Universidad Zaragoza
1542

Trabajo Fin de Grado

Simulador en ROS de una plataforma de robots móviles

ROS simulator of a mobile robot platform

Autor/es

Ariadna Elena Chavarría

Director/es

Cristian Mahulea
Eduardo Montijano

Grado en Ingeniería de Tecnologías Industriales

Escuela de Ingeniería y Arquitectura

2021

Resumen.

Los simuladores de robots móviles se han convertido en una herramienta fundamental para el desarrollo y la investigación en el campo de la robótica, así como un medio para el aprendizaje de esta disciplina en el ámbito educativo.

En el presente trabajo se quiere simular una plataforma de robots móviles utilizada para el aprendizaje en el ámbito de la automatización y la robótica en la Universidad de Zaragoza. Para ello se hará uso de Robot Operating System (ROS), un entorno de trabajo para el desarrollo de software en esta área.

El punto de partida son los datos reales que esta plataforma proporciona sobre sus dimensiones, características y regiones de interés que se sitúan en ella. A partir de estos se ha desarrollado un programa en C++ que genera de forma automática un entorno de simulación de la plataforma utilizando ROS y el simulador en 2D Stage.

Una vez se tiene el entorno simulado la segunda parte del trabajo se centrará en llevar a cabo distintas simulaciones en las que los robots alcancen las coordenadas deseadas mediante una navegación autónoma.

Se consigue finalmente simular distintos escenarios de la plataforma mediante el uso del software creado. Además, en estas simulaciones se envía a los robots a distintas posiciones objetivo y estos consiguen alcanzarlas satisfactoriamente evitando los obstáculos del entorno.

Este trabajo de Fin de Grado pretende dotar a la plataforma de nuevas funciones para facilitar el desarrollo de futuras aplicaciones de robótica en la misma en distintos trabajos de fin de grado o máster.

Abstract.

Mobile robot's simulators have become a fundamental tool in robotics development and investigation, just as a way for learning of this subject in the educational field.

The goal of this project is to simulate a mobile robot's platform which is used in Zaragoza University for learning about automation and robotics. Robot Operating System (ROS), that is a framework to software development in this scope, will be used to achieve this goal.

This work starts with the information and real data provided by the platform. Those data are about dimensions and characteristics of the platform and the objects that are placed in it. Based on this information a C++ program will be created in order to generate automatically a map of the platform for a 2D simulator called Stage.

When the simulated environment is set up the second part of the project will be focus on autonomous navigation simulations. The simulated robots will be sent to a goal position that they should catch.

Finally, different scenarios of the platform are simulated by using the software created. Furthermore, in these simulations the robots are sent to different positions, and they accomplish their goal of catching properly those positions avoiding the obstacles of the environment.

This final degree project aims to provide the platform with new functions to facilitate the development of future robotics applications in different final degree or master´s projects.

Índice.

Capítulo 1. 9

Introducción 9

1.1 Motivación 9

1.2 Objetivos. 9

1.3 Alcance. 10

1.4 Planificación 10

1.5 Contenido. 11

Capítulo 2. 12

Estado del arte. 12

2.1 Robótica. 12

2.2 Sistemas multi robot. 13

2.3 Simuladores de robótica móvil 13

2.4 Stage. 14

2.5 Navegación autónoma. 15

Capítulo 3. 17

El *framework* ROS. 17

3.1 Características y objetivos. 17

3.2 Instalación y puesta en marcha. 18

3.3 Fundamentos básicos 18

3.4 Rviz. 20

3.5 *Launch files*. 20

3.5.1 Remapeado. 21

3.5.2 *Include* de otros archivos. 21

3.5.3 Rosparam. Parámetros. 21

Capítulo 4 23

Simulación de plataforma de robots móviles. 23

4.1 Plataforma de robots móviles. 23

4.1.1 Datos de entrada. 23

4.2 Ficheros. 24

4.2.1 Fichero de entrada 25

4.2.2 Fichero de salida. 25

4.2.2.1 Posicionamiento y dimensión de los modelos. 28

4.3 Desarrollo del programa para la creación del mapa de simulación. 30
4.3.1 Principales recursos de C++ utilizados. 30
4.4 Implementación en ros 31
4.4.1 Simulación. 32
Capítulo 5. 34
Navegación autónoma en simulación. 34
5.1 Navigation stack 34
5.1.1 Transformaciones. (*sensor transform*s) 35
5.1.2 Información de sensores. (*sensor sources*) 35
5.1.3 Información sobre odometría. (*odometry source*) 36
5.1.4 Controlador. (*base controller*) 36
5.2 *Launch* de navegación. 36
5.2.1 Stage. 37
5.2.2 Map 37
5.2.3 Move Base. 37
5.2.3.1 Costmap2d: global costmap y local costmap. 37
5.2.3.2 Planners. 38
5.2.4 AMCL. 39
5.3 Simulación de navegación con un solo robot 40
5.3.1 Configuración dinámica de parámetros. 41
5.3.2 Resultados. 43
Capítulo 6 46
Conclusiones y líneas futuras. 46
6.1 Conclusiones 46
6.2 Líneas de trabajo futuras 46
Capítulo 7. 48
Bibliografía. 48
ANEXOS. 49
Anexo I. Fichero de entrada, datos modulo. (Caso 1) 49
Anexo II. Fichero de salida, mapa. (Caso 1) 50
Anexo III. Mapeado del entorno simulado. *gmapping*. 53
Anexo IV. Navegación un solo robot. 55
Anexo V. Contenido de la carpeta Drive. 56

Archivos. 56
Grabaciones pruebas. 57

Índice de figuras.

Figura 1. Modelo simple y complejo de bloques de un robot Pioneer. [9] 15
Figura 2. Modelo de Stage a partir de un bitmap. [5] 15
Figura 3. Esquema de comunicación de los nodos [2]. 19
Figura 4. Esquema de organización en ROS. [17]. 19
Figura 5. Imagen plataforma de robots móviles del laboratorio. 24
Figura 6. Esquema de archivos. 24
Figura 7. Representación gráfica del modelo object1. 29
Figura 8. Plataforma real Caso 1. 32
Figura 9. Plataforma simulada Caso 1. 32
Figura 10. Plataforma simulada Caso 2. ..... **¡Error! Marcador no definido.**
Figura 11. Plataforma real Caso 2. 32
Figura 12. Plataforma real Caso 3. 32
Figura 13. Plataforma simulada Caso 3. 32
Figura 14. Esquema de nodos y topics activos. 33
Figura 15. Esquema general del funcionamiento de navigation stack. [11] 34
Figura 16. Transform tree 35
Figura 17. Esquema rqt_graph navegación de robot móvil. 39
Figura 18. Simulación en Stage. 40
Figura 19. Visualización ventana Rviz. 40
Figura 20. Visualización map inicial. 41
Figura 21. Visualización local costmap inicial. 41
Figura 22. Visualización global costmap inicial. 41
Figura 23. Vista ventana rtq_reconfigure 42
Figura 24. Visualización map ajustado. 43
Figura 25. Visualización local costmap ajustado. 43
Figura 26. Visualización global costmap ajustado. 43
Figura 27. Robot trazando trayectoria. Robot alcanzando objetivo. 44
Figura 28. Terminal con las coordenadas alcanzadas. 44
Figura 29. map2.pgm. 53
Figura 30. Transform tree completo para un solo robot. 55

Índice comandos de ROS.

Código ROS 1. Creación de espacio de trabajo. 18
Código ROS 2. Actualización de espacio de trabajo. 18
Código ROS 3. Ejecución del master. 20
Código ROS 4. Ejecutar un archivo tipo launch. 20
Código ROS 5. Visualizar transform tree. 35
Código ROS 6. Comando para ejecutar rqt_reconfigure. 42
Código ROS 7. Publicar objetivo en move_base_simple/goal 44
Código ROS 8. Comprobación de coordenadas del sistema de referencia base_link respecto a map 44
Código ROS 9. Ejecutar nodo slam_gmapping para mapeado. 53
Código ROS 10. Guardar mapa generado con el nombre map2. 53

Capítulo 1.
Introducción.

En este apartado se introducen los objetivos del presente proyecto, así como su alcance que servirá de introducción para el posterior desarrollo del trabajo.

1.1 Motivación.

El uso de la robótica está cada vez más extendido en muchos ámbitos de la sociedad. Son muchas las tareas que los robots pueden desempeñar, en especial los robots autónomos que mediante sus sensores y actuadores son capaces de percibir el entorno en el que se encuentran e interactuar con él. La navegación es una parte fundamental en el comportamiento de los robots móviles autónomos, esta capacidad permite a los dispositivos desplazarse de un punto a otro de forma segura. El desarrollo de software para robots móviles y autónomos es por tanto fundamental para lograr avances y aumentar las aplicaciones de la robótica.

A nivel docente se trabaja con distintos proyectos para el aprendizaje en el desarrollo de software. En este caso se parte de una plataforma de robots móviles disponible en la Universidad de Zaragoza. Este trabajo surge a partir de la idea de trasladar la plataforma real a un simulador. La principal motivación es dotar a esta plataforma de una nueva funcionalidad que facilite el futuro desarrollo de software en robots móviles.

El uso de simuladores en esta área ofrece la posibilidad de realizar pruebas de forma rápida y eficaz. Un elemento clave es la repetitividad en la experimentación. En el desarrollo de software en general es necesario realizar numerosas pruebas para comprobar el funcionamiento y los posibles errores de los programas creados. Esto resulta mucho más sencillo cuando se dispone de un entorno simulado en el que realizar dichas pruebas con facilidad.

Por otro lado, es posible trabajar de manera simultánea en distintos proyectos de la plataforma ya que no será necesario el uso del escenario real. Esto supone un ahorro en coste y tiempo en la investigación.

El hecho de poder llevar el escenario de la plataforma al entorno de simulación de forma automática puede suponer una gran ventaja para la implementación de futuras aplicaciones.

1.2 Objetivos.

El objetivo principal que se persigue es dotar a una plataforma de robots móviles de funcionalidades como la simulación en dos dimensiones de la misma de forma automática. Para ello se divide el trabajo en dos subobjetivos específicos.

El primero se centra en desarrollar un programa que lance la simulación con el entorno y los robots deseados. Esta primera parte se basa en el uso del lenguaje de programación C++ para el desarrollo del programa, así como del simulador Stage con el que se trabaja durante todo el proyecto y de distintas herramientas que proporciona ROS.

La navegación autónoma constituye la segunda parte de este trabajo. El fin de las simulaciones de la plataforma es el de poder evaluar de forma sencilla y eficaz distintos algoritmos y aplicaciones sin la necesidad de trabajar con un entorno real.

Mediante distintas herramientas y paquetes de ROS se configuran los robots simulados para que de manera autónoma se trasladen a unos objetivos deseados evitando los obstáculos del entorno.

Desde otro punto de vista en este trabajo se busca familiarizarse con el entorno de ROS y algunas de sus funcionalidades. ROS se presenta como uno de los softwares de robótica más prometedores, es por esto que a lo largo del proyecto se quiere adquirir cierto conocimiento sobre conceptos básicos del mismo.

En la misma línea se pretende desarrollar aptitudes en cuanto a lenguaje de programación y creación de programas ampliando así conocimientos en el ámbito de la informática y la automatización.

1.3 Alcance.

El proyecto se desarrolla utilizando una plataforma disponible en el Departamento de Informática e Ingenieria de Sistemas en la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza. Esta plataforma es utilizada para el desarrollo de distintas líneas de investigación y el aprendizaje en problemas de planificación y percepción de robots. Por ello el objetivo final es proporcionar a la plataforma nuevas funcionalidades.

El alcance del proyecto consiste en conseguir distintas simulaciones del entorno de la plataforma en las que los robots alcancen unos objetivos o posiciones determinadas del escenario de la plataforma evitando los obstáculos que se disponen en ella. Como se expone en los objetivos se va a realizar un programa que lance la simulación convirtiendo el entorno real de la plataforma en un entorno simulado en Stage con todos los componentes necesarios.

Las herramientas que se utilizan en el desarrollo de este trabajo son lenguaje de programación C++, el simulador Stage y el *framework* ROS.

1.4 Planificación

Con el fin de cumplir los objetivos planteados es necesario tener una adecuada planificación. A continuación, se detallan las fases que se han seguido a lo largo del trabajo.

- Aprendizaje sobre el lenguaje de programación C++ a través de distintas fuentes.
- Familiarización con el entorno de ros y sus conceptos básicos mediante la realización de tutoriales oficiales.
- Estructuración de los datos de partida procedentes de la plataforma en un fichero.
- Desarrollo del programa para cumplir el primero de los objetivos planteados.

- Adquisición de conocimientos sobre navegación autónoma en ROS y distintos paquetes cumplimentando distintos tutoriales.

- Implementación de estos conocimientos para la realización de las simulaciones deseadas realizando distintas pruebas.

- Conclusiones sobre el trabajo realizado.

Durante la realización del trabajo se van cumpliendo los puntos señalados tratando de alcanzar objetivos a corto plazo, así como solucionando los problemas y retos que van surgiendo.

1.5 Contenido.

En este punto se resumen los capítulos que se desarrollan a lo largo del trabajo y se describe brevemente de que tratan.

- **Introducción**: marca los objetivos que se abordan en este trabajo y expone el contexto en el que surge.

- **Estado del arte**: describe aspectos básicos relacionados con el tema del trabajo que sirven a su vez de introducción.

- **El *framework* ROS**: capítulo en el que se explica qué es ROS y algunas de sus herramientas que posteriormente se utilizan en el proyecto.

- **Simulación de plataforma de robots móviles**: se expone el planteamiento seguido para el desarrollo del software que hace posible la creación del entorno para la simulación de la plataforma en el entorno de Stage.

- **Navegación autónoma**: se explica el proceso para la configuración del sistema de navegación adaptándolo a la simulación de la plataforma.

- **Conclusiones y líneas futuras**: punto en el que se exponen las conclusiones del trabajo y posibles líneas de desarrollo futuro a partir del mismo.

Capítulo 2.
Estado del arte.

En este capítulo se comentan aspectos relacionados con el contexto en el que surge el trabajo. Con la finalidad de tener una base y un marco teórico adecuado para afrontar los objetivos planteados, se comentan puntos como el concepto de robótica, los sistemas multi robot, los simuladores de robótica móvil como herramienta de aprendizaje, el simulador Stage y la navegación autónoma.

