Escuela Politécnica Superior

El framework ROS.

Implementación y experimentación de herramientas para el diseño y desarrollo de comportamientos en robots tipo Turtlebot.

Grado en Ingeniería Robótica

Trabajo Fin de Grado

Autor:
Sandra Sánchez Gea

Tutor:
María del Pilar Arques Corrales

Noviembre 2019

Resumen

En este trabajo se utilizan diferentes herramientas proporcionadas por el framework ROS en el robot Turtlebot, trabajando tanto en la versión de simulador como experimentando con los robots reales.

En primer lugar, se ha llevado a cabo un estudio del estado del arte actual del framework ROS, de los campos de aplicación actuales de ROS, y el concepto de robot móvil. A continuación, se ha explicado el funcionamiento de la arquitectura de la red en la que se basan las conexiones en ROS. Tras esto, se ha realizado una descripción de los componentes que conforman al robot Turtlebot utilizado en este proyecto.

A continuación, se ha llevado a cabo la puesta en marcha del robot Turtlebot real y en simulación. Donde se han habilitado los sensores y actuadores que conforman al robot y se han realizado unas pruebas iniciales para verificar el correcto funcionamiento del éstos.

Después de la puesta en marcha, se han realizado diferentes experimentos. En primer lugar, en cuanto a ROS, se ha visto cómo crear un espacio de trabajo y cómo crear unos paquetes básicos de un nodo publicador y un suscriptor.

En cuanto a la simulación, se ha visto cómo insertar unos mundos creados en el simulador Gazebo junto con el robot Turtlebot. A continuación, se ha creado un paquete donde se ha utilizado uno de los mundos creados, el cual simula una carretera. Este paquete realiza un procesamiento de las imágenes capturadas por el Turtlebot donde se detectan las líneas de la carretera y a partir de éstas, se calculan las velocidades lineales y angulares que se publican para mover al robot a través de esta carretera de una forma autónoma.

También se ha creado un paquete donde se han insertado varios robots Turtlebots dentro del mundo especificado en el simulador Gazebo, el cual ayudará a desarrollar futuros algoritmos para crear comportamientos multirobots.

En cuanto al robot real, se ha conseguido realizar una conexión entre los dos PC, el del Turtlebot y el ordenador personal, en la que ambos se encuentran dentro de la misma red de nodos de ROS, donde se consigue una comunicación transversal en la que desde los dos PC se pueden ejecutar nodos de forma bidireccional.

También se han construido diversos mapas a través de los datos capturados a partir de los ficheros bags y en tiempo real, del robot en simulación y del Turtlebot real. También se ha conseguido mover al robot Turtlebot de forma autónoma solo indicando el objetivo a alcanzar a partir de estos mapas conocidos.

Por último, se han realizado distintas pruebas con la herramienta ROS Toolbox de Matlab, en las que Matlab se ha conectado al mismo ROS Master en el que se encontraba también conectados los nodos del Turtlebot, real y simulado. A través de Matlab se han realizado las mismas pruebas realizadas con las otras herramientas de ROS, creación de un suscriptor y un publicador, también, la construcción de un mapa a partir de una rosbag.

Este proyecto ha sido financiado por el Ministerio de Ciencia, Innovación y Universidades (Spain), proyecto RTI2018-096219-B-I00. Proyecto co-financiado con fondos FEDER.

Justificación y objetivos

Uno de los retos a los que se enfrentan muchos usuarios nuevos de ROS es saber por dónde empezar. Hay realmente dos fases para comenzar con ROS: la Fase 1 implica aprender los conceptos básicos y las técnicas de programación, mientras que la Fase 2 trata sobre el uso de ROS para controlar su propio robot.

El objetivo de este trabajo es comprender el funcionamiento la tecnología ROS de una forma global, gracias a la realización de una serie de pruebas y experimentos con el robot móvil Turtlebot.

En este trabajo se va a realizar la puesta en marcha del robot TurtleBot en simulación y en el robot real. Posteriormente, se realizarán una serie de pruebas y tutoriales con distintas herramientas proporcionadas por el framework ROS que nos proporcionan una base sólida para crear comportamientos más complejos utilizando ROS y el robot Turtlebot. Entre otras se utilizarán, el simulador Gazebo, el visualizador RVIZ y la librería ROS Toolbox de Matlab.

Por otro lado, con este trabajo, también se pretende dar al usuario algunas rutinas básicas y comunes para trabajar con robots tipo Turtlebot y con ROS, para que, a partir de éstas, puedan desarrollar comportamientos más complejos.

Como se ha comentado, este trabajo está centrado en la comprensión de ROS y la experimentación con el robot Turtlebot. Aunque estas rutinas son específicas para el robot móvil Turtlebot, algunas de estas rutinas también se podrían aplicar a otros robots móviles que utilicen ROS, dado que la base de funcionamiento en este tipo de robots es la misma. Por lo que el objetivo de este trabajo es también entender el funcionamiento de ROS de una forma global para poder aplicarlo a cualquier dispositivo que utilice esta tecnología.

Índice de contenido

Índice de figuras

1. Introducción

El Sistema Operativo de Robots [1] (ROS) es un conjunto de bibliotecas de software y herramientas que ayudan a crear aplicaciones de robots. Desde controladores hasta algoritmos de última generación, con potentes herramientas de desarrollo, cuyo objetivo es simplificar la tarea de crear un comportamiento de robot complejo y robusto a través de una amplia variedad de plataformas robóticas.

Características de ROS [2]:

- Un conjunto de controladores que le permiten leer los datos de los sensores y enviar comandos a motores y otros actuadores, en un formato abstracto y bien definido. Una amplia variedad de hardware popular, incluyendo un creciente número de aplicaciones comerciales sistemas de robots disponibles.

- Una amplia y creciente colección de algoritmos robóticos fundamentales que permiten construir mapas de mundos, navegar a su alrededor, representar e interpretar los datos de los sensores, planificar los movimientos, manipular objetos y hacer muchas otras cosas.

- Toda la infraestructura computacional que permite mover datos, conectar los distintos componentes de un sistema robótico complejo e incorporar sus propios algoritmos. ROS está intrínsecamente distribuido y permite dividir la carga de trabajo entre varios ordenadores sin problema.

- Un gran conjunto de herramientas que facilitan la visualización del estado del robot y de los algoritmos, depurar los comportamientos defectuosos y registrar los datos de los sensores.

- Por último, ROS incluye un amplio conjunto de recursos, como por ejemplo que documenta muchos de los aspectos del marco de trabajo, un sitio de preguntas y respuestas en el que puede pedir ayuda y compartir lo que ha aprendido, y una próspera comunidad de usuarios y desarrolladores.

Actualmente hay mucha variedad de robots de distintas categorías usando el framework ROS [3]. Hay robots aéreos como el Gapter, terrestres como el Turtlebot o el Pepper, robots manipuladores como Fanuc o ABB y marinos como el Clearpath Heron USV.

Por otro lado, también hay una gran cantidad de componentes que utilizan ROS. Como manos, cámaras, sensores 3D, láseres…

Con el crecimiento exponencial de la robótica, se han encontrado algunas dificultades en términos de escritura del software compatible para distintos robots. Si se varía notablemente el hardware, la reutilización de código no es trivial. Frente a esta tendencia, ROS proporciona las bibliotecas y herramientas para ayudar a los desarrolladores de software a crear aplicaciones robóticas. Los objetivos principales de ROS son la abstracción de hardware de bajo nivel de control de dispositivos, la implementación de funcionalidades de uso común, el paso de mensajes entre procesos y la gestión de paquetes (ROS packages). ROS proporciona todas las partes de un sistema de software de robot que de otro modo tendría que escribir. Le permite concentrarse en las partes del sistema que le importan, sin preocuparse por las partes que no le importan.

En las últimas décadas se han conseguido increíbles avances en la robótica móvil. Temas como la navegación autónoma, la percepción, la reactividad, la creación de mapas, la evitación de obstáculos y la auto-localización, han sido profundamente estudiados durante los últimos años, con aplicaciones en la industria, el transporte, el área militar y entornos de seguridad. Además, algunos robots móviles se han convertido en productos de consumo para el entretenimiento o para realizar las tareas domésticas diarias, como, por ejemplo, aspirar el suelo.

Un robot móvil [4] es una máquina automática capaz de trasladarse en cualquier ambiente dado. Los robots móviles tienen la capacidad de moverse en su entorno y no están fijados a una ubicación física.

Los robots móviles pueden ser autónomos, lo que significa que son capaces de navegar en un entorno no controlado y tomar sus propias decisiones de forma automática, sin la intervención humana. Se componen de tres herramientas para ello: sensores, procesadores y actuadores.

- La percepción implica el uso de sensores, que es un dispositivo capaz de captar magnitudes físicas (distancia, temperatura, sonido, luz…) y es la fuente de información que nutre al robot. Con estos instrumentos, la máquina puede determinar en qué lugar se encuentra, en qué condiciones y son el fundamento del procesador.
- El procesador es lo que permite al robot tomar decisiones.
- Finalmente, los actuadores permiten llevar a cabo las decisiones tomadas por el elemento anterior y que conducen, por ejemplo, a que un robot gire en un momento dado si se acerca demasiado a una pared.

2. Tecnologías utilizadas

En este apartado se hará una breve descripción de todas las tecnologías utilizadas para realizar el trabajo. Se describirán las características las del robot móvil utilizado, en este caso, del Turtlebot y de las herramientas software.

2.1 Turtlebot

La plataforma Turtlebot [5], es un robot móvil de bajo coste con software de código abierto. Es una plataforma básica para trabajar con ROS y algoritmos avanzados de SLAM, visión artificial, cámaras RGB-D y nubes de puntos.

El robot está compuesto por una base móvil Kobuki que incluye un sistema diferencial de movimiento con encoders y giroscopio electrónico. El turtlebot utilizado en este trabajo también dispone de unos dispositivos adicionales, en concreto un escáner láser 2D Hokuyo y una cámara 3D Astra, dispositivos son por los adquiere información del espacio que lo rodea. Y todo está integrado desde un computador netbook a través del framework ROS.

Figura 2.1: Robot Turtlebot

La base móvil Kobuki [6] es un robot similar al irobot roomba, de la empresa irobot, donde su principal función era el aseo y la limpieza de viviendas. El Kobuki cuenta con sensores infrarrojos para detectar escaleras, tiene un bumper que le ayuda a recibir el golpe y reaccionar ante el evento, cuenta con un giroscopio para controlar la odometría, llantas especiales para pasar por encima de obstáculos y moverse en cualquier

superficie, botones de accionamiento, LEDs programables, entradas y voltajes para alimentación, puerto Usb y Un puerto DB-25.

El láser Hokuyo UST-10LX [7] es un escáner láser 2D pequeño, preciso, y con alta velocidad de detección de obstáculos. Ideal para aplicaciones de robótica. Que puede escanear con una apertura de hasta 270 grados con un alcance de hasta 10 metros. Hokuyo es más preciso que el sensor Kinect y sería más útil para hacer mapas con gmapping y aplicaciones SLAM.

Astra [8] es una potente y fiable cámara 3D que incluye el microchip 3D y VGA propietario de Orbbec. La cámara Astra ha sido desarrollada para ser altamente compatible con las aplicaciones desarrolladas con OpenNI, siendo ideal para aplicaciones preexistentes que fueron desarrolladas con OpenNI. El rango de detección de la cámara va desde 0.4 a 8 metros.

2.2 Software

En este apartado se van a describir las herramientas softwares utilizadas para la realización de este trabajo. En concreto, las herramientas ROS, el simulador 3D Gazebo. El visualizador Rviz, el lenguaje de programación C++ junto con las librerías de OpenCV de visión artificial y el programa Matlab.

2.2.1 ROS

ROS [9] (en inglés Robot Operating System) es un framework para el desarrollo de software para robots, provee de librerías y herramientas a los desarrolladores de software para crear sus propias aplicaciones.

A pesar de su nombre, ROS no es un sistema operativo propiamente dicho, de hecho, funciona sobre Linux. Lo que hace realmente es proveer los servicios estándar de un sistema operativo, tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y administración de paquetes, dejando las labores de procesamiento, manejo de memoria, gestión de interfaces gráficas, etc. También proporciona herramientas y bibliotecas para obtener, construir, escribir y ejecutar código en varios ordenadores.

Características de ROS

Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros.

ROS se ejecuta como una red de procesos (nodos) de igual a igual de tal forma que cada nodo pueda recibir mensajes de estado de los demás. Estos nodos se acoplan libremente utilizando la infraestructura de comunicación de ROS.

Cada nodo se agrupa en paquetes que pueden ser compartidos en otros sistemas ROS. Además, es independiente del lenguaje, pues puede ser implementado en cualquier lenguaje de programación moderno, estando implementado ya en Python, C++ y Lisp. Por otra parte, es totalmente compatible con OpenCV, lo que nos posibilita implementar los programas desarrollados en ROS.

El objetivo principal de ROS es apoyar la reutilización de código en la investigación y desarrollo de robótica. ROS es un marco distribuido de procesos (también conocido como Nodos) que permite que los ejecutables se diseñen individualmente y se acoplen libremente en tiempo de ejecución. Estos procesos se pueden agrupar en paquetes y pilas, que se pueden compartir y distribuir fácilmente.

Conceptos básicos de ROS

ROS tiene dos niveles de conceptos: el nivel de Sistema de Archivos y el nivel de Gráficos de Computación [10].

Sistema de archivos de ROS

Los conceptos a nivel de sistema de ficheros cubren principalmente los recursos ROS que se encuentran en el disco, como, por ejemplo:

- **Paquetes**: Los paquetes son la unidad principal para organizar el software en ROS. Un paquete puede contener procesos en tiempo de ejecución ROS (nodos), una biblioteca dependiente de ROS, conjuntos de datos, archivos de configuración o cualquier otra cosa que esté organizada de forma útil. Los paquetes son el elemento de construcción y liberación más atómico en ROS. Lo que significa que lo más granular que se puede construir y liberar es un paquete.

- **Metapaquetes**: Los metapaquetes son paquetes especializados que sólo sirven para representar un grupo de otros paquetes relacionados.

- **Datos de los paquetes:** Estos archivos (*package.xml*) proporcionan metadatos sobre un paquete, incluyendo su nombre, versión, descripción, información de licencia, dependencias y otra información meta como paquetes exportados.

- **Tipos de mensajes (msg)**: Las descripciones de los mensajes definen las estructuras de datos para los mensajes enviados en ROS.

- **Tipos de servicio (srv)**: Las descripciones de servicio definen las estructuras de datos de solicitud y respuesta para servicios en ROS.

Arquitectura de grafos de ROS

ROS está basado en una arquitectura de grafos que procesa los datos en conjunto. Está basada en una arquitectura cliente-servidor; un cliente realiza peticiones a otro programa, el servidor, quien le da respuesta.

Los principales elementos de esta red de grafos son nodos, master, servidor de parámetros, mensajes, servicios, topics y bags, todos ellos proporcionan datos a la red de distintas maneras.

- **Nodos**: Cada nodo dentro de la red es un proceso que realiza una tarea. ROS está diseñado para ser modular, con lo que de esta forma podemos tener varios nodos ejecutando distintas tareas. Un sistema de control de robot normalmente comprende muchos nodos. Por ejemplo, un nodo controla un láser, un nodo controla los motores de las ruedas, un nodo realiza la localización, un nodo realiza la planificación de rutas, un nodo proporciona una vista gráfica del sistema, etc. Un nodo es un cliente ROS, escrito con la biblioteca de C++ (roscpp) o de Python (rospy).

- **Maestro**: El ROS Master es el nodo maestro que proporciona un registro de nombres de tal forma que nos permita hacer una búsqueda dentro del grafo de computación. Sin el nodo maestro los distintos nodos del grafo no podrían encontrarse entre ellos, intercambiar mensajes o invocar servicios.

- **Servidor de parámetros**: El servidor de parámetros permite almacenar datos identificados por una clave. Actualmente, el servidor de parámetros forma parte del ROS Master.

- **Mensajes**: Los nodos se comunican entre ellos intercambiando mensajes. Los mensajes son estructuras de datos, compuestos de distintos campos tipados. Dentro de estos campos se utilizan distintos tipos primitivos como enteros, booleanos, float, etc… Algunos mensajes pueden incluir distintas estructuras o arrays anidados.

- **Topic**: Los mensajes se enrutan a través de un sistema de transporte con semántica de publicación/suscripción. De tal forma que un nodo envía un mensaje publicándolo en un topic determinado. El topic es un nombre que se utiliza para identificar el contenido del mensaje. Un nodo que esté interesado en un determinado tipo de datos se suscribirá al topic correspondiente. Puede haber múltiples editores y suscriptores concurrentes para un mismo topic, y un solo nodo puede publicar y/o suscribirse a múltiples topics. En general, los publicadores y los suscriptores no son conscientes de la existencia de los demás. Lógicamente, se puede pensar en un topic como un bus de mensajes. Cada bus tiene un nombre, y cualquiera puede conectarse al bus para enviar o recibir mensajes siempre y cuando sean del tipo correcto.

- **Servicios**: El modelo de publicación/suscripción es un paradigma de comunicación muy flexible, pero su transporte unidireccional de muchos a muchos no es apropiado para las interacciones de solicitud/respuesta. De esto se encargan los servicios encargados de servir la tarea para la que se les invoca, que se definen mediante un par de estructuras de mensajes: una para la solicitud y otra para la respuesta (Cliente/Servidor). De esta forma un nodo ofrece un servicio y otro nodo lo utiliza enviando una petición y esperando una respuesta.

- **Bags**: Los bags son un formato de almacenamiento de datos de ROS, que permiten guardar y reproducir distintos tipos de datos que podemos utilizar posteriormente en distintos nodos. Los bags son un mecanismo importante para almacenar datos, como los datos de los sensores, que pueden ser difíciles de recopilar pero que son necesarios para desarrollar y probar algoritmos.

El ROS Master [11] proporciona servicios de nombres y de registros al resto de los nodos del sistema ROS. Los nodos se comunican con el Maestro para reportar su información de registro. A medida que estos nodos se comunican con el Maestro, pueden recibir información sobre otros nodos registrados y hacer las conexiones apropiadas. El Maestro también hará llamadas de retorno (*callbacks*) a estos nodos cuando esta información de registro cambie, lo que permite a los nodos crear conexiones dinámicamente cuando se ejecutan nuevos nodos.

Los nodos se conectan directamente a otros nodos; el Maestro sólo proporciona información de búsqueda. Los nodos que se suscriban al topic solicitarán conexiones de los nodos que publiquen en ese topic, y establecerán esa conexión sobre un protocolo de conexión acordado. El protocolo más común utilizado en un ROS se llama TCPROS, que utiliza sockets TCP/IP estándar.

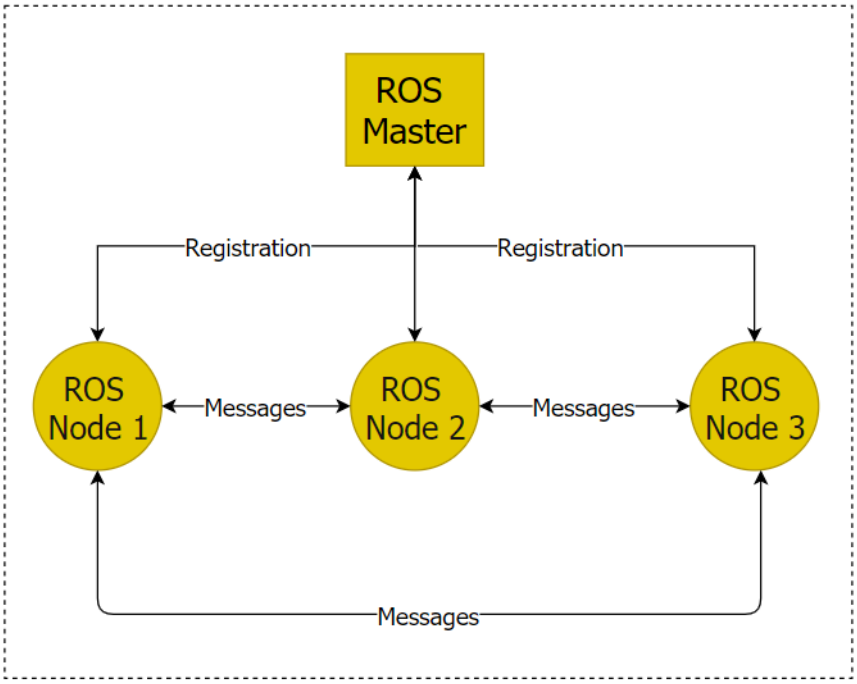

Figura 2.2: Arquitectura conexión de ROS

Esta arquitectura permite la operación desacoplada, donde los nombres son el medio principal por el cual se pueden construir sistemas más grandes y complejos. Los nombres tienen un papel muy importante en ROS: los nodos, topics, servicios y parámetros tienen todos nombres. Los nodos de ROS admiten la redistribución de nombres en línea de comandos, lo que significa que un programa compilado puede reconfigurarse en tiempo de ejecución para que funcione en una topología de la arquitectura de grafos de ROS.

Por ejemplo, para controlar el láser detector de distancias de Hokuyo, podemos iniciar el driver *hokuyo_node*, que se comunica con el láser y publica mensajes *sensor_msgs/LaserScan* sobre el topic del escáner (*/scan*). Para procesar esos datos, podemos escribir un nodo usando *laser_filters* que se suscribe a los mensajes sobre el topic */scan*. Después de la suscripción, nuestro filtro comenzará a recibir automáticamente los mensajes del láser.

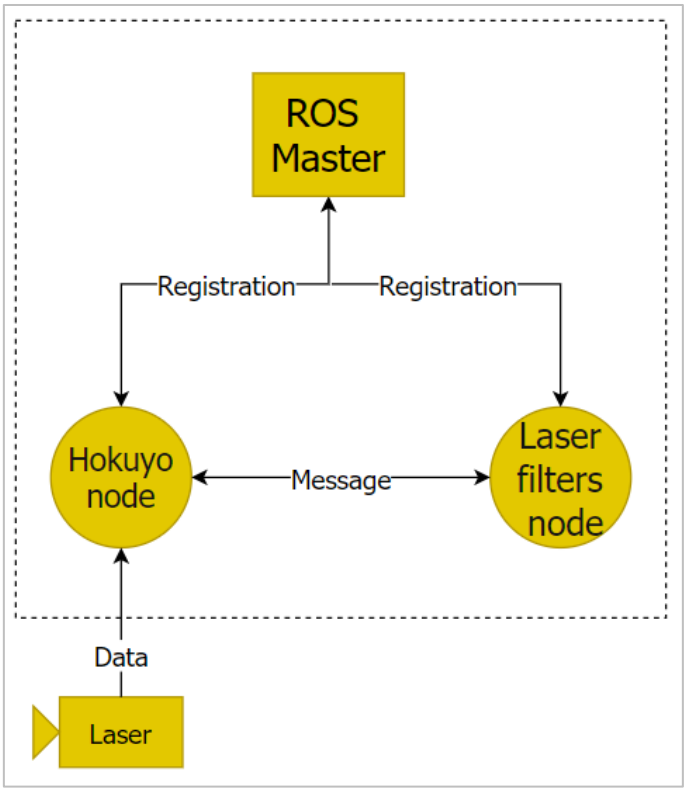

Figura 2.3: Conexión de nodos cuando se incluye el láser

Aquí se observa cómo de desacoplados están los dos lados. Todo lo que hace el nodo *hokuyo_node* es publicar sus capturas, sin saber si hay alguien suscrito. Todo lo que hace el filtro es suscribirse a */scan*, sin saber si alguien los está publicando. Los dos nodos pueden ser iniciados, desactivados y reiniciados, en cualquier orden, sin inducir condiciones de error.

2.2.2 Simulador 3D Gazebo

Gazebo [12] es un simulador 3D, cinemático, dinámico y multi-robot que permite realizar simulaciones de robots articulados en entornos complejos, interiores o exteriores, realistas y tridimensionales, con un alto grado de fidelidad: aceleraciones, fuerzas gravitatorias, inercias, masas, etc. Además de esto, también permite simular una gran cantidad de sensores.

Este simulador es idóneo para la realización de este trabajo, ya que tiene una relación nativa con ROS. Gazebo está implementado en ROS mediante la utilización de paquetes, los cuales se denominan *gazebo_ros_pkgs*. Estos paquetes contienen *plugins* que

interactúan con cualquier objeto de la escena del simulador y proporcionan la interfaz necesaria para usar ROS conjuntamente con Gazebo. Gazebo es el simulador 3D, mientras que ROS sirve como interfaz para el robot. Combinando ambos resultados en un poderoso simulador de robot.

Gazebo es considerado como un nodo para ROS, que permite lanzarlo mediante un archivo de tipo *.launch*. ROS permite ejecutar varios nodos a la vez, y uno de ellos puede ser Gazebo. Así, se facilita mucho la comunicación entre ellos, ya que, al ser como un nodo para ROS, éste puede publicar y suscribirse a cualquier topic del sistema. Por ejemplo, las velocidades del robot que estén siendo publicadas en un topic de ROS podrán ser aplicadas a los motores del robot virtual en Gazebo gracias a esto.

En definitiva, mediante el uso de esta plataforma podremos lograr una simulación bastante fiel del comportamiento del Turtlebot, de manera que su comportamiento en el mundo virtual de Gazebo sea similar al del mundo real.

2.2.3 Rviz

Rviz [13], abreviatura de visualización de ROS, es una herramienta de visualización 3D para ROS. Proporciona una vista del modelo de robot simulado, permite capturar la información de los sensores del robot y reproducir la información de datos capturados de los sensores.

Al visualizar lo que el robot está viendo, pensando y haciendo, se puede depurar una aplicación de robot desde las entradas del sensor hasta las acciones planificadas (o no planificadas).

Rviz muestra datos de sensores 3D de cámaras estéreo, láseres, Kinects y otros dispositivos 3D en forma de nubes de puntos o imágenes de profundidad. Los datos de sensores 2D de cámaras web, cámaras RGB y telémetros láser 2D se pueden ver en Rviz como datos de imagen.

2.2.4 Matlab

Matlab [14] combina un entorno de escritorio perfeccionado para el análisis iterativo y los procesos de diseño con un lenguaje de programación que expresa las matemáticas de matrices y arrays directamente.

Matlab (abreviatura de *MATrix LABoratory*, «laboratorio de matrices») es un sistema de cómputo numérico que ofrece un entorno de desarrollo integrado con un lenguaje de programación propio (lenguaje M). Entre sus prestaciones básicas se hallan la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario (GUI) y la comunicación con programas en otros lenguajes y con otros dispositivos *hardware*.

ROS Toolbox [15] proporciona una interfaz que conecta Matlab y Simulink con ROS, lo que permite crear una red de nodos ROS. La toolbox incluye funciones de Matlab y bloques de Simulink para importar, analizar y reproducir datos ROS grabados en archivos *rosbag*. También puede conectar con una red ROS en tiempo real para acceder a mensajes ROS.

La toolbox permite verificar los nodos ROS a través de la simulación de escritorio y mediante la conexión a simuladores de robots externos como Gazebo.

2.2.5 C++

C++ [16] es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup. La intención de su creación fue extender al lenguaje de programación C mecanismos que permiten la manipulación de objetos.

Roscpp [17] es una implementación C++ de ROS. Proporciona una biblioteca cliente que permite a los programadores de C++ interactuar rápidamente con Topics, Servicios y Parámetros de ROS. Roscpp es la biblioteca cliente de ROS más utilizada y está diseñada para ser la biblioteca de alto rendimiento para ROS.

2.2.6 OpenCV

OpenCV [18] es una biblioteca libre de visión artificial originalmente desarrollada por Intel, escrita originalmente en C/C++. Su publicación se da bajo licencia BSD, que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas.

Contiene más de 500 funciones que abarcan una gran gama de áreas en el proceso de visión, como reconocimiento de objetos (reconocimiento facial), calibración de cámaras y visión robótica.

3. Puesta en marcha

En este apartado se explica cómo empezar a trabajar con el Turtlebot, tanto con el robot real como en simulación. Se describirán los pasos para comunicarse remotamente con un PC al ordenador propio del Turtlebot, teleoperarlo con el teclado de nuestro PC y utilizar las librerías y paquetes ya existentes para los Turtlebots.

Partiremos de la premisa de que ya se dispone del sistema operativo Ubuntu y de ROS instalados en nuestro PC de trabajo y en nuestro Turtlebot. Junto con todas las librerías necesarias para ejecutar todas las herramientas utilizadas.

3.1 Turtlebot real

Se describirán todos los pasos a realizar para la puesta en marcha del robot Turtlebot real. En concreto, cómo realizar la conexión del PC del robot con otro ordenador. También la habilitación de los sensores y actuadores del Turtlebot junto con distintos test para la verificación de estos. Y después, se describirán distintas opciones para poder utilizar dichos sensores y actuadores.

3.1.1 Comunicación Turtlebot/PC

Para poder utilizar ROS en nuestro PC personal, se hace uso de los periféricos externos de nuestro ordenador (Pantalla, teclado y ratón). Por lo que para usar ROS en el Turtlebot también se debería disponer de dichos periféricos conectados al ordenador del robot. Esto limitaría el propósito final del robot móvil, que es desplazarse libremente por un entorno.

Por este motivo, se trabajará con el PC del Turtlebot de forma remota. Esta conexión, se puede realizar de distintas formas. A continuación, se van a describir dos opciones para realizar la conexión entre los dos ordenadores, a través de la conexión VNC y a partir del protocolo de comunicación SSH.

3.1.1.1 Conexión VNC

VNC [19] son las siglas en inglés de *Virtual Network Computing* (Computación Virtual en Red). VNC es un programa de software libre basado en una estructura cliente-servidor que permite observar las acciones del ordenador servidor remotamente a través de un ordenador cliente. VNC no impone restricciones en el sistema operativo del ordenador servidor con respecto al del cliente: es posible compartir la pantalla de una máquina con cualquier sistema operativo que admita VNC conectándose desde otro ordenador o dispositivo que disponga de un cliente VNC portado.

Para realizar esta conexión VNC se va a utilizar Remmina [20] que es un cliente de escritorio remoto para Linux escrito en GTK+, permite conexión remota a otro ordenador utilizando VNC sin la necesidad de instalar software adicional en el ordenador cliente.

La conexión VNC de los dos ordenadores se realiza a través del protocolo de comunicación TCP/IP. Para ello se crear una red LAN inalámbrica propia a la que deberá estar conectado el robot y el PC con el programa Remmina.

Para poder conectar el robot en la red deseada, la primera vez se tendrá que disponer de unos periféricos externos, y conectar al Turtlebot directamente a un monitor, un teclado y un ratón. Una vez se establece la conexión a esta red, cada vez que se encienda al robot Turtlebot, éste se conectará a dicha red directamente sin necesidad de utilizar dichos periféricos ni realizar ninguna acción adicional.

El robot Turtlebot está configurado con IP estáticas, esto significa que, para cada red, tiene una IP asignada, por lo que siempre que esté conectado a esa red, tendrá siempre la misma IP.

También se tiene que configurar el servidor VNC. Para esta configuración solo hay que configurar la contraseña que permite esta conexión.

A continuación, tenemos un ejemplo de conexión entre los dos ordenadores a partir del programa Remmina. Hay que especificar el tipo de conexión, la IP del robot y su contraseña.

Una vez se ha conectado, como se puede observar en la figura 3.1, se tendrá el escritorio remoto del ordenador del Turtlebot.

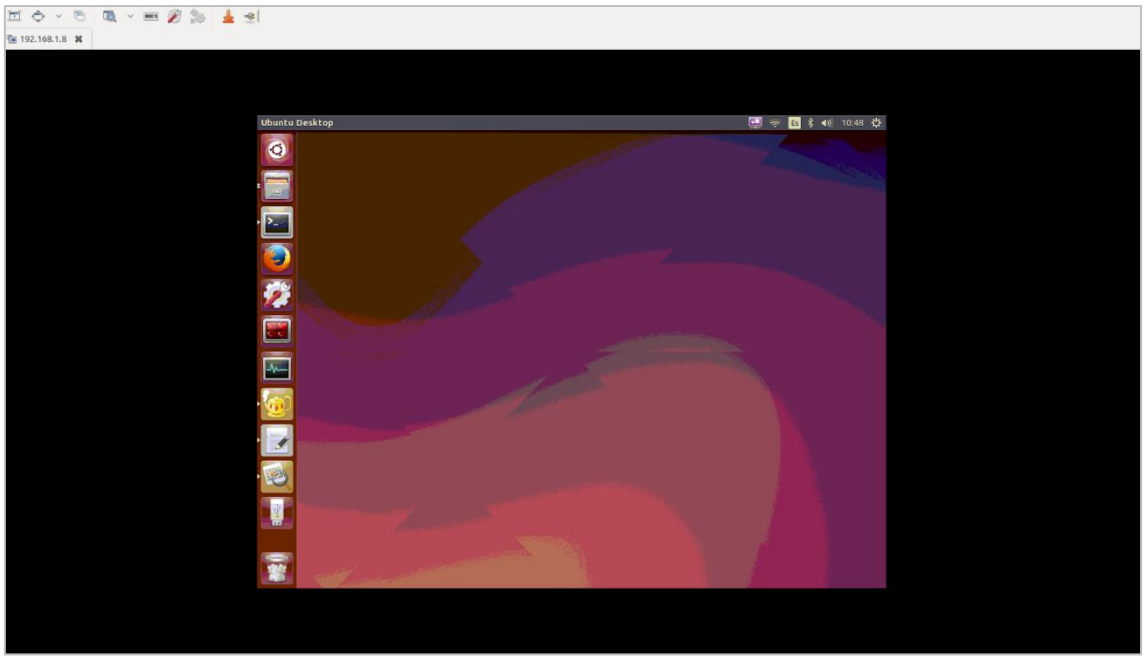

Figura 3.1: Escritorio remoto del Turtlebot

Este escritorio pertenece al ordenador del robot Turtlebot, por lo que cualquier acción realizada, afectará directamente al robot.

3.1.1.2 Protocolo SSH

También existen otros protocolos que también permiten esta conexión remota, como por ejemplo SSH. SSH, [21] (o *Secure Shell*) es el nombre de un protocolo y del programa que lo implementa cuya principal función es el acceso remoto a un servidor por medio de un canal seguro en el que toda la información está cifrada.

Se utilizará este protocolo a partir de un comando en la terminal de nuestro PC. Para poder realizar la conexión a partir de este protocolo, ambos ordenadores tendrán que tenerlo instalado.

SSH, utiliza como información, el usuario del robot, la IP del Turtlebot y la contraseña sudo del PC del Turlebot.

El comando para la conexión remota es el siguiente:

```
>> ssh usuario@ip
```

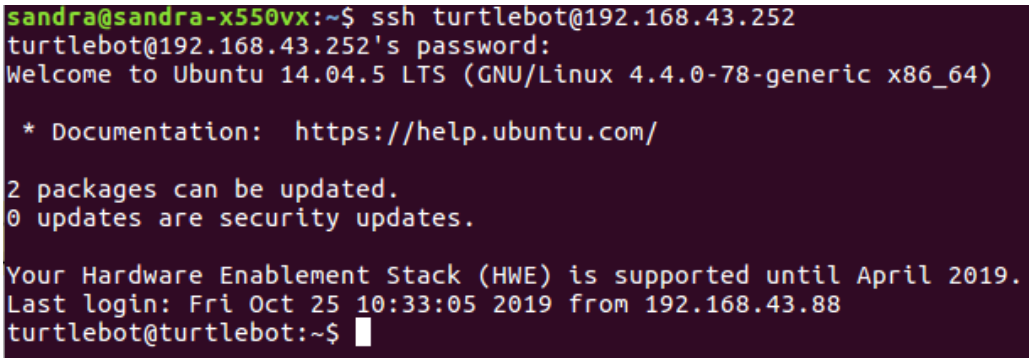

Figura 3.2: Conexión a la terminal del PC del Turtlebot

Como se puede observar en la figura 3.2, después de ejecutar el comando SSH, se conecta al robot y se abre la terminal del robot en nuestro PC personal.

Con este protocolo, no tendremos una visualización del escritorio del PC del robot, sino que tendremos acceso a su consola. Si se necesitase otra terminal, habría que ejecutar otra vez el comando SSH en otra terminal de nuestro PC.

3.1.2 Ros Master

El objetivo de ROS, es facilitar la comunicación entre sus distintos módulos, llamados nodos. Básicamente, los nodos son procesos que realizan alguna acción. Los nodos en sí son realmente módulos de software, pero con la capacidad de registrarse con el nodo ROS Master y comunicarse con otros nodos en el sistema. La idea de diseño de ROS es que cada nodo es un módulo independiente que interactúa con otros nodos utilizando la capacidad de comunicación de ROS.

La comunicación que se establece entre los nodos es gracias al ROS Master [22]. El ROS Master proporciona servicios de nombres y registros a los nodos en el sistema ROS y realiza un seguimiento de los publicadores y suscriptores de los topics. El papel del maestro es permitir que los nodos ROS individuales se ubiquen entre sí, para comunicarse entre sí de igual a igual.

Para crear un ROS Master, se utiliza la herramienta *roscore*. Es una colección de nodos y programas que son requisitos para un sistema basado en ROS. Se debe tener un *roscore* ejecutándose con el objetivo de la comunicación entre nodos.

```
>> roscore
```

Este comando iniciará:

- un Ros Master
- un Servicio de Parámetros en ROS
- un Rosout

Si se utiliza el comando *roslaunch*, automáticamente se iniciará *roscore* si se detecta que no está activo.

3.1.3 Nodos ROS

Para poder enviar instrucciones al Turtlebot y recibir información de sus sensores y odometría se deben crear nodos de comunicación, con distintos topics y mensajes [23].

En este apartado se va a describir cómo activar las funcionalidades del Turtlebot mediante la operación de activación que inicia todos los sensores y actuadores del robot.

Como se ha explicado en el apartado 2.1, el robot Turtlebot se compone de tres componentes diferentes, la base móvil Kobuki, el láser Hokuyo y la cámara Astra.

Para poder utilizarlos hay que lanzar por terminal estos tres comandos, que habilitan su utilización:

```
>> roslaunch turtlebot_bringup minimal.launch
>> roslaunch turtlebot_bringup hokuyo_ust10lx.launch
>> roslaunch astra_launch astra.launch
```

El fichero *minimal.launch* habilita la base móvil del robot. *Hokuyo_ust10lx.launch* habilita el láser. El fichero *astra.launch* es para la utilización de la cámara Astra.

Estos comandos se deben ejecutar cada vez que se encienda el robot Turtlebot. Estos comandos crean un nodo para cada dispositivo que se conecta al ROS master.

A continuación se van a utilizar una serie de herramientas de ROS que permiten ver y comprobar el resultado de la activación de los sensores y actuadores del robot Turtlebot al lanzar los 3 archivos *.launch*.

Rosnode [24] es una herramienta por línea de comandos que muestra información sobre los nodos ROS. Incluyendo publicadores, suscriptores y conexiones.

Con el siguiente comando se pueden observar los nodos ROS que se encuentran activos.

```
>> rosnode list
```

Cómo ejemplo, en la figura 3.3 se puede ver la lista de nodos que se obtiene con este comando tras haber ejecutado el fichero *minimal.launch* que habilita la base Kobuki.

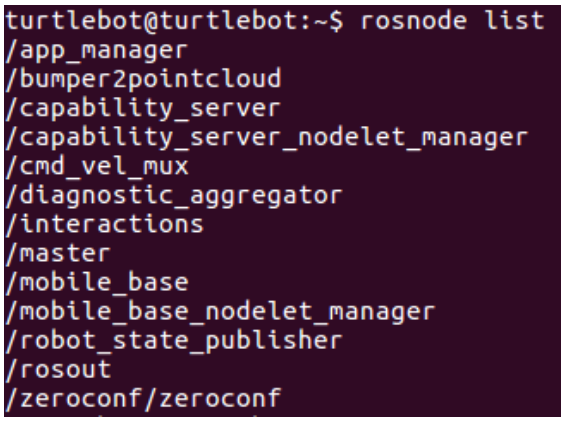

Figura 3.3: Lista de los nodos activos tras ejecutar minimal.launch

Por otro lado*, rqt_graph* [25] es una herramienta muy útil para ver lo que sucede en el gráfico ROS. Es un complemento GUI del conjunto de herramientas *Rqt*.

Con *rqt_graph* se obtiene una visión global del sistema ROS. Esta herramienta también permite comprobar la comunicación entre los nodos y detectar posibles problemas.

```
>> rqt graph
```

En la figura 3.4, tenemos el gráfico de las conexiones de los nodos tras ejecutar el fichero *minimal.launch* creado por la herramienta *rqt_graph*.

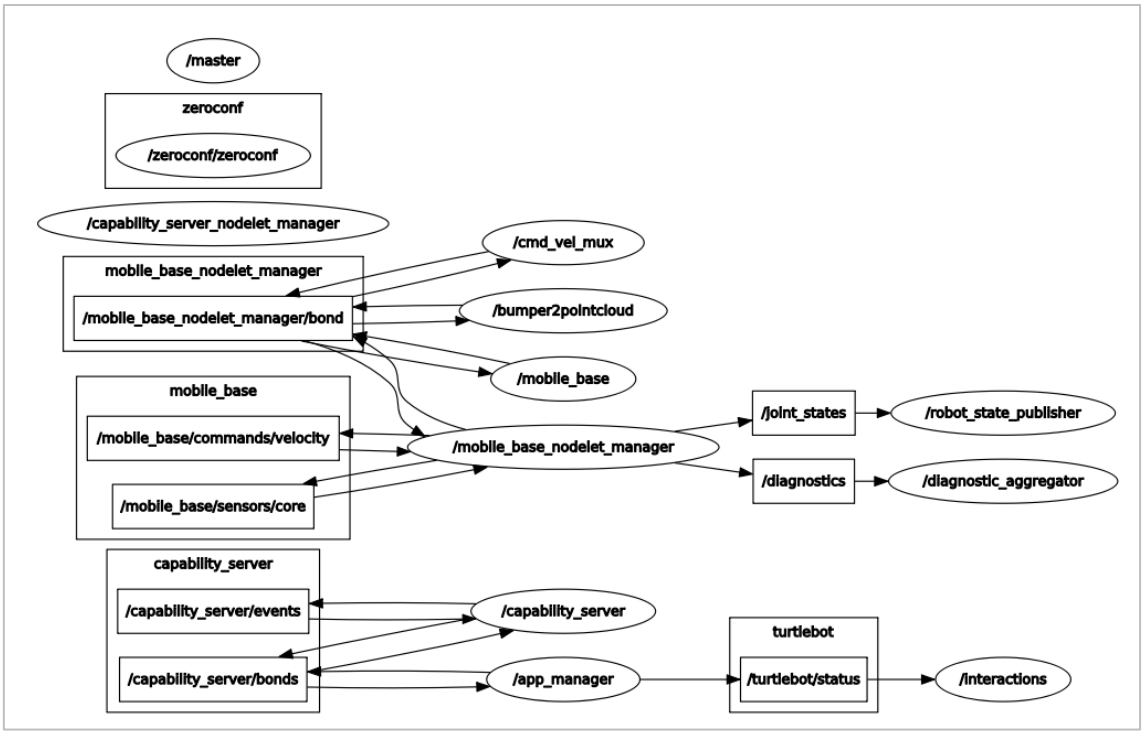

Figura 3.4: Gráfico de nodos activos del paquete minimal.launch

Rostopic [26] es otra herramienta por línea de comandos que muestra información de los topics de ROS, incluidos los publicadores, suscriptores y mensajes ROS. Con el siguiente comando, se obtiene una lista de los topics, publicadores y suscriptores, que se encuentran activos.

```
>> rostopic list
```

```
turtlebot@turtlebot:~$ rostopic list
/capability_server/bonds
/capability_server/events
/cmd_vel_mux/active
/cmd_vel_mux/input/navi
/cmd_vel_mux/input/safety_controller
/cmd_vel_mux/input/switch
/cmd_vel_mux/input/teleop
/cmd_vel_mux/parameter_descriptions
/cmd_vel_mux/parameter_updates
/diagnostics
/diagnostics_agg
/diagnostics_toplevel_state
/gateway/force_update
/gateway/gateway_info
/info
/interactions/interactive_clients
/interactions/pairing
/joint_states
/mobile_base/commands/controller_info
```

Figura 3.5: Lista de los topics activos tras ejecutar minimal.launch

En la figura 3.5 se muestra la lista de los topics activos tras ejecutar *minimal.launch*.

En el anexo 1 se tiene la salida por pantalla del comando *rostopic* al ejecutar cada uno de los comandos *roslaunch* que habilitan los sensores y actuadores, para conocer qué topic pertenece a cada paquete y poder comprobar que no hay ningún problema y los dispositivos se han activado correctamente.

3.1.4 Primeras pruebas

Antes de utilizar cualquier sensor o actuador del Turtlebot, es conveniente verificar que estos funcionan como se esperaba y que se comportan correctamente. En este apartado se verán una serie de pasos para comprobar y verificar las funcionalidades del Turtlebot.

Lo primero que se debería hacer con el Turtlebot es comprobar que todos sus sensores y actuadores funcionan correctamente. En este apartado se verán una serie de pasos para comprobar y verificar algunas de las funcionalidades del Turtlebot.

Una vez tenemos los nodos lanzados, se van comprobar sus valores directamente por terminal. También se puede publicar en ellos si son *publishers*, como, por ejemplo, en el topic que teleopera el robot. Y se va a lanzar un paquete, concretamente el paquete *Keyboard Teleop*, que permite la teleoperación de la base móvil del robot Turtlebot.

3.1.4.1 Comprobación de los sensores y actuadores

A continuación, tenemos una serie de comandos de la herramienta *rostopic* comentada en el apartado 3.1.3 que permiten trabajar directamente sobre la terminal del Turtlebot sin la necesidad de ejecutar ningún paquete.

Para mostrar los valores publicados en un topic:

```
>> rostopic echo /topic_name
```

Para mostrar la información de un topic:

```
>> rostopic info /topic_name
```

Para mostrar el tipo de mensaje del topic. El tipo de mensaje del topic es necesario conocerlo para crear un subscriptor o un publicador de ese topic. Por ejemplo, para ver qué tipo de valores se pueden escribir.

```
>> rostopic type /topic_name
```

Para publicar un mensaje en un topic se utiliza el siguiente comando:

```
>> rostopic pub /topic_name std_msgs/String hello
```

A continuación, se va a ver cómo utilizar estos comandos en distintos sensores y actuadores en unas pruebas iniciales para comprobar su correcto funcionamiento.

Testear los bumpers

Hay tres sensores bumpers en la base Kobuki: uno en el frente, otro en el lado derecho y otro en el izquierdo. Los sensores bumpers ayudan al Turtlebot a detectar colisiones con obstáculos. A continuación, vamos a testear los bumpers:

```
>> rotopic echo /mobile_base/events/bumper
```

Deberíamos observar la siguiente salida:

```
state: 0
bumper: 0
```

Esto significa que todos los bumpers se encuentran activados y publicados, pero no presionados.

La variable state puede tener como valores 0 (no presionado) y 1 (presionado). La variable bumper puede tener los valores 0 (izquierda), 1 (centro) y 2 (derecha)

A continuación, para chequear el bumper central, vamos a presionarlo. Deberíamos observar la siguiente salida por el terminal:

```
state: 1
bumper: 1
```

Se debería repetir el proceso para los otros dos bumpers para asegurarnos de que funcionan correctamente.

Testear la cámara

A través de la terminal, también se puede visualizar la imagen que está capturando la cámara del Turtlebot en tiempo real [27].

```
>> rosrun image_view image_view image:=/camera/rgb/image_raw
```

Este comando abre una ventana con la imagen que se está capturando en tiempo real.

Figura 3.6: Captura de imagen de la cámara del robot Turtlebot

También hay un comando que permite grabar un video de la imagen que la cámara está capturando.

```
>> rosrun image_view video_recorder image:=/camera/rgb/image_raw
```

Testear el láser

A través de la terminal, también se pueden visualizar los datos del sensor láser. Este topic a parte de devolver un vector con los datos del láser, también devuelve las especificaciones del sensor. Como el ángulo mínimo y el máximo que el láser es capaz de leer, junto con el valor del incremento del ángulo que lee. El rango mínimo y máximo de la distancia que el láser es capaz de medir.

Figura 3.7: Lectura de los valores del láser Hokuyo

El sensor devuelve un valor NaN cuando en ese ángulo, el obstáculo no se encuentra dentro del valor máximo o mínimo que es capaz de leer el láser. Es decir, que no hay ningún objeto entre el 0.4 y 10 metros, pero que este objeto puede estar o muy próximo o muy lejos del robot.

El sensor no es capaz de distinguir dónde se encuentra el obstáculo, pero se puede llegar a asumir dónde está, dependiendo del entorno en el que se encuentra el robot. Por ejemplo, si el entorno es muy cerrado o se encuentra dentro de algún edificio, se puede asumir que el objeto se encuentre cerca dado que es bastante improbable que el láser no lea nada en 10 metros.

3.1.4.2 Paquetes ROS

En este apartado, se va a mostrar cómo ejecutar un paquete, concretamente el paquete que permite la teleoperación de la base móvil del robot Turtlebot. Mediante este paquete se comprueban las movilidades del robot.

El paquete *keyboard_teleop* [28] permite controlar el robot TurtleBot a través de la comunicación remota con el teclado de nuestro ordenador. Este paquete proporciona archivos *.launch* para la teleoperación con diferentes dispositivos de entrada, como un joystick, el teclado… Este paquete sirve tanto para el robot real como el simulado.

Para ejecutar el paquete, hay que escribir por terminal el siguiente comando:

```
>> roslaunch turtlebot_teleop keyboard_teleop.launch
```

Se debería obtener la siguiente salida, donde se muestran las instrucciones para teleoperar el Turtlebot, el cual se puede mover hacia delante y hacia atrás, rotar o aumentar/disminuir la velocidad.

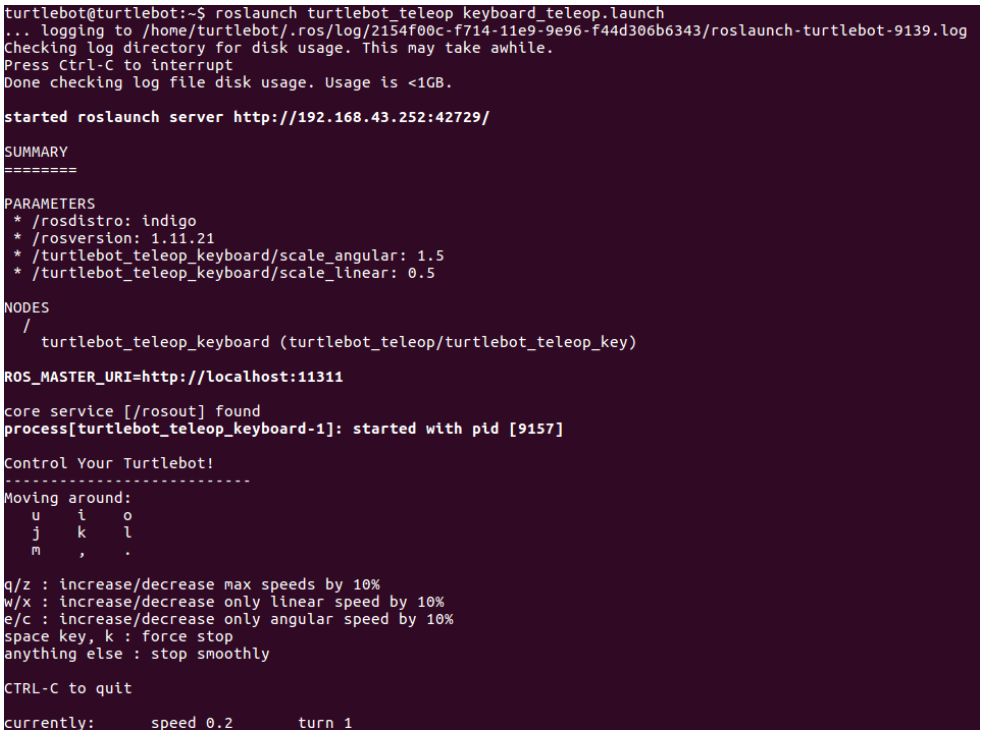

Figura 3.8: Ejecución del paquete keyboard_teleop

El paquete publica diferentes valores en el topic */cmd_vel_mux/input/teleop* dependiendo de las instrucciones que el usuario le indica, en este caso, a partir de una serie de teclas.

3.1.4.3 Publicar en un topic

A continuación, se va a explicar cómo publicar en un topic. En este caso, sobre el mismo topic que publica el paquete *keyboard_teleop* utilizado en el apartado anterior.

El primer paso es obtener información sobre el topic, la cual va a ser necesaria para publicar un mensaje sobre este topic. En concreto se necesita el tipo de mensaje que publica en el topic y la estructura del mensaje.

Con la herramienta *rostopic info* se puede obtener información general de los topics activos. En concreto, este comando devuelve el tipo de mensaje del topic, los *publisher* que se encuentran publicando en este topic, y también sus *subscribers*.

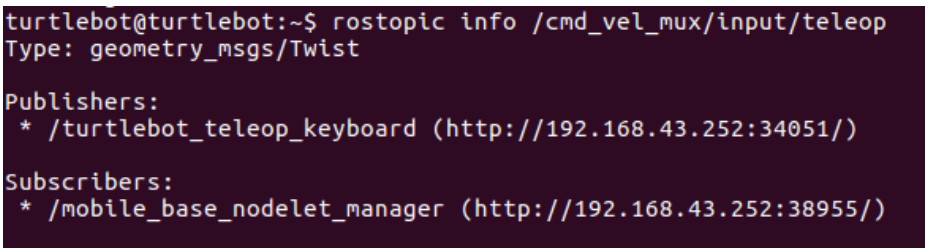

Figura 3.9: Información general del topic /cmd_vel_mux/input/teleop

En este momento, en la figura 3.9, se observa un publicador, el cual es el paquete de teleoperación lanzado antes. Y un suscriptor, que es el nodo encargado de la base móvil del robot.

Con el siguiente comando también se puede obtener el tipo de mensaje de los topics:

```
>> rostopic type </topic>
```

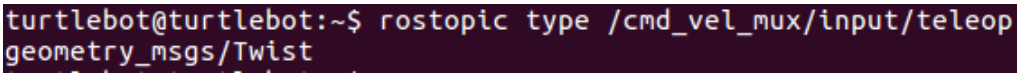

Figura 3.10: Tipo de mensaje del topic /cmd_vel_mux/input/teleop

Como se observa en la figura 3.10, el tipo del mensaje que utiliza el topic es *geometry_msgs/Twist*.

Con el comando *rosmsg* podemos ver los detalles del tipo de mensaje. Su estructura, y el tipo de dato que publica.

```
>> rosmsg show <message_type>
```

```
turtlebot@turtlebot:~$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Figura 3.11: Estructura del mensaje geometry_msgs/Twist

Como se puede observar en la figura 3.11, el mensaje del topic se compone de 6 variables. Correspondientes a los ejes X, Y y Z del movimiento lineal y angular del Turtlebot.

Con el siguiente comando, se observan los datos que están siendo publicados en el topic especificado. Este comando también permite ver la estructura del mensaje del topic y los datos que se suelen publicar en el topic.

```
>> rostopic echo </topic>
```

```
turtlebot@turtlebot:~$ rostopic echo /cmd_vel_mux/input/teleop
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Figura 3.12: Salida por pantalla del topic /cmd_vel_mux/input/teleop

Se observa la misma estructura que previamente se había observado con el comando *rosmsg* al obtener los detalles del mensaje.

Antes de escribir sobre cualquier topic, es necesario observar los valores usuales que se suelen publicar. Por lo que se realizan una serie de pruebas, donde se teleopera el robot con el paquete *keyboard_teleop* controlando el robot para que realice todos los movimientos posibles, y se muestra la información que este paquete escribe en el topic.

En la primera imagen es un *echo* del Turtlebot avanzando hacia delante. La segunda cuando el robot gira a la derecha. La tercera, cuando avanza recto y mientras gira a la izquierda.

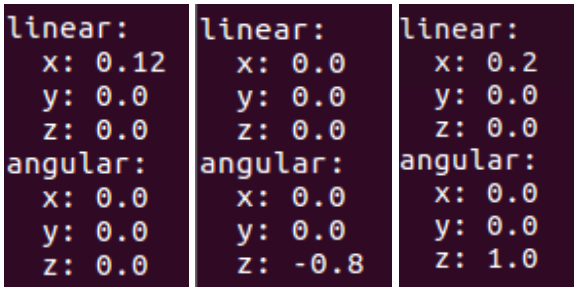

Figura 3.13: Valores publicados en el topic /cmd_vel_mux/input/teleop tras diferentes movimientos

Se observa que las únicas variables en las que se publican los valores, es *Linear.X* y *Angular.Z*, que controlan el movimiento lineal y el movimiento angular respectivamente. Dependiendo de la velocidad que se le indique, aumentará el valor que escribe sobre cada una de las variables. También se observa que el giro a la derecha es negativo en vez de ser en la izquierda como usualmente es.

Por último, se publica un mensaje en el topic sobre cada variable del topic de la misma forma que publica el paquete. En este comando, se necesita toda la información del topic obtenida antes con los comandos.

```
>> rostopic pub </topic> <topic_type> [data]
```

```
turtlebot@turtlebot:~$ rostopic pub /cmd_vel_mux/input/teleop geometry_msgs/Twist
'{linear: {x: 0.1}, angular:{z: 0.0} }'
publishing and latching message. Press ctrl-C to terminate
```

Figura 3.14: Publicar en el topic /cmd_vel_mux/input/teleop

Este mensaje que se publica en el topic, solo se publica una vez, mientras que en el paquete Turtlebot hay un bucle que está publicando a una frecuencia determinada el valor que el usuario le indica con las teclas.

3.2 Turtlebot en simulación

Es muy interesante poder trabajar a partir de simuladores, lo que permite testear diferentes comportamientos en diferentes entornos. Y posteriormente realizar pruebas con el robot real una vez se tenga el algoritmo bastante desarrollado y con el comportamiento del Turtlebot ajustado.

En este apartado vamos a ver cómo utilizar herramientas diferentes para simular nuestro Turtlebot, en concreto el simulador 3D Gazebo, junto con el visualizador Rviz de Ros.

3.2.1 Simulador Gazebo

En gazebo se pueden crear diferentes escenarios o mundos 3D para poder simular los robots en los entornos en los que finalmente va a trabajar y así adecuar su comportamiento a estos.

Este simulador de robots permite probar rápidamente algoritmos, diseñar robots, realizar pruebas de regresión y entrenar el sistema de inteligencia artificial utilizando escenarios realistas. Gazebo ofrece la capacidad de simular con precisión y eficiencia poblaciones de robots en entornos complejos interiores y exteriores.

Para inicializar este simulador [29], junto con el robot Turtlebot, hay que escribir por terminal el siguiente comando:

```
>> roslaunch turtlebot_gazebo turtlebot_world.launch
```

Este comando lanza el paquete preinstalado *turtlebot_gazebo*. El cual abre Gazebo e inserta el Turtlebot en el mundo por defecto que está definido en el fichero *.launch* del paquete como se observa en la figura 3.15. También habilita todas las funcionalidades del Turtlebot, de la misma forma en la que se habilitan el Turtlebot real.

Este comando también inicia un ROS Master si detecta que no hay ninguno activo, por lo que el ROS Master se puede iniciar directamente con este comando o mediante la herramienta *roscore* explicada en el apartado 3.1.2

En el simulador, se ve un entorno, un mundo en Gazebo, con el Turtlebot y otros objetos. Esta es una escena de la parte principal del simulador, donde los objetos están animados y somos capaces de interactuar con el entorno.

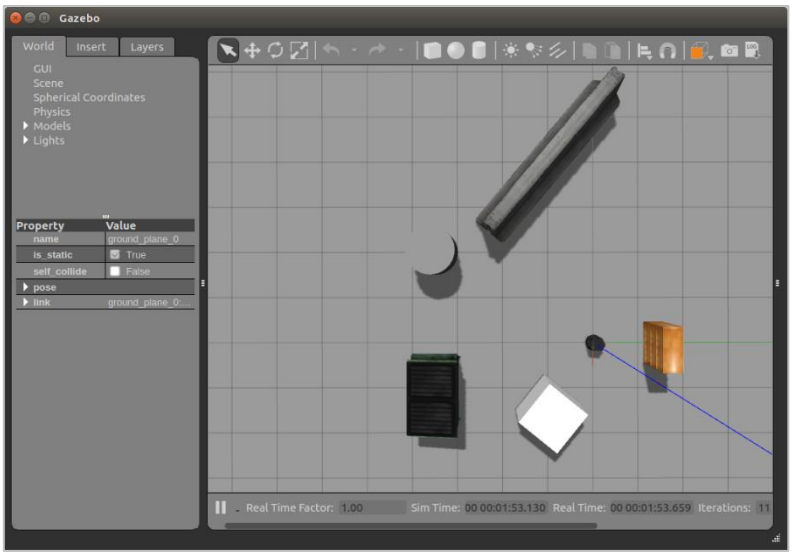

Figura 3.15: Simulación del robot Turtlebot en Gazebo

Para comprobar que todo se ha lanzado correctamente, comprobamos los topics activos en este momento:

```
sandra@sandra-x550vx:~$ rostopic list
/camera/depth/camera_info
/camera/depth/image_raw
/camera/depth/points
/camera/parameter_descriptions
/camera/parameter_updates
/camera/rgb/camera_info
/camera/rgb/image_raw
/camera/rgb/image_raw/compressed
/camera/rgb/image_raw/compressed/parameter_descriptions
/camera/rgb/image_raw/compressed/parameter_updates
/camera/rgb/image_raw/compressedDepth
/camera/rgb/image_raw/compressedDepth/parameter_descriptions
/camera/rgb/image_raw/compressedDepth/parameter_updates
/camera/rgb/image_raw/theora
/camera/rgb/image_raw/theora/parameter_descriptions
/camera/rgb/image_raw/theora/parameter_updates
/clock
/cmd_vel_mux/active
/cmd_vel_mux/input/navi
/cmd_vel_mux/input/safety_controller
/cmd_vel_mux/input/switch
/cmd_vel_mux/input/teleop
/cmd_vel_mux/parameter_descriptions
/cmd_vel_mux/parameter_updates
/depthimage_to_laserscan/parameter_descriptions
/depthimage_to_laserscan/parameter_updates
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gazebo_gui/parameter_descriptions
/gazebo_gui/parameter_updates
/joint_states
/laserscan_nodelet_manager/bond
/mobile_base/commands/motor_power
/mobile_base/commands/reset_odometry
/mobile_base/commands/velocity
/mobile_base/events/bumper
/mobile_base/events/cliff
/mobile_base/sensors/bumper_pointcloud
/mobile_base/sensors/core
/mobile_base/sensors/imu_data
/mobile_base_nodelet_manager/bond
/odom
/rosout
/rosout_agg
/scan
/tf
/tf_static
```

Figura 3.16: Lista de topics activos del Turtlebot simulado en Gazebo

Como se puede observar en la figura 3.16, el paquete *turtlebot_gazebo*, aparte de insertar el Turtlebot en el entorno 3D simulado, también crea los topics correspondientes a los sensores y actuadores del robot.

Este paquete, también permite especificar en qué mundo queremos realizar la simulación en Gazebo, con el argumento *world_file*, en vez de utilizar el mundo por defecto, o modificar el fichero *.launch*.

```
>> roslaunch turtlebot_gazebo turtlebot_world.launch world_file:=<full
path to the world file>
```

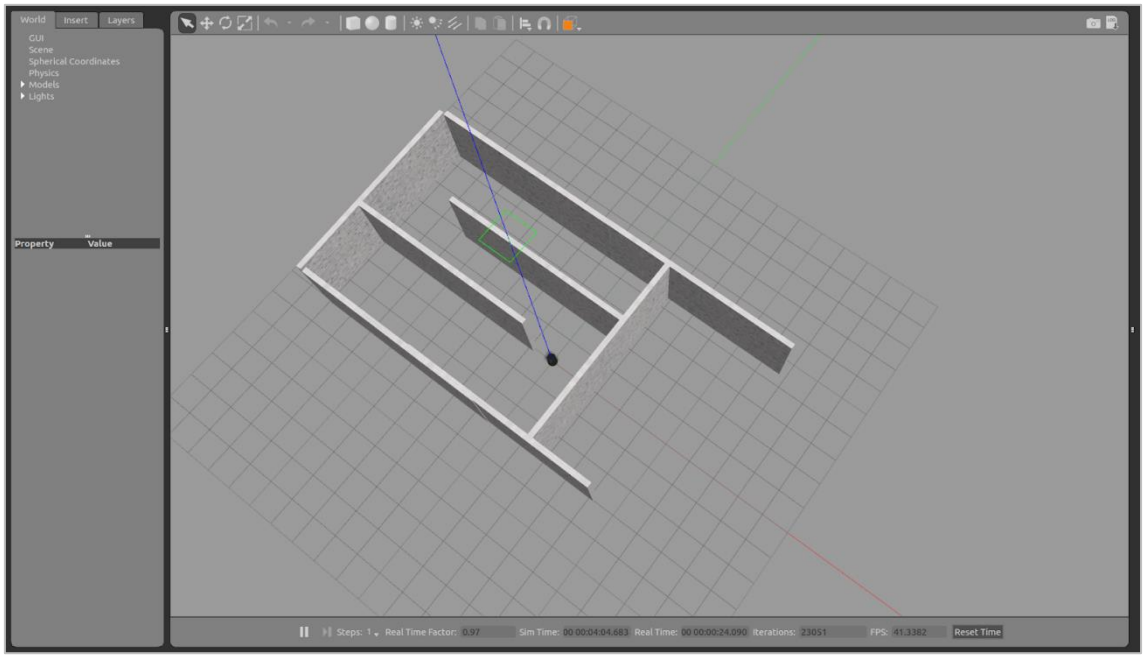

Figura 3.17: Simulación del Turtlebot en Gazebo con el mundo especificado

Como se ha comentado, el paquete *turtlebot_gazebo* se encarga de habilitar todos los sensores y actuadores del robot Turtlebot que va a simular. En este paquete también se define los tipos de sensores y actuadores que va a tener el robot simulado. Como se ha comentado en el apartado 2.1, el robot Turtlebot puede tener distintos sensores. Por ejemplo, la base móvil del robot puede ser Kobuki o Create. El sensor 3D también puede ser diferente, Astra o Kinect o Asus xtion pro.

En el robot real, cada uno de estos dispositivos se habilitan por un comando, por lo que directamente se especifica el tipo que se quiere utilizar. En el caso de la simulación, estos sensores ya se encuentran especificados en el paquete *turtlebot_gazebo*.

Para poder simular el robot Turtlebot de la manera más realista posible, se puede modificar este paquete y definir los dispositivos con el mismo tipo que utiliza el robot real.

A continuación, en la figura 3.18, se tiene la modificación del fichero *turtlebot_world.launch* para que el simulador gazebo habilite los mismos dispositivos en el Turtlebot simulado que los utilizados en el Turtlebot real.

```
1  <launch>
2    <arg name="world_file"  default="$(env TURTLEBOT_GAZEBO_WORLD_FILE)"/>
3
4    <arg name="base"      value="$(optenv TURTLEBOT_BASE kobuki)"/> <!-- create, roomba -->
5    <arg name="battery"   value="$(optenv TURTLEBOT_BATTERY /proc/acpi/battery/BAT0)"/>
6    <arg name="gui" default="true"/>
7    <arg name="stacks"    value="$(optenv TURTLEBOT_STACKS hexagons)"/>  <!-- circles, hexagons -->
8    <arg name="3d_sensor" value="$(optenv TURTLEBOT_3D_SENSOR kinect)"/>  <!-- kinect, asus_xtion_pro -->
9
10   <include file="$(find gazebo_ros)/launch/empty_world.launch">
11     <arg name="use_sim_time" value="true"/>
12     <arg name="debug" value="false"/>
13     <arg name="gui" value="$(arg gui)" />
14     <arg name="world_name" value="$(arg world_file)"/>
15   </include>
16
17   <include file="$(find turtlebot_gazebo)/launch/includes/$(arg base).launch.xml">
18     <arg name="base" value="$(arg base)"/>
19     <arg name="stacks" value="$(arg stacks)"/>
20     <arg name="3d_sensor" value="$(arg 3d_sensor)"/>
21   </include>
22
23   <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher">
24     <param name="publish_frequency" type="double" value="30.0" />
25   </node>
26
27   <!-- Fake laser -->
28   <node pkg="nodelet" type="nodelet" name="laserscan_nodelet_manager" args="manager"/>
29   <node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
30         args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet laserscan_nodelet_manager">
31     <param name="scan_height" value="10"/>
32     <param name="output_frame_id" value="/camera_depth_frame"/>
33     <param name="range_min" value="0.45"/>
34     <remap from="image" to="/camera/depth/image_raw"/>
35     <remap from="scan" to="/scan"/>
36   </node>
37 </launch>
```

Figura 3.18: Fichero turtlebot_world.launch

En la línea 2 se define el tipo de la base móvil. Por defecto se encuentra definida la misma base que en el robot real, en este caso Kobuki.

En la línea 8 se define el sensor 3D. Por defecto, está definido Kinect, hay que modificar la línea de la siguiente forma para que se habilite Astra.

```
 <arg name="3d_sensor" value="$(optenv TURTLEBOT_3D_SENSOR
astra)"/>
```

En estas dos líneas se definen el tipo, luego el paquete automáticamente llamará a los paquetes correspondientes a los tipos especificados.

El láser se encuentra definido de la línea 27 a la 36. El paquete turtlebot_gazebo no llama al fichero del láser que se le especifica, como hacía con los otros dos dispositivos. Si no

que crea un láser falso a partir de los datos de las imágenes de profundidad que se obtienen con el sensor 3D. En el paquete solo define el rango máximo y mínimo de este láser. Esto es debido a que la plataforma turtlebot se compone de una base móvil y de un sensor 3D, pero no se compone de un láser. El turtlebot real utilizado tiene el láser Hokuyo como un dispositivo añadido, por lo que los paquetes de ROS correspondientes al Turtlebot, no tienen definido ningún láser. Para asemejarlo al láser real, habría que modificar los valores de las variables que definen este láser falso, con los mismos valores que tiene el láser Hokuyo. Cómo el ángulo mínimo y máximos, la distancia mínima y máxima capaz de detectar…

3.2.2 Rviz

Rviz [30] es un programa que permite visualizar los datos obtenidos por los sensores del turtlebot de una forma gráfica. En este apartado se va a mostrar cómo utilizar este programa y cómo visualizar los datos de los sensores que tiene el robot Turtlebot activos.

Rviz es un programa que necesita mucha capacidad de cómputo, por lo que no se puede utilizar en el PC del Turtlebot Real. Existen dos opciones para poder utilizar este visualizador con los datos reales de los sensores y actuadores, a partir de datos ya capturados en un archivo *.bag*, o en tiempo real, realizando una conexión de los dos PC, donde ambos dos se encuentren conectados por el mismo ROS Master.

En este apartado se va a mostrar cómo utilizar el programa Rviz a partir de la simulación del robot en Gazebo. Aunque en el apartado 4.4.1.2 se mostrará cómo utilizar Rviz con el robot real.

Mientras Gazebo está simulando el turtlebot en un entorno, Rviz permite visualizar los datos que los sensores están capturando en tiempo real.

El programa se puede iniciar a partir del siguiente comando, que también inserta directamente el modelo del turtlebot y algunos de sus sensores.

```
>> roslaunch turtlebot_rviz_launchers view_robot.launch
```

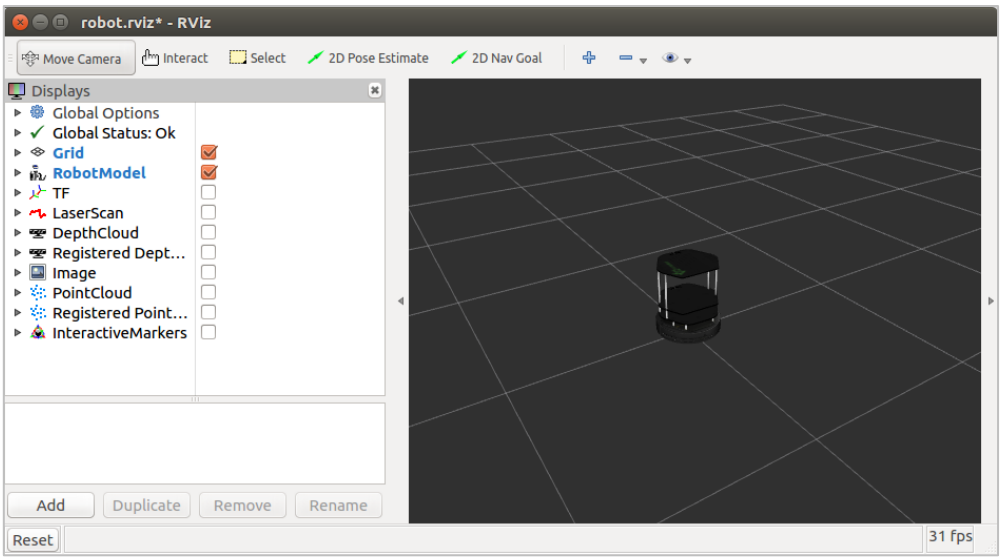

Figura 3.19: Turtlebot en el visualizador Rviz

En la parte izquierda del programa Rviz, se muestra los sensores disponibles que se pueden visualizar. Realmente lo que hace el programa es subscribirse a los topics y mostrarnos los datos de los sensores.

En algunas ocasiones, el visualizador Rviz, inicialmente no muestra la lista de sensores disponibles del robot, por lo que se tienen que añadir con el botón *add*.

Existen dos opciones al agregar: por dispositivo o por topic. Es mejor agregarlo por topic, porque por dispositivo puede que aparezca en la lista pero que realmente el sensor no esté lanzado y no haya topic, mientras que al agregar por topic, nos aseguramos de que el topic si se encuentra lanzado.

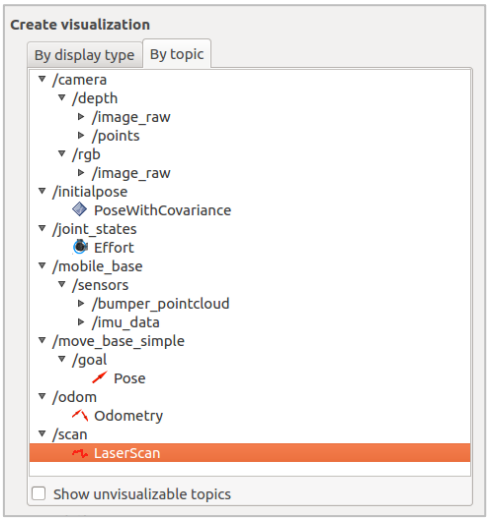

Figura 3.20: Lista de topics activos

Puede ocurrir que el sensor no tenga bien o no tenga ningún topic asignado, por lo que también debemos escribirlo nosotros. O que tenga varios topic asignados y que nosotros elijamos el topic que deseemos visualizar.

Una vez añadido el sensor, el programa lo añade a la lista de displays disponibles. Se puede observar el topic al que está subscrito, también se puede configurar la forma en que se quiere visualizar los datos proporcionados por el topic:

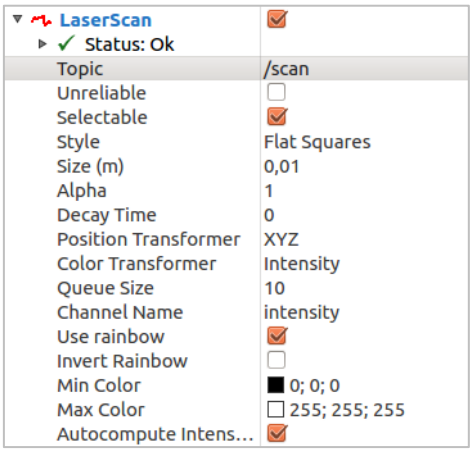

Figura 3.21: Display LaserScan en Rviz

A continuación, marcamos varios sensores y observamos como la herramienta de Rviz representa sus datos. Justo debajo de la lista, se imprime la imagen en tiempo real que está capturando el Turtlebot.

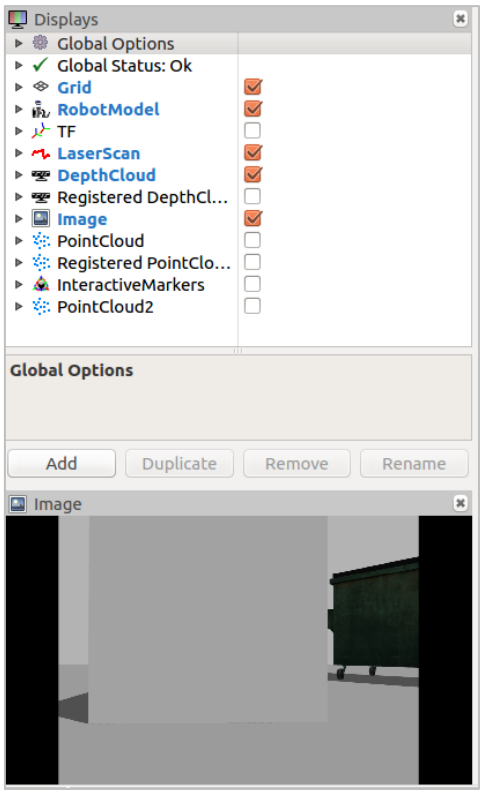

Figura 3.22: Lista de displays de Rviz e imagen en tiempo real

También, se puede observar los datos del sensor láser del Turtlebot, el cual, como se puede ver en la imagen de la derecha del simulador gazebo, está en dirección al cubo gris. Rviz, en la imagen de la izquierda, representa la distancia que está leyendo el láser en todo su rango.

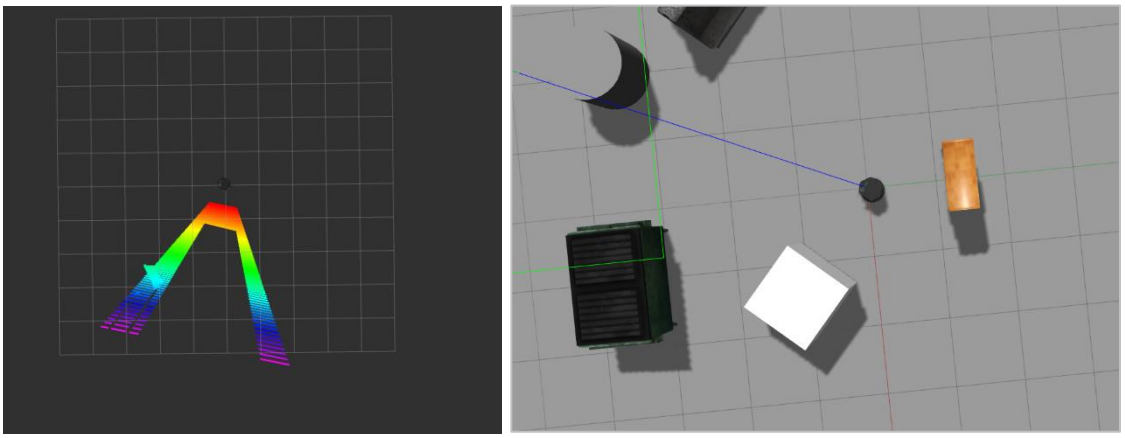

Figura 3.23: Izquierda, captura del láser en Rviz. Derecha, posición y orientación del Turtlebot en Gazebo

A través del programa Rviz también se puede teleoperar el turtlebot a partir de unos marcadores interactivos [31].

Mientras los programas Gazebo y Rviz están corriendo, ejecutar en otra terminal:

```
>> roslaunch turtlebot_interactive_maker interactive_makers.launch
```

Este comando lanza un topic */turtlebot_maker_server/update* el cual es un subscriptor y un publicador del topic */mobile_base_nodelet_manager*, que es el que maneja todos los topics relacionados con los movimientos del turtlebot.

Para poder teleoperar al Turtlebot desde Rviz, hay que insertar la herramienta y habilitarla de la misma forma que con los sensores.

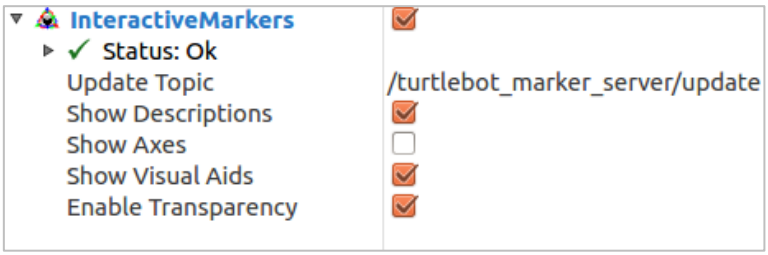

Figura 3.24: Display InteractiveMakers en Rviz

Al habilitar la opción de *InteractiveMakers* en Rviz, alrededor del Turtlebot aparecerán una especie de ejes, un anillo azul y unas flechas rojas como se observa en la figura 3.25.

Al arrastrar las flechas rojas, podemos mover el turtlebot hacia delante o hacia atrás. Al arrastrar el círculo azul, podemos rotar el robot, y también se puede rotar y conducir simultáneamente.

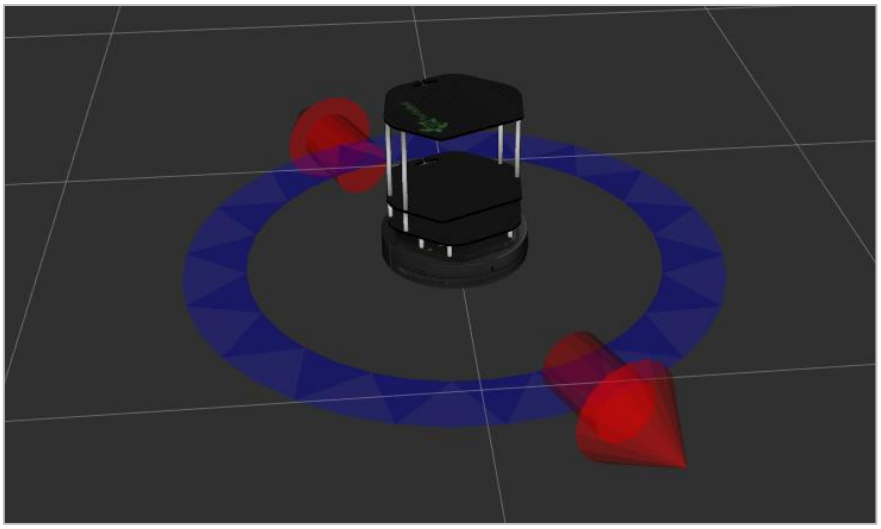

Figura 3.25: Interactive makers

4. Experimentación realizada

En este apartado se van a realizar distintas pruebas con el robot TurtleBot en el entorno real y en el simulado. Estos van a ser comportamientos un poco más complejos, a través de distintas herramientas que proporciona ROS. En concreto,

- Cómo crear nuestro espacio de trabajo y cómo crear un paquete. Este paquete consistirá en realizar un comportamiento con el turtlebot a través de código.
- Cómo crear un paquete que simule varios robots Turtlebots en Gazebo.
- Cómo crear un mundo en Gazebo.
- Cómo construir un mapa y posteriormente cómo localizarse en el mapa, en simulación y con el robot real.
- Cómo realizar una conexión entre el turtlebot y el programa Matlab que leerá y analizará los datos de los sensores en tiempo real.

4.1 Creación de espacio de trabajo y creación de paquete Ros

En primer lugar, en este apartado se va a explicar cómo crear nuestro espacio de trabajo el cual, después, es el que se utilizará para crear y ejecutar nuestros propios paquetes.

A continuación, se creará un paquete ros en este espacio de trabajo, el cual será un código escrito en C++, donde se leerán los datos de distintos sensores del turtlebot, y se moverá al robot dependiendo de dichos valores, publicando en el topic adecuado.

4.1.1 Creación del espacio de trabajo

Un espacio de trabajo catkin [32] es un directorio en el que se puede crear o modificar paquetes catkin existentes. Un espacio de trabajo catkin contiene tres subdirectorios diferentes (/*build*, /*devel* y /*src*), cada uno de los cuales cumple una función diferente en el proceso de desarrollo de software.

Este espacio de trabajo solo es necesario crearlo una primera vez, ya que, a partir de este *workspace*, se podrá trabajar con cualquier paquete que se desee.

Para crear el espacio de trabajo *catkin*, hay que escribir una serie de comandos en la terminal. Para trabajar con el espacio de trabajo, se creará un espacio de trabajo en nuestro ordenador personal y otro en el robot turtlebot.

El primer paso para crear nuestro espacio de trabajo, es crear dos carpetas, la principal *catkin_ws* y dentro de ella, *src*:

```
>> mkdir -p catkin_ws/src
```

Después, dentro de la carpeta *catkin_ws*, ejecutamos el siguiente comando:

```
>> catkin_make
```

El comando *catkin_make*, creará un *CMakeLists.txt* en la carpeta *src*. Además, creará las carpetas *build* y *devel*. Dentro de la carpeta *devel* puede ver que ahora hay varios archivos de configuración *.sh.

Antes de continuar trabajando, deberemos ejecutar el siguiente comando. El archivo *setup.bash* simplemente agrega variables de entorno a la ruta actual para permitir que ROS funcione:

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

Este comando lo que hace es modificar el valor de la variable *ROS_PACKAGE_PATH* asignándole la ruta del directorio actual. Este cambio en la variable se realiza de manera local, es decir, sólo lo modifica en la terminal en la que se ejecuta. Si se abre otra consola, se tendrá que volver a ejecutar este comando.

4.1.2 Creación del nuevo paquete ROS

Como ya se ha comentado, en ROS, los nodos se distribuyen en paquetes, por lo que, para crear un nodo, se necesita crear un paquete.

En este apartado se va a crear un nuevo paquete, el cual creará un nodo que controlará los movimientos del robot Turtlebot dependiendo de los datos que se capturen de distintos sensores.

Los paquetes se crean con el comando *catkin_create_pkg* [33], y debe ejecutarse en la carpeta *src* del espacio de trabajo:

```
>> catkin_create_pkg package_name [required packages]
```

catkin_create_pkg requiere el nombre del paquete y, opcionalmente, una lista de dependencias de las que depende ese paquete.

En este caso, se ha creado el paquete *mi_paquete* con la dependencia *roscpp*, que es una librería básica de C++ para ROS.

Se creará una carpeta con el nombre del paquete y los ficheros *CMakeList.txt* y *package.xml*.

- *CMakeLists.txt* - son instrucciones de compilación del paquete.
- *package.xml* - contiene metadatos del paquete, como el autor, la descripción, la versión, los paquetes requeridos…

A continuación, en este paquete se van a crear dos archivos de código escritos en C++, donde se crearán un nodo publicador y un nodo suscriptor [34].

Se crean los ficheros *publisher.cpp* y *subscriber.cpp* dentro de la carpeta *src* del paquete creado: */catkin_ws/src/mi_paquete/src.*

En el fichero *publisher.cpp,* se creará un nodo publicador llamado *test_publisher* que escribirá sobre el topic */contador*.

```cpp
#include <ros/ros.h>
#include "std_msgs/Int32.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "publish_node");
    ros::NodeHandle n;
    ros::Rate loop_rate(50);

    ros::Publisher pub_contador = n.advertise<std_msgs::Int32>("/contador",10);

    int count = 0;

    std_msgs::Int32 msg;

    while (ros::ok())
    {
        msg.data = count;
        ROS_INFO("%i", msg.data);

        pub_contador.publish(msg);


        ros::spinOnce();
        loop_rate.sleep();

        count++;
    }
}
```

Figura 4.1: Código del fichero publisher.cpp

A continuación, se van a explicar las líneas de código del fichero *publisher.cpp*

```
ros::init(argc, argv, "test_publisher");
```

Inicialización del nodo ROS *test_publisher*. Esta función contacta con el maestro ROS y registra el nodo en el sistema.

```
Ros::NodeHandle nh;
```

Crea un *handle* para el proceso del nodo.

```
ros::Rate loop_rate(50);
```

Permite especificar la frecuencia a la que se quiere crear el bucle, en este caso se define a 50 Hz.

```
ros::Publisher pub_contador =
nh.advertise<std_msgs::String>("contador", 1000);
```

Se crea un publicador que publicará mensajes de tipo *std_msgs/String* sobre el topic */contador*. El segundo argumento es el tamaño de la cola de publicación. En este caso, si estamos publicando demasiado rápido, almacenará un máximo de 1000 mensajes antes de empezar desechar los mensajes más antiguos.

```
while( ros::ok() ){
```

Se comprueba si ROS está activo. Si ROS Master se encuentra parado, o hay una señal de paro en el sistema, esta función devolverá false.

```
std_msgs::String msg;
std:: stringstream ss = "";
ss << count;
msg.data = ss.str();
```

Crea una variable (*msgs*) del mismo tipo que el publicador, y le asigna el valor de la variable *count* al campo *data* del mensaje.

```
ROS_INFO("%s", msg.data.s_str());
```

ROS_info es el equivalente a *printf/cout*.

```
pub_contador.publish(msg);
```

Se publica el mensaje en el topic especificado en el publicador.

```
ros::SpinOnce();
loop_rate.sleep();
```

Se espera hasta que el tiempo definido de 10Hz haya pasado.

A continuación, se crea el nodo suscriptor en el fichero *subscriber.cpp*. Este nodo, se suscribirá al topic donde se publican las imágenes capturadas por la cámara del Turtlebot: /camera/rgb/image_raw.

Esta imagen se mostrará por pantalla y también se guardará.

El código del fichero *subscriber.cpp* es el siguiente:

```cpp
#include "ros/ros.h"
#include <image_transport/image_transport.h>
#include <cv.h>
#include <cv_bridge/cv_bridge.h>
#include <opencv2/highgui/highgui.hpp>

using namespace cv;
using namespace std;

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr;

    try
    {
        cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }

        imwrite("view.png", cv_ptr->image);
        imshow( "view", cv_ptr->image);

        cv::waitKey(30);
}

int main(int argc, char **argv)
{
   ros::init(argc, argv, "subscriber_node");
   ros::NodeHandle nh;

   image_transport::ImageTransport it(nh);
   image_transport::Subscriber sub = it.subscribe("/camera/rgb/image_raw", 1, imageCallback);

   while (ros::ok())
   {
     ros::spin();
   }

   return 0;
}
```

Figura 4.2: Código del fichero subscriber.cpp

Se necesita una función por cada suscriptor. En este caso *imageCallback()* es la función que se llamará cuando el suscriptor reciba un mensaje del topic al que está suscrito.

Esta es la función que se llamará cuando llegue una nueva imagen sobre el topic */camera/rgb/image_raw*. Aunque la imagen puede haber sido enviada en algún tipo de mensaje de transporte, solo se necesita manejar el tipo normal *sensor_msgs/Image*. Toda la codificación/decodificación de imágenes se maneja automáticamente.

```
cv_bridge::CvImagePtr cv_ptr;
try{
  cv_ptr= cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
}
catch (cv_bridge::Exception& e) {
      ROS_ERROR("cv_bridge exception: %s", e.what());
       return;
}
```

Se convierte el mensage ROS de la imagen en una imagen de OpenCV con codificación BGR

```
      imwrite("view.png", cv_ptr->image);
      imshow( "view", cv_ptr->image);
      cv::waitKey(30);
```

Con *imwrite*() se guarda la imagen con el nombre especificado.

Con *imshow*(), se muestra la imagen por pantalla.

En la función principal del programa: *main*(), como se ha comentado con el publicador, se inicializa ROS, y registra un nodo en el sistema, en este caso *subscriber_node*, que se encargará de manejar este código.

```
image_transport::ImageTransport it(nh);

image_transport::Subscriber sub =
it.subscribe("/camera/rgb/image_raw", 1, imageCallback);
```

Se crea una instancia de *ImageTransport*, inicializandola con el *NodeHandle*.

Después, se crea un suscriptor del topic */camera/rgb/image_raw*. ROS llamará a la función *imageCallback* cada vez que llegue una nueva imagen.

Antes de compilar el paquete, hay que editar el fichero *CMakeList.txt*, que se encuentra dentro de la carpeta del paquete. El contenido del fichero ya modificado es el siguiente:

```
1  cmake_minimum_required(VERSION 2.8.3)
2  project(mi_paquete)
3
4  add_compile_options(-std=c++11)
5
6  find_package(catkin REQUIRED COMPONENTS
7    cv_bridge
8    roscpp
9    std_msgs
10   image_transport
11 )
12
13 find_package(OpenCV REQUIRED)
14
15 catkin_package(
16
17 )
18
19 include_directories(include ${catkin_INCLUDE_DIRS})
20 include_directories(SYSTEM ${OpenCV_INCLUDE_DIRS})
21
22 add_executable(publisher src/publisher.cpp)
23 target_link_libraries(publisher ${catkin_LIBRARIES})
24
25 add_executable(subscriber src/subscriber.cpp)
26 target_link_libraries(subscriber ${catkin_LIBRARIES} ${OpenCV_LIBRARIES})
```

Figura 4.3: Contenido del fichero CMakeList.txt

De la línea 6 a la 11, se especifican los paquetes requeridos para compilar los códigos escritos antes con el comando *catkin*. En este caso, los paquetes que se necesitan son *roscpp*, *cv_bridge*, *std_msg* y *image_transport*. Este apartado se edita cuando se crea el paquete, pero en este caso, se ha tenido que editar dado que se creó el paquete solo con la dependencia *roscpp*.

Después, en la línea 13, se especifica que se requiere la librería OpenCV.

En las líneas 19 y 20, se incluyen los directorios donde se encuentran estas librerías, catkin y OpenCV.

En las líneas 22 y 23, 25 y 26, se define el ejecutable del *publisher* y del *subscriber* respectivamente. Esto le permitirá al compilador saber que debe crear un ejecutable a partir de los archivos definidos. Esto hará que el compilador vincule las bibliotecas requeridas por el paquete.

En este caso, también se tiene que modificar el archivo *package.xml.*

```xml
<?xml version="1.0"?>
<package format="2">
  <name>mi_paquete</name>
  <version>0.0.0</version>
  <description>The mi_paquete package</description>
  <maintainer email="sandra@todo.todo">sandra</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>
  <build_depend>cv_bridge</build_depend>
  <build_depend>image_transport</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>std_msgs</build_depend>

  <build_export_depend>cv_bridge</build_export_depend>
  <build_export_depend>image_transport</build_export_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>sensor_msgs</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>

  <exec_depend>cv_bridge</exec_depend>
  <exec_depend>image_transport</exec_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>sensor_msgs</exec_depend>
  <exec_depend>std_msgs</exec_depend>

  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

Figura 4.4: Contenido del fichero package.xml

En este archivo, hay que indicar las dependencias del compilador y las de ejecución, de la herramienta catkin. Estas son: *roscpp, cv_bridge, image_transport sensor_msgs* y *std_msg*

Una vez modificados estos ficheros, compilamos el proyecto. Para ello, desde el directorio principal del espacio de trabajo, se ejecuta el siguiente comando:

```
>> catkin_make
```

Por último, se ejecuta el nodo *publisher* con el siguiente comando.

```
>> rosrun mi_paquete publisher
```

Por último, se ejecuta el nodo *subscriber* con el siguiente comando.

```
>> rosrun mi_paquete subscriber
```

4.2 Creación de mundo 3D

Cómo ya se ha comentado, los simuladores son muy útiles para realizar distintas pruebas antes de observar su comportamiento en la realidad. Gazebo permite simular el turtlebot en entornos realistas, por lo que, en este apartado se va a describir los pasos para la creación de nuestro entorno. El cual será lo más realista posible al entorno donde el robot turtlebot trabajará.

4.2.1 Crear mundo en Gazebo

Gazebo es un simulador y un editor 3D, por lo que también nos permite editar nuestro mundo [35].

Gazebo permite editar un mundo de dos maneras diferentes. Insertar objetos que ya se encuentran en la *database* del propio programa.

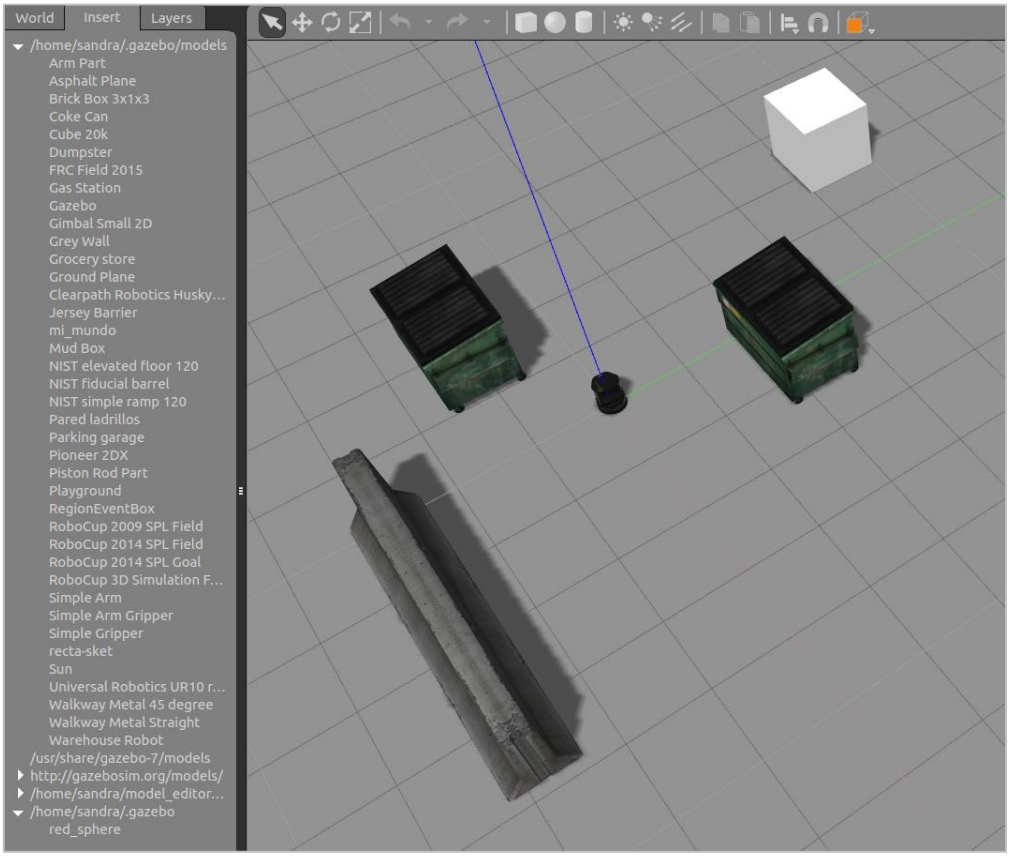

Figura 4.5: Objetos insertados en un mundo de Gazebo

El editor de gazebo consta de 3 áreas.

- La paleta, donde se puede elegir las características y materiales de la construcción.
- Vista 2D, donde se puede importar un plano o intentar muros, ventanas puertas y escaleras.
- Vista 3D, donde se puede pre visualizar la construcción.

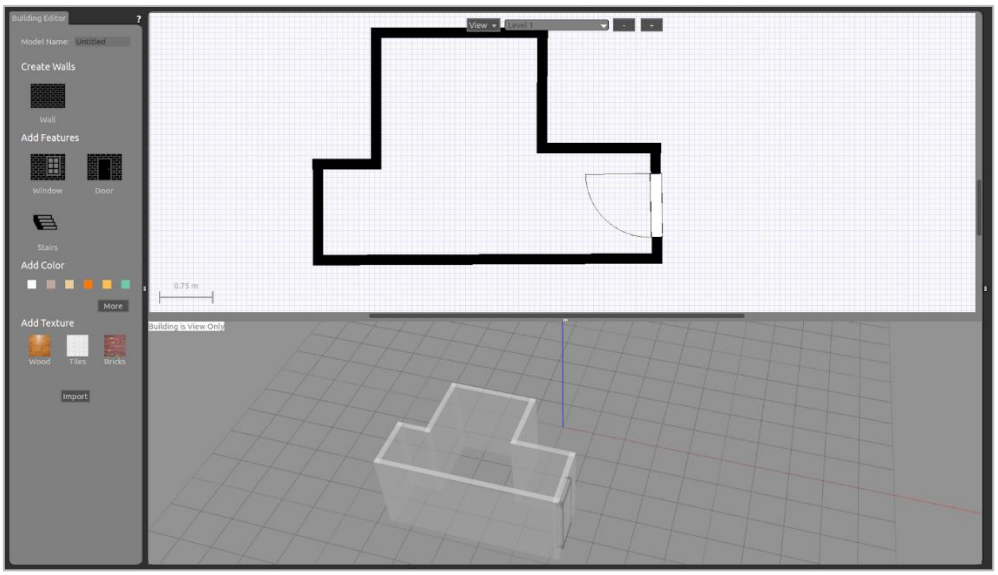

Figura 4.6: Editor del simulador Gazebo

Como se puede observar en la figura 4.6, el editor de Gazebo no permite crear mundos muy complejos. Por lo que, si se quiere simular el robot en un mundo que se asemeje lo máximo posible al entorno real en el que finalmente va a trabajar, se debería utilizar otro editor 3D más potente.

4.2.2 Creación mundo en editor 3D

En este apartado se va a insertar un mundo 3D en Gazebo, el cual ya ha sido creado en otro programa editor 3D, de la misma forma en la que se han insertado los objetos de la *database* del simulador Gazebo.

Partimos de que ya hemos creado nuestro mundo en otro simulador 3D. En este caso, como se observa en la figura 4.7, se ha creado un circuito con una serie de curvas en el programa *SketchUp*.

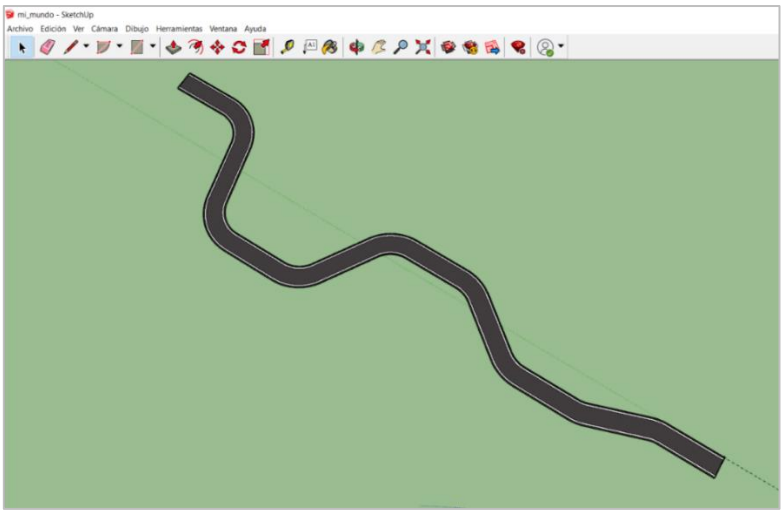

Figura 4.7: Diseño de una carretera en el editor SketchUp

Al crear este mundo, se han utilizado una serie de texturas, las cuales se guardan en una imagen, por lo que nuestro mundo se compone de dos ficheros:

- *mundo.dae*
- *asphalt.jpg*

Los objetos de la *database* de gazebo se encuentran en la carpeta oculta concretamente en la siguiente ruta, siendo *sandra* mi carpeta personal: */home/sandra/.gazebo/models*

En este directorio es donde se tiene que crear la carpeta en la cual se va a definir nuestro mundo creado: *mi_mundo*

Para definir cada uno de los objetos 3D de la *database* de Gazebo, se utilizan dos ficheros:

- *model.config*
- *model.sdf*

En el fichero *model.config*, se define el nombre que va a tener el objeto (*mi_mundo*) también el fichero donde está definido el mundo (*model.sdf*)

```
<?xml version="1.0" ?>
<model>
    <name>mi_mundo</name>
    <version>1.0</version>
    <sdf version="1.6">model.sdf</sdf>
    <author>
        <name></name>
        <email></email>
    </author>
    <description></description>
</model>
```

Figura 4.8: Contenido del fichero model.config

Normalmente, los modelos 3D están directamente definidos en el fichero *model.sdf*, pero nuestro mundo se encuentra definido en el archivo *mi_mundo.dae*, por lo que en el fichero *model.sdf* sólo tenemos que definir la ubicación del archivo *.dae*.

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <model name="mi_mundo">
    <static>true</static>
    <link name="link">
      <collision name="collision">
        <geometry>
         <mesh>
            <uri>model://mi_mundo/mi_mundo.dae</uri>
      </mesh>
        </geometry>
      </collision>
      <visual name="visual">
        <cast_shadows>false</cast_shadows>
        <geometry>
          <mesh>
            <uri>model://mi_mundo/mi_mundo.dae</uri>
            </mesh>
        </geometry>
      </visual>
    </link>
  </model>
</sdf>
```

Figura 4.9: Contenido del fichero model.sdf

Dentro del fichero *.dae*, se encuentra la definición del mundo y también se especifica la ruta donde se encuentra la imagen *asphalt.jpg* que define la textura del circuito, por lo que hay que comprobar que dicha imagen se encuentra en la ruta especificada. O modificar la ruta si queremos guardar la imagen en otro directorio.

```
              </technique>
          </profile_COMMON>
      </effect>
  </library_effects>
  <library_images>
      <image id="ID40">
          <init_from>mi_mundo/Asphalt_New.jpg</init_from>
      </image>
  </library_images>
  <scene>
      <instance_visual_scene url="#ID2" />
  </scene>
```

Figura 4.10: Contenido del fichero mi_mundo.dae

Finalmente se comprueba que el mundo se encuentra en la lista de la *database* del simulador, y también se comprueba que se pueda insertar. En la figura 4.11, tenemos el mundo insertado en el simulador Gazebo.

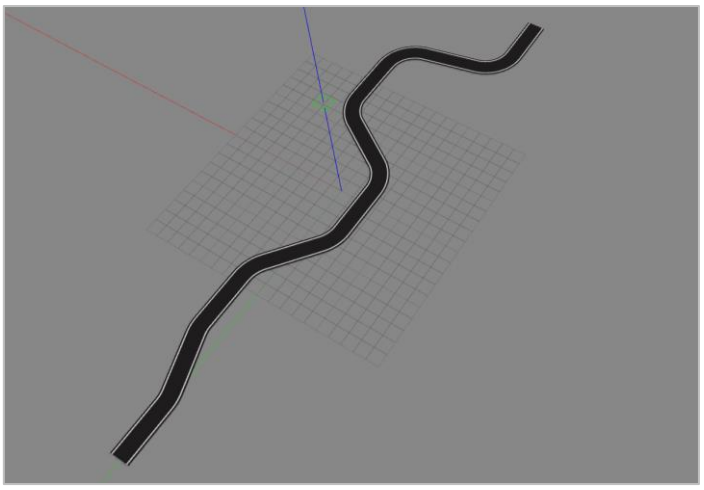

Figura 4.11: Mundo creado insertado en el simulador Gazebo

Si se va a trabajar continuamente con el mundo, para no tener que insertar el circuito cada vez que se inicie el simulador gazebo y también para que siempre se encuentre en la misma posición, se puede guardar como un mundo en un archivo *.world*. Y así especificar el entorno en el que se quiere trabajar en el comando que inicia gazebo junto con el robot Turtlebot, que ha sido comentado en el apartado 3.2.

También se podría definir un nuevo paquete que inicie el simulador en el mundo que se especifique en el fichero *.launch*, donde el turtlebot se inserte en la posición inicial deseada.

A continuación, se va a mostrar un ejemplo en el que se va a utilizar el mapa creado en este apartado. Como se ha comentado, este mundo corresponde a la simulación de un circuito, donde se observan dos líneas continúas simulando una carretera. En la figura 4.12, tenemos una captura de la cámara del Turtlebot que se encuentra justo en medio de esta carretera.

Figura 4.12: Mundo creado insertado en el simulador Gazebo

Este mundo va a ser utilizado para crear un paquete con el objetivo de la detección de las líneas de la carretera a partir de visión artificial con las librerías de C++ de OpenCV.

El objetivo de este paquete es guiar al robot Turtlebot a través de la carretera del mundo simulado, distinguiendo las líneas del carril a partir de visión artificial, aplicada a las imágenes capturadas por la cámara del turtlebot.

Una vez se han detectado las líneas del carril, se analizan sus características. En este caso interesan el número de líneas y el ángulo de éstas, para posteriormente guiar al robot dependiendo del resultado obtenido.

Dependiendo de la inclinación de las líneas, el robot avanzará, girará a la derecha, girará a la izquierda o parará.

A continuación, se va a explicar los pasos seguidos en el paquete para el procesamiento de las imágenes capturadas por el robot TurtleBot.

El primer paso a realizar para la segmentación de las líneas convertir las imágenes capturadas a formato HSV, y encontrar los valores adecuados de los umbrales que definen el color que se quiere detectar.

Figura 4.13: Imagen segmentada de la carretera

En la figura 4.13, tenemos el resultado de umbralizar la imagen capturada por el Turtlebot. A continuación, se detectan las líneas.

En primer lugar, surgió la idea de encontrar los contornos de las líneas y a partir de los contornos detectados, se podían aplicar distintas técnicas para el análisis.

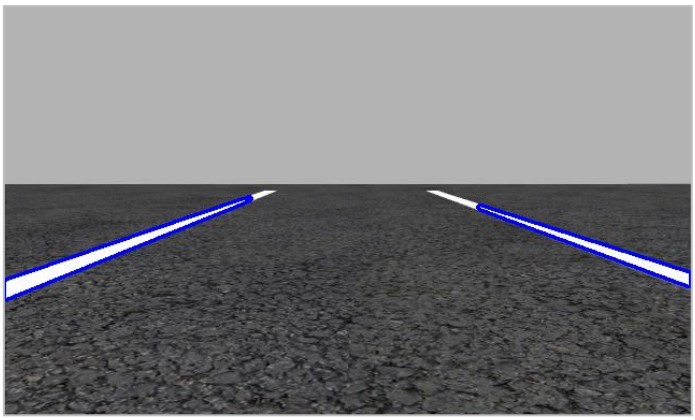

Figura 4.14: Detección de contornos

A partir de los contornos detectados, hay diversas opciones para obtener información sobre las líneas.

- El cálculo del área de los contornos a partir de sus momentos.
- La función *cv2.boundingRect*() ajusta los contornos a un rectángulo recto.
- La función *cv2.minAreaRect*() ajusta los contornos a un rectángulo girado.
- La función *cv2.fitLine*() ajusta los contornos a una línea.

En segundo lugar, surgió la idea de detectar las líneas a partir de dos métodos distintos, la transformada estándar de Hough y la transformada probabilística de Hough. Con estos métodos se obtienen el ángulo de las líneas detectadas y con estos ángulos y se realiza una media. Dependiendo del resultado este el ángulo medio se guía al robot.

Estos dos métodos permiten definir un umbral para ajustar la detección de líneas. En la figura 4.15, se observa las líneas detectadas con Transformada estándar de Hough, con un umbral donde se detectan bastantes líneas.

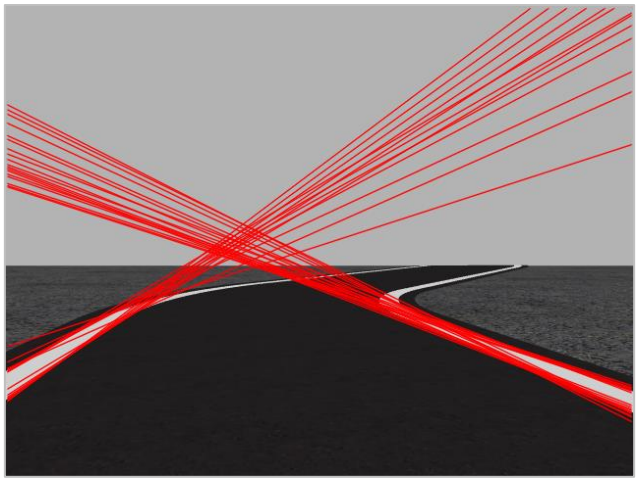

Figura 4.15: Detección de las líneas con la Transformada de Hough

A parte de la detección de las líneas, también se analizan los resultados que se obtienen al realizar otros procesamientos en las imágenes. El primer procesamiento de la imagen fue el cambio de perspectiva, ya que la perspectiva de la imagen que captura el Turtlebot no devuelve mucha información con respecto a las líneas.

Figura 4.16: Cambio de perspectiva

También se recorta la imagen original capturada por la imagen, para solo quedarnos con la zona de interés, y así eliminar las otras zonas que pueden llevar a error el algoritmo de detección de líneas.

Figura 4.17: Recorte de la imagen

Después de diversas pruebas con las distintas opciones y métodos comentados, se llega a la conclusión de que los mejores resultados obtenidos para guiar al robot TurtleBot gracias a la visión artificial al detectar las líneas de la carretera, es la Transformada de Hough estándar sobre la imagen umbralizada y recortada sobre la zona de interés. También se observa que el comportamiento es mejor cuando el umbral del método de la transformada es más alto y se detectan más líneas, que cuando sólo se detectan pocas líneas.

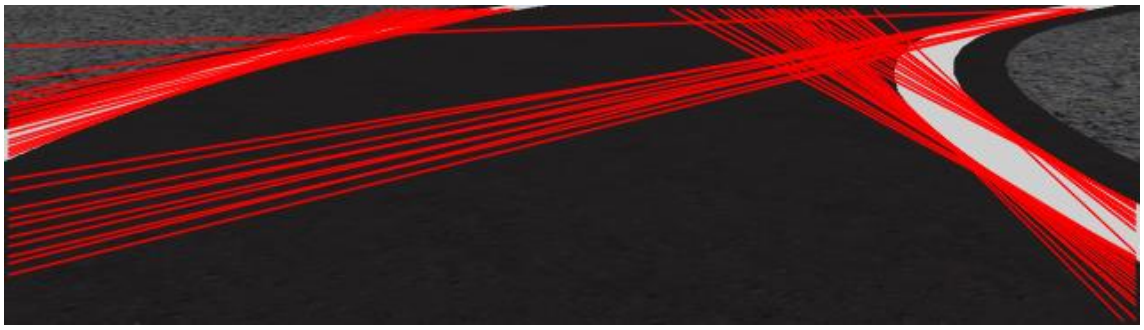

Figura 4.18: Resultado final de la detección de las líneas de la carretera

Se han obtenido muy buenos resultados con estos métodos. El robot Turtlebot es capaz de desplazarse por todo el circuito gracias a la detección de las líneas blancas de la carretera sin salirse en ningún momento. Como se podía ver en las figuras, el circuito ha sido creado con diversas curvas, algunas de ellas bastante pronunciadas, tanto hacia la izquierda como a la derecha, y el robot es capaz de realizar el recorrido completo de este circuito.

4.3 Paquete varios Turtlebots en simulación

En este apartado se va a crear un paquete que va a insertar dos robots Turtlebots en el simulador Gazebo, en un mismo mundo, el cual se ha definido en el apartado anterior [36]. Este paquete podrá insertar el número de turtlebots deseados, aunque la explicación de la creación del paquete se realizará solo con dos robots.

En algunas aplicaciones donde hubiese varios robots colaborando, la única forma de probar y observar el comportamiento entre ambos sería en la realidad. Con este paquete se podrían realizar las pruebas previas directamente en el simulador, sin la necesidad de los robots reales.

El primer paso a realizar es la creación de un nuevo paquete dentro de nuestro espacio de trabajo.

```
>> catkin_create_pkg nombre_paquete roscpp geometry_msgs sensor_msgs
nav_msgs
```

Este comando creará una carpeta con el nombre de nuestro paquete, que contiene un *package.xml* y un *CMakeLists.txt*, que se completan parcialmente con la información

que se le proporciona a *catkin_create_pkg*. El comando requiere que se le dé un nombre y opcionalmente una lista de las dependencias de ese paquete.

A continuación, se copia el contenido del paquete *turtlebot_gazebo* ya instalado en la carpeta del paquete creado. El paquete *turtlebot_gazebo*, simula el robot Turtlebot en Gazebo en el mundo que se le especifique. Por lo que este paquete va a permitir simular dos Turtlebots en el simulador y además especificar el mundo en el que se quiere simular los robots.

Dentro de la carpeta *launch* del paquete se crean 3 ficheros *.launch*:

- *main.launch*
- *robots.launch*
- *one_robot.launch*

Con estos tres ficheros se van a definir cuantos Turtlebots se van a simular, en qué mundo y las propiedades que definen a los robots. Al ejecutar el paquete, se llamará primeramente al fichero *main.launch*. Después, este fichero llamará a *robots.launch*, que llamará a *one_robot.launch*

main.launch

El archivo principal es *main.launch*, el cual llama a los otros dos archivos. El contenido de este fichero se muestra en la figura 4.19.

```
1  <launch>
2    <param name="/use_sim_time" value="true" />
3    <!-- start world -->
4    <node name="gazebo" pkg="gazebo_ros" type="gazebo"
5     args="$(find prueba-multi)/worlds/mi_mundo.sdf" respawn="false" output="screen" />
6
7    <!-- start gui -->
8    <node name="gazebo_gui" pkg="gazebo_ros" type="gui" respawn="false" output="screen"/>
9
10
11   <include file="$(find prueba-multi)/launch/robots.launch"/>
12 </launch>
```

Figura 4.19: Contenido del fichero main.launch

Este fichero inicia el simulador Gazebo con el mundo que le especificamos. El mundo se pasa por argumento en la línea 5. En este caso, el mundo que va a simular es *mi_mundo.sdf* que se encuentra en la carpeta *worlds* de nuestro paquete. En la penúltima línea llama al fichero *robots.launch*

robots.launch

En este archivo se definen los robots que se van a simular. En nuestro caso son dos robots, aunque pueden ser todos lo que se necesiten. El contenido de este fichero se muestra en la figura 4.20.

```
<launch>
  <param name="robot_description"
    command="$(find xacro)/xacro.py $(find turtlebot_description)/urdf/turtlebot_gazebo.urdf.xacro" />
<arg name="stacks"    value="$(optenv TURTLEBOT_STACKS hexagons)"/>  <!-- circles, hexagons -->
<arg name="3d_sensor" value="$(optenv TURTLEBOT_3D_SENSOR kinect)"/>  <!-- kinect, asus_xtion_pro -->
<arg name="base"      value="$(optenv TURTLEBOT_BASE kobuki)"/> <!-- create, roomba -->
  <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find turtlebot_description)
    /robots/$(arg base)_$(arg stacks)_$(arg 3d_sensor).urdf.xacro'" />
  <param name="robot_description" command="$(arg urdf_file)" />

  <!-- BEGIN ROBOT 1-->
  <group ns="robot1">
    <param name="tf_prefix" value="robot1_tf" />
    <include file="$(find prueba-multi)/launch/one_robot.launch" >
     <arg name="init_pose" value="-x 0 -y -1.0 -z 0 -Y -1.5" />
      <arg name="robot_name"  value="Robot1" />
    </include>
  </group>

  <!-- BEGIN ROBOT 2-->
  <group ns="robot2">
    <param name="tf_prefix" value="robot2_tf" />
    <include file="$(find prueba-multi)/launch/one_robot.launch" >
      <arg name="init_pose" value="-x 0 -y 0.0 -z 0 -Y -1.5" />
      <arg name="robot_name"  value="Robot2" />
    </include>
  </group>
</launch>
```

Figura 4.20: Contenido del fichero robots.launch

En la tercera línea del fichero, se le indica el fichero donde se encuentra la descripción del robot Turtlebot para la simulación del robot en Gazebo.

También se define el nombre de cada robot y el prefijo que se le va a añadir a cada topic para diferenciarlos, junto con la posición inicial de cada Turtlebot en el mundo simulado.

Cada robot, llama al archivo *one_robot.launch* y le pasa los datos que se han definido por argumento.

one_robot.launch

Este fichero define el resto de propiedades de los robots, como los sensores que va a tener y las especificaciones y limitaciones de cada sensor.

```
<launch>
    <arg name="robot_name"/>
    <arg name="init_pose"/>

    <!-- The odometry estimator, throttling, fake laser etc. go here -->
    <!-- All the stuff as from usual robot launch file -->

  <arg name="base"      value="$(optenv TURTLEBOT_BASE kobuki)"/> <!-- create, roomba -->
  <arg name="battery"   value="$(optenv TURTLEBOT_BATTERY /proc/acpi/battery/BAT0)"/>  <!-- /proc/acpi/battery/BAT0 -->
  <arg name="gui" default="true"/>
  <arg name="stacks"    value="$(optenv TURTLEBOT_STACKS hexagons)"/>  <!-- circles, hexagons -->
  <arg name="3d_sensor" value="$(optenv TURTLEBOT_3D_SENSOR kinect)"/>  <!-- kinect, asus_xtion_pro -->

  <!-- Gazebo model spawner -->
  <node name="spawn_turtlebot_model" pkg="gazebo_ros" type="spawn_model"
        args="$(arg init_pose) -unpause -urdf -param /robot_description -model $(arg robot_name)"
    respawn="false">
  </node>

 <!-- velocity mux nodelet -->
  <node pkg="nodelet" type="nodelet" name="nodelet_manager" args="manager"/>
  <node pkg="nodelet" type="nodelet" name="cmd_vel_mux"
        args="load yocs_cmd_vel_mux/CmdVelMuxNodelet nodelet_manager">
    <param name="yaml_cfg_file" value="$(find turtlebot_bringup)/param/mux.yaml" />
    <remap from="cmd_vel_mux/output" to="commands/velocity"/>
  </node>

  <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher">
      <param name="publish_frequency" type="double" value="30.0" />
    <param name="tf_prefix" type="string" value="$(arg robot_name)" />
  </node>

  <!-- Fake laser -->
  <node pkg="nodelet" type="nodelet" name="laserscan_nodelet_manager" args="manager"/>
  <node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
        args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet laserscan_nodelet_manager">
    <param name="scan_height" value="10"/>
    <param name="output_frame_id" value="$(arg robot_name)/camera_depth_frame"/>
    <param name="range_min" value="0.45"/>
    <remap from="image" to="/camera/depth/image_raw"/>
    <remap from="scan" to="scan"/>
  </node>

 <!-- Velocity smoother -->
  <node pkg="nodelet" type="nodelet" name="navigation_velocity_smoother"
    args="load yocs_velocity_smoother/VelocitySmootherNodelet nodelet_manager">
    <rosparam file="$(find turtlebot_bringup)/param/defaults/smoother.yaml" command="load"/>
    <remap from="navigation_velocity_smoother/smooth_cmd_vel" to="cmd_vel_mux/input/navi"/>
    <remap from="navigation_velocity_smoother/odometry" to="odom"/>
    <remap from="navigation_velocity_smoother/robot_cmd_vel" to="commands/velocity"/>
  </node>

  <!-- Safety controller-->
  <node pkg="nodelet" type="nodelet" name="kobuki_safety_controller"
    args="load kobuki_safety_controller/SafetyControllerNodelet nodelet_manager">
    <remap from="kobuki_safety_controller/cmd_vel" to="cmd_vel_mux/input/safety_controller"/>
    <remap from="kobuki_safety_controller/events/bumper" to="events/bumper"/>
    <remap from="kobuki_safety_controller/events/cliff" to="events/cliff"/>
    <remap from="kobuki_safety_controller/events/wheel_drop" to="events/wheel_drop"/>
  </node>

</launch>
```

Figura 4.21: Contenido del fichero one_robot.launch

Este fichero finalmente define los topic y las propiedades de cada robot.

Finalmente, para ejecutarlo el paquete se ejecuta el siguiente comando:

```
>> roslaunch two-turtlebots main.launch
```

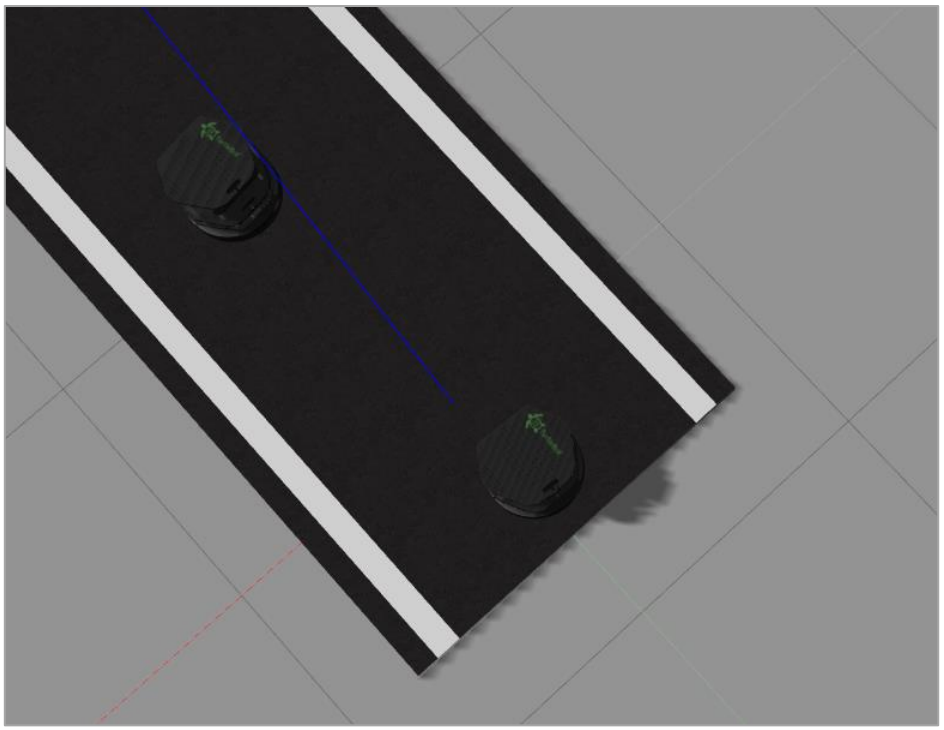

Figura 4.22: Resultado del paquete two-turtlebots en el simulador Gazebo

En la figura 4.22 podemos observar los dos robots Turtlebot en la posición inicial indicada y en el mundo que se ha especificado.

Para comprobar que se han lanzados los nodos ROS necesarios para controlar todos los sensores y actuadores de los dos Turtlebots se ejecuta la herramienta *rqt_graph*:

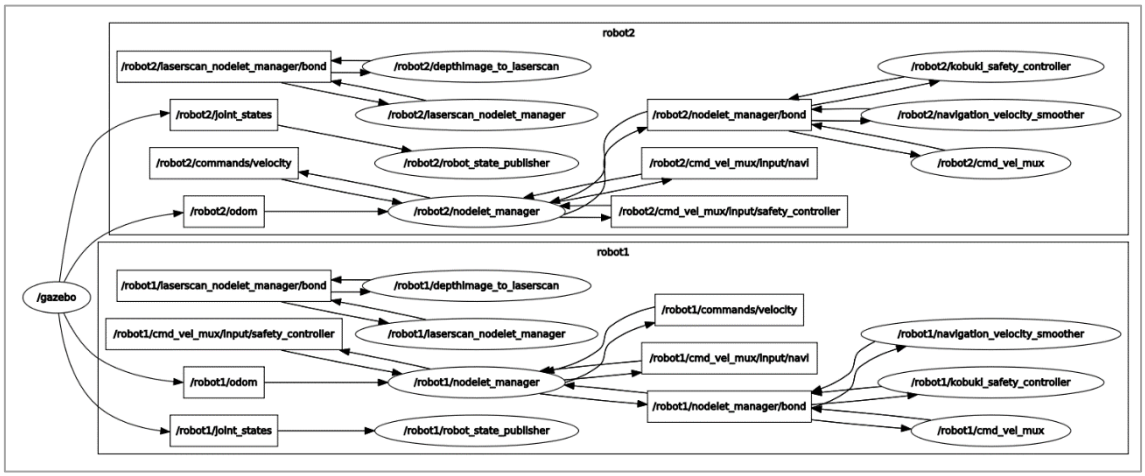

Figura 4.23: Conexiones de los nodos de los dos Turtlebots en Gazebo

Con la herramienta *rostopic list* se comprueban que se han lanzado todos los sensores y actuadores necesarios de los turtlebot. En la figura 4.24, se tienen algunos de los topics creados tras ejecutar este paquete.

```
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/robot1/camera/depth/camera_info
/robot1/camera/depth/image_raw
/robot1/camera/depth/points
/robot1/camera/parameter_descriptions
/robot1/camera/parameter_updates
/robot1/camera/rgb/camera_info
/robot1/camera/rgb/image_raw
/robot2/mobile_base/sensors/imu_data
/robot2/navigation_velocity_smoother/parameter_descriptions
/robot2/navigation_velocity_smoother/parameter_updates
/robot2/navigation_velocity_smoother/raw_cmd_vel
/robot2/nodelet_manager/bond
/robot2/odom
/robot2/scan
/rosout
/rosout_agg
/statistics
/tf
/tf_static
```

Figura 4.24: Rostopic list de los dos Turtlebots simulados

Como se puede observar, se han creado los topics para cada robot con el prefijo que define a cada robot.

Como se ha comentado al principio del apartado, este paquete está hecho para que se simulen tantos turtlebots como se desee. Solo habría que modificar el archivo *robots.launch* y copiar el trozo de código que define cada robot. En la figura 4.25, se tienen las líneas de código que definen al Turtlebot y que habría que copiar para definir más robots.

```
<!-- BEGIN ROBOT 1-->
<group ns="robot1">
  <param name="tf_prefix" value="robot1_tf" />
  <include file="$(find prueba-multi)/launch/one_robot.launch" >
   <arg name="init_pose" value="-x 0 -y -1.0 -z 0 -Y -1.5" />
    <arg name="robot_name"  value="Robot1" />
  </include>
</group>
```

Figura 4.25: Código donde se define el Turtlebot

Sólo habría que modificar algunos parámetros. En concreto habría que modificar el número del robot cuando se define el nombre del grupo y también modificar la posición inicial.

En la figura 4.26 se observan los 5 robots Turtlebots insertados en el simulador Gazebo.

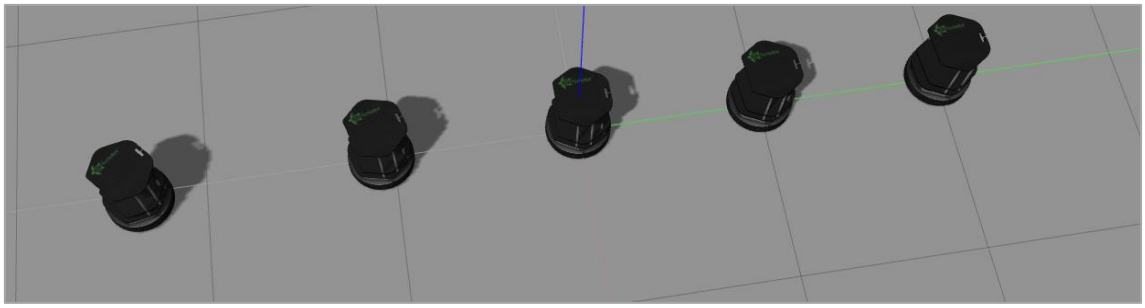

Figura 4.26: Simulación de 5 Turtlebots en Gazebo

4.4 Construcción de un mapa y navegación autónoma

En este apartado se van a explicar los pasos a seguir para construir un mapa del entorno del robot. Este mapa se va construir a partir del mundo 3D simulado en Gazebo, y también del entorno real en el que se encuentra el Turtlebot real.

Una vez se tiene el mapa, se verá cómo localizar al robot dentro de este y después hacer que el robot se mueva de manera autónoma solo indicando el punto objetivo a alcanzar. La localización también se realizará en las dos situaciones, simulación y realidad [37].

4.4.1 Construcción de un mapa

En ROS existe un paquete que permite construir mapas del entorno en el que se encuentra el robot a partir de las lecturas del láser.

En este apartado se muestra cómo construir un mapa para que el robot recuerde el entorno. Ya que, a partir del mapa generado, el robot podrá navegar sobre él de forma autónoma.

4.4.1.1 Simulación

Para poder construir un mapa a partir de los datos obtenidos por el turtlebot, en primer lugar, debe estar iniciado el simulador Gazebo con el Turtlebot en el mundo que se quiere mapear.

En estas pruebas iniciales se va a mapear un mundo simple creado en el apartado 4.2.1 en el editor de Gazebo. En la figura 4.27 se tiene el mundo a mapear.

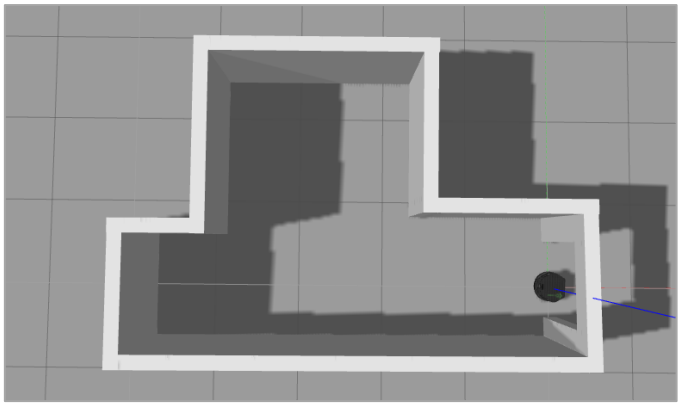

Figura 4.27: Mundo en Gazebo

Los mapas de Turtlebot se crean con la herramienta de ROS, *gmapping*. Este algoritmo se puede ajustar modificando los parámetros en el archivo .launch del */includes /gmapping.launch* del paquete turtlebot_navigation.

```
>> roslaunch turtlebot_navigation gmapping_demo.launch
```

Por último, se ejecuta Rviz, que será el programa donde se irá construyendo el mapa a partir de los datos de los sensores.

```
>> roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Para construir el mapa, Rviz tiene que estar subscrito al topic */map*

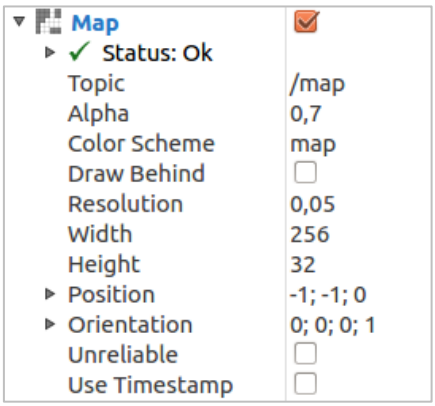

Figura 4.28: Especificaciones del topic /map en Rviz

Como se muestra en la imagen 4.29, inicialmente tendremos el robot en la posición inicial, donde se observará lo que está capturando ya del mapa si está en dirección a algún objeto.

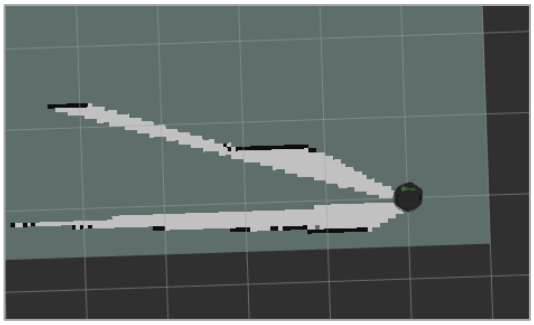

Figura 4.29: Inicio de la contrucción del mapa en Rviz

Las partes donde no está rellenado nada significa que el láser ha obtenido un valor fuera de su rango, por lo que el valor obtenido es un valor Nan (Not a number).

Este valor puede ser porque está fuera del rango máximo o del mínimo, es decir, que puede ser que no haya ningún objeto cerca o que el obstáculo esté lejos. El paquete que crea el mapa no puede distinguirlo por lo que deja esa zona como inexplorada.

Para construir el mapa, se teleoperará al robot a partir del paquete *keyboard_teleop*, por todo el mapa, en dirección a todos los objetos, hasta que se observe en Rviz, que el mapa se ha construido en su totalidad.

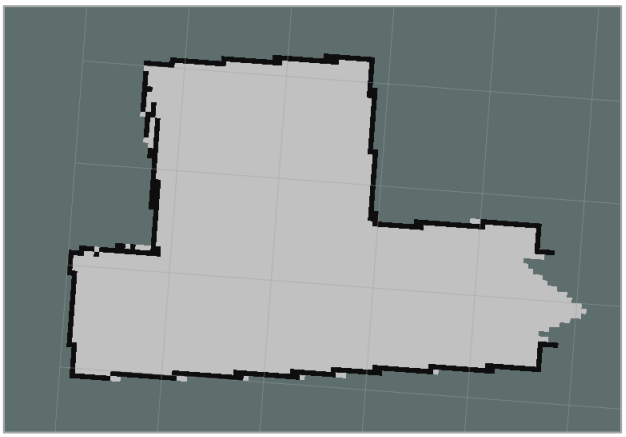

Figura 4.30: Mapa construido en Rviz

En la figura 4.30, se puede observar cómo Rviz ha creado el mapa de acuerdo a los datos capturados por el Turtlebot simulado. Se observa que los obstáculos los dibuja como líneas negras y el espacio disponible en color gris claro.

Rviz es el programa donde se puede visualizar como se está construyendo el mapa, pero realmente el paquete que está creando este mapa es el paquete *gmapping_demo.launch*, por lo que no se debe cerrar este programa hasta que no se haya guardado el mapa correctamente.

Una vez se observa en Rviz que se ha construido el mapa completo, se guarda con el siguiente comando:

```
>> rosrun map_server map_saver -f <map_name>
```

El mapa guardado consiste en dos archivos: *my_map.pgm* y *my_map.yaml*. El archivo *.yaml* (figura 4.31) describe los metadatos del mapa, incluida la ubicación del archivo de imagen del mapa, su resolución, su origen, el umbral de celdas ocupadas y el umbral de celdas libres. El archivo de imagen *.pgm* es una imagen en escala de grises del mapa que se puede abrir con cualquier programa de edición de imágenes y codifica los datos de ocupación del mapa [38].

```
image: mapa.pgm
resolution: 0.050000
origin: [-1.000000, -12.200000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Figura 4.31: Contenido de mapa.yaml

Con el comando *rostopic* también se puede obtener información del mapa creado:

```
>> rostopic echo map_metadata
```

```
map_load_time:
  secs: 0
  nsecs:         0
resolution: 0.0500000007451
width: 320
height: 480
origin:
  position:
    x: -0.1
    y: -12.9
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
```

Figura 4.32: Metadatos del mapa creado

Este comando muestra que el mapa contiene 320 x 480 celdas, con una resolución de 5cm por celda. También muestra la posición y la orientación del mapa con respecto a las coordenadas del mundo.

Como se puede observar en la figura 4.30 el mapa no es muy bueno. Una de las razones es que los sensores en Turtlebot no son excelentes para crear mapas es que el paquete *gmapping* espera mensajes de *LaserScan* y, como se mencionó anteriormente, Turtlebot no tiene láser; en su lugar, utiliza los datos de un sensor *Kinect* de Microsoft para sintetizar mensajes *LaserScan*, que pueden usarse mediante *gmapping*. El problema es que este láser falso tiene un campo de visión más corto y más estrecho que un sensor láser típico. El paquete *gmapping* usa los datos del láser para estimar cómo se mueve el robot, y esta estimación es mejor con datos de largo alcance en un amplio campo de visión.

Se puede mejorar la calidad del mapeo estableciendo algunos de los parámetros de *gmapping* en diferentes valores [39]. La herramienta *rosparam* sirve para obtener y configurar los parámetros ROS:

```
>> rosparam set /slam_gmapping/angularUpdate 0.1
>> rosparam set /slam_gmapping/linearUpdate 0.1
>> rosparam set /slam_gmapping/xmin -10
>> rosparam set /slam_gmapping/xmax 10
>> rosparam set /slam_gmapping/ymin -10
>> rosparam set /slam_gmapping/ymax 10
>> rosparam set /slam_gmapping/delta 0.005
```

AngularUpdate y *LinearUpdate*, definen la distancia o ángulo que el robot avance o gire hasta que el paquete vuelve a procesar los datos del láser. *Xmin*, *xmax*, *ymin* y *ymax*, son el tamaño inicial del mapa en metros. *Delta* es la resolución del mapa.

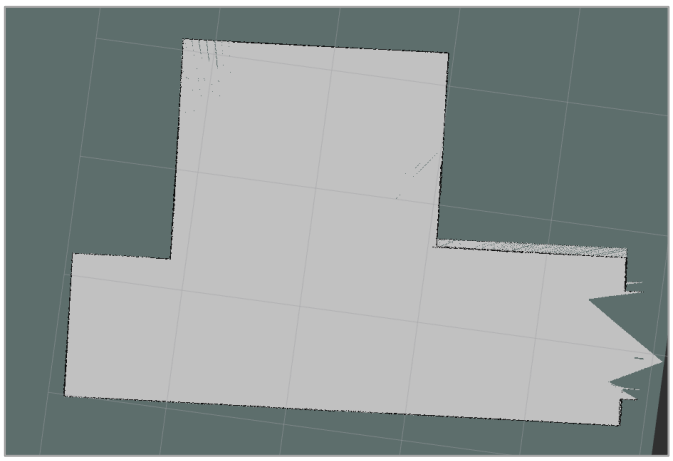

Figura 4.33: Mapa resultante de los parámetros modificados

En la figura 4.34, tenemos los ficheros .pgm de los mapas creados con distinta resolución.

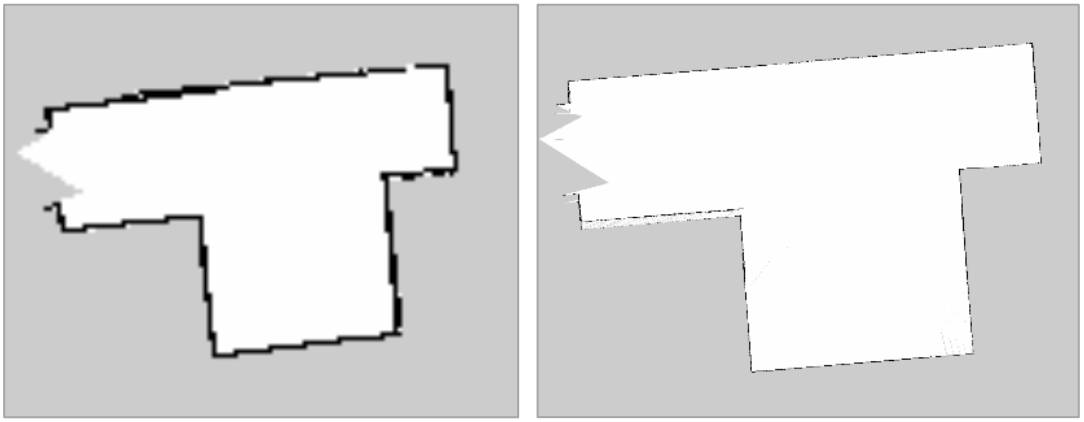

Figura 4.34: Contenido de los archivos .pgm de los mapas construidos

4.4.1.2 Turtlebot real

Como se ha comentado en el apartado 3.2.2, el PC del Turtlebot no tiene la capacidad de cómputo necesaria para ejecutar el visualizador Rviz. Aunque existen dos opciones para poder utilizar este programa con los datos reales de los sensores y actuadores, a partir de datos ya capturados en un archivo *.bag*, o en tiempo real, realizando una conexión de los dos PC, donde ambos dos se encuentren conectados por el mismo ROS Master. En este apartado se mostrará cómo construir el mapa con estas dos opciones.

En este apartado también se pretende comprobar si los resultados obtenidos al construir los mapas con las dos opciones son los mismos o existe alguna diferencia.

Para poder comprobarlo, se van a capturar los datos de todos los topics en un archivo *.bag* mientras se construye en tiempo real el mapa. Posteriormente se construirá otro mapa a partir de los datos capturados en el *rosbag*, por lo que ambos mapas se crearán a partir de los mismos datos. A partir de estos dos mapas construidos, se podrán comparar sus resultados.

4.4.1.2.1 Tiempo real

Para poder construir el mapa en tiempo real a partir de los datos reales que se obtienen de los sensores y actuadores del Turtlebot, hay que conectar los dos PC, nuestro ordenador personal y el PC del Turtlebot, al mismo ROS Master [40].

Para poder conectar los dos ordenadores al mismo ROS Master, hay que ejecutar los siguientes comandos, en los computadores indicados.

En nuestro PC personal, ejecutar los siguientes comandos:

```
>>  echo  export  ROS_MASTER_URI=http://<IP_OF_TURTLEBOT>:11311  >>
~/.bashrc

>> echo export ROS_HOSTNAME=<IP_OF_WORKSTATION> >> ~/.bashrc

>> source ~/.bashrc
```

En el PC del TurtleBot ejecutar los siguientes comandos:

```
>> echo export ROS_MASTER_URI=http://localhost:11311>> ~/.bashrc

>> echo export ROS_HOSTNAME=<IP_OF_TURTLEBOT> >> ~/.bashrc

>> source ~/.bashrc
```

A partir de estos comandos se modifican las variables globales *ROS_MASTER_URI* y *ROS_HOSTNAME* de cada ordenador. Inicialmente, en ambos ordenadores, estas variables tenían el valor por defecto *localhost*, por lo que siempre que se crease algún nodo ROS en cada ordenador, siempre iban a estar conectados al ROS Master creado en su propio PC.

Con esta modificación, los nodos ROS que se creen en ambos ordenadores, estarán conectados al mismo ROS master, por lo que ambos ordenadores tendrán a los mismos topics y, por lo tanto, también podrán leer todos los datos. Incluso, permite ejecutar comandos desde nuestro PC que se aplicará sobre el PC del TurtleBot, y viceversa.

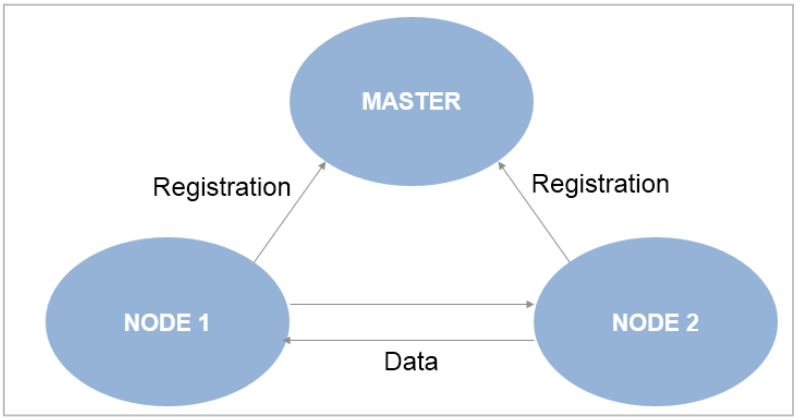

Figura 4.35: Conexiones de ROS de los dos PC

Se puede comprobar si se ha realizado correctamente esta modificación al imprimir por pantalla los valores actuales de las dos variables modificadas en ambos ordenadores.

Con el siguiente comando, se puede ver el valor actual de la variable *ROS_MASTER_URI* y comprobar si se ha modificado correctamente este valor.

```
>> echo $ROS_MASTER_URI
```

Se comprueba que la IP devuelta sea la del robot TurtleBot. Esto significa que el ROS Master al cual va a estar conectado nuestro PC estará corriendo en el PC del TurtleBot.

Una vez se han terminado estas pruebas, hay que volver a poner los valores por defecto que tenían estas variables, tanto en el PC personal como en el PC del Turtlebot.

Para construir el mapa, hay que lanzar los nodos de ROS pertenecientes a los sensores y actuadores necesarios para construir el mapa. Se necesita la base móvil y el láser Hokuyo. En este caso no es necesaria la cámara Astra.

Para construir el mapa, se utilizan los mismos comandos utilizados en la simulación, solo hay que conocer, en qué ordenador hay que lanzar cada comando.

En el PC del turtlebot se ejecuta el paquete *gmapping*, con el siguiente comando

```
>> roslaunch turtlebot_navigation gmapping_demo.launch
```

Como se ha comentado antes, en el PC personal se ejecuta el visualizador Rviz por la gran capacidad de cómputo necesaria de este programa.

```
>> roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Después se teleopera al robot con el paquete *keyboard_teleop*, por todo el entorno del cual se quiere construir el mapa hasta que se observe en Rviz que el mapa está enteramente construido de manera que represente la realidad del entorno, y comprobar que no falte ningún obstáculo estático que vaya a estar permanentemente en el entorno.

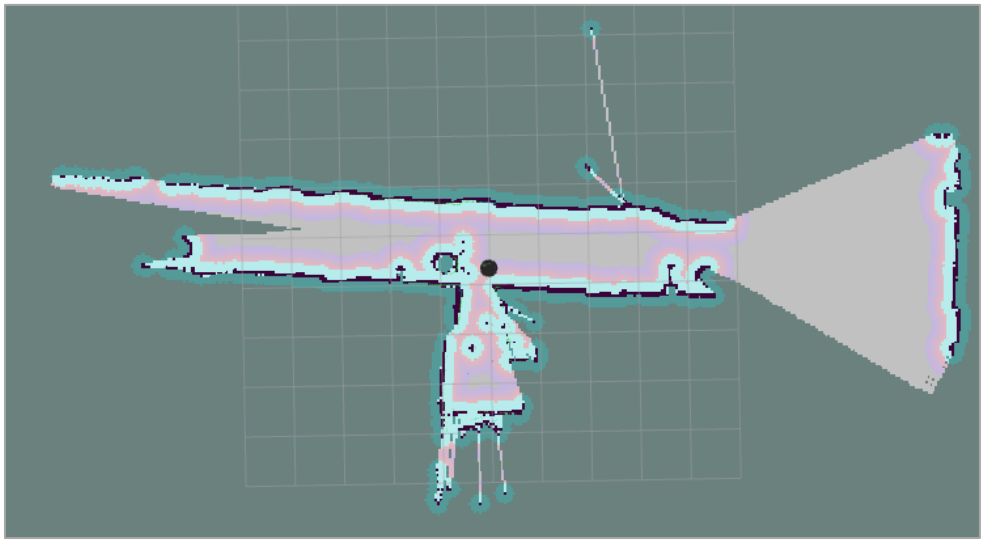

Figura 4.36: Construcción del mapa en tiempo real con el Turlebot

Una vez se ha construido el mapa en su totalidad y comprobamos que todos los obstáculos del se corresponden con la realidad del entorno, se guarda el mapa.

Al mismo momento en el que se estaba construyendo el mapa en tiempo real, se lanzó el comando rosbag para capturar los mismos topics y los mismos datos en un fichero *.bag*, para posteriormente construir otro mapa reproduciendo estos datos guardados.

4.4.1.2.2 Rosbag

Rosbag [41] es un conjunto de herramientas para grabar y reproducir topics en ROS. Con la herramienta *rosbag*, se pueden grabar datos reales de los sensores y actuadores del turtlebot, y posteriormente utilizarlo en nuestro PC para analizar dichos datos y también reproducirlos.

Esta herramienta es muy útil ya que en ocasiones puede no haber mucha disponibilidad de los robots reales, por lo que, en vez de trabajar sobre el robot real, esta herramienta permite grabar dichos datos, y luego posteriormente trabajar sobre ellos en nuestro ordenador personal sin la necesidad del robot real. También tiene una ventaja añadida, y es que estos datos se pueden reproducir cuantas veces se quiera.

Para poder construir un mapa a partir de los datos obtenidos por los sensores del turtlebot real, en primer lugar hay que grabar un archivo *.bag*.

Con el siguiente comando, capturamos todos los topics disponibles del robot turtlebot:

```
>> rosbag record -a
```

El comando *record* también permite otras opciones, como, por ejemplo, solo capturar los topics que se le especifiquen, dado que en muchas ocasiones no se necesitan todos los valores de los sensores. En este caso se capturan todos los topic activos. Para construir un mapa, solo son necesarios los topics /*scan* y /*tf.*

```
>> rosbag record /scan /tf
```

```
sandra@sandra-x550vx:~/mapapeque$ rosbag record /tf /scan
[ INFO] [1573990653.016837627]: Subscribing to /scan
[ INFO] [1573990653.021674283]: Subscribing to /tf
[ INFO] [1573990653.257712461, 55.771000000]: Recording to 2019-11-17-12-37-33.bag.
```

Figura 4.37: Salida por pantalla al ejecutar el comando para grabar un rosgab

Una vez grabado el *rosbag*, se puede obtener información sobre lo que se ha capturado con el siguiente comando:

```
>> rosbag info <archivo.bag>
```

```
path:        2019-11-17-12-37-33.bag
version:     2.0
duration:    2:13s (133s)
start:       Jan 01 1970 01:00:55.77 (55.77)
end:         Jan 01 1970 01:03:09.73 (189.73)
size:        23.9 MB
messages:    139067
compression: none [30/30 chunks]
types:       sensor_msgs/LaserScan [90c7ef2dc6895d81024acba2ac42f369]
             tf2_msgs/TFMessage    [94810edda583a504dfda3829e70d7eec]
topics:      /scan      1165 msgs    : sensor_msgs/LaserScan
             /tf      137902 msgs    : tf2_msgs/TFMessage    (2 connections)
```

Figura 4.38: Salida por pantalla de la información del rosbag

Entre otra información, muestra la duración de la captura, junto con el tamaño del archivo. También muestra qué tipo de mensajes ha capturado y qué topics, junto con la cantidad de mensajes capturados para cada topic.

Una vez se ha grabado el *rosbag* con el robot real. En nuestro PC, se reproduce para que se simule que hay un robot Turtlebot que está publicando y capturando información, y mientras tanto, se abre una nueva terminal donde se ejecuta los comandos necesarios

para construir el mapa del entorno real del robot, a partir de los datos capturados, por el robot Turtlebot.

A continuación, con el archivo *.bag* en nuestro PC personal, se reproduce con el siguiente comando:

```
>> rosbag play archivo.bag
```

Al mismo momento en el que empieza a reproducirse este *bag*, hay que ejecutar los dos comandos comentados en el apartado anterior para construir el mapa:

```
>> roslaunch turtlebot_navigation gmapping_demo.launch

>> roslaunch turtlebot_rviz_launchers view_navigation.launch
```

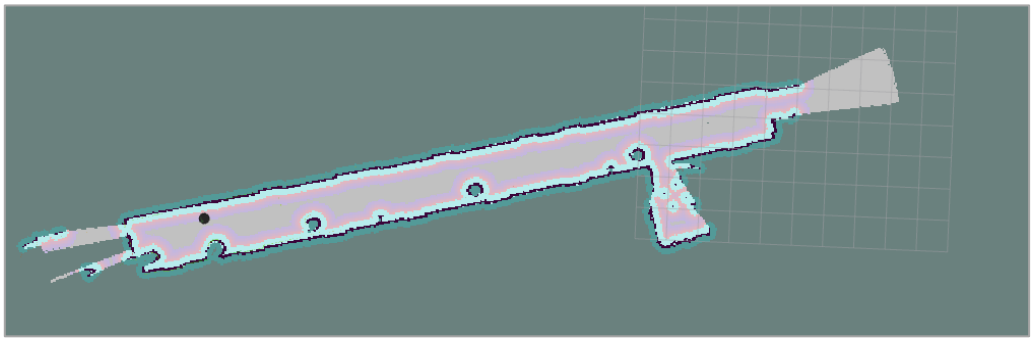

Figura 4.39: Construcción del mapa con el Turlebot real a partir de un rosbag

Como se puede observar, no tiene ninguna dificultad construir el mapa a partir de los datos de los sensores reales que ya habían sido capturados en otro momento.

A continuación, en la figura 4.40. tenemos el resultado de la construcción de los dos mapas. Estas figuras se corresponden a los dos archivos *.pgm* que crean al guardar los mapas. El de la izquierda es el mapa construido en tiempo real y el de la derecha, el que ha sido construido a partir de los datos capturados en un rosbag.

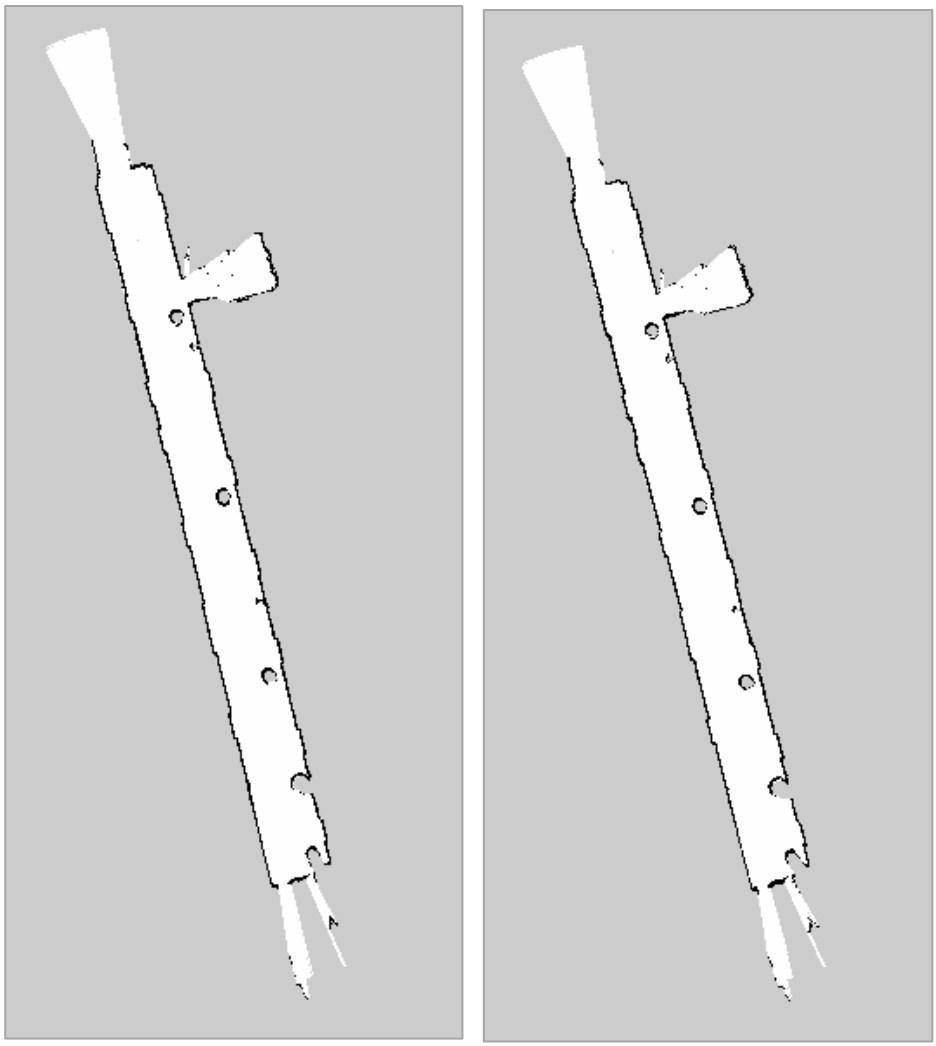

Figura 4.40: Mapas contruidos. Derecha: en tiempo real. Izquierda: rosbag

Como se puede observar en la figura 4.40, ambos mapas, el que ha sido creado en tiempo real, y el que ha sido construido a partir de un rosbag, son exactamente iguales.

Esto significa que con la herramienta *rosbag*, se pueden obtener los mismos resultados que al capturar los datos en tiempo real, y con una ventaja añadida, estos datos siempre se van a tener guardados y se puede reproducir tantas veces como se quiera.

Esta ventaja es muy interesante, ya que se pueden realizar distintas pruebas con los diferentes algoritmos modificando sus variables y así observar los cambios que se producen con los mismos datos. ya sea el algoritmo de construcción del mapa o cualquier otro.

4.4.2 Navegación autónoma a partir de un mapa conocido

En este apartado veremos cómo el turtlebot podrá moverse de forma autónoma a partir del mapa conocido que ya se ha sido creado en el apartado anterior.

Con ROS tenemos la capacidad de mover Turtlebot (o cualquier otro robot) de un lugar a otro, evitando obstáculos tanto estáticos como dinámicos, todo con unas pocas líneas de código.

4.4.2.1 Simulación

Como se va a mover el robot de forma autónoma, hay que iniciar el Turtlebot en el simulador Gazebo y en el mismo mundo que se ha mapeado. En este caso, será el mundo utilizado antes para construir el mapa.

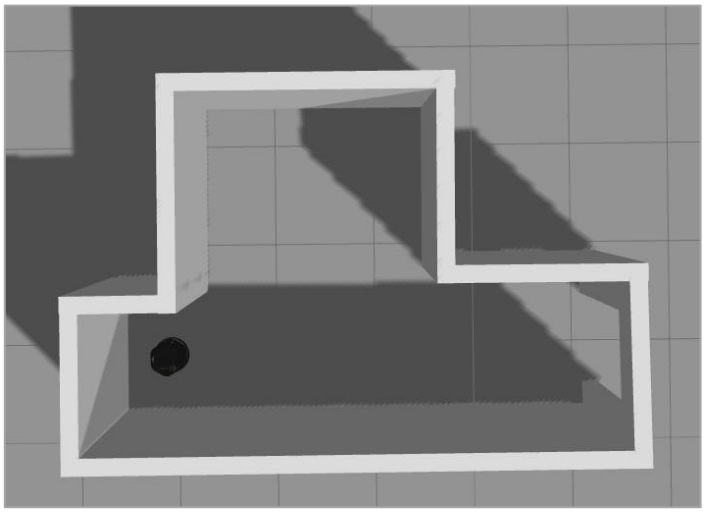

Figura 4.41: Mapa de Gazebo donde se va a realizar la navegación autónoma

Para la navegación del turtlebot de forma autónoma, lanzamos el siguiente comando indicando el mapa que se ha construido:

```
>> roslaunch turtlebot_navigation amcl_demo.launch
map_file:=my_map.yaml
```

El paquete ROS *Amcl* proporciona nodos para localizar al robot en un mapa estático. El nodo Amcl se subscribe a los datos del láser capturados en la construcción del mapa y

la información del robot. El nodo Amcl estima la pose del robot en el mapa y publica su posición respecto a él.

Cuando aparezca la salida: *Odom recieved!*, significa que el paquete *Amcl* está funcionando correctamente y se puede continuar con los siguientes pasos. En ocasiones este comando no recibe correctamente algunos parámetros, por lo que habrá que relanzarlo varias veces.

A continuación, podemos comenzar a ordenarle al robot que se coloque en una posición en el mapa usando Rviz. A continuación, se lanza el programa Rviz

```
>> roslaunch turtlebot_rviz_launchers view_navigation.launch
```

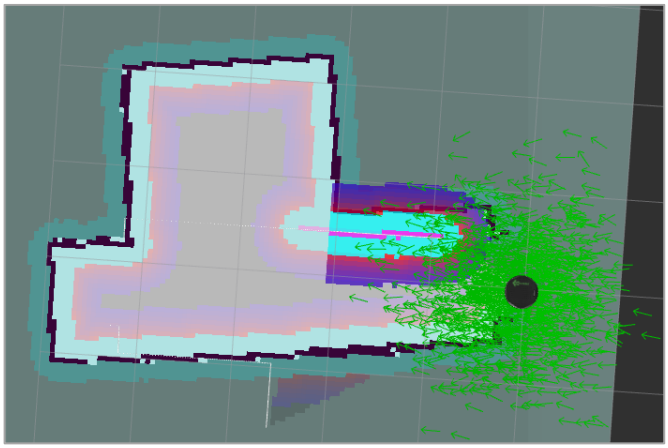

Figura 4.42: Posición inicial erronea en Rviz

Como se puede observar en la figura anterior, al iniciar Rviz el robot TurtleBot no es capaz de estimar su pose inicial, por lo que hay que proporcionársela.

En el programa Rviz, se selecciona la opción 2D Pose Estimate y a través de una flecha verde, se indica la posición y orientación aproximada del TurtleBot en el mapa.

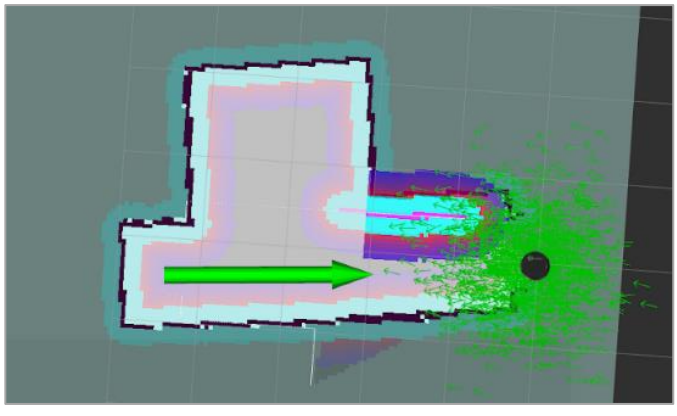

Figura 4.43: Opción 2D Pose Estimate de Rviz

Se observará una colección de flechas que son hipótesis de la posición de Turtlebot. El escaneo láser debe alinearse aproximadamente con las paredes del mapa. Si los obstáculos no se alinean bien, se puede repetir el procedimiento.

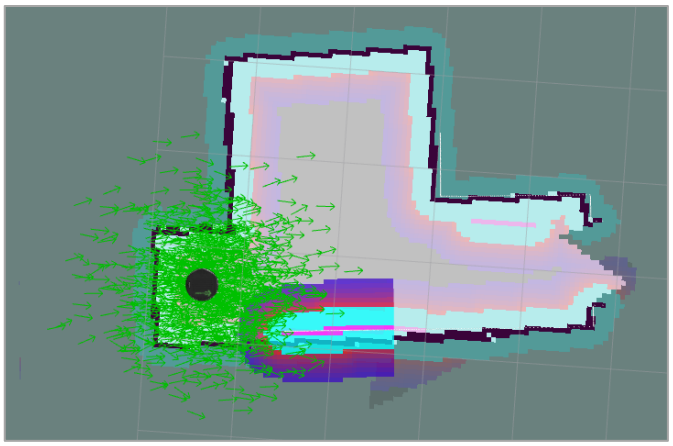

Figura 4.44: Posición inicial correcta en Rviz

Después de establecer su posición actual, podemos mover al robot turtlebot alrededor del mapa. Con la opción 2D Nav Goal, se puede indicar la posición y orientación objetivo que queremos que alcance el robot.

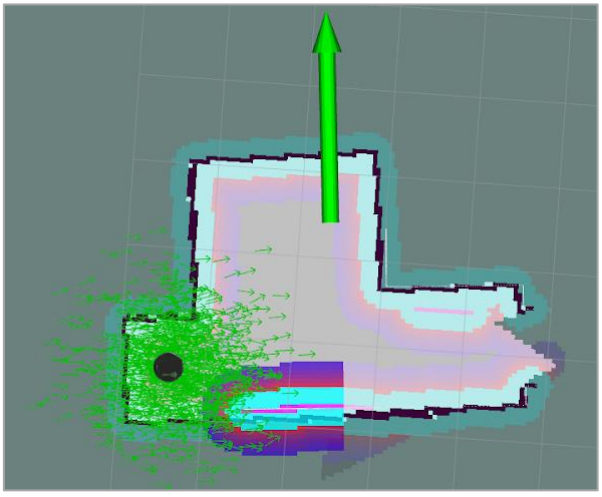

Figura 4.45: Opción 2D Nav Goal de Rviz

El turtlebot se moverá autónomamente sobre el mapa evitando los obstáculos para intentar alcanzar el objetivo indicado. En la siguiente imagen se observa el camino que va a recorrer para alcanzar el objetivo.

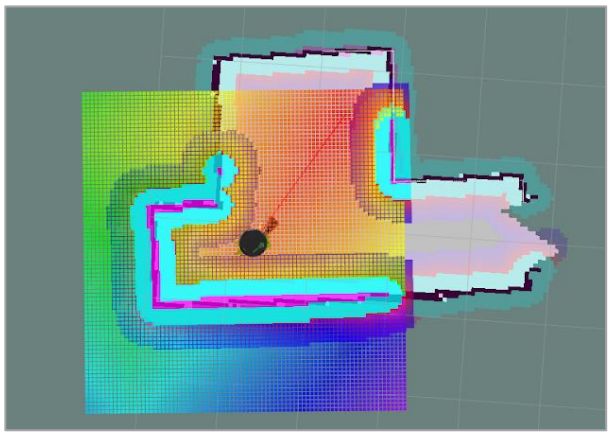

Figura 4.46: Navegación del Turtlebot al objetivo establecido

Finalmente, el robot llegará a la posición y orientación indicadas.

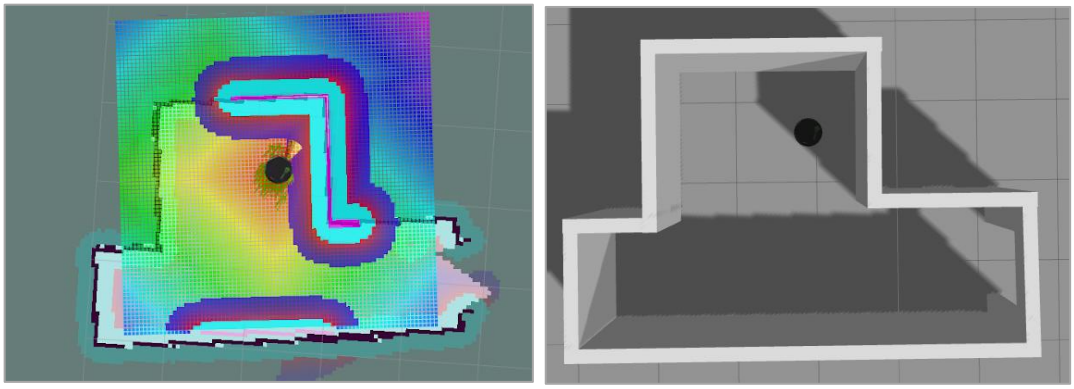

Figura 4.47: Objetivo alcanzado

A menudo, el robot tiene la tarea de moverse a una ubicación objetivo utilizando una herramienta preexistente como Rviz junto con un mapa. Sin embargo, también es importante poder enviar los objetivos del robot para moverse a una ubicación particular usando el código, al igual que Rviz.

Para desarrollar un programa ROS que permita al robot navegar hacia unas ubicaciones, primero necesitamos saber cuál es la coordenada (x,y) de esas ubicaciones en el mapa.

Utilizando la herramienta *rostopic echo*, podemos obtener la posición y orientación actual del robot.

```
sandra@sandra-x550vx:~/catkin_ws$ rostopic echo /amcl_pose
header:
  seq: 25
  stamp:
    secs: 847
    nsecs: 996000000
  frame_id: "map"
pose:
  pose:
    position:
      x: 0.701753631702
      y: -0.139564827027
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -0.167759481517
      w: 0.985827954747
```

Figura 4.48: Rostopic echo de /amcl_pose

A continuación, utilizamos estas coordenadas para definir una misión de navegación que enviamos a la pila de navegación del robot para ejecutarla [42].

Para crear un nodo ROS que envíe objetivos a la pila de navegación, lo primero que tendremos que hacer es crear un paquete. Se crea un paquete con las siguientes dependencias *move_base_msgs, actionlib* y *roscpp*.

Una vez el paquete se ha creado, se creará un archivo llamado *navigation.cpp* dentro de la carpeta *src* del paquete creado, donde se escribirá el código que moverá el robot Turtlebot al objetivo deseado.

```cpp
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>

int main(int argc, char** argv){

  ros::init(argc, argv, "map_navigation_node");
  ros::NodeHandle n;
  ros::spinOnce();

  actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> ac("move_base", true);

  //wait for the action server to come up
  while(!ac.waitForServer(ros::Duration(5.0))){
    ROS_INFO("Waiting for the move_base action server to come up");
  }

  move_base_msgs::MoveBaseGoal goal;

  //set up the frame parameters
  goal.target_pose.header.frame_id = "map";
  goal.target_pose.header.stamp = ros::Time::now();

  /* moving towards the goal*/
  goal.target_pose.pose.position.x =  3.7;
  goal.target_pose.pose.position.y =  -0.139;
  goal.target_pose.pose.orientation.w = 1.0;

  ROS_INFO("Sending goal location ...");
  ac.sendGoal(goal);
  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED){
    ROS_INFO("You have reached the destination");
  }
  else{
    ROS_INFO("The robot failed to reach the destination");
  }
  return 0;
}
```

Figura 4.49: Código del fichero navigation.cpp

A continuación, se van a explicar las líneas de código utilizadas en este fichero de navegación.

```
 actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
ac("move_base", true);
```

Esta línea construye un cliente de acción que se utilizará para la comunicación con move_base.

```
 while(!ac.waitForServer(ros::Duration(5.0))){
      ROS_INFO("Waiting for the move_base action server to come up");
 }
```

Estas líneas esperan a que el servidor de acción informe de que ha surgido y está listo para comenzar a procesar los objetivos.

```
 move_base_msgs::MoveBaseGoal goal;
 goal.target_pose.header.frame_id = "map";
 goal.target_pose.header.stamp = ros::Time::now();
```

Aquí se crea una meta para enviar a move_base.

```
 goal.target_pose.pose.position.x =  3.7;
 goal.target_pose.pose.position.y =  -0.139;
 goal.target_pose.pose.orientation.w = 1.0;
```

Aquí se especifican las coordenadas y la posición del objetivo a alcanzar.

```
 ac.sendGoal(goal);
 ac.waitForResult();
 ac.getState()
```

Aquí se envía la meta. Se espera por el resultado y por último se lee el estado de la acción, para comprobar si se ha alcanzado el objetivo o ha fallado.

Antes de compilar el paquete, se añaden las líneas siguientes al fichero *CMakeList.txt*

```
add_executable(navigation src/navigation.cpp)
target_link_libraries(navigation ${catkin_LIBRARIES})
```

Para ejecutar el paquete, tenemos que tener ejecutado los paquetes *amcl* y *Rviz*:

```
>> roslaunch turtlebot_navigation amcl_demo.launch
map_file:=my_map.yaml

>> roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Por último, se ejecuta el paquete con el siguiente comando:

```
>> rosrun mi_paquete navigation
```

4.4.2.2 Turtlebot real

En este apartado se va a mover el robot real de forma autónoma a partir de un mapa conocido. El mapa va a ser el que se ha construido en el apartado anterior.

Para poder localizar al robot, se utilizan los mismos comandos que se han utilizado en simulación. Solo hay que conocer, que comando utilizar en cada PC.

Como se ha comentado también en el apartado anterior, para poder realizar la localización del robot turtlebot real, los dos PCs, el del turtlebot y el personal, deben estar conectados al mismo ROS Master.

Una vez, tenemos los nodos ROS lanzados que activan los sensores y actuadores necesarios para realizar esta aplicación, ejecutamos el siguiente comando en el PC del turtlebot.

```
>> roslaunch turtlebot_navigation amcl_demo.launch
map_file:=my_map.yaml
```

El turtlebot va a utilizar el fichero *my_map.yaml* para la localización, el cual es el mapa que se ha construido antes. Hay que tener en cuenta que este mapa debe estar en el PC del turtlebot, por lo que si se ha guardado en el PC personal, se tiene que transferir los archivos del mapa (*my_map.yaml* y *my_map.png*)

Como también se ha comentado en el anterior apartado, Rviz necesita mucha capacidad de cómputo, por lo que se ejecuta en el pc personal el comando que inicia el visualizador:

```
>> roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Rviz utiliza los datos de los sensores y actuadores para poder localizar y mover al robot turtlebot. En el pc personal, al instalar el paquete turtlebot, este incluye por defecto los archivos necesarios para la utilización de la base Kobuki y la cámara Astra. Para esta aplicación también es necesario el láser Hokuyo.

Al ejecutar el comando que inicia el visualizador Rviz, se obtiene un error en el que se indica que no se localiza un fichero perteneciente al láser Hokuyo.

```
[ERROR] [1573207147.426277939]: Could not load resource [package://turtlebot_description/meshes/
sensors/hokuyo_ust_10lx.dae]: Unable to open file "package://turtlebot_description/meshes/sensor
s/hokuyo_ust_10lx.dae".
[ERROR] [1573207147.426535853]: Could not load model 'package://turtlebot_description/meshes/sen
sors/hokuyo_ust_10lx.dae' for link 'hokuyo_laser_link': OGRE EXCEPTION(6:FileNotFoundException):
 Cannot locate resource package://turtlebot_description/meshes/sensors/hokuyo_ust_10lx.dae in re
source group Autodetect or any other group. in ResourceGroupManager::openResource at /build/ogre
-1.9-mqY1wq/ogre-1.9-1.9.0+dfsg1/OgreMain/src/OgreResourceGroupManager.cpp (line 756)
```

Figura 4.50: Error obtenido al ejecutar el comando del Rviz

Este error es debido a que el láser Hokuyo no pertenece en sí a la plataforma TurtleBot, por lo que no se instala al instalar sus paquetes. El paquete perteneciente al láser, el cual ya se encontraba instalado en el TurtleBot, es necesario también en el PC personal si se quieren utilizar sus datos.

Para solucionar este problema, solo es necesario copiar un archivo *.dae* perteneciente al láser Hokuyo en el paquete *turltebot_description* de nuestro ordenador.

Con el siguiente comando, se copia el fichero *hokuyo_ust_10lx.dae* en el directorio que se le indica.

```
>> cp hokuyo_ust_10lx.dae
/opt/ros/kinetic/share/turtlebot_description/meshes/sensors
```

Rviz abrirá el mapa especificado en el comando e insertará al robot en una posición cualquiera del mapa. Como se ha comentado en simulación, tenemos que indicar la posición y orientación inicial del Turtlebot.

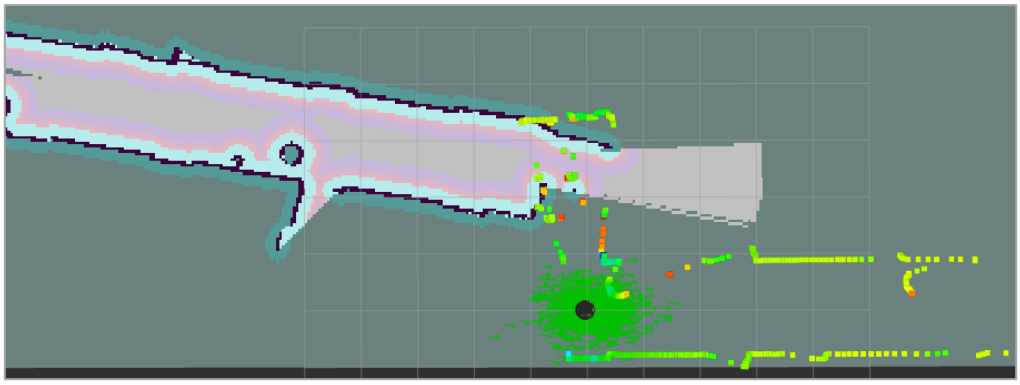

Figura 4.51: Inicio de la localización en Rviz

Como ocurría en simulación, el robot Turtlebot desconoce su posición y orientación inicial. De la misma forma, se le proporciona estos datos con la opción 2D Pose Estimate.

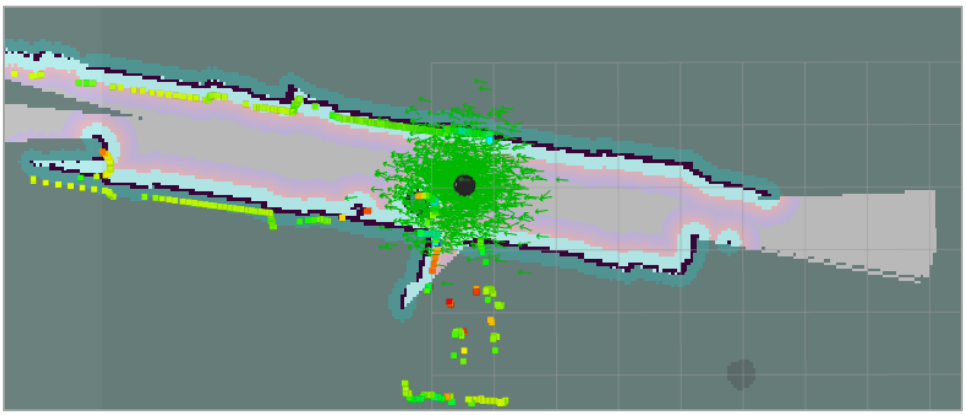

Figura 4.52: Turlebot localizado en el mapa en Rviz

En estas pruebas realizadas, se observa que el robot real se comporta de la misma forma que el Turtlebot simulado. El robot se mueve para alcanzar el objetivo indicado sin ningún problema.

4.5 Conexión Matlab-ROS

En este apartado se va a explicar cómo conectarse y controlar el robot turtlebot con otro ordenador con sistema Windows desde el programa Matlab. Con esta conexión se van a realizar pruebas con el robot Turtlebot simulado en Gazebo, y con el robot real [43].

En este caso, se va a iniciar al maestro en el PC del Turtlebot. A continuación, Matlab creará un ROS NODE el cual va a estar conectado a ese ROS MASTER. Después, el robot Turtlebot también se conectará al Master a partir de otro nodo ROS.

Básicamente consiste en realizar una conexión a partir de la IP del ordenador donde se encuentra el Master de ROS ejecutándose, en este caso, en el PC del Turtlebot. Para poder realizar la conexión a través de Matlab, es necesario que ambos ordenadores estén conectados en la misma red.

Desde el programa Matlab se puede controlar el Turtlebot o a través de la ventana de comandos o a partir de un script.

4.5.1 Command Window

En primer lugar, se van realizar una serie de pruebas iniciales con la Command Window de Matlab.

Dado que, en estas pruebas, se está trabajando con el sistema operativo Windows, se hace una conexión SSH con el PC del Turtlebot, y así, desde nuestro PC personal, podemos tener acceso a los dos ordenadores.

Con el Ros Master ya funcionando en el PC del Turtlebot. En Matlab, el primer paso que se tiene que hacer es apagar la conexión con cualquier otro nodo, para cerrar las entidades ROS globales creadas por rosinit, por si alguna conexión anterior no se ha cerrado:

```
>> rosshutdown
```

rosshutdown cierra el nodo global y el ROS Master.

A continuación, se establece la conexión con el ROS Master que se encuentra en el PC del robot con el siguiente comando:

```
>> rosinit (hostname)
```

Este comando inicia el nodo ROS global con un nombre MATLAB predeterminado e intenta conectarse al ROS Master que se ejecuta en localhost y en el puerto 11311. Si el nodo ROS global no puede conectarse al maestro ROS, *rosinit* también inicia un núcleo ROS en MATLAB, que consiste de un maestro ROS, un servidor de parámetros ROS y un nodo de registro de *rosout*.

Una vez se ha establecido la conexión, ejecutamos el comando *rostopic list* para comprobar que la conexión se ha realizado correctamente.

Figura 4.53: Command Window de Matlab

Como se puede ver en la figura 4.53, la conexión entre Matlab y el ROS Master, el cual también está conectado el Turtlebot, está bien establecida.

Desde esta terminal de Matlab también se puede publicar sobre un topic o subscribirse, aunque en un script se puede hacer mucho más. En Matlab ocurre lo mismo que en ROS de Ubuntu, a partir del terminal se pueden realizar ciertas acciones, pero a partir de un archivo de código se pueden crear comportamientos mucho más complejos.

4.5.2 Script

A continuación, tenemos un ejemplo de conexión a partir de un script de Matlab, donde se publicará en el topic que controla el Turtlebot un movimiento circular continuo.

```
1
2     rosshutdown;
3     rosinit('192.168.56.103',11311)
4     rosinit('192.168.43.252',11311)
5
6     node_teleop = rospublisher('/cmd_vel_mux/input/teleop');
7     teleop_msg = rosmessage ('geometry_msgs/Twist');
8
9     msg.Linear.X = 0.3;
10    msg.Linear.Y = 0;
11    msg.Linear.Z = 0;
12
13    msg.Angular.X = 0;
14    msg.Angular.Y = 0;
15    msg.Angular.Z = 0.2;
16
17    while(1)
18        send (node_teleop, msg);
19    end
```

Figura 4.54: Script en Matlab, publicador al topic de teleoperación

Como se ha comentado antes, lo primero es cerrar cualquier conexión anterior, y a continuación, conectarnos al ROS Master que se encuentra en el PC con el sistema operativo Ubuntu a través de la IP.

Una vez se ha establecido la conexión, si se quiere publicar en un topic, tenemos que crear un *publisher* con:

```
pub = publisher(topicname)
```

Después hay que crear un mensaje del mismo tipo que el del topic.

```
msg = rosmessage(messagetype)
```

A continuación, se escribe el valor que se quiere publicar en el topic. Esto depende de la estructura que tenga el mensaje del topic. Por ejemplo, si el topic en el que queremos escribir es el */mobile_base/commands/velocity*, el cual es el que controla los movimientos del Turtlebot. El mensaje del topic se compone de 6 variables. Correspondientes a los ejes X, Y y Z del movimiento lineal y angular.

Por lo que, para darle valor a la variable *Linear.X*, se escribiría:

```
msg.Linear.X = 0.5
```

Por último, se publicaría el mensaje como:

```
send(pub, msg)
```

send() publica un mensaje al tema especificado por el editor, pub. Este mensaje puede ser recibido por todos los suscriptores de la red ROS que estén suscritos al tema especificado por pub.

En Matlab también se puede subscribirse a un topic.

```
node = rossubscriber('/topic_name')
```

Después, con la siguiente orden, se leen los mensajes publicados en ese topic.

```
msg = receive(node, timeout)
```

se especifica en el *timeout* el número de segundos para esperar a recibir un mensaje del topic al que está suscrito node. Si no se recibe ningún mensaje en el tiempo especificado, se obtendrá un error.

A continuación, se va a escribir otro script, donde se creará un nodo que se suscribirá al topic del láser y mostrará por pantalla los datos recibidos.

```
1
2   node_subscriptor = rossubscriber('/scan');
3   laser_msg = receive(node_subscriptor, 2);
4
5   figure
6   plot(laser_msg,'MaximumRange',7)
```

Figura 4.55: Script en Matlab, suscriptor al topic del láser

En Matlab, se puede observar la estructura del mensaje que se recibe del topic *scan*. Las variables que contiene el mensaje y sus valores.

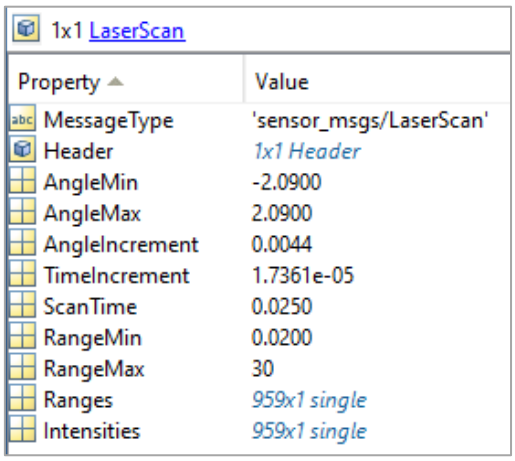

Figura 4.56: Estructura del mensaje del láser

En la siguiente línea de código se tiene un ejemplo de cómo acceder a estas variables que componen el mensaje recibido por el láser, en concreto, se va a acceder al campo RanMax del mensaje.

```
valor = msg.RangeMax
```

Algunos tipos de mensajes tienen visualizadores asociados a ellos. El tipo de mensaje *LaserScan*, tiene la opción *plot*, que muestra una gráfica de los datos capturados.

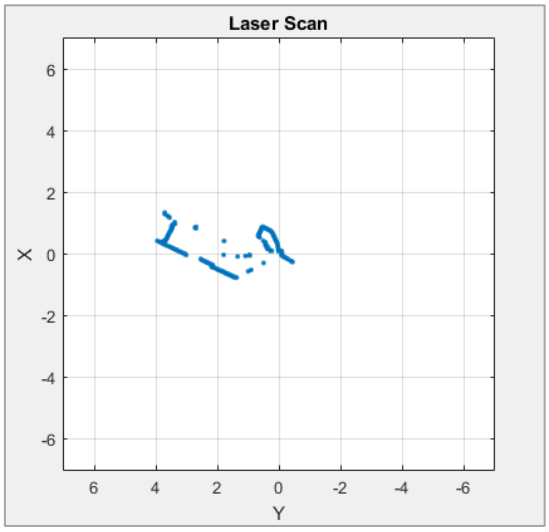

Figura 4.57: Captura del láser

En vez de usar la función *receive()* para obtener los datos, se puede especificar una función para que sea llamada cuando un mensaje se reciba. Esto permite que otro código de Matlab se ejecute mientras el suscriptor está esperando nuevos mensajes. Los *callbacks* son esenciales si se quiere utilizar varios subscriptores.

Desde Matlab, también se pueden recibir las imágenes capturadas por la cámara del Turtlebot.

```
1
2       imsub = rossubscriber('/camera/rgb/image_raw');
3       img = receive(imsub);
4       figure
5       imshow(readImage(img))
6
```

Figura 4.58: Script en Matlab, suscriptor al topic de la cámara

En la figura 4.58 tenemos el código de un script de Matlab, donde se crea un nodo suscriptor al topic de la cámara del Turtlebot y se muestra por pantalla, el mensaje recibido por ese topic, en este caso, una imagen rgb, como se muestra en la figura 4.59.

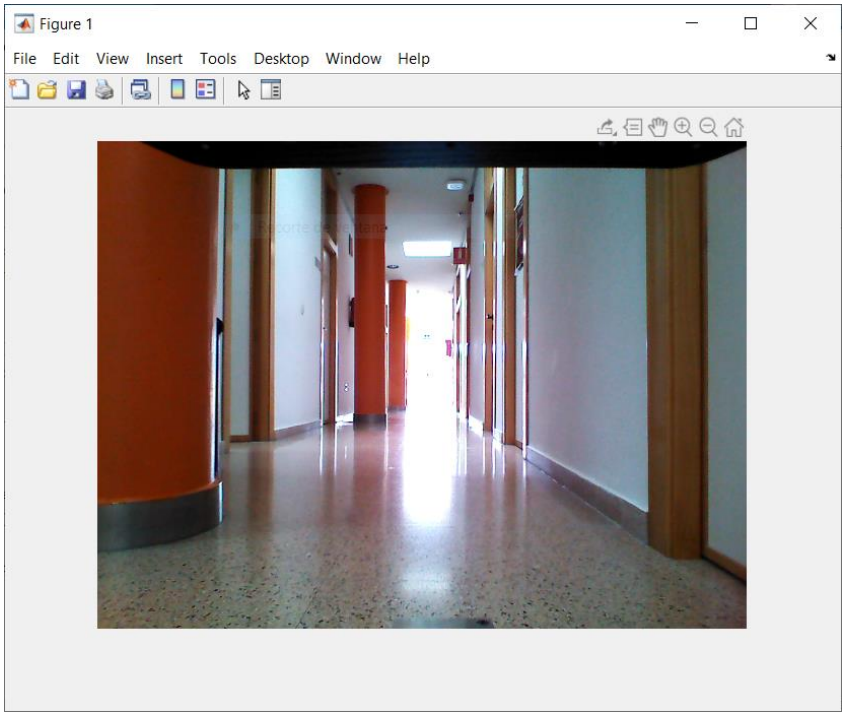

Figura 4.59: Captura de la cámara del Turltebot recibida en Matlab

Matlab también permite trabajar con rosbags [44]. Con el siguiente comando, se insertan los ficheros .bag en Matlab.

```
>> bag = rosbag('ex_multiple_topics.bag')
```

El objeto devuelto por la llamada es una representación de todos los mensajes en el rosbag.

La visualización del objeto muestra los detalles sobre cuántos mensajes están contenidos en el archivo y la hora en que se registraron el primer y el último mensaje.

Con el siguiente comando se puede obtener información sobre los topics y tipos de mensaje que están registrados en el *rosbag*:

```
>> bag.AvailableTopics
```

	NumMessages	MessageType
/camera/depth_registered/sw_registered/image_rect/compressed/parameter_descriptions	1	dynamic_reconfigure/ConfigDescription
/camera/depth_registered/sw_registered/image_rect/compressed/parameter_updates	1	dynamic_reconfigure/Config
/camera/depth_registered/sw_registered/image_rect/compressedDepth/parameter_descriptions	1	dynamic_reconfigure/ConfigDescription
/camera/depth_registered/sw_registered/image_rect/compressedDepth/parameter_updates	1	dynamic_reconfigure/Config
/camera/depth_registered/sw_registered/image_rect/theora/parameter_descriptions	1	dynamic_reconfigure/ConfigDescription
/camera/depth_registered/sw_registered/image_rect/theora/parameter_updates	1	dynamic_reconfigure/Config
/capability_server/bonds	366	bond/Status
/cmd_vel_mux/active	1	std_msgs/String
/cmd_vel_mux/input/teleop	3639	geometry_msgs/Twist
/cmd_vel_mux/parameter_descriptions	1	dynamic_reconfigure/ConfigDescription
/cmd_vel_mux/parameter_updates	1	dynamic_reconfigure/Config

Figura 4.60: Información del .bag

En Matlab, también tenemos la aplicación SLAM Map Builder [45] que carga capturas del láser y datos del sensor de odometría para construir una cuadrícula de ocupación 2D utilizando algoritmos de localización y mapeo simultáneos (SLAM).

Esta aplicación permite ajustar la configuración del algoritmo SLAM para mejorar la creación automática de mapas.

Para utilizar esta herramienta, ejecutamos en la Command Window de Matlab el siguiente comando:

```
>> slamMapBuilder(bag)
```

Este comando abre la aplicación SLAM Map Builder e importa el archivo de registro *rosbag* especificado en *bag*.

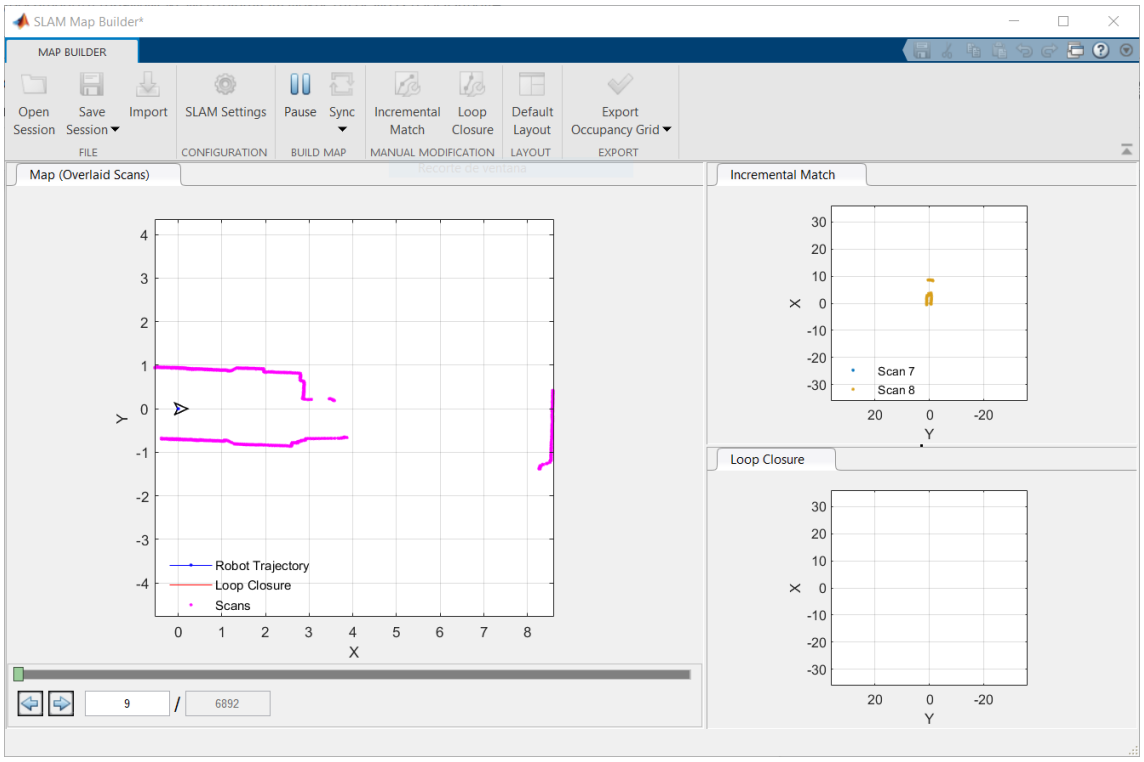

Figura 4.61: Aplicación SLAM Map Builder de Matlab

A partir de este mapa construido por esta aplicación de Matlab, también se podría crear un script que envíe posiciones a alcanzar para que el robot Turtlebot navegue de forma autónoma.

5. Conclusiones y trabajo futuro

En este trabajo se ha explicado el funcionamiento del framework ROS. Se ha explicado la arquitectura de grafos en la que funcionan las conexiones entre los nodos ROS. También, se han explicado los principales elementos de esta red de grafos. Junto con las distintas herramientas proporcionadas por ROS para la realización de este trabajo.

Se ha conseguido hacer la puesta en marcha del robot Turtlebot y de sus dispositivos adicionales. También hacer la puesta en marcha del robot Turtlebot en el simulador Gazebo. Y se han realizado las primeras pruebas con los sensores y actuadores del robot, para verificar el correcto comportamiento de éste.

Con el robot real, se ha conseguido realizar una conexión entre los dos PC, el del Turtlebot y el ordenador personal, en la que ambos PC se encuentran dentro de la misma arquitectura de grafos de ROS, donde se consigue una comunicación transversal en la que desde los dos PC se pueden leer y escribir sobre los mismos nodos, y ejecutar nodos de forma bidireccional. Gracias a esta conexión, se ha podido utilizar el visualizador Rviz para visualizar las capturas de los sensores del Turtlebot en tiempo real.

En cuanto a la simulación, se ha visto cómo crear distintos mundos en diversas herramientas y posteriormente cómo insertar estos mundos dentro del simulador Gazebo junto con el robot Turtlebot. Se ha construido un entorno básico con el editor 3D del simulador Gazebo, y otro mundo un poco más detallado creado en el editor SketchUp, en concreto, una carretera con distintas curvas que se encuentra delimitada por unas líneas blancas que intenta simular el entorno de una carretera real. También se ha creado un paquete donde se han insertado varios robots Turtlebot dentro del mundo especificado en el simulador Gazebo, el cual ayudará a desarrollar futuros algoritmos para crear comportamientos multirobots.

En cuanto a ROS, se ha visto cómo crear un espacio de trabajo y cómo utilizarlo para diversos paquetes con distintos desarrollos sobre el Turtlebot, tanto paquetes para el robot real como para la simulación. En concreto, se han creado unos paquetes básicos de un nodo publicador y un suscriptor conectados con los nodos del Turtlebot.

También se han creado otro paquete donde se ha utilizado la base del funcionamiento de los paquetes anteriores. Este paquete realiza un procesamiento de las imágenes obtenidas del Turtlebot simulado en Gazebo en el mundo creado en el editor 3D. Se utiliza la visión artificial desarrollada con la librería de OpenCV, donde se detectan las líneas de la carretera del mundo y a partir de los datos de estas rectas, se calculan las

velocidades lineales y angulares que se publican para mover al robot a través de esta carretera de una forma autónoma.

Por otro lado, se han construido diversos mapas a través de los datos capturados a partir de los ficheros *bags* y en tiempo real, del robot en simulación y del turtlebot real. Y cómo mejorar estos mapas al modificar distintas variables del paquete que lo construye. También se ha conseguido mover al robot Turtlebot de forma autónoma solo indicando el objetivo a alcanzar a partir de estos mapas conocidos.

Por último, se ha conseguido realizar distintas pruebas con la herramienta ROS Toolbox de Matlab, en las que Matlab se ha conectado al mismo ROS Master en el que se encontraba también conectados los nodos del Turtlebot, real y simulado. A través de Matlab se han leído los datos de los sensores del Turtlebot a través de la creación de suscriptores; y también se ha conseguido publicar sobre el topic que controla los movimientos del robot a través de la creación de un publicador. Por otro lado, se ha construido un mapa a partir de un *rosbag* grabado con los datos del Turtlebot.

Como trabajo futuro se pueden utilizar las distintas herramientas ROS utilizadas en este trabajo para diseñar y desarrollar comportamientos más complejos con el robot Turtlebot. Estos comportamientos se podrían desarrollar con Rospy, la librería de clientes ROS de Python y también la librería desarrollada en C++ (PCL), para el análisis de las nubes de puntos.

Por otro lado, se podría ver la puesta en marcha y la utilización del brazo PhantomX reactor que se encuentra instalado en el robot Turtlebot utilizado en este trabajo. También se podrían estudiar y utilizar otras herramientas de ROS junto con el Turtlebot, como el simulador 2D Stage, el simulador Unity, también Simulink de Matlab para el control del Turtlebot.

Bibliografía

[1] «ROS,» [En línea]. Available: https://www.ros.org/. [Último acceso: Noviembre 2019].

[2] B. G. a. W. D. S. Morgan Quigley, «Programming Robots with ROS,» 2015, pp. 13-14.

[3] «ROS robots,» [En línea]. Available: https://robots.ros.org/. [Último acceso: Noviembre 2019].

[4] Robot móvil, «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Robot_m%C3%B3vil. [Último acceso: Septiembre 2019].

[5] «ROS Components,» Robot Turtlebot, [En línea]. Available: https://www.roscomponents.com/es/robots-moviles/9-turtlebot-2.html#/3d_sensor-no/controlador-no/estacion_de_carga-no/tipo_de_cable-europa_cee_7_16/bateria_adicional-no/disco_extra-no/montaje-no/brazo_robotico-no/cursos-no. [Último acceso: Octubre 2019].

[6] «TD Robótica,» [En línea]. Available: http://aprender.tdrobotica.co/turtlebot-2/. [Último acceso: Septiembre 2019].

[7] «ROS Components,» Laser Hokuyo UST-10LX, [En línea]. Available: https://www.roscomponents.com/es/precio-visible/ 85-ust-10lx.html. [Último acceso: Septiembre 2019].

[8] «ROS Components,» Cámara 3D Astra Orbbec, [En línea]. Available: https://www.roscomponents.com/es/camaras/ 76-orbbec.html. [Último acceso: Septiembre 2019].

[9] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/es. [Último acceso: Octubre 2019].

[10] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/ROS/Concepts. [Último acceso: Septiembre 2019].

[11] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/Master. [Último acceso: Septiembre 2019].

[12] «Gazebo simulator,» [En línea]. Available: http://gazebosim.org/. [Último acceso: Septiembre 2019].

[13] «ROS Wiki,» Visualizador Rviz, [En línea]. Available: http://wiki.ros.org/rviz. [Último acceso: Septiembre 2019].

[14] «Mathworks,» Matlab, [En línea]. Available: https://www.mathworks.com/products/matlab.html. [Último acceso: Septiembre 2019].

[15] «MathWorks,» ROS Toolbox, [En línea]. Available: https://es.mathworks.com/products/ros.html. [Último acceso: Septiembre 2019].

[16] «Wikipedia,» Lenguaje de programación C++, [En línea]. Available: https://es.wikipedia.org/wiki/C%2B%2B. [Último acceso: Septiembre 2019].

[17] «ROS Wiki,» Roscpp, [En línea]. Available: http://wiki.ros.org/roscpp. [Último acceso: Octubre 2019].

[18] «Wikipedia,» OpenCV, [En línea]. Available: https://es.wikipedia.org/wiki/OpenCV. [Último acceso: Septiembre 2019].

[19] «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/VNC. [Último acceso: Septiembre 2019].

[20] Reminna, [En línea]. Available: https://www.redeszone.net/2014/12/20/remmina-un-cliente-de-escritorio-remoto-para-linux/. [Último acceso: Septiembre 2019].

[21] «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Secure_Shell. [Último acceso: Septiembre 2019].

[22] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/roscore. [Último acceso: Septiembre 2019].

[23] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/turtlebot_bringup. [Último acceso: Septiembre 2019].

[24] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/rosnode. [Último acceso: Septiembre 2019].

[25] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/rqt_graph. [Último acceso: Septiembre 2019].

[26] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/rostopic. [Último acceso: Septiembre 2019].

[27] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/image_view. [Último acceso: Septiembre 2019].

[28] «Gaitech EDU,» [En línea]. Available: http://edu.gaitech.hk/turtlebot/teleop-doc.html. [Último acceso: Septiembre 2019].

[29] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/turtlebot_gazebo. [Último acceso: Septiembre 2019].

[30] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/turtlebot_rviz_launchers. [Último acceso: Septiembre 2019].

[31] «ROS WIki,» [En línea]. Available: http://wiki.ros.org/turtlebot_interactive_markers/Tutorials/indigo/UsingTurtlebotInteractiveMarkers. [Último acceso: Septiembre 2019].

[32] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/catkin. [Último acceso: Septiembre 2019].

[33] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/ROS/Tutorials/catkin/CreatingPackage. [Último acceso: Septiembre 2019].

[34] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29. [Último acceso: Septiembre 2019].

[35] «Gazebo simulator,» [En línea]. Available: http://gazebosim.org/tutorials?cat=build_world&tut=building_editor. [Último acceso: Octubre 2019].

[36] «ROS Answers,» [En línea]. Available: https://answers.ros.org/question/197701/multiple-turtlebots-simulation-in-gazebo-and-visualization-in-rviz/. [Último acceso: Octubre 2019].

[37] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/turtlebot_gazebo/Tutorials/indigo/Make%20a%20map%20and%20navigate%20with%20it.

[38] «EDU Gaitech,» [En línea]. Available: https://edu.gaitech.hk/turtlebot/create-map-kenict.html. [Último acceso: Noviembre 2019].

[39] «ROS Answers,» [En línea]. Available: https://answers.ros.org/question/269280/how-to-make-better-maps-using-gmapping/. [Último acceso: Noviembre 2019].

[40] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/turtlebot/Tutorials/indigo/Network%20Configuration. [Último acceso: Noviembre 2019].

[41] «ROS Wiki,» [En línea]. Available: http://wiki.ros.org/rosbag/Commandline. [Último acceso: Noviembre 2019].

[42] «EDU Gaitech,» [En línea]. Available: http://edu.gaitech.hk/turtlebot/map-navigation.html. [Último acceso: Noviembre 2019].

[43] «Mathworks,» [En línea]. Available: https://www.mathworks.com/help/ros/ug/get-started-with-ros.html. [Último acceso: Noviembre 2019].

[44] «Mathworks,» [En línea]. Available: https://es.mathworks.com/help/robotics/ref/rosbag.html. [Último acceso: Noviembre 2019].

[45] «Mathworks,» [En línea]. Available: https://es.mathworks.com/help/nav/ref/slammapbuilder-app.html. [Último acceso: Noviembre 2019].

ANEXO 1. Lista de topics de los sensores y actuadores del Turtlebot

Paquete bringup minimal

```
turtlebot@turtlebot:~/Downloads/sandra/catkin_ws$ rostopic list
/capability_server/bonds
/capability_server/events
/cmd_vel_mux/active
/cmd_vel_mux/input/navi
/cmd_vel_mux/input/safety_controller
/cmd_vel_mux/input/switch
/cmd_vel_mux/input/teleop
/cmd_vel_mux/parameter_descriptions
/cmd_vel_mux/parameter_updates
/diagnostics
/diagnostics_agg
/diagnostics_toplevel_state
/gateway/force_update
/gateway/gateway_info
/info
/interactions/interactive_clients
/interactions/pairing
/joint_states
/mobile_base/commands/controller_info
/mobile_base/commands/digital_output
/mobile_base/commands/external_power
/mobile_base/commands/led1
/mobile_base/commands/led2
/mobile_base/commands/motor_power
/mobile_base/commands/reset_odometry
/mobile_base/commands/sound
/mobile_base/commands/velocity
/mobile_base/controller_info
/mobile_base/debug/raw_control_command
/mobile_base/debug/raw_data_command
/mobile_base/debug/raw_data_stream
/mobile_base/events/bumper
/mobile_base/events/button
/mobile_base/events/cliff
/mobile_base/events/digital_input
/mobile_base/events/digital_input
/mobile_base/events/power_system
/mobile_base/events/robot_state
/mobile_base/events/wheel_drop
/mobile_base/sensors/bumper_pointcloud
/mobile_base/sensors/core
/mobile_base/sensors/dock_ir
/mobile_base/sensors/imu_data
/mobile_base/sensors/imu_data_raw
/mobile_base/version_info
/mobile_base_nodelet_manager/bond
/odom
/rosout
/rosout_agg
/tf
/tf_static
/turtlebot/incompatible_rapp_list
/turtlebot/rapp_list
/turtlebot/status
/zeroconf/lost_connections
/zeroconf/new_connections
turtlebot@turtlebot:~/Downloads/sandra/catkin_ws$
```

Paquete bringup Hokuyo

```
turtlebot@turtlebot:~/Downloads/sandra/catkin_ws$ rostopic list
/diagnostics
/hokuyo/parameter_descriptions
/hokuyo/parameter_updates
/rosout
/rosout_agg
/scan
/zeroconf/lost_connections
/zeroconf/new_connections
```

Paquete Astra

```
turtlebot@turtlebot:~/Downloads/sandra/catkin_ws$ rostopic list
/camera/camera_nodelet_manager/bond
/camera/depth/camera_info
/camera/depth/image
/camera/depth/image/compressed
/camera/depth/image/compressed/parameter_descriptions
/camera/depth/image/compressed/parameter_updates
/camera/depth/image/compressedDepth
/camera/depth/image/compressedDepth/parameter_descriptions
/camera/depth/image/compressedDepth/parameter_updates
/camera/depth/image/theora
/camera/depth/image/theora/parameter_descriptions
/camera/depth/image/theora/parameter_updates
/camera/depth/image_raw
/camera/depth/image_raw/compressed
/camera/depth/image_raw/compressed/parameter_descriptions
/camera/depth/image_raw/compressed/parameter_updates
/camera/depth/image_raw/compressedDepth
/camera/depth/image_raw/compressedDepth/parameter_descriptions
/camera/depth/image_raw/compressedDepth/parameter_updates
/camera/depth/image_raw/theora
/camera/depth/image_raw/theora/parameter_descriptions
/camera/depth/image_raw/theora/parameter_updates
/camera/depth/image_rect
/camera/depth/image_rect/compressed
/camera/depth/image_rect/compressed/parameter_descriptions
/camera/depth/image_rect/compressed/parameter_updates
/camera/depth/image_rect/compressedDepth
/camera/depth/image_rect/compressedDepth/parameter_descriptions
/camera/depth/image_rect/compressedDepth/parameter_updates
/camera/depth/image_rect/theora
/camera/depth/image_rect/theora/parameter_descriptions
/camera/depth/image_rect/theora/parameter_updates
/camera/depth/image_rect_raw
/camera/depth/image_rect_raw/compressed
```

```
/camera/depth/image_rect_raw/compressed
/camera/depth/image_rect_raw/compressed/parameter_descriptions
/camera/depth/image_rect_raw/compressed/parameter_updates
/camera/depth/image_rect_raw/compressedDepth
/camera/depth/image_rect_raw/compressedDepth/parameter_descriptions
/camera/depth/image_rect_raw/compressedDepth/parameter_updates
/camera/depth/image_rect_raw/theora
/camera/depth/image_rect_raw/theora/parameter_descriptions
/camera/depth/image_rect_raw/theora/parameter_updates
/camera/depth/points
/camera/depth_rectify_depth/parameter_descriptions
/camera/depth_rectify_depth/parameter_updates
/camera/depth_registered/camera_info
/camera/depth_registered/image_raw
/camera/depth_registered/image_raw/compressed
/camera/depth_registered/image_raw/compressed/parameter_descriptions
/camera/depth_registered/image_raw/compressed/parameter_updates
/camera/depth_registered/image_raw/compressedDepth
/camera/depth_registered/image_raw/compressedDepth/parameter_descriptions
/camera/depth_registered/image_raw/compressedDepth/parameter_updates
/camera/depth_registered/image_raw/theora
/camera/depth_registered/image_raw/theora/parameter_descriptions
/camera/depth_registered/image_raw/theora/parameter_updates
/camera/depth_registered/points
/camera/depth_registered/sw_registered/camera_info
/camera/depth_registered/sw_registered/image_rect
/camera/depth_registered/sw_registered/image_rect/compressed
/camera/depth_registered/sw_registered/image_rect/compressed/parameter_descriptions
/camera/depth_registered/sw_registered/image_rect/compressed/parameter_updates
/camera/depth_registered/sw_registered/image_rect/compressedDepth
/camera/depth_registered/sw_registered/image_rect/compressedDepth/parameter_descriptions
/camera/depth_registered/sw_registered/image_rect/compressedDepth/parameter_updates
/camera/depth_registered/sw_registered/image_rect/theora
/camera/depth_registered/sw_registered/image_rect/theora/parameter_descriptions
/camera/depth_registered/sw_registered/image_rect/theora/parameter_updates
/camera/depth_registered/sw_registered/image_rect_raw
/camera/depth_registered/sw_registered/image_rect_raw/compressed
```