ranges` se actualiza. El siguiente paso es publicar estos datos.

EL mensaje que se necesita publicar es un mensaje del tipo `sensor_msg/LaserScan`, cuya estructura, definida en ROS, es:

- `Header header` una subestructura formada por
  - `time stamp` variable temporal que quedará definida mediante la función de ROS ros::time::now()
  - `string frame_id` referencia asignada al mensaje
- `float angle_min`, ángulo en el que comienza a medir el láser. Medido partiendo del eje X y en sentido antihorario. (radianes)
- `float angle_max`, ángulo en el que termina de medir el sensor láser. (radianes)
- `float angle_increment`, distancia angular entre medidas. (radianes)
- `float time_increment`, tiempo entre medidas consecutivas. (segundos)
- `float scan_time`, tiempo que tarda en completarse un barrido completo. (segundos)
- `float range_min`, alcance mínimo de medida. (metros)
- `float range_max`, alcance máximo de medida. (metros)
- `float ranges[]`, vector de medidas captadas por el láser, los valores por encima de `range_max` y por debajo de `range_min` deben ser descartados.

En este punto solo se dispone de la parte de `ranges` del mensaje, como se ha explicado anteriormente, por lo que se necesita completar el resto del mensaje.

El primer paso para completar el mensaje es rellenar la cabecera (`header`) del mensaje. Este se compone de dos partes, el `timestamp` y el `frame_id`.

Para el `timestamp`, se necesita una variable de tipo `Time`, la cual está definida en ROS1. Sin embargo, esta variable no existe en ROS2, por lo que, para obtener algo equivalente al Time de ROS, en ROS2, se necesita llamar a la función `now()` de un nodo. Esto es así ya que ROS2 no dispone de un nodo global con un tiempo único, como hacia su predecesor, ROS1. Por lo que, para usar el Time de ROS, ROS2 necesita tener un nodo activo el cual tendrá su propio tiempo:

```
time_ros = self.get_clock().now().to_msg()
```

El parámetro `frame_id`, se trata del nombre asignado a los datos que se vayan a publicar. Se asigno el nombre de "`base_scan`". En cuanto al resto de parámetros del mensaje, se tratan de características del sensor. Para completar el mensaje, es necesario mirar las especificaciones del sensor laser. Estas especificaciones se pueden encontrar tanto en VREP como en las especificaciones del sensor real. Para ver las especificaciones en VREP, basta con irse a la jerarquía de la escena y hacer doble clic sobre los sensores de visión que componen el sensor *Hokuyo*. En este caso, no todos los datos necesarios están disponibles en el simulador por lo que algunos campos, como la frecuencia del sensor, hay que obtenerlos de las especificaciones del modelo del sensor. El resultado final fue:

```
        msg = LaserScan()

        time_ros = self.get_clock().now()
        msg.header.stamp = time_ros.to_msg()
        msg.header.frame_id = "base_scan"
        msg.angle_max = 2.08621382713
        msg.angle_min = -2.08621382713
        msg.angle_increment = 4.173427654/684.0
        msg.range_max = 5.000
        msg.range_min = 0.009
        msg.scan_time = 0.100
```

```
        msg.time_increment = (1/10)/684.0
        msg.ranges  = ranges
```

Una vez completado todos los campos del mensaje, hay que publicarlo y decidir cada cuánto publicar ese mensaje. Para poder publicar un mensaje en ROS2, primero hay que crear lo que se conoce como un *Publisher*, es decir, inicializar el proceso que va a publicar datos en un *topic*. La función para crear un *Publisher* es:

```
self.publisher_ = self.create_publisher(String, 'topic', 10)
```

`create_publisher` declara que el nodo publica mensajes de tipo *`String`* sobre un tema nombrado *topic*, y que el "tamaño de la cola" es 10. Debido a que SLAM Toolbox, está suscrito al *topic* con nombre `"/scan"` ese debe ser el nombre del *topic* en el cual se publiquen los datos obtenidos. En cuanto a la frecuencia a la debe publicar en el *topic*, se va a publicar cada vez que el robot se desplace, es decir, cada vez que reciba un nuevo mensaje en el *topic* *`/cmd_vel`* .

Para comprobar los resultados se ha usado la herramienta rviz2 y el comando `/topic echo "nombre del topic"`. Este último se trata de una herramienta que muestra en ventana de comandos los datos del *topic* que le indiques, mientras en este se esté publicando algo. Para lanzar rviz2, en la ventana de comandos hay que ejecutar el comando `ros2 run rviz2 rviz2`.

En la Figura 4.13 se pueden observar los resultados de la herramienta `/topic echo`, mientras que en las figuras Figura 4.11 y Figura 4.12 podemos observar la comparativa entre lo que ve el sensor laser en el simulador con los datos que están siendo publicados.

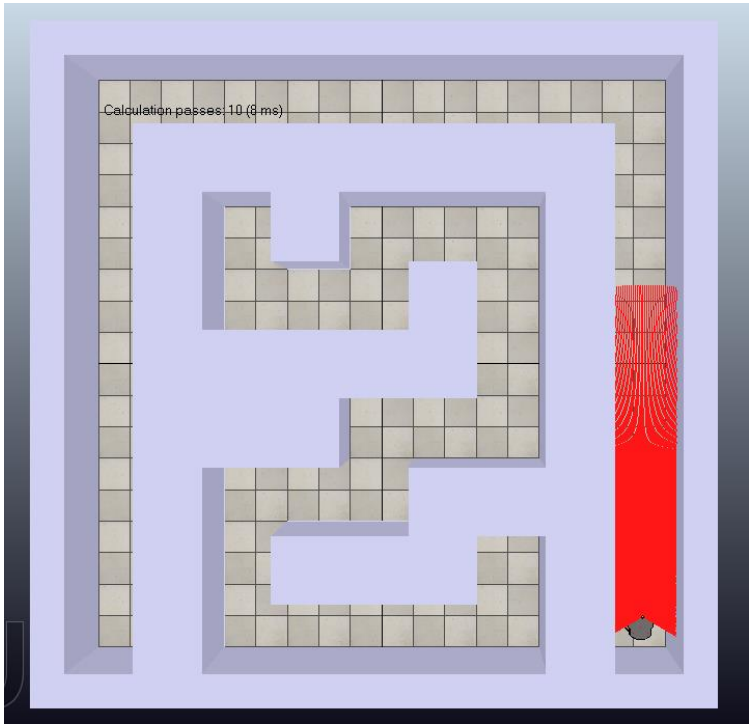


Figura 4.11 Entorno de simulación final funcionando (la zona roja se trata del área de visión del sensor)

