industriales
etsii

Escuela Técnica Superior de Ingeniería Industrial

Universidad Politécnica de Cartagena

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industrial

Optimización del sistema de navegación de un robot móvil asistente para entornos domésticos

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor: **Raquel Tortosa López**
Director: Nieves Pavón Pulido
Codirector: Francisco José Ortiz Zaragoza

Cartagena, 9 de septiembre de 2022

RESUMEN

En este proyecto se presenta el desarrollo de un software basado en el sistema operativo de ROS, que permite la navegación autónoma de un robot móvil en un entorno conocido. Para llevarlo a cabo se ha hecho uso del mapeado láser del entorno, así como la aplicación de diversos paquetes de ROS y la creación de nodos de navegación personalizados.

CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

1.1 Objetivos

El objetivo principal de este proyecto es la navegación autónoma por un entorno doméstico de un robot móvil para mejorar la calidad de vida de personas mayores. El software utilizado será ROS, en él incorporaremos el stack de navegación para la panificación de rutas, así como la evitación de obstáculos móviles.

El sistema se implantará en el robot Turtlebot, aunque antes de la puesta en marcha del robot real, se probarán los comportamientos y el planificador de rutas en el simulador Gazebo.

Se realizará el mapeado del entorno mediante un láser y se crearán nodos de navegación para una navegación básica entre salas en donde el robot planifique una ruta óptima y pueda atravesar puertas o evitar objetos con facilidad.

Los nodos consistirán en una bienvenida a casa tipo guía de museo y la navegación del entorno hasta las estancias solicitadas por el usuario.

2 ESTADO DEL ARTE

2.1 Robótica móvil

Los robots móviles se pueden definir como sistemas robóticos capaces de desplazarse de forma autónoma. Estos desplazamientos son posibles gracias a sensores que les permiten saber su posición relativa, además de componentes como patas, ruedas u orugas, los cuales posibilitan su movimiento por el espacio.

La robótica móvil es un sector en constante crecimiento debido al aumento de recursos que se producen año tras año. Aunque nada tengan que ver los primeros robots móviles con el robot utilizado para este proyecto, todos tienen la misma finalidad: son máquinas capaces de realizar diversas labores que el ser humano no puede llevar a cabo o, simplemente, que mejoran su calidad de vida.

Ya en la antigua Grecia se hablaban de sistemas mecánicos, pero no fue hasta principios del siglo XX cuando surgió el término robot, el cual proviene de la palabra checa *robota*, que significa servidumbre o trabajo forzado. Sin embargo, se empezó a hablar de la robótica móvil como ciencia a mediados de los cuarenta.

A finales de la década de los cuarenta W. Walter Grey construyó un robot, que acabó conociéndose popularmente como “robot tortuga” (ver figura 2.1). Estas máquinas realimente sólo tenían tres comportamientos: evitar objetos de forma torpe, volver a su lugar de reposo y, el más notorio, recargar sus baterías antes de que se agotasen. Esta última característica es la que los hacía totalmente autónomos.

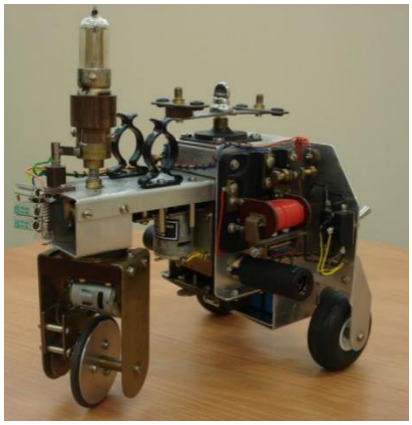

Figura 2.1. Robot tortuga de W. Walter Grey.

A partir de la década de los 70 la robótica móvil creció de forma exponencial y en esa misma década se produjeron multitud de avances, de los cuales tenemos que destacar el primer robot dotado con inteligencia artificial: Shakey (ver figura 2.2). Este robot estaba programado para evitar obstáculos y navegar por rampas y puertas, además, su sistema de localización era bastante preciso.

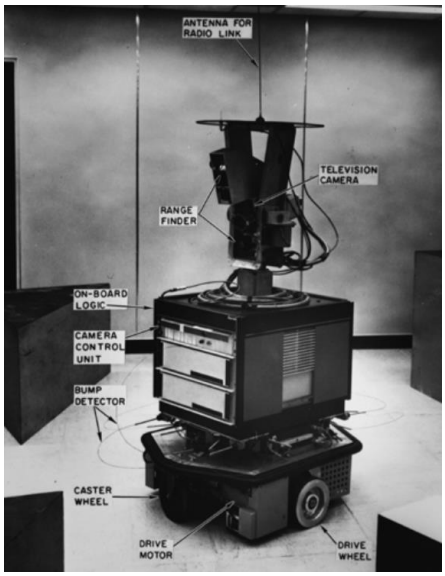

Figura 2.2. Robot Shakey.

También fue en la década de los 70 cuando se empezaron a desarrollar los robots con forma humanoide, así como exoesqueletos, con la finalidad de ayudar a personas discapacitadas. Fue ya en los 90 cuando la empresa HONDA fabricó el modelo P3 (ver figura 2.3), el primer robot con forma humanoide capaz de emular movimientos humanos, totalmente autónomo.

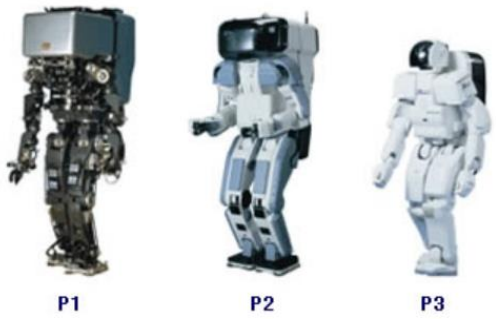

Figura 2.3. De izquierda a derecha, modelo P1, P2 y P3 de la marca HONDA.

Ya en 2002 empiezan a fabricarse robots domóticos como las “Roombas” (ver figura 2.4), un robot móvil completamente autónomo que aspira el suelo de la casa y tiene la capacidad de mapear un entorno y poder navegar por él.

Figura 2.4. Roomba de primera generación.

Cabe destacar todos los avances realizados por la NASA en sus misiones de exploración de Marte: desde el Mars Pathfinder (ver figura 2.5) capturando imágenes, hasta el Perseverance (ver figura 2.5), el último enviado en 2020 que hasta toma muestras y las analiza, pasando por el Opportunity o el Curiosity.

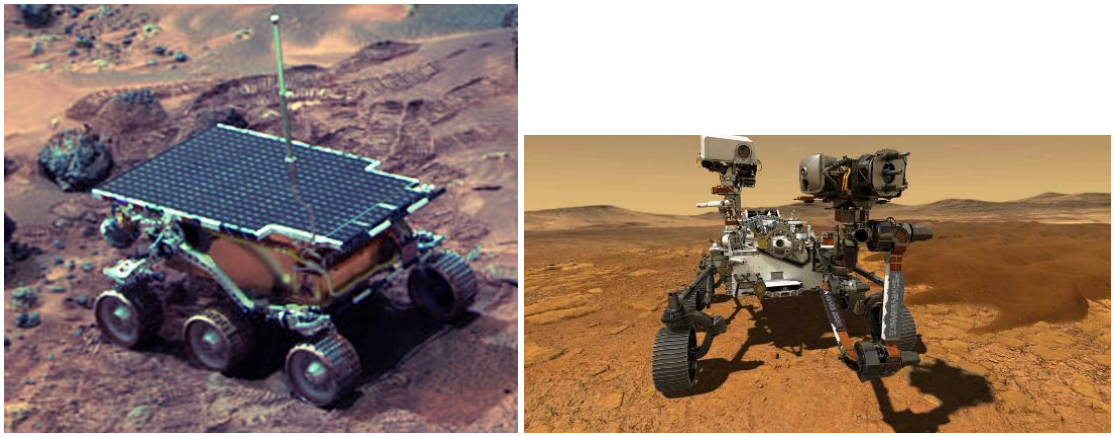

Figura 2.5. Robot Mars Pathfinder y Perseverance de la NASA.

En el año 2010 Melonee Wise y Tully Foote crearon en Willlow Garage el robot utilizado para este proyecto: el Turtlebot. Este robot está diseñado para que sea fácil de comprar y ensamblar, además, sus piezas están hechas con materiales estándar. La gran característica es un software de código abierto. Gracias a la ejecución de algoritmos estándar de localización y mapeo simultáneos (SLAM) sus usuarios pueden mapear y navegar por un entorno evitando objetos a tiempo real y crear sus propias aplicaciones. Se puede controlar de forma remota mediante un ordenador o teléfono y puede seguir las piernas de una persona que anda por una habitación.

El primer Turtlebot fue el Turtlebot1, el más simple de todos. Más adelante se lanzó el Turtlebot2 (ver figura 2.6), utilizado en este proyecto, que incluye una base de carga rápida en la que puede acoplarse de forma autónoma. Después del Turtlebot2 surgieron 2 generaciones más: el Turtlebot3 (ver figura 2.6), diseñado para reducir drásticamente el tamaño de la plataforma, y, el Turtlebot4 (ver figura 2.6), que ofrece una mejor potencia informática y mejores sensores.

Figura 2.6. De izquierda a derecha: Turtlebot2, Turtlebot3 Burguer y Turtlebot4 Estándar.

2.2 Robótica asistencial

Como se ha comentado en el punto anterior, la robótica móvil está en continuo desarrollo, por lo que cada vez tenemos más aplicaciones de esta tecnología; hemos visto que sus aplicaciones pueden ir desde la exploración de un planeta desconocido hasta la rehabilitación de personas discapacitadas. Sin embargo, todas las aplicaciones tienen un común denominador: trabajar en entornos peligrosos para el ser humano o, simplemente, para mejorar la calidad de vida de este. Este proyecto se centra en la aplicación última, concretamente en la robótica asistencial.

La robótica asistencial está destinada a ayudar a centros hospitalarios y residencias, así como a personas con movilidad reducida en el hogar o para trabajos del hogar, entre otros. Sus aplicaciones son muy diversas, podemos encontrar tareas de limpieza, de rehabilitación, de alimentación, de asistencia personal en general, de entretenimiento o educativas, entre otras muchas más.

Como ejemplo, a continuación, se muestran distintos robots con diferentes aplicaciones en la robótica asistencial:

- De aplicación terapéutica podemos destacar el robot Paro (ver figura 2.7) de forma animaloide, el cual se asemeja a una foca bebé. Tiene la misma función que la terapia asistida por animales.
- Con una aplicación más comercial podemos encontrarnos al robot Nao (ver figura 2.7), diseñado para ejercer de asistente en empresas y centros sanitarios para recibir, informar o entretener a los visitantes.
- My Spoon (ver figura 2.7) ofrece una solución a personas que no son capaces de comer por ellas solas, ya que tienen el movimiento de brazos o manos limitado.
- El robot de limpieza por antonomasia es la “Roomba” (ver figura 2.4). Debido a su precio asequible y la comodidad que ofrece al aspirar la casa de forma autónoma, se encuentra presente en muchos hogares.
- Pepper (ver figura 2.7) es el primer robot humanoide con la capacidad de entender y reconocer caras y emociones. Ha sido creado únicamente para socializar.

Figura 2.7. De izquierda a derecha y de arriba abajo: Robot Paro, Robot Nao, My Spoon y Pepper.

Junto con el auge de esta nueva tecnología, también surgen diversas cuestiones éticas que engloban una problemática moral y las consecuencias socio-políticas que puede traer consigo las distintas innovaciones en esta rama. De esta forma, surge la roboética, que se ocupa del diseño y uso de esta tecnólogia.

La roboética presente en la robótica asistencial se encarga de que exista una robótica inclusiva que al mismo tiempo contribuya al desarrollo humano; la robótica tiene que estar adaptada para las distintas fases del ser humano, así como su contexo temporal, en donde se tiene que tener en cuenta su diseño y su accesibilidad para todo el que lo necesite.

2.3 Arquitectura y subsistemas robóticos

2.3.1 Introducción

La navegación en un sistema robótico abarca multitud de factores y tareas, por lo que debemos dividirla en subsistemas que lleven a cabo dichas tareas. Las diferentes arquitecturas de navegación surgen de los diferentes subsistemas utilizados y como se relacionan entre sí.

Los subsistemas más básicos son:

- Percepción.
- Navegación.
- Control de movimiento.

La navegación, a su vez, estará dividida en dos tipos:

- Navegación local → Evita el choque con obstáculos
- Navegación global → Crea la trayectoria a partir de la información que dispone de un mapa.

2.3.2 Arquitecturas robóticas

Arquitectura jerárquica

La arquitectura jerárquica se trata de una arquitectura secuencial en lazo cerrado, la cual primero tiene una etapa de medición, y, respecto a esa medición, planifica un comportamiento que más adelante ejecuta (ver figura 2.8).

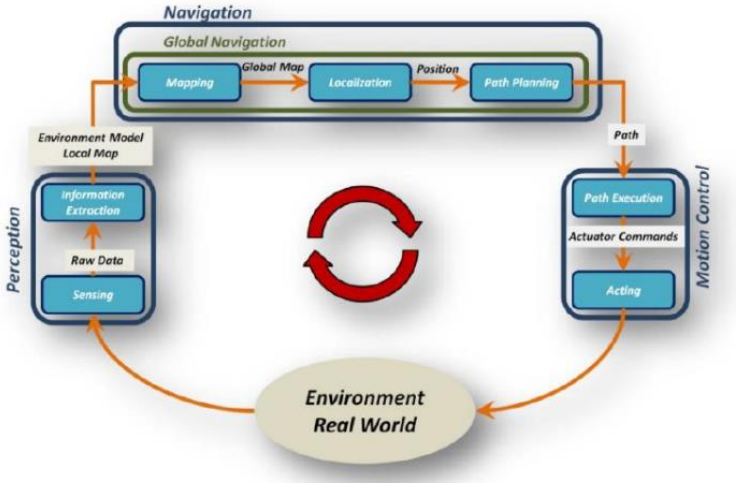

Figura 2.8. Esquema de funcionamiento de la arquitectura jerárquica de un robot móvil. Figura tomada de Melo, J. M. (2012). Implementation of Algorithms for Navigation of an Autonomous Mobile Robot by using software component based frameworks .

Arquitectura reactiva

La diferencia de la arquitectura jerárquica y la reactiva reside en que esta última elimina la planificación, dejando los módulos de control directamente conectados con los sensores y actuadores. El comportamiento global se determinará por sus conductas más que por un razonamiento deliberativo (ver figura 2.9).

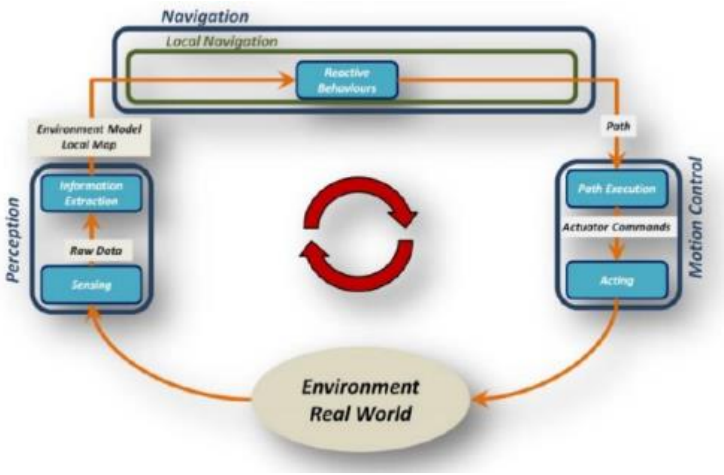

Figura 2.9. Esquema de funcionamiento de la arquitectura reactiva de un robot móvil. Figura tomada de Melo, J. M. (2012). Implementation of Algorithms for Navigation of an Autonomous Mobile Robot by using software component based frameworks .

Arquitectura híbrida

La arquitectura híbrida es la suma de las dos arquitecturas anteriores, de manera que planifica su trayectoria deliberadamente, además de poder reaccionar a estímulos externos (ver figura 2.10). Esta última arquitectura es la utilizada en el Turtlebot, por ser la óptima en este tipo de aplicaciones.

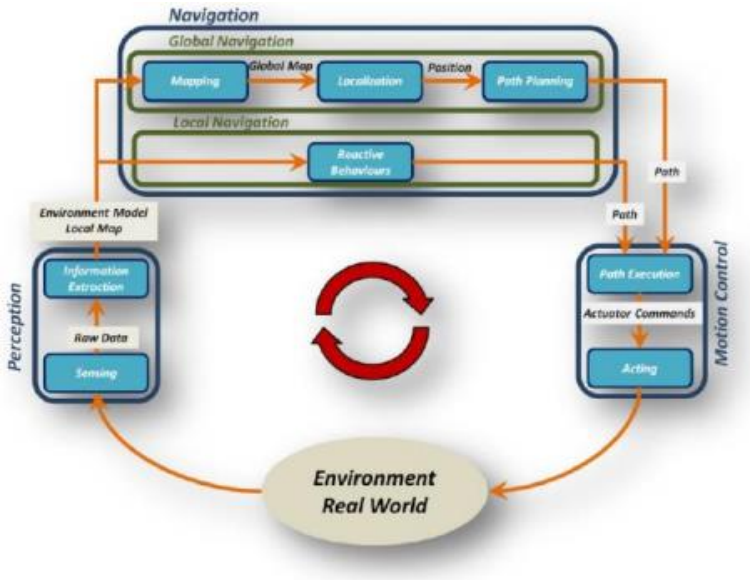

Figura 2.10. Esquema de funcionamiento de la arquitectura híbrida de un robot móvil. Figura tomada de Melo, J. M. (2012). Implementation of Algorithms for Navigation of an Autonomous Mobile Robot by using software component based frameworks .

2.3.3 Subsistemas robóticos

En este apartado nos vamos a centrar en los subsistemas ya nombrados: percepción y navegación.

Percepción

La percepción es el subsistema encargado de recopilar información del exterior para poder actuar en consecuencia, o, simplemente, crear un mapa del entorno. Destacamos las balizas y los sensores.

Las balizas son elementos del entorno fácilmente identificables que se pueden dividir en:

- Artificiales: Elementos ajenos al entorno introducidos especialmente para ser reconocidos por el robot (ver figura 2.11).
- Naturales: Elementos del entorno que se utilizan como características que pueden ser reconocidas por el robot, por ejemplo, un frigorífico, un horno o una cama (ver figura 2.11).

Figura 2.11. Ejemlos de balizas. A la izquierda artificiales y a la derecha naturales.

Los sensores se dividen en:

- Propioceptivos: Toman medidas internas del robot y generalmente son estimadores de la posición. A su vez se pueden dividir en absolutos (GPS, brújula, etc) y relativos (encoders, giróscopo, etc)
- Exteroceptivos: Son exteriores al robot y permiten obtener información del entorno. Pueden ser desde cámaras de visión artificial hasta medidores de distancia de ultrasonidos.

Debemos valernos de ambos tipos de sensores para la localización y la construcción de mapas, ya que los propioceptivos se relacionan con el cálculo de la odometría y los exteroceptivos se encargan de la extracción de información del entorno.

Los sensores exteroceptivos son los que captan las balizas y, además, son cruciales en la construcción de la representación del entorno, en el ámbito de la planificación de caminos y en la evitación de obstáculos.

En este proyecto utilizaremos un sensor exteroceptivo de barrido láser 2D, en concreto un LIDAR. Un LIDAR (Light Detection and Ranging o Laser Imaging Detection and Ranging)

determina la distancia a un objeto o superficie usando un haz láser pulsado. La distancia se mide a partir del tiempo de retraso entre la emisión del pulso y su detección mediante la señal reflejada (ver figura 2.12). El haz del láser ocupa un cierto ángulo, que, en nuestro caso, serán 180°

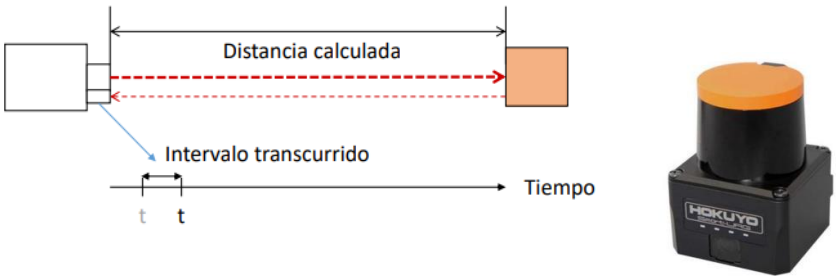

Figura 2.12. Principio de funcionamiento de un LIDAR y sensor exteroceptivo de barrido láser.

Navegación

Como ya se ha comentado anteriormente, la navegación se divide en navegación global y local. La local hace uso de los sensores exteroceptivos para evitar la colisión y la global se encarga de la planificación de la trayectoria hasta el objetivo deseado. Esta última se sustenta en tres componentes (ver figura 2.13):

- Localización: Localiza al robot en el entorno gracias a los sensores.
- Mapeado: Construcción del mapa mediante los sensores exteroceptivos.
- Planificación: Encuentra el camino necesario para navegar a un punto deseado.

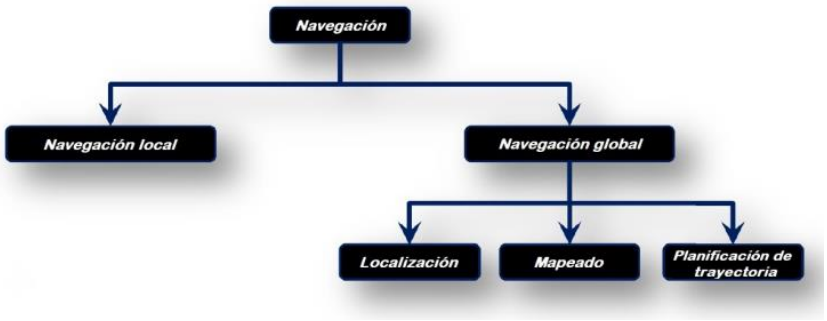

Figura 2.13. Esquema navegación global y local.

De los múltiples algoritmos de navegación existentes, procedemos a explicar dos: el Dynamic Window Approach (DWA) y el enfoque de localización de Monte Carlo adaptativo (AMCL), puesto que son los que se utilizan en este proyecto.

Navegación local: Dynamic Window Approach

La navegación local se encarga de la parte reactiva, ya mencionada. Se han desarrollado multitud de algoritmos que implementan el funcionamiento del comportamiento reactivo, pero

en este apartado nos vamos a centrar en el DWA (Dynamic Window Approach) por ser el que utilizamos en este proyecto.

El DWA trabaja directamente sobre el campo de velocidades del robot, este incorpora la dinámica del robot con la finalidad de reducir las velocidades a seleccionar, de esta forma se escogerán las velocidades más seguras respecto a los objetos cercanos. El primer paso del algoritmo consiste en buscar un rango de velocidades seguras que permitan parar al robot antes del objeto más cercano. En el segundo paso, se optimiza la velocidad mediante una función y permite descartar candidatas.

Navegación global: Localización de Monte Carlo adaptativo

Respecto a la localización ya mencionada, va a ser imposible conocer la ubicación exacta del robot debido a los diversos errores que pueden cometer los sensores. De esta forma, se asume una ubicación mediante un método probabilístico, que no nos dará la posición real, pero sí será bastante cercana a esta.

La hipótesis más utilizada es la hipótesis múltiple; esta propone múltiples localizaciones posibles que serán respaldadas por los sensores propioceptivos y exteroceptivos. El algoritmo más utilizado es la implementación del enfoque de localización de Monte Carlo adaptativo (AMCL), que utiliza un filtro de partículas para rastrear la pose de un robot en un mapa conocido. Este algoritmo crea un número de partículas distribuidas por el espacio (cuantas más partículas, más recurso computacional) y, comparando con las mediciones de los sensores, dará más peso a las partículas que más se asemejen a las mediciones, además, conforme nos vamos moviendo las partículas se mueven simultáneamente, de esta forma, cada vez se irán acercando más a nuestra posición real (ver figura 2.14).

A continuación, un ejemplo del funcionamiento del AMCL:

Figura 2.14. Ejemplo de funcionamiento del AMCL.

3 TURTLEBOT

El hardware utilizado para este proyecto es el Turtlebot2 (ver figura 3.1). Es la segunda generación del Turtlebot y se caracteriza por ser un robot de bajo costo y código abierto.

Figura 3.1. Turtlebot2.

En nuestro proyecto, no contaremos con las varillas, los platos o la cámara. Los materiales que componen nuestro robot son:

- Base móvil Kobuki
- Batería
- Sensor láser Hokuyo
- Ordenador NUC

3.1 Base Kobuki

La base Kobuki (ver figura 3.2) está diseñada para navegar de forma autónoma, contando con una odometría y un giroscopio para las tareas de navegación. La alimentación de todo el sistema se hará desde la base. Sus especificaciones funcionales son:

Tabla 3.1. Especificaciones funcionales de la base Kobuki.

Especificaciones	Valor
Velocidad máxima de traslación	70 cm/s
Velocidad máxima de rotación	180 deg/s
Carga útil	5 kg
Cliff	5 cm
Umbral de escalón	12 mm o menos
Escalada de alfombra	12 mm o menos
Tiempo de operacion	3/7 horas
Tiempo de carga	1.5/2.6 horas
Área libre en la estación	10 m^2

Las especificaciones hardware son:

Tabla 3.2. Especificaciones hardware de la base Kobuki.

Especificaciones	Valor
Conexión a PC	Vía USB o vía RX/TX en puerto serie
Detección de sobrecarga del motor	I < 3 A
Odometría	52 ticks/enc rev, 2578.33 ticks/wheel rev, 11.7 ticks/mm
Giroscopio	3 ejes
Bumpers	Izquierda, derecha y centro
Conectores de alimentación	5V/1A, 12V/1.5A,12V/5A
Pines de expansión	3.3V/1A, 5V/1A, 4 x analog in, 4 x digital in, 4 x digital out
Audio	Sonidos de beep
Leds programables	2
Led de estado	1
Botones	3
Batería	4400 mAh (4S2P -large)
Velocidad de datos en sensores	50 Hz
Fuente de alimentación	Input: 100-240V AC, 50/60Hz, 1.5A máx; Output: 19V DC, 3.16A
Receptores infrarrojos	Izquierda, centro y derecha
Diámetro	351.5 mm
Altura	124.8 mm
Peso	2.35 kg

Dispone de dos motores DC alimentados a 12 V controlados por señal PWM. Sus características son:

Tabla 3.3. Especificaciones del motor de la base Kobuki.

Tipo	Motor de DC Brushed
Fabricante	Standard Motor
Número de serie	RP385-ST-2060
Tensión nominal	12 V
Carga	5 mN·m
Corriente sin carga	210 mA
Velocidad sin carga	9960 rpm ± 15%
Corriente con carga nominal	750 mA
Velocidad sin carga nominal	8800 rpm ± 15%
Resistencia de armadura	1.5506 Ω a 25 ° C
Inductancia de la armadura	1.51 mH
Par constante (k_t)	10.913 mN·m/A
Velocidad constante (K_v)	830 rpm/V
Corriente de bloqueo	6.1 A

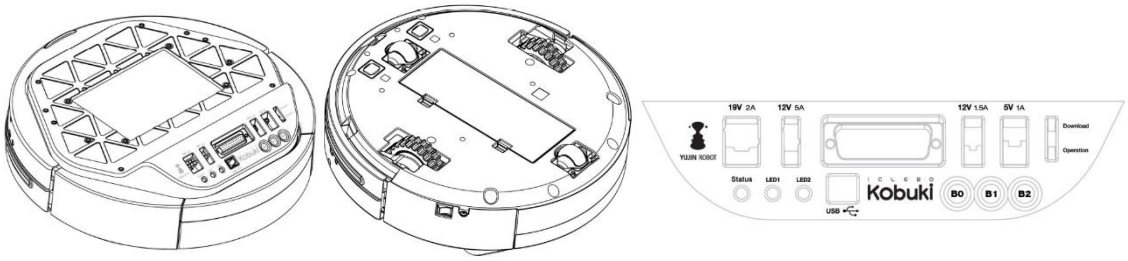

Figura 3.2. De izquierda a derecha: Vista superior, vista inferior y cuadro de control de la base Kobuki.

3.2 Batería

Disponemos de dos baterías de ion litio de 4400 mA. Sus principales características son:

Tabla 3.4. Especificaciones de la batería.

Fabricante	Yujin Robot
Referencia	EL-BAT-4S2P-4400
Voltaje	14.8 V
Capacidad	4400 mAh

3.3 Sensor láser Hokuyo

El sensor láser que se va a utilizar es el Hokuyo UST-10LX (ver figura 3.3). Es un sensor de dimensiones reducidas y buena precisión. Utiliza una interfaz ethernet en su conexión con el NUC, por este motivo siempre hay que inicializarlo desde el Turtlebot. Además, hay que destacar su bajo consumo de energía.

Las especificaciones del láser son:

Tabla 3.5. Especificaciones del láser Hokuyo.

Nombre	Scanning Laser Range Finder
Modelo	UST-10LX
Voltaje de amilentación	12 VDC/24VDC
Corriente de alimentación	150 mA
Precisión	±40 mm
Ángulo de escaneo	270°
Velocidad de escaneo	25 ms
Resolución angular	0.25°
Interfaz	Ethernet

Figura 3.3. Láser Hokuyo.

3.4 Ordenador NUC

El ordenador incorporado en el robot es un NUC (ver figura 3.4). Una de sus características es su reducido tamaño, ya que puede ser montado perfectamente encima de la base Kobuki. Además, su memoria, procesador y almacenamiento son totalmente configurables. Sus especificaciones son:

Tabla 3.6. Especificaciones NUC.

Procesador	Intel Core i7-5557U
Memoria	8 GB DDR4
Almacenamiento	120 GB SSD M.2
USB 2.0	2
USB 3.0	2
Conectividad	Intel Wireless-AC 7265 M.2/Bluetooth 4/Ethernet
Gráficos	HD Iris 6100
Salida de audio y vídeo	Mini HDMI 1.4 A/Mini DisplayPort 1.2
Alimentación	12 V 5 A

Figura 3.4. Ordenador NUC.

4 SOFTWARE

4.1 Ubuntu

El sistema operativo Ubuntu es una distribución de código abierto basada en Debian, otro sistema operativo, cuyo punto en común es Linux. Este último es una familia de sistemas operativos basados en el kernel de Linux, el núcleo del sistema operativo. En resumen, Ubuntu es una distribución de Linux.

A lo largo de los años ha ganado mayor popularidad debido a una serie de características que se exponen a continuación:

- Es un software gratuito y de código abierto, por lo que puede llegar a más gente.
- Tiene una gran facilidad de uso ya que su interfaz es sencilla e intuitiva.
- Mayor seguridad puesto que está en revisión constante y admite una serie de prácticas de seguridad que pueden ser configuradas.
- Tiene una estricta política de privacidad, lo que supone mayor privacidad que otros sistemas.
- Se pueden instalar multitud de aplicaciones, las cuales sólo funcionan en este sistema operativo.

Cada seis meses se lanza una nueva versión de Ubuntu. En este proyecto se han tenido que utilizar dos versiones debido a la compatibilidad con las diferentes versiones de ROS utilizadas. Con ROS Melodic se ha utilizado Ubuntu 18.04.5 y con ROS Kintetic, la versión 16.04 LTS.