2.1 Robótica.

La **robótica** es una disciplina que ha sufrido un desarrollo muy importante en los últimos años. Por definición la robótica es la ciencia que trata del diseño, la fabricación y la programación de robots con el objetivo de que estos desarrollen distintas funcionalidades.

Un robot es una máquina capaz de ser programada para la realización de distintas tareas a través de su interacción con el entorno. En la definición de estas máquinas que diferencian tres partes fundamentales.

- **Actuadores**: elementos como brazos, ruedas, motores… que permiten a la maquina actuar sobre el entorno e interactuar con él.

- **Sensores**: conjunto de dispositivos como sonars, láseres o cámaras que proporcionan información sobre el entorno.

- **Inteligencia**: algoritmos y programas que dotan al robot de la capacidad de procesar la información que proporcionan los sensores y convertirla en acciones.

La mayor parte de investigación se centra en desarrollar la inteligencia de estas máquinas proporcionándoles una mayor capacidad de autonomía. Esta autonomía es la que hace que los robots abarquen cada vez más campos de aplicación.

Los avances de los últimos años se deben a las numerosas aplicaciones que estos dispositivos son capaces de desempeñar en distintos ámbitos. La robótica y la automatización se han convertido en elementos clave principalmente en la industria debido a sus grandes ventajas. En este aspecto la aplicación de robots en el ámbito industrial permite flexibilizar procesos productivos, aumentar los niveles de productividad o conseguir un proceso más homogéneo y de mayor calidad.

No solo en la industria encontramos las ventajas de la automatización sino también en el ámbito doméstico o de servicios, sectores en los que facilitan al ser humano la realización de distintas tareas.

2.2 Sistemas multi robot.

En ocasiones se utilizan equipos de robots que cooperan para la realización de distintas tareas. La ventaja que presentan estos sistemas es la descomposición de una tarea compleja en subtareas más sencillas realizadas por distintos robots individualmente.

Los sistemas multi robots se caracterizan por el control distribuido, la autonomía, la tolerancia a fallos y la comunicación [6]. Estas características aumentan la eficiencia en la realización de algunas operaciones, sobre todo en tareas de exploración en las que múltiples robots pueden trabajar conjuntamente para realizar un mapeado del entorno. Resulta más eficaz hacer uso de múltiples robots que tener que utilizar un robot muy potente de alto coste para la realización de una tarea.

2.3 Simuladores de robótica móvil.

Dada la situación de auge en la que se encuentra el campo de la robótica, no es de extrañar que cada vez aumente más su presencia en el ámbito educativo principalmente vinculada a estudios de ingeniería.

En este contexto surge el problema del elevado coste que supone la realización de prácticas con robots reales y que está al alcance de pocos centros educativos. La aparición de micro robots más económicos aumenta las posibilidades en la enseñanza de robótica móvil, sin embargo, se ha demostrado que este tipo de robots presentan elevadas limitaciones a la hora de implementar algoritmos más sofisticados.

Los **simuladores** para robots móviles se presentan como una alternativa para el desarrollo de robótica en el ámbito de la educación y también en el profesional. Este tipo de herramientas ofrecen resultados sobre entornos simulados lo cual supone una ventaja considerable frente a prácticas en un entorno real.

Esta metodología ofrece la posibilidad de repetir experimentos o prácticas de manera sencilla y económica por lo que son de gran utilidad por ejemplo en investigación con grupos de numerosos robots. La repetitividad en experimentos reales resulta difícil de conseguir ya que las condiciones del entorno pueden variar, algo que no ocurre con una simulación. Además de abaratar costes en investigación se consigue una gran facilidad para la obtención de datos experimentales.

En un artículo sobre el uso de simuladores en docencia de robótica móvil del IEEE [3] comparan algunos de los simuladores más importantes que se van a señalar a continuación.

1. **SRIsim/Saphira**: un sistema de control de robots dirigido a la simulación de un único robot en un mundo bidimensional estático.

2. **Webots**: este simulador forma parte de *Microsoft Robotics Studio* y se trata de un simulador tridimensional que puede simular multitud de robots permitiendo a su vez el uso de distintos lenguajes.

3. **Player/Stage/Gazebo**: en este caso se diferencian dos simuladores Stage y Gazebo. Stage es un simulador en 2D al contrario que Gazebo que simula escenarios en tres dimensiones. La plataforma Player permite el uso de cualquier lenguaje de programación para el desarrollo de aplicaciones.

4. **USARSim**: se trata de un simulador caracterizado por ofrecer alto grado de realismo en cuanto a escenarios y simulación de elementos.

En el artículo los autores concluyen que el conjunto de **Player/Stage/Gazebo** es la mejor alternativa debido a sus características de licencia libre, extensibilidad, variedad en cuando modelos de robots a simular y la posibilidad de realizar simulaciones multi robot. Este trabajo se sitúa entonces en la utilización de Stage para la simulación 2D de múltiples robots móviles.

2.4 Stage.

Stage es un simulador en 2D que surge como parte del proyecto Player. **Player** es una capa de abstracción de hardware [1] para dispositivos robóticos. El proyecto Player crea un software de código libre para la investigación en robots y sistemas de sensores.

El simulador Stage se puede utilizar en distintas plataformas como Linux, Ubuntu o Windows ya que está diseñado para ser independiente del lenguaje y la plataforma que se utilice.

Algunas de las características más importantes de Stage son la **cooperación con Player**, cuyo uso está muy extendido en el mundo de la robótica, su **uso** relativamente **sencillo** y el hecho de simular un entorno con el **suficiente realismo** para muchas aplicaciones. Proporciona **modelos** para muchos de los **sensores** más comunes y es capaz de simular **múltiples robots** en un mismo escenario. Además del hecho de poder usarlo en distintas plataformas [9].

Por otro lado, ser un software libre y tener una comunidad activa de usuarios y desarrolladores hace que esta herramienta sea una de las más utilizadas actualmente. Principalmente su uso está muy extendido en el panorama educativo.

Las simulaciones en Stage son de entornos bidimensionales totalmente configurables. El diseño del entorno y los elementos de una simulación se realiza mediante la creación de un archivo llamado ***worldfile***. En este archivo se definen los modelos que componen la simulación describiendo parámetros como su tamaño y forma además de otras características.

Stage se presenta como un simulador bidimensional ya que es su principal función, sin embargo, su última versión ofrece una vista en 3D. La forma física de modelos se define por bloques, en modelos más complejos su forma se puede entender como el agrupamiento de una serie de bloques. Los bloques se definen como polígonos en el plano x-y a los que luego se les da una altura en el eje z con la que se extruye el polígono o la forma definida. Estas formas pueden definirse como un vector de coordenadas formando bloques como en el modelo de la Figura 2 o mediante algún archivo de imagen o *bitmap* como en la Figura 1.

[1] Capa de abstracción de hardware: elemento del sistema operativo que funciona como interfaz entre el software y el hardware del sistema.

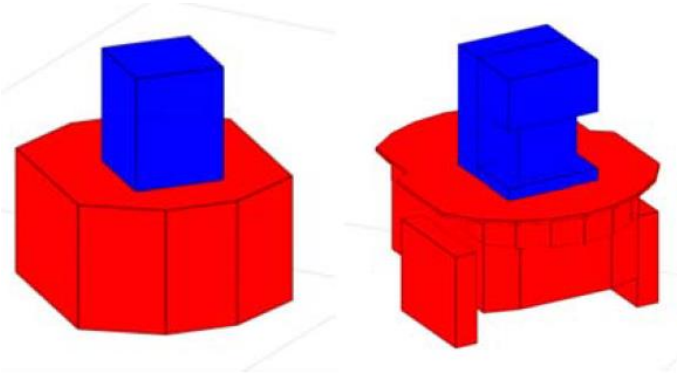

Figura 1. Modelo simple y complejo de bloques de un robot Pioneer. [9]

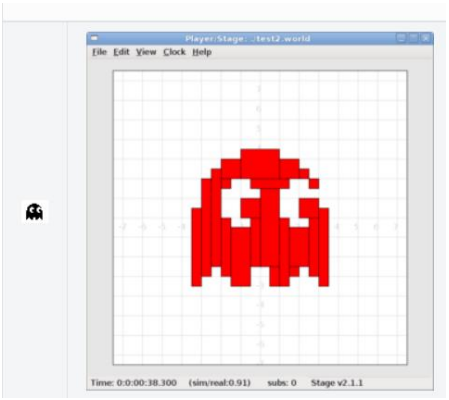

Figura 2. Modelo de Stage a partir de un bitmap. [5]

2.5 Navegación autónoma.

Los **robots autónomos** son cada vez más frecuentes en la industria. Un robot móvil autónomo es aquel capaz de realizar tareas en su entorno sin la necesidad de un control explicito por parte del usuario.

La navegación autónoma de robots móviles juega un papel fundamental en el desarrollo de este tipo de robots. La navegación dota al robot de la capacidad de moverse en un entorno con obstáculos de manera que este consiga llegar a su destino de forma segura. El objetivo es que el robot vaya desde un punto inicial a un destino final.

Los puntos clave en la navegación son la obtención de datos del entorno a través de los sensores disponibles, la planificación de la trayectoria para alcanzar el destino y el guiado del vehículo a través de la referencia construida [4] .

El problema de la navegación se divide en cuatro bloques:

1. **Percepción del mundo**. Se efectúa la creación de un mapa a partir de los datos que proporcionan los sensores del robot. Será en ese mapa donde se efectuará la navegación.

2. **Planificación de la ruta**. Creación de una secuencia de objetivos intermedios que el robot debe alcanzar utilizando el mapa generado.

3. **Generación del camino**. Mediante la interpolación de los puntos intermedios se crea el camino que debe seguir la máquina.

4. **Seguimiento del camino**. Desplazamiento del robot a través de la ruta trazada.

La realización de estas tareas conforma el esquema básico de la navegación. Es importante destacar que en el punto de planificación se puede distinguir entre **planificación global y local**. La planificación global es en sí la creación de objetivos comentada en el punto dos, mientras el aspecto de la planificación local hace referencia a la obtención de información de los sensores sobre el entorno más cercano del robot. La importancia de la planificación local reside en evitar los posibles obstáculos, eventualmente dinámicos, que aparezcan en el entorno

y no estén contemplados en el mapa generado. De esta forma se puede ir ajustando la planificación inicial para conseguir llegar al destino con éxito.

En definitiva, estos son los aspectos básicos que hacen posible la navegación de este tipo de dispositivos en distintos entornos de manera autónoma. Esto constituye la base para un gran número de aplicaciones.

Estas aplicaciones van desde exploración espacial a robots utilizados en el ámbito industrial. En almacenes y fabricas industriales se hace uso de este tipo de robots para que lleven mercancías o cargas de un punto a otro. Otro ejemplo muy cercano en el ámbito doméstico son los robots aspiradores que actualmente están presentes en muchos hogares.

Capítulo 3.
El *framework* ROS.

En este apartado se van a explicar los fundamentos básicos de ROS que se usarán más adelante en este trabajo.

Robot Operating System es un meta sistema operativo de código abierto que proporciona servicios como abstracción de software, control de bajo nivel, implementación de funcionalidades comunes, pasaje de mensajes entre procesos y manejo de paquetes. También brinda herramientas y librerías para obtener, construir, escribir y correr código a través y mediante varias computadoras [15].

3.1 Características y objetivos.

El campo de la robótica ha avanzado considerablemente en los últimos años, aun así, la investigación y desarrollo de software sigue presentando ciertos retos. ROS surge como una plataforma para la creación de software de robótica capaz de solventar algunos de ellos. De forma que este sistema operativo nos permite afrontar aspectos [7] como:

- La **computación distribuida**, muchos de los robots actuales funcionan mediante softwares que ejecutan y manejan distintos procesos al mismo tiempo. Es necesaria una comunicación entre los distintos procesos para el correcto funcionamiento del sistema, ROS presenta mecanismos para este tipo de comunicación.

- La **reutilización de software** lo que permite aprovechar algoritmos de tareas comunes como navegación o mapeo.

- La comprobación del funcionamiento de algoritmos con robots físicos supone un esfuerzo considerable. ROS proporciona herramientas para facilitar esta **comprobación mediante simulador**.

Estos aspectos son los que diferencian a ROS de otras plataformas robóticas, lo cual se consigue por su característica estructura de funcionamiento. Se trata de una estructura distribuida de nodos que permite el diseño individualizado pero adaptable. Estos procesos llamados nodos se pueden agrupar en paquetes o pilas que son fácilmente intercambiables, distribuidos y compartidos.

ROS está diseñado de modo que el código escrito se pueda utilizar en otros *frameworks* o estructuras de desarrollo. Es fácil de implementar en cualquier lenguaje de programación, este trabajo en concreto se desarrolla mediante lenguaje C++, si bien se podría haber empleado Python otro de los lenguajes más comunes. Por tanto, una de las cualidades principales de ROS es la posibilidad de utilizar las bibliotecas desarrolladas con este software en otra plataforma de desarrollo.

3.2 Instalación y puesta en marcha.

ROS no es un sistema operativo en sí mismo si no que funciona sobre otro. Principalmente esta desarrollado para trabajar en Ubuntu (Linux) aunque están disponibles versiones para trabajar en sistemas operativos como Mac y Windows en líneas experimentales.

Debido a su estabilidad, durante el proyecto se trabajará con **Ubuntu** que es una distribución de Linux, un software libre y de código abierto. Para ello se hará uso de una máquina virtual en la que se instala el sistema operativo Linux y el *framework* ROS. Se hace uso de **Oracle VM Virtual Box**, un software de virtualización con el que se instalan sistemas operativos adicionales, será como tener un ordenador dentro de otro.

Una vez se dispone del sistema operativo junto con ROS instalados, el siguiente paso es crear un **espacio de trabajo**. En él se crearán los distintos paquetes con los que se vaya a trabajar. Para crear este espacio de trabajo se ejecutarán los comandos mostrados a continuación.

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

Código ROS 1. Creación de espacio de trabajo.

Mediante estos comandos dentro del espacio *catkin_ws* creado, aparecerán tres carpetas. La carpeta principal es **src** en la que se crean o instalan los paquetes deseados, las dos restantes serán **devel** y **build** que aparecen al compilar el espacio de trabajo. Estas dos últimas carpetas no deben ser modificadas.

Al construir un paquete será necesario actualizar el entorno de trabajo con el siguiente comando.

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

Código ROS 2. Actualización de espacio de trabajo.

Así queda establecido el espacio de trabajo que se utilizará a lo largo del proyecto.

3.3 Fundamentos básicos.

ROS se estructura de forma modular mediante los llamados **nodos**, que son básicamente archivos ejecutables. Los nodos pueden escribirse en C++ o Python mediante las librerías que posee ROS, *roscpp* en el caso de C++ y *rospy* en el caso de Python.

Estos nodos se ejecutan dentro de un maestro (*master*) que es el núcleo del sistema y el que permite la comunicación entre nodos. Esta comunicación se lleva a cabo mediante ***topics*** o servicios (*services*) y se ejemplifica mediante el esquema de la Figura 3.

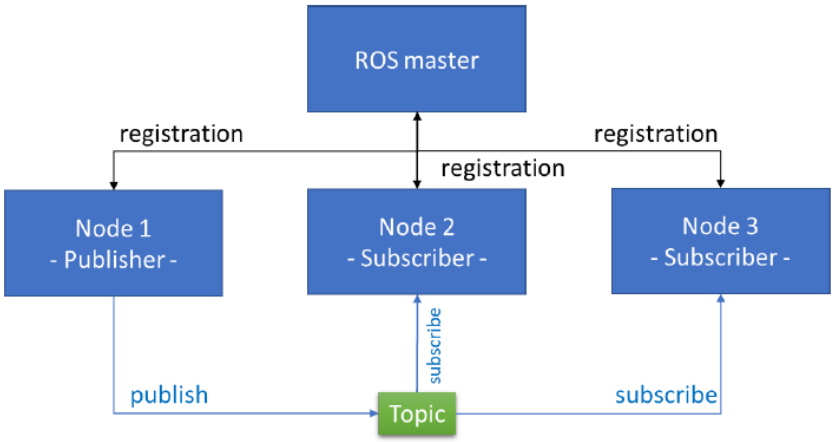

Figura 3. Esquema de comunicación de los nodos [2].

- **Messages**: datos e información que comparten los nodos entre sí.

- **Topics**: los mensajes (*messages*) se organizan en *topics*. Los *topics* funcionan de forma que cada nodo puede publicar o suscribirse a uno o varios *topics* para enviar o recibir mensajes. Como sucede en el esquema de la Figura 1 el nodo 1 publica en el *topic* señalado en color verde mientras los nodos 2 y 3 están suscritos a dicho *topic* y obtendrán los datos publicados por el nodo 1. Se puede entender un topic como un *bus* de un determinado tipo de mensajes.

- **Services**: los *topics* siguen una estructura de comunicación unidireccional de publicación/suscripción mientras los *services* permiten utilizar un modelo de solicitud/respuesta lo que permite una mayor interacción.

Los nodos se agrupan principalmente en **paquetes** (*package*) que constituyen la unidad principal de organización del sistema del sistema. Dichos paquetes se pueden apilar en pilas (*stacks*) formando una librería. La Figura 4 muestra como una pila puede agrupar varios paquetes.

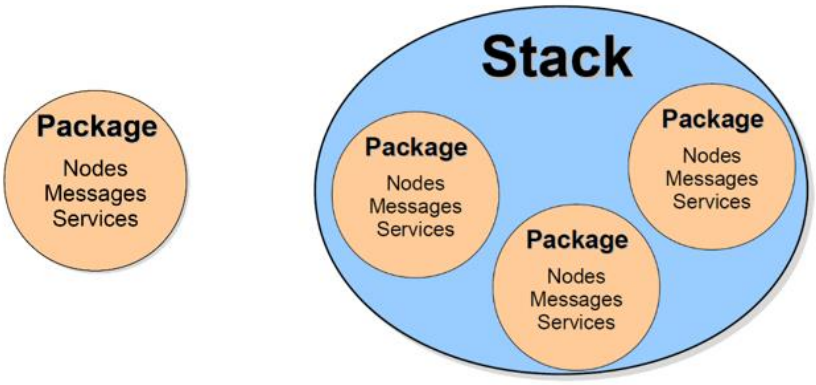

Figura 4. Esquema de organización en ROS. [17].

3.4 Rviz.

Rviz es una herramienta de visualización 3D muy útil de la que dispone ROS. Permite representar a los robots, su entorno, así como los datos sobre sensores de una manera muy visual. De forma que se ve lo que el propio robot está percibiendo del entorno o datos como la orientación y posición de este.

Esta herramienta cobra gran importancia a la hora de realizar simulaciones de navegación como las que se abordaran en la parte final del trabajo. Mediante esta visualización es más fácil comprender todos los datos e información que se maneja en la simulación. Además, Rviz también permite publicar mensajes en *topics* como enviar al robot simulado un objetivo.

3.5 *Launch files*.

Los archivos tipo ***launch*** son una herramienta que permite iniciar varios nodos a la vez además del *master* lo que supone una notable ventaja cuando se tiene un programa que hace uso de múltiples nodos. Normalmente para ejecutar el maestro y que el resto de nodos se comuniquen correctamente hay usar el comando:

```
roscore
```

Código ROS 3. Ejecución del master.

Al utilizar un archivo *launch* el *master* se ejecuta automáticamente.

Estos archivos son del tipo XML por lo que tienen una etiqueta principal conocida como *root element* que engloba a los demás elementos. En este caso y como se muestra en el ejemplo siguiente el *root element* es <launch>.

```
<launch>
<node name="listener" pkg="roscpp_tutorials" type="listener"
output="screen"/>
<node name="talker" pkg="roscpp_tutorials" type="talker"
output="screen"/>
</launch>
```

Código 1. Ejemplo de archivo launch.

En ella se especifican los nodos que se quieren ejecutar al mismo tiempo y de qué forma. Para ello se definen los siguientes parámetros básicos para cada nodo:

Si se quiere que los nodos se ejecuten en su forma predeterminada por la terminal, habrá que añadir el campo *output="screen"* en la definición del nodo.

Para ejecutar un archivo tipo *launch* se utiliza el comando **roslaunch** seguido del paquete en el que se encuentra y del nombre del archivo.

```
roslaunch package-name launch-file-name
```

Código ROS 4. Ejecutar un archivo tipo launch.

En el presente trabajo se hará uso de esta herramienta para lanzar las simulaciones pertinentes en el simulador Stage como se verá más adelante.

3.5.1 Remapeado.

El remapeado de *topics* no es más que un cambio de nombre. En ocasiones un nodo asigna un nombre especifico a un *topic* como puede ser "base_scan" mientras en otro nodo el *topic* que maneja esa misma información se llama "scan". Para que los nodos que están interconectados se entiendan correctamente hay que hacer que esos *topics* tengan el mismo nombre.

Un modo de hacer un remapeado es incluirlo en el *launch file*. Siguiendo con el ejemplo expuesto cambiaría el nombre de "base_scan" a "scan".

```
<remap from="original-name" to="new-name" />

<remap from="base_scan" to="scan" />
```

Código 2. Remapeado de topics.

3.5.2 *Include* de otros archivos.

En situaciones en las que se realiza un proyecto que requiere el uso de muchos nodos interconectados y una gran cantidad de parámetros se hace uso de ***includes*** como herramienta de organización.

De esta forma se puede incluir en un archivo principal otros *launch* con sus correspondientes nodos y parámetros definidos. Para ello se incluyen las rutas de fichero a los archivos que se quieren incluir.

```
<include file="path-to-launch-file" />
```

Código 3. Estructura Include de otros archivos en el launch file.

3.5.3 Rosparam. Parámetros.

Los **parámetros** son otro mecanismo que tiene ROS para proporcionar o recibir información de los nodos. Se trata de una herramienta que permite la configuración de los nodos.

Es común establecer los valores de ciertos parámetros de cada nodo en el archivo *launch*. Para ello se usa la siguiente estructura.

```
<param name="param-name" value="param-value" />
```

Código 4. Definición de parámetros en un launch.

De otra forma es posible definir una serie de parámetros deseados en un archivo y cargar ese archivo en el *launch*. Esto resulta útil cuando se tiene una larga lista de parámetros que corresponde a distintos nodos o distintos espacios.

```
<rosparam command="load" file="path-to-param-file" />
```

Código 5. Cargar archivos en un launch.

Capítulo 4.
Simulación de plataforma de robots móviles.

Este capítulo trata de cumplir el primero de los objetivos principales. A lo largo del capítulo se explica el planteamiento seguido.

En esta primera parte del proyecto se desarrolla un programa en lenguaje C++ para la obtención de un fichero que constituirá el escenario para la simulación de una plataforma de robots móviles en Stage. Esto se conseguirá procesando los datos e información que nos proporciona la propia plataforma.

4.1 Plataforma de robots móviles.

La plataforma que se simula en este trabajo es una plataforma multi robot disponible en uno de los laboratorios de la Escuela de Ingeniería y Arquitectura sobre la que se realizan distintos proyectos. En ella los robots evitan distintos obstáculos gracias a la función de una cámara que detecta las posiciones de estos para enviar las órdenes a los robots.

4.1.1 Datos de entrada.

Gracias al funcionamiento de esta plataforma y a la cámara de la que dispone se puede extraer información sobre las dimensiones y posiciones de los objetos y los robots que constituirán los datos de partida del presente proyecto y que se describen a continuación.

- **Plataforma.** La plataforma es de forma rectangular, la cámara proporciona sus dimensiones de ancho y largo en metros. Estas dimensiones son uno de los principales parámetros de entrada para poder dimensionar la simulación adecuadamente.

- **Robots**. En lo que se refiere a los robots se conoce la forma y el tamaño de estos. Se utilizan siempre los mismos ya que los robots disponibles en el laboratorio son del tipo “Turtle 2WD Mobile Robot” y tienen una forma circular con un radio 9 cm.

- **Objetos**. Se disponen por la plataforma varios obstáculos que pueden ser de distintas formas geométricas y tamaños. La cámara reconoce las esquinas de cada uno y obtiene sus coordenadas en la plataforma quedando así definida su forma y tamaño. Un aspecto a tener en cuenta es que el origen de coordenadas de la plataforma se sitúa en la esquina inferior izquierda del rectángulo. Además, se puede conocer el color de cada obstáculo, utilizándose principalmente tres colores como son rojo, azul y verde.

Estos son los componentes básicos de la plataforma que conformarán los datos de entrada para generar el mapa de simulación. En la Figura 5 se muestran todos los componentes citados.

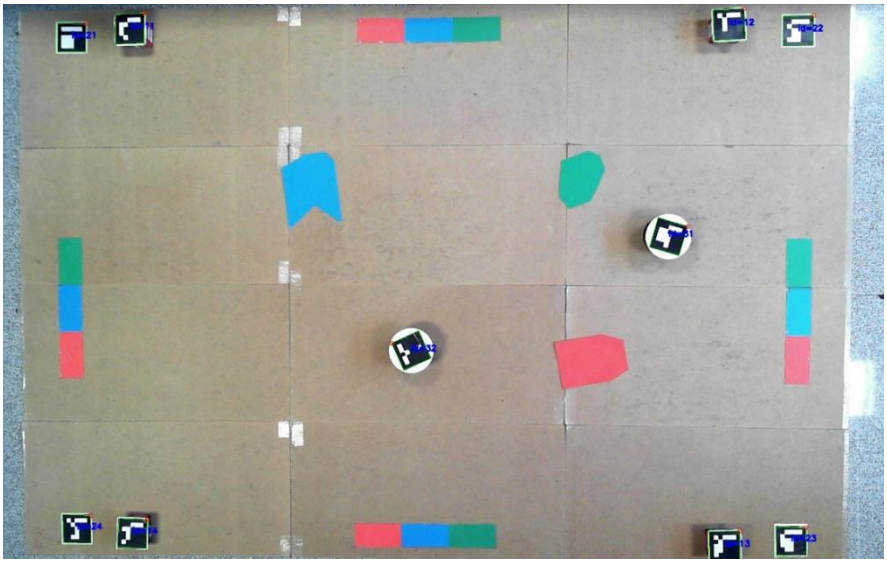

Figura 5. Imagen plataforma de robots móviles del laboratorio.

4.2 Ficheros.

Esta primera parte del proyecto trata de a generar un mapa para el simulador Stage a partir de los datos de entrada. Por lo tanto, se va a trabajar con dos ficheros principales, el de entrada y el de salida. Los datos de entrada comentados anteriormente se dispondrán en un fichero de texto con una estructura específica.

De la misma forma el fichero de salida se escribirá como un fichero de texto con la estructura que debe tener un archivo de mapa en Stage para su posterior simulación. El objetivo es desarrollar un programa escrito en lenguaje C++ que consiga a partir del fichero de entrada generar otro fichero con el contenido que define el mundo o escenario de la simulación de Stage. En la Figura 6 se esquematiza el planteamiento seguido.

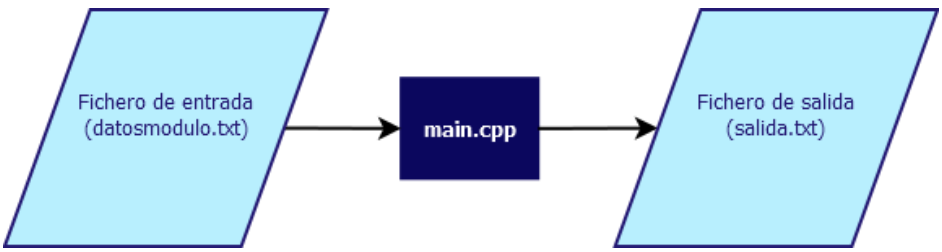

Figura 6. Esquema de archivos.

4.2.1 Fichero de entrada.

En este apartado se explica la estructura que va a tener el fichero de entrada ejemplificándolo con un primer caso de simulación, el escenario de la Figura 5. Dicha estructura es un aspecto importante para la creación de la programación.

En primer lugar, se especifican los parámetros referidos a la plataforma como son las dimensiones de está, quedando definido el ancho y el largo de la misma en metros.

```
Salida del modulo de vision
ESCENARIO:
Ancho: 2.34
Largo: 3.32
```

A continuación, se detallan los datos comentados anteriormente sobre los robots, es decir, se especifica el radio de los mismos y las coordenadas de sus posiciones iniciales. Esta información acerca de los robots viene dada en centímetros.

```
----Initial Position of Robots(x(cm), y(cm), rad)----
Radio: 9
Robot[1]=(249,124,2.85286)
Robot[2]=(163,75,-1.16488)
```

En este caso hay dos robots en el escenario, sin embargo, se diseñará el programa de forma que se puedan disponer cualquier otro número de robots dándole versatilidad y generalidad al *software* creado.

Por último, se tiene la información requerida sobre los distintos obstáculos. Las coordenadas de las esquinas también vienen dadas en centímetros.

```
---------List of obstacles (Format -->[x,y])---------
Obstaculo 1
           color: azul
           esquina[1]: (88,156)
           esquina[2]: (98,162)
           esquina[3]: (107,163)
           esquina[4]: (111,158)
           esquina[5]: (114,132)
           esquina[6]: (92,129)
```

Esta estructura en la que se indica el obstáculo, su color y coordenadas se repite tantas veces como objetos haya. Al igual que ocurre con los robots, tanto el número de obstáculos como el número de esquinas de cada uno es aleatorio por lo que se diseñará el programa teniendo en cuenta que el número de estos no es conocido de antemano.

Una vez conocida la información de partida y la estructura en la que se dispone se procede a explicar el contenido del fichero de salida que se desea obtener.

4.2.2 Fichero de salida.

Como se ha indicado el fichero de salida será el que se utilizará como archivo de mapa en Stage por lo que en él se definen todos los componentes del escenario de simulación. Un ***worldfile*** es básicamente una lista de modelos que definen todos estos componentes como son

los robots, obstáculos, paredes… La forma de definir estos modelos se muestra en la siguiente estructura.

```
define model_name model
(
# parameters
)
```

A continuación, se explican los modelos que serán necesarios para la definición de la plataforma y del escenario de simulación.

En el *worldfile* es posible configurar la ventana de simulación, **window**, especificando los parámetros de tamaño (*size*) en pixeles y escala (*scale*). Es decir, la escala que se especifique serán los metros a los que equivale cada *pixel*.

Una vez definida la ventana de simulación se pueden ir creando los diferentes modelos necesarios.

- **Floorplan model.**

  El modelo *floorplan* describe el escenario básico de la simulación. En su definición se pueden incluir distintos parámetros como los que se definen a continuación.

  1. Color.

  2. Boundary: parámetro binario con el que definir o no un cuadro delimitado para el mapa de simulación. Si se establece el valor 1 por tanto se definirá la limitación que los robots no podrán sobrepasar.

  3. Gui_nose: indica la orientación del modelo.

  4. Gui_grid: dispone o no una cuadrícula sobre el modelo.

  5. Fiducial_return: como cualquier parámetro que lleve añadido *return* habilita al sensor que hace referencia, en este caso al sensor de tipo fiducial.

  6. Ranger_return: establecer este parámetro a 1 permite que el modelo pueda ser identificado por un *ranger sensor*.

- **RobotP position.**

  Este modelo definirá el robot a simular, en él se especificará la forma, el tamaño y los sensores que tenga. Existen dos métodos para describir la forma de un modelo, mediante el método *block* o insertando un *bitmap*. En el caso de usar ***block model*** la forma se define estableciendo el número de esquinas y las coordenadas de cada una. En el caso de utilizar ***bitmap*** simplemente se incluye el nombre del *bitmap* con la forma a utilizar que puede ser un archivo de imagen png.

```
block                                   bitmap "circulo.png"
(                                       size [0.09 0.09 0.02]
        points 6
        point[0] [2.58 0.36]
        point[1] [2.64 0.5]
        point[2] [2.72 0.65]
        point[3] [2.93 0.54]
        point[4] [2.84 0.34]
        point[5] [2.79 0.26]
        z [0 1]
)
```

Dado que los robots utilizados en la plataforma tienen forma circular es más adecuado utilizar un *bitmap* para su definición. En ambos casos se especifica el tamaño con el parámetro *size*.

Por otro lado, en este modelo se especifica el parámetro *drive* que establece el cómo es conducido el robot, se utiliza **"diff"** en este caso lo que significa que el robot simulado se controla mediante cambios de velocidad independientes para sus ruedas derecha e izquierda. Existen otras dos posibilidades en cuanto al control de los robots, una es **"car"** en la que el robot utiliza la velocidad y un ángulo de conducción. La otra opción es **"omni"** con la que el robot puede moverse a lo largo de los ejes x e y en la simulación.

Por último, en este modelo se añaden los sensores que se quieran incluir en la simulación del robot. Los sensores que se incluyan han debido ser definidos previamente, en este caso se incluye el modelo *RPsonar* que es un sensor tipo ***ranger*** el cual se explica en el siguiente punto.

```
define robotP position
(
        bitmap "circulo.png"
        size [0.18 0.18 0.09]
        origin [-0.050 0.000 0.000 0.000]
        gui_nose 1
        drive "diff"
        RPsonar(pose [0.18 0 0 0])
)
```

- **RPsonar ranger**.

  Un *ranger* simula distintos dispositivos de detección de objetos como pueden ser láseres, sonars o infrarrojos. Este sensor puede funcionar en modelos en los que el parámetro *ranger_return* está establecido a 1 como en este caso se ha señalado en el modelo *floorplan* visto anteriormente.

A continuación, se muestra el *ranger* que se define para los robots utilizados en esta simulación y se explican brevemente los parámetros que componen el modelo.

```
define RPsonar ranger(
  sensor(
          range [ 0.0 10.0]
          fov 270
          samples 1081
  )
  color "black"
  size [0.045 0.045 0.09]
)
```

- Sensor: se define su rango (*range*) estableciendo el mínimo y el máximo alcanzable a medir. Se define el campo de visión en grados mediante el parámetro *fov*. *Samples* se define únicamente si se utiliza un láser como es el caso.

- Por otro lado, se definen los parámetros *color* y *size* que ya se han visto en otros modelos.

- **Object model.**

  Se crea un modelo llamado *object* para definir los obstáculos, la definición de la forma de los objetos se realiza por el método block comentado anteriormente. Una vez descritas las esquinas se establece el tamaño y color del objeto con *size* y *color*.

  Dado que cada obstáculo tiene una forma aleatoria y distinta a los demás con distinto número de esquinas, se crea un modelo de objeto para cada uno.

4.2.2.1 Posicionamiento y dimensión de los modelos.

Una vez definidos todos los modelos que se requieren, estos deben ser posicionados en el escenario de simulación. Para ello se escribe el nombre del modelo y mediante la orden *pose* se establecen las coordenadas donde aparecerá situado el modelo. De esta forma es posible posicionar en el mapa varios objetos del mismo modelo simplemente repitiendo esta estructura.

Por ejemplo, en el caso del modelo definido para el robot si queremos simular varios, basta con repetir la siguiente fórmula.

```
robotP
(name "R1" pose [2.65 1.1 0 -2.17839] color "black")

robotP
(name "R2" pose [0.51 0.89 0 1.58025] color "black")
```

Así en la simulación aparecerá un "robotP" en cada una de las coordenadas especificadas con *pose*.

Un aspecto importante a tener en cuenta es el posicionamiento del modelo *floorplan*. Este modelo es el que sitúa el rectángulo que conforma los límites de la plataforma y tiene unas

dimensiones de 2,34 x 3,32 metros. Dado que los datos de entrada sobre las posiciones de los obstáculos están referidas al origen de coordenadas de la plataforma, que se sitúa en la esquina inferior izquierda del rectángulo, se posiciona el modelo *floorplan* de forma que la esquina inferior izquierda del rectángulo coincida con el origen de coordenadas en la simulación.

```
floorplan
(
        name "willow"
        bitmap "rectangulo.png"
        size [ 3.32 2.34 0.25 ]
        pose [ 1.66 1.17  0 0 ]
)
```

De esta forma se consigue que al posicionar los objetos estos queden bien referenciados.

En relación con la posición y el tamaño del modelo *floorplan* se establecen las dimensiones y el origen de la ventana de simulación. De forma que al lanzar Stage la ventana aparezca con la vista adecuada y no sea necesario ajustarla cada vez que se ejecuta una simulación. Se hace coincidir el centro de la ventana con el centro de la plataforma.

```
      window
(
        size [ 600 600 ]
        scale 160
        center [1.66 1.17 ]
        rotate [ 0.000  0.000 ]
        show_data 1
)
```

En cuanto a los objetos, estos vienen descritos por las coordenadas de sus esquinas, sin embargo, en Stage hay que especificar su tamaño redimensionándolos. Para establecer el tamaño se cogerán las distancias máximas en ambas coordenadas x e y. Estas distancias serán las que definan el parámetro *size* de cada objeto.

Por ejemplo en el siguiente caso se cogen la x máxima y mínima y se calcula la distancia entre ellas, 0,35 , este es el valor del tamaño del objeto en la coordenada x.

```
define object1 model
(
        block
        (
                points 6
                point[0] [2.58 0.36]
                point[1] [2.64 0.5]
                point[2] [2.72 0.65]
                point[3] [2.93 0.54]
                point[4] [2.84 0.34]
                point[5] [2.79 0.26]
                z [0 1]
        )
        size [0.35 0.39 0.1]
        color "blue"
)
```

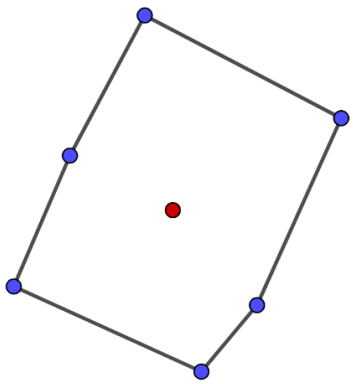

Figura 7. Representación gráfica del modelo object1.

Para su posicionamiento se calcula la media de las coordenadas de las esquinas tanto para x como y. En el caso expuesto resultaría como se muestra a continuación.

```
object1( pose [2.75 0.441667 0 0 ])
```

4.3 Desarrollo del programa para la creación del mapa de simulación.

Descritos ya los ficheros con los que se trabaja y la información que estos contienen se procede a la creación del programa, que a partir del fichero con los datos de entrada genere el archivo de salida que constituirá el mapa para su posterior simulación en Stage.

La programación se va a llevar a cabo en C++, el programa a desarrollar será básicamente un programa de lectura y escritura de ficheros ya que el objetivo principal es leer el fichero de entrada con los datos de la plataforma y procesarlos para obtener de manera automática el archivo que servirá como *worldfile*.

Al tratarse de ficheros de texto se ira leyendo el archivo de entrada carácter a carácter en su mayor parte. Para este desarrollo hay que tener en cuenta las estructuras de los distintos ficheros explicadas anteriormente.

4.3.1 Principales recursos de C++ utilizados.

Inicialmente es importante declarar adecuadamente los ficheros. Cada uno de ellos se abre de forma diferente ya que uno es para lectura y el otro para escritura. En este caso se utilizan las ordenes:

```
ifstream datos("datosmodulo", ios::in);
ofstream mapa("salida", ios::out);
```

Código 6. Declaración de ficheros en C++.

De esta forma se crean unos ficheros ficticios llamados “datos” y “mapa” que serán los nombres utilizados de ahora en adelante durante la programación. De manera que si se quiere cambiar el fichero de entrada por otro solo sería necesario cambiar el nombre en la declaración del fichero *ifstream* u *ofstream*.

En cuanto a la lectura del fichero datos se emplean dos herramientas que son ***get*** para leer carácter a carácter y ***getline*** que será utilizada para saltar las líneas que no contengan información a procesar.

En la escritura de “mapa” mayormente se hará uso de funciones para escribir la descripción de cada modelo como *window*, *floorplan* o la definición del modelo *RobotP*. De esta forma se consigue una estructura de programa más organizada.

Para procesar y escribir adecuadamente la estructura de “mapa” se utilizarán bucles *while* y estructuras condicionales de forma que se ira escribiendo en el fichero de salida conforme se va leyendo la información de “datos”.

4.4 Implementación en ROS.

Llegados a este punto se tiene ya una forma automatizada de convertir los datos de la plataforma en un archivo con la estructura adecuada. El siguiente paso es lanzar la simulación y para ello se va a crear un *launch file* en el que se lance el simulador con el mapa deseado. A este archivo se le dará el nombre de **stage.launch**.

Anteriormente se ha explicado la estructura que tienen los archivos tipo *launch* y que su objetivo es lanzar varios nodos a la vez. En este caso los nodos a utilizar son /stageros y /teleop_twist_keyboard.

- **Stage_ros** es el nodo que ejecuta el simulador de Stage. Lo importante es definir correctamente el parámetro *args* ya que marca la diferencia entre lanzar un mapa u otro. Por tanto, hay que especificar la dirección en la que se encuentra el *worldfile* que se quiere simular.

- **Teleop_twist_keyboard** es un nodo que permite el control de los robots simulados mediante el teclado y que se usa en estas primeras simulaciones para comprobar el correcto funcionamiento de las mismas.

```
<launch>

  <node name="stageros" pkg="stage_ros" type="stageros"
output="screen"
args="/home/osboxes/catkin_ws/src/first_pack/WorldTFG/worldp
rueba1.world"/>

<group ns="robot_0">
  <node name="teleop_twist_keyboard"
pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py"
output="screen" launch-prefix="xterm -e"/>
</group>

<group ns="robot_1">
  <node name="teleop_twist_keyboard"
pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py"
output="screen" launch-prefix="xterm -e"/>
</group>


</launch>
```

Código 7. stage.launch.

Cabe destacar que al simular varios robots se pueden generar problemas para su manejo mediante *teleop_twist_keyboard* por eso se genera un grupo para cada robot de forma que se lanza una terminal para el manejo independiente de cada uno.

4.4.1 Simulación.

Se simula la plataforma con tres configuraciones diferentes para comprobar la efectividad del programa creado y se muestra el resultado en las siguientes figuras.

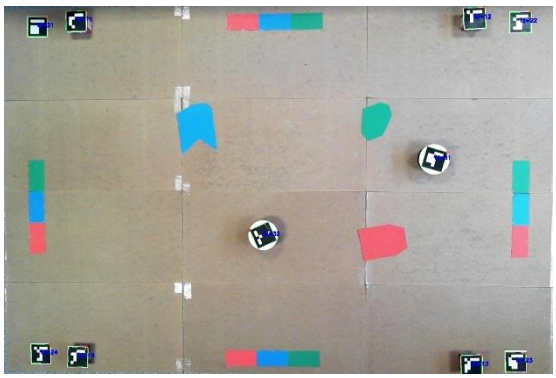

Figura 8. Plataforma real Caso 1.

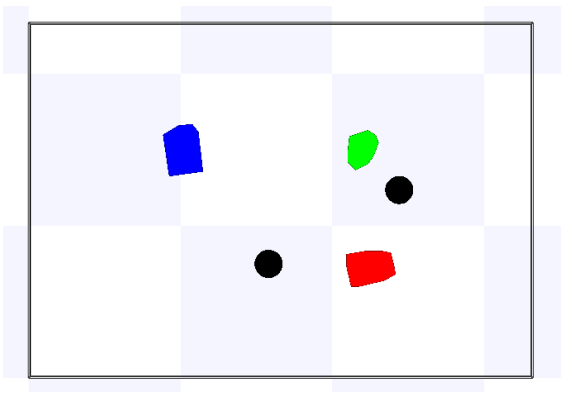

Figura 9. Plataforma simulada Caso 1.

Figura 10. Plataforma real Caso 2.

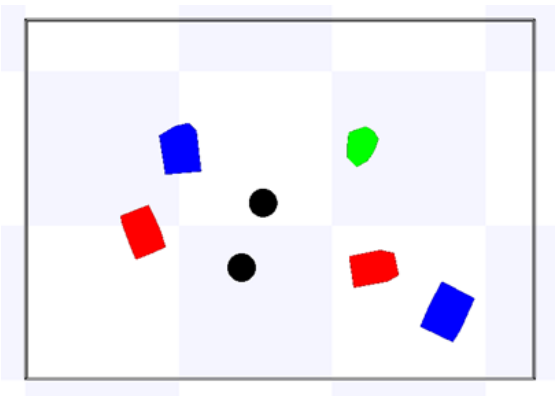

Figura 11. Plataforma simulada Caso 2.

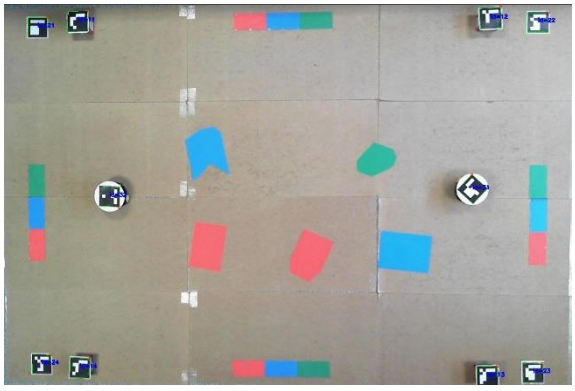

Figura 12. Plataforma real Caso 3.

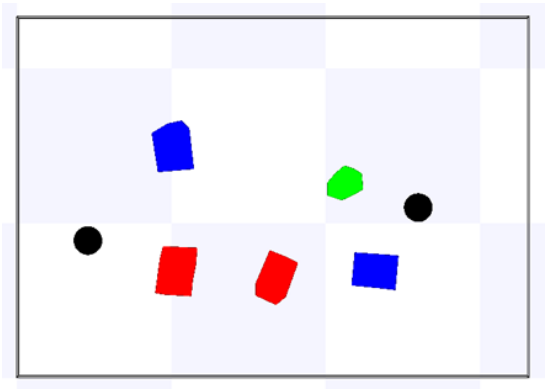

Figura 13. Plataforma simulada Caso 3.

En la simulación aparecen los robots con su forma circular y en color negro para distinguirlos de los obstáculos. Se sitúan en cada caso en sus posiciones iniciales ya que este es un dato de entrada.

Cabe destacar que pueden existir discrepancias en cuanto a la forma real y la forma simulada de los obstáculos debido al número de esquinas. Esto se debe a pequeños errores en la toma de datos por parte del algoritmo de visión que utiliza la cámara de la plataforma real que no están dentro del alcance de este trabajo. Sin embargo, el número de esquinas y la forma de la simulación si se corresponde con los datos de entrada proporcionados que es el objetivo que se ha planteado durante el proyecto.

Mediante la herramienta **rqt_graph** de ROS podemos visualizar los nodos y *topics* que están activos en esta simulación como muestra la Figura 14, lo que será de utilidad más adelante.

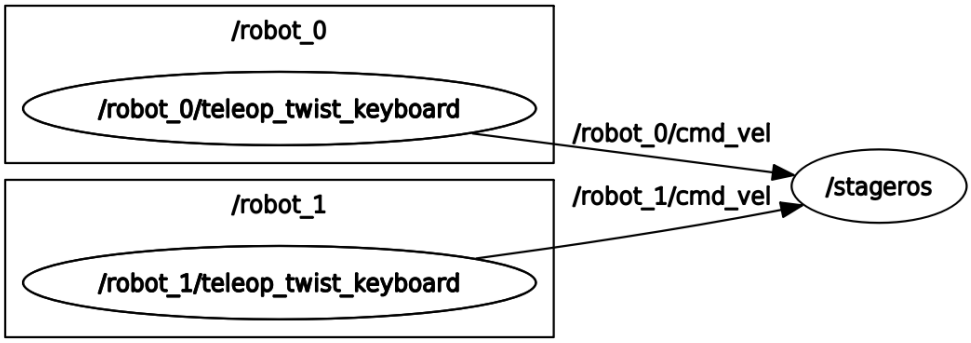

Figura 14. Esquema de nodos y topics activos.

Los *topics* del tipo **cmd_vel** se corresponden con comandos de velocidad. Como se puede apreciar al simular dos robots se añade un prefijo con el nombre del robot a cada uno de los *topics* que le corresponde en este caso ocurre con /robot_0/cmd_vel y /robot_1/cmd_vel .

Los archivos y el código creados en este capítulo y en los siguientes se recogen en una carpeta de Drive [16].

Capítulo 5.
Navegación autónoma en simulación.

En este capítulo se va a abordar el objetivo final del proyecto que es la navegación autónoma de los robots simulados.

Para conseguir que los robots simulados alcancen unas posiciones deseadas se va a hacer uso del paquete *navigation stack* de ROS. Esta herramienta consigue enviar un robot de un punto a otro a partir de la información y los datos que el robot tiene sobre el entorno.

En primer lugar, se explicará una visión general de lo que es el paquete *navigation stack*, cómo funciona y que información utiliza. Una vez claro, se aborda la configuración del paquete para adaptarlo a la simulación de este trabajo. De nuevo se hará uso de un archivo tipo *launch* que permita lanzar los nodos con las configuraciones apropiadas.

5.1 Navigation stack.

El paquete *navigation stack* recoge la información de la que dispone el robot y la transforma en comandos de velocidad. Esta información que utiliza se trata principalmente de la odometría y la proporcionada por los sensores de los que disponga.

El esquema de la Figura 15 muestra un resumen de la información que maneja el paquete y de la configuración que sigue. Además, se ven los *topics* que utiliza para transmitir dicha información y el tipo de mensajes que trata.

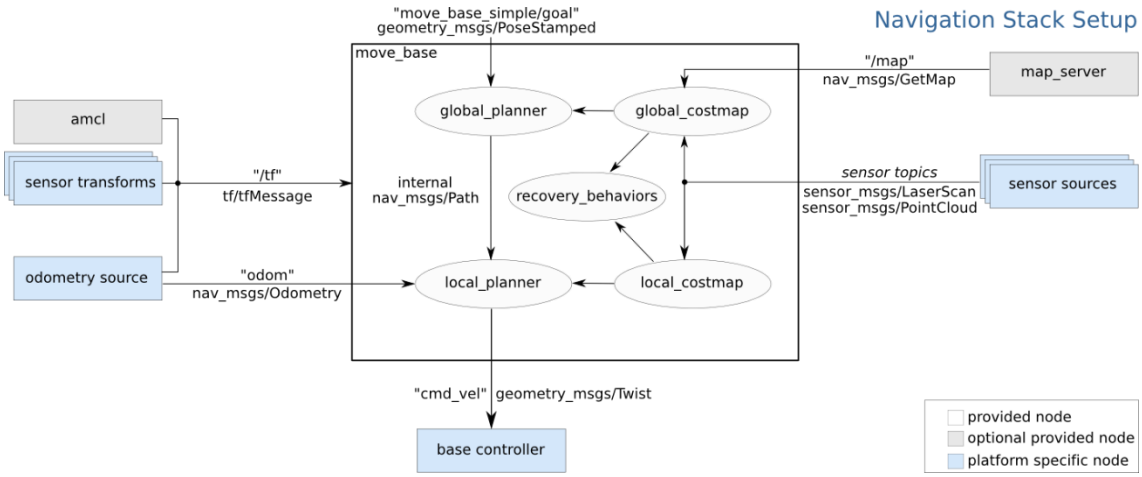

Figura 15. Esquema general del funcionamiento de navigation stack. [11]

Para el correcto funcionamiento de este paquete es necesario que los nodos que aparecen en azul estén adecuadamente configurados. Se va a explicar uno por uno en que consiste la información de estos nodos y cuál es su función en la navegación.

5.1.1 Transformaciones. (*sensor transforms*).

En una simulación con robots móviles existen varios **sistemas de referencia**. Para poder utilizar la información de forma correcta en ocasiones será necesario realizar transformaciones de un sistema de referencia a otro.

Trabajando con ROS estas transformaciones se pueden realizar de forma sencilla y automática mediante el paquete **tf**. Conceptualmente con esta herramienta lo que se hace es crear un diagrama llamado "*transform tree*" en el que cada nodo se corresponde con uno de los sistemas de referencia y las flechas indican el sentido de estas transformaciones. Dicho sentido va siempre desde los nodos que se consideran "padres" a los nodos llamados "hijos".

A modo de ejemplo en el robot que se ha simulado en Stage se ha incluido un sensor *ranger* de tipo láser por lo que existirá un sistema referido al laser (base_laser_link) y otro al centro del robot (base_link). La herramienta tf permite visualizar el diagrama de sistemas de referencia mediante el siguiente comando.

```
rosrun tf view_frames
```

Código ROS 5. Visualizar transform tree.

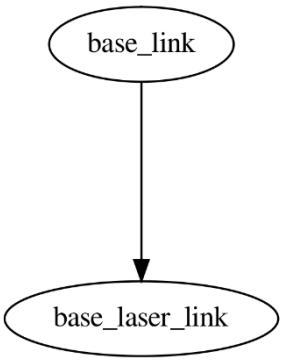

Figura 16. Transform tree.

Como se ve en el diagrama de la Figura 16 en este caso base_link es el padre y base_laser_link está definido como el hijo. De esta forma se consigue convertir los datos obtenidos en el sistema de referencia del láser al sistema situado en el centro del robot. Esto permite al robot usar esta información correctamente para poder evitar los obstáculos del entorno simulado.

Cabe destacar que este pequeño diagrama es solo una parte del diagrama completo de la simulación que se compone de más sistemas de referencia como se muestra en el Anexo IV (Figura 30).

5.1.2 Información de sensores. (*sensor sources*).

Los sensores son la principal fuente de información de los robots sobre el entorno que les rodea. Es importante que los datos que toman estos sensores sean transmitidos al nodo de navegación. Estos son publicados como un tipo concreto de mensajes en ROS que son sensor_msgs/LaserScan o sensor_msgs/PointCloud.

5.1.3 Información sobre odometría. (*odometry source*).

La odometría es, como se ha dicho, otra de las principales fuentes de información para hacer posible la navegación. La odometría se define como la estimación de la posición y la velocidad de un vehículo con ruedas como son los robots móviles.

El nodo de navegación necesita conocer la posición y velocidad del robot en el mapa simulado. Este tipo de información es publicada en ROS mediante mensajes del tipo nav_msgs/Odometry. Estos mensajes se publican en el *topic* **/odom**. Además, cabe destacar que existe un sistema de referencia llamado *odom* que es a donde está referida la información de este tipo de mensajes.

5.1.4 Controlador. (*base controller*).

En la navegación es necesario transmitir los comandos de velocidad que se generan al planificar las trayectorias al robot. La función del controlador es coger las velocidades publicadas en /cmd_vel y transformarlas en comandos para el motor del robot, en este caso serán comandos para el robot simulado. Los mensajes de velocidad son del tipo geometry_msgs/Twist y deben estar referenciados al sistema del propio robot para que este tome las velocidades deseadas.

El simulador Stage dispone de sus propios controladores por lo que no es necesario utilizar otros nodos de ROS que funcionen como controlador del robot. En la propia definición del modelo de robot en el *worldfile* se especifica el parámetro *drive* que establece el tipo de control y se comentan las otras dos opciones “car” y “omni”. En este caso se ha utilizado un control diferencial en el que el robot móvil se maneja mediante la conducción de sus dos ruedas independientemente.

5.2 *Launch* de navegación.

En el apartado anterior se ha visto el funcionamiento general del nodo move_base y la información que maneja el paquete de navegación.

Analizada una visión general de dicho paquete se pasa a configurar el archivo *launch* y los parámetros necesarios para un correcto funcionamiento de la simulación. Para una mayor simplicidad se realizan las pruebas con un solo robot.

Al crear un archivo *launch* lo primero que hay que tener claro es qué nodos se quieren ejecutar. Como muestra el esquema de la Figura 15 el nodo principal de navegación es el conocido como **move_base**, además habrá que ejecutar los nodos **amcl** y **map_server**.

Por otro lado, en este caso se trabaja con una simulación de Stage por lo que al igual que se ha hecho en el capítulo 4 (4.4) se incluye el nodo de stage_ros que tendrá como argumento el mapa con el que se trabaja. El nodo de Stage es el que va a proporcionar la información explicada anteriormente, odometría, datos de sensores e información sobre los distintos sistemas de referencia.

Se va a explicar nodo por nodo los parámetros importantes que debe incluir cada uno y que se deben configurar adecuadamente. El conjunto de todos ellos forma el *launch* final que se llamará **nav_stage.launch**.

5.2.1 Stage.

El nodo de Stage se configura básicamente como se ha hecho en el Capítulo 4.

```
<!-- STAGE -->
<node name="stage_ros" pkg="stage_ros" type="stageros"
args="/home/osboxes/catkin_ws/src/first_pack/WorldTFG/worldprueba1.w
orld" output="screen">
        <remap from="base_scan" to="scan" />
</node>
```

Código 8. Fragmento de nav_stage.launch definiendo el nodo de Stage.

A lo largo de la configuración de los distintos nodos se realizan varios remapeados de *topics*. Esto se debe a que en Stage el *topic* que transmite por ejemplo los datos del láser es nombrado como base_scan mientras en el nodo move_base esta información se publica mediante un *topic* llamado scan.

5.2.2 Map.

El segundo nodo a incluir es el de map_server. Este nodo proporciona el mapa necesario para la navegación mediante un topic llamado **/map**. Para la creación de este mapa se hace uso del paquete **gmapping** con el que se genera el mapa del entorno en pgm y un archivo YAML que describe los meta datos del mapa. Este paso queda reflejado en el Anexo III (Figura 29).

En adelante se hace un constante uso de los archivos tipo YAML para la definición de parámetros. Al incluir el nodo en nav_stage.launch se pasa como argumento el archivo **map2.yaml** con los datos del mapa de simulación.

```
<!-- MAP -->
<node name="map_server" pkg="map_server" type="map_server"
args="/home/osboxes/catkin_ws/src/map2.yaml"/>
```

Código 9. Fragmento de nav_stage.launch definiendo el nodo map_server.

5.2.3 Move Base.

El nodo de navegación move_base está compuesto por varios nodos a su vez como se puede apreciar en el esquema de la Figura 15.

5.2.3.1 Costmap2d: global costmap y local costmap.

Tras incluir en el *launch* el nodo move_base se establece la configuración de los *costmaps*. El paquete **costmap_2d** es un elemento importante en la navegación de los robots. Su función es proporcionar información acerca de por dónde puede navegar el robot. Para ello, utiliza los datos proporcionados por los sensores y forma una red de ocupación con el objetivo de tener información sobre los obstáculos y límites del entorno.

Se tratada de una red en dos dimensiones, configurable y formada por celdas. A su vez existen distintas capas según la funcionalidad, por ejemplo, una capa se corresponde con el mapa estático mientras otra se encarga de la información sobre los obstáculos. La capa dedicada

a los obstáculos puede recoger información sobre los mismos en 3D por lo que estos se almacenarán en columnas que luego se proyectan en la red bidimensional del *costmap*.

A cada celda se le da un valor (*cost value*) en función del cual se marcan como ocupadas, libres o desconocidas. En base a esto se asigna un coste en la proyección en dos dimensiones del *costmap* que acaba marcando la celda como *lethal obstacle*, *no information* o *free space*. Esta información se va actualizando con una determinada frecuencia y la red creada va variando.

Existe otra capa de datos llamada "*inflation layer*" que añade valores a las celdas que se encuentran alrededor de un obstáculo, es decir, alrededor de las celdas marcadas como *lethal obstacle*. Esta capa por tanto hace que el robot se mantenga a una cierta distancia de los obstáculos en su trayectoria.

Por otro lado, como se ha expuesto en la introducción de este trabajo, en la navegación se distinguen los conceptos global y local refiriéndose el global a todo el entorno mientras el mapa local se encarga de las zonas más próximas al robot.

El **global_costamap** será el responsable de planificar trayectorias teniendo en cuenta el escenario completo mientras el **local_costmap** se centra en evitar obstáculos que están en los alrededores más próximos del robot. En este aspecto existen parámetros de configuración comunes a ambos mapas y otros específicos de cada uno, de ahí los distintos archivos que se incluyen y que se nombran a continuación.

En nav_stage.launch se cargan distintos archivos YAML que contienen los parámetros que configuran y definen el funcionamiento del paquete *costmap*.

- costmap_common_params.yaml.
- local_costmap_params.yaml.
- global_costmap_params.yaml.

Además, en el *launch* se establecen parámetros como los sistemas de referencia que van a manejar tanto global_costmap como local_costmap.

5.2.3.2 Planners.

El otro punto clave en la navegación es la planificación de las trayectorias. El paquete **base_local_planner** es el que proporciona el controlador que el conduce robot a lo largo de las trayectorias.

A partir de los datos del mapa se crea una trayectoria cinemática que el robot debe seguir para moverse desde su posición inicial hasta el punto deseado. El paquete utiliza el sistema de celdas de ocupación que crea el *costmap* para definir sus trayectorias. El coste asignado a cada celda es un valor de lo que costaría atravesar esa posición, en base a estos datos se determinan las velocidades que debe seguir el robot.

Se incluye un archivo YAML con los parámetros del dwa_local_planner que es algoritmo encargado de calcular dichas trayectorias.

5.2.4 AMCL.

El paquete AMCL funciona como un sistema probabilístico de localización para robots móviles en dos dimensiones. Su objetivo es proporcionar la posición del robot en el mapa a través de los datos del sensor laser e información sobre las transformaciones entre sistemas de referencia.

Finalmente se representa en la Figura 17 el gráfico de nodos y *topics* que resultaría de la ejecución de una simulación básica de navegación con un solo robot.

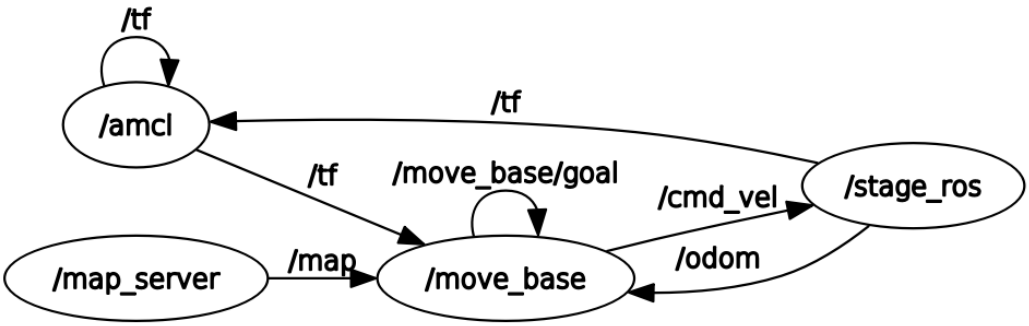

Figura 17. Esquema rqt_graph navegación de robot móvil.

5.3 Simulación de navegación con un solo robot.

Finalmente, teniendo el archivo *launch* definido se puede pasar a la simulación y ver el resultado obtenido. La zona sombreada de la Figura 18 corresponde con el área que visualiza el sensor láser del robot.

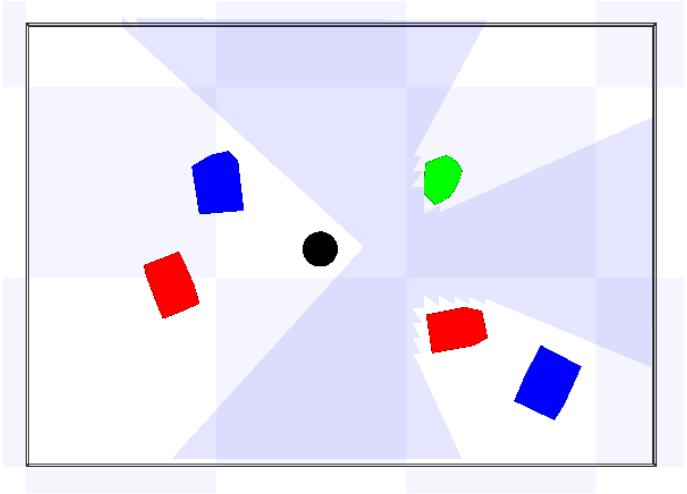

Figura 18. Simulación en Stage.

Esta primera prueba de simulación se lanza con los parámetros predeterminados en el nodo move_base por lo que el resultado no es el adecuado ya que al mandar el robot a un objetivo este no puede alcanzar su posición. Esto se puede apreciar visualizando la simulación en Rviz como en la Figura 19.

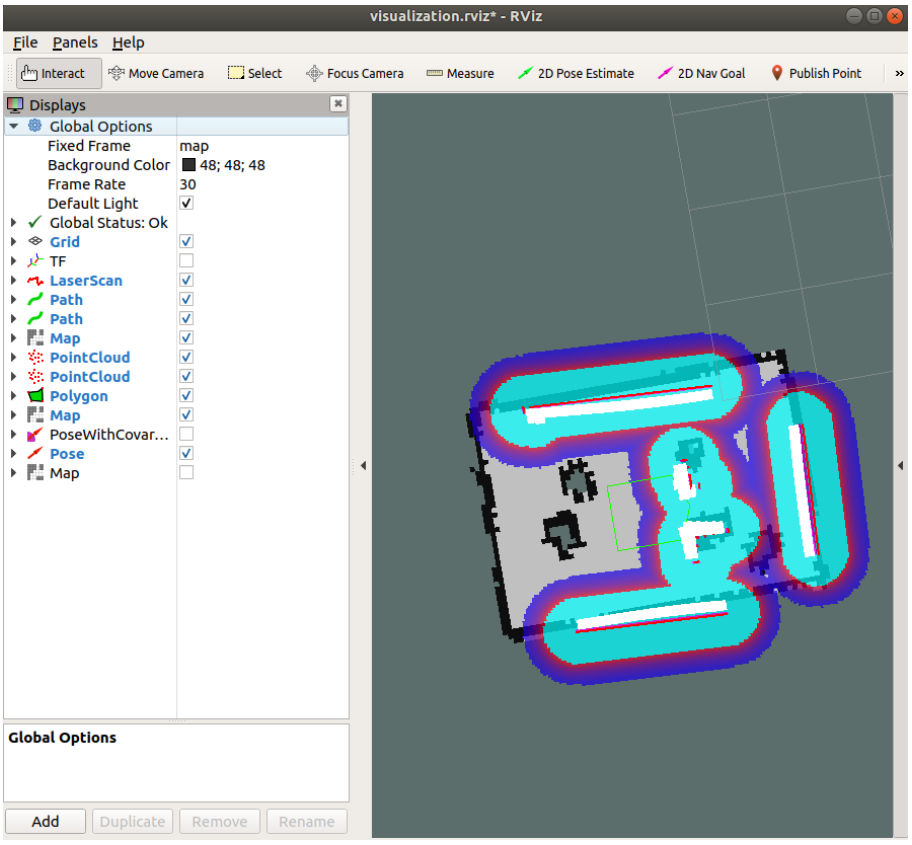

Figura 19. Visualización ventana Rviz.

Mediante la vista en Rviz se observan varias cosas que pueden suponer un problema. En primer lugar, la forma y tamaño del robot representada como un polígono de color verde no se corresponde con la de la simulación en Stage.

Por otro lado, otro de los aspectos importantes son los mapas representados. Anteriormente se han explicado los conceptos de *global costamp* y *local costmap*. En las siguientes figuras se muestran tres mapas, el publicado por el map_server **map,** el **local costmap** y el **global costmap.**

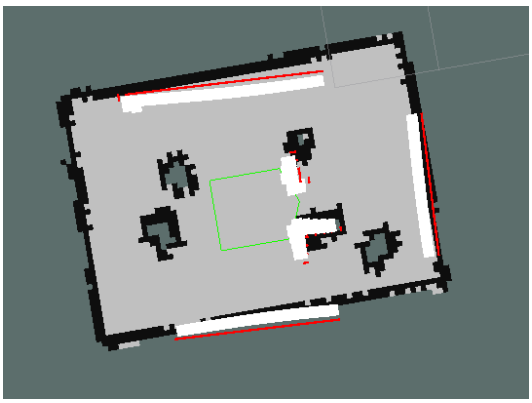

Figura 20. Visualización map inicial.

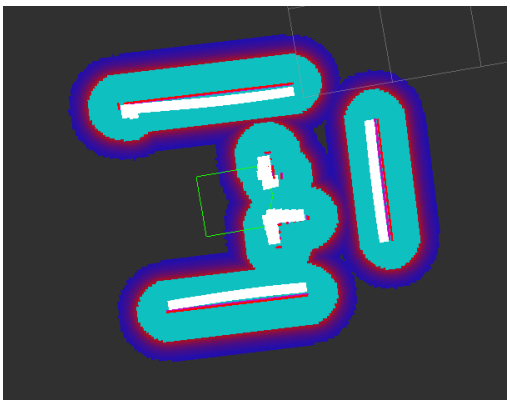

Figura 21. Visualización local costmap inicial.

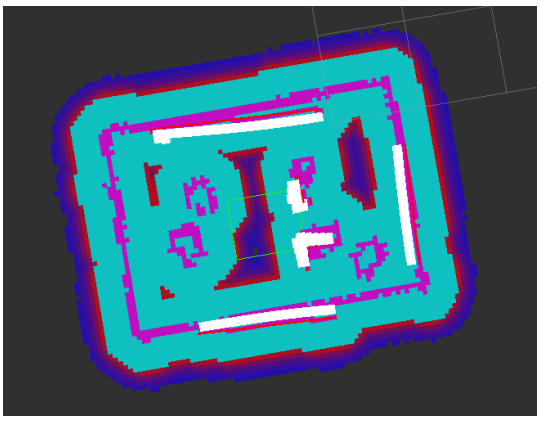

Figura 22. Visualización global costmap inicial.

Lo que se puede apreciar es que el robot percibe los obstáculos y límites del entorno mucho más grandes de lo que son de forma que no puede crear una trayectoria porque piensa que no tiene espacio suficiente para atravesar los obstáculos. Esto es fruto del sistema de celdas explicado en el apartado anterior.

Estos problemas se pueden solucionar modificando los parámetros de los nodos move_base y amcl en el archivo *launch*.

5.3.1 Configuración dinámica de parámetros.

En este punto se modifican una serie de parámetros para terminar de configurar la navegación de forma que funcione correctamente en el entorno que se ha simulado. Estos parámetros se definen en el archivo *launch*.

Sin embargo, para ir ajustando y variando los parámetros se va a utilizar la herramienta **rqt_reconfigure** de ROS. Ejecutando rqt_reconfigure se ven en una misma ventana todos los parámetros que se pueden modificar. La ventaja que proporciona esta herramienta es que se pueden ir realizando sucesivas pruebas y variando distintos parámetros mientras la simulación está en marcha. De otra forma cada vez que se quisiera cambiar un campo habría que modificarlo en el *launch* y por tanto volver a ejecutar la simulación.

```
rosrun rqt_reconfigure rqt_reconfigure
```

Código ROS 6. Comando para ejecutar rqt_reconfigure.

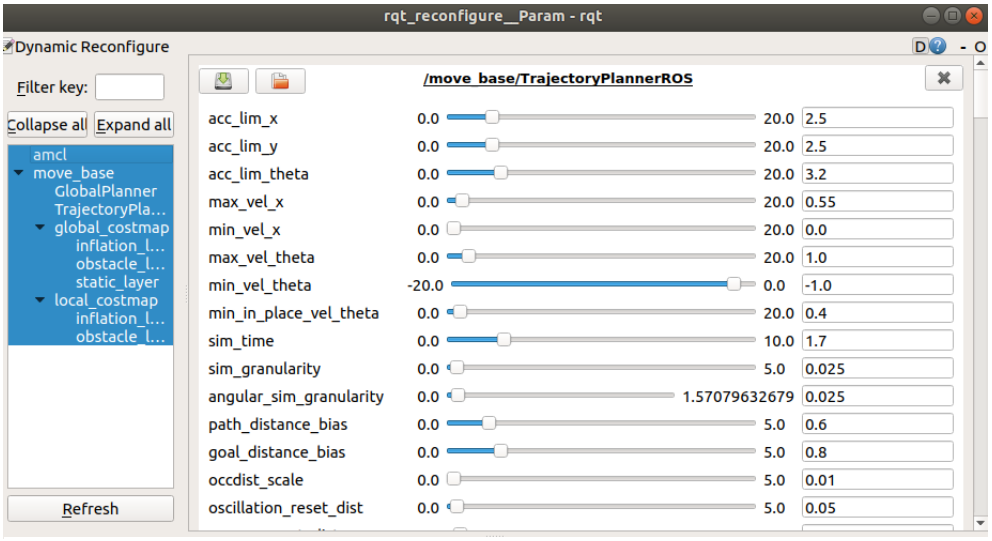

Figura 23. Vista ventana rtq_reconfigure.

La Tabla 1 muestra los parámetros que se han modificado, el valor que se establece y una breve descripción de cada uno.

PARÁMETRO	DEFINICIÓN	VALOR
footprint	Se define como una serie de puntos que establecen la forma del robot. Al trabajar con un robot circular habrá que definirlo a 0.	0
robot_radius	Radio del robot en metros.	0,09
inflation_radius	Radio en metros en el que el mapa incrementa el cost value de los obstáculos, es decir, lo que estos ocupan.	0,12
cost_escaling_factor	Factor de escalado que se le aplica a los valores de coste (cost values)	10
controller_frequency	Ratio en Hz en el que se mandan comandos de velocidad al robot.	5

Tabla 1. Parámetros ajustados.

A excepción de la frecuencia el resto de parámetros deben ser establecidos tanto para el *local costmap* como para el *global costmap*.

Otros parámetros que se han tenido en cuenta son el tamaño de los obstáculos observables por el láser. Tanto a nivel local como global en la definición de base_scan en el nodo de navegación se establece que los obstáculos que el láser detecta deben estar entre 0,08 y 0,4 metros de altura. En este caso en vez de variar esos parámetros nos aseguramos de que los obstáculos y los límites del entorno definidos en Stage estén dentro de ese rango.

5.3.2 Resultados.

Las pruebas de navegación realizadas se recogen en vídeo en una carpeta Drive https://drive.google.com/drive/folders/1HaMmxosDbpX7cwbYif07b8W713QPHaD-?usp=sharing.

Se ejecuta de nuevo la simulación con los parámetros modificados y se visualiza el resultado en Rviz como muestran las Figuras.

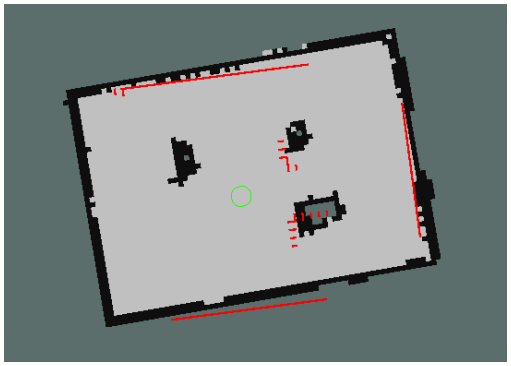

Figura 24. Visualización map ajustado.

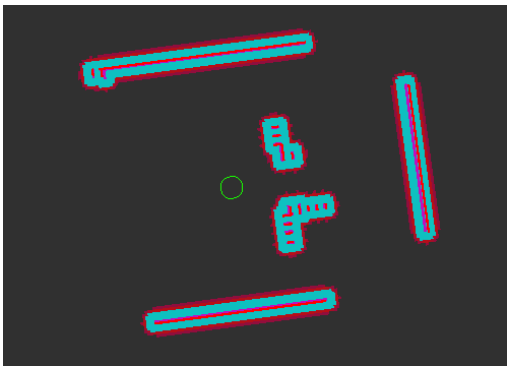

Figura 25. Visualización local costmap ajustado.

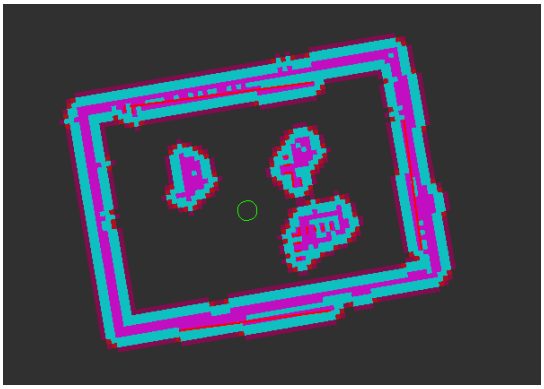

Figura 26. Visualización global costmap ajustado.

Se puede apreciar que en esta situación el robot si va a tener espacio para planificar sus trayectorias y moverse por el entorno.

Finalmente se publica un **objetivo** en el *topic* move_base_simple/goal al que enviar el robot y comprobar que funciona correctamente. La posición del objetivo se puede publicar mediante línea de comando o directamente en Rviz con la opción 2D Nav Goal.

```
rostopic pub move_base_simple/goal geometry_msgs/PoseStamped '{header
: {stamp: now, frame_id: "map"}, pose: {position: {x: -5, y:  -5, z:
0.0}, orientation: {w: 1}}}'
```

Código ROS 7. Publicar objetivo en move_base_simple/goal.

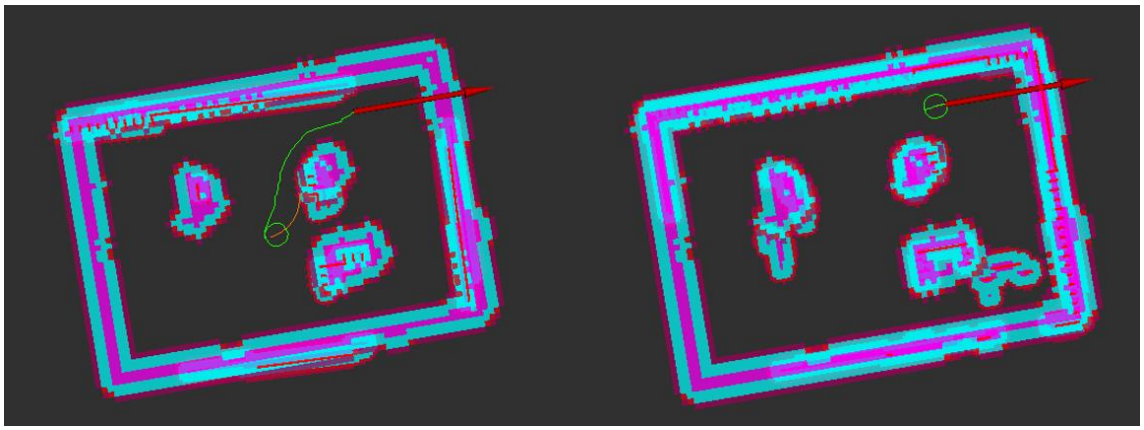

Figura 27. Robot trazando trayectoria. Robot alcanzando objetivo.

En la imagen de la Figura 27 el objetivo se muestra como una gran flecha roja. A la izquierda se ve el instante inicial en el que el robot traza dos trayectorias, la verde se trata de la trayectoria global, el camino más corto, mientras la línea naranja es la trayectoria local. Esta última va trazándose conforme el robot se mueve por el entorno variando en función de los obstáculos más cercanos.

Una vez llega al destino marcado se comprueba si las coordenadas coinciden con las indicadas. Esto se realiza mediante línea de comando haciendo uso de las herramientas del paquete tf.

```
rosrun tf tf_echo map base_link
```

Código ROS 8. Comprobación de coordenadas del sistema de referencia base_link respecto a map.

```
osboxes@osboxes: ~/catkin_ws 101x19
- Rotation: in Quaternion [0.000, 0.000, 0.003, 1.000]
            in RPY (radian) [0.000, -0.000, 0.005]
            in RPY (degree) [0.000, -0.000, 0.308]
At time 125.800
- Translation: [-5.071, -5.011, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.003, 1.000]
            in RPY (radian) [0.000, -0.000, 0.005]
            in RPY (degree) [0.000, -0.000, 0.308]
At time 126.800
- Translation: [-5.071, -5.011, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.003, 1.000]
            in RPY (radian) [0.000, -0.000, 0.005]
            in RPY (degree) [0.000, -0.000, 0.308]
At time 127.800
- Translation: [-5.071, -5.011, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.003, 1.000]
            in RPY (radian) [0.000, -0.000, 0.005]
            in RPY (degree) [0.000, -0.000, 0.308]
```

Figura 28. Terminal con las coordenadas alcanzadas.

Existe cierta tolerancia en cuanto a la posición del robot ya que como muestra la Figura 28 las coordenadas alcanzadas no son exactamente [-5 -5 0]. Esta tolerancia se puede modificar y ajustar mediante parámetros, por defecto la tolerancia en x-y está establecida en 0,2 (*xy_goal_tolerance*) y en orientación en 0,17 (*yaw_goal_tolerance*) por lo que está dentro del rango permitido.

Se realizan varias pruebas de navegación en las que se publican objetivos tanto por línea de comando como a través de Rviz con la herramienta **2DNavGoal** [16].

- Navegación Caso 2. En el vídeo se muestra como el robot alcanza las posiciones marcadas con éxito.

- Navegación Caso 3. En la simulación se prueba a enviar un objetivo no alcanzable, que esté sobre uno de los objetos. Se obtiene un mensaje de error al trazar la trayectoria. Tras el error el robot comienza a rotar sobre sí mismo en lo que se llama *"recovery behaviour"* un comportamiento mediante el cual el robot trata de buscar una ruta posible. Realiza varias rotaciones hasta que aborta declarando que el objetivo marcado no es alcanzable.

Capítulo 6.
Conclusiones y líneas futuras.

En este último capítulo se comentan las conclusiones obtenidas a lo largo del proyecto y las posibles líneas de trabajo para futuros proyectos.

6.1 Conclusiones.

Tras el desarrollo del trabajo expuesto se puede afirmar haber cumplido los objetivos planteados. En primer lugar, se consigue llevar un entorno real como es el de una plataforma de robots móviles a una simulación con la que poder trabajar. Se adapta un paquete de navegación a la simulación creada lo que permite enviar los robots simulados a posiciones objetivo del escenario simulado.

La implementación de un sistema que permite simular de manera automática el entorno de la plataforma puede proporcionar varias ventajas. La posibilidad de ejecutar aplicaciones sin la necesidad de montar la plataforma físicamente, pudiendo realizar numerosas pruebas con rapidez y eficacia. Poder trabajar con distintos escenarios que se pueden guardar como *worldfiles* y utilizarlos en cualquier momento para distintos proyectos.

Se introduce la cuestión de la navegación autónoma adaptando la simulación creada para que sea capaz de enviar al robot simulado a los objetivos que se desee. Esta funcionalidad es una de las bases del comportamiento de robots móviles autónomos y puede ser implementada en numerosas aplicaciones.

Para lograr estos objetivos se han introducido varios conceptos de ROS y se ha hecho uso de algunas de sus herramientas más comunes comprobando su gran potencial en el desarrollo de software robótico y sus numerosas posibilidades. Ha sido necesario el aprendizaje en lenguaje C++ para la creación del software que permite crear los mundos simulados en Stage. Además de familiarizarse con el simulador y sus posibilidades a la hora de representar distintos modelos.

6.2 Líneas de trabajo futuras.

A partir de este proyecto se abren distintos caminos en los que trabajar tanto para desarrollar nuevas funcionalidades para la plataforma como para mejorar las aportadas con este trabajo.

En lo que se refiere a este trabajo, una posibilidad es implementar el *software* desarrollado para la creación del mapa de simulación para otro tipo de archivos como son los *json.* En la plataforma se hace uso tanto de archivos *txt* como *json* por lo que otra implementación en este aspecto adaptando el código creado al manejo de archivos *json* sería de utilidad.

Se han observado pequeñas discrepancias en los datos de entrada por lo que mejorar el algoritmo de la toma de datos de la plataforma ayudaría a obtener simulaciones más precisas.

A partir de la navegación configurada en la simulación se pueden desarrollar distintas aplicaciones. Una posibilidad es crear un nodo capaz de leer una serie de objetivos de un fichero y que un robot los siga consecutivamente.

En este trabajo se utilizan mapas creados a partir del mapeado de un solo robot. Existen aplicaciones en entornos más amplios en las que se utilizan sistemas multi robot en los que estos colaboran para generar un mapa compartiendo los datos de sus sensores. Utilizar la navegación multi robot para este uso es un punto interesante de cara al aprendizaje en mapeado para su aplicación en otros ámbitos.

Estas son algunas ideas con las que seguir ganando funcionalidades en la plataforma de trabajo y aumentar las aplicaciones y el aprendizaje que puede ofrecer.

Capítulo 7.
Bibliografía.

[1] (s.f.). *Autonomous Robots. Master on robotics, graphics and computer vision. Laboratory class 2: Low level control of a mobile robot in ROS.*

[2] (s.f.). *Autonoumous Robots. Master on Robotics, graphics and computer vision. Laboratory class 1: Introduction to ROS.*

[3] Cañas Plaza, J. M., Cazorla Quevedo, M. Á., & Matellán, V. (2009). Uso de simuladores en docencia de robótica móvil. *IEEE-RITA*. Obtenido de http://rita.det.uvigo.es/200911/uploads/IEEE-RITA.2009.V4.N4.A4.pdf

[4] *Capítulo 2. Navegación en Robots Móviles.* (s.f.). Obtenido de http://webpersonal.uma.es/~VFMM/PDF/cap2.pdf

[5] *How to use Player / Stage*. (s.f.). Obtenido de https://player-stage-manual.readthedocs.io/en/latest/WORLDFILES/

[6] Martins, J. A. (s.f.). *Multi Robot Simultaneus Localization and Mapping.* Obtenido de https://core.ac.uk/download/pdf/43576212.pdf

[7] O'Kane, J. M. (2013). *A Gentle Introduction to ROS.* Columbia.

[8] Prinz, U., & Prinz, P. (2002). *A Complete Guide to Programming in C++.* Jones and Bartlett Publishers.

[9] Vaughan, R. (2007). Massively multi-robot simulation in stage. Obtenido de https://link.springer.com/content/pdf/10.1007/s11721-008-0014-4.pdf

[10]*wikiros*. (s.f.). Obtenido de http://wiki.ros.org/ROS/Tutorials

[11]*wikiros*. (s.f.). Obtenido de http://wiki.ros.org/navigation/Tutorials/RobotSetup

[12]*wikiros*. (s.f.). Obtenido de http://wiki.ros.org/base_local_planner

[13]*wikiros*. (s.f.). Obtenido de http://wiki.ros.org/costmap_2d

[14]*wikiros*. (s.f.). Obtenido de http://wiki.ros.org/roslaunch

[15]*wikiros*. (s.f.). Obtenido de http://wiki.ros.org/es/ROS/Introduccion

[16] https://drive.google.com/drive/folders/1HaMmxosDbpX7cwbYif07b8W713QPHaD-?usp=sharing

[17] Rusu, R. B. (s.f.). *Robot Operating System overview*.

ANEXOS.

Anexo I. Fichero de entrada, datos modulo. (Caso 1)

```
Salida del modulo de vision
ESCENARIO:
Ancho: 2.34
Largo: 3.32
----Initial Position of Robots(x(cm), y(cm), rad)----
Radio: 9
Robot[1]=(249,124,2.85286)
Robot[2]=(163,75,-1.16488)

---------List of obstacles (Format -->[x,y])---------
Obstaculo 1
            color: rojo
            esquina[1]: (88,156)
            esquina[2]: (98,162)
            esquina[3]: (107,163)
            esquina[4]: (111,158)
            esquina[5]: (114,132)
            esquina[6]: (92,129)
Obstaculo 2
            color: verde
            esquina[1]: (209,72)
            esquina[2]: (209,80)
            esquina[3]: (221,82)
            esquina[4]: (233,82)
            esquina[5]: (238,81)
            esquina[6]: (240,73)
            esquina[7]: (241,67)
            esquina[8]: (234,63)
            esquina[9]: (217,59)
            esquina[10]: (212,59)

Obstaculo 3
            color: azul
            esquina[1]: (210,142)
            esquina[2]: (210,150)
            esquina[3]: (211,159)
            esquina[4]: (223,163)
            esquina[5]: (228,160)
            esquina[6]: (230,155)
            esquina[7]: (227,147)
            esquina[8]: (226,145)
            esquina[9]: (223,141)
            esquina[10]: (215,137)
```

Anexo II. Fichero de salida, mapa. (Caso 1)

```
window
(
        size [ 600 600 ]
        scale 160
        center [ 1.66 1.17 ]
        rotate [ 0.000  0.000 ]
        show_data 1
)
define floorplan model
(
        color "gray30"
        boundary 1
        gui_nose 0
        gui_grid 1
        gui_outline 0
        gripper_return 0
        fiducial_return 0
        ranger_return 1.000
)
resolution 0.08
floorplan
(
        name "willow"
        bitmap "rectangulo.png"
        size [ 3.32 2.34 0.25 ]
        pose [ 1.66 1.17  0 0 ]
)
define RPsonar ranger
(
        sensor(
                range [ 0.0 10.0]
                fov 270
                samples 1081
        )
        color "black"
        size [0.050 0.050 0.09]
)
define robotP position
(
        bitmap "circulo.png"
        size [0.18 0.18 0.09]
        origin [-0.050 0.000 0.000 0.000]
        gui_nose 1
        drive "diff"
        RPsonar(pose [0.18 0 0 0])
)
robotP
(
        name "R1"
        pose [2.49 1.24 0 2.85286]
        color "black"
)
robotP
```

```
54 (
55         name "R2"
56         pose [1.63 0.75 0 -1.16488]
57         color "black"
58 )
59 define object1 model
60 (
61         block
62         (
63                 points 6
64                 point[0] [0.88 1.56]
65                 point[1] [0.98 1.62]
66                 point[2] [1.07 1.63]
67                 point[3] [1.11 1.58]
68                 point[4] [1.14 1.32]
69                 point[5] [0.92 1.29]
70                 z [0 1]
71         )
72         size [0.26 0.34 0.1]
73         color "blue"
74 )
75 object1( pose [1.01667 1.5 0 0 ])
76 define object2 model
77 (
78         block
79         (
80                 points 10
81                 point[0] [2.09 0.72]
82                 point[1] [2.09 0.8]
83                 point[2] [2.21 0.82]
84                 point[3] [2.33 0.82]
85                 point[4] [2.38 0.81]
86                 point[5] [2.4 0.73]
87                 point[6] [2.41 0.67]
88                 point[7] [2.34 0.63]
89                 point[8] [2.17 0.59]
90                 point[9] [2.12 0.59]
91                 z [0 1]
92         )
93         size [0.32 0.23 0.1]
94         color "red"
95 )
96 object2( pose [2.254 0.718 0 0 ])
97 define object3 model
98 (
99         block
100        (
101                points 10
102                point[0] [2.1 1.42]
103                point[1] [2.1 1.5]
104                point[2] [2.11 1.59]
105                point[3] [2.23 1.63]
106                point[4] [2.28 1.6]
107                point[5] [2.3 1.55]
108                point[6] [2.27 1.47]
109                point[7] [2.26 1.45]
110                point[8] [2.23 1.41]
```

```
111                 point[9] [2.15 1.37]
112                 z [0 1]
113         )
114         size [0.2 0.26 0.1]
115         color "green"
116 )
117 object3( pose [2.203 1.499 0 0 ])
```

Anexo III. Mapeado del entorno simulado. *gmapping*.

El paquete **gmapping** de ROS permite generar un mapa del entorno por medio de técnicas SLAM (*Simultaneous Localization and Mapping*). Haciendo uso del nodo **slam_gmapping** se crea una red de ocupación que conforma el mapa del entorno a través de los datos del sensor laser del robot.

Se lanza una simulación con el entorno a simular y un solo robot controlado por tecleado con el nodo telep_twist_keyboard. Al ejecutar el nodo slam_gmaping como de muestra en el código siguiente se empiezan a registrar los datos del sensor láser.

```
rosmake gmapping
rosrun gmapping slam_gmapping scan:=base_scan
```

Código ROS 9. Ejecutar nodo slam_gmapping para mapeado.

Para obtener un mapa completo se va conduciendo el robot por todo el entorno. Mediante el uso de Rviz se puede ver cómo se va generando el mapa y por lo tanto cuando se completa visualizándose todos los componentes del entorno.

Una vez completado es necesario guardar el mapa, se generarán dos archivos, una imagen del entorno y un YAML con los metadatos del mapa.

```
rosrun map_server map_saver -f map2
```

Código ROS 10. Guardar mapa generado con el nombre map2.

Al ejecutar el Código ROS 10 se guardan los datos del mapa y se crean los archivos citados.

Figura 29. map2.pgm.

En el archivo YAML se especifican los datos que se muestran a continuación y se explican brevemente.

```
image: map2.pgm
resolution: 0.050000
origin: [-107.629, -107.032, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Código 10. Contenido archivo map2.yaml.

- Resolución. Define la resolución del mapa incluido en la imagen en metros/pixel. En este caso se define por defecto ya que el mapa se genera con unas dimensiones 4000x4000 pixels.

- El origen se corresponde con la posición de la esquina inferior izquierda de la imagen insertada en el mapa que se crea. Originalmente se establece en [-100, -100, 0] sin embargo se modifica para ajustar los sistemas de referencia. Se posiciona en ese punto ya que [-7.629, -7.032, 0] es la diferencia que hay entre los sistemas de referencia de /map y /odom.

- Los campos occupied_thresh y free_thresh son indicadores sobre la ocupación de los pixeles. En cuanto al parámetro *negate* sirve para invertir el sentido sobre qué pixeles se consideran zona ocupada, los blancos o los negros.

Este YAML es el que se carga en el nodo map_server para la navegación. Por tanto, cada vez que se modifique el escenario habría que realizar un mapeado y generar un mapa nuevo. En este caso se ha mostrado el ejemplo para el Caso 2 (Figura 18) que es el utilizado durante toda la parte de navegación.

En el repositorio Drive [16] se incluye un ejemplo de la realización del mapeado del Caso 3.

Anexo IV. Navegación un solo robot.

En la Figura 30 se muestra el árbol de transformaciones completo para la navegación de un solo robot.

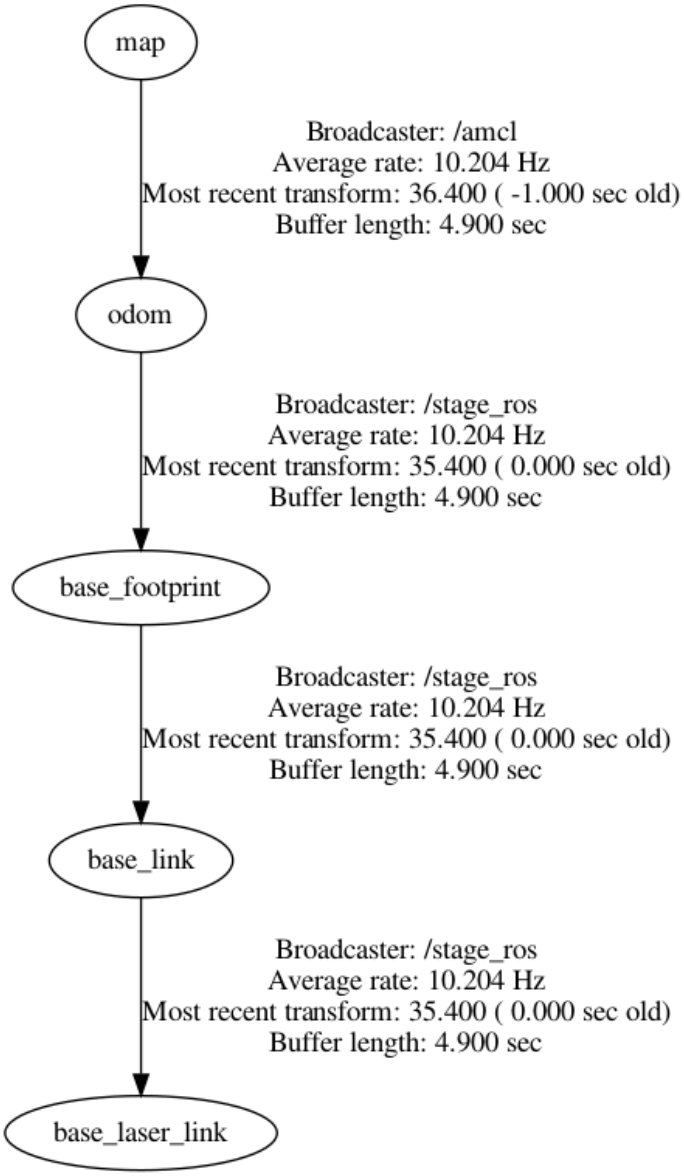

Figura 30. Transform tree completo para un solo robot.

Anexo V. Contenido de la carpeta Drive.

Durante el desarrollo del trabajo se han ido creando distintos archivos y pruebas de simulación que como ha quedado indicado se recogen en un repositorio de Drive. En él existen dos carpetas una denominada “Archivos” y otra llamada “Grabaciones pruebas”.

Archivos.

- **datos_entrada.txt** : archivo de texto que recoge los datos de partida de los tres escenarios de la plataforma o casos simulados en el trabajo.

- **datosmodulo.txt** : archivo de texto que funciona como fichero de entrada en el software creado y que contiene los datos del Caso 2 a modo de ejemplo.

- **main.cpp** : software creado durante el trabajo para la obtención del escenario de simulación.

FUNCIÓN	FINALIDAD	ENTRADAS
window	Escribir en el fichero de salida la definición del modelo window	• f: fichero de salida.
floorplan_def	Escribir en el fichero de salida la definición de floorplan.	• f: fichero de salida.
RPsonar	Escribir en el fichero de salida la definición del modelo RPsonar.	• f: fichero de salida.
RobotP_def	Escribir en el fichero de salida la definición del modelo RobotP.	• f: fichero de salida. • x: radio del robot.
media	Calcular la media de los componentes del vector de coordenadas.	• esq: vector con las coordenadas (x o y) de las esquinas de un objeto. • n: número de esquinas
Obtain_size	Obtener la diferencia entre el máximo y el mínimo de un vector.	• esq: vector con las coordenadas (x o y) de las esquinas de un objeto. • n: número de esquinas
Object_def	Escribir en el fichero de salida la definición del modelo de un objeto.	• f: fichero de salida. • x: vector con coordenadas x. • y:vector con coordenadas y. • sx: tamaño en coord.x. • sy: tamaño en coord.. y. • c: color del objeto. • nesq: número de esquinas. • obji : número de objeto.

Tabla 2. Descripción de las funciones creadas en el código.

En la Tabla 2 se recogen las funciones utilizadas en el código main.cpp con el fin de facilitar la comprensión del mismo.

- **salida.txt** : fichero de salida que se obtiene al ejecutar el programa teniendo como entrada el archivo "datosmodulo.txt" ejemplificando el Caso 2 de simulación.

- **stage.launch** : Archivo tipo *launch* que lanza el simulador Stage con dos robots conducidos mediante teclado.

- **nav_stage.launch :** Archivo tipo *launch* creado para lanzar la simulación de navegación con un solo robot.

Grabaciones pruebas.

Esta carpeta contiene grabaciones sobre dos pruebas de navegación y una de mapeado del entorno que ya han sido descritas.

- Ejemplo de mapeado Caso 3.

- Navegación Caso 2.

- Navegación Caso 3.