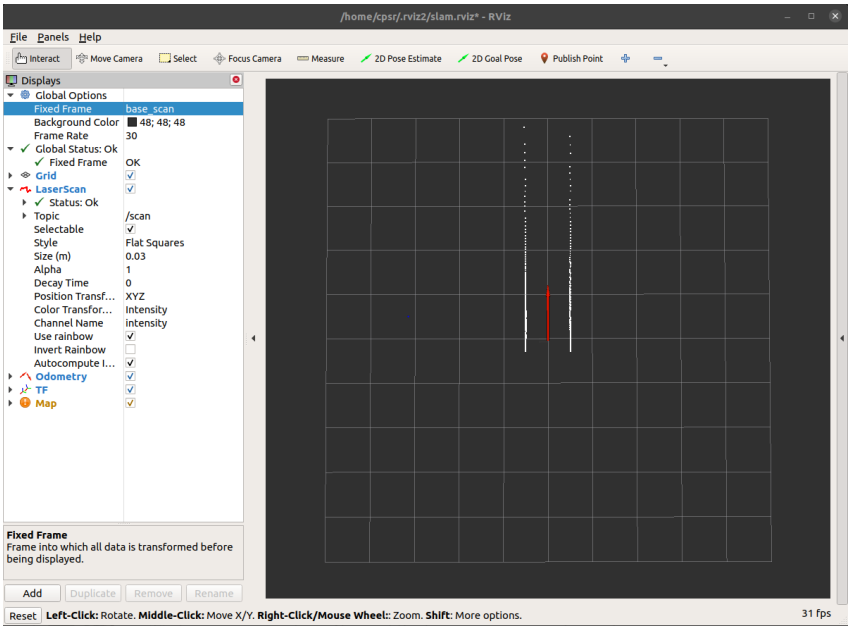


Figura 4.12 RViz mostrando los datos obtenidos por el sensor laser

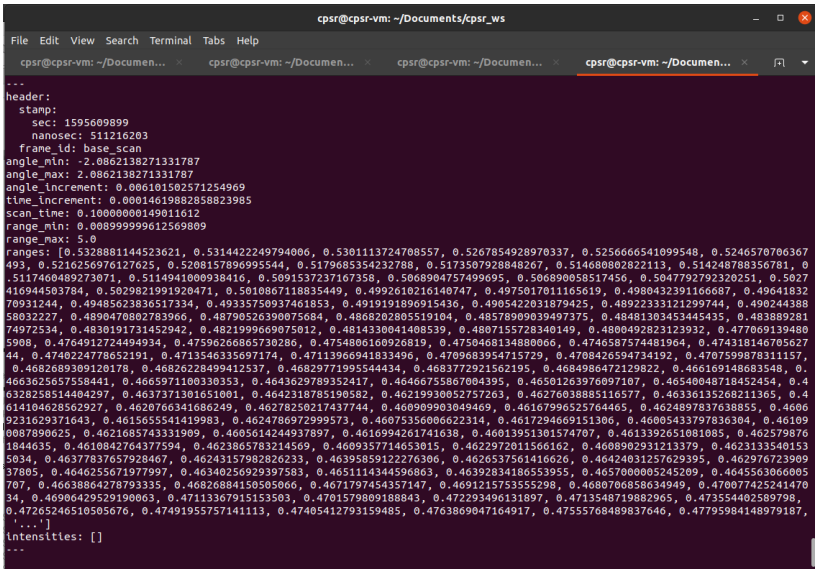


Figura 4.13 Herramienta topic echo mostrando los datos publicados en el topic /scan

4.3.4 CONFIGURACIÓN Y SOLUCIÓN DE ERRORES EN SLAM TOOLBOX

Tras publicar los datos, se ejecutó SLAM Toolbox y comprobar si ocurría algún error con los datos que se estaban publicando. Primero hay que explicar que SLAM Toolbox proporciona varios modos de mapeo:

- Síncrono: Se procesan todas las medidas. Puede causar lag en espacios grandes si la capacidad de cómputo es pequeña. Su opción offline proporciona la mayor calidad. Este modo es el que se aproxima más al mapeo en tiempo real.
- Asíncrono: Procesa las medidas basándose en la filosofía del mayor esfuerzo, es decir, intenta no quedarse atrás al procesar nuevas medidas, pero no las va acumulando. Es la mejor opción para lugares grandes y es la recomendada para mapear mientras navegas. Es la opción online.

Hay que mencionar que cada modo de mapeo tiene asociado un archivo de configuración con distintos valores. Estos archivos de configuración se encuentran en el directorio *slam_toolbox/config*. Dependiendo de la opción de mapeo que se elija, habrá que modificar estos valores para adaptarlos al proyecto. En este proyecto, se han utilizado dos modos de mapeo, el offline y el asíncrono online. Para las primeras pruebas, se utilizó el modo offline.

Para ejecutar cualquier modo de SLAM Toolbox se utiliza el comando `ros2 launch slam_toolbox online_async_launch.py` (modo asíncrono online). Si se quiere ejecutar otro modo solo hay que cambiar el nombre del archivo que se va a ejecutar, los cuales se encuentran en el directorio *slam_toolbox/launch.* Al ejecutar por primera vez SLAM Toolbox apareció el siguiente mensaje, el cual se trata de un error (Figura 4.14):

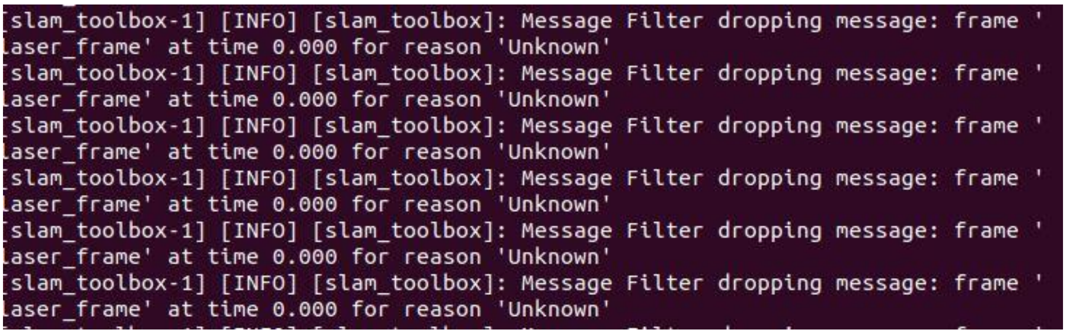


Figura 4.14 Error filtro SLAM Toolbox

Este error proviene de que el nodo SLAM Toolbox no está recibiendo las transformadas (tf) necesarias, del marco del láser al marco del mapa. Por lo que, para solucionarlo, se necesita implementar las transformadas. Como se mencionó en 2.4.3, en robótica existen diversos marcos de referencia o coordenadas. En este caso, los algoritmos de SLAM, tienen una estructura definida, la cual se puede organizar como un árbol (Figura 4.15).

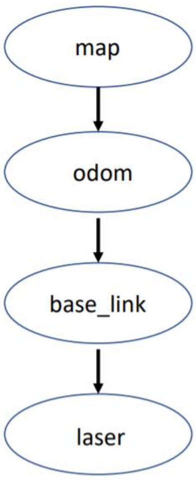


Figura 4.15 Árbol de transformadas para SLAM

- *Map*: El origen de este marco es arbitrario, por lo que se puede elegir donde se origina. En nuestro caso, el propio paquete se encarga de generarlo.
- *Odom*: Este se origina en la posición donde el robot empieza. Este marco esta fijo, no se mueve con el robot
- *Base_link*: Se define como el centro del robot. Este se mueve con el robot (no es fijo)
- *Laser*: Se trata de la posición del láser en el entorno. Este suele moverse con el robot, es decir con *base_link*, y puede estar situado justo en el mismo marco (si el láser esta justo en el centro del robot) o ligeramente desplazado, dependiendo de donde se encuentre el sensor laser con respecto al robot.

Para que estos marcos de coordenadas tengan sentido entre ellos, es necesario ir transformando las coordenadas de uno marco a otro para que se sepa, por ejemplo, donde está el robot con respecto al marco de odometría o con respecto al marco de coordenadas del mapa. SLAM Toolbox, como el creador indica, proporciona la transformadas entre el marco del mapa (*map*) y el de odometría (*odom*), de manera que el resto de transformadas se tienen que implementar de forma externa.

Para solucionar este error de forma parcial, se decide proporcionar transformadas de forma externa, desde una nueva ventana de comandos, y sin haber implementado la odometría, para comprobar si había algún error más con los mensajes publicados del sensor laser.

Los siguientes errores que han aparecido estaban relacionados con el archivo de configuración del nodo que se estaba ejecutando (modo offline). El archivo de configuración de SLAM Toolbox juega un papel esencial a la hora de poder llevar a cabo el algoritmo de forma óptima. El archivo de configuración ofrece diversas opciones que puedes modificar y adaptarlas al proyecto que está llevando a cabo. En cuanto a este proyecto respecta, los parámetros de configuración, comunes a todos los nodos, que se han modificado son:

- `base_frame`: Nombre del marco de referencia de la base del robot. Modificado a "`base_link`"

- `max_laser_range`: Maxima distancia medida por el láser. Modificado a 5
- `minimum_travel_distance`: Distancia mínima de viaje antes de procesar un nuevo escaneo. Modificada a 0.3
- `minimum_travel_heading`: Cambio mínimo de rumbo para justificar una actualización. Modificada a 0.3
- `map_update_interval`: Intervalo para actualizar el mapa de ocupación 2D para otras aplicaciones / visualización. Para el modo offline se modificó a 1.0 y para el modo online_async a 5.0.

Por último, el ultimo error que el nodo SLAM Toolbox mostro por ventana de comandos contiene el siguiente mensaje:

```
LaserRangeScan contains 685 range readings, expected 684
```

Este error se trata de un error común en el paquete, según su creador, y se produce debido a que en el archivo *slam_toolbox/lib/karto_sdk/include/karto_sdk/Karto.h* en la función `Update()`, al obtener el número de lecturas de rangos (`NumberOfRangeReadings`), es decir el número total de lecturas que obtiene el sensor laser, se suma una variable residual con valor 1.

```
void Update()
  {
    int residual = 1;
    if (GetIs360Laser()) {
      // residual is 0 by 360 lidar conventions
      residual = 0;
    }
    m_NumberOfRangeReadings =
static_cast<kt_int32u>(math::Round((GetMaximumAngle() -
      GetMinimumAngle()) /
      GetAngularResolution()) + residual);
  }
```

Para solucionar este problema se debe quitar esa variable *"residual"* de manera que la variable final `NumberOfRangeReadings` se reduzca una unidad o en su defecto, restar a la variable `NumberOfRangeReadings` una unidad. Se ha optado por la segunda opción:

```
  void Update()
  {
    int residual = 1;
    if (GetIs360Laser()) {
      // residual is 0 by 360 lidar conventions
      residual = 0;
    }
    m_NumberOfRangeReadings =
static_cast<kt_int32u>(math::Round((GetMaximumAngle() -
      GetMinimumAngle()) /
      GetAngularResolution()) + residual - 1);
  }
```

Para finalizar con este error, hay que comentar que el mensaje de error estaba generado por la siguiente función en el archivo *slam_toolbox/lib/karto_sdk/src/Karto.cpp*, más concretamente en la línea 209:

```
kt_bool LaserRangeFinder::Validate(SensorData * pSensorData)
{
  LaserRangeScan * pLaserRangeScan = dynamic_cast<LaserRangeScan *>(pSensorData);

  // verify number of range readings in LaserRangeScan matches the number of
expected range readings
  if (pLaserRangeScan->GetNumberOfRangeReadings() != GetNumberOfRangeReadings())
{
    std::cout << "LaserRangeScan contains " << pLaserRangeScan-
>GetNumberOfRangeReadings() <<
      " range readings, expected " << GetNumberOfRangeReadings() << std::endl;
    return false;
  }
```

Después de corregir el error previo se ha ejecutado de nuevo SLAM Toolbox. Para comprobar que el algoritmo estaba funcionando y se estaba generando algún mapa, se ha usado de herramienta rviz2, de manera que se pueda visualizar el *topic* `/map`, sobre el cual, SLAM Toolbox publica datos del tipo `nav_msgs/OccupancyGrid`.

En la Figura 4.16 se pueden ver estos resultados, los cuales se corresponden a los primeros datos extraídos por el sensor laser (Figura 4.11).

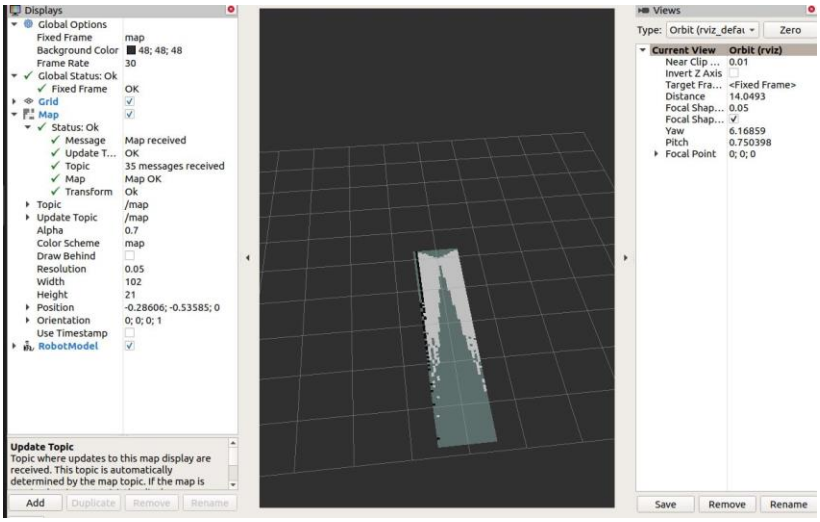


Figura 4.16 Primer fragmento de mapa generado por SLAM Toolbox

4.3.5 POSICIÓN DEL ROBOT Y MARCOS DE REFERENCIA

Pese a que el algoritmo estaba generando un mapa, este mapa no se actualiza ni ampliaba por mucho que el robot continuase. Debido a que el propio nodo SLAM Toolbox, no tiene ninguna noción de que el movimiento del robot, por mucho que recibiese nuevas lecturas del sensor, el mapa no se actualizaba. Por lo que el siguiente es implementar la odometría del robot.

A pesar de que este algoritmo de SLAM no está suscrito a ningún *topic* de odometría, necesita aun así conocer la odometría del robot, es decir, como se ha movido lineal y angularmente el robot con respecto a su posición inicial, para conocer su posición a medida que avanza.

Para calcular la odometría del robot, dos cosas son esenciales. La posición del robot y la velocidad (lineal y angular) de este. A partir de la posición inicial y la velocidad que tiene el robot en cada momento de su recorrido, se calcula la posición relativa del robot.

```
double dt = (current_time - last_time).toSec();
double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
double delta_th = vth * dt;

x += delta_x;
y += delta_y;
th += delta_th;
```

La velocidad que tiene el robot en cada momento la tenemos gracias al nodo `robot_p3dx`, que nos permite obtener la velocidad que tiene el robot en el momento que se desee.

Para implementar toda la odometría, ROS proporciona ejemplos básicos de cómo hacerlo. Estos ejemplos se tienen que adaptar a ROS2. En este proceso, para que sistema pueda trabajar con componentes 2D y 3D de forma conjunta, se suele crear un cuaternion (extensión de los números reales), y para ello se usa la función llamada `quaternion from Euler`. Esta función viene incluida en una librería de ROS1, sin embargo, para ROS2, esa librería no está implementada. Para poder implementar esta función en ROS2, se tiene que buscar en una librería externa de Python, de manera que se incluya esa librería en el script. Debido a que solo se tenía que utilizar esta función de su librería se ha decidido añadir esa función en el script del nodo `node_simulator` y adaptarla a este. La función final está en incluida en el ANEXO II.

Solucionado este problema, se ha creado un nuevo Publisher para el *topic* de odometria. Para comprobar que se estaban publicando datos en este *topic* se ha comprobado de la misma forma que se comprobó el *topic* "`/scan`", del sensor laser, usando el comando `/topic echo`.

Para comprobar que la odometría estaba implementada correctamente falta implementar las transformadas entre los marcos de referencia dentro del script. La transformada entre el marco del láser y el marco de la base del robot se trata de una transformada simple, ya que ambos marcos de referencia están unidos, esta se ha incluido en la función que publica el

mensaje "`/scan`", de manera que cada vez que se publica un nuevo mensaje (el robot avanza), se manda una transformada entre el láser y la base del robot:

```
t = TransformStamped()
        t.header.stamp = time_ros.to_msg()
        t.header.frame_id = "base_link"
        t.child_frame_id = "base_scan"
        t.transform.translation.x = 0.05
        t.transform.translation.y = 0.0
        t.transform.translation.z = 0.0
        t.transform.rotation.x = 0.0
        t.transform.rotation.y = 0.0
        t.transform.rotation.z = 0.0
        t.transform.rotation.w = 1.0

        self._base_broadcaster.sendTransform(t)
```

Para la transformada entre la base del robot y el marco de odometría, se necesitan los parámetros que se han usado para la odometría, ya que el marco de odometría esta fijo mientras que el de la base del robot se mueve con el robot. Esta transformada se ha incluido dentro de la función que se encarga de calcular la odometría y de publicarla:

```
time_now = self.get_clock().now().to_msg()
        # first, we'll publish the transform over tf
        t = TransformStamped()
        t.header.stamp = time_now
        t.header.frame_id = "odom"
        t.child_frame_id = "base_link"
        t.transform.translation.x = self._x
        t.transform.translation.y = self._y
        t.transform.translation.z = 0.0
        t.transform.rotation.x = odom_quat[0]
        t.transform.rotation.y = odom_quat[1]
        t.transform.rotation.z = odom_quat[2]
        t.transform.rotation.w = odom_quat[3]

        self._odom_broadcaster.sendTransform(t)
```

Para comprobar que la odometría y las transformadas están bien implementadas se ha vuelto a ejecutar el nodo SLAM Toolbox. Esta vez, los resultados han sido los esperados. En la Figura 4.17 podemos observar una trayectoria en rojo, que se trata del movimiento del

robot en el escenario 1 (Figura 4.4), la cual se corresponde a la trayectoria que el robot estaba siguiendo en el simulador.

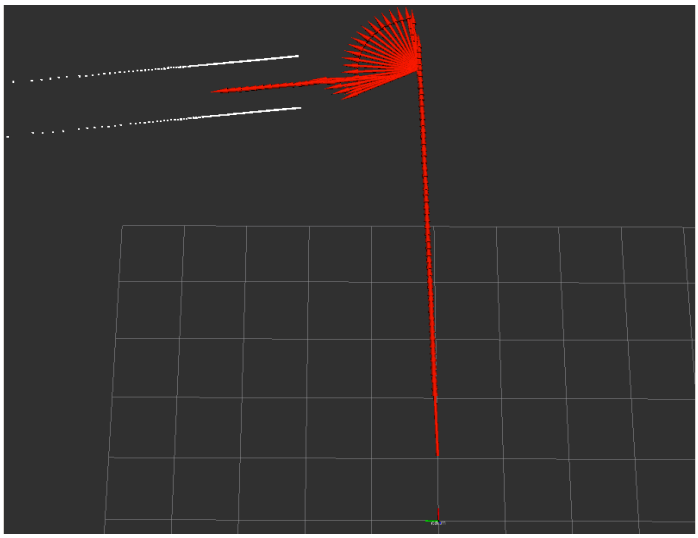


Figura 4.17 Odometria correcta en RViz

Por último, tanto ROS2 como el propio paquete de SLAM Toolbox cuentan con herramientas que permiten guardar el mapa que se ha generado en un formato determinado para que puedan ser retomados en algún momento, se utilicen para navegación o para que sean usados por cualquier usuario. La versión de SLAM Toolbox, es una versión más simple de la de ROS2. Para guardar estos mapas se utiliza el comando `ros2 run nav2_map_server map_saver`.

Para concluir con este capítulo, en la Figura 4.18, se muestra la estructura final de los nodos (node_simulator, node_navigator y SLAM Toolbox, rviz) y *topics* mediante la herramienta rqt_graph.

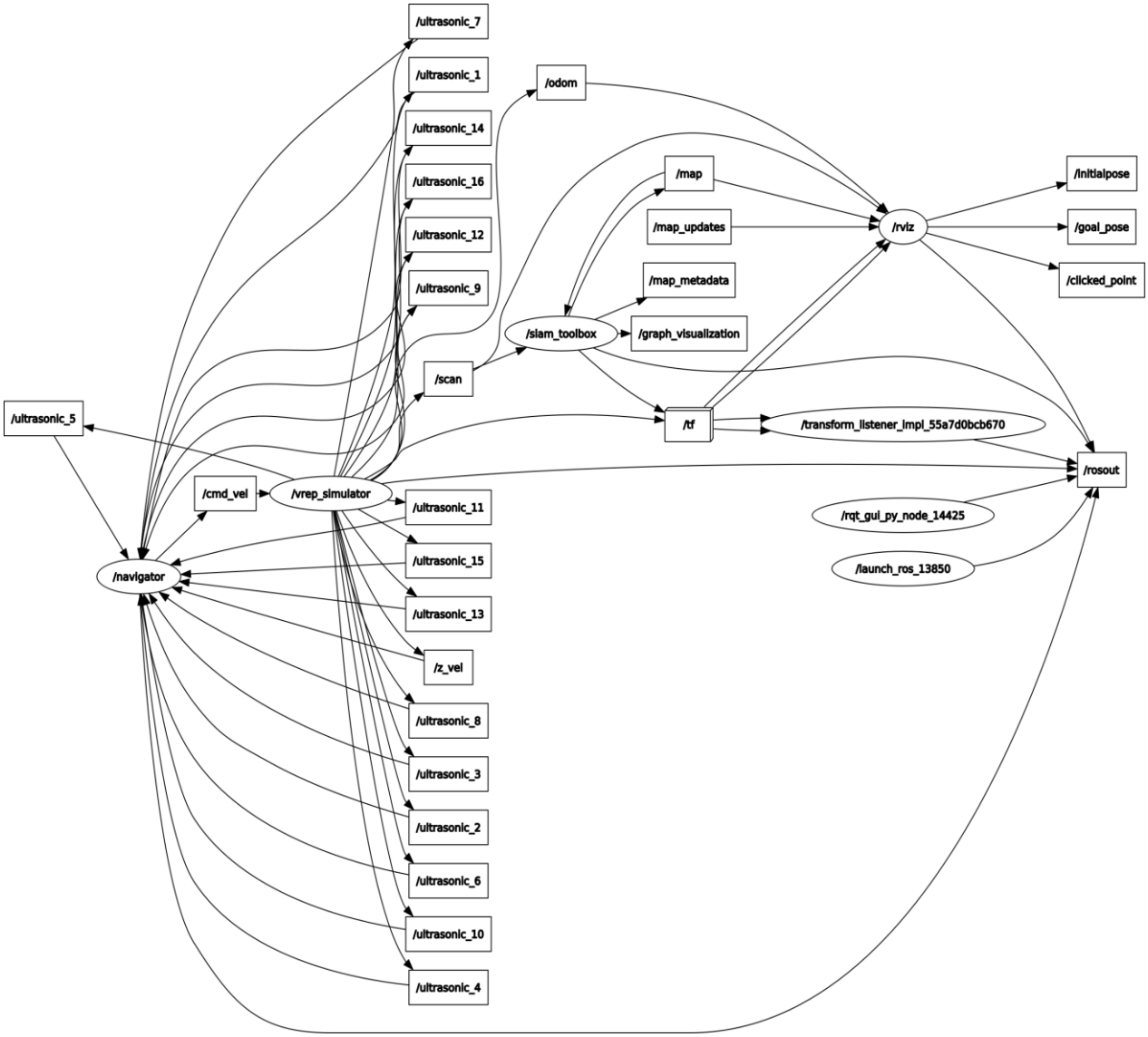


Figura 4.18 rqt_graph final

Capítulo 5. ANÁLISIS DE RESULTADOS

En este capítulo se analizarán los resultados obtenidos después de implementar SLAM Toolbox

El primer escenario donde se trabajó con el algoritmo de SLAM implementado, mediante el paquete SLAM Toolbox fue el escenario 1 (Figura 4.4). En la Figura 5.1 se puede observar, el mapa que está siendo generado, mientras el robot avanza por el escenario. Las líneas rojas en la Figura 5.1 representan la trayectoria que ha ido siguiendo el robot y las líneas blancas se tratan de los datos que está observando el sensor laser.

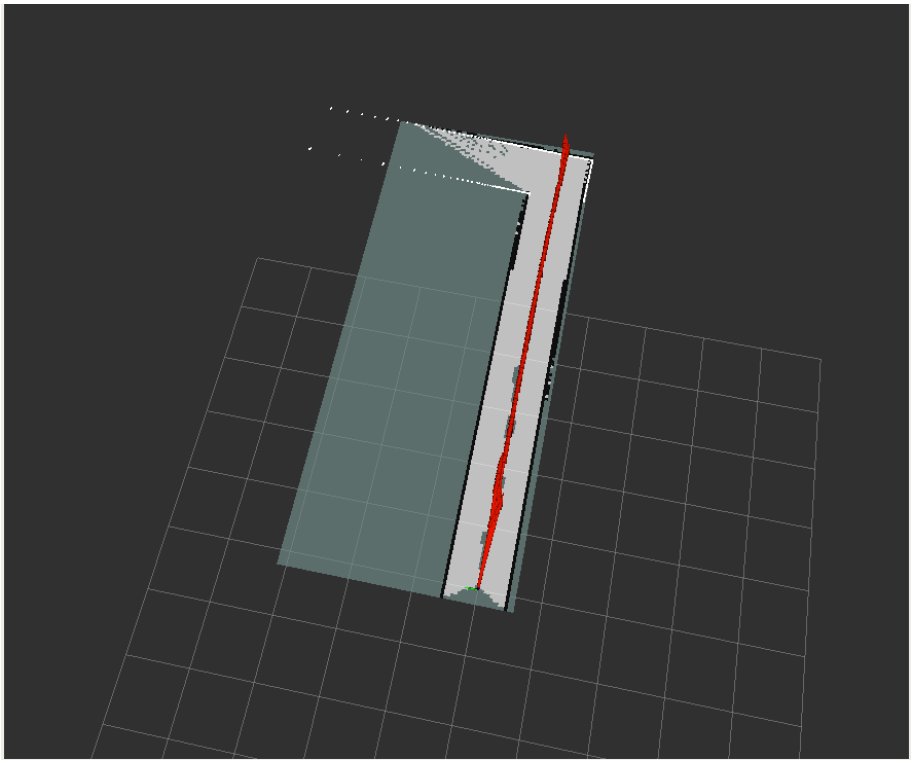


Figura 5.1 Progreso de mapeado del primer entorno

Como se puede observar en la Figura 5.2 los resultados del primer escenario son satisfactorios. Mencionar el escenario 1, tanto la parte externa como la interna, han sido mapeados mediante el uso del nodo, `node_navigator`. Al tratarse de escenarios en los

cuales el espacio para explorar eran pasillos estrechos, era complicado operar manualmente el robot ya que a veces chocaba con una pared, lo que hacía que el robot girase, provocando que se perdiese la referencia, y la odometría del robot fallase. Es por eso por lo que se usó el nodo `node_navigator` para estos escenarios, ya que evita que el robot choque.

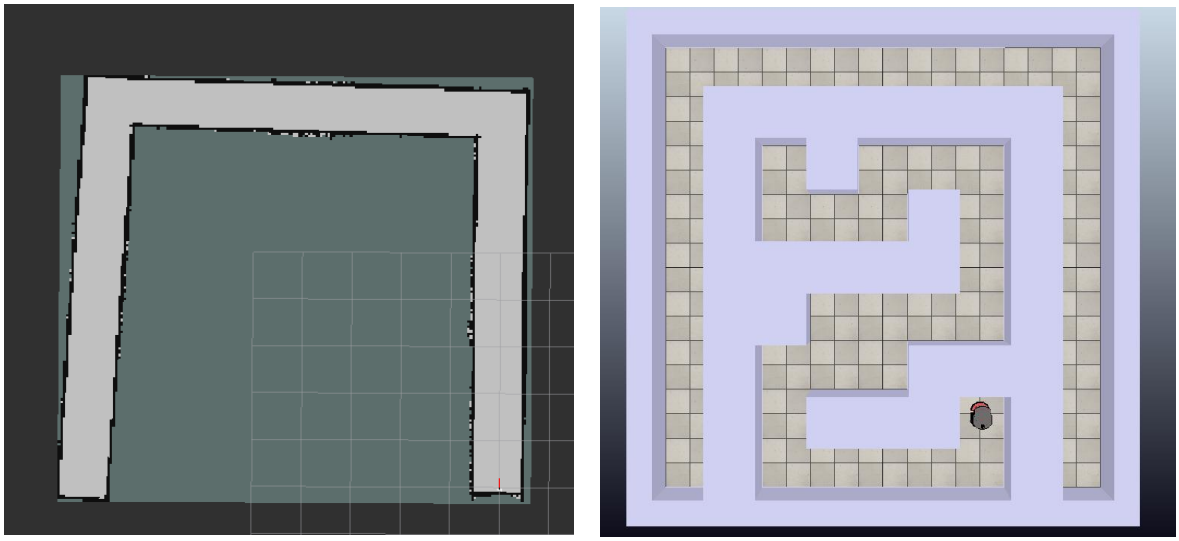


Figura 5.2 Mapa del escenario 1(parte exterior)

Para este primer escenario, se usó el modo de mapeo offline. Para los siguientes escenarios, se decidió probar las dos modalidades de mapeo. En la Figura 5.3 se puede observar el resultado del mapeo con modo offline, mientras que en la Figura 5.4 se muestra el mapeo con la opción asíncrona online. En este ultima se puede observar que durante el proceso hubo algún fallo con la odometría y se desoriento el robot.

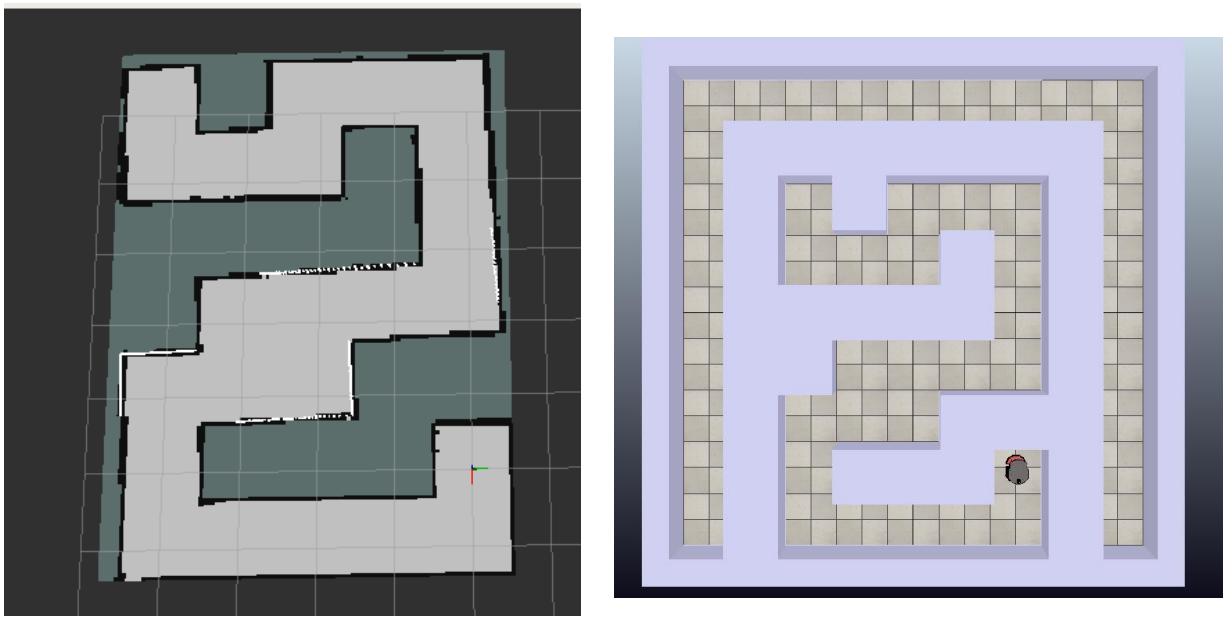


Figura 5.3 Escenario 1 (parte interior) modo offline

Figura 5.4 Escenario 1(parte interior) modo online asíncrono

Para el ultimo escenario, escenario 2 (Figura 4.5) sí que se decidió operar manualmente el robot. Este escenario tenía más espacios abiertos, o al menos más caminos disponibles. En cuanto al modo de mapeo, en la Figura 5.5, se puede observar el mapeo de este escenario en modo offline. Como se explicó en 4.3.4, al tratarse de un escenario más grande y con más espacios, el modo offline no funciona del todo bien ya que la capacidad computacional de la máquina virtual donde se está ejecutando no es muy grande; lo que significa que con este modo no se es capaz de mapear todo el entorno

En la Figura 5.6, se pueden observar los resultados obtenidos con el modo online.

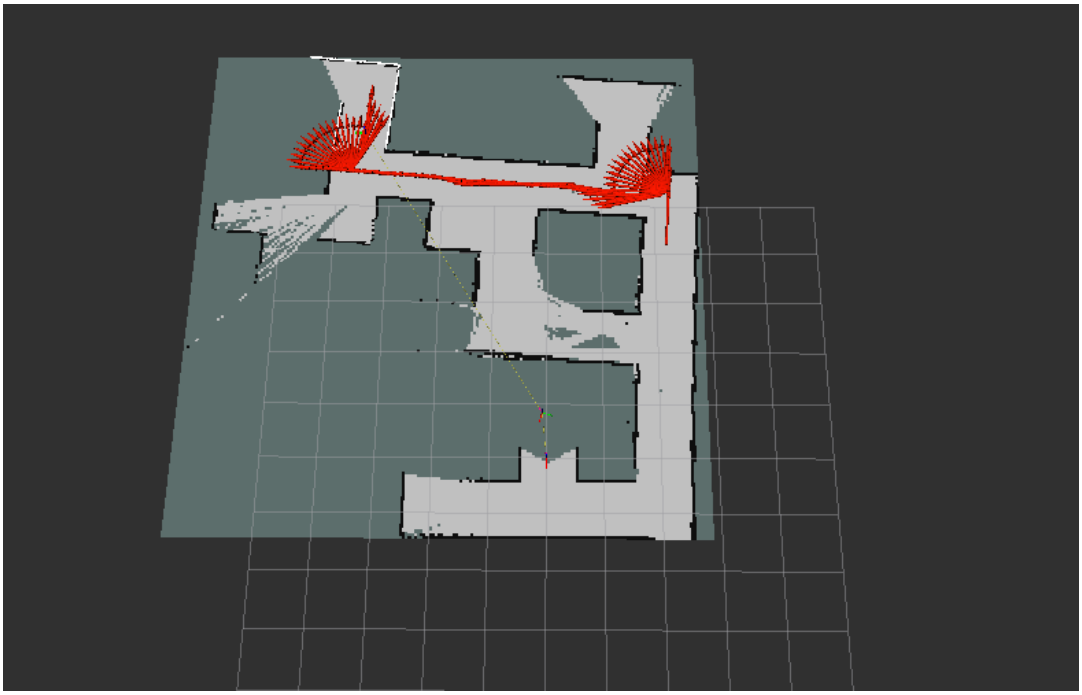


Figura 5.5 Mapeo del escenario 2 modo offline

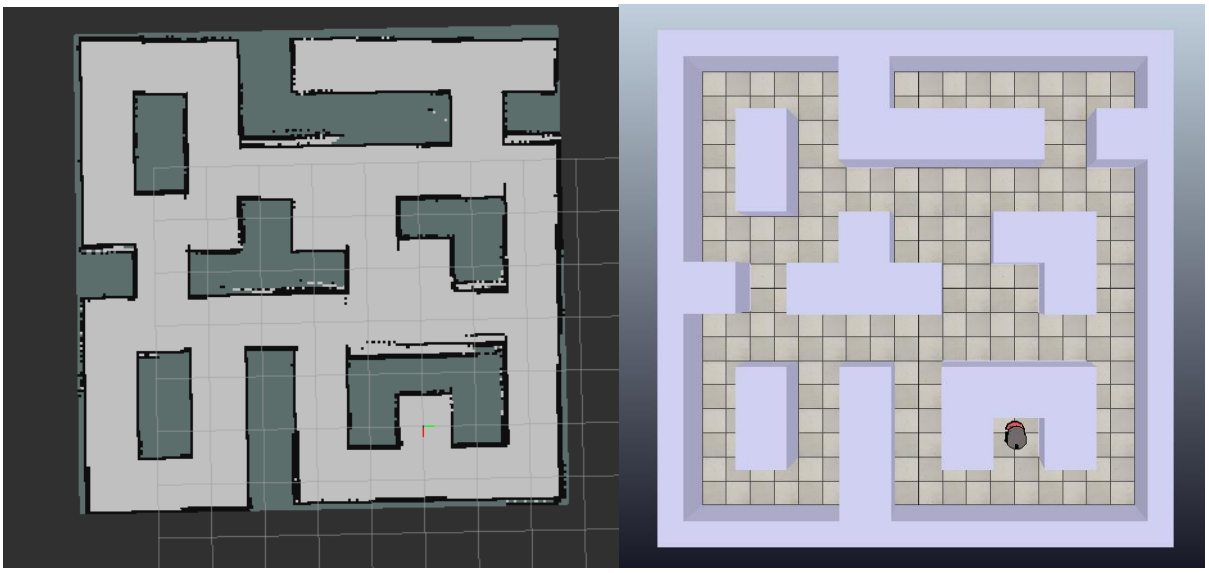


Figura 5.6 Mapa escenario 2 modo online

Como se puede observar, los mapas obtenidos al mapear todos los escenarios son satisfactorios, por lo que podemos concluir que la implementación del algoritmo en este proyecto ha sido satisfactoria.

Capítulo 6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1 CONCLUSIONES

La finalidad de este proyecto era conseguir que un robot móvil sea capaz de construir un mapa de un entorno desconocido a medida que lo explora, mediante un algoritmo de localización y mapeado simultaneo (SLAM). Para ello se han comparado distintos algoritmos SLAM disponibles en ROS 2, y se ha integrado el más adecuado.

El proyecto se ha desarrollado e integrado con el entorno de simulación V-REP, donde se han simulado 3 entornos realistas. Dos de estos tres entornos realistas se trataban de entornos sencillos, mientras que el ultimo se trataba de un entorno más complejo. Para mapear estos entornos se han utilizado dos modos de mapeo disponibles, el modo offline y el modo online asíncrono. Para los entornos pequeños, ambos modos han dado los resultados esperados, sin embargo, para el ultimo entorno (más complejo) el modo offline ha dado problemas ya que este modo, al estar navegando mientras se mapea el entorno, está limitado a espacios pequeños. El modo online asíncrono ha proporcionado resultados satisfactorios sobre el ultimo entorno.

Como usuario que ha tenido que aprender ROS desde cero de forma autodidacta, considero que faltan recursos en internet para la gente que ha estado o estará en mi situación, ya que la mayoría de las cuestiones que se encuentran, consideran que el usuario parte de un punto avanzado en ROS.

También mencionar los conocimientos adquiridos desde la fase previa de documentación sobre el proyecto y los algoritmos disponibles, como durante la realización del trabajo, Conocimientos en programación en Python, lenguaje que conocía, pero no estaba muy familiarizado con él; aprender el lenguaje Lua, manejo del simulador VREP, trabajo con sensores laser, y haber aprendido tanto ROS como ROS2.

Por último, hay que comentar que quedan pendientes trabajos futuros tanto para mejorar el sistema, continuar con el desarrollo de este proyecto o para utilizar sus resultados o su funcionalidad en futuros proyectos.

6.2 TRABAJOS FUTUROS

6.2.1 OPTIMIZAR LOS PARÁMETROS DE CONFIGURACIÓN DE SLAM TOOLBOX, PARA SU ÓPTIMA UTILIZACIÓN.

Una de las principales características a la hora de utilizar SLAM Toolbox son sus parámetros de configuración. Estos juegan un papel esencial a la hora de optimizar los resultados y la creación de los mapas. Por lo que una futura investigación en cómo utilizar cada uno de estos parámetros acorde al modelo de robot y laser que se está utilizando resultaría en un sistema que carecería de fallos y cuyo resultado sería el más optimo

6.2.2 IMPLEMENTAR OTRAS FUNCIONALIDADES AL PROYECTO UTILIZANDO PAQUETES DE ROS2

El *framework* ROS, cuenta con una gran cantidad de paquetes y librerías usados para el desarrollo de múltiples aplicaciones de robótica móvil. Desde navegación con mapa, detección de obstáculos, etc. Por lo que ya establecida una base como es este proyecto sería idóneo implementar más aplicaciones en este.

6.2.3 PATH PLANNING Y SLAM

Una de las aplicaciones más comunes donde se requiere el mapa de un entorno es la planificación de la ruta. La tarea de la planificación de la ruta para el robot móvil es determinar la secuencia de maniobras que debe realizar el robot para moverse desde el punto de partida hasta el destino evitando la colisión con obstáculos.

Por lo que, al tener ya un mapa disponible, es posible llevar a cabo la planificación de ruta, consiguiendo que el robot se mueva de un punto a otro de manera autónoma.

6.2.4 ALGORITMO DE CONTROL Y SLAM

Acorde con el punto, anterior, para conseguir que el robot llegue a un punto de destino evitando objetos, sería necesario hacer uso del sistema de mapeo y localización simultaneo con un algoritmo de control que consiga detectar, en tiempo real, un obstáculo y evitarlo; de forma que recalcule la ruta optima, evitando ese obstáculo.

Capítulo 7. BIBLIOGRAFÍA

[1] «Industria 4.0, la cuarta revolución industrial y la inteligencia operacional,» 20 Abril 2020. [En línea]. Available: https://www.cic.es/industria-40-revolucion-industrial/.

[2] «La robótica avanza gracias a la tecnología de mapeo SLAM,» 29 Agosto 2019. [En línea]. Available: https://www.ittrends.es/infraestructura/2019/08/la-robotica-avanza-gracias-a-la-tecnologia-de-mapeo-slam.

[3] «Mundo Seat,» 7 Febreo 2020. [En línea]. Available: https://mundoseat.seat.com/mediacenter_netstor/seat-media-center/Img/2020/02/2020-02-07/SEAT-incorpora-robots-autonomos-para-automatizar-el-transporte-exterior-de-piezas.pdf.

[4] «Wikipedia,» 12 Mayo 2020. [En línea]. Available: https://es.wikipedia.org/wiki/Sistema_Operativo_Rob%C3%B3tico.

[5] «geeksroom,» 2013 Agosto 21. [En línea]. Available: https://geeksroom.com/2013/08/ros-el-estandar-de-facto-de-la-industria-de-los-robots/78002/.

[6] «GenerationROBOTS,» 26 Marzo 2016. [En línea]. Available: https://www.generationrobots.com/blog/en/ros-robot-operating-system-2/.

[7] «ROS Distributions,» 2020. [En línea]. Available: http://wiki.ros.org/Distributions.

[8] «Geek Gasteiz,» [En línea]. Available: https://geekgasteiz.wordpress.com/2018/11/01/ros2-vs-ros-1-migramos/.

[9] «ROS Mesages,» [En línea]. Available: http://wiki.ros.org/Messages.

[10] «robohub,» [En línea]. Available: https://robohub.org/ros-101-intro-to-the-robot-operating-system/.

[11] «ROS.org,» [En línea]. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics.

[12] « MathWorks -Access the tf Transformation Tree in ROS,» [En línea]. Available: https://es.mathworks.com/help//ros/ug/access-the-tf-transformation-tree-in-ros.html.

[13] «ROS tf2,» [En línea]. Available: http://wiki.ros.org/tf2.

[14] «MIT Technology review - La aspiradora Roomba ahora ve tu casa y hace un mapa de ella,» 18 Septiembre 2015. [En línea]. Available: https://www.technologyreview.es/s/5188/la-aspiradora-roomba-ahora-ve-tu-casa-y-hace-un-mapa-de-ella.

[15] «El coche autónomo de Google (Waymo) se vuelve completamente autónomo y por primera vez sale a la calle sin conductor,» Xataka, 8 Noviembre 2017. [En línea]. Available: xataka.com/automovil/el-coche-autonomo-de-google-waymo-se-vuelve-completamente-autonomo-y-por-primera-vez-sale-a-la-calle-sin-conductor.

[16] «DARPA,» [En línea]. Available: https://www.darpa.mil/.

[17] I. S. Alonso, Septiembre 2018. [En línea]. Available: http://www.geintra-uah.org/system/files/private/TFC-Ines.pdf.

[18] J. B. Martín-Larrauri, Indoor topological SLAM using frontal computer vission, Madrid, 2014.

[19] B. Yamauchi, Alan Schultz y W. Adams, «"Mobile robot exploration and map-building with continuous localization,» 20 Mayo 1998. [En línea]. Available: https://static.aminer.org/pdf/PDF/000/355/760/mobile_robot_exploration_and_map_building_with_continuous_localization.pdf.

[20] F. Andrade y M. Llofriu, «SLAM Estado del arte,» [En línea]. Available: https://www.fing.edu.uy/inco/grupos/mina/pGrado/pgSLAM/documentos/eda.pdf.

[21] S. Riisgaard y M. R. Blas, «A Tutorial Approach to Simultaneous Localization and Mapping,» [En línea]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/119149/16-412j-spring-2005/contents/projects/1aslam_blas_repo.pdf.

[22] J. M. Santos, D. Portugal y R. P. Rocha, «An Evaluation of 2D SLAM Techniques Available,» 2013. [En línea].

[23] M. Montemerlo, S. Thrun, D. Koller y B. Wegbrei, «FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem,» 2002. [En línea]. Available: http://robots.stanford.edu/papers/montemerlo.fastslam-tr.pdf.

[24] S. Thrun y M. Montemerlo, «The GraphSLAM Algorithm With Applications to Large-Scale Mapping of Urban Structures,» [En línea]. Available: http://robots.stanford.edu/papers/thrun.graphslam.pdf.

[25] «CopeliaSim,» [En línea]. Available: https://www.coppeliarobotics.com/helpFiles/index.html.

[26] T. C. Authors, «Cartographer ROS Integration,» 2018 . [En línea]. Available: https://google-cartographer-ros.readthedocs.io/en/latest/.

[27] S. Macenski, «SLAM Toolbox,» 2020. [En línea]. Available: https://github.com/SteveMacenski/slam_toolbox.

ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS

En 2015 la organización de las naciones unidas (ONU) aprobó la Agenda 2030 sobre el Desarrollo Sostenible, una oportunidad para que los países y sus sociedades emprendan un nuevo camino con el que mejorar la vida de todos, sin dejar a nadie atrás.

Esta agenda se compone de 17 objetivos propuestos de diferente índole. Estos van desde la erradicación de la pobreza hasta el combate contra el cambio climático, la educación, la igualdad de la mujer, la defensa del medio ambiente o el diseño de nuestras ciudades. Estos objetivos se pueden separar principalmente en 3 grandes grupos: social, económico y medioambiental. En el grupo social se incluyen los objetivos 1, 2, 3, 4, 5, 7, 11 y 16, en el grupo económico los objetivos 8, 9, 10 y 12 y por último el grupo medioambiental incluye los objetivos 6, 13, 14 y 15.

Al principio del proyecto se habló de Industria 4.0, una nueva resolución, y de cómo ROS y la robótica que lo usa es una parte fundamental de ella. Y es que esta nueva industria tiene como pilares fundamentales la eficiencia energética y la sostenibilidad. Cada vez tiene más sentido el uso de soluciones tecnológicas de este tipo si consideramos que el sector industrial en España es responsable del 31% del consumo total de energía.

La identificación de los ODS para la Industria es una condición necesaria para conseguir el objetivo de sostenibilidad y de los 17, dos de ellos guardan una estrecha relación con el sector. Estos ODS son; 9 “Desarrollar infraestructuras resilientes, promover la industrialización inclusiva y sostenible, y fomentar la innovación.”; 12 “Garantizar las pautas de consumo y de producción sostenibles.”.

Como objetivo principal se puede considerar el número 12. Para alcanzar los objetivos de sostenibilidad y crear nuevas oportunidades serán necesarias, entre otras medidas, la

implantación de soluciones IoT de eficiencia energética capaces de controlar y monitorizar el consumo energético, redes inteligentes que permitan tomar decisiones en tiempo real.

Gracias a los sensores, algoritmos y diferentes redes de comunicación, hoy la automatización a la hora de crear y distribuir energía se enfoca en una línea mucho más sostenible. Podemos anticipar la demanda eléctrica de una ciudad o una industria con varios meses de antelación, al tiempo que se puede hacer llegar la energía a núcleos más pequeños o aislados de población.

Según señala el informe del Foro Económico Mundial, la llamada energía inteligente, aquella que utiliza redes de información y sensores de IoT para su almacenamiento y distribución, permitirá una mayor eficiencia y provocará una reducción en el precio del kW. Todo esto puede llevarnos a que de aquí al año 2030, el ahorro en el consumo energético esté por encima de de 1.300 millones de MWh.

Como objetivo secundario del proyecto encontraríamos el objetivo número 9. El desarrollo de la automatización y la robótica está permitiendo que las empresas medianas y pequeñas puedan incorporar sistemas de guiado automático de vehículos AGV en sus procesos productivos. Las compañías están exigiendo manejar los materiales necesarios con más rapidez, precisión y exactitud. El conocimiento del proceso productivo y el control de su gasto resultan imprescindibles si se quiere mantener una posición dentro del mercado.

Estos desarrollos ayudarían a las pequeñas y medianas empresas que se dedican al procesamiento industrial y la producción manufactura, ya que estas son las más críticas en las primeras etapas de la industrialización y, por lo general, son los mayores creadores de empleos. Constituyen más del 90% de las empresas de todo el mundo y representan entre el 50 y el 60% del empleo.

En conclusión, el uso de las nuevas tecnologías y la trasformación digital de la industria serán factores importantes a la hora de cumplir los objetivos establecidos.

ANEXO II

Función sysCallSensing del sensor laser en VREP:

```
function sysCall_sensing()
    measuredData={}
    ranges = {} ----------------------------------------------------añadido

    if notFirstHere then
        -- We skip the very first reading
        sim.addDrawingObjectItem(lines,nil)

showLines=sim.getScriptSimulationParameter(sim.handle_self,'showLaserSegments')
        r,t1,u1=sim.readVisionSensor(visionSensor1Handle)
        r,t2,u2=sim.readVisionSensor(visionSensor2Handle)
        m1=sim.getObjectMatrix(visionSensor1Handle,-1)
        m01=simGetInvertedMatrix(sim.getObjectMatrix(sensorRef,-1))
        m01=sim.multiplyMatrices(m01,m1)
        m2=sim.getObjectMatrix(visionSensor2Handle,-1)
        m02=simGetInvertedMatrix(sim.getObjectMatrix(sensorRef,-1))
        m02=sim.multiplyMatrices(m02,m2)
        if u1 then
            p={0,0,0}
            p=sim.multiplyVector(m1,p)
            t={p[1],p[2],p[3],0,0,0}
            for j=0,u1[2]-1,1 do
                for i=0,u1[1]-1,1 do
                    w=2+4*(j*u1[1]+i)
                    v1=u1[w+1]
                    v2=u1[w+2]
                    v3=u1[w+3]
                    v4=u1[w+4]
                    if (v4<maxScanDistance_) then
                        p={v1,v2,v3}
                        p=sim.multiplyVector(m01,p)
                        table.insert(measuredData,p[1])
                        table.insert(measuredData,p[2])
                        table.insert(measuredData,p[3])
                        table.insert(ranges, v4) ------------------añadido
                    else ------------------------------------------añadido
                        table.insert(ranges, 0) -------------------añadido

                    end

                    if showLines then
                        p={v1,v2,v3}
                        p=sim.multiplyVector(m1,p)
                        t[4]=p[1]
                        t[5]=p[2]
```

```
                    t[6]=p[3]
                    sim.addDrawingObjectItem(lines,t)
                end
            end
        end
    end
    if u2 then
        p={0,0,0}
        p=sim.multiplyVector(m2,p)
        t={p[1],p[2],p[3],0,0,0}
        for j=0,u2[2]-1,1 do
            for i=0,u2[1]-1,1 do
                w=2+4*(j*u2[1]+i)
                v1=u2[w+1]
                v2=u2[w+2]
                v3=u2[w+3]
                v4=u2[w+4]
                if (v4<maxScanDistance_) then
                    p={v1,v2,v3}
                    p=sim.multiplyVector(m02,p)
                    table.insert(measuredData,p[1])
                    table.insert(measuredData,p[2])
                    table.insert(measuredData,p[3])
                    table.insert(ranges, v4) ------------------añadido
                else ----------------------------------------añadido
                    table.insert(ranges, 0) -------------------añadido
                end
                if showLines then
                    p={v1,v2,v3}
                    p=sim.multiplyVector(m2,p)
                    t[4]=p[1]
                    t[5]=p[2]
                    t[6]=p[3]
                    sim.addDrawingObjectItem(lines,t)
                end
            end
        end
    end
 --data=sim.packFloatTable(measuredData)
 --sim.setStringSignal("measuredDataAtThisTime",data
 ranges = sim.packFloatTable(ranges) ----------------------añadido
 sim.setStringSignal('scan ranges', ranges) ----------------añadido

end
```

Quaternion from Euler:

```
# axis sequences for Euler angles
_NEXT_AXIS = [1, 2, 0, 1]

# map axes strings to/from tuples of inner axis, parity, repetition, frame
_AXES2TUPLE = {
    'sxyz': (0, 0, 0, 0), 'sxyx': (0, 0, 1, 0), 'sxzy': (0, 1, 0, 0),
    'sxzx': (0, 1, 1, 0), 'syzx': (1, 0, 0, 0), 'syzy': (1, 0, 1, 0),
    'syxz': (1, 1, 0, 0), 'syxy': (1, 1, 1, 0), 'szxy': (2, 0, 0, 0),
    'szxz': (2, 0, 1, 0), 'szyx': (2, 1, 0, 0), 'szyz': (2, 1, 1, 0),
    'rzyx': (0, 0, 0, 1), 'rxyx': (0, 0, 1, 1), 'ryzx': (0, 1, 0, 1),
    'rxzx': (0, 1, 1, 1), 'rxzy': (1, 0, 0, 1), 'ryzy': (1, 0, 1, 1),
    'rzxy': (1, 1, 0, 1), 'ryxy': (1, 1, 1, 1), 'ryxz': (2, 0, 0, 1),
    'rzxz': (2, 0, 1, 1), 'rxyz': (2, 1, 0, 1), 'rzyz': (2, 1, 1, 1)}

_TUPLE2AXES = dict((v, k) for k, v in _AXES2TUPLE.items())

def quaternion_from_euler(ai, aj, ak, axes='sxyz'):
    """Return quaternion from Euler angles and axis sequence.
    ai, aj, ak : Euler's roll, pitch and yaw angles
    axes : One of 24 axis sequences as string or encoded tuple
    >>> q = quaternion_from_euler(1, 2, 3, 'ryxz')
    >>> numpy.allclose(q, [0.310622, -0.718287, 0.444435, 0.435953])
    True
    """
    try:
        firstaxis, parity, repetition, frame = _AXES2TUPLE[axes.lower()]
    except (AttributeError, KeyError):
        _ = _TUPLE2AXES[axes]
        firstaxis, parity, repetition, frame = axes

    i = firstaxis
    j = _NEXT_AXIS[i+parity]
    k = _NEXT_AXIS[i-parity+1]

    if frame:
        ai, ak = ak, ai
    if parity:
        aj = -aj

    ai /= 2.0
    aj /= 2.0
    ak /= 2.0
    ci = math.cos(ai)
    si = math.sin(ai)
    cj = math.cos(aj)
    sj = math.sin(aj)
    ck = math.cos(ak)
    sk = math.sin(ak)
    cc = ci*ck
    cs = ci*sk
```

```
    sc = si*ck
    ss = si*sk

    quaternion = numpy.empty((4, ), dtype=numpy.float64)
    if repetition:
        quaternion[i] = cj*(cs + sc)
        quaternion[j] = sj*(cc + ss)
        quaternion[k] = sj*(cs - sc)
        quaternion[3] = cj*(cc - ss)
    else:
        quaternion[i] = cj*sc - sj*cs
        quaternion[j] = cj*ss + sj*cc
        quaternion[k] = cj*cs - sj*sc
        quaternion[3] = cj*cc + sj*ss
    if parity:
        quaternion[j] *= -1

    return quaternion
```