4.2 ROS

4.2.1 Introducción

ROS (Robot Operating Sistem) es un framework que permite desarrollar software para robots. Algunas de las características de ROS son:

- Control de dispositivos de bajo nivel.
- Abstracción de hardware.
- Sistema de comunicación de procesos basado en el paso de mensajes.
- Sistema de administración de paquetes.
- Diseño de software distribuido basado en el concepto de paquete software.
- Implementación de utilidades que se utilizan comúnmente en la comunidad para robots.

Conceptualmente, tenemos tres niveles: nivel del sistema de ficheros, nivel del grafo de computación y nivel de la comunidad.

Nivel de sistema de ficheros

Los paquetes son la unidad fundamental para organizar el software en ROS, contienen nodos, bibliotecas, conjuntos de datos, archivos de configuración o cualquier otra cosa que esté organizada de manera útil. Los paquetes tienen un fichero de tipo manifiesto (package.xml) que proporciona metadatos sobre un paquete, esto es; nombre, versión, descripción, dependencias, etc.

Los repositorios son una colección de paquetes que tienen un sistema de versiones común (VCS). Estos paquetes se pueden lanzar juntos usando unas herramientas específicas.

Se definen también tipos de mensajes (msg), las descripciones de los mensajes definen las estructuras de datos de los mensajes enviados en ROS. Lo mismo pasa con los servicios (srv), las descripciones de servicios definen las estructuras de datos de solicitud y respuesta para los servicios de ROS.

Nivel de grafo de computación

El grafo de computación es una red de comunicación peer-to-peer (punto a punto) que procesan datos conjuntamente. Los elementos básicos que forman la red de computación son:

- **Nodos:** Los nodos son procesos que realizan computación. ROS está diseñado para que el comportamiento del robot pueda ser definido mediante un conjunto de nodos actuando a la vez. Un nodo ROS se escribe con el uso de una biblioteca de cliente ROS, como *roscpp* (para C++) o *rospy* (para Python).
- **Master:** Es el que se encarga de la comunicación entre nodos. Si no se ejecuta, los nodos no podrán intercambiar mensajes o llamar a servicios.
- **Parameter Server:** Forma parte del Master, permite que los datos se almacenen por clave en una ubicación central.
- **Mensajes**: Los nodos se comunican entre sí mediante mensajes. Son datos con una estructura determinada, se admiten tipos primitivos estándar o incluso struct y matrices anidadas, al igual que en C.
- **Topics:** Los topics son los canales a través de los cuales se comunican los nodos, por lo que su contenido son los mensajes. Un nodo envía un mensaje publicándolo en un topic determinado y cualquier nodo interesado en ese tipo de mensaje estará suscrito a dicho topic (ver figura 4.1). No hay un número máximo de publicadores o suscriptores a un mismo topic, y estos desconocen la existencia de los demás.
- **Servicios:** A menudo, la comunicación unidireccional no es apropiada, por este motivo están los servicios, que es un sistema de solicitud/respuesta. Un nodo ofrece un servicio y un cliente utiliza ese servicio enviando una solicitud y esperando una respuesta.
- **Bags:** Son un formato para guardar y reproducir mensajes ROS, como, por ejemplo, el valor de un sensor.

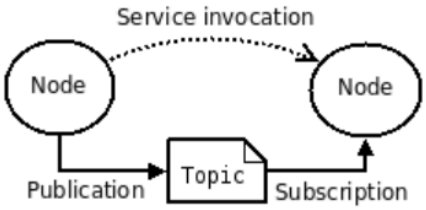

Figura 4.1. Esquema de funcionamiento publicación-suscripción.

Nivel de comunidad

Este nivel permite el intercambio de software y conocimiento entre comunidades. Los recursos son:

- **Distribuciones:** Desempeñan un papel similar al de las distribuciones de Linux; facilitan la instalación de una colección de software y también mantienen versiones consistentes en un conjunto de software.
- **Repositorios:** ROS se basa en una red federada de repositorios de código, donde diferentes instituciones pueden desarrollar y lanzar sus propios componentes de software de robot.
- **ROS Wiki:** Es el foro principal para compartir información sobre ROS. Cualquiera puede entrar y contribuir con su propia documentación,
  proporcionar correcciones y actualizaciones, escribir tutoriales, etc.
- **Mailing Lists:** Es el principal canal de comunicación, con foros para realizar consultas sobre el software de ROS.
- **ROS Answers:** Un sitio de preguntas y respuestas para responder cuestiones relacionadas con ROS.
- **Blog:** El blog de ros.org proporciona actualizaciones periódicas, incluidas fotos y vídeos.

4.2.2 Comandos básicos

El paquete rosbash contiene los comandos básicos similares a ciertos comandos de Linux, que facilitan la navegación a través del sistema de ficheros de ROS. Estos son:

- **roscd:** Permite cambiar entre directorios usando el nombre un paquete, stack o una localización especial.
- **roscp:** Equivalente a cp, permite copiar un fichero que hay en un paquete en la localización que especifiquemos como segundo argumento.
- **rosrun:** Permite ejecutar un fichero ejecutable de un paquete como un nodo.

Otros comandos esenciales de ROS:

- **rosnode:** Permite visualizar información de depuración sobre nodos ROS, incluyendo publicaciones, suscripciones y conexiones del nodo. Los modificadores disponibles son: *info, kill, list, machine, ping* y *cleanup*.

- **rostopic:** Permite visualizar información de depuración sobre topics ROS, incluyendo publicadores, suscriptores, tasa de publicación y mensajes ROS. Los modificadores disponibles son: *bw, delay, echo, find, hz, info, list, pub* y *type*.
- **rosservice:** Permite listar y consultar los servicios ROS. Contiene una biblioteca Python para recuperar información sobre Services y llamarlos de forma dinámica. Los modificadores disponibles son: *args, call, find, list, info, node, type* y *uri*.
- **rosparam:** Permite obtener y actualizar parámetros de ROS almacenados en el Parameter Server usando ficheros codificados como YAML. Los modificadores disponibles son: *list, get, set, delete, dump* y *load*.
- **roslaunch:** Es una herramienta que permite lanzar varios nodos ROS, tanto localmente, como remotamente (mediante SSH), así como configurar los parámetros del Parameter Server. Incluye opciones para hacer automáticamente "respawn" sobre procesos que ya han muerto. Toma uno o más ficheros de configuración XML (con extensión .launch), que especifican los parámetros a configurar y los nodos a lanzar, así como las máquinas donde deberían ejecutarse. Los archivos .launch podrían considerarse similares a los archivos por lotes.
- **roscore:** Es una especialización de la herramienta roslaunch que permite "levantar" el sistema ROS. Este comando sólo se ejecuta una vez en una determinada máquina y ejecuta el maestro ROS y el servidor de parámetros. Si se ejecuta roslaunch, sin ejecutar roscore, roslaunch se encarga de ejecutar roscore automáticamente (a menos que se use el argumento --wait con el comando roslaunch).

4.2.3 Descarga y creación de paquetes

Para instalar un paquete usamos el comando *apt*, la instalación se realizará en la carpeta de ROS */opt/ros* y no en la carpeta de trabajo, por tanto, podemos ejecutar *apt* desde cualquier carpeta. Un ejemplo de aplicación del comando:

Sudo apt install ros-melodic-navigation

Si lo que queremos es descargar un paquete, únicamente tendremos que descargarlo y mover el archivo de descargas a nuestra carpeta personal.

También podemos crear nuestros propios paquetes, codificando programas en C++ o Python, usando las librerías *roscpp* o *rospy*, respectivamente. Para ello usamos el comando *catkin_create_pkg*.

Por ejemplo, creamos un nuevo paquete llamado turtlebot_navegacion considerando que necesitamos incluir las dependencias *roscpp, actionlib, move_base_msgs, sensor_msgs* y *tf*:

Catkin_create_pkg turtlebot_navegacion roscpp move_base_msgs sensor_msgs tf

Además de crearse las carpetas *src* e *include* dentro del paquete turtlebot_navegacion, también se han creado los archivos CmakeList.txt y package.xml.

Cada paquete usado como dependencia tiene una entrada de tipo *build_depend*, otra de tipo *build_export_depend* y otra de tipo *exec_depend* (ver figura 4.2). Si se desea añadir una

nueva dependencia, habría que generar esas tres entradas del nuevo paquete que se deseara incluir. Para eliminar o modificar una dependencia, simplemente se eliminan o modifican las entradas correspondientes.

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>move_base_msgs</build_depend>
<build_depend>roscpp</build_depend>
<build_depend>sensor_msgs</build_depend>
<build_depend>tf</build_depend>
<build_export_depend>move_base_msgs</build_export_depend>
<build_export_depend>roscpp</build_export_depend>
<build_export_depend>sensor_msgs</build_export_depend>
<build_export_depend>tf</build_export_depend>
<exec_depend>move_base_msgs</exec_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>sensor_msgs</exec_depend>
<exec_depend>tf</exec_depend>
```

Figura 4.2. Ejemplo de estructura del archivo package.xml perteneciente a turtlebot_navegacion.

Una vez quitemos los comentarios de *add_executable* y *target_link_libraries* en el archivo CMakeList.txt, tendremos un archivo que permitirá especificar las dependencias de otros paquetes, definir qué archivo con código fuente se utiliza para generar el archivo con código objeto y todos los archivos de código objeto con las bibliotecas necesarias para generar el ejecutable (ver figura 4.3).

```
cmake_minimum_required(VERSION 3.0.2)
project(turtlebot_navegacion)

## Compile as C++11, supported in ROS Kinetic and newer
# add_compile_options(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  move_base_msgs
  roscpp
  sensor_msgs
  tf
)
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES turtlebot_navegacion
#  CATKIN_DEPENDS move_base_msgs roscpp sensor_msgs tf
#  DEPENDS system_lib
)

include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)

add_executable(simple_goals_node src/simple_goals_node.cpp)

target_link_libraries(simple_goals_node
   ${catkin_LIBRARIES}
 )
```

Figura 4.3. Ejemplo del archivo CMakeList.txt del paquete turtlebot_navegación donde se encuentra el nodo simple_goals_node.

4.3 WSL 2

El Subsistema de Windows para Linux (WSL) permite ejecutar un entorno de GNU/Linux directamente en Windows. Esto incluye la mayoría de los comandos, utilidades y aplicaciones sin una configuración de arranque dual.

El WSL 2 es una nueva versión de la arquitectura WSL que permite que ejecute archivos binarios de ELF64 de Linux en Windows. Sus principales objetivos son aumentar el rendimiento del sistema de archivos y agregar compatibilidad completa con las llamadas del sistema**.**

Las distribuciones de Linux individuales se pueden ejecutar con la arquitectura de WSL 1 o WSL 2. Cada distribución se puede actualizar o degradar en cualquier momento, y se puede ejecutar distribuciones de WSL 1 y WSL 2 en paralelo. WSL 2 usa una arquitectura completamente nueva que aprovecha las ventajas de un kernel de Linux real.

4.3.1 Descarga de paquetes

Podemos instalar nuevos paquetes ROS de dos maneras: Usando *apt* o bien copiando el paquete en la carpeta *src* dentro de *catkin_ws* y ejecutando, posteriormente, *catkin_make*.

Los archivos descargados se guardan en la carpeta Descargas de Windows, cuya ruta absoluta, en este caso, es:

C:\Users\raque\Downloads

Cabe destacar que, si usamos una instalación nativa de Ubuntu Linux, directamente descargaríamos en la carpeta personal. Todo el sistema de ficheros de Windows se monta como una unidad externa en el contexto de Linux cuando se usa WSL.

Una vez descomprimidos los archivos, se procede a copiarlos a */home/raquel/catkin_ws/src*, por tanto, nos situamos en la carpeta *src* dentro de *catkin_ws*. Ejecutamos el comando *cp* (que sirve para copiar desde un origen -primer argumento-, hasta un destino específico -segundo argumento-), usando el modificador –r para especificar que la carpeta a copiar debe copiarse recursivamente, es decir, con todas las subcarpetas y archivos que contiene.

cp -r /mnt/c/Users/raque/Downloads/turtlebot2 .

Una vez ejecutado el comando *cp*, donde el origen era */mnt/c/Users/raque/Dowloads/turtlebot2* y el destino era el repositorio actual, representado por ".", comprobamos que la copia se ha realizado correctamente, usando ls. Después, volvemos a *catkin_ws* y ejecutamos *catkin_make*. Si la compilación es correcta, ya podemos utilizar el paquete descargado.

4.3.2 Xming

El WSL nos permite tener un entorno Linux en nuestro ordenador, pero no dispone de X Window, que es el modo gráfico de GNU Linux. Por tanto, necesitamos instalar un servidor X en nuestro ordenador, el cual nos permita tener una interfaz gráfica. Elegimos el Xming por ser gratuito y tener una fácil instalación.

Para que funcione adecuadamente, debemos especificar la IP en *$DISPLAY*. Podemos utilizar el comando:

export DISPLAY=192.168.0.104:0

O bien, escribiendo esa misma línea en el archivo .bashrc. Si cambiamos de dirección IP, tendremos que escribir los cambios.

4.4 Gazebo

Gazebo es una herramienta de simulación de robots independiente de ROS, pero este incorpora una serie de herramientas que garantizan una adecuada compatibilidad entre las simulaciones de Gazebo y el ecosistema ROS a través de la descripción de los robots usando URDF (Unified Robot Description Format).

La descripción visual del robot, así como las propiedades relacionadas con la dinámica y la interacción con el entorno simulado se definen en archivos que utilizan el formato URDF con extensiones *xacro*, que permiten definir propiedades, e incluso incluir operaciones para facilitar la parametrización de los archivos que describen los modelos de robots.

Los elementos visuales, además, están asociados a *plugins* que simulan el comportamiento de los diferentes elementos de un robot, por ejemplo, simulan la cinemática, permitiendo calcular la odometría a partir de las consignas de velocidad publicadas en un topic (normalmente llamado */cmd_vel*), y publican dicha odometría en otro topic (generalmente llamado */odom*). La descripción de estos elementos se realiza en un archivo con extensión .gazebo.

Esta herramienta es muy útil si no disponemos del hardware o, simplemente, si queremos probar nuestro código sin tener que utilizar el robot real. Podemos crear entornos y guardarlos para poder crear un archivo .world (ver figura 4.4), que utilizaremos para poder visualizarlos en un archivo .launch (ver figura 4.5). Además, podemos utilizar todas las herramientas que ofrece ROS como *gmapping* o *navigation*, como si fuera un entorno real.

```xml
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <!-- Own physics settings to speed up simulation -->
    <physics type='ode'>
      <max_step_size>0.01</max_step_size>
      <real_time_factor>1</real_time_factor>
      <real_time_update_rate>100</real_time_update_rate>
      <gravity>0 0 -9.8</gravity>
    </physics>
  </world>
</sdf>
```

Figura 4.4. Ejemplo de archivo con extensión .world. Corresponde a un entorno Gazebo vacío (empty.world).

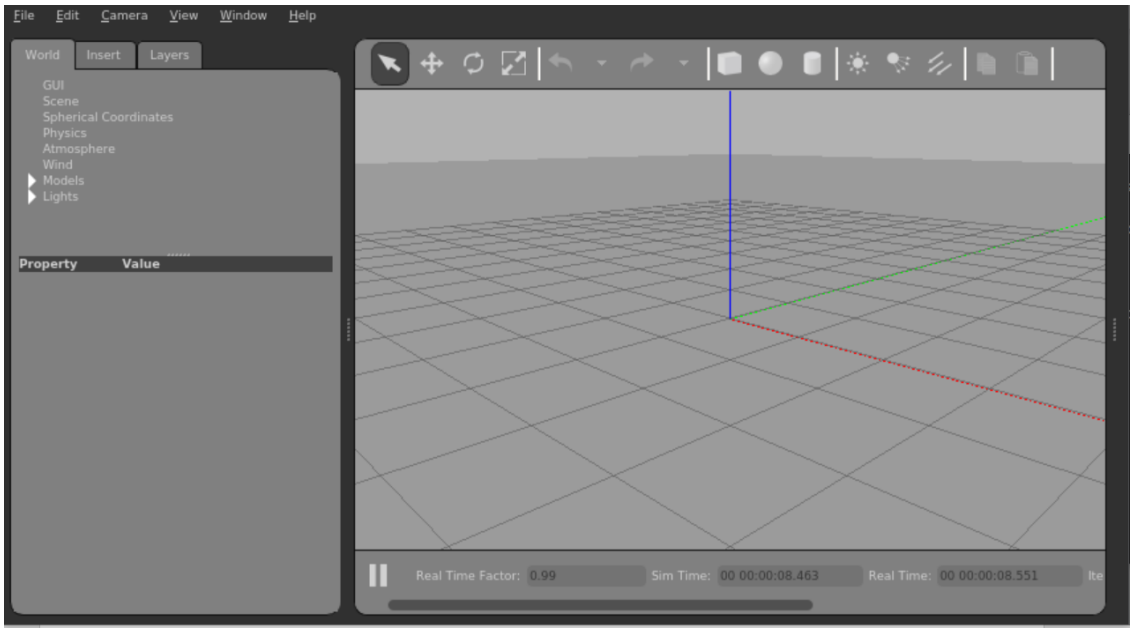

Figura 4.5. Entorno vacío de Gazebo al lanzar un archivo .launch que contiene el archivo de la figura 4.6.

4.5 RViz

RViz es un programa de visualización 3D de los robots que utilizan ROS. Se trata de una manera gráfica de ver los mensajes de los topics que están generando los nodos (ver figura 4.6)
.

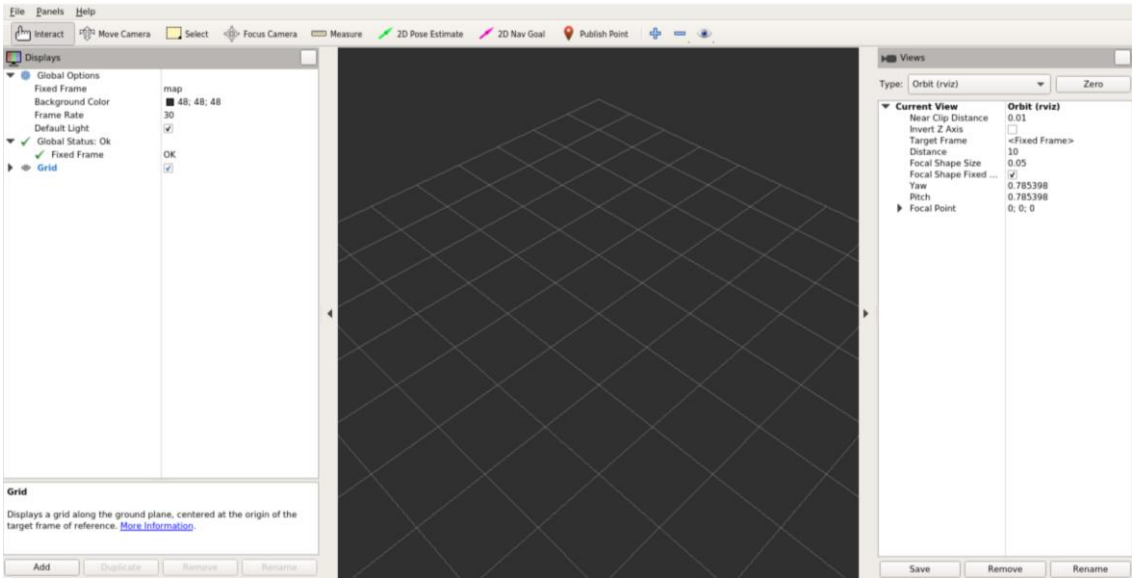

Figura 4.6. Visualización de un entorno RVIZ vacío.

Los topics que se quiere visualizar se añaden seleccionando “add” y se quitan pulsando “remove” (ver figura 4.7).

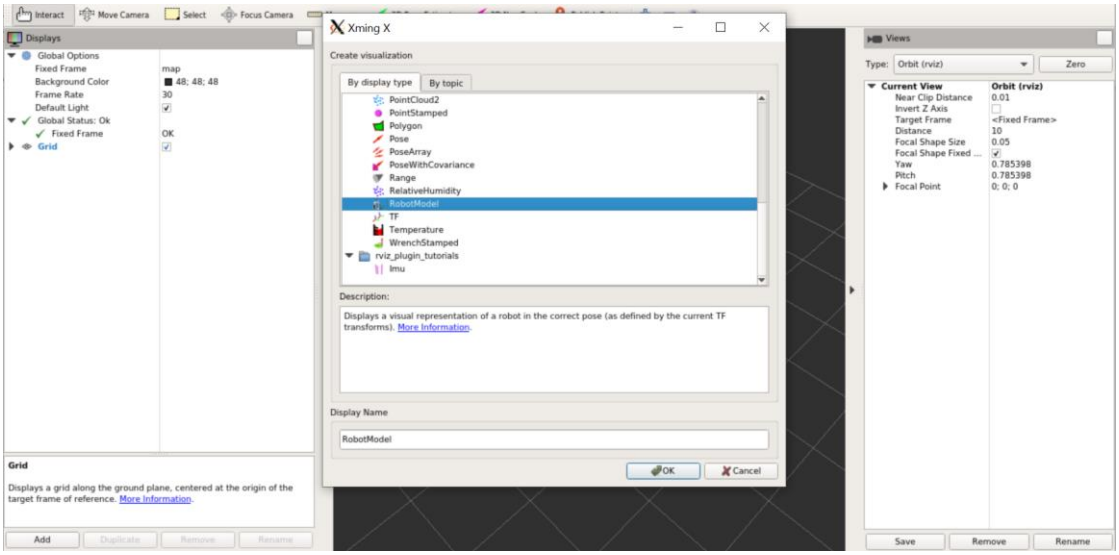

Figura 4.7. Salida en pantalla al pulsar "add" de RViz.

Es una herramienta bastante útil para controlar el estado del robot, ya sea en simulación o en la visualización de forma remota.

RViz tiene una serie de herramientas útiles para la navegación del robot (ver figura 4.8). Por ejemplo, para comprobar que la localización y la planificación del robot están correctamente desarrolladas, utilizamos las herramientas *2D Pose Estimate* o *2D Nav Goal*:

- ***2D Pose Estimate:*** Es útil cuando el robot no aparece bien posicionado y queremos moverlo a su sitio u orientación real. Además, gracias al topic */amcl_pose* podemos saber el punto en el que se encuentra el robot, de esta

forma podemos ir moviendo el robot por distintas estancias e ir guardando puntos.

- ***2D Nav Goal:*** Nos permite decirle al robot a que punto queremos que se dirija sin tener que escribir ningún código. Una vez pulsado, seleccionamos el punto elegido y, sin dejar de pulsar, indicamos la orientación final del robot. Si todo funciona correctamente, el robot se dirigirá al punto escogido utilizando los parámetros declarados para el nodo *move_base*.

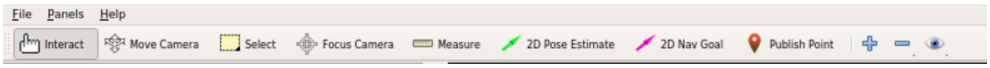

Figura 4.8. Herramientas de RViz, entre ellas 2D Pose Estimate y 2D Nav Goal.

5 NAVEGACIÓN CON ROS

Para este proyecto se ha utilizado la navegación en 2D. Nos serviremos de la información de la odometría, de los sensores y de la posición del destino, así como el mapa del entorno.

Es imprescindible que el láser utilizado sea del tipo LIDAR, esto es, un láser que pueda determinar la distancia desde el emisor hasta cualquier objeto que se encuentre en su rango.

Realizar la tarea de navegación es compleja, por lo que necesitaremos una serie de requerimientos explicados más adelante en este punto. Podemos ver gráficamente la complejidad que esta tarea supone mediante la herramienta *rqt_graph* una vez lanzados los nodos de navegación en el robot real (ver figura 5.1), en este esquema podemos encontrar los nodos y los topics activos, así como la relación entre ellos.

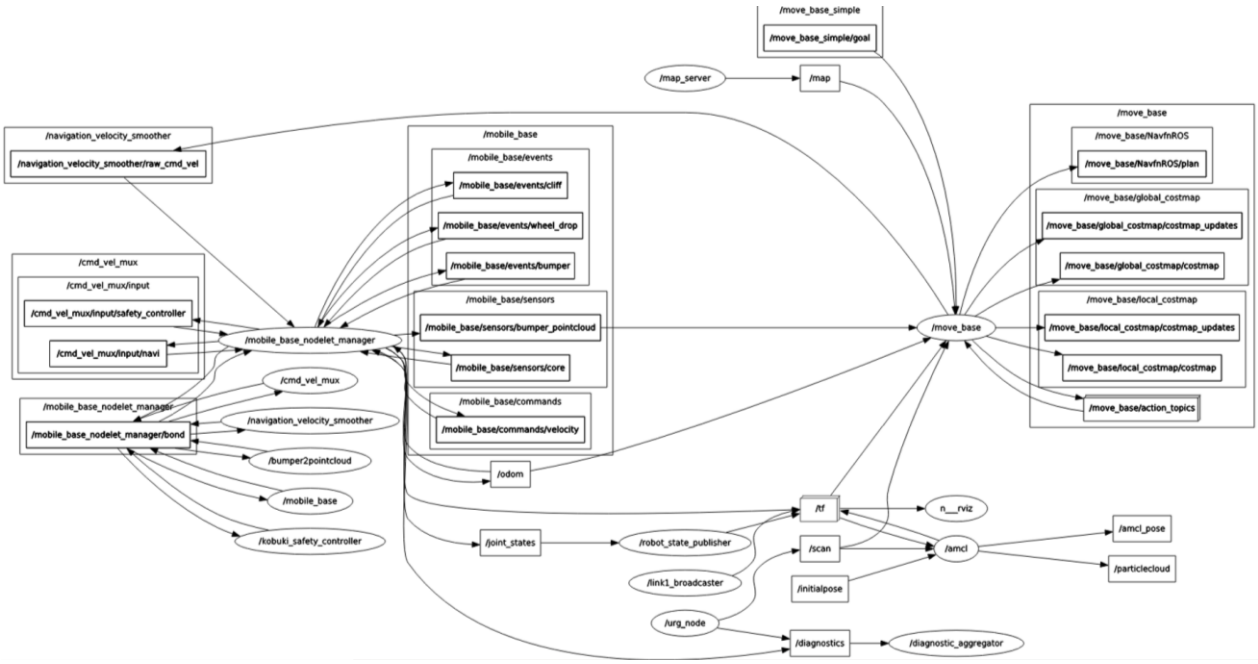

Figura 5.1. Herramienta rqt_graph. Nodos y topics activos y la relación entre ellos cuando se ejecuta la pila de navegación en el robot real.

5.1 Odometría

La odometría es fundamental para poder estimar la pose de un robot. La idea fundamental es la integración de información incremental del movimiento a lo largo del tiempo, por este motivo, los sensores más utilizados para la medición son los sensores propioceptivos, en concreto, los encoders. Hablamos de estimación, y no de determinación, ya que los sensores para calcular la posición del robot no tienen una precisión absoluta, lo que producirá una cierta acumulación de errores.

La posición de la odometría siempre es respecto al punto de inicio, considerando el punto de inicio, la pose inicial del robot en la que empieza a estimar la posición. Por lo que el punto de inicio del robot no tiene por qué coincidir con el punto de inicio de la odometría.

A pesar de los posibles errores que puede producir no utilizar un medio exacto, es una parte muy importante del sistema de navegación de un robot. La pila de navegación recibe la información de la odometría mediante el *tf* y el tipo de mensaje *nav_msgs/Odometry*.

5.2 Mapeado del entorno: Gmapping

Este paquete de ROS se basa en la técnica SLAM (localización y mapeo simultáneos), utiliza un nodo llamado *slam_gmapping*, el cual crea un mapa de celda en 2D a partir de los datos del láser y la odometría del robot.

La técnica SLAM es una técnica usada en robótica para crear un mapa de un entorno. La ventaja de esta técnica reside en que el mapa se va creando simultáneamente mientras el robot se mueve.

El contenido del paquete se puede ver en la plataforma de Github. Su funcionamiento es muy sencillo: Cada celda del mapa está representada por un número del 0 al 100, el 0 es totalmente libre y el 100 es totalmente ocupada, el -1 indica que no se conoce nada de esa celda.

Para el correcto funcionamiento del paquete es muy importante que se realicen una serie de transformaciones en ROS:

- Escaneos del láser → Base_link: Transmitido periódicamente por *robot_state_publisher* o un *tf static_transform_publisher*.
- Base_link → odom: Generalmente proporcionado por el controlador de la base móvil.

A su vez, el paquete realizará otra transformación:

- Map → odom: La estimación actual de la pose del robot dentro del mapa.

Topics suscritos

- ***Tf*** con mensajes *tf/tfMessage* para las transformaciones necesarias.
- ***Scan*** con mensjes *sensor_msgs/LaserScan* para crear el mapa en base a los datos del láser.

Topics publicados

- ***Map_metadata*** con mensajes *nav_msgs/MapMetaData* para los datos del mapa.
- ***Map*** con mensajes *nav_msgs/OccupancyGrid* para los datos del mapa.

Parámetros para destacar

- ***Map_frame*****:** Sistema de referencia del mapa, por defecto, *map*.
- ***Odom_frame*****:** Sistema de referencia de la odometría, por defecto, *odom*.
- ***Base_frame*****:** Sistema de referencia de la base, por defecto, *base_link*.

5.3 Navigation stack

La pila de Navegación en 2D toma datos de la odometría, sensores y objetivos y genera comandos de velocidad que se envían a la base móvil. Dentro de este stack, nos serán de gran ayuda los paquetes *Map_server* (para almacenar y guardar los datos del mapa), *amcl* (para localización) y *Move_base* (para la planificación).

5.3.1 Guardado y utilización del mapa: Map_server

Una vez utilizado el nodo de *slam_gmapping* (o cualquier otro nodo SLAM), antes de cerrarlo, debemos ejecutar el nodo *map_saver* del paquete *map_server*. Este nodo permite guardar los datos del mapa para que puedan ser ejecutados por el nodo *map_server* del mismo paquete. Ejemplo de utilización del nodo *map_saver*:

rosrun map_server map_saver -f nombre_del_mapa

Map_saver escribe los datos del mapa en dos archivos: Uno es una imagen de extensión pgm, el otro es de extensión yaml, que es el que almacena los metadados.

En la imagen, están descritos los estados de ocupación de las celdas por colores. El píxel negro se refiere a que está ocupado, el más blanco significa que está libre, y, el intermedio es desconocido.

En el archivo yaml nos encontramos con los siguientes parámetros:

- ***Image**:* Ruta al archivo de imagen que contiene los datos de ocupación.
- ***Resolution**:* Resolución del mapa, en metros/píxel.
- ***Origin:*** La pose 2-D del píxel inferior izquierdo en el mapa.
- ***Ocupied_thresh:*** Los píxeles con una probabilidad de ocupación superior a este umbral se consideran completamente ocupados.
- ***Free_thresh:*** Los píxeles con una probabilidad de ocupación inferior a este umbral se consideran completamente libres.
- ***Negate:*** Si la semántica blanco/negro libre/ocupado debe invertirse (la interpretación de los umbrales no se ve afectada).

El nodo *map_server* publica en los topics map (*nav_msgs/OccupancyGrid*) y map_metadata (*nav_msgs/MapMetaData*). Ejemplo de utilización del nodo *map_server*:

rosrun map_server map_server nombre_del_mapa.yaml

5.3.2 Localización: AMCL

El AMCL es un sistema de localización probabilística para un robot que se mueve en 2D. Implementa la técnica de localización de Monte Carlo adaptativa (KLD sampling), que utiliza un filtro de partículas para determinar la pose del robot en un mapa conocido.

Al iniciar el paquete, el *amcl* inicializa su filtro de partículas de acuerdo con los parámetros proporcionados, si estos no están establecidos, el estado del filtro inicial será una nube de partículas centrada en (0,0,0).

Topics suscritos

- ***Scan*** con mensajes *sensor_msgs/LaserScan* para los datos del láser.
- ***Tf*** con mensajes *tf/tfMessage* para las transformaciones.
- ***Initialpose*** con mensajes *geometry_msgs/PoseWithCovarianceStamped* para la media y covariancia con la que iniciar el filtro de partículas.
- ***Map*** con mensajes *nav_msgs/OccupancyGrid* para utilizar el mapa conocido.

Topics publicados

- ***Amcl_pose*** con mensajes *geometry_msgs*/*PoseWithCovarianceStamped* para la pose estimada del robot en el mapa, con covarianza.
- ***Particlecloud*** con mensajes *goemetry_msgs*/*PoseArray* para las diferentes poses estimadas por el filtro.
- ***Tf*** con mensajes tf/tfMessage para publicar las transformaciones de *odom* al mapa.

Parámetros para destacar

- ***Base_frame_id****:* Es muy importante definir bien los sistemas de referencia, en nuestro caso sería base_link.
- ***Odom_model_type:*** La configuración de nuestro robot es diferencial, por lo que *odom_model_type* debe ser *diff-corrected*.
- ***Min_particles*** **y** ***max_particles:*** Número mínimo y máximo de partículas para el filtro, por defecto son 100 y 5000, respectivamente.
- ***Inital_pose_x/y/z*****:** Para especificar la pose inicial.

5.3.3 Planificación: Move_base

Como ya se ha explicado anteriormente, la navegación va a utilizar dos mapas de costos para almacenar información sobre obstáculos en el mundo; el mapa global, para crear planes a largo plazo, y el mapa local, para la evitación de obstáculos. Ambos mapas tendrán parámetros comunes además de sus parámetros individuales. Por este motivo, debemos crear tres tipos de configuraciones para el *move_base*: opciones de configuración comunes, opciones de configuración globales y opciones de configuración locales.

Configuración común

Donde definimos los parámetros comunes. De este archivo de configuración destacamos los siguientes parámetros:

- ***Obstacle_range:*** La distancia máxima a un objeto para que se coloque en el mapa de costos.
- ***Raytrace_range:*** El espacio que se intente marcar como libre después de la lectura del sensor.

- ***Robot_radius:*** El radio del robot, ya que nuestro robot es circular. En caso contrario, *footprint.*
- ***Inflation_radius:*** Distancia máxima de los obstáculos en la que se debe incurrir en un costo.
- ***Observation_sources:*** Define una lista de sensores que van a pasar información al mapa de costos separados por espacios. En nuestro caso es importante poner el láser.

Configuración global

Donde definimos los parámetros para la planificación. De este archivo de configuración destacamos los siguientes parámetros:

- ***Global_frame:*** Define en qué marco de coordenadas debe ejecutarse el mapa de costos, en este caso, elegiremos */map*.
- ***Robot_base_frame:*** Define el marco de coordenadas al que debe hacer referencia el mapa de costos para la base del robot.
- ***Update_frequency:*** Determina la frecuencia, en Hz, a la que costmap ejecutará su bucle de actualización.
- ***Static_map:*** Determina si el mapa de costos debe o no inicializarse en función de un mapa servido por *map_server*.

Configuración local

Donde definimos los parámetros para la planificación global. De este archivo de configuración destacamos los siguientes parámetros:

- ***Global_frame, robot_base_frame, update_frequency y static_map***: Los mismos que en la configuración global.
- ***Rolling_window:*** en *true* significa que el mapa de costos permanecerá centrado alrededor del robot mientras el robot se mueve por el entorno.
- ***Width, height y resolution:*** Ancho y alto, en metros, del mapa de costos. Y resolución (metros/celda) del mapa de costos.

Además de los archivos ya mencionados, necesitaremos otras configuraciones para la planificación de la trayectoria, en este caso, basada en el DWA ya explicado. En estos archivos declararemos la velocidad lineal o angular, así como los planes alternativos (recovery_behaviors) cuando el robot se encuentra atascado, entre otros muchos más parámetros.

Nodo move_base

El nodo *move_base*, que es un componente principal de la pila de navegación (ver figura 5.2). A continuación, se muestra una descripción detallada del nodo:

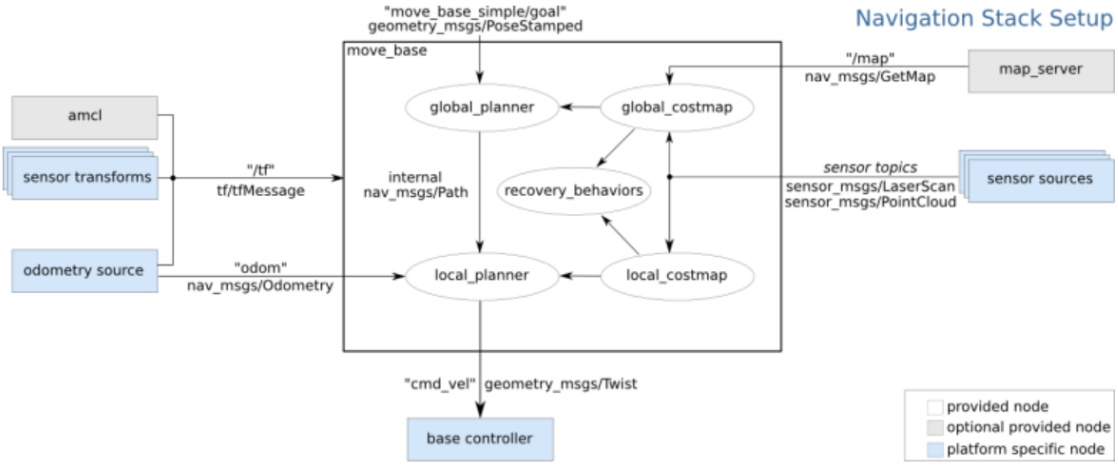

Figura 5.2. Vista de alto nivel del nodo move_base y su interacción con otros componentes.

Este proporciona una interfaz ROS para configurar, ejecutar e interactuar con la pila de navegación en un robot. En la figura 5.2 los nodos azules varían según la plataforma del robot, los grises son opcionales, pero se proporcionan para todos los sistemas y los nodos blancos son obligatorios, pero también se proporcionan para todos los sistemas.

El nodo *move_base* proporciona una implementación de *SimpleActionServer*, que acepta objetivos que contienen mensajes de *geometry_msgs/PoseStamped.* Puede comunicarse con el nodo *move_base* a través de ROS directamente, pero la forma recomendada de enviar objetivos a *move_base* si desea realizar un seguimiento de su estado es mediante *SimpleActionClient*.

Action está suscrito a los topics:

- ***Move_base/goal*** con mensajes *move_base_msgs/MoveBaseActionGoal*: El objetivo a seguir.
- ***Move_base/cancel*** con mensajes *actionlib/GoalID*: Cancelación del objetivo

Action publica en los topics:

- ***Move_base/feedback*** con mensajes *move_base_msgs/MoveBaseActionFeedback:* La posición actual.
- ***Move_base_status*** con mensajes *actionlib_msgs/GoalStatusArray*: Información de estado sobre los objetos que se envían a la acción *move_base*.
- ***Move_base/result*** con mensajes *move_base_msgs/MoveBaseActionResult.*

Topics suscritos de move_base

- ***Move_base_simple/goal*** con mensajes *geometry_msgs/PoseStamped.*

Topics publicados de move_base

- ***Cmd_vel*** con mensajes *geometry_msgs/Twist.*

6 METODOLOGÍA

6.1 Creación del espacio de trabajo

6.1.1 Descarga de WSL, Ubuntu y ROS

Debido a que el ordenador inicial utilizado no poseía el sistema operativo Linux, fue necesaria la descarga del WSL2 (Windows Subsystem Linux). Para la descarga se utilizó la guía de *Microsoft docs: Pasos de la instalación manual para versiones anteriores de WSL,* que utiliza una serie de comandos en Power Shell.

Una vez instalado el WSL, el siguiente paso fue la descarga de la distribución de Linux necesaria. En nuestro caso, se instaló la versión 18.04.5 de Ubuntu en Microsoft Store, la más adecuada para ROS Melodic, por lo que se realizó la descarga de ROS Melodic. Con el siguiente ordenador, se utilizará ROS Kinetic por ser el que ya estaba instalado, tanto en el ordenador como en el NUC del Turtlebot.

La primera vez que se ejecuta Ubuntu se pide que se especifique un usuario y contraseña. El usuario utilizado en Melodic es raquel y, en el ordenador con ROS Kinetic, himtae.

El siguiente paso solo es requerido en el ordenador que no posee Linux: la instalación de algunas utilidades como son el editor Gedit o un servidor X, en este caso, Xming. Este último es necesario al haber instalado el WSL, y se descarga gratuitamente de internet. Una vez instalado, lanzamos XLaunch y configuramos las opciones (ver imagen 6.1).

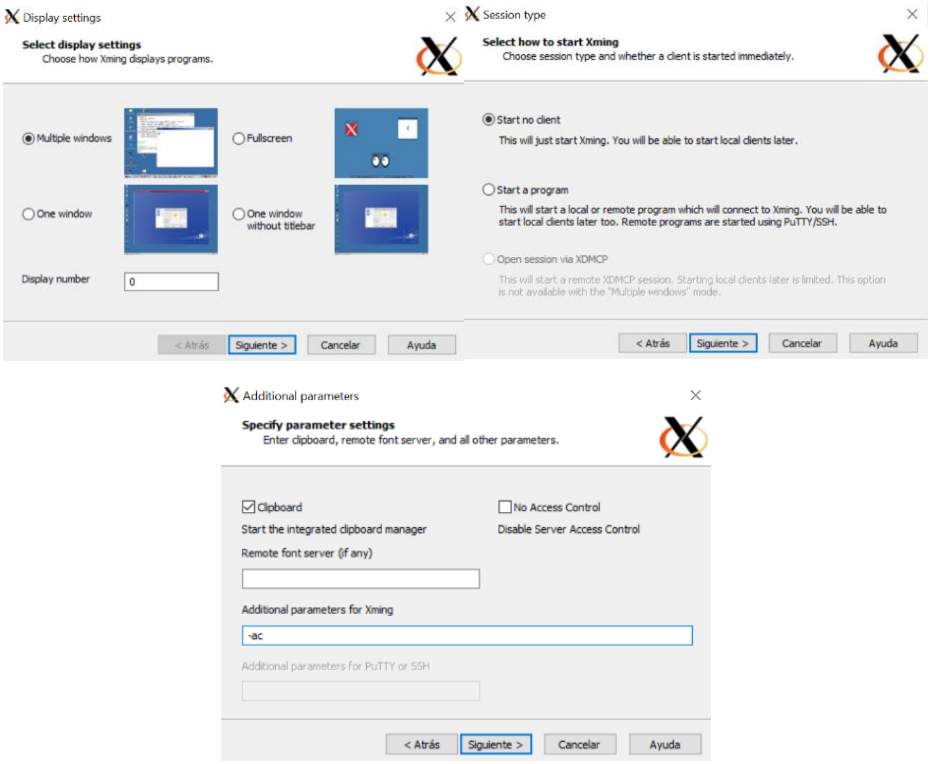

Figura 6.1. Configuración de Xming.

Una vez configuradas las opciones, nos aparecerá el icono en la barra de herramientas (ver figura 6.2).

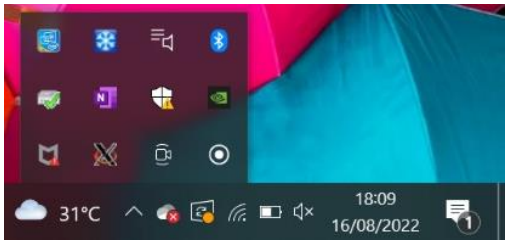

Figura 6.2. Icono de Xming en la barra de herramientas.

Seguramente cuando ejecutemos Gedit aparezca un error, ya que la variable *$DISPLAY* no está actualizada. Podemos escribir en el *.bashrc: export DISPLAY=dirección_IP:0* o ejecutando el comando *echo DISPLAY=dirección_IP:0.*

export DISPLAY=192.168.0.104:0

Si lo hemos escrito en *.bashrc* debemos hacer un *source .bashrc* para que se actualice.

Una vez comprobado que Gedit funciona correctamente, pasamos a la instalación de ROS Melodic. Para la descarga de este simplemente hay que seguir uno por uno los pasos de la *Wiki* de ROS.

6.1.2 Catkin

Tras la correcta descarga de ROS Melodic es necesario crear un espacio de trabajo en donde podamos ejecutar nuestros programas. Siguiendo también los tutoriales de ROS:

1) Comprobamos que estamos en la carpeta personal con el comando *pwd*:

```
raquel@LAPTOP-TQ85PTKE:~$ pwd
/home/raquel
```

2) Creamos la carpeta *catkin_ws* que a su vez contendrá la carpeta *src*:

```
raquel@LAPTOP-TQ85PTKE:~$ mkdir catkin_ws/src
```

3) Vamos a la carpeta *catkin_ws* y ejecutamos el comando *catkin_make*:

```
raquel@LAPTOP-TQ85PTKE:~/catkin_ws$ catkin_make
```

Si se ha hecho correctamente, cuando ejecutemos *ls* se habrán creado dos carpetas dentro de *catkin_ws*:

```
raquel@LAPTOP-TQ85PTKE:~/catkin_ws$ ls
build  devel  src
```

4) Abrimos el *.bashrc* con Gedit y añadimos:

source /home/raquel/catkin_ws/devel/setup.bash

Una vez hecho esto, podemos cerrar todas las terminales abiertas, o poner *source .bashrc* para que se actualice el *.bashrc*.

6.1.3 Descarga del paquete de Turtlebot

Para poder simular el Turtlebot en nuestro ordenador existen ciertos paquetes en el repositorio de Github. Además, no sólo contienen el URDF del Turtlebot, también poseen otras muchas aplicaciones ya creadas. Una vez situados en nuestra carpeta *src*, el comando necesario será:

Curl -sLf https://raw.githubuser/content.com/gaunthan/Turtlebot2-On-Melodic/master/install_all.sh | bash

Se selecciona este enlace por ser el más completo y por utilizar ROS Melodic en el primer ordenador. Se ha utilizado el mismo enlace en el ordenador que posee Ros Kinetic y no ha dado ningún problema y todo ha funcionado con normalidad.

Se descargarán una gran cantidad de carpetas (ver figura 6.3), pero de nuestro interés básicamente serán turtlebot, turtlebot_simulator y turtlebot_apps.

```
raquel@LAPTOP-TQ85PTKE:~/catkin_ws/src/turtlebot2$ ls
depthimage_to_laserscan  kobuki_desktop  kobuki_safety_controller   turtlebot_apps          yocs_cmd_vel_mux
kobuki                   kobuki_keyop    orocos-bayesian-filtering  turtlebot_interactions  yocs_controllers
kobuki_bumper2pc         kobuki_msgs     robot_pose_ekf             turtlebot_msgs
kobuki_description       kobuki_node     turtlebot                  turtlebot_simulator
raquel@LAPTOP-TQ85PTKE:~/catkin_ws/src/turtlebot2$
```

Figura 6.3. Contenido de paquete turtlebot.

En la carpeta de turtlebot encontramos los archivos básicos para el funcionamiento del Turtlebot:

- Turtlebot_bringup contiene archivos para inicialización del robot.

```
raquel@LAPTOP-TQ85PTKE:~/catkin_ws/src/turtlebot2/turtlebot/turtlebot_bringup/launch$ ls
3dsensor.launch  concert_client.launch  concert_minimal.launch  includes  minimal.launch
```

- Turtlebot_description contiene los archivos necesarios para la descripción y visualización del robot, es decir, el urdf.

```
raquel@LAPTOP-TQ85PTKE:~/catkin_ws/src/turtlebot2/turtlebot/turtlebot_description/urdf$ ls
common_properties.urdf.xacro  stacks                          turtlebot_gazebo.urdf.xacro
sensors                       turtlebot_common_library.urdf.xacro  turtlebot_properties.urdf.xacro
```

- Turtlebot_teleop para la teleoperación del robot.

```
raquel@LAPTOP-TQ85PTKE:~/catkin_ws/src/turtlebot2/turtlebot/turtlebot_teleop/launch$ ls
includes  keyboard_teleop.launch  logitech.launch  ps3_teleop.launch  xbox360_teleop.launch
```

En la carpeta turtlebot_simulator se encuentran todos los archivos necesarios para lanzar la simulación del robot. Nos centraremos en el paquete turtlebot_gazebo, pues es la que contiene los archivos de los mapas, mundos y lanzamientos básicos del Turtlebot en la herramienta Gazebo, los cuales tomaremos como referencia para visualizar las simulaciones personalizadas para este proyecto.

De la misma forma, de la carpeta de turtlebot_apps, nos quedaremos con el paquete turtlebot_navigation, en el cual tomaremos diversos archivos de referencia para la creación de nuestros propios paquetes de navegación. Entre estos archivos se encuentra la integración del láser en el *amcl* o la estructura de los parámetros del *move_base*, entre otros.

6.2 Simulación en Gazebo y visualización en RViz

Antes de la prueba en el robot real, es conveniente probar los códigos realizados en algún simulador. El simulador elegido es Gazebo, ya que, como se ha explicado anteriormente, el paquete del Turtlebot descargado cuenta con grandes ayudas para la visualización en este simulador.

6.2.1 Simulación del robot

Para la simulación del robot en Gazebo, utilizamos el archivo turtlebot_world.launch de la carpeta turtlebot_gazebo:

```
<launch>
<arg name="gui"         default="true"/>
<arg name="world_file" default="$(env TURTLEBOT_GAZEBO_WORLD_FILE)"/>

<arg name="base"         value="$(optenv TURTLEBOT_BASE kobuki)"/> <!-- create,
roomba -->

<arg     name="battery"                    value="$(optenv     TURTLEBOT_BATTERY
/proc/acpi/battery/BAT0)"/>      <!--   /proc/acpi/battery/BAT0   -->      <arg
name="stacks"      value="$(optenv TURTLEBOT_STACKS hexagons)"/>  <!-- circles,
hexagons -->

<arg  name="3d_sensor"   value="$(optenv  TURTLEBOT_3D_SENSOR  kinect)"/>   <!--
kinect, asus_xtion_pro -->

<include file="$(find gazebo_ros)/launch/empty_world.launch">
<arg name="use_sim_time" value="true"/>
<arg name="debug" value="false"/>
<arg name="gui" value="$(arg gui)" />
<arg name="world_name" value="$(arg world_file)"/>
</include>

<include          file="$(find          turtlebot_gazebo)/launch/includes/$(arg
base).launch.xml">
<arg name="base" value="$(arg base)"/>
<arg name="stacks" value="$(arg stacks)"/>
<arg name="3d_sensor" value="$(arg 3d_sensor)"/>
</include>

<node          pkg="robot_state_publisher"          type="robot_state_publisher"
name="robot_state_publisher">
<param name="publish_frequency" type="double" value="30.0" />
```

```
</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="camera_depth_frame"/>
<param name="range_min" value="0.45"/>
<remap from="image" to="/camera/depth/image_raw"/>
<remap from="scan" to="/scan"/>
</node>
</launch>
```

Nos fijamos en la línea:

```
  <arg name="world_file" default="$(env TURTLEBOT_GAZEBO_WORLD_FILE)"/>
```

Debemos comprobar qué tenemos declarado en la variable *TURTLEBOT_GAZEBO_WORLD_FILE* antes de la simulación, para comprobar qué entorno estamos abriendo.:

echo $TURTLEBOT_GAZEBO_WORLD_FILE

Para la simple visualización será necesario que esta variable apunte a un entorno vacío de gazebo (*empty.world*).

Además, como podemos observar, turtlebot_world.launch abre también el archivo kobuki.launch.xml (ver figura 6.4), siendo:

```
<launch>
  <arg name="base"/>
  <arg name="stacks"/>
  <arg name="3d_sensor"/>

  <arg name="urdf_file" default="$(find xacro)/xacro '$(find turtlebot_description)/robots/$(arg base)_$(arg stacks)_$(arg
3d_sensor).urdf.xacro'"/>
  <param name="robot_description" command="$(arg urdf_file)"/>

  <!-- Gazebo model spawner -->
  <node name="spawn_turtlebot_model" pkg="gazebo_ros" type="spawn_model"
        args="$(optenv ROBOT_INITIAL_POSE) -unpause -urdf -param robot_description -model mobile_base"/>

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

  <!-- Bumper/cliff to pointcloud (not working, as it needs sensors/core messages) -->
  <include file="$(find turtlebot_bringup)/launch/includes/kobuki/bumper2pc.launch.xml"/>
</launch>
```

Figura 6.4. Archivo kobuki.launch.xml.

Efectivamente, el archivo también abrirá las extensiones *xacro* del URDF de la descripción del robot, como se ha comentado previamente.

Una vez comprobado, podemos lanzar turtlebot_world.launch (ver figura 6.5).

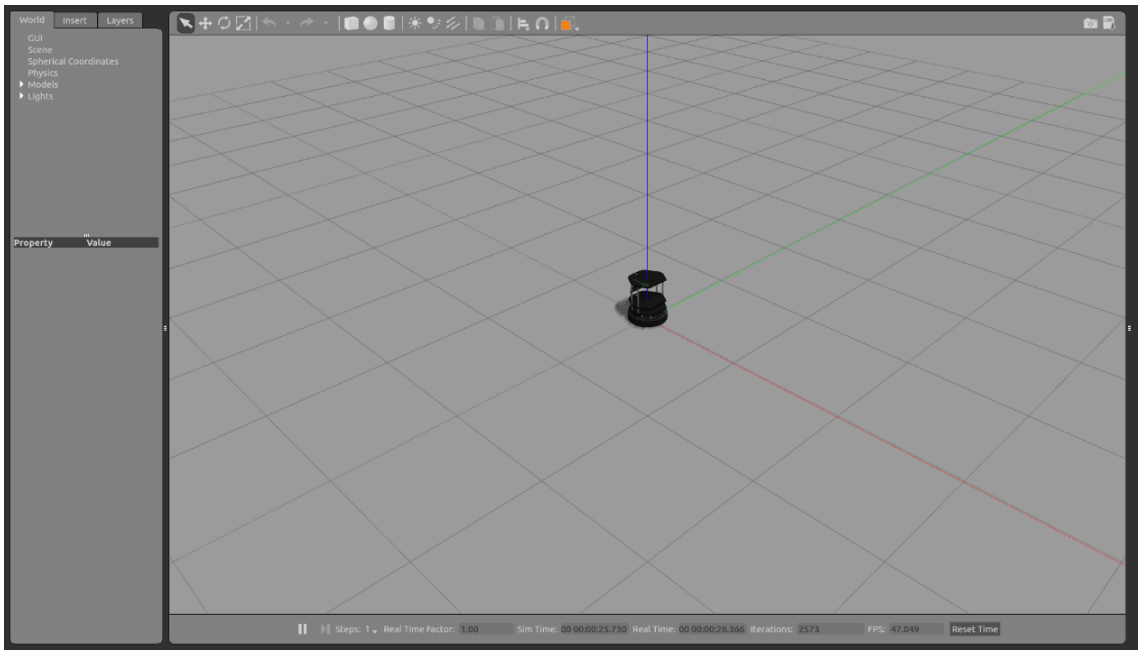

Figura 6.5. Imagen de Gazebo una vez lanzado turtlebot.world.launch.

6.2.2 Creación del entorno

Para la creación de un entorno en Gazebo, en primer lugar, lanzamos turtlebot_world.launch. Una vez que Gazebo está abierto, desplegamos la pestaña “edit” y seleccionamos ”building editor”. De esta forma, se abre una nueva pestaña (ver figura 6.6) en la que podemos añadir paredes con “Create Walls” o cargar un plano a partir de una imagen con “Import”.

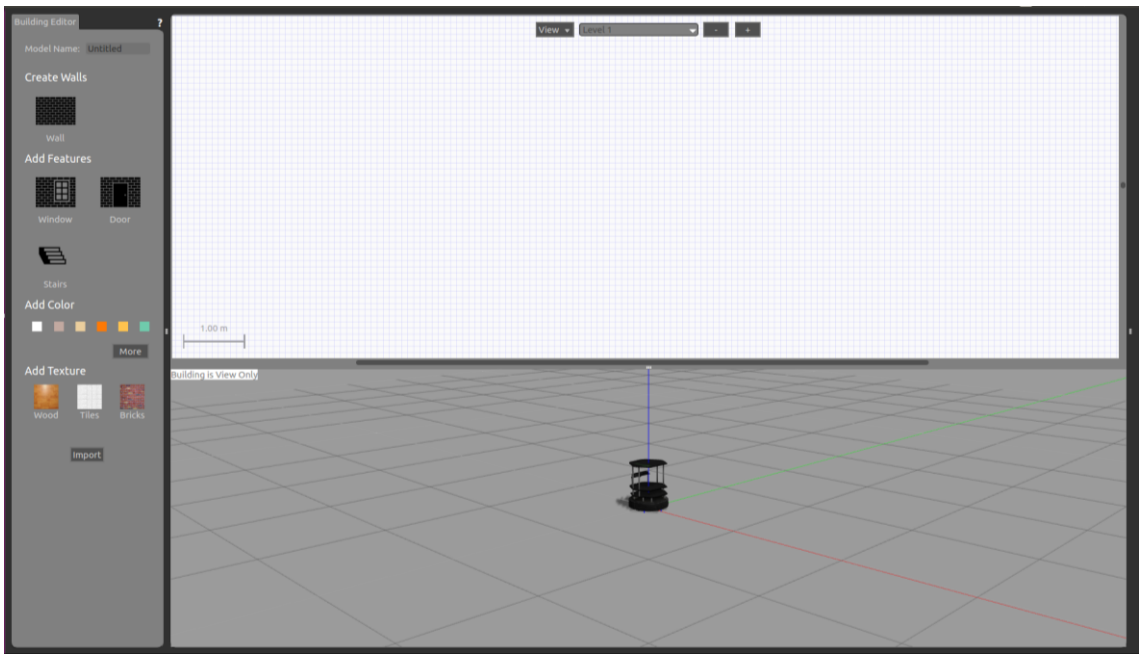

Figura 6.6. Imagen de Gazebo con la herramienta Building Editor abierta.

Levantamos las paredes de forma que el entorno construido se asemeje a una casa simple que contenga una habitación, un aseo, una cocina y un salón (ver figura 6.7), así nuestro entorno simulado será más fiel a la realidad.

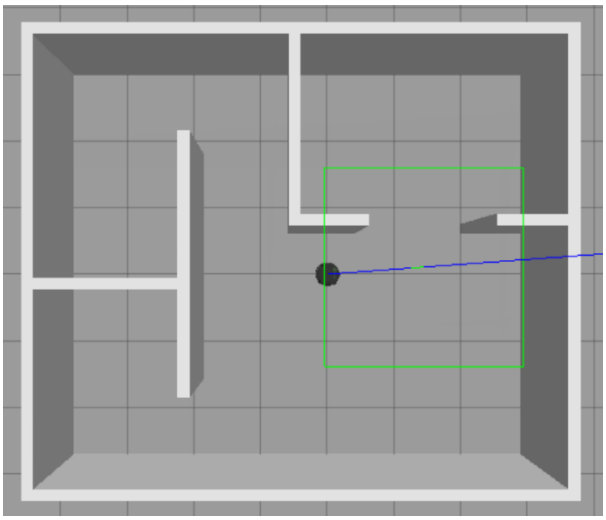

Figura 6.7. Casa construida en Gazebo con la herramienta Building Editor.

Una vez creado el entorno, procedemos a guardarlo con "Save as" (ver figura 6.8). Gazebo por defecto va a guardar el mapa en */home/raquel/building_editors_models* pero podemos modificarlo para que directamente nos lo guarde en */home/raquel/.gazebo/models*, que es donde necesitamos que esté para que podamos utilizarlo en un archivo .world (ver figura 6.9).

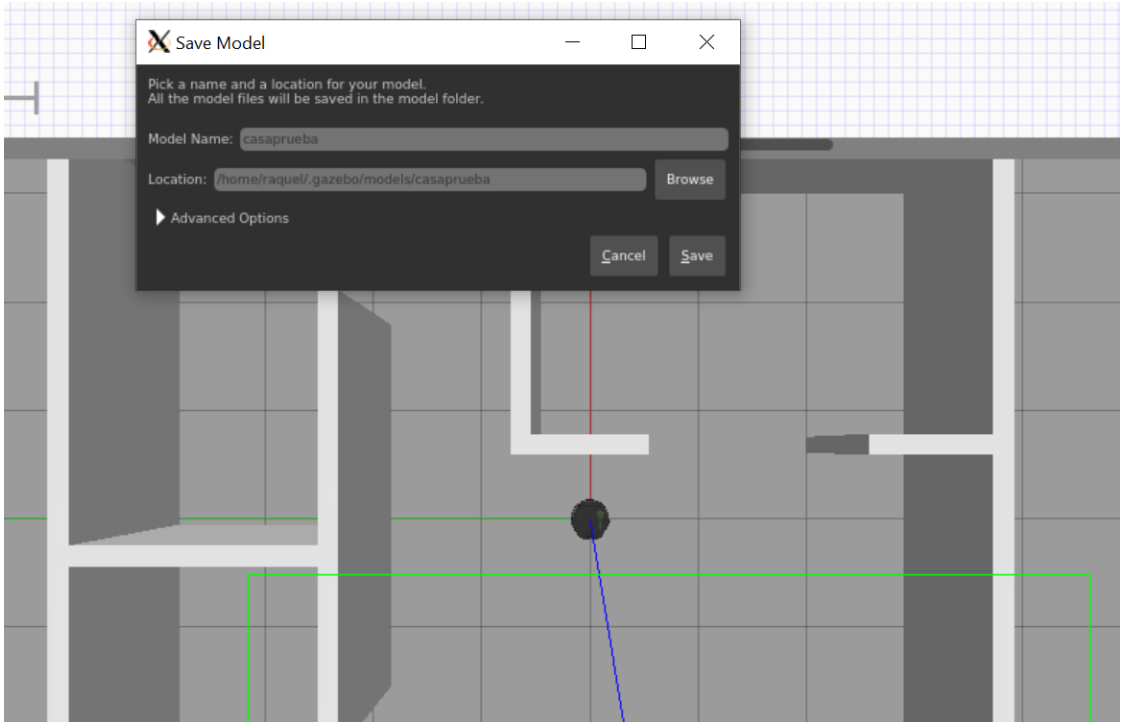

Figura 6.8. Ejemplo de guardado de un entorno en Gazebo.

```
GNU nano 2.9.3                    prueba.world                    Modified

<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
    <!-- A ground plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://casaprueba</uri>
    </include>
  </world>
</sdf>
```

Figura 6.9. Ejemplo de uso de un entorno creado en Gazebo en un archivo .world, en este caso, prueba .world.

Para abrir el robot y el entorno en Gazebo (ver figura 6.11), podemos editar el archivo turtlebot_world.launch (ver figura 6.10), o bien, editar la variable *TURTLEBOT_GAZEBO_WORLD_FILE* ya mencionada:

- Editando turtlebot_world.launch en un nuevo fichero llamado turtlebot_casaprueba.launch:

```
GNU nano 2.9.3                  turtlebot_casaprueba.launch

<launch>
  <arg name="gui"        default="true"/>
  <arg name="world_file" value="$(find turtlebot_gazebo)/worlds/prueba.world"/>
```

Figura 6.10. Detalle de archivo turtlebot_casaprueba.launch.

- Editando la variable *TURTLEBOT_GAZEBO_WORLD_FILE*:
  Echo TURTLEBOT_GAZEBO_WORLD_FILE=/home/raquel/src/catkin_ws/turtlebot2/turtlebot_simulator/turtlebot_gazebo/worlds/prueba.world

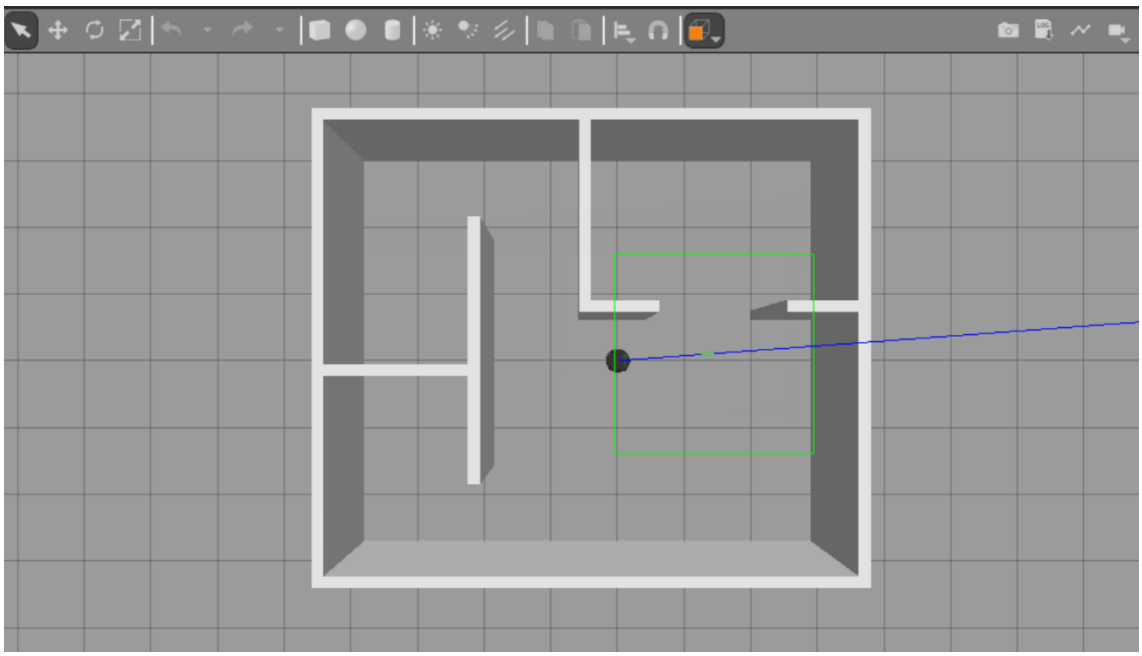

Figura 6.11. Imagen de Gazebo una vez lanzado turtlebot_casaprueba.launch donde se muestra el robot y el entorno construido en Gazebo.

6.2.3 Creación del mapa con Gmapping

En primer lugar, se deberá realizar la instalación del paquete *gmapping*:

sudo apt-get install ros-kinetic-gmapping

Una vez descargado, abrimos turtlebot_casaprueba.launch y la herramienta de teleoperación keyboard_teleop.launch, del paquete turtlebot_teleop.

A continuación, lanzamos el nodo *slam_gmapping* del paquete *gmapping*:

rosrun gmapping slam_gmapping _base_frame:=base_link

Debemos asegurarnos cuál es el nombre del _base_frame porque puede que no siempre se llame igual.

Después de lanzar el comando nos aparecerá por pantalla que se ha registrado el primer escaneo (ver figura 6.12).

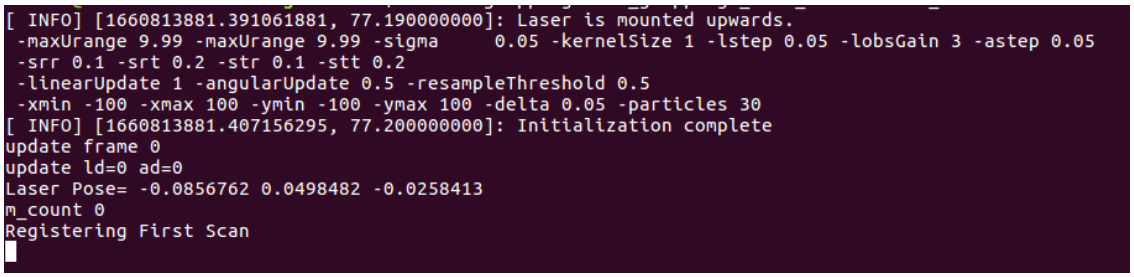

Figura 6.12. Salida por pantalla al ejecutar el nodo slam_gmapping.

Si hacemos *rostopic list* podemos comprobar cómo se han creado topics pertenecientes al mapa como son */map* y */map_metadata* (ver figura 6.13).

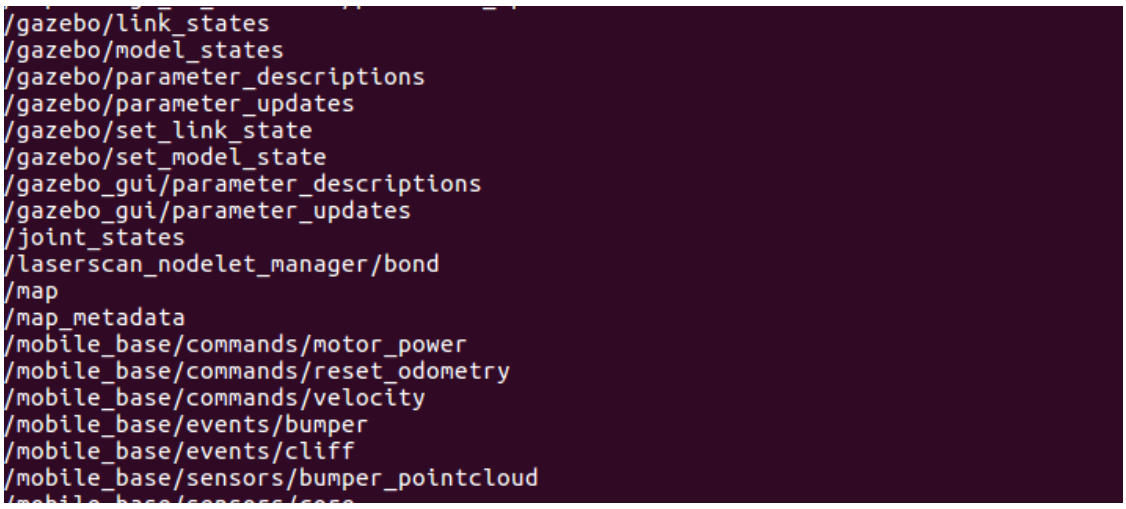

Figura 6.13. Topics referentes al mapa en rostopic list una vez lanzado el nodo slam_gmapping.

A continuación, abrimos RViz y observamos como se ha creado un *frame map* y, si comenzamos a teleoperar, el mapa se va a ir formando en pantalla (ver figura 6.14).

Para poder visualizar correctamente la creación del mapa, debemos añadir en RViz:

- Modelo del robot.
- Láser publicado en */scan*.
- Mapa publicado en */map*.

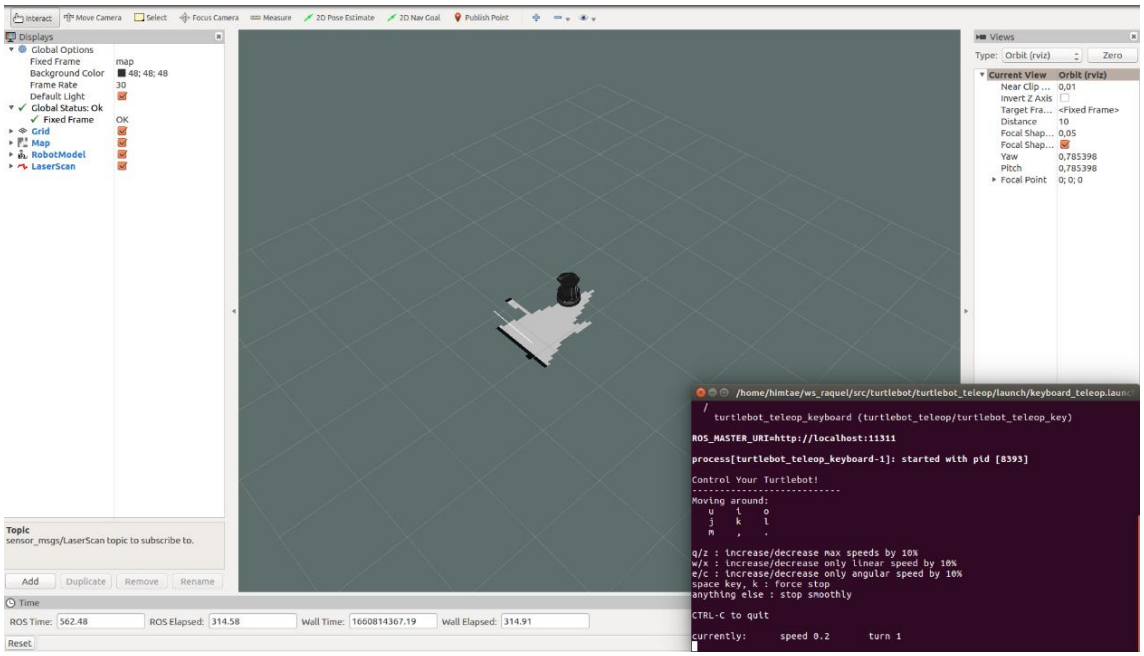

Figura 6.14. Imagen de RViz mientras se forma el mapa del enorno Gazebo mediante la teleoperación.

6.2.4 Guardado del mapa con Map_server

Después de haber teleoperado el robot múltiples veces por el entorno, el mapa estará creado. El siguiente paso será guardar el mapa, por lo que instalamos el paquete *map_server*, que está contenido dentro del stack de navegación:

sudo apt-get install ros-kinetic-map-navigation

Al descargarnos este stack también se nos descargarán los paquetes de *move_base* y *amcl*, que nos serán de ayuda más adelante.

Creamos la carpeta mapas y utilizamos el nodo *map_saver* de *map_server* para guardar el mapa:

mkdir mapas
cd mapas
rosrun map_server map_saver -f mapa_casa

En la carpeta mapas se han creado dos archivos: mapacasaprueba.pgm y mapacasaprueba.yaml. El primero es una imagen y en el segundo se observan los parámetros que definen el mapa, como la resolución o la ocupación de celdas, entre otros (ver figura 6.15).

```
 GNU nano 2.9.3                          mapacasaprueba.yaml

image: mapacasaprueba.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Figura 6.15. Archivo mapacasaprueba.yaml formado al guardar el mapa con el nodo map_saver.

Para abrir el mapa previamente guardado es necesario haber cerrado el nodo *gmapping*. Una vez cerrado el nodo, se ejecuta el nodo *map_server* del paquete *map_server* y este lo publica en el topic */map*:

rosrun map_server map_server mapa_casa.yaml

Por comodidad, para poder incluir el mapa en otros ficheros, creamos un archivo .launch.xml llamado mapa_casaprueba.launch.xml, el cual ejercerá la misma función que el comando anterior (ver figura 6.16).

```
 GNU nano 2.9.3                      mapa_casaprueba.launch.xml

<launch>
<node pkg="tf" type="static_transform_publisher" name="static_transform_publisher" args="0 0 0 0 0 0 odom map 100" />
<arg name="map_file" default="$(find robot_gazebo)/mapas/mapacasaprueba.yaml"/>
<node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" respawn="true" />
</launch>
```

Figura 6.16. Archivo .launch para publicar en el topic /map el mapa del entorno de Gazebo: mapa_casaprueba.launch.xml.

6.2.5 Visualización en RViz

Podemos crear un archivo que lance turtlebot_casaprueba.launch junto con mapa_casaprueba.launch.xml. A continuación, abrimos RViz y añadimos el modelo del robot, el láser y el mapa (ver figura 6.17). RViz nos permite guardar esta configuración en un archivo de extensión .rviz, que podemos añadir al final del archivo de la forma:

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find turtlebot_gazebo)/conf_rviz/turtlebot.rviz"/>

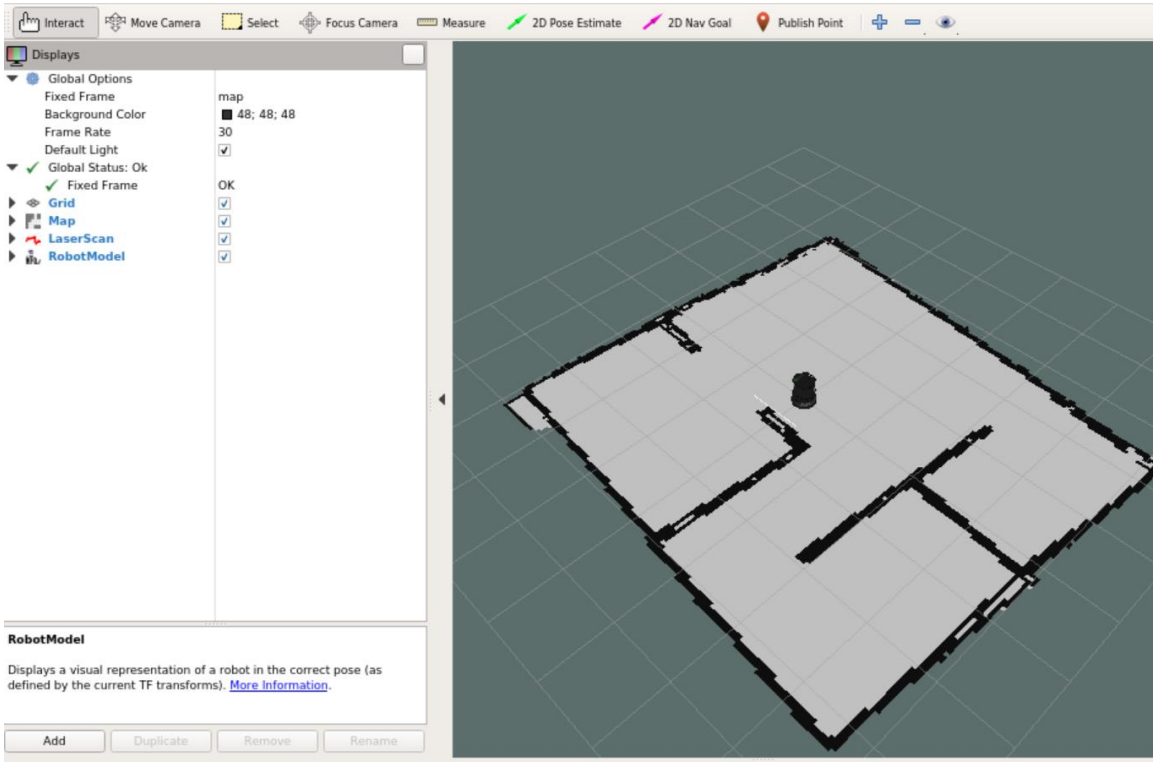

Figura 6.17. Imagen de RViz cuando lanzamos el mapa del entorno generado en Gazebo junto con el robot y el láser.

6.3 Navegación en simulación

Una vez creado el mapa de un entorno es importante saber movernos por él. Hay diferentes paquetes de navigation stack que nos serán de gran ayuda, en concreto el *amcl* y el *move_base*. Estos paquetes nos permitirán localizarnos y planificar rutas en el mapa, respectivamente.

6.3.1 Localización con AMCL

El paquete *amcl* juega un papel fundamental en la localización del robot en el mapa. Implementa la técnica de localización de Monte Carlo adaptativa, que utiliza un filtro de partículas para determinar la pose del robot en un mapa conocido.

Utilizaremos los archivos:

- turtlebot_casaprueba.launch
- mapa_casaprueba.launch.xml
- amcl.launch.xml.

Los dos primeros se han explicado en apartados anteriores. Amcl.launch.xml es un archivo que contiene los parámetros necesarios para configurar el nodo *amcl*, en donde se especifica el filtro en general, el modelo del láser o la odometría.

El nuevo archivo se llamará amcl_raquel.launch, el cual lanzaremos junto con RViz y la teleoperación del turtlebot.

Para poder verificar que el nodo de localización funciona correctamente, en RViz añadiremos:

- La información del láser.
- El modelo del robot.
- El mapa.
- La pose con covarianza.
- El array de partículas.

Al principio el array de partículas saldrá un poco disperso (ver figura 6.18), a medida que nos vamos moviendo por el entorno las partículas se condensarán en un mismo punto, siendo este la ubicación aproximada del robot (ver figura 6.19).

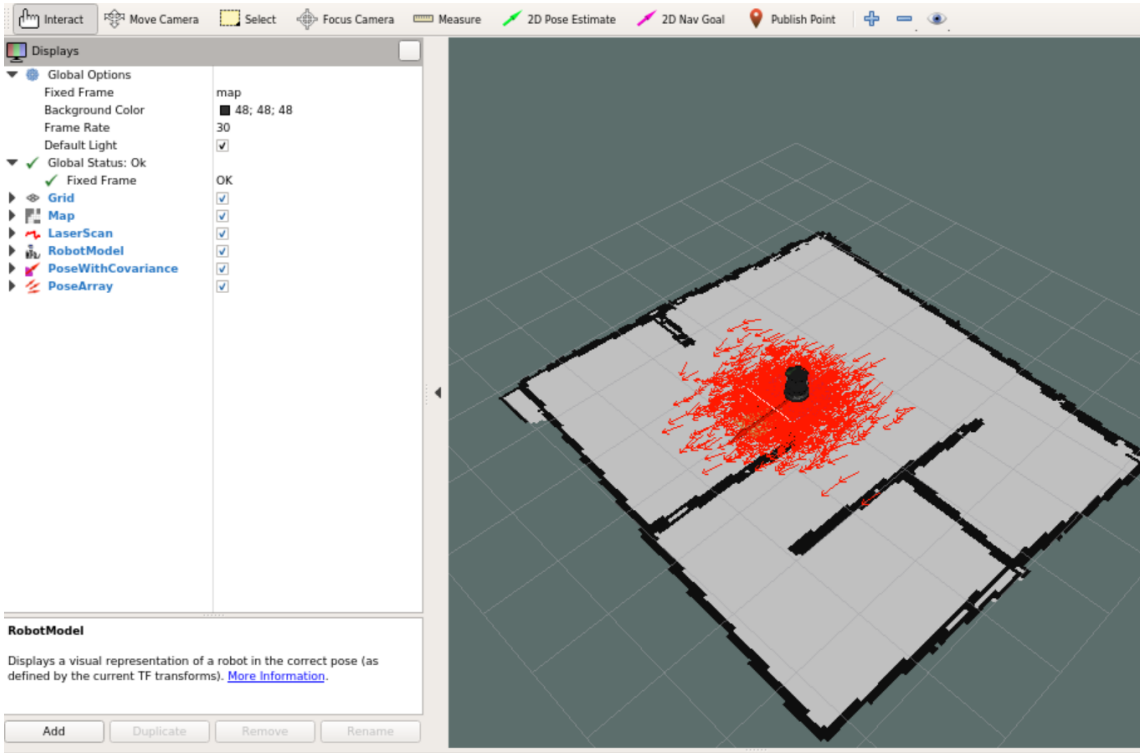

Figura 6.18. Imagen de RViz cuando lanzamos amcl_raquel.launch en donde podemos observar el array de partículas disperso.

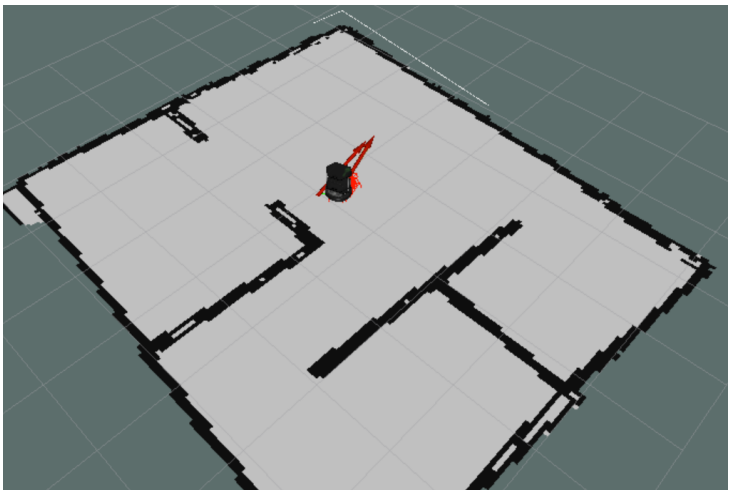

Figura 6.19. Imagen de RViz cuando lanzamos amcl_raquel.launch en donde podemos observar el array centrado en un punto.

De igual forma que en el apartado anterior, podemos guardar la configuración de RViz y añadirla en el archivo para poder abrirla directamente en otras ocasiones.

6.3.2 Planificación con move_base

Una vez que sabemos localizarnos en el mapa, podemos pasar a planificar rutas. El paquete de ROS encargado de la planificación es el *move_base*. Para poder utilizarlo, es necesario declarar una serie de parámetros, ya explicados anteriormente. Es importante recalcar que los parámetros que se exponen a continuación son utilizados únicamente en simulación, por motivos que se explican en el apartado 8.2.2.

- Raq_costmap_common_params.yaml ➔ Donde definimos los parámetros comunes:

```
max_obstacle_height: 0.60  # assume something like an arm is mounted on top
of the robot

# Obstacle Cost Shaping (http://wiki.ros.org/costmap_2d/hydro/inflation)
robot_radius: 0.18  # distance a circular robot should be clear of the
obstacle (kobuki: 0.18)
# footprint: [[x0, y0], [x1, y1], ... [xn, yn]]  # if the robot is not
circular

map_type: voxel

obstacle_layer:
  enabled:              true
  max_obstacle_height:  0.6
  origin_z:             0.0
  z_resolution:         0.2
  z_voxels:             2
  unknown_threshold:    15
  mark_threshold:       0
  combination_method:   1
  track_unknown_space:   true      #true needed for disabling global path
planning through unknown space
  obstacle_range: 3
  raytrace_range: 2
```

```
  origin_z: 0.0
  z_resolution: 0.2
  z_voxels: 2
  publish_voxel_map: false
  observation_sources:  scan bump
  scan:
    data_type: LaserScan
    topic: scan
    marking: true
    clearing: true
    min_obstacle_height: 0.25
    max_obstacle_height: 0.35
  bump:
    data_type: PointCloud2
    topic: mobile_base/sensors/bumper_pointcloud
    marking: true
    clearing: false
    min_obstacle_height: 0.0
    max_obstacle_height: 0.15
  # for debugging only, let's you see the entire voxel grid

#cost_scaling_factor   and   inflation_radius   were   now   moved   to   the
inflation_layer ns
inflation_layer:
  enabled:              true
  cost_scaling_factor:  5.0  # exponential rate at which the obstacle cost
drops off (default: 10)
  inflation_radius:     0.5  # max. distance from an obstacle at which costs
are incurred for planning paths.

static_layer:
  enabled:              true
```

- Local_costmap_params.yaml ➔ Donde definimos los parámetros para la planificación local:

```
local_costmap:
   global_frame: odom
   robot_base_frame: /base_footprint
   update_frequency: 5.0
   publish_frequency: 2.0
   static_map: false
   rolling_window: true
   width: 4.0
   height: 4.0
   resolution: 0.05
   transform_tolerance: 0.5
   plugins:
    - {name: obstacle_layer,      type: "costmap_2d::VoxelLayer"}
    - {name: inflation_layer,     type: "costmap_2d::InflationLayer"}
```

- Global_costmap_params.yaml ➔ Donde definimos los parámetros para la planificación global:

```
global_costmap:
   global_frame: /map
   robot_base_frame: /base_footprint
   update_frequency: 1.0
   publish_frequency: 0.5
   static_map: true
   transform_tolerance: 0.5
   plugins:
     - {name: static_layer,           type: "costmap_2d::StaticLayer"}
     - {name: obstacle_layer,         type: "costmap_2d::VoxelLayer"}
     - {name: inflation_layer,        type: "costmap_2d::InflationLayer"}
```

- Raq_dwa_local_planner_params.yaml y navfn_global_planner_params.yaml ➔ Donde definimos los parámetros para el planificador de trayectorias:

```
DWAPlannerROS:

# Robot Configuration Parameters - Kobuki
  max_vel_x: 0.5  # 0.55
  min_vel_x: 0.1

  max_vel_y: 0.0  # diff drive robot
  min_vel_y: 0.0  # diff drive robot

# max_trans_vel: 0.5 # choose slightly less than the base's capability
#   min_trans_vel: -0.5   # this is the min trans velocity when there is
negligible rotational velocity
#  trans_stopped_vel: 0.1

  # Warning!
  #    do not set min_trans_vel to 0.0 otherwise dwa will always think
translational velocities
  #    are non-negligible and small in place rotational velocities will be
created.

  max_rot_vel: 5.0  # choose slightly less than the base's capability
  min_rot_vel: -0.5   # this is the min angular velocity when there is
negligible translational velocity
  rot_stopped_vel: 0.4

  acc_lim_x: 1.0 # maximum is theoretically 2.0, but we
  acc_lim_theta: 2.5
  acc_lim_y: 2.5      # diff drive robot

# Goal Tolerance Parameters
  yaw_goal_tolerance: 0.3  # 0.05
  xy_goal_tolerance: 0.15  # 0.10
  # latch_xy_goal_tolerance: false

# Forward Simulation Parameters
  sim_time: 1.0       # 1.7
  vx_samples: 6       # 3
  vy_samples: 1       # diff drive robot, there is only one sample
  vtheta_samples: 20  # 20

# Trajectory Scoring Parameters
  path_distance_bias: 64.0      # 32.0   - weighting for how much it should
stick to the global path plan
  goal_distance_bias: 24.0      # 24.0   - wighting for how much it should
attempt to reach its goal
  occdist_scale: 0.5               # 0.01    - weighting for how much the
controller should avoid obstacles
  forward_point_distance: 0.325  #  0.325    - how far along to place an
additional scoring point
  stop_time_buffer: 0.2         # 0.2    - amount of time a robot must stop
in before colliding for a valid traj.
  scaling_speed: 0.25            # 0.25    - absolute velocity at which to
start scaling the robot's footprint
  max_scaling_factor: 0.2        # 0.2     - how much to scale the robot's
footprint when at speed.

# Oscillation Prevention Parameters
  oscillation_reset_dist: 0.05  # 0.05   - how far to travel before resetting
oscillation flags

# Debugging
  publish_traj_pc : true
  publish_cost_grid_pc: true
  global_frame_id: odom
```

```
# Differential-drive robot configuration - necessary?
    #  holonomic_robot: false
```

El fichero move_base.launch.xml es el que se encarga de llamar al nodo de *move_base* junto con los parámetros creados y el control de la velocidad.

Finalmente, con todos los ficheros creados hasta ahora, el robot está capacitado para moverse por un mapa de forma autónoma. Por lo que creamos un fichero llamado move_base_raquel.launch, el cual incluye el lanzamiento de amcl_raquel.launch y de move_base.launch.xml.

De la misma forma que en apartados anteriores guardamos una configuración de RViz en donde podamos ver:

- El mapa.
- El láser.
- El modelo del robot.
- El mapa global y local.
- La trayectoria.
- La pose con covarianza.
- El array de partículas.

Al final del archivo creado incluimos la configuración guardada de RViz, de esta forma, cuando lanzamos el archivo:

roslaunch turtlebot_gazebo move_base_raquel.launch

Nos aparecerá por pantalla el entorno en Gazebo y la configuración de RViz guardada (ver figura 6.20).

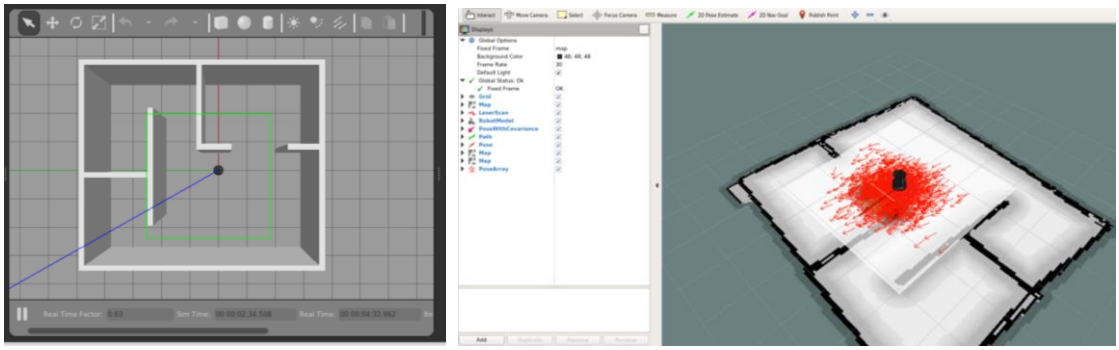

Figura 6.20. Pantallas abiertas después de lanzar el archivo move_base_raquel.launch.

Para la verificar que *move_base* funciona correctamente utilizamos la herramienta *2D Nav Goal* para indicarle que se dirija a un punto del mapa (ver figura 6.21). Después de pulsar *2D Nav Goal*, debemos pulsar otro punto del mapa, nos aparecerá una flecha que nos indica la orientación con la que queremos que se quede el robot una vez llegado al objetivo.

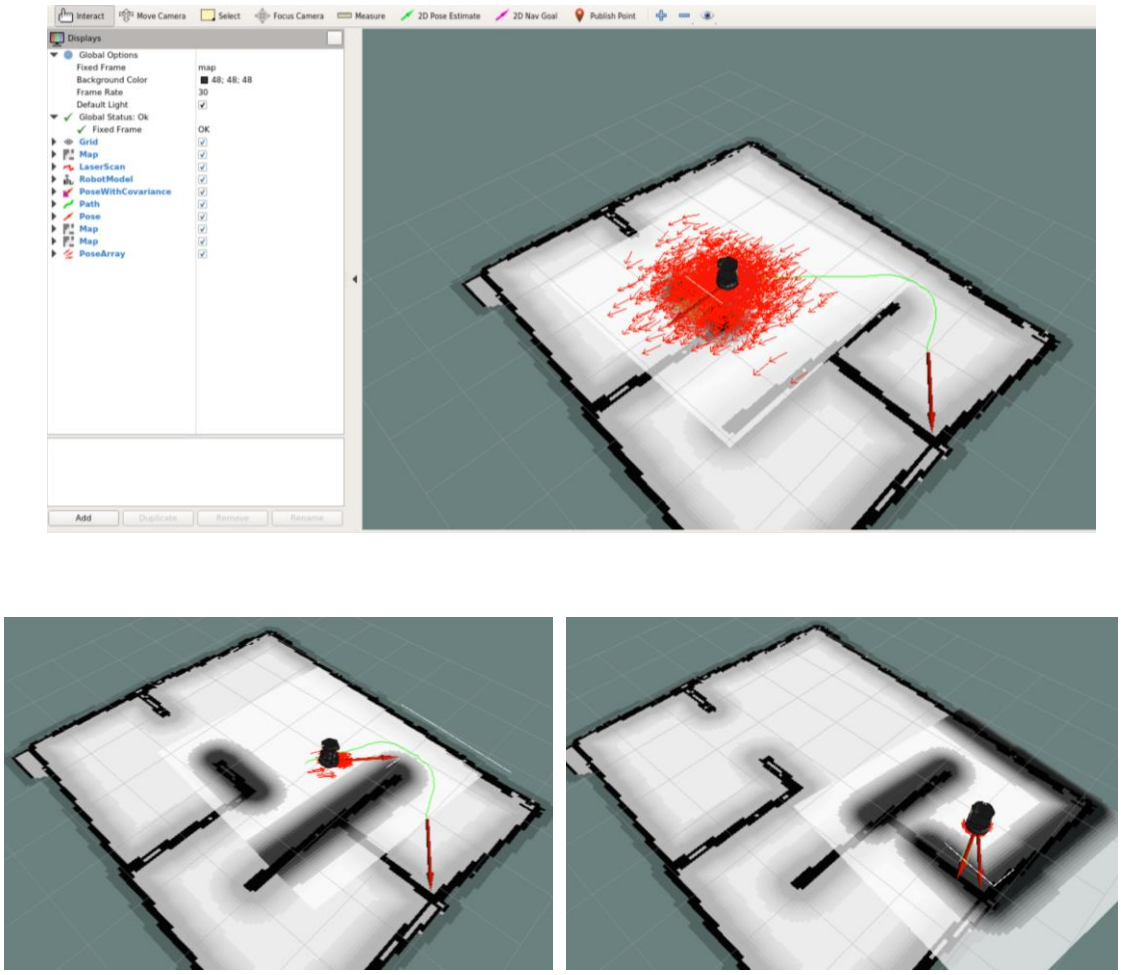

Figura 6.21. Comprobación del funcionamiento del nodo move_base mediante 2D Nav Goal.

En la Figura 6.21 podemos observar una sombra que se va volviendo más negra conforme más se aproxima a las paredes, esta sombra indica la proximidad a un obstáculo que, cuánto más cercano, más negra será. También nos fijamos en que la orientación final no es la misma que la seleccionada, pero se asemeja bastante debido a la tolerancia elegida.

Una vez comprobado que la planificación de trayectorias funciona correctamente, podemos probar con la evitación de obstáculos que aparecen en nuestra trayectoria. Desde Gazebo podemos poner una serie de obstáculos (ver figura 6.22) y verificar que el robot los va esquivando (ver figura 6.23). A continuación, se muestran los obstáculos colocados desde Gazebo:

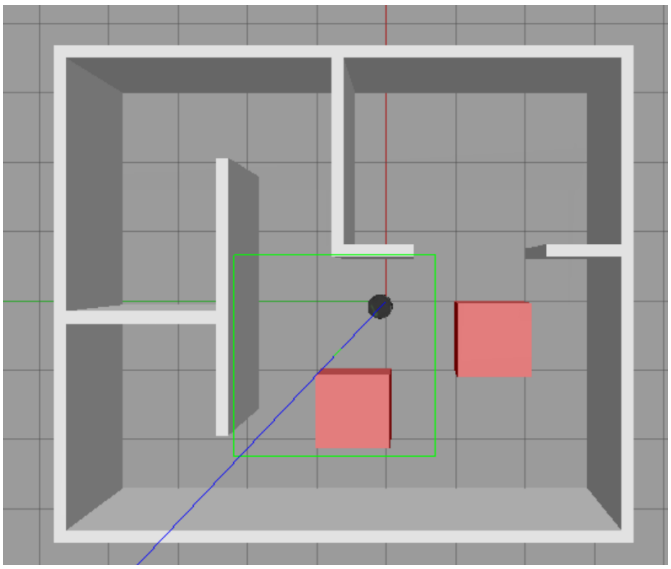

Figura 6.22. Obstáculos colocados en el entorno de Gazebo

Desde RViz seleccionamos un punto con *2D Nav Goal* para el cual tenga que atravesar los obstáculos:

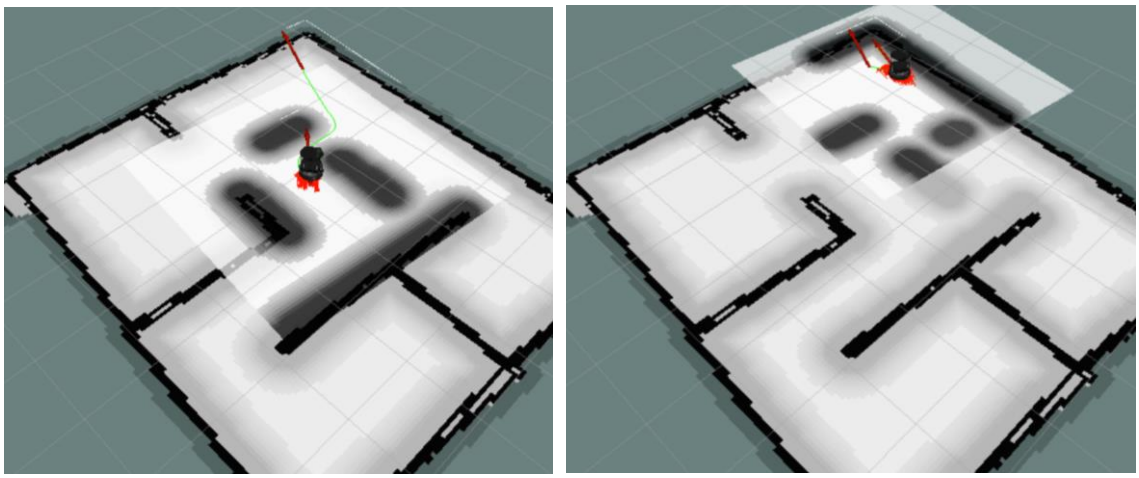

Figura 6.23. Verificación del esquivo de obstáculos en el entorno simulado.

Finalmente, podemos afirmar que nuestro robot en simulación está capacitado para la planificación de trayectorias y la evitación de obstáculos, por lo que podemos empezar a crear nodos de navegación personalizados.

6.4 Uso de los nodos de navegación

Creación de dependencias

En este apartado se exponen los dos nodos realizados para el uso de la navegación autónoma del robot por el entorno de Gazebo. Ambos necesitan los paquetes necesarios del

stack Navigation, por tanto, utilizaremos nuestros archivos creados en las simulaciones para la navegación del robot con los nodos creados.

Lo primero que debemos hacer antes de escribir nuestros nodos es crear un paquete nuevo con las dependencias necesarias o, directamente, escribirlas en nuestros archivos CMakeList.txt y package.xml del paquete ya creado. Las dependencias serán:

- Actionlib
- Move_base_msgs
- Roscpp

Si lo que queremos es crear un nuevo paquete:

Catkin_create_pkg simple_goals roscpp actionlib move_base_msgs

Si lo que queremos es incluir el nodo directamente en la carpeta *src* de nuestro paquete, debemos escribir en el archivo CMakeList.txt:

```
find_package(catkin REQUIRED COMPONENTS
actionlib
move_base_msgs
roscpp
)
```

Y en nuestro archivo package.xml:

```
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>actionlib</build_depend>
  <build_depend>move_base_msgs</build_depend>
  <build_depend>roscpp</build_depend>
  <build_export_depend>actionlib</build_export_depend>
  <build_export_depend>move_base_msgs</build_export_depend>
  <build_export_depend>roscpp</build_export_depend>
  <exec_depend>actionlib</exec_depend>
  <exec_depend>move_base_msgs</exec_depend>
  <exec_depend>roscpp</exec_depend>
```

Metodología

Para ambos nodos necesitamos conocer los puntos a los que queremos que el robot navegue de forma autónoma. Utilizaremos la herramienta de RViz *2D Pose Estimate* para posicionar el robot en el punto elegido y, para visualizar sus coordenadas escribiremos:

rostopic echo /amcl_pose

Por pantalla nos aparecerán las coordenadas en el mapa de la pose actual del robot, así como su orientación (ver figura 6.24). De esta forma, guardaremos los puntos que nos sean útiles para poder utilizarlos en nuestros nodos de navegación.

```
raquel@LAPTOP-TQ85PTKE:~$ rostopic echo /amcl_pose
header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 847000000
  frame_id: "map"
pose:
  pose:
    position:
      x: 0.0282609660838
      y: 0.0342560425716
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.000408754580512
      w: 0.99999991646
  covariance: [0.241583629087713, -0.002199722426769097, 0.0, 0.0, 0.0, 0.0, -0.002199722426769098
, 0.2389537876349595, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0678687137201339 8]
---
```

Figura 6.24. Salida por pantalla después de escribir rostopic echo /amcl_pose

Creación del ejecutable

Debemos tener en cuenta que una vez que se han creado los nodos, debemos agregarlos a nuestro archivo CMakeList.txt para que se construyan (ver figura 6.25).

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

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

Figura 6.25. Ejemplo de cómo añadir los nodos creados a CMakeList.txt.

6.4.1 Nodo de bienvenida

El nodo *bienvenida_node* realiza una ruta por el entorno enseñando las distintas zonas que lo componen. En este caso, la ruta empieza en la cocina, sigue por el salón y aseo y, finalmente acaba en la habitación.

Para poder ejecutar el nodo es necesario lanzar previamente el archivo donde ejecutamos los nodos de navegación move_base_raquel.launch y, en otra terminal, el nodo creado:

rosrun bienvenida bienvenida_node

Lo primero que nos aparecerá por pantalla es “BIENVENIDO” y a continuación iniciará la ruta hacia la cocina (ver figura 6.26).

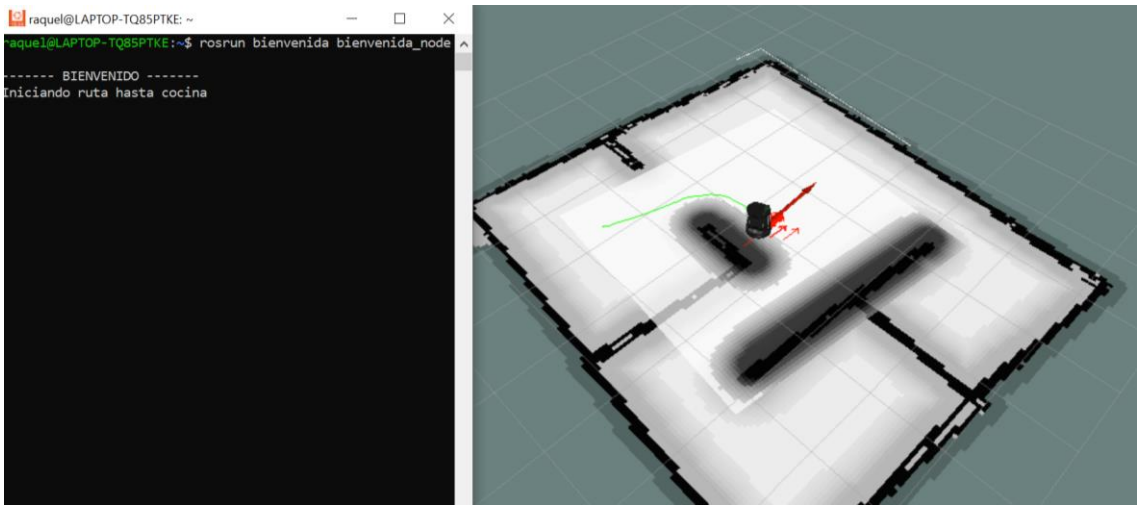

Figura 6.26. Salida por pantalla inicial del nodo bienvenida_node en simulación.

Cuando llegue a la cocina se parará y aparecerá por pantalla “Bienvenido a cocina”, a continuación, iniciará la ruta hacia el salón (ver figura 6.25).

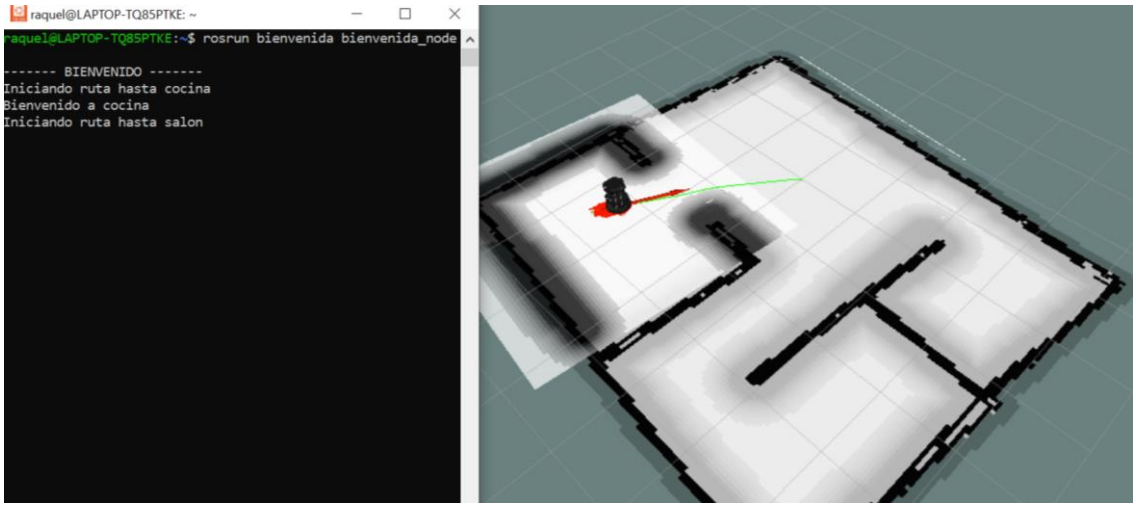

Figura 6.27. Primera parada del nodo bienvenida_node en simulación.

Una vez en el salón iniciará la ruta hacia el aseo (ver figura 6.28). Los mensajes por pantalla serán los mismos para todas las estancias.

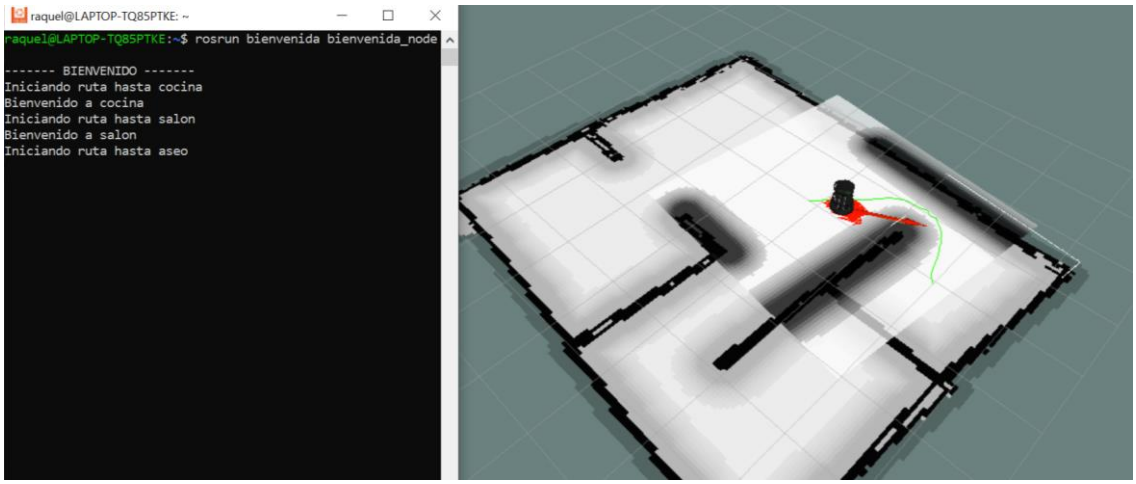

Figura 6.28. Segunda parada del nodo bienvenida_node en simulación.

Después de pararse en el aseo navegará hacia su último destino de la ruta, que es la habitación (ver figura 6.29).

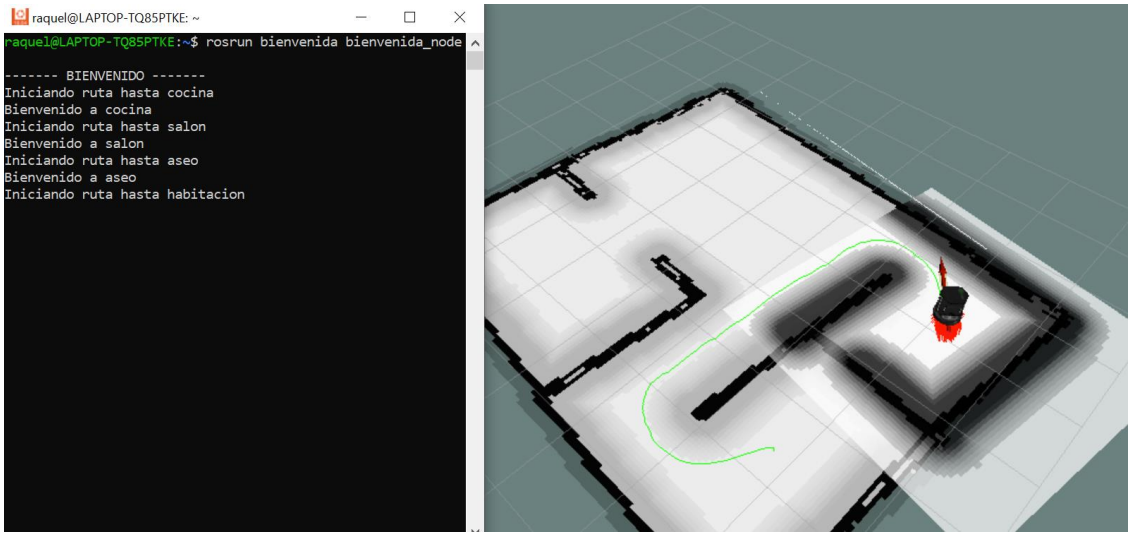

Figura 6.29. Tercera parada del nodo bienvenida_node en simulación.

Finalmente, se parará en la habitación y nos aparecerá por pantalla el mensaje: “La ruta ha finalizado” (ver figura 6.30).

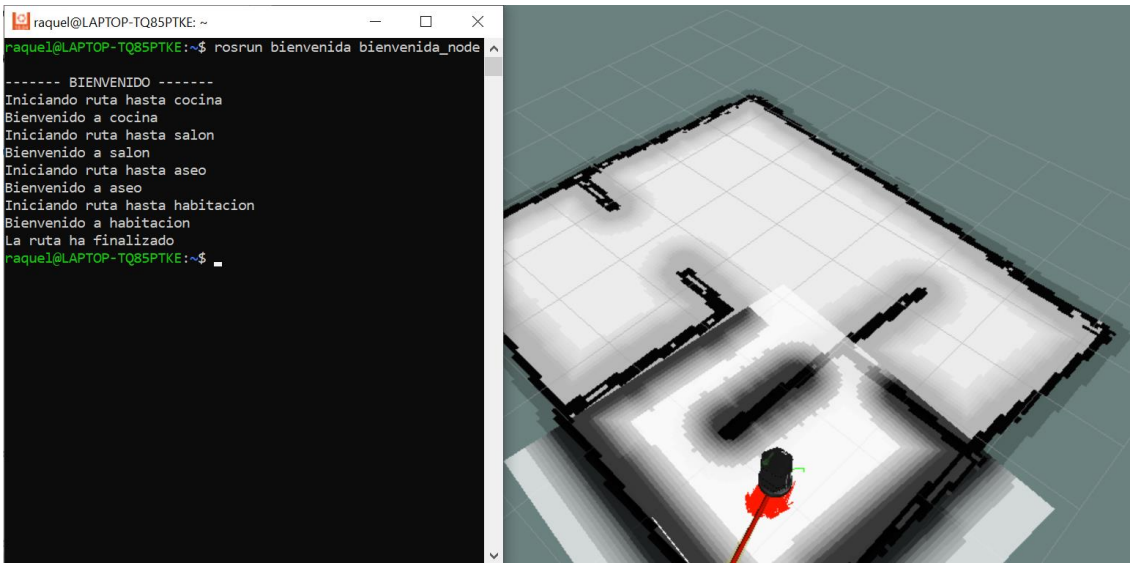

Figura 6.30. Útima para del nodo bienvenida_node en simulación y fin del nodo.

Estudiando el código

A continuación, se expone el código utilizado para el nodo de bienvenida llamado *bienvenida_node.cpp,* el cual se explica detalladamente más adelante.

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

typedef           actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;

std::vector<double> coordenadas(int e){
    std::vector<double> xy(3,0);
    switch(e){
        case 0:{
            xy[0] = 1.556275185;
            xy[1] = -1.22068588;}
            break;
        case 1:{
            xy[0] = -0.956734;
            xy[1] = 0.02313059;}
            break;
        case 2:{
            xy[0] = -0.93203659;
            xy[1] = 3.002267535;}
            break;
        case 3:{
            xy[0] = 1.638476542;
            xy[1] = 2.78161279;}
            break;

    }
    xy[2] = e;
    return xy;
}

void Move_goal(std::vector<double> &v){

  //llamamos a action server
  MoveBaseClient ac("move_base", true);
```

```
  //Esperamos a que action server responda
  while(!ac.waitForServer(ros::Duration(5.0))){
    std::cout << "Esperando a action server" << std::endl;
  }

  move_base_msgs::MoveBaseGoal goal;

  //Enviamos los objetivos

  //map para un punto del mapa, base_link para referenciar a partir de la base
  goal.target_pose.header.frame_id = "map";

  goal.target_pose.header.stamp = ros::Time::now();

  goal.target_pose.pose.position.x = v[0];
  goal.target_pose.pose.position.y = v[1];
  goal.target_pose.pose.position.z = 0.0;
  goal.target_pose.pose.orientation.w = 1.0;

  std::string lugar;
  if(v[2] == 0) lugar = {"cocina"};
  else if(v[2] == 1) lugar = {"salon"};
  else if(v[2] == 2) lugar = {"aseo"};
  else lugar = {"habitacion"};

  std::cout << "Iniciando ruta hasta " <<lugar<<std::endl;
  ac.sendGoal(goal);

  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
  std::cout << "Bienvenido a " <<lugar<<std::endl;
  else
  std::cout << "No se ha podido llegar a " <<lugar<<std::endl;
}

int main(int argc, char** argv){
  ros::init(argc, argv, "bienvenida_node");

    std::cout << "------- BIENVENIDO -------" << std::endl;
    int aux = 0;
    while(aux<4){
       std::vector<double> objetivo = coordenadas(aux);
        Move_goal(objetivo);
       aux++;
    }

    std::cout << "La ruta ha finalizado" << std::endl;
  return 0;
}
```

Vamos a repasar las líneas que componen el código:

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

En primer lugar, incluimos los archivos de cabecera necesarios, además de utilizar los mensajes de *move_base_msgs* y *actionlib*, que son los que nos permiten el movimiento de la base, también hemos incluido la librería *Vector* de C++.

```
typedef           actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;
```

Esta línea crea un *typedef* conveniente para un *SimpleActionClient* que nos permitirá comunicarnos con acciones que se adhieren a la interfaz de acción *MoveBaseAction*.

Vamos a saltarnos las funciones declaradas (que explicaremos más adelante) para ver el funcionamiento del código en sí con el *main*:

```
int main(int argc, char** argv){
  ros::init(argc, argv, "bienvenida_node");

    std::cout << "------- BIENVENIDO -------" << std::endl;
    int aux = 0;
    while(aux<4){
       std::vector<double> objetivo = coordenadas(aux);
        Move_goal(objetivo);
       aux++;
    }

    std::cout << "La ruta ha finalizado" << std::endl;
  return 0;
}
```

Lo primero será declarar el nodo, al que hemos llamado *bienvenida_node*, y mostraremos por pantalla el mensaje "BIENVENIDO". Declaramos la variable *aux*, cuyo valor inicial es 0, que utilizaremos para dar paso de una estancia a otra, es decir, la variable irá incrementándose y cada uno de estos valores estará asociado a una estancia de la casa. De esta forma, entraremos en un bucle que sólo se acabará cuando el valor de la variable supere el 3, e iremos llamando a las funciones *coordenadas()* y *Move_goal(),* que devolverán las coordenadas de la estancia y harán que el robot se mueva hacia ellas, respectivamente. Finalmente, avisaremos que la ruta se ha acabado y el *main* se acabará.

```
std::vector<double> coordenadas(int e){
    std::vector<double> xy(3,0);
    switch(e){
        case 0:{
            xy[0] = 1.556275185;
            xy[1] = -1.22068588;}
            break;
        case 1:{
            xy[0] = -0.956734;
            xy[1] = 0.02313059;}
            break;
        case 2:{
            xy[0] = -0.93203659;
            xy[1] = 3.002267535;}
            break;
        case 3:{
            xy[0] = 1.638476542;
            xy[1] = 2.78161279;}
            break;

    }
    xy[2] = e;
    return xy;
}
```

La función anterior es la encargada de devolver las coordenadas de la estancia a la que el robot deberá moverse. Esta función crea una variable local de tipo vector, el valor de este vector contendrá las coordenadas de la estancia y será el valor que devuelva la función. Tenemos 4 casos posibles en el *switch* dependiendo del valor de la variable *aux* mencionada anteriormente:

- Si *aux* = 0 → El vector que devuelve la función contendrá las coordenadas de la cocina, además del valor de *aux*.
- Si *aux* = 1 → El vector que devuelve la función contendrá las coordenadas del salón, además del valor de *aux*.
- Si *aux* = 2 → El vector que devuelve la función contendrá las coordenadas del aseo, además del valor de *aux*.
- Si *aux* = 3 → El vector que devuelve la función contendrá las coordenadas de la habitación, además del valor de *aux*.

No se considera el caso *default* debido a que el bucle del *main* no va a dar otra opción de valores.

La función *Move_goal()* es la que se encarga del movimiento de la base hacia el punto enviado por *coordenadas()*, la explicaremos línea por línea:

```
//llamamos a action server
MoveBaseClient ac("move_base", true);
```

Esta línea construye un cliente de acción que usaremos para comunicarnos con *move_base* que se adhiere a la interfaz *MoveBaseAction*. También le dice al cliente de acción que inicie un hilo para llamar a ros::spin() para que las devoluciones de llamada de ROS se procesen pasando "true" como segundo argumento del constructor *MoveBaseClient.*

```
//Esperamos a que action server responda
while(!ac.waitForServer(ros::Duration(5.0))){
  std::cout << "Esperando a action server" << std::endl;
}
```

Aquí esperamos a que *action server* esté listo para procesar objetivos.

```
move_base_msgs::MoveBaseGoal goal;

//Enviamos los objetivos

//map para un punto del mapa, base_link para referenciar a partir de la base
goal.target_pose.header.frame_id = "map";

goal.target_pose.header.stamp = ros::Time::now();

goal.target_pose.pose.position.x = v[0];
goal.target_pose.pose.position.y = v[1];
goal.target_pose.pose.position.z = 0.0;
goal.target_pose.pose.orientation.w = 1.0;

std::string lugar;
if(v[2] == 0) lugar = {"cocina"};
else if(v[2] == 1) lugar = {"salon"};
else if(v[2] == 2) lugar = {"aseo"};
else lugar = {"habitacion"};

std::cout << "Iniciando ruta hasta " <<lugar<<std::endl;
ac.sendGoal(goal);
```

En estas líneas creamos un objetivo para enviar a *move_base* usando el tipo de mensaje *move_base_msgs::MoveBaseGoal* que se incluye automáticamente con el encabezado *MoveBaseAction.h*. Le diremos a la base que avance hacia las coordenadas deseadas en el marco de coordenadas "map" y la llamada a *ac.sendGoal*() enviará el objetivo al nodo *move_base* para su procesamiento. Además, el vector recibido contiene un tercer valor asociado a las estancias de la casa, de esta forma, las podemos declarar como strings e informar por pantalla a dónde nos estamos dirigiendo.

```
  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
  std::cout << "Bienvenido a " <<lugar<<std::endl;
  else
  std::cout << "No se ha podido llegar a " <<lugar<<std::endl;
}
```

Lo único que queda por hacer ahora es esperar a que el objetivo termine usando la llamada *ac.waitForGoalToFinish* que se bloqueará hasta que la acción *move_base* termine de procesar el objetivo que le enviamos. Una vez que finaliza, podemos verificar si el objetivo tuvo éxito o falló y enviar un mensaje al usuario en consecuencia.

6.4.2 Navegación semántica

El segundo nodo creado, llamado *simple_navigation_goals_node*, nos permite movernos hasta la estancia deseada del entorno en Gazebo. Al igual que el nodo *bienvenida_node* asociaremos una serie de números a las distintas partes de la casa diseñada, y, según el número elegido, navegaremos hacia la estancia asociada a dicho número.

Para lanzar el nodo, primero ha de estar ejecutándose *move_base_raquel.launch* y, en otra terminal, el nodo creado:

rosrun simple_navigation_goals simple_navigation_goals_node

Por pantalla anunciará que está esperando un objetivo al que navegar y nos facilitará una serie de instrucciones indicando el número que tenemos que marcar según la estancia a la que queramos ir (ver figura 6.31). A continuación, una serie de ejemplos de la utilización del nodo *simple_navigation_goals:*

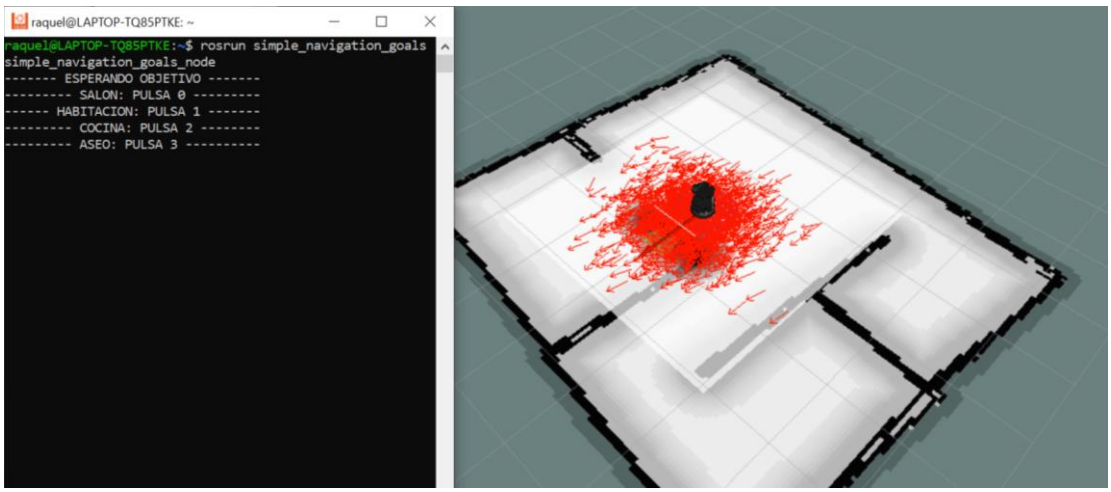

Figura 6.31. Salida por pantalla inicial del nodo simple_navigation_goals_node.

Pulsamos el 1, que está asociado a la habitación, e inmediatamente se dirigirá hasta dicha estancia (ver figura 6.32).

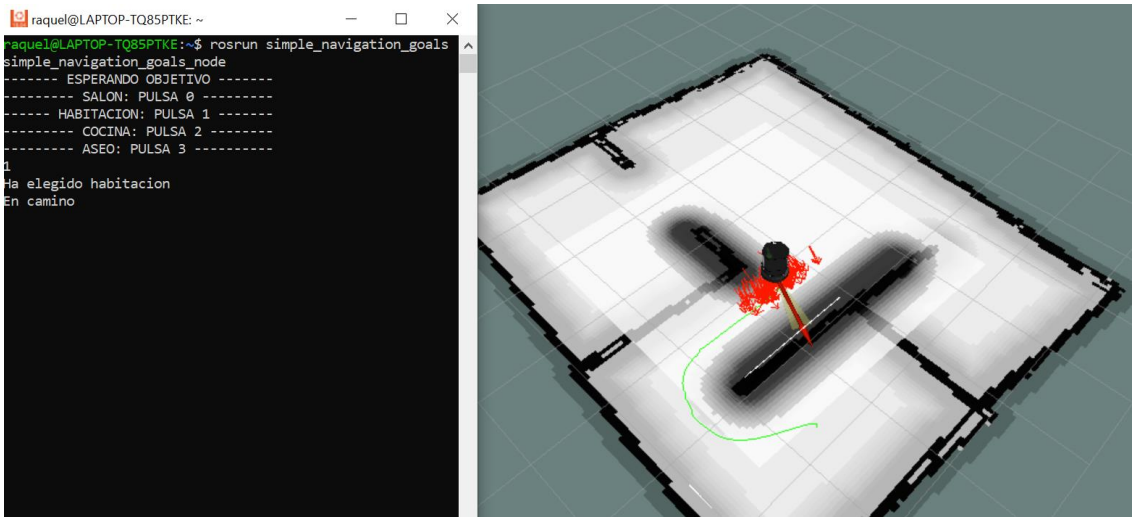

Figura 6.32. Salida por pantalla del nodo simple_navigation_goals_node después de pulsar la tecla 1.

Cuando llega a la estancia, nos avisará por pantalla y, además, nos preguntará si deseamos ir a otro objetivo, si la respuesta es sí, se marcará “y”, en caso contrario, se marcará “n” y el nodo finalizará (ver figura 6.33).

Figura 6.33. Salida por pantalla del nodo simple_navigation_goals_node después de llegar a su objetivo.

Si marcamos otra opción que no sea “y” ni “n” volverá a aparecer la pregunta tantas veces como no marquemos las opciones predeterminadas. Al marcar “y” el programa empezará de nuevo, y, de la misma forma, esperará a que marquemos la opción que contenga una estancia (ver figura 6.34).

Figura 6.34. Salida por pantalla del nodo simple_navigation_goals_node después de marcar opciones que no están predeterminadas en el nodo.

Finalmente, marcamos la tecla 3, el aseo, y una vez alcanzado el objetivo, cerramos el nodo pulsando “n” (ver figura 6.35).

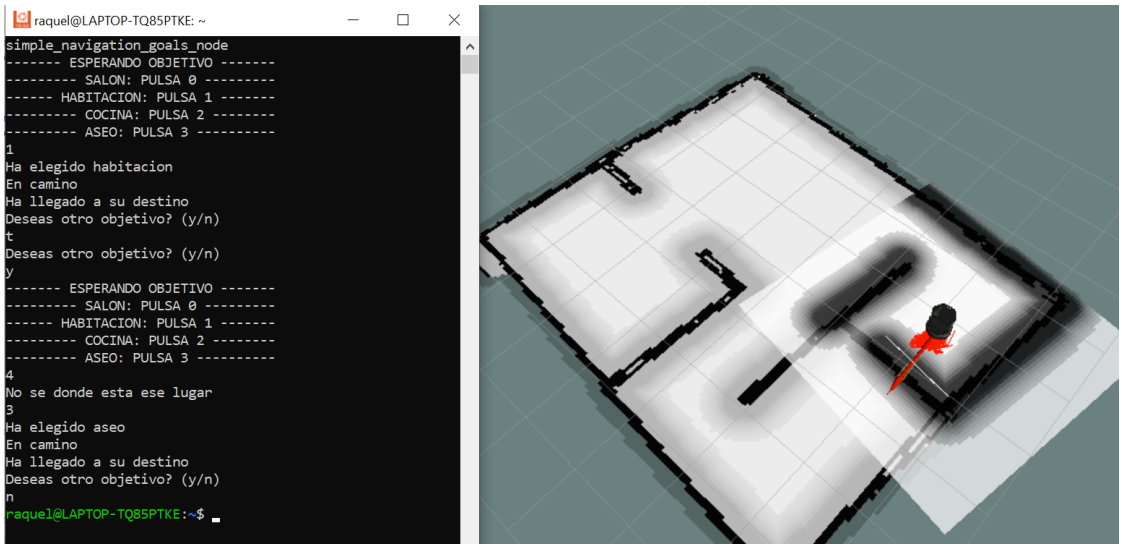

Figura 6.35. Cierre del nodo simple_navigation_goals_node

Estudiando el código

Para la ejecución del nodo *simple_navigation_goals_node* hemos creado el archivo *simple_navigation_goals_node.cpp*, el cual exponemos a continuación y explicamos más adelante:

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

typedef           actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;

std::vector<double> coordenadas(int e){
    std::vector<double> xy(2,0);
    switch(e){
        case 0:{
            std::cout << "Ha elegido salon" << std::endl;
            xy[0] = -0.956734;
            xy[1] = 0.02313059;}
            break;
        case 1:{
            std::cout << "Ha elegido habitacion" << std::endl;
            xy[0] = 1.638476542;
            xy[1] = 2.78161279;}
            break;
        case 2:{
            std::cout << "Ha elegido cocina" << std::endl;
            xy[0] = 1.556275185;
            xy[1] = -1.22068588;}
            break;
        case 3:{
            std::cout << "Ha elegido aseo" << std::endl;
            xy[0] = -0.93203659;
            xy[1] = 3.002267535;}
            break;

    }
    return xy;
}


void Move_goal(std::vector<double> &v){

  //llamamos a action server
  MoveBaseClient ac("move_base", true);

  //esperamos a que action server responda
  while(!ac.waitForServer(ros::Duration(5.0))){
       ROS_INFO("Esperando a action server");
  }

  move_base_msgs::MoveBaseGoal goal;

  //enviamos los objetivos
  //map para un punto del mapa, base_link para referenciar a partir de la base
  goal.target_pose.header.frame_id = "map";
  goal.target_pose.header.stamp = ros::Time::now();

  goal.target_pose.pose.position.x = v[0];
  goal.target_pose.pose.position.y = v[1];
  goal.target_pose.pose.position.z = 0.0;
  goal.target_pose.pose.orientation.w = 1.0;

  std::cout << "En camino" << std::endl;
  ac.sendGoal(goal);
```

```
  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    std::cout << "Ha llegado a su destino" << std::endl;
  else
    std::cout << "No ha podido llegar a su destino"<< std::endl;
}


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

  ros::init(argc, argv, "simple_navigation_goals_node");

    bool loop = true;

    do{

    std::cout << "------- ESPERANDO OBJETIVO -------" << std::endl;
    std::cout << "--------- SALON: PULSA 0 ---------" << std::endl;
    std::cout << "------ HABITACION: PULSA 1 -------" << std::endl;
    std::cout << "--------- COCINA: PULSA 2 --------" << std::endl;
    std::cout << "--------- ASEO: PULSA 3 ----------" << std::endl;

    int eleccion;
    char continuar;

    do{
       std::cin >> eleccion;
       if(eleccion != 0 && eleccion != 1 && eleccion != 2 && eleccion != 3)
              std::cout << "No se donde esta ese lugar" << std::endl;
    }

    while(eleccion != 0 && eleccion != 1 && eleccion != 2 && eleccion != 3);

    std::vector<double> objetivo = coordenadas(eleccion);
    Move_goal(objetivo);

    do{
    std::cout << "Deseas otro objetivo? (y/n)" << std::endl;
    std::cin >> continuar;
    }
    while(continuar != 'y' && continuar != 'n');

    if(continuar == 'y') loop = true;
    else loop = false;
    }

    while(loop);
  return 0;
}
```

Repasando el código:

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

En primer lugar, incluimos los archivos de cabecera necesarios, además de utilizar los mensajes de *move_base_msgs* y *actionlib*, que son los que nos permiten el movimiento de la base, también hemos incluido la librería *Vector* de C++.

```
typedef           actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;
```

Esta línea crea un *typedef* conveniente para un *SimpleActionClient* que nos permitirá comunicarnos con acciones que se adhieren a la interfaz de acción *MoveBaseAction*.

A continuación, explicaremos las líneas del *main* del programa, para más adelante explicar las funciones utilizadas.

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

    bool loop = true;

    do{

    std::cout << "------- ESPERANDO OBJETIVO -------" << std::endl;
    std::cout << "--------- SALON: PULSA 0 ---------" << std::endl;
    std::cout << "------ HABITACION: PULSA 1 -------" << std::endl;
    std::cout << "--------- COCINA: PULSA 2 --------" << std::endl;
    std::cout << "--------- ASEO: PULSA 3 ----------" << std::endl;

    int eleccion;
    char continuar;

    do{
       std::cin >> eleccion;
       if(eleccion != 0 && eleccion != 1 && eleccion != 2 && eleccion != 3)
              std::cout << "No se donde esta ese lugar" << std::endl;
    }

    while(eleccion != 0 && eleccion != 1 && eleccion != 2 && eleccion != 3);
```

En primer lugar, declaramos el nodo *simple_navigation_goals_node* y, a continuación, la variable *loop* de tipo *bool*, la cual será la encargada de repetir el main, siempre que el usuario decida navegar hacia otro lugar. A continuación, mostramos por pantalla las instrucciones para ir a una estancia y repetimos la entrada de la elección tantas veces como no se pulse las teclas acordes a las instrucciones.

```
    std::vector<double> objetivo = coordenadas(eleccion);
    Move_goal(objetivo);
```

Proseguimos con la llamada de las dos funciones; primero enviamos la elección a la función *coordenadas(),* que se encargará de devolver un vector con las coordenadas del punto perteneciente a la estancia deseada, y, después, las enviaremos a la función *Move_base()* para que mueva el robot hasta dichas coordenadas.

```
    do{
    std::cout << "Deseas otro objetivo? (y/n)" << std::endl;
    std::cin >> continuar;
    }
    while(continuar != 'y' && continuar != 'n');

    if(continuar == 'y') loop = true;
    else loop = false;
    }

    while(loop);
  return 0;
}
```

Finalmente, preguntará si se quiere continuar con la navegación. En caso afirmativo, la variable *loop* se mantendrá *true* y el primer *while* seguirá ejecutando el *main,* en caso negativo, la variable *loop* se hará *false,* por lo que se acabará el bucle y finalizará el *main*.

```
std::vector<double> coordenadas(int e){
    std::vector<double> xy(2,0);
    switch(e){
```

```
        case 0:{
            std::cout << "Ha elegido salon" << std::endl;
            xy[0] = -0.956734;
            xy[1] = 0.02313059;}
            break;
        case 1:{
            std::cout << "Ha elegido habitacion" << std::endl;
            xy[0] = 1.638476542;
            xy[1] = 2.78161279;}
            break;
        case 2:{
            std::cout << "Ha elegido cocina" << std::endl;
            xy[0] = 1.556275185;
            xy[1] = -1.22068588;}
            break;
        case 3:{
            std::cout << "Ha elegido aseo" << std::endl;
            xy[0] = -0.93203659;
            xy[1] = 3.002267535;}
            break;

    }
    return xy;
}
```

La función *coordenadas()* expuesta es la encargada de devolver los puntos del mapa pertenecientes a la estancia deseada. Según el valor de la variable *elección* escrita por el usuario, tenemos 4 casos:

- Elección = 0 → La función devuelve las coordenadas del salón.
- Elección = 1 → La función devuelve las coordenadas de la habitación.
- Elección = 2 → La función devuelve las coordenadas de la cocina.
- Elección = 3 → La función devuelve las coordenadas del aseo.

A continuación, explicamos línea a línea la función *Move_base(),* encargada del movimiento de la base hasta el punto enviado por la función *coordenadas().* La función es muy parecida a la utilizada con el mismo nombre en el nodo *bienvenida_node.*

```
//llamamos a action server
  MoveBaseClient ac("move_base", true);
```

Esta línea construye un cliente de acción que usaremos para comunicarnos con *move_base* que se adhiere a la interfaz *MoveBaseAction*. También le dice al cliente de acción que inicie un hilo para llamar a ros::spin() para que las devoluciones de llamada de ROS se procesen pasando "true" como segundo argumento del constructor *MoveBaseClient.*

```
  //Esperamos a que action server responda
  while(!ac.waitForServer(ros::Duration(5.0))){
       ROS_INFO("Esperando a action server");

  }
```

Aquí esperamos a que action server esté listo para procesar objetivos.

```
  move_base_msgs::MoveBaseGoal goal;

  //Enviamos los objetivos

  //map para un punto del mapa, base_link para referenciar a partir de la base
  goal.target_pose.header.frame_id = "map";

  goal.target_pose.header.stamp = ros::Time::now();

  goal.target_pose.pose.position.x = v[0];
  goal.target_pose.pose.position.y = v[1];
```

```
  goal.target_pose.pose.position.z = 0.0;
  goal.target_pose.pose.orientation.w = 1.0;

  std::cout << "En camino" << std::endl;
  ac.sendGoal(goal);
```

En estas líneas creamos un objetivo para enviar a *move_base* usando el tipo de mensaje *move_base_msgs::MoveBaseGoal* que se incluye automáticamente con el encabezado *MoveBaseAction.h*. Le diremos a la base que avance hacia las coordenadas deseadas en el marco de coordenadas "map" y la llamada a *ac.sendGoal*() enviará el objetivo al nodo *move_base* para su procesamiento.

```
  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    std::cout << "Ha llegado a su destino" << std::endl;
  else
    std::cout << "No ha podido llegar a su destino"<< std::endl;
}
```

Esperamos a que la navegación termine usando la llamada *ac.waitForGoalToFinish* que se bloqueará hasta que la acción *move_base* termine de procesar el objetivo que le enviamos. Una vez que finaliza, podemos verificar si el objetivo tuvo éxito o falló y enviar un mensaje al usuario en consecuencia.

7 PUESTA EN MARCHA DEL TURTLEBOT

7.1 Comunicación del PC con Turtlebot

Para la transferencia de datos entre nuestro ordenador y el NUC del robot se forma una red local mediante el router ADSL de Vodafone AD1018, pues es la solución más sencilla, ya que solo necesitamos el router y configurar las distintas IPs asociadas en el ordenador y el NUC.

La red se llamará *red_turtlebot* y su contraseña es *turtlebot2_inteligenciaambiental*. El robot y el PC tendrán asociada una IP estática, 192.168.10.154 y 192.168.10.10, respectivamente.

Para que PC y robot puedan comunicarse, es necesario editar el archivo *.bashrc* de ambos. Se tiene que especificar cuál es el master y cuál el hostname; el máster será el Turtlebot y el hostname cambiará dependiendo del *.bashrc* que estemos editando:

- Para el PC: *export ROS MASTER URI=http://192.168.10.154:11311*

  export ROS_IP=192.168.10.10

- Para el robot: *export ROS MASTER URI=http://192.168.10.154:11311*

  export ROS_IP=192.168.10.154

Para verificar la conexión entre ambos, podemos hacer “ping” desde el Turtlebot o desde el PC (ver figuras 7.1 y 7.2), este comando nos permitirá saber si ambos están intercambiando datos.

Desde turtlebot: *ping 192.168.10.10*

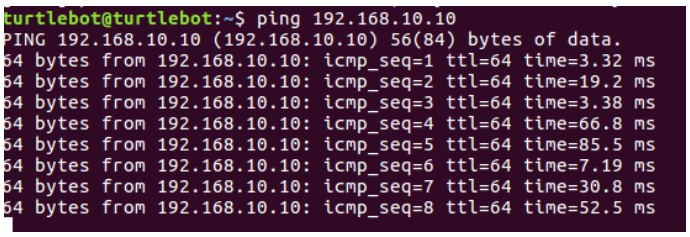

Figura 7.1. Salida por pantalla después de hacer "ping" desde el turtlebot.

Desde PC: *ping 192.168.10.154*

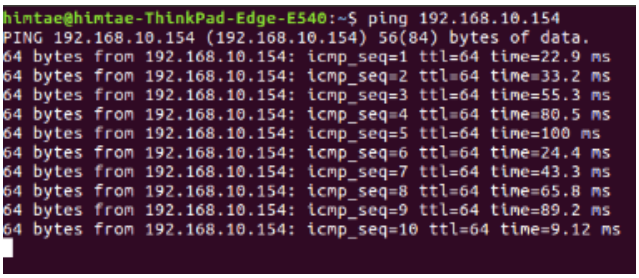

Figura 7.2. Salida por pantalla después de hacer "ping" desde el PC.

Para poder acceder de forma remota al turtlebot se instala el servidor SSH:

sudo apt-get install openssh-server

Para ejecutar el SSH desde el PC se utiliza la estructura *ssh nombre_turtlebot@IP_turtlebot*. En este caso:

ssh turtlebot@192.168.10.154

Nos pedirá la contraseña del turtlebot, que es *ros* y, una vez escrita, estaremos dentro del turtlebot (ver figura 7.3).

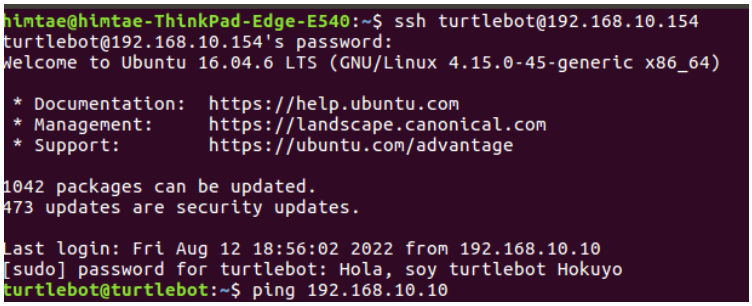

Figura 7.3. Ejecución del comando ssh.

7.2 Comandos básicos

7.2.1 Inicialización básica

Antes utilizar cualquier nodo, hay que inicializar el robot. Para hacerlo de la forma más simple, lanzamos un launch llamado minimal.launch del paquete turtlebot_bringup en el usuario turtlebot:

roslaunch turtlebot_bringup minimal.launch

La indicación de que la inicialización ha salido bien es un sonido en forma ascendente en el robot, en caso contrario, será descendente. También, podemos hacer *rostopic list* desde el PC y veremos los topics que está publicando turtlebot (ver figura 7.4).

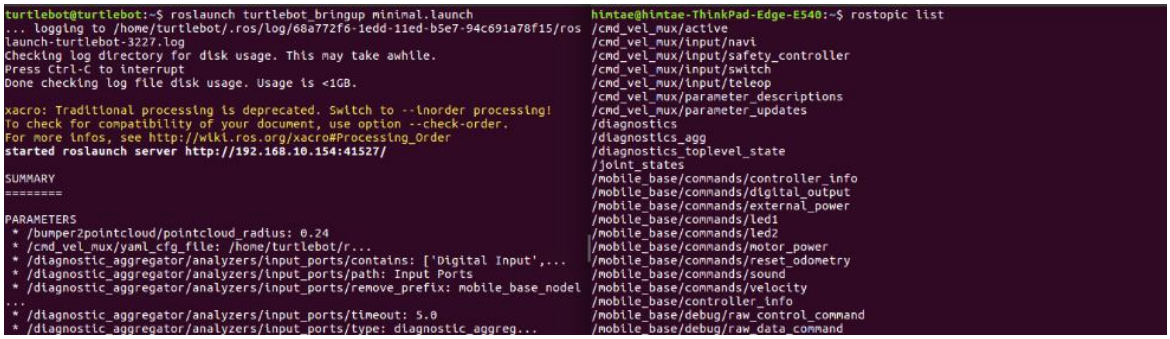

Figura 7.4. Visualización de los topics activos del turtlebot desde el PC una vez inicializado el turtlebot.

Ejecutando este comando nos aparecerá un siguiente error (ver figura 7.5), pero no afectará al correcto funcionamiento del robot:

```
[ERROR] [1660817426.112111393]: Kobuki : malformed sub-payload detected. [199][170][C7 AA 55 4D 01 0F B0 0C 00 00 ]
[ERROR] [1660817431.198909362]: Kobuki : malformed sub-payload detected. [140][170][8C AA 55 4D 01 0F 9C 20 00 00 ]
[ERROR] [1660817432.216415147]: Kobuki : malformed sub-payload detected. [35][170][23 AA 55 4D 01 0F 98 24 00 00 00 00 ]
```

Figura 7.5. Error al inicializar el turtlebot con minimal.launch.

7.2.2 Inicialización del láser Hokuyo

Debido a que los paquetes descargados en Github del Turtlebot no contienen el láser Hokuyo, es necesario descargarlo por separado. Será necesario la instalación de los siguientes paquetes disponibles en ROS Kinetic: *urg_node, urg_c* y *laser_proc*.

Una vez hecha la instalación, entramos en nuestra carpeta *src* y descargamos los siguientes paquetes:

git clone https:/github.com/ros-drivers/urg_node

git clone https:/github.com/ros-drivers/urg_c

git clone https:/github.com/ros-drivers/laser_proc

Es muy importante editar la IP en el archivo urg_lidar.launch dentro del paquete *urg_node* (ver figura 7.6). La nueva IP será 192.168.0.10, siendo esta la dirección asociada al láser.

```
<launch>

<!-- A simple launch file for the urg_node package. -->

<!--  When using an IP-connected LIDAR, populate the "ip_address" parameter with the address of the LIDAR.
      Otherwise, leave it blank. If supported by your LIDAR, you may enable the publish_intensity
      and/or publish_multiecho options. -->
 <node pkg="tf" type="static_transform_publisher" name="link1_broadcaster" args="0.122 0 0.0822 0 0 0 base_link scan 100" />

  <node name="urg_node" pkg="urg_node" type="urg_node" output="screen">
    <param name="ip_address" value="192.168.0.10"/>
    <param name="serial_port" value=""/>
    <param name="serial_baud" value="115200"/>
    <param name="frame_id" value="scan"/>
    <param name="calibrate_time" value="true"/>
    <param name="publish_intensity" value="false"/>
    <param name="publish_multiecho" value="false"/>
    <param name="angle_min" value="-1.5707963"/>
    <param name="angle_max" value="1.5707963"/>
  </node>

</launch>
```

Figura 7.6. Archivo urg_lidar.launch.

Una vez realizados los pasos anteriores será posible inicializar el láser correctamente. Para lanzarlo se debe utilizar el siguiente comando en turtlebot:

roslaunch urg_node urg_lidar.launch

Si se quiere comprobar que, efectivamente se están recogiendo los datos del láser, basta con hacer el siguiente comando en himtae o en turtlebot:

rostopic echo /scan

De esta forma, se puede ver en pantalla un flujo de datos, correspondientes a las mediciones del láser Hokuyo (ver figura 7.7).

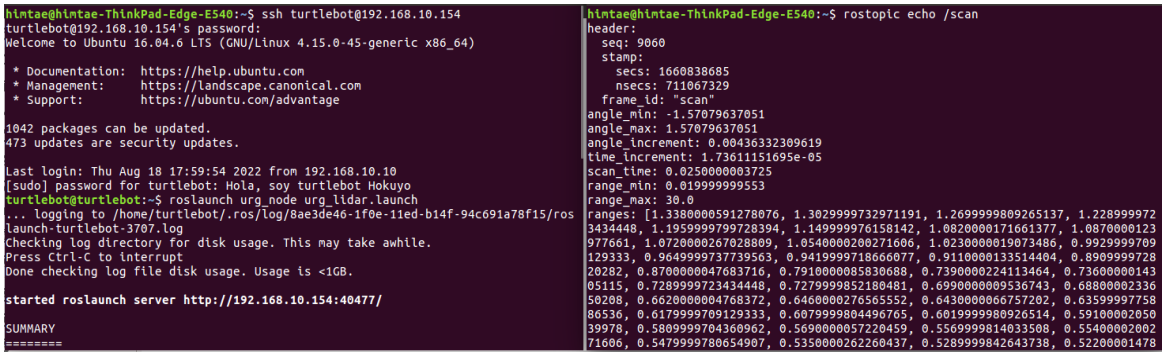

Figura 7.7. Flujo de datos pertenecientes al topic /scan procedentes del láser Hokuyo, inicializado previamente con urg_lidar.launch.

7.2.3 Teleoperación

En la carpeta de turtlebot existe un paquete llamado turtlebot_teleop que contiene archivos para realizar la teleoperación desde el teclado o desde mandos de la Xbox o ps3, entre otros.

En nuestro caso se utiliza únicamente la teleoperación desde el teclado:

roslaunch turtlebot_teleop keyboard_teleop.launch

La trayectoria del robot se puede controlar mediante velocidades lineales utilizando las teclas “i”, “,”, “j”, “l” o mediante velocidades angulares con las teclas “u”, “.”, “o”, “m”. La tecla “k” parará al robot (ver figura 7.8).

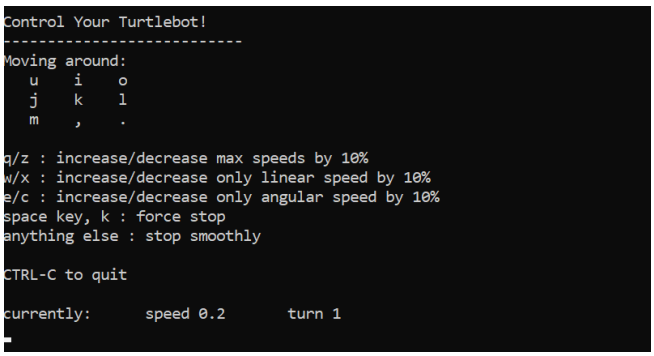

Figura 7.8. Salida por pantalla al ejecutar el archivo keyboard_teleop.launch.

8 Mapeado y navegación en un entorno real

Los pasos para el mapeado y la navegación de un entorno real se asemejan en gran medida a los pasos realizados para la simulación, la diferencia es la inclusión del láser Hokuyo, necesario para la toma de medidas. De esta forma, tendremos una gran similitud entre los archivos de la simulación y los del entorno real, con la excepción de que, en los últimos, en vez de tener un láser ficticio, incluimos la llamada y las características del láser Hokuyo.

8.1 Mapeado del entorno

El entorno a mapear será una casa con cuatro estancias: salón (ver figura 8.1), cocina (ver figura 8.3), aseo y habitación (ver figura 8.4). La posición inicial del robot será el salón (ver figura 8.2). A continuación, se muestran las cuatro estancias, el pasillo y la posición inicial del robot:

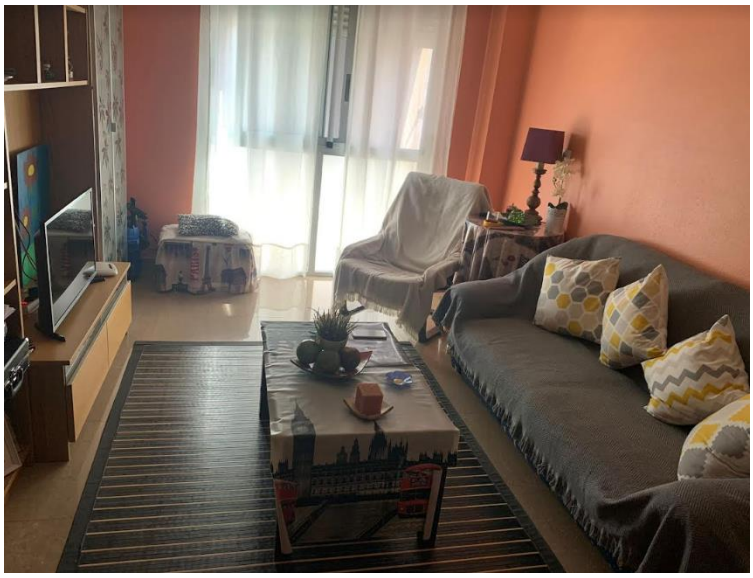

Figura 8.1. Salón del entorno real.

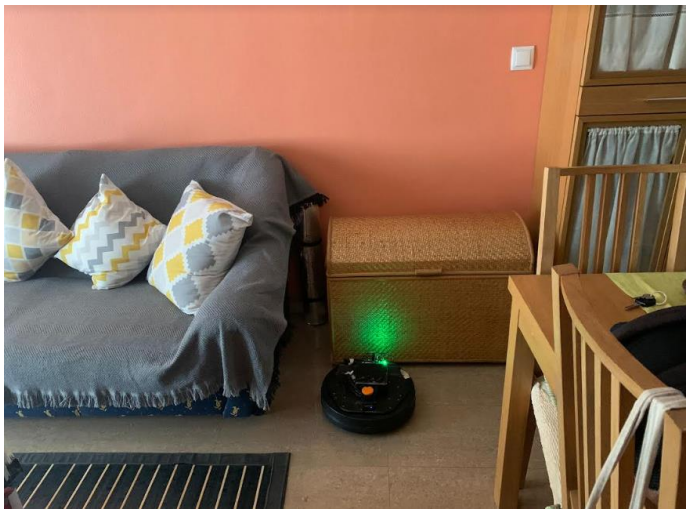

Figura 8.2. Posición inicial del robot en el salón del entorno real.

Figura 8.3. De izquierda a derecha: Cocina y pasillo del entorno real.

Figura 8.4. De izquierda a derecha: habitación y aseo del entorno real.

El primer paso es crear el paquete turtlebot_gmapping en nuestro *src*. El paquete contendrá los archivos necesarios para el mapeo, así como los mapas o las configuraciones de RViz necesarias. Dispondremos de un archivo que inicialice el láser y realice el mapeo utilizando el paquete de *gmapping*, y, otro, que se encargue de la visualización y la teleoperación:

- Gmapping_laser.launch ➔ Se lanza desde turtlebot. Es el encargado del mapeo. En él incluimos la inicialización del láser Hokuyo con el *urg_node* y el lanzamiento del nodo *slam_gmapping* customizado para el láser (ver figura 8.5). Esta adaptación del láser se encuentra en la carpeta *include* del paquete turtlebot_gmapping, se llama lidar.launch.xml.

  roslaunch turtlebot_gmapping gmapping_laser.launch

- View_teleop_mapeo_kobuki.launch ➔ Se lanza desde himtae. Este archivo permite observar cómo se forma el mapa en RViz. También incluye la teleoperación del robot (ver figura 8.6).

  roslaunch turtlebot_gmapping view_teleop_mapeo_kobuki.launch

```
<launch>
  <!-- Laser -->
  <arg name="3d_sensor" default="lidar"/>  <!-- r200, kinect, asus_xtion_pro -->
  <include file="$(find urg_node)/launch/urg_lidar.launch"/>

  <!-- Gmapping -->
  <arg name="custom_gmapping_launch_file" default="$(find turtlebot_gmapping)/include/$(arg 3d_sensor).launch.xml"/>
  <include file="$(arg custom_gmapping_launch_file)"/>

</launch>
```

Figura 8.5. Archivo gmapping_laser.launch

```
<launch>
  <!-- turtlebot_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key" name="turtlebot_teleop_keyboard"  output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="cmd_vel_mux/input/teleop"/>
  </node>

  <include file="$(find turtlebot_gmapping)/launch/view_mapeo_kobuki.launch"/>
</launch>
```

Figura 8.6. Archivo View_teleop_mapeo_kobuki.launch

Cuando comenzamos la teleoperación comprobamos como el mapa se va creando (ver figura 8.7). Y, después de recorrer las estancias de la casa múltiples veces, el mapa estará listo para ser guardado (ver figura 8.8).

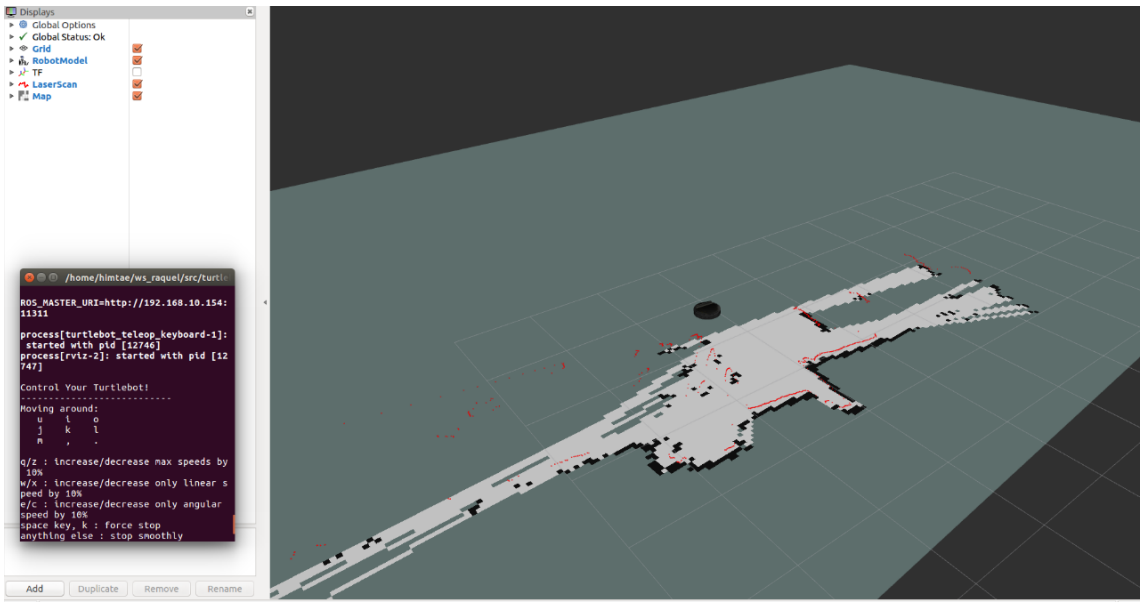

Figura 8.7. Vista desde RViz de la creación del mapa en el entorno real mediante la teleoperación.

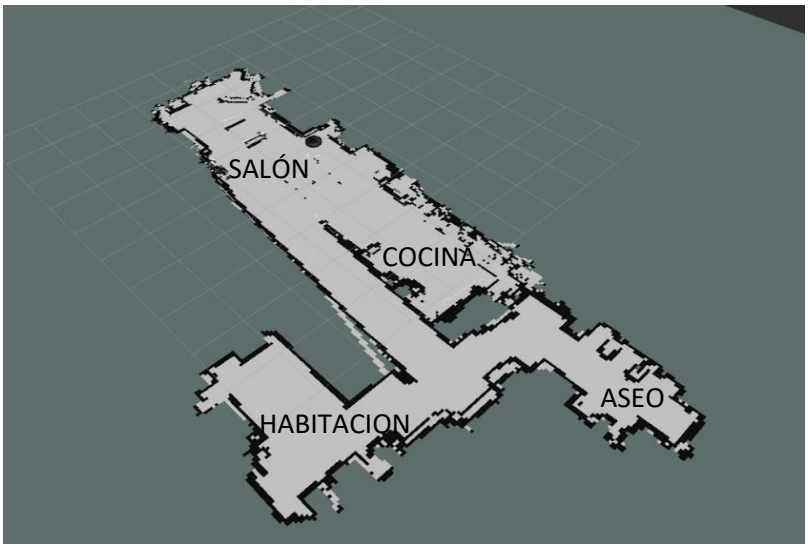

Figura 8.8. Mapa final del entorno real.

Seguimos los mismos pasos que en la simulación para guardar el mapa; ejecutamos el nodo *map_saver* del paquete *map_server* en la carpeta mapas.

Nuevamente se habrán creado dos archivos:mapa_casa.pgm y mapa_casa.yaml. Ya sabemos que el primero será una imagen y, el segundo, contendrá la serie de parámetros que definen el mapa.

De la misma forma que en la simulación, cuando se necesite utilizar el mapa creado, ejecutaremos el nodo *map_server* del paquete *map_server*:

rosrun map_server map_server mapa_casa.yaml

8.2 Navegación por el entorno

Para la navegación, como hemos visto anteriormente son fundamentales el paquete *amcl* y el *move_base* de la pila de navegación.

Creamos el paquete turtlebot_navegacion dentro de nuestro *src*. El paquete contendrá los ficheros más simples de navegación como son el *amcl* o el *move_base* personalizados y, además, los nodos creados para diferentes aplicaciones. También contendrá las configuraciones de RViz y los mapas.

Para simplificar la inicialización del robot, se creará un nuevo archivo en el paquete turtlebot_bringup que contenga el lanzamiento mínimo del minimal.launch, el láser y el raq_navegacion.launch, explicado más adelante. De esta forma nos ahorramos abrir los múltiples archivos uno por uno (ver figura 8.9).

```
<launch>
<include file="$(find turtlebot_bringup)/launch/minimal.launch"/>
<include file="$(find urg_node)/launch/urg_lidar.launch"/>
<include file="$(find turtlebot_navegacion)/launch/raq_navegacion.launch"/>
</launch>
```

Figura 8.9. Archivo hokuyo_navegacion.launch para inicializar el robot, el láser y el stack de navegación.

8.2.1 Localización

Como ya se ha mencionado, utilizaremos el paquete *amcl* para la localización del robot en el entorno.

Necesitaremos:

- El nodo *map_server* publicando el mapa.
- La teleoperación para poder movernos por el mapa.
- Los parámetros imprescindibles para el correcto funcionamiento del AMCL.
- RViz con la información del láser, el modelo del robot, el mapa, la pose con covarianza y array de partículas.

A partir de estas necesidades, creamos dos archivos .launch en nuestra carpeta turtlebot_navegacion:

- Raq_amcl.launch ➔ Lo lanzaremos desde turtlebot. El archivo contiene la publicación del mapa, la teleoperación y los parámetros del *amcl* (ver figura 8.10). Estos últimos los podemos encontrar en la carpeta de *include* en el archivo amcl.launch.xml. Este último es el archivo utilizado también para las simulaciones.
- Amcl_rviz.launch ➔ Lo abrimos desde himtae y nos facilita la visualización del entorno mediante RViz con las informaciones necesarias del robot (ver figura 8.11).

```
<launch>

  <!-- Map server -->
  <arg name="map_file" default="$(find turtlebot_gmapping)/mapas/mapa_casa.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" respawn="true" />

  <!-- AMCL -->
  <arg name="custom_amcl_launch_file" default="$(find turtlebot_navegacion)/include/amcl.launch.xml"/>
  <arg name="initial_pose_x" default="0.0"/> <!-- Use 17.0 for willow's map in simulation -->
  <arg name="initial_pose_y" default="0.0"/> <!-- Use 17.0 for willow's map in simulation -->
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(arg custom_amcl_launch_file)">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <!-- Teleop -->
   <node pkg="turtlebot_teleop" type="turtlebot_teleop_key" name="turtlebot_teleop_keyboard"  output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="cmd_vel_mux/input/teleop"/>
  </node>

</launch>
```

Figura 8.10. Archivo raq_amcl.launch.

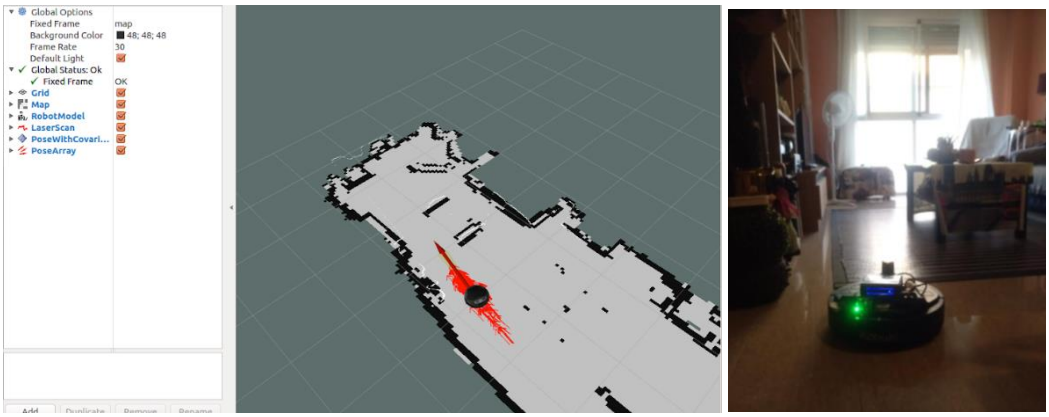

Figura 8.11. Imagen de RViz una vez lanzado raq_amcl.launch y amcl_rviz.launch y posición real del robot.

8.2.2 Planificación

Como ya hemos visto, para las tareas de planificación utilizaremos el paquete de *move_base*. En primer lugar, debemos crear una carpeta llamada param que contendrá los archivos en los que se fundamenta el *move_base*: Raq_costmap_common_params.yaml, local_costmap_params.yaml, global_costmap_params.yaml, raq_dwa_local_planner_params.yaml y navfn_global_planner_params.yaml.

Al utilizar los parámetros de la simulación en un entorno real, nos encontramos con una serie de problemas que no teníamos en simulación. Esto es normal, puesto que en la realidad nos encontramos con más factores que se escapan de nuestro alcance en una simulación, además, el entorno real es bastante más estrecho, lo que ocasiona problemas añadidos. Por este motivo, se ajustan algunos valores de nuestros archivos, los cuales se explican a continuación.

Raq_costmap_common_params.yaml

Del archivo donde definimos los parámetros comunes, hemos cambiado los siguientes parámetros:

- ***Min_obstacle_height*** → El cambio más significativo es la modificación del *min_obstacle_height* del láser. Sin este cambio era imposible la detección de obstáculos debido a que la transformación láser → base_link no tenía en cuenta la altura del láser.
- ***Obstacle_range y raytrace_range*** → Hemos modificado la distancia máxima a la que los obstáculos entran en el mapa de costos (*obstacle_range*) y la distancia a la que limpia los obstáculos que ya no están en el mapa (*raytrace_range*) debido a que, al ser un entorno tan pequeño y tener el *obstacle_range* mayor que el *raytrace_range*, con los obstáculos que podían aparecer a lo lejos en el camino para después desaparecer, el robot no podía trazar una ruta alternativa y daba lugar a muchos fallos.
- ***Map_type*** → Cambiamos el tipo de mapa, de voxel a costmap puesto que estamos en 2D, aunque realmente, para los parámetros, no influye.
- ***Inflation_radius*** → También se ha cambiado la distancia a los objetos que influyen en el cambio de ruta (*inflation_radius*) debido a algunos errores en la planificación de la ruta, en la que se acercaba mucho a los marcos de las puertas y las paredes.

A continuación, el archivo raq_costmap_common_params.yaml ya modificado:

```
max_obstacle_height: 0.60  # assume something like an arm is mounted on top
of the robot

# Obstacle Cost Shaping (http://wiki.ros.org/costmap_2d/hydro/inflation)
robot_radius: 0.18   # distance a circular robot should be clear of the
obstacle (kobuki: 0.18)
# footprint: [[x0, y0], [x1, y1], ... [xn, yn]]   # if the robot is not
circular

map_type: costmap

obstacle_layer:
  enabled:              true
  max_obstacle_height:  0.6
  origin_z:             0.0
  z_resolution:         0.2
  z_voxels:             2
  unknown_threshold:    15
  mark_threshold:       0
  combination_method:   1
  track_unknown_space:   true      #true needed for disabling global path
planning through unknown space
  obstacle_range: 1
  raytrace_range: 1.5
  origin_z: 0.0
  z_resolution: 0.2
  z_voxels: 2
  publish_voxel_map: false
  observation_sources:  scan bump
  scan:
    data_type: LaserScan
    topic: scan
    marking: true
    clearing: true
    min_obstacle_height: 0.0
    max_obstacle_height: 0.15
  bump:
    data_type: PointCloud2
    topic: mobile_base/sensors/bumper_pointcloud
    marking: true
    clearing: false
    min_obstacle_height: 0.0
    max_obstacle_height: 0.15
  # for debugging only, let's you see the entire voxel grid

#cost_scaling_factor   and   inflation_radius   were   now   moved   to   the
inflation_layer ns
inflation_layer:
  enabled:              true
  cost_scaling_factor:  9.0  # exponential rate at which the obstacle cost
drops off (default: 10)
  inflation_radius:    0.55  # max. distance from an obstacle at which costs
are incurred for planning paths.

static_layer:
  enabled:              true
```

Local_costmap_params.yaml

En el archivo donde definimos los parámetros para la planificación local, hemos modificado la altura y anchura (*width* y *height*) de la ventana del mapa local puesto que al ser una casa tan pequeña y tener la distancia máxima a objetos de 1 m, no era necesario más. El archivo modificado es el siguiente:

```
local_costmap:
```

```
   global_frame: odom
   robot_base_frame: /base_footprint
   update_frequency: 5.0
   publish_frequency: 2.0
   static_map: false
   rolling_window: true
   width: 3.0
   height: 3.0
   resolution: 0.05
   transform_tolerance: 0.5
   plugins:
    - {name: obstacle_layer,      type: "costmap_2d::VoxelLayer"}
    - {name: inflation_layer,     type: "costmap_2d::InflationLayer"}
```

Raq_dwa_local_planner_params.yaml

En el archivo donde definimos los parámetros para el planificador de trayectorias, las modificaciones han sido básicamente mediante prueba y error, comprobando las distintas trayectorias que podía hacer el robot con una configuración u otra. El archivo es el siguiente:

```
DWAPlannerROS:

# Robot Configuration Parameters - Kobuki
  max_vel_x: 0.5  # 0.55
  min_vel_x: 0.1

  max_vel_y: 0.5  # diff drive robot
  min_vel_y: 0.0  # diff drive robot

  max_trans_vel: 0.5 # choose slightly less than the base's capability
  min_trans_vel:  0.1   # this is the min trans velocity when there is
negligible rotational velocity
  trans_stopped_vel: 0.1

  # Warning!
  #    do not set min_trans_vel to 0.0 otherwise dwa will always think
translational velocities
  #   are non-negligible and small in place rotational velocities will be
created.

  max_rot_vel: 5.0  # choose slightly less than the base's capability
  min_rot_vel:  -0.5   # this is the min angular velocity when there is
negligible translational velocity
  rot_stopped_vel: 0.4

  acc_lim_x: 1.0 # maximum is theoretically 2.0, but we
  acc_lim_theta: 2.5
  acc_lim_y: 0.0      # diff drive robot

# Goal Tolerance Parameters
  yaw_goal_tolerance: 0.3  # 0.05
  xy_goal_tolerance: 0.15  # 0.10
  # latch_xy_goal_tolerance: false

# Forward Simulation Parameters
  sim_time: 1.0       # 1.7
  vx_samples: 6       # 3
  vy_samples: 1       # diff drive robot, there is only one sample
  vtheta_samples: 20  # 20

# Trajectory Scoring Parameters
  path_distance_bias: 64.0      # 32.0   - weighting for how much it should
stick to the global path plan
  goal_distance_bias: 24.0      # 24.0   - wighting for how much it should
attempt to reach its goal
```

```
    occdist_scale: 0.5               # 0.01    - weighting for how much the
  controller should avoid obstacles
    forward_point_distance: 0.325  #  0.325   - how far along to place an
  additional scoring point
    stop_time_buffer: 0.2          # 0.2     - amount of time a robot must stop
  in before colliding for a valid traj.
    scaling_speed: 0.25             # 0.25    - absolute velocity at which to
  start scaling the robot's footprint
    max_scaling_factor: 0.2        # 0.2     - how much to scale the robot's
  footprint when at speed.

  # Oscillation Prevention Parameters
    oscillation_reset_dist: 0.05  # 0.05   - how far to travel before resetting
  oscillation flags

  # Debugging
    publish_traj_pc : true
    publish_cost_grid_pc: true
    global_frame_id: odom


  # Differential-drive robot configuration - necessary?
#  holonomic_robot: false
```

Finalmente, estos parámetros junto con la modulación de velocidad se incluirán en el archivo move_base.launch.xml, y, este último se incluirá en raq_navegacion.launch junto con el lanzamiento del mapa y el AMCL (ver figura 8.12).

```
<launch>

  <!-- Map server -->
  <arg name="map_file" default="$(find turtlebot_gmapping)/mapas/mapa_casa.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" respawn="true" />

  <!-- AMCL -->
  <arg name="custom_amcl_launch_file" default="$(find turtlebot_navegacion)/include/amcl.launch.xml"/>
  <arg name="initial_pose_x" default="0.0"/> <!-- Use 17.0 for willow's map in simulation -->
  <arg name="initial_pose_y" default="0.0"/> <!-- Use 17.0 for willow's map in simulation -->
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(arg custom_amcl_launch_file)">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <!-- Move base -->
  <include file="$(find turtlebot_navegacion)/include/move_base.launch.xml">
  </include>

</launch>
```

Figura 8.12. Archivo raq_navegacion.launch.

Al igual que en el apartado anterior, creamos un archivo que abra la visualización en RViz, para poder observarlo desde el ordenador. En esta configuración de RViz, además de contener la información necesaria para el AMCL, incluiremos la visualización del mapa de costos local y global y la trayectoria (ver figura 8.13).

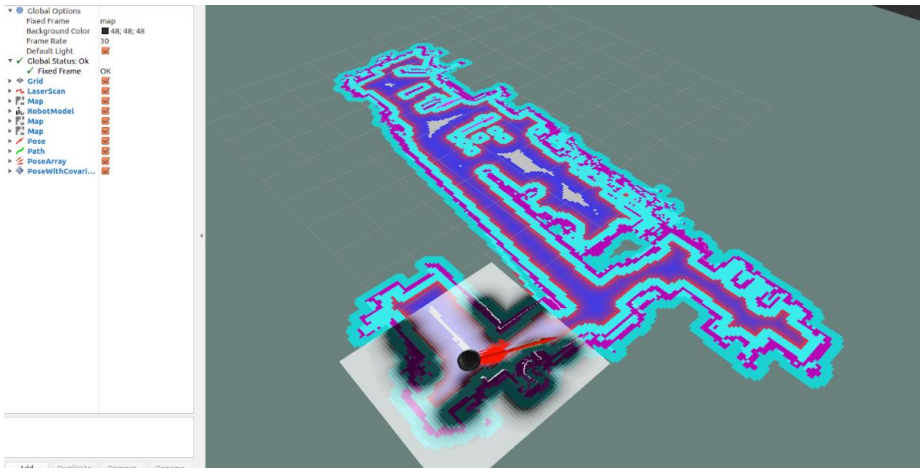

Figura 8.13. Imagen de RViz utilizando el nodo move_base.

Al igual que en la simulación, comprobaremos que todo funciona correctamente gracias a la herramienta *2D Nav Goal,* con la que enviaremos un punto objetivo en RViz y comprobaremos la navegación autónoma del robot hasta llegar a él (ver figura 8.14).

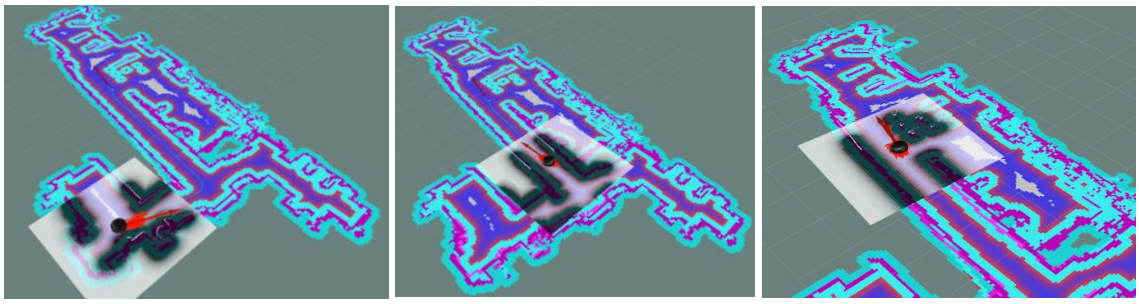

Figura 8.14. Ejemplo de uso de la herramienta 2D Nav Goal en un entorno real: de la habitación al salón. Inicio en la primera imagen, siguiendo la ruta en la segunda y, llegando al objetivo en la tercera imagen.

También comprobamos su capacidad de esquivar objetos no presentes en el mapa global (ver figura 8.15). El robot debería seguir su trayectoria normal hasta que se encuentre con el objeto y planee una nueva trayectoria para llegar a su destino, además, se tiene que percatar si el objeto sigue estando o no, en el entorno.

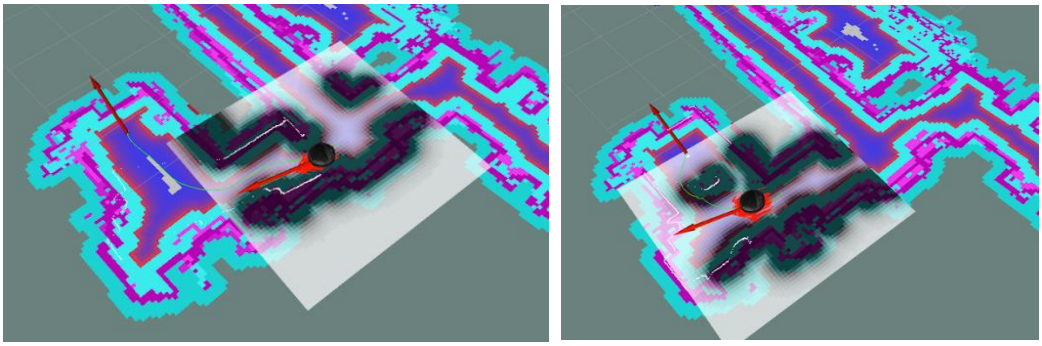

Figura 8.15. Ejemplo de evitación de obstáculos en un entorno real. En la primera imagen se observa la trayectoria inicial y en la segunda la modificada para evitar el obstáculo.

8.3 Uso de los nodos de navegación

Los nodos de navegación serán exactamente iguales que los nodos utilizados en simulación, exceptuando los puntos de las estancias, que tendremos que modificar acorde a nuestras necesidades.

8.3.1 Nodo bienvenida

El nodo de bienvenida llamado bienvenida_node se encarga de hacer una ruta por las distintas estancias de la casa, empezando por el salón y siguiendo por la cocina, aseo y habitación. Una vez llega a la habitación, el nodo de bienvenida se acabará. Exponemos a continuación el código:

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

typedef           actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;

std::vector<double> coordenadas(int e){
    std::vector<double> xy(3,0);
    switch(e){
        case 0:{
            xy[0] = 1.37786747411;
            xy[1] = -0.519725030186;}
            break;
        case 1:{
            xy[0] = 0.573037619893;
            xy[1] = 2.33116349821;}
            break;
        case 2:{
            xy[0] = 0.734003648292;
            xy[1] = 6.77233614459;}
            break;
        case 3:{
            xy[0] = 4.4005135082;
            xy[1] = 5.24558900024;}
            break;

    }
    xy[2] = e;
    return xy;
}

void Move_goal(std::vector<double> &v){

  //llamamos a action server
  MoveBaseClient ac("move_base", true);

  //Esperamos a que action server responda
  while(!ac.waitForServer(ros::Duration(5.0))){
    std::cout << "Esperando a action server" << std::endl;
  }

  move_base_msgs::MoveBaseGoal goal;

  //Enviamos los objetivos
```

```
  //map para un punto del mapa, base_link para referenciar a partir de la base
  goal.target_pose.header.frame_id = "map";

  goal.target_pose.header.stamp = ros::Time::now();

  goal.target_pose.pose.position.x = v[0];
  goal.target_pose.pose.position.y = v[1];
  goal.target_pose.pose.position.z = 0.0;
  goal.target_pose.pose.orientation.w = 1.0;

  std::string lugar;
  if(v[2] == 0) lugar = {"salon"};
  else if(v[2] == 1) lugar = {"cocina"};
  else if(v[2] == 2) lugar = {"aseo"};
  else lugar = {"habitacion"};

  std::cout << "Iniciando ruta hasta " <<lugar<<std::endl;
  ac.sendGoal(goal);

  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
  std::cout << "Bienvenido a " <<lugar<<std::endl;
  else
  std::cout << "No se ha podido llegar a " <<lugar<<std::endl;
}

int main(int argc, char** argv){
  ros::init(argc, argv, "bienvenida_node");

    std::cout << "------- BIENVENIDO -------" << std::endl;
    int aux = 0;
    while(aux<4){
       std::vector<double> objetivo = coordenadas(aux);
        Move_goal(objetivo);
       aux++;
    }

    std::cout << "La ruta ha finalizado" << std::endl;
  return 0;
}
```

Como podemos observar, los puntos han cambiado, en primer lugar, se dirigirá al salón, y, a continuación, a la cocina, el aseo y la habitación:

```
switch(e){
        case 0:{
            xy[0] = 1.37786747411;
            xy[1] = -0.519725030186;}
            break;
        case 1:{
            xy[0] = 0.573037619893;
            xy[1] = 2.33116349821;}
            break;
        case 2:{
            xy[0] = 0.734003648292;
            xy[1] = 6.77233614459;}
            break;
        case 3:{
            xy[0] = 4.4005135082;
            xy[1] = 5.24558900024;}
            break;

    }
```

La explicación del código expuesto la podemos encontrar en el apartado 6.4.1.

A continuación, una prueba del funcionamiento del nodo bienvenida_node en un entorno real. La posición inicial del robot cuando se llama al nodo es la habitación, una vez lanzado, este se dirige al salón (ver figura 8.16), posteriormente a la cocina (ver figura 8.17), después al aseo (ver figura 8.18) y finalmente a la habitación (ver figura 8.19). Una vez llegado a la habitación, el nodo se acabará.

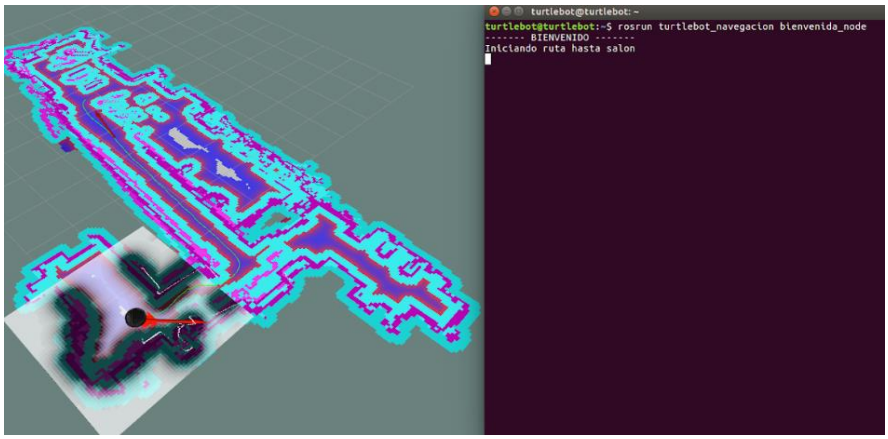

Figura 8.16. Inicio del nodo bienvenida_node en un entorno real en RViz.

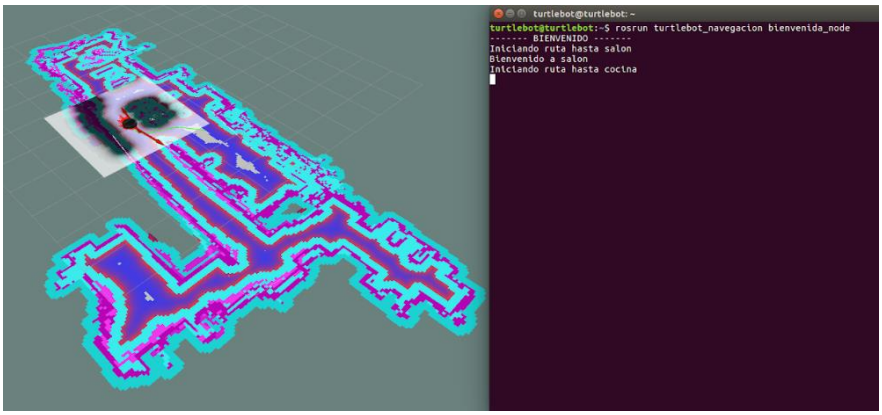

Figura 8.17. Imagen de RViz del nodo bienvenida_node de camino a la cocina.

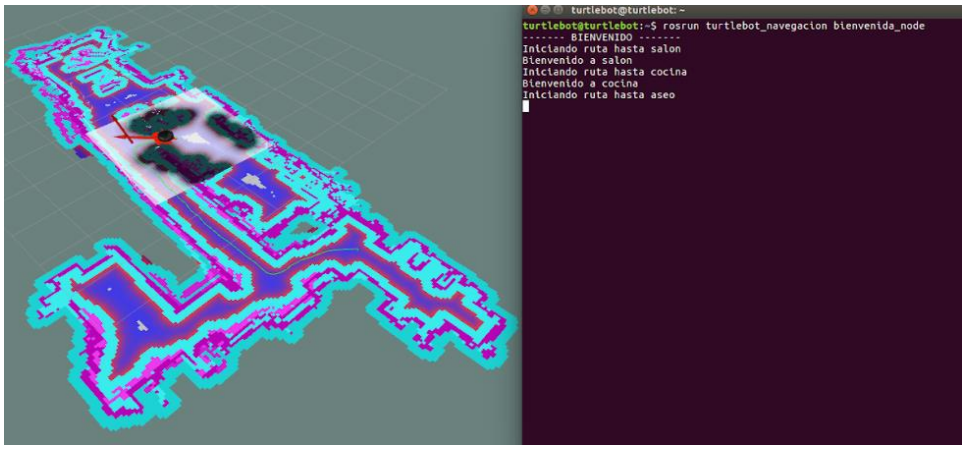

Figura 8.18. Imagen de RViz del nodo bienvenida_node dirigiéndose al aseo.

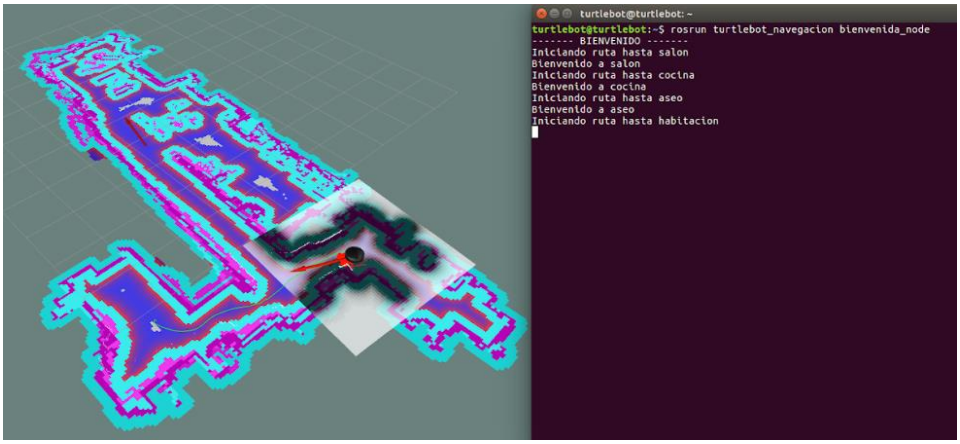

Figura 8.19. Imagen de RViz del nodo bienvenida_node dirigiéndose hacia la habitación.

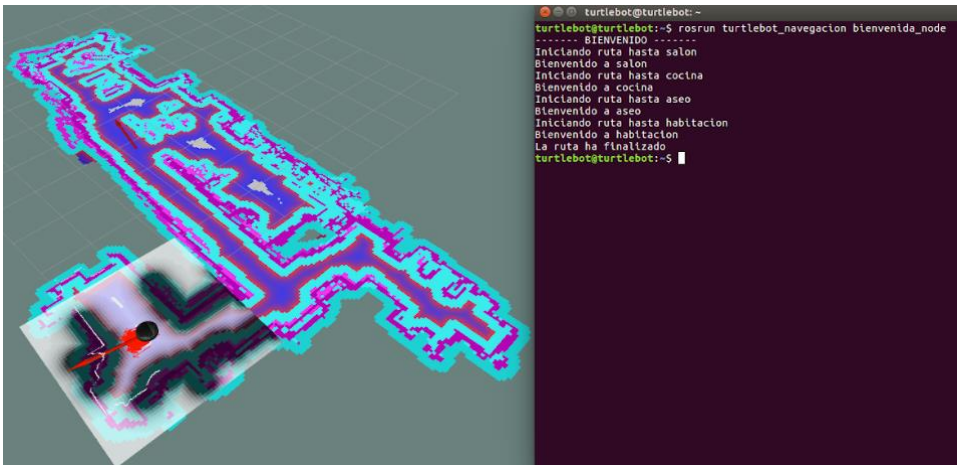

Figura 8.20. Finalización del nodo bienvenida_node en RViz.

8.3.2 Navegación semántica

Al igual que en el nodo bienvenida_node, el código utilizado para este nodo será el mismo que el utilizado en simulación, a excepción de los puntos necesarios para alcanzar los objetivos propuestos.

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

typedef           actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;

std::vector<double> coordenadas(int e){
    std::vector<double> xy(2,0);
    switch(e){
        case 0:{
            std::cout << "Ha elegido salon" << std::endl;
            xy[0] = 1.37786747411;
            xy[1] = -0.519725030186;}
            break;
        case 1:{
            std::cout << "Ha elegido habitacion" << std::endl;
            xy[0] = 4.4005135082;
            xy[1] = 5.24558900024;}
            break;
        case 2:{
            std::cout << "Ha elegido cocina" << std::endl;
            xy[0] = 0.573037619893;
            xy[1] = 2.33116349821;};}
            break;
        case 3:{
            std::cout << "Ha elegido aseo" << std::endl;
            xy[0] = 0.734003648292;
            xy[1] = 6.77233614459;}
            break;

    }
    return xy;
}


void Move_goal(std::vector<double> &v){

  //llamamos a action server
  MoveBaseClient ac("move_base", true);

  //esperamos a que action server responda
  while(!ac.waitForServer(ros::Duration(5.0))){
       ROS_INFO("Esperando a action server");
  }

  move_base_msgs::MoveBaseGoal goal;

  //enviamos los objetivos
  //map para un punto del mapa, base_link para referenciar a partir de la base
  goal.target_pose.header.frame_id = "map";
  goal.target_pose.header.stamp = ros::Time::now();

  goal.target_pose.pose.position.x = v[0];
  goal.target_pose.pose.position.y = v[1];
  goal.target_pose.pose.position.z = 0.0;
  goal.target_pose.pose.orientation.w = 1.0;
```

```
  std::cout << "En camino" << std::endl;
  ac.sendGoal(goal);

  ac.waitForResult();

  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    std::cout << "Ha llegado a su destino" << std::endl;
  else
    std::cout << "No ha podido llegar a su destino"<< std::endl;
}


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

  ros::init(argc, argv, "simple_navigation_goals_node");

    bool loop = true;

    do{

    std::cout << "------- ESPERANDO OBJETIVO -------" << std::endl;
    std::cout << "--------- SALON: PULSA 0 ---------" << std::endl;
    std::cout << "------ HABITACION: PULSA 1 -------" << std::endl;
    std::cout << "--------- COCINA: PULSA 2 --------" << std::endl;
    std::cout << "--------- ASEO: PULSA 3 ----------" << std::endl;

    int eleccion;
    char continuar;

    do{
       std::cin >> eleccion;
       if(eleccion != 0 && eleccion != 1 && eleccion != 2 && eleccion != 3)
              std::cout << "No se donde esta ese lugar" << std::endl;
    }

    while(eleccion != 0 && eleccion != 1 && eleccion != 2 && eleccion != 3);

    std::vector<double> objetivo = coordenadas(eleccion);
    Move_goal(objetivo);

    do{
    std::cout << "Deseas otro objetivo? (y/n)" << std::endl;
    std::cin >> continuar;
    }
    while(continuar != 'y' && continuar != 'n');

    if(continuar == 'y') loop = true;
    else loop = false;
    }

    while(loop);
  return 0;
}
```

Como se ha mencionado anteriormente, el código es el mismo para un entorno real y para simulación, la diferencia son los puntos de los objetivos buscados:

```
switch(e){
        case 0:{
            std::cout << "Ha elegido salon" << std::endl;
            xy[0] = 1.37786747411;
            xy[1] = -0.519725030186;}
            break;
        case 1:{
            std::cout << "Ha elegido habitacion" << std::endl;
            xy[0] = 4.4005135082;
```

```
            xy[1] = 5.24558900024;}
            break;
        case 2:{
            std::cout << "Ha elegido cocina" << std::endl;
            xy[0] = 0.573037619893;
            xy[1] = 2.33116349821;};}
            break;
        case 3:{
            std::cout << "Ha elegido aseo" << std::endl;
            xy[0] = 0.734003648292;
            xy[1] = 6.77233614459;}
            break;

}
```

La explicación de las líneas del código la podemos encontrar en el apartado 6.4.2.

A continuación, un ejemplo de funcionamiento del nodo simple_goals_node en un entorno real. En primer lugar, utilizamos el nodo para dirigirnos a la cocina (ver figura 8.21) y en segundo, para ir a la habitación (ver figura 8.22).

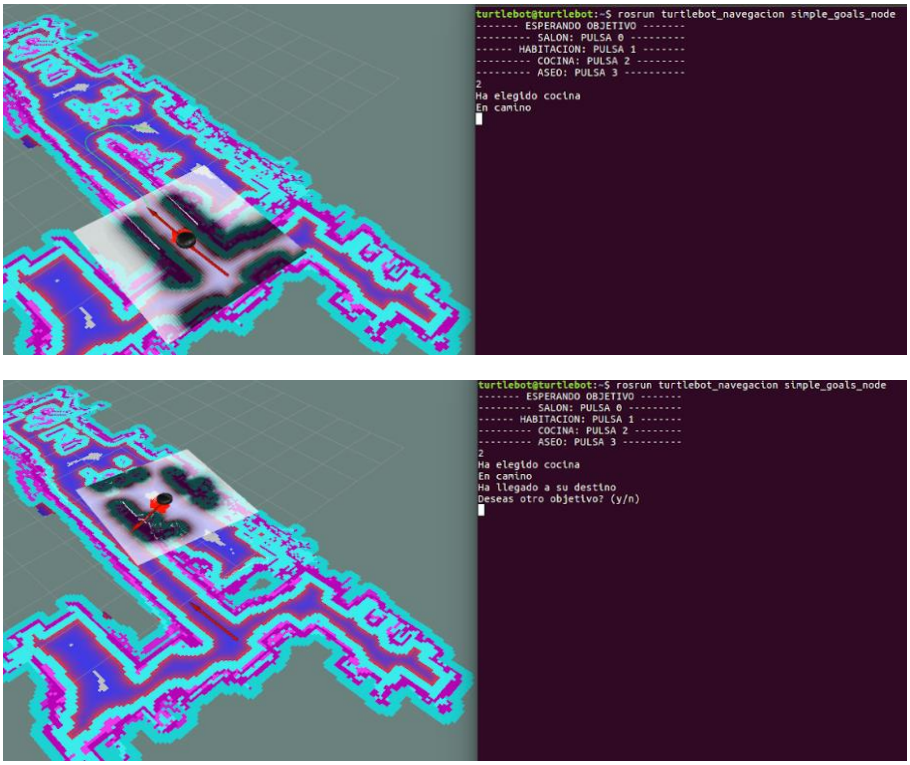

Figura 8.21. Imagen de RViz del nodo simple_goals_node ejecutándose desde el pasillo hasta la cocina.

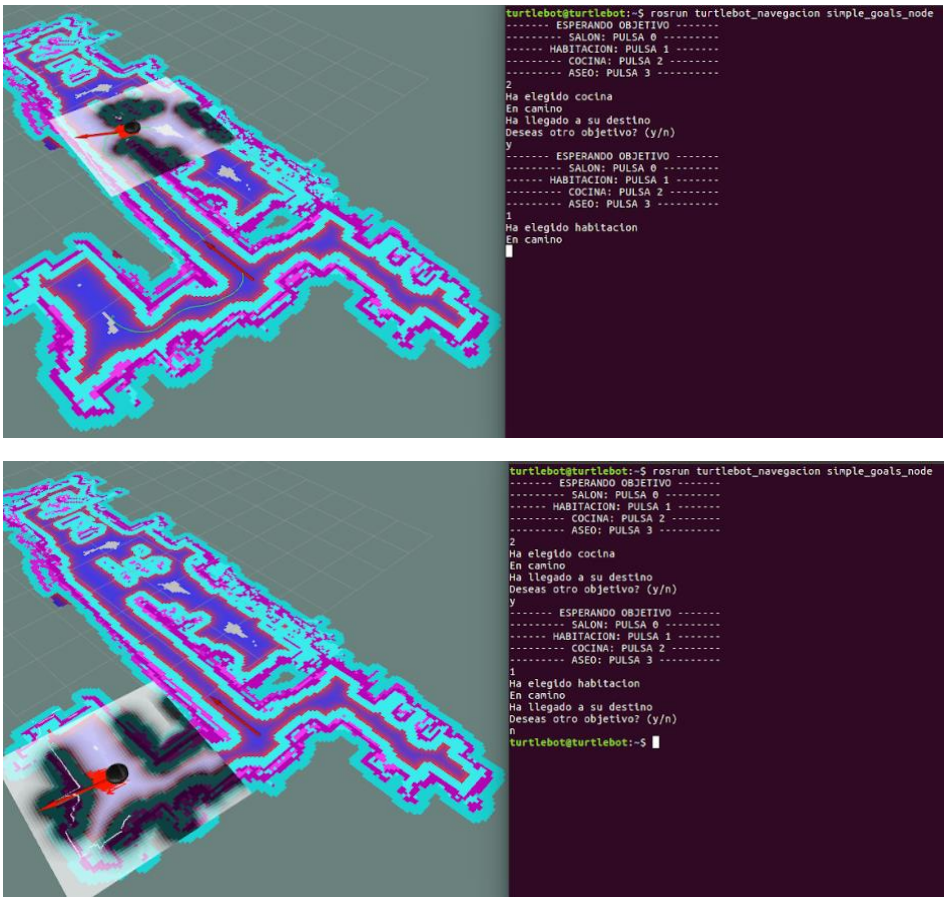

Figura 8.22. Imagen de RViz del nodo simple_goals_node ejecutándose desde la cocina hasta la habitación y cerrando el nodo.

9 Conclusiones y vías futuras

9.1 Conclusiones

Respecto a la parte técnica que el proyecto engloba, podemos afirmar que el resultado de la navegación autónoma del Turtlebot por un entorno conocido ha sido favorable. Hemos conseguido integrar el láser, así como las conexiones remotas entre diferentes programas Ubuntu, para poder crear un mapa del entorno y poder navegar por él de forma autónoma, evitando los objetos móviles que puedan aparecer en la ruta. Además, el robot se mueve con soltura por el entorno y no se atranca en marcos de puertas o en esquinas.

Los dos nodos de navegación creados funcionan correctamente; el Turtlebot llega adecuadamente al destino solicitado y recorre la casa estancia por estancia de forma autónoma y ligera. Hay que recalcar que esta tarea lleva mucha prueba y error debido al poco espacio presente en el entorno escogido. Hay zonas en las que era muy difícil que el robot navegara, ya sea por la dificultad para moverse por ellas y dar la vuelta, o porque prácticamente no cabía.

Aunque al principio hubo una serie de problemas a la hora de realizar las conexiones con el Turtlebot debido al WSL, al cambiar al ordenador con Linux todos estos problemas desaparecieron. También se tuvieron que modificar los códigos debido a que la actualización del software del segundo ordenador y del NUC era anterior a la utilizada en el primero, sin embargo, no requirió mucho esfuerzo.

El proyecto requiere de mucha documentación previa y constante aprendizaje, pero gracias a la asignatura de Robótica Móvil, el inicio del proyecto fue mucho más sencillo y llevadero, puesto que empezar de cero con Ubuntu hubiera sido una tarea mucho más compleja. Además, al haber tanta documentación puede ser un poco abrumador, pero, al mismo tiempo, hace que sea más interesante debido a la cantidad de formas posibles para resolver un mismo problema.

9.2 Vías futuras

Como hemos visto al principio de este proyecto, el sector de la robótica asistencial se encuentra en un crecimiento exponencial debido a la cantidad de recursos que tenemos hoy en día. Aunque haya cierta controversia en cómo o para qué utilizarla, el hecho de que mejore nuestra calidad de vida es innegable. De esta forma, es muy útil integrar un robot de estas características en una casa con personas mayores, y mejorarlo para que cada vez sea más práctico, pero a la vez más económico.

Se podría instalar una cámara 3D para poder hacer la navegación basada en Inteligencia Artificial, tomando datos de referencia como una lavadora o una cama, para que, una vez que esté en la estancia, pueda ir a un sitio concreto de esta sin tener que cambiar código o volver a guardar nuevos puntos. Además, sería ideal que el robot fuera de una altura mayor, puesto que así ayudaría a llevar bandejas con comida u otros objetos sin necesidad de que la persona haga un esfuerzo agachándose.

10 Bibliografía

Burgo, X. G. (2013). *Semantic mapping in ROS.*

Datasheet Kobuki. (s.f.). Obtenido de https://iclebo-kobuki.readthedocs.io/_/downloads/en/latest/pdf/

Dieter Fox, Wolfram Burgard, Sebastian Thrun. (s.f.). *The Dynamic Window Approach to Collision Avoidance .*

Gallart, X. (s.f.). *Approaches to mobile robot localitation in indoor enviroments.*

GitHub. (s.f.). Obtenido de Turtlebot2: https://github.com/gaunthan/Turtlebot2-On-Melodic

GitHub. (s.f.). Obtenido de Follow waypoints: https://github.com/danielsnider/follow_waypoints

Husarion docs. (s.f.). Obtenido de Path Planning: https://husarion.com/tutorials/ros-tutorials/7-path-planning/

Jose. (2007). Historia de los robots de Honda: del proyecto E0 a ASIMO. *Abadía digital*.

José Guillermo Guarnizo Marin, Daniela Bautista Díaz, Juan Sebastián Sierra Torres. (s.f.). *Una Revisión Sobre la Evolución de la Robótica Móvil.*

Manuel APARICIO PAYÁ, Mario TOBOSO MARTÍN, Txetxu AUSÍN DÍEZ, Anibal MONASTERIO ASTOBIZA, Ricardo MORTE FERRER, Daniel LÓPEZ CASTRO. (s.f.). Un marco ético-político para la robótica asistencial. *Revista de estudios de la ciencia y la tecnología*.

Melo, J. M. (2012). *Implementation of Algorithms for Navigation of an Autonomous Mobile Robot by using software component based frameworks .*

Microsoft Docs. (s.f.). Obtenido de Pasos de instalación manual para versiones anteriores de WSL: Pasos de instalación manual para versiones anteriores de WSL | Microsoft Docs

NASA. (s.f.). Obtenido de Mars Pathfinder: https://mars.nasa.gov/mars-exploration/missions/pathfinder/

Roland Siegwart, Illah Reza Nourbakhsh, Davide Scaramuzza. (s.f.). *Introduction to autonomous mobile robot.*

ROS Wiki. (s.f.). Obtenido de ROS/Introduction - ROS Wiki

ROS Wiki. (s.f.). Obtenido de Navigation: http://wiki.ros.org/navigation

ROS Wiki. (s.f.). Obtenido de Move_base: http://wiki.ros.org/move_base?distro=noetic

ROS Wiki. (s.f.). Obtenido de Concepts ROS: ROS/Conceptos - ROS Wiki

ROS Wiki. (s.f.). Obtenido de Descarga de ROS Melodic: melodic/Installation/Ubuntu - ROS Wiki

ROS Wiki. (s.f.). Obtenido de Creación del espacio de trabajo: http://wiki.ros.org/es/ROS/Tutoriales/catkin/CreateWorkspace

ROS Wiki . (s.f.). Obtenido de AMCL: http://wiki.ros.org/amcl?distro=noetic

Ros, P. L. (2021). *Navegación autónoma basada en un mapa topológico enriquecido semánticamente: Aplicación a un robot de servicio asistente en un entorno interior.* Cartagena.

ROS.org. (s.f.). Obtenido de Simple Navigation goals: http://wiki.ros.org/navigation/Tutorials/SendingSimpleGoals#:~:text=Description%3A%20The%20Navigation%20Stack%20serves,in%20conjunction%20with%20a%20map.

Softbankrobotics. (s.f.). Obtenido de Robot Nao: https://www.softbankrobotics.com/emea/es/nao

SourceForge.net. (s.f.). Obtenido de Descarga de Xming: Xming X Server for Windows download | SourceForge.net

Thrun, S. (1997). *Learning metric-topological maps for indoor mobile robot navigation .*

Turtlebot. (s.f.). Obtenido de Turtlebot2: https://www.turtlebot.com/turtlebot2/

Turtlebot. (s.f.). Obtenido de https://www.turtlebot.com/

Víctor Ricardo Barrientos Sotelo, José Rafael García Sánchez, Dr. Ramón Silva Ortigoza. (s.f.). En *Robots Móviles: Evolución y Estado del Arte.* Mexico.

Wiki ROS. (s.f.). Obtenido de Instalación y configuración de la pila de navegación en un robot: http://wiki.ros.org/navigation/Tutorials/RobotSetup

Wiki ROS. (s.f.). Obtenido de Costmap_2d: http://wiki.ros.org/costmap_2d

Wiki ROS. (s.f.). Obtenido de Dwa_local_planner: http://wiki.ros.org/dwa_local_planner

Wikipedia. (s.f.). Obtenido de Turtlebot: https://en.wikipedia.org/wiki/TurtleBot

ANEXOS

1 LIDAR.LAUNCH.XML

```
<launch>
  <arg name="scan_topic"  default="scan" />
  <arg name="base_frame"  default="base_footprint"/>
  <arg name="odom_frame"  default="odom"/>

  <node      pkg="gmapping"      type="slam_gmapping"      name="slam_gmapping"
output="screen">
    <param name="base_frame" value="$(arg base_frame)"/>
    <param name="odom_frame" value="$(arg odom_frame)"/>
    <param name="map_update_interval" value="5.0"/>
    <param name="maxUrange" value="6.0"/>
    <param name="maxRange" value="8.0"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.075"/>
    <param name="ogain" value="3.0"/>
    <param name="lskip" value="0"/>
    <param name="minimumScore" value="200"/>
    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
    <param name="stt" value="0.02"/>
    <param name="linearUpdate" value="0.5"/>
    <param name="angularUpdate" value="0.436"/>
    <param name="temporalUpdate" value="-1.0"/>
    <param name="resampleThreshold" value="0.5"/>
    <param name="particles" value="80"/>
  <!--
    <param name="xmin" value="-50.0"/>
    <param name="ymin" value="-50.0"/>
    <param name="xmax" value="50.0"/>
    <param name="ymax" value="50.0"/>
  make the starting size small for the benefit of the Android client's memory...
  -->
    <param name="xmin" value="-1.0"/>
    <param name="ymin" value="-1.0"/>
    <param name="xmax" value="1.0"/>
    <param name="ymax" value="1.0"/>

    <param name="delta" value="0.05"/>
    <param name="llsamplerange" value="0.01"/>
    <param name="llsamplestep" value="0.01"/>
    <param name="lasamplerange" value="0.005"/>
    <param name="lasamplestep" value="0.005"/>
    <remap from="scan" to="$(arg scan_topic)"/>
  </node>
</launch>
```

2 AMCL.LAUNCH.XML

```
<launch>
  <arg name="use_map_topic"   default="false"/>
  <arg name="scan_topic"      default="scan"/>
  <arg name="initial_pose_x"  default="0.0"/>
  <arg name="initial_pose_y"  default="0.0"/>
  <arg name="initial_pose_a"  default="0.0"/>
  <arg name="odom_frame_id"   default="odom"/>
  <arg name="base_frame_id"   default="base_footprint"/>
  <arg name="global_frame_id" default="map"/>

  <node pkg="amcl" type="amcl" name="amcl">
    <param name="use_map_topic"             value="$(arg use_map_topic)"/>
    <!-- Publish scans from best pose at a max of 10 Hz -->
    <param name="odom_model_type"           value="diff"/>
    <param name="odom_alpha5"               value="0.1"/>
    <param name="gui_publish_rate"          value="10.0"/>
    <param name="laser_max_beams"             value="60"/>
    <param name="laser_max_range"           value="12.0"/>
    <param name="min_particles"             value="500"/>
    <param name="max_particles"             value="2000"/>
    <param name="kld_err"                   value="0.05"/>
    <param name="kld_z"                     value="0.99"/>
    <param name="odom_alpha1"               value="0.2"/>
    <param name="odom_alpha2"               value="0.2"/>
    <!-- translation std dev, m -->
    <param name="odom_alpha3"               value="0.2"/>
    <param name="odom_alpha4"               value="0.2"/>
    <param name="laser_z_hit"               value="0.5"/>
    <param name="laser_z_short"             value="0.05"/>
    <param name="laser_z_max"               value="0.05"/>
    <param name="laser_z_rand"              value="0.5"/>
    <param name="laser_sigma_hit"           value="0.2"/>
    <param name="laser_lambda_short"        value="0.1"/>
    <param name="laser_model_type"          value="likelihood_field"/>
    <!-- <param name="laser_model_type" value="beam"/> -->
    <param name="laser_likelihood_max_dist" value="2.0"/>
    <param name="update_min_d"              value="0.25"/>
    <param name="update_min_a"              value="0.2"/>
    <param name="odom_frame_id"             value="$(arg odom_frame_id)"/>
    <param name="base_frame_id"             value="$(arg base_frame_id)"/>
    <param name="global_frame_id"           value="$(arg global_frame_id)"/>
    <param name="resample_interval"         value="1"/>
    <!-- Increase tolerance because the computer can get quite busy -->
    <param name="transform_tolerance"       value="1.0"/>
    <param name="recovery_alpha_slow"       value="0.0"/>
    <param name="recovery_alpha_fast"       value="0.0"/>
    <param name="initial_pose_x"            value="$(arg initial_pose_x)"/>
    <param name="initial_pose_y"            value="$(arg initial_pose_y)"/>
    <param name="initial_pose_a"            value="$(arg initial_pose_a)"/>
    <remap from="scan"                      to="$(arg scan_topic)"/>
  </node>
</launch>
```

3 MOVE_BASE.LAUNCH.XML

```
<!--
    ROS navigation stack with velocity smoother and safety (reactive) controller
-->
<launch>
  <include                                                          file="$(find
turtlebot_navegacion)/include/velocity_smoother.launch.xml"/>
  <include                                                          file="$(find
turtlebot_navegacion)/include/safety_controller.launch.xml"/>

  <arg name="odom_frame_id"   default="odom"/>
  <arg name="base_frame_id"   default="base_footprint"/>
  <arg name="global_frame_id" default="map"/>
  <arg name="odom_topic" default="odom" />
  <arg name="laser_topic" default="scan" />
  <arg                 name="custom_param_file"                default="$(find
turtlebot_navegacion)/param/dummy.yaml"/>

  <node   pkg="move_base"   type="move_base"   respawn="false"   name="move_base"
output="screen">
    <rosparam                                                         file="$(find
turtlebot_navegacion)/param/raq_costmap_common_params.yaml"      command="load"
ns="global_costmap" />
    <rosparam                                                         file="$(find
turtlebot_navegacion)/param/raq_costmap_common_params.yaml"      command="load"
ns="local_costmap" />
    <rosparam                                                         file="$(find
turtlebot_navegacion)/param/local_costmap_params.yaml" command="load" />
    <rosparam                                                         file="$(find
turtlebot_navegacion)/param/global_costmap_params.yaml" command="load" />
    <rosparam                                                         file="$(find
turtlebot_navegacion)/param/raq_dwa_local_planner_params.yaml"   command="load"
/>
    <rosparam  file="$(find  turtlebot_navegacion)/param/move_base_params.yaml"
command="load" />
    <rosparam                                                         file="$(find
turtlebot_navegacion)/param/global_planner_params.yaml" command="load" />
    <rosparam                                                         file="$(find
turtlebot_navegacion)/param/navfn_global_planner_params.yaml" command="load" />
    <!-- external params file that could be loaded into the move_base namespace
-->
    <rosparam file="$(arg custom_param_file)" command="load" />

    <!-- reset frame_id parameters using user input data -->
    <param name="global_costmap/global_frame" value="$(arg global_frame_id)"/>
    <param          name="global_costmap/robot_base_frame"          value="$(arg
base_frame_id)"/>
    <param name="local_costmap/global_frame" value="$(arg odom_frame_id)"/>
    <param name="local_costmap/robot_base_frame" value="$(arg base_frame_id)"/>
    <param name="DWAPlannerROS/global_frame_id" value="$(arg odom_frame_id)"/>

    <remap from="cmd_vel" to="navigation_velocity_smoother/raw_cmd_vel"/>
    <remap from="odom" to="$(arg odom_topic)"/>
    <remap from="scan" to="$(arg laser_topic)"/>
  </node>
</launch>
```

4 NAVFN_GLOBAL_PLANNER_PARAMS.XAML

```
NavfnROS:
  visualize_potential: false    #Publish potential for rviz as pointcloud2, not
really helpful, default false
  allow_unknown: false            #Specifies whether or not to allow navfn to
create plans that traverse unknown space, default true
                                 #Needs to have track_unknown_space: true in the
obstacle / voxel layer (in costmap_commons_param) to work
  planner_window_x: 0.0          #Specifies the x size of an optional window to
restrict the planner to, default 0.0
  planner_window_y: 0.0          #Specifies the y size of an optional window to
restrict the planner to, default 0.0

  default_tolerance: 0.0        #If the goal is in an obstacle, the planer will
plan to the nearest point in the radius of default_tolerance, default 0.0
                                 #The area is always searched, so could be slow
for big values
```

5 GLOBAL_PLANNER_PARAMS.YAML

```
GlobalPlanner:                                    # Also see:
http://wiki.ros.org/global_planner
  old_navfn_behavior: false                       # Exactly mirror behavior of
navfn, use defaults for other boolean parameters, default false
  use_quadratic: true                             # Use the quadratic approximation
of the potential. Otherwise, use a simpler calculation, default true
  use_dijkstra: true                              # Use dijkstra's algorithm.
Otherwise, A*, default true
  use_grid_path: false                            # Create a path that follows the
grid boundaries. Otherwise, use a gradient descent method, default false

  allow_unknown: true                             # Allow planner to plan through
unknown space, default true
                                                  #Needs          to          have
track_unknown_space:    true    in    the    obstacle    /    voxel    layer    (in
costmap_commons_param) to work
  planner_window_x: 0.0                           # default 0.0
  planner_window_y: 0.0                           # default 0.0
  default_tolerance: 0.0                          # If goal in obstacle, plan to
the closest point in radius default_tolerance, default 0.0

  publish_scale: 100                              # Scale by which the published
potential gets multiplied, default 100
  planner_costmap_publish_frequency: 0.0          # default 0.0

  lethal_cost: 253                                # default 253
  neutral_cost: 50                                # default 50
  cost_factor: 3.0                                # Factor to multiply each cost
from costmap by, default 3.0
  publish_potential: true                         # Publish Potential Costmap
(this is not like the navfn pointcloud2 potential), default true
```

6 MOVE_BASE_PARAMS.YAML

```
# Move base node parameters. For full documentation of the parameters in this
file, please see
#
#  http://www.ros.org/wiki/move_base
#
shutdown_costmaps: false

controller_frequency: 5.0
controller_patience: 3.0


planner_frequency: 1.0
planner_patience: 5.0

oscillation_timeout: 10.0
oscillation_distance: 0.2

# local planner - default is trajectory rollout
base_local_planner: "dwa_local_planner/DWAPlannerROS"

base_global_planner:              "navfn/NavfnROS"              #alternatives:
global_planner/GlobalPlanner, carrot_planner/CarrotPlanner


#We plan to integrate recovery behaviors for turtlebot but currently those belong
to gopher and still have to be adapted.
## recovery behaviors; we avoid spinning, but we need a fall-back replanning
#recovery_behavior_enabled: true

#recovery_behaviors:
  #- name: 'super_conservative_reset1'
    #type: 'clear_costmap_recovery/ClearCostmapRecovery'
  #- name: 'conservative_reset1'
    #type: 'clear_costmap_recovery/ClearCostmapRecovery'
  #- name: 'aggressive_reset1'
    #type: 'clear_costmap_recovery/ClearCostmapRecovery'
  #- name: 'clearing_rotation1'
    #type: 'rotate_recovery/RotateRecovery'
  #- name: 'super_conservative_reset2'
    #type: 'clear_costmap_recovery/ClearCostmapRecovery'
  #- name: 'conservative_reset2'
    #type: 'clear_costmap_recovery/ClearCostmapRecovery'
  #- name: 'aggressive_reset2'
    #type: 'clear_costmap_recovery/ClearCostmapRecovery'
  #- name: 'clearing_rotation2'
    #type: 'rotate_recovery/RotateRecovery'

#super_conservative_reset1:
  #reset_distance: 3.0
#conservative_reset1:
  #reset_distance: 1.5
#aggressive_reset1:
  #reset_distance: 0.0
#super_conservative_reset2:
  #reset_distance: 3.0
#conservative_reset2:
  #reset_distance: 1.5
#aggressive_reset2:
  #reset_distance: 0.0
```

7 SAFETY_CONTROLLER.LAUNCH.XML

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

8 VELOCITY_SMOOTHER.LAUNCH.XML

```
<!--
         Velocity smoother
-->
<launch>
  <node   pkg="nodelet"   type="nodelet"   name="navigation_velocity_smoother"
args="load                        yocs_velocity_smoother/VelocitySmootherNodelet
mobile_base_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"/>

    <!-- Robot velocity feedbacks; use the default base configuration -->
    <remap from="navigation_velocity_smoother/odometry" to="odom"/>
    <remap                     from="navigation_velocity_smoother/robot_cmd_vel"
to="mobile_base/commands/velocity"/>
  </node>
</launch>
```