Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Integración de sistema láser y visión en la navegación de un robot móvil

Autor: Manuel Mora Nieto
Tutor: Miguel Ángel Ridao Carlini

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Proyecto Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Integración de sistema láser y visión en la navegación de un robot móvil

Autor:
Manuel Mora Nieto

Tutor:
Miguel Ángel Ridao Carlini
Profesor titular

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2018

Proyecto Fin de Carrera: Integración de sistema láser y visión en la navegación de un robot móvil

Autor: Manuel Mora Nieto

Tutor: Miguel Ángel Ridao Carlini

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2018

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

En estas líneas tengo la fortuna de poder nombrar a mi familia, un apoyo incondicional. Gracias a mi padre, mi madre y mi hermana por ofrecerme todo lo posible para que el sueño de ser ingeniero sea cada día un poco más real. Sin su paciencia, sabiduría y esfuerzo, no estaría escribiendo esta memoria.

También tengo que agradecer a los compañeros con los que he ido compartiendo momentos en la carrera, por ofrecer su ayuda constante y desinteresada.

Hay que resaltar la labor del INTA (Instituto Nacional de Técnica Aeroespacial), el cual deja el robot Summit-XL en manos de la universidad para tareas de investigación, haciendo posible que podamos trabajar con él.

Por último, y no menos importante, me gustaría destacar el trabajo de todos y cada uno de mis profesores. La manera de transmitir sus conocimientos y vocación han servido para que pueda progresar en el ámbito de la ingeniería y, por supuesto, para ser mejor persona. Especial mención a mi tutor Miguel Ángel por brindarme la oportunidad de realizar este trabajo, el cuál era un reto por enfrentarme a mundos del campo de la robótica que eran desconocidos para mí.

Manuel Mora Nieto

Alumno de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla

Sevilla, 2018

Resumen

El trabajo se ha realizado con el Summit-XL, robot móvil creado por la empresa Robotnik y programado con ROS (Robot Operating System). El hecho de elegir ese robot es que el departamento posee un ejemplar del mismo en el laboratorio, cedido por el INTA (Instituto Nacional de Técnica Aeroespacial) para investigación.

La finalidad del robot es la navegación en exteriores para tareas de vigilancia, bien teleoperado o bien de manera autónoma. En cualquier caso, es necesario interactuar con el medio a través de sensores y obtener información del mismo. Por ello, el robot cuenta con un sensor láser que gira sobre el chasis del robot y que permite detectar objetos y calcular la distancia a la que se encuentran.

Al estar situado sobre el chasis y realizar un barrido horizontal, todo obstáculo que se encuentre por debajo no será detectado. Como la navegación está orientada a exteriores, existe una amplia posibilidad de encontrar agujeros en el suelo que el robot no tendría en cuenta. Por ello, se pretende en este trabajo hacer uso de la cámara que también posee el robot y así vigilar el suelo, reconociendo agujeros en el mismo. De esta manera, el mapa del entorno de navegación que se construye con el láser se complementa con la información obtenida por la cámara.

Abstract

The work has been made with the Summit-XL, a mobile robot created by the company Robotnik and programmed in ROS (Robot Operating System). The fact of choosing that robot is that the department has a copy of it in the laboratory, provided by INTA (National Institute of Aerospace Technology) for research.

The objective of the robot is outdoor navigation for surveillance tasks, either teleoperated or autonomously. In anyway, it is necessary to interact with the environment through sensors and obtain information from it. Therefore, the robot has a laser sensor which rotates over the chassis of the robot and that allows detecting objects and calculating the distance they are.

Because the sensor is located over the chassis and makes a horizontal scan, any obstacle that is below will not be detected. As the navigation is oriented outdoors, there is a large possibility of finding holes in the ground that the robot would not take into account. Therefore, it is hoped in this work to make use of the camera that also owns the robot and thus watch the ground, recognizing holes in it. In this way, the map of the navigation environment that is built with the laser is complemented by the information obtained by the camera.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

{A}	Sistema de referencia A
{B}	Sistema de referencia B
$\hat{X}_A$	Eje X del sistema de referencia A
$\hat{Y}_A$	Eje Y del sistema de referencia A
θ	Ángulo de rotación del eje X del sistema de referencia móvil respecto a la horizontal
$\boldsymbol{p^A}$	Posición del punto p en el sistema de referencia {A}
p_x^A	Coordenada x del punto p en el sistema de referencia {A}
p_y^A	Coordenada y del punto p en el sistema de referencia {B}
$\boldsymbol{{}_B^A R}$	Matriz de rotación del sistema de referencia {B} respecto al {A}

1 INTRODUCCIÓN

La robótica ha intentado desde sus orígenes complementar a los seres humanos para facilitar tareas en la vida cotidiana y avanzar en investigaciones difíciles. Así, se buscan patrones en tareas que antes realizaban humanos o que ni se podían realizar para que los robots puedan llevarlas a cabo. Sin embargo, la naturaleza no es tan sistemática como podría parecer y cada situación es un mundo diferente con una enorme casuística.

Por tanto, es necesario dotar a los robots de sensores que le permitan tener contacto con el medio físico. Con ellos será más fácil planificar tareas y se podrá, en la medida de lo posible, superar algún contratiempo en la ejecución. Es muy importante saber qué información queremos captar, con qué la queremos captar, en qué momento captarla y, sobre todo, cómo interpretarla.

Al fin y al cabo, con los sensores se trata de imitar la percepción del medio que tiene el ser humano. Por ello, cuánto mejor la imiten, mejor será la tarea que desarrolle el robot. Aquí entra en juego una disciplina muy potente: la visión artificial o visión por computador. La tarea de recibir imágenes, procesarlas, discriminarlas y reconocer e interpretar las escenas requiere de técnicas muy avanzadas que a día de hoy siguen en estudio y en continua mejoría. Ahí está el reto de este trabajo, conseguir que este campo se integre con el resto de sensores y arquitectura con que partía el robot, consiguiendo así mejorar la funcionalidad y fiabilidad del Summit-XL.

1.1 Marco contextual

Este TFG se ha realizado como continuación de trabajos realizados principalmente por otros compañeros en años anteriores. Los trabajos que se realizan con el Summit-XL se encuentran dentro de un proyecto de investigación que tiene la Universidad de Sevilla con el INTA, al cual pertenece el robot y lo presta a la universidad.

Con el robot se puede trabajar tanto de forma física como en un entorno de simulación en el framework ROS, para el cual hace falta tener instalado el sistema operativo Ubuntu. En nuestro caso, debido a que el grueso de la realización del trabajo coincidió con la ausencia del robot en el laboratorio, se ha optado por realizar únicamente trabajos de simulación. No obstante, dichos trabajos son extrapolables al robot real. Es, por tanto, el trabajo previo de simulación que hay que llevar a cabo siempre antes de la implementación física para evitar así posibles fallos en las pruebas que puedan dañar al sistema.

Para la simulación nos beneficiamos de trabajar con ROS, ya que este sistema provee para diferentes tipos de robots un trabajo previo que constituye la base de la cual partir en nuestros trabajos. Los modelos del robot en sus diferentes configuraciones, todos sus sensores y el entorno de simulación, entre otros, son códigos desarrollados por la empresa Robotnik que se pueden descargar del enlace dedicado a ello [1].

1.2 Objetivo del trabajo

El objetivo principal es el de conseguir un mapa de obstáculos del entorno por el que navega el robot que integre la información obtenida por el sensor láser con la que obtiene una cámara que apunta hacia el suelo. El hecho de que el robot sea teleoperado o navegue de manera autónoma es indiferente para conseguir este objetivo, por lo que se escoge la teleoperación, controlando así a distancia y en todo momento por donde se mueve el robot.

En trabajos previos se consiguió un mapa de ambas formas, teleoperación y navegación autónoma. Incluso se

incluyeron algoritmos de planificación de la trayectoria en función de la información sacada del mapa. Sin embargo, este mapa no contenía información completa por el problema ya comentado. Por tanto, esa información únicamente de los obstáculos que se encontraban a la altura de láser (aproximadamente la del chasis del robot) debía ser completada con los posibles obstáculos que el robot se pueda encontrar en el suelo. En concreto, nuestra aplicación reconocerá agujeros (forma circular) oscuros.

Para ello hay que ser capaz de lanzar la visión de la cámara, realizar el tratamiento de la imagen obtenida en tiempo real y reconocer los posibles obstáculos. Si se reconoce un agujero, se estima la posición en la que se encuentra y se añade al mapa de obstáculos generado por el láser. Se supondrá que sólo nos encontramos un agujero en la misma imagen (o ninguno); es decir, no habrá dos agujeros tan cercanos que los captemos a la vez con la cámara en la misma escena. En caso contrario, se reconoce más de un agujero en la misma toma de imagen pero no se garantiza que se pueda estimar la distancia.

Entre tanto objetivo hay otro secundario pero no menos importante: familiarizarse con el entorno de ROS. Se pretende llegar a programar en un sistema operativo tan usado hoy en día en el campo de la robótica y el cuál no es nada intuitivo cuando se empieza a trabajar. Para ello es necesario saber cómo está estructurado, entender la filosofía de otros programadores e integrar códigos de procedencias diversas en una misma aplicación.

1.3 Metodología empleada

Para llevar a cabo los objetivos anteriormente citados, se han seguido una serie de pasos:

- Instalación de Ubuntu y ROS en el PC.
- Aprendizaje práctico mediante tutoriales sobre la organización y los principios básicos de programación de ROS. Para ello se trabajan los tutoriales de [2].
- Obtención de información acerca del Summit-XL de [3] y [4] y códigos de partida de [1], [5] y [6].
- Discriminación de los códigos necesarios y lanzamiento de simulación.
- Teleoperación del robot y lanzamiento de la cámara.
- Obtención de información sobre la librería de tratamiento de imágenes OpenCV. Se realiza el tratamiento de imágenes y se publica la información obtenida.
- Se obtiene el código del mapeado por sensor láser y se lanza junto a la simulación.
- El código del sensor láser lee la información publicada por la cámara y la incluye en el mapa de obstáculos que genera.

Por tanto, esta memoria constará de partes bien diferenciadas. Tras esta primera introducción y unas nociones de historia de la robótica móvil, se detallará en el capítulo 3 la situación inicial del robot. Para ello haremos una descripción del mismo a nivel software y hardware y comentaremos brevemente trabajos que ya se le habían realizado.

En el capítulo 4 entraremos más en profundidad en el mundo de ROS para que se entiendan los conceptos básicos que rigen su funcionamiento y sus herramientas complementarias para la simulación, como pueden ser Gazebo o RViz. Además, se enlazará con los códigos disponibles para el Summit-XL.

La librería de tratamiento de imágenes OpenCV será el objeto del capítulo 5, donde se explicarán las funcionalidades de esta librería y cómo es posible conectarla con ROS.

El capítulo 6 está dedicado a la resolución del problema de visión y su consiguiente integración en el mapa. Por tanto, es el capítulo más importante.

Por último, se cierra la memoria con los capítulos 7 (dedicado al análisis de los resultados obtenidos en las diferentes pruebas realizadas) y 8 (para conclusiones y aplicaciones y mejoras futuras).

Se incluye también al final de la memoria un manual de usuario, que en un futuro puede servir de guía a otras personas que continúen con el Summit-XL. En él, se incluirán temas de instalación, descarga y compilación de

códigos y lanzamiento de simulaciones principalmente, puesto que no es trivial saber qué códigos hay que usar y cómo emplearlos.

Los códigos incluidos como anexo son simplemente los que se han creado desde cero o se han modificado en mayor medida: el de visión y el del mapeado. Se tratará de comentar las modificaciones realizadas en otros códigos a la vez que describimos la aplicación en esta memoria, pues son modificaciones leves aunque no menos importantes.

2 HISTORIA DE LA ROBÓTICA MÓVIL. IMPORTANCIA DE LA VISIÓN ARTIFICIAL

El ritmo al que avanza la robótica móvil es cada vez más elevado. Sin embargo, aunque algunos puedan verlo como algo que aún es futurista, hay que remontarse a la Segunda Guerra Mundial, donde aparecieron los primeros robots móviles cuyo medio de transporte era el aire: unas bombas volantes que usaban guías para determinar su explosión [7].

En los años 60, se lanzaron "Bestia" (por la Universidad John Hopkins) y "Mowbot". El primero de ellos era un robot acuático cuyas condiciones de autonomía poco óptimas limitaron sus aplicaciones. Mowbot, por su parte, tenía la tarea de cortar el césped.

Figura 2-1. Robot "Bestia" (Fuente: [8])

Hay que esperar a los años 70 para que se produzca un gran avance: la interpretación de la situación por parte del robot en su interacción con el medio. Esto se debe a los sensores y a un sistema de visión que se incluyeron en el robot "Sakey". También en esa década la Unión Soviética lanza un rover para explorar la Luna.

Los robots móviles se van extendiendo cada vez más a raíz de que, en 1990, John Engelberger, padre de los brazos robóticos, tratase de introducir estos robots en la industria. A esta expansión de la robótica también le precede la gran cantidad de ideas que surgían durante la década de los 80.

Los siguientes años destacan por las tareas de exploración en zonas de difícil acceso humano. Los robots "Dante I" y "Dante II", de la Universidad Carnegie Mellon, fueron introducidos en volcanes en 1994. Ya en 1997, la NASA teleoperó el rover "Sojouner" en la misión "Mars Pathfinder" desde la Tierra para que explorara el planeta Marte.

El siglo XXI, como ya se ha comentado, depara muchos avances y cada vez más aplicaciones en este campo de la robótica. Ya no sólo se centran en tareas difíciles, evitando poner en peligro a las personas, sino que empieza a crecer la robótica de servicio, instalándose en los hogares como forma de complementar la vida cotidiana.

De esta manera, los robots móviles (al igual que otro tipo de robots), han pasado por una serie de etapas [9]. Primero eran teleoperados o tenían instrucciones muy básicas. Más adelante, los robots pasan a observar a los

humanos para interiorizar algunas tareas a base de repetición. A esto le sigue el avance de incluir sensores, cuya información es usada por el ordenador para mandar instrucciones al robot. Por último y más reciente, el avance de los sensores permiten conocer con mayor detalle la situación y actuar antes situaciones que no sean tan usuales. Son los denominados "Robots inteligentes".

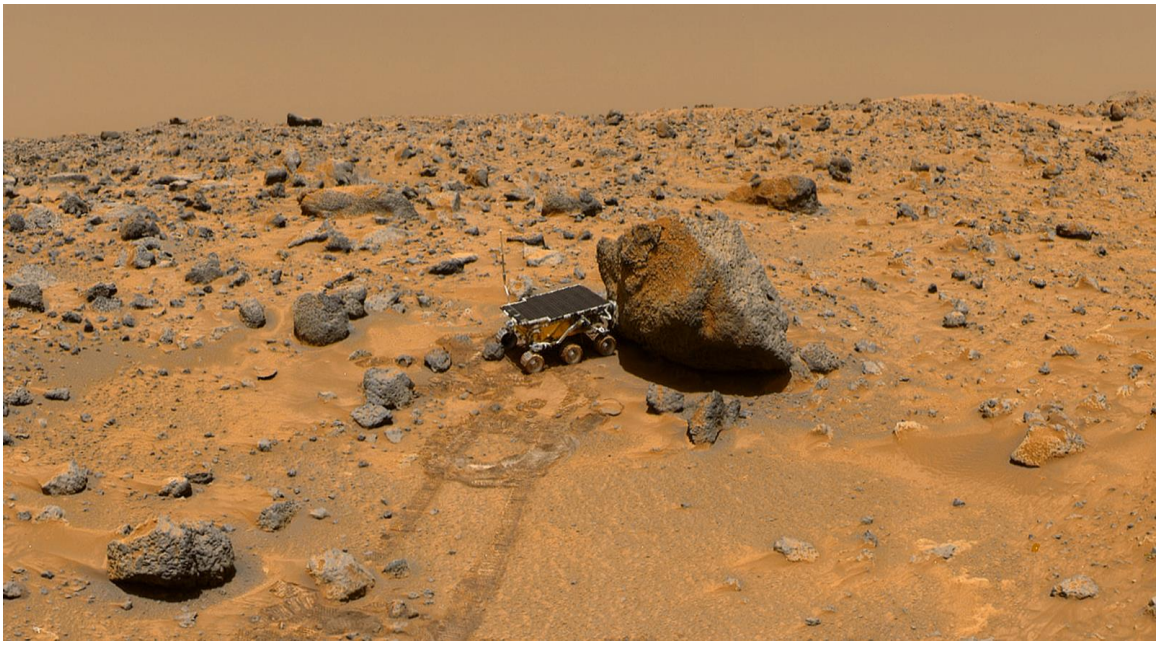

Figura 2-2. Rover Sojouner en la misión Mars Pathfinder (Fuente: [10])

En ese último grupo entra en juego la visión por computador. Lo tradicional en la industria eran robots manipuladores con una capacidad sensorial algo limitada [11]. Esto se intentó mejorar con la introducción de visión artificial, cuyo objetivo es que el robot tenga un sistema de visión como los humanos. Se gana en precisión y versatilidad y es posible manipular más productos de los que se podían antes. La idea es poder representar la realidad en el ordenador para poder manipularla y entenderla desde los ojos del robot. Sin embargo, no es una tarea fácil, pues la escena que capta la cámara depende del entorno: cambios de luz o movimientos de la escena principalmente. Por tanto, a la visión por computador para exteriores aún le queda camino por recorrer, pues las condiciones pueden ser muy cambiantes. Mientras tanto, en la industria, al ser movimientos repetitivos y situaciones muy similares las que se dan, la visión si está más consolidada.

3 SITUACIÓN DE PARTIDA

Se da a conocer en este capítulo el robot con el que se ha trabajado. Para ello se habla de sus componentes, aplicaciones, entorno de programación y, para completar, se explica brevemente sobre qué se había trabajado previamente con él.

3.1 Descripción del robot

3.1.1 A nivel hardware

El Summit-XL es una plataforma robótica creada por la empresa Robotnik [4]. Es lo que se conoce en robótica como un rover. Tiene cuatro ruedas con motor de alta potencia sin escobilla y con caja de cambios. Estas pueden ser de dos tipos según usemos el robot para una aplicación u otra:

- "Skid-Steering". Son ruedas de goma que se emplean tanto en exteriores para terrenos más complicados como en interiores. Son más útiles en el primer caso.

Figura 3-1. Summit-XL en configuración Skid-Steering (Fuente: [12])

- "Omnidirectional Drive". Ruedas mecanum que permiten moverse con mayor soltura en interiores.

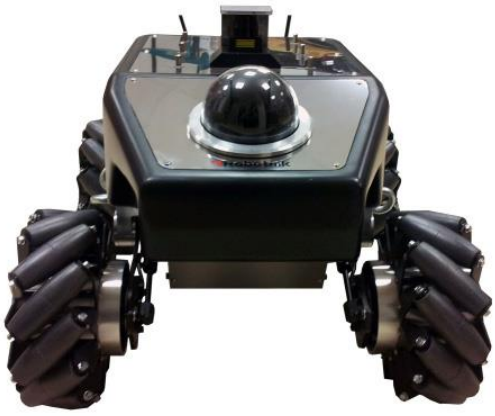

Figura 3-2. Summit-XL en configuración Omnidirectional Drive (Fuente: [12])

Se trata, por tanto, de un robot cuya configuración cinemática es la conocida como "Skid-Steering" y que alcanza una velocidad de 3 m/s. Esta configuración (Figura 3-3) es similar a las pistas de deslizamiento pero con ruedas a la izquierda y a la derecha [13]. El robot se desplazará debido a la diferencia de velocidades entre las ruedas de un lado y otro. Si queremos ir de frente, las ruedas del lado izquierdo y del derecho deberán girar hacia delante con la misma velocidad. Si, por ejemplo, queremos girar a la izquierda, el lado derecho deberá girar con mayor velocidad que el izquierdo. Cuanto mayor sea la diferencia de velocidades entre un lado y otro, más brusco será el giro.

Figura 3-3. Robot con configuración skid-steering (Fuente: [13])

Esta robusta configuración, unida a la fuerte estructura mecánica y la suspensión del Summit-XL hacen que sea adecuado para terrenos duros en los exteriores. Según el datasheet del robot [12], este es apto para: investigación, vigilancia, aplicaciones militares, monitorización remota y acceso a zonas peligrosas.

El robot cuenta con una serie de accesorios que lo dotan de mayor funcionalidad [4]. Entre ellos destacan:

- Cámara PTZ (Pan-Tilt-Zoom: ángulo de barrido panorámico, ángulo de inclinación y zoom para agrandar o disminuir la imagen) en la parte delantera.
- Sensor láser Hokuyo situado sobre el chasis.
- Gama de kits RTK-DGPS (Real-Time Kinematics and Differential GPS).
- Sistema IMU de medida inercial (Acelerómetro 3D, giroscopio 3D y campo magnético 3D).
- 4 encoders y un sensor angular de alta precisión dentro del chasis para la odometría.
- Conectividad interna: USB, RS232 y GPIO.
- Conectividad externa: USB, RJ45 y alimentación a 12 VDC.

Figura 3-4. Parte trasera del Summit-XL (Fuente: [4])

3.1.2 A nivel software

El Summit-XL tiene una arquitectura de control de código abierto ROS [12], del cual hablaremos más en detalle en el capítulo 4. Este framework o metasistema operativo apareció para usuarios Linux con su distribución Ubuntu, aunque a día de hoy ya se prueba en otros sistemas operativos.

El robot lleva embebida una CPU con Linux (Intel BayTrail J1900 o parecido) a la cual podemos conectarnos desde nuestro ordenador. Posee sistema WiFi 802.11n para su comunicación.

Como el ordenador en el que se realiza el trabajo tiene Windows 8.1, es necesario crear una partición del disco duro para poder instalar Ubuntu si queremos conservar Windows. La versión escogida es **Ubuntu 14.04 LTS**, la cual funciona de manera fiable. Ubuntu es un sistema operativo basado en Unix y que no resulta familiar para usuarios acostumbrados a otras distribuciones. Sin embargo, trabajar con Ubuntu resulta, tal y cómo dicen en su propia web [14], rápido, seguro y simple. En el Anexo A. Manual de usuario, se pueden encontrar algunos consejos a la hora de instalarlo junto a Windows.

Figura 3-5. Ubuntu 14.04 LTS

ROS cuenta con más de 10 versiones diferentes, las cuales van saliendo casi a cada año. Siempre hay más de una activa al mismo tiempo, pero las más antiguas pueden estar ya en desuso [15]. Para este proyecto empleamos una versión compatible con Ubuntu 14.04 LTS, que no es tan novedosa con el fin de que esté consolidada: **ROS Indigo Igloo**, versión número 8 de ROS lanzada el 22 de junio de 2014. En el Anexo A. Manual de usuario, se muestra cómo instalarlo en nuestra sesión de Ubuntu.

Figura 3-6. ROS Indigo Igloo

Por último, para la cámara usaremos OpenCV. Se trata de una librería abierta de funciones muy potente para el tratamiento de imágenes. En ella se trabaja con código C++, por lo que se puede integrar con ROS. Nos encargaremos de esta librería en el capítulo 5.

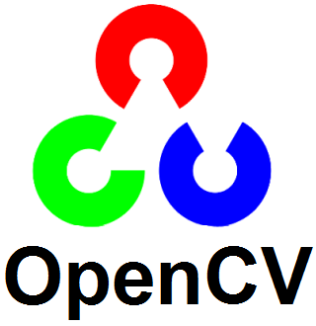

Figura 3-7. Logo de OpenCV

3.2 Trabajos previos con el Summit-XL: navegación autónoma con mapeado por sensor láser

Previo a la realización de este trabajo, hubo dos compañeros que trabajaron recientemente con el robot. En [16] se logró la navegación del Summit-XL de manera autónoma sin conocimiento previo del medio. Para ello se basaba en técnicas de navegación denominadas SLAM (Simultaneous Localization and Mapping) y en mapas de coste o costmap. Estas técnicas requieren obtener información de la odometría del robot y de los sensores como el láser. Todo ello debe ser filtrado con un filtro de Kalman extendido (EKF) para obtener un resultado lo más preciso posible y actualizado continuamente por si cambian las condiciones del medio o se observa algo que antes no se veía.

Sin embargo, para lanzar el mapeado se usaba un ejecutable que viene por defecto en la instalación de ROS. Más adelante veremos esta alternativa y cómo podemos acceder a ese código para modificarlo e integrarlo en nuestro trabajo.

El trabajo de [16] estaba limitado por el hecho ya comentado de la altura a la que se encuentra el láser, pues al ser navegación autónoma, si no ve un obstáculo que no llegue a la altura del sensor (por debajo de la línea amarilla en la Figura 3-8; es decir, la zona naranja), chocará con él. Por tanto, se trata de continuar ese trabajo, pero con un paso intermedio.

Figura 3-8. Zonas de visión del Summit-XL (Fuente: [16])

Es decir, el fin último debe ser que el robot navegue de manera autónoma en sus tareas de vigilancia pero con la información conjunta del láser y la cámara que apunte al suelo. El paso intermedio es obtener esa información pero con el robot teleoperado, de lo cual nos encargamos nosotros. Realmente, al manejarlo nosotros y apuntar la cámara hacia el terreno, parece que no hiciera falta integrar la información del suelo en el mapa. Sin embargo, es necesario mover el robot a nuestro antojo y comprobar que realiza el mapa de obstáculos que esperamos, todo ello como paso previo antes de lanzar una navegación autónoma y crear algoritmos de planificación en un futuro.

En [17], se lleva a cabo navegación del robot tanto de manera autónoma de una manera similar a [16] como basada en teleoperación haciendo uso de mandos de videoconsolas como la PS3. Pero el objetivo de ese trabajo está más centrado en la forma de obtener la energía por parte del robot, para poder aumentar la autonomía de su batería. Por tanto, no continuaremos por esa línea, pues nos centramos en el mapeado como se dijo antes.

4 ROS

Dedicamos este capítulo a hablar de la principal herramienta utilizada: ROS. Se dará desde nociones básicas y generales hasta llegar a particularizar el uso de este sistema operativo en el Summit-XL. Se recuerda que la versión escogida es ROS Indigo Igloo y que para su instalación nos remitimos al Anexo A. Manual de usuario. Además, en dicho manual se incorpora un apartado con comandos útiles para trabajar en la terminal con ROS.

4.1 Visión general: conceptos básicos

ROS, tal y como se definen en su propia web [18], es un framework pensado para el desarrollo de software de cuantas plataformas robóticas sea posible. Debido a que no es tarea sencilla, la filosofía de este metasistema operativo es la cooperación entre distintos desarrolladores. Así, la programación de los robots es modular de forma que cada subtarea del mismo pueda pertenecer a programadores distintos. Además, ROS es accesible a todo el mundo y proporciona una enorme cantidad de herramientas para trabajar. Posee licencia BSD.

Para entender el funcionamiento de ROS hay que dividir los conceptos en varios niveles [19] y [2]: nivel de sistema de archivos, nivel gráfico y nivel de comunidad.

4.1.1 Nivel de sistema de archivos

Para trabajar con ROS hay que crearse y configurar un espacio de trabajo que se conoce como workspace. Dentro de ese workspace podemos encontrarnos:

- **Packages** (o paquetes): son la unidad en la que se organizan los códigos, librerías, ejecutables y otros archivos necesarios para la ejecución de procesos o nodos, los cuales realizarán alguna tarea concreta.
- **Repositories** (o repositorios): son agrupaciones de conjuntos de packages que pueden ser lanzados a la vez porque comparten versiones.
- **Metapackages**: packages especiales para representar a una grupo de paquetes relacionados, por ejemplo dentro de un repository.
- **Package Manifests**: archivos que se encuentran dentro de los packages y que se escriben en el meta-lenguaje xml (eXtensible Markup Language). Siguen una estructura predefinida y contienen toda la información relativa al package, como pueden ser el autor del mismo o las dependencias.
- **Message (msg) types** (o tipos de mensaje): archivos de extensión .msg que contienen el tipo de dato de todos los campos que se mandan en un mensaje, dejándolo completamente definido.
- **Service (srv) types** (o tipos de servicio): archivos de extensión .srv que contienen el tipo de dato de todos los campos del mensaje de petición y del mensaje de respuesta de un servicio, dejándolo completamente definido.

En la Figura 4-1 observamos cómo se organizan los conceptos explicados anteriormente dentro del sistema de archivos. En ella aparecen una serie de archivos no explicados:

- **build**: directorio donde cmake y make van a configurar y compilar los packages. Cmake y make son las herramientas usadas para construir los packages. Se usará cmake y la configuración necesaria para la perfecta construcción de los packages se encuentra en los archivos CMakeLists.txt.
- **devel**: directorio donde se guardan las librerías y los ejecutables de todos los packages del workspace.

- **include**: directorio donde se encuentran las cabeceras para los códigos C++.
- **launch**: directorio para los archivos de extensión .launch, los cuales servirán para lanzar más de un proceso o nodo a la vez. Son tipos de archivos muy útiles, pues las tareas del robot son complejas y necesita iniciar una serie de procesos para su funcionamiento.
- **msg** y **srv**: directorios donde se almacenarán los archivos de extensión .msg y .srv.
- **src** (dentro de un package): directorio en el que se encuentran los códigos fuente. Estos códigos pueden ser escritos en lenguaje C++ o Python.

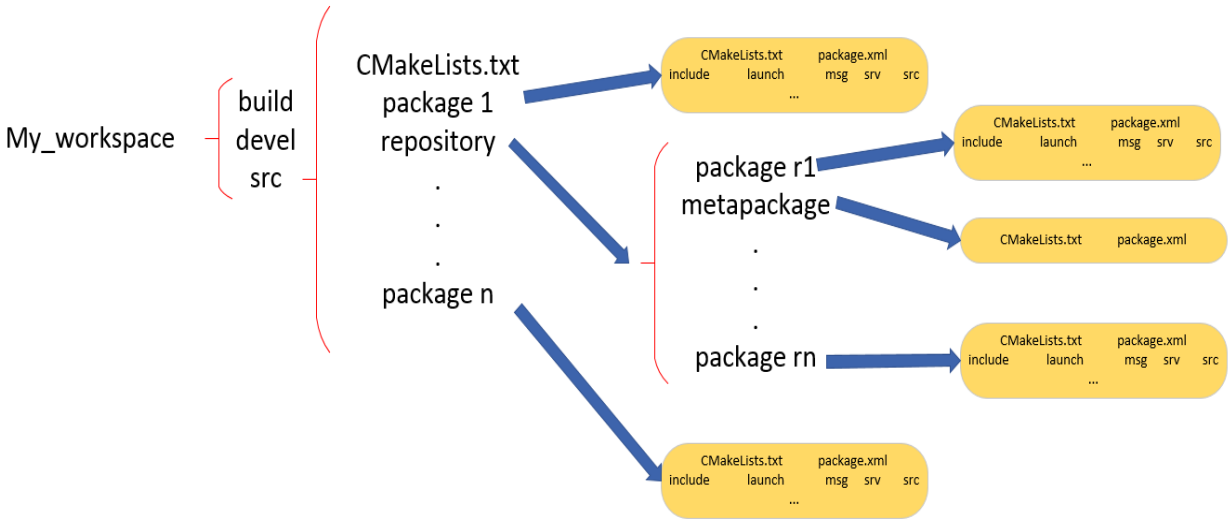

Figura 4-1. Estructura del sistema de archivos de ROS

4.1.2 Nivel gráfico

ROS viene a simular sistemas de tiempo real, por lo que su forma de trabajar se basa en la ejecución de procesos en paralelo que comparten información entre ellos. Un mismo proceso puede tanto generar información como recibirla de otros procesos. El resultado es una red de comunicación (Figura 4-2) en la que se comparten datos en tiempo real para conseguir un fin común. Cada elemento del grafo tiene que tener un nombre único, estructurado en espacios de nombres ("namespaces") [20]. Para entender mejor el funcionamiento gráfico de ROS es necesario definir una serie de conceptos [19] y [2]:

- **Nodes** (o nodos): es la manera de denominar a los procesos. Cada robot tiene una gran cantidad de nodos, dedicados cada uno a una tarea distinta. Se escriben en código C++ o Python y se pueden lanzar en multitud con archivos .launch.
- **Master**: el ROS Master permite a los nodos conectarse entre sí (que lo hacen directamente a través del protocolo TCP/IP) para compartir información, por lo que es una pieza fundamental.
- **Parameter Server**: Complementa al Master y permite almacenar los datos de manera ordenada. Una forma de iniciar tanto el Master como el Parameter Server es ejecutar el nodo **roscore** (basta con escribir en la línea de comandos de la terminal la palabra roscore). Esto se debe hacer siempre que vayamos a trabajar con ROS, a no ser que ejecutemos un archivo .launch con el comando roslaunch, pues este iniciaría roscore en caso de que no estuviera ejecutándose (es la manera que empleamos con el Summit-XL).
- **Topics** (o temas): son el canal a través del cual se comunican los nodos entre sí. Los nodos pueden publicar información en un topic (nodo **publisher**) o suscribirse a ellos (nodo **suscriber**) para tener acceso a la información que publica otro nodo. Se evita así mezclar la información de producción de datos con su recogida, pues el nodo que publica no sabe a quién llegará su mensaje y el que se suscribe no sabe quién lo genera. Puede haber tantos publicadores y suscriptores como se quiera en un mismo topic.
- **Messages** (o mensajes): contienen los datos que se pasan unos nodos a otros. Los mensajes se

comparten a través de los topics.

- **Services** (o servicios): son otra forma de compartir información en la que se requiere paso de mensajes en ambos sentidos, no sólo de un nodo a otro. Por tanto, tendremos uno o varios nodos **client** (o cliente) que lanzarán una pregunta (**request**) y un único nodo **server** (o servidor) que le devolverá una respuesta (**response**). Por ejemplo, un nodo cliente puede pasarle dos números al nodo servidor y que este le devuelva el mayor de ambos.
- **Bags**: son un tipo de archivo que permiten almacenar datos producidos en la ejecución de un sistema ROS y reproducirlos más adelante.

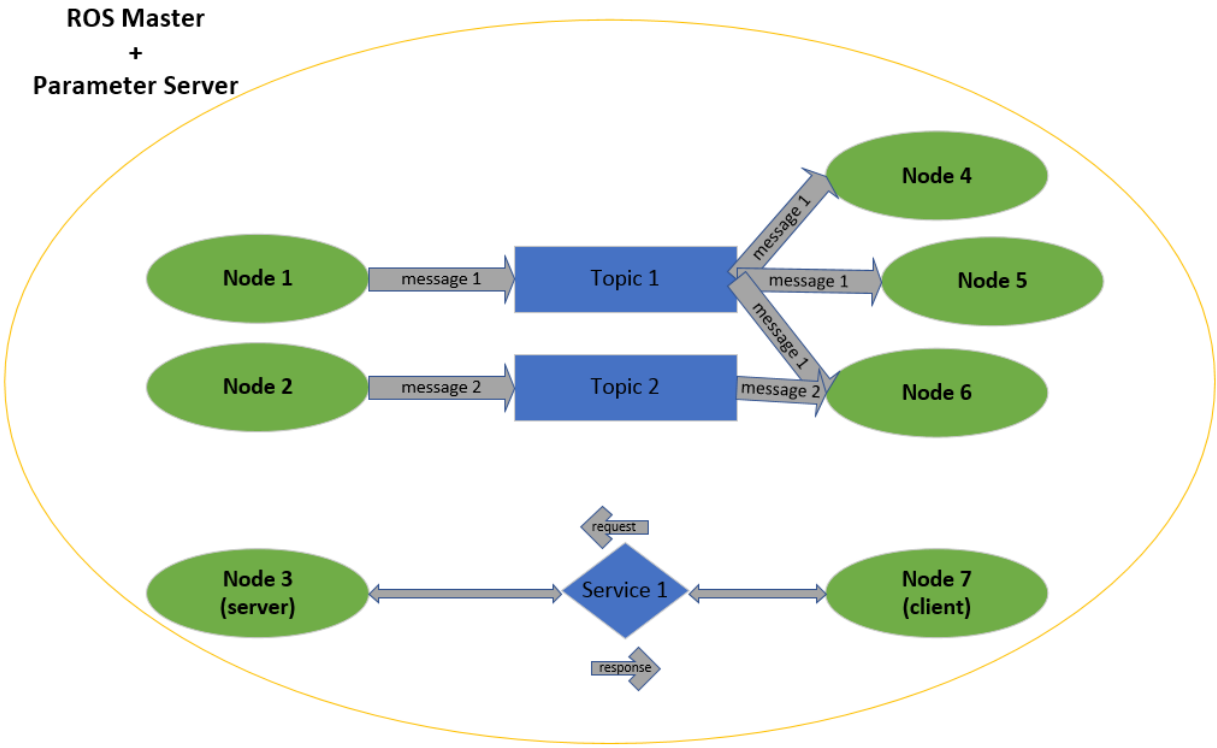

Figura 4-2. Esquema de conexiones gráficas en un sistema ROS

4.1.3 Nivel de comunidad

ROS, como ya se ha comentado, se basa en la cooperación de una gran cantidad de colaboradores. Existe mucha información disponible para todo el mundo que igualmente puede ser compartida por cualquier persona, simplemente registrándote en la comunidad ROS.

Existen **distribuciones** y **repositorios** desde donde obtener **stacks** (o pilas) y código, respectivamente. Los stacks son un mecanismo de agrupación de packages que facilita la conexión entre los mismos.

Para consultar información se tienen varios recursos muy útiles:

- **Wiki de ROS** [21]: colección de recursos relacionados con ROS, entre los que podemos encontrar tutoriales o enlaces para colaborar con la comunidad ROS.
- **ROS Answers** [22]: extenso foro en los que plantear preguntas acerca de ROS y encontrar respuestas por parte de otros usuarios.

4.2 Entorno de simulación: Gazebo

Debido a que este trabajo está basado en simulación, es fundamental contar con una herramienta que sea capaz imitar la realidad con bastante precisión para que nuestros trabajos puedan llevarse a cabo en el robot real en un futuro. Para ello, ROS cuenta con Gazebo, que es un entorno de simulación 3D que posee muchos escenarios, robots, sensores y herramientas para proporcionar la precisión buscada en pruebas de diseño y

programación de robots [23]. Se necesita un buen procesador gráfico del PC, puesto que los gráficos de gazebo son de alta calidad.

Figura 4-3. Logo del simulador Gazebo

Gazebo ofrece simulaciones realistas en la que los cuerpos actúan con normalidad [24]: existe gravedad que no permite que estén flotando, no se puede atravesar un cuerpo sólido o una colisión del robot puede dañarlo. Se interpretan los modelos de escenarios o mundos (worlds) y los de los robots, escritos en URDF (Unified Robot Description Format), que son archivos de extensión .xml. Podemos crear nuestro propio robot, con su escenario y obstáculos o reutilizar alguno de los muchos ya existentes.

Para conectarse con ROS se usan los ROS API's (Application Programming Interface), que son subrutinas para conectar procesos. Al fin y al cabo, iniciar Gazebo es iniciar un nodo que publica información de la realidad simulada y, a través de un código en ROS, podemos obtener la información que se publica. Igualmente podemos publicar información desde ROS para mandar órdenes al robot que estamos simulando. Un uso común de API's son los plugins de Gazebo [23], que son rutinas independientes a través de las cuales se puede obtener información de Gazebo de manera rápida. Entre ellos destacan los de tipo sensor, que servirán para obtener la información que recoja algún sensor.

4.3 Herramientas complementarias

4.3.1 RViz

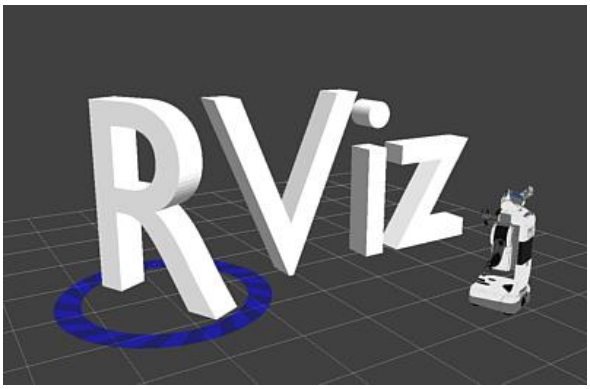

Figura 4-4. Logo de RViz

Es una herramienta de visualización de datos del robot en tiempo real [2]. Se puede ver el robot en su modelo 3D, la imagen obtenida por la cámara, el barrido del sensor láser o la información publicada por la odometría, por ejemplo. Para ello, en su interfaz gráfica se pueden añadir los denominados "Displays", que tras configurarlos para recibir información de un topic que se esté publicando y sea compatible con el Display elegido, recrearán en la pantalla lo elegido. Por ejemplo, con el display "TF" podemos visualizar los ejes de todas las articulaciones del robot y cómo se mueven a medida que avanza la simulación (ver Figura 4-5).

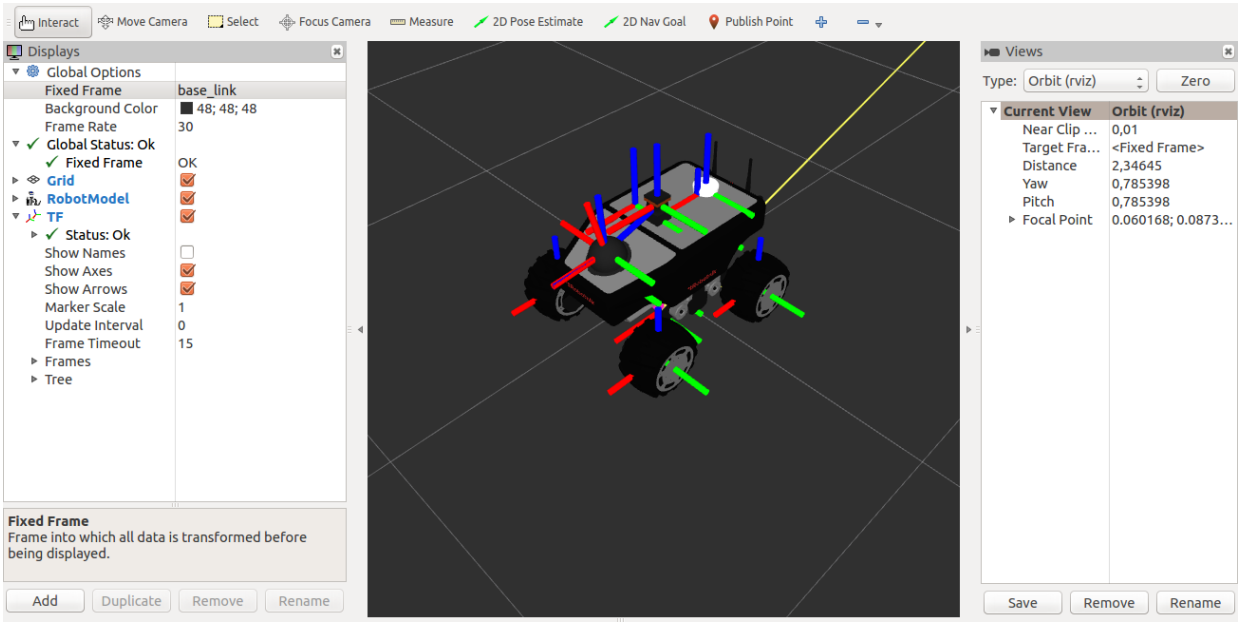

Figura 4-5. Vista de los ejes de las articulaciones del Summit-XL en RViz

En nuestro caso, emplearemos RViz para recrear el mapa de obstáculos que estamos generando con la información de la cámara y el láser. Se empleará para ello el display "Map".

4.3.2 rqt_graph

Se trata de una funcionalidad de ROS que nos permite ver los nodos activos en el momento y la conexión entre ellos a través de los topics [2]. Es muy útil para comprobar que un nodo se ha suscrito a un determinado topic o para ver qué nodo es el que genera un determinado mensaje. En la Figura 4-6 podemos ver un ejemplo del grafo de nodos y topics que se obtiene al ejecutar rqt_graph en nuestro sistema (en el Anexo A. Manual de usuario aparece como acceder a él desde la línea de comandos).

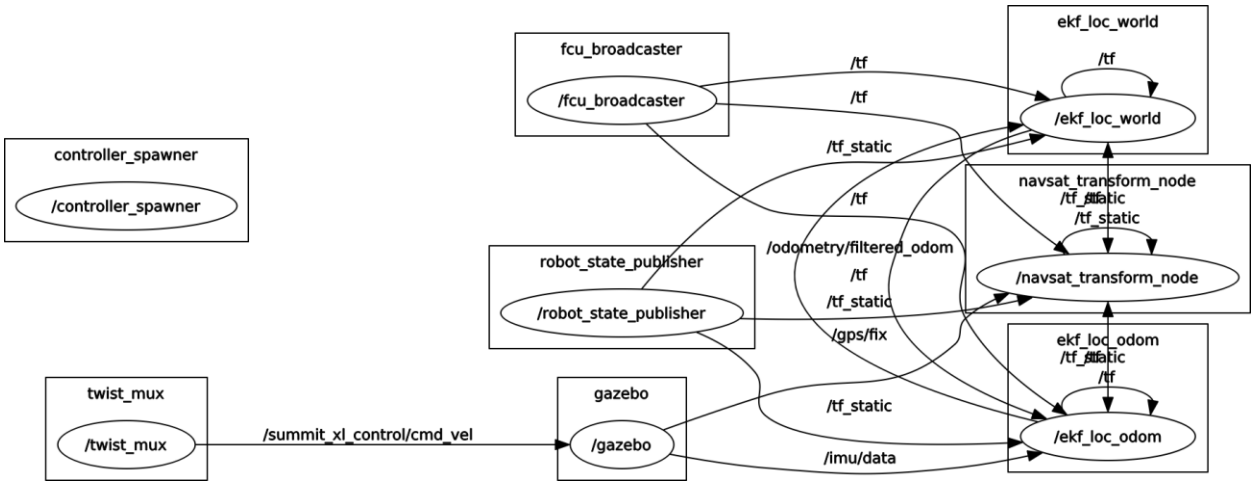

Figura 4-6. Ejemplo de uso de rqt_graph

4.4 Estado del arte: ROS en el Summit-XL

Para trabajar con ROS se tienen dos opciones: programar un robot desde cero o reutilizar lo que ya han programado otras personas. Sin duda, lo más extendido es lo segundo y, para ese caso, se suele partir de una situación en la que ya se tiene el modelo del robot creado y simulaciones del mismo dispuestas a ser lanzadas. Es el caso del Summit-XL, cuyos códigos son accesibles gracias a la empresa Robotnik, creadora y distribuidora del mismo.

Se pasa en esta sección a hablar de los packages de partida que tiene el Summit-XL, así como del lanzamiento de una primera simulación del mismo. También se comentará acerca de la teleoperación del robot.

4.4.1 Software de partida: packages existentes para el robot

Vamos a realizar una breve descripción de los packages. La obtención de los mismos es objeto del Anexo A. Manual de usuario.

4.4.1.1 summit_xl_common

Contiene los archivos que definen al robot y permiten su simulación [25]. En él encontramos los siguientes packages:

- **summit_xl_description:** en él se encuentran todos los archivos necesarios para que el robot quede definido.
- **summit_xl_localization:** archivos que permiten la localización del robot mediante el uso de la odometría y el EKF (Extended Kalman Filter).
- **summit_xl_navigation:** sirve para llevar a cabo las técnicas de navegación AMCL y SLAM.
- **summit_xl_pad:** permite conectar una teleoperación externa (por ejemplo un mando de videoconsola) para que publique sobre el robot y lo haga moverse. También se puede controlar la configuración PTZ de la cámara.

4.4.1.2 summit_xl_sim

Contiene los lanzadores para realizar una simulación [26]. Consta de:

- **summit_xl_gazebo:** en este package se encuentran los lanzadores necesarios para obtener los modelos tanto del robot como del mundo o escenario de simulación.

Figura 4-7. Robot simulado en gazebo

- **summit_xl_robot_control:** contiene lo necesario para controlar el robot en la simulación en gazebo, como sus 4 ruedas en configuración skid-steering.
- **summit_xl_sim_bringup:** lanzadores para llevar a cabo la simulación completa.

4.4.1.3 robotnik_msgs

Este package almacena la descripción de todos los mensajes y servicios de los packages de Robotnik, como es son los del Summit-XL [27].

4.4.1.4 robotnik_sensors

Encontramos dentro la descripción URDF de todos los sensores de Robotnik [28]. Tenemos, por ejemplo, las descripciones del sensor láser, la IMU, el GPS o la cámara PTZ.

4.4.1.5 robotnik_trajectory_suite

Es un metapackage que se usó para trabajos anteriores con el Summit-XL pero que para este no se usará. Consiste en planificadores para la navegación autónoma y control del robot [29].

4.4.2 Lanzamiento de una simulación completa

Una vez tenemos los packages descargados y ordenados y nuestro workspace preparado, podemos lanzar una simulación en gazebo para probar que todo funciona bien. Se opta por lanzarla en un escenario que trae los packages del Summit-XL que simula un recinto de interiores en el que hay muchas habitaciones con paredes y pasillos (ver Figura 4-8). Es un buen escenario para, más adelante, probar la construcción del mapa de obstáculos. Recordamos que esto es una simulación, por lo que no tiene que ser estrictamente un terreno de exteriores. Para la preparación del workspace y el lanzamiento de la simulación en distintos mundos nos remitimos al Anexo A. Manual de usuario.

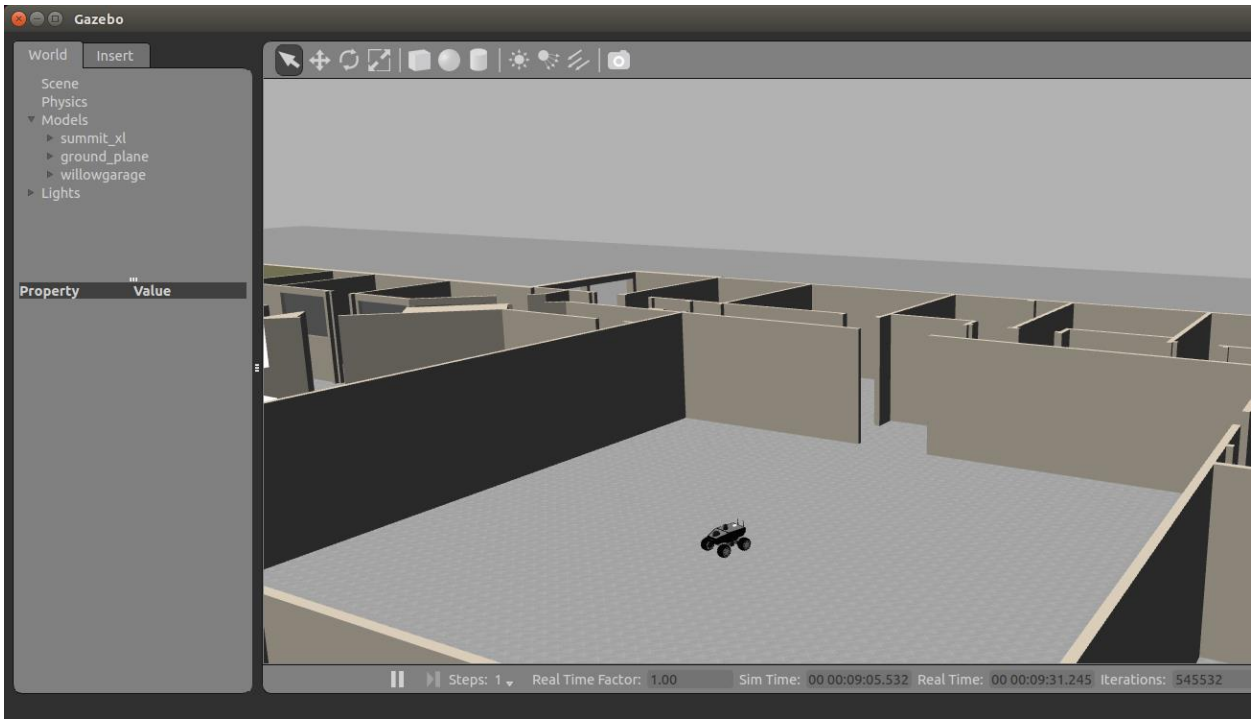

Figura 4-8. Lanzamiento de una simulación en gazebo

4.4.3 Teleoperación

En el apartado anterior se lanzaba una simulación. Pero esa simulación hay que complementarla con funcionalidades que afecten al robot, bien sea para controlarlo o bien para tratar la información que capta del medio. El primer paso para esto será hacer que se mueva. Como se dijo al principio, se opta por la teleoperación. Se podría hacer con algún mando de control externo como el de una videoconsola; sin embargo, por comodidad, se elige una teleoperación que será dirigida por las flechas del teclado.

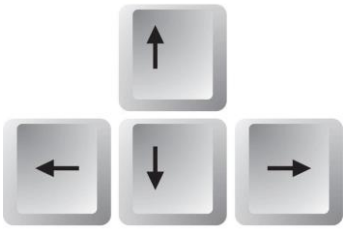

Figura 4-9. Flechas del teclado con las que se dirigirá el robot

La flecha delantera implicará que el robot se mueva hacia delante con una velocidad igual a 0.8 m/s [30]. La trasera le hará moverse hacia atrás a 0.5 m/s y las de derecha e izquierda llevarán consigo un giro a velocidad de 1 m/s. La información proporcionada por las flechas se publicará en el topic /cmd_vel, y será transformada para adaptarla a la configuración skid-steering que tiene el robot y conseguir traducir el movimiento de manera correcta.

Nuevamente, la manera de lanzar la teleoperación y cómo manejarla se recoge en el Anexo A. Manual de usuario. En la Figura 4-10 podemos ver, marcada en rojo, la pantalla de teleoperación que obtenemos con una simulación en gazebo de fondo. Los valores del campo "Linear" se actualizarán si pulsamos las flechas de arriba o de abajo y los del campo "Angular", si tratamos de girar con las flechas de izquierda y derecha.

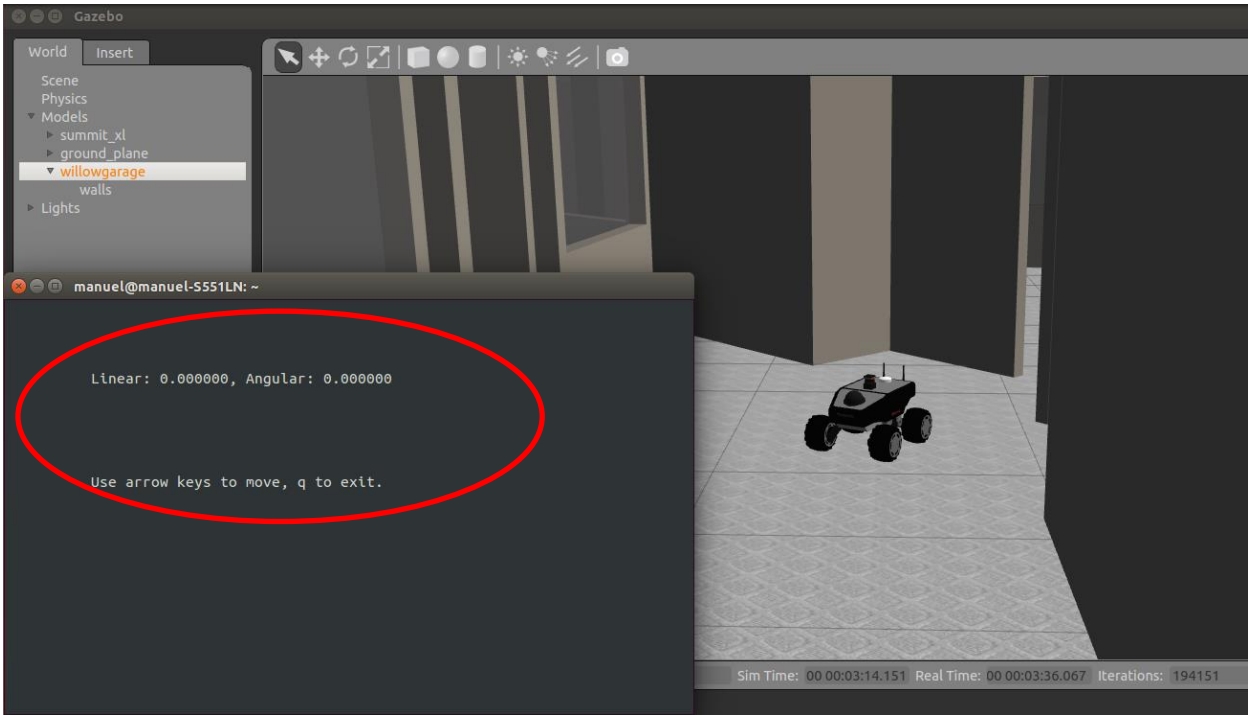

Figura 4-10. Panel de teleoperación en una simulación

5 OPENCV

Se trata de introducir la librería de funciones para tratamiento de imágenes con el fin de obtener una visión general de la misma. En un trabajo como este es muy importante el tratamiento que se le da a la imagen recibida y se verá que, con esta librería, se pueden llevar a cabos infinidad de métodos para trabajar con imágenes. La forma de conectar la librería con ROS es un aspecto importante que también se explicará en este capítulo.

5.1 Introducción a la librería y conexión con ROS

OpenCV es una librería de funciones de código abierto para el tratamiento de imágenes, ideal para aplicaciones de tiempo real [31]. Al igual que ROS, tiene una gran comunidad de colaboradores y existen tutoriales para aprender a manejarla, así como un foro propio para preguntas y repuestas. También tiene licencia BSD. El código fuente que usa esta librería puede estar escrito en los lenguajes C++, Java o Python y es soportada por diferentes sistemas operativos, entre los que se incluye obviamente Linux (y Ubuntu, por tanto).

El stack o pila para usar OpenCV en ROS debemos tenerlo instalado siguiendo los pasos dados hasta ahora en el Anexo A. Manual de usuario. En caso de no tenerlo, habría que descargarlo manualmente.

El nombre del stack es **visión_opencv** y contiene, a su vez, dos packages [32]:

- **image_geometry**: contiene una gran cantidad de funciones para realizar el tratamiento de imágenes a través de una geometría basada en píxeles. Aunque la información que se obtiene en ROS viene en forma de imagen preparada, este package permite realizar una serie de ajustes en la imagen que mejoran y facilitan su tratamiento.
- **cv_bridge**: package que sirve de conexión o puente entre la interfaz de la cámara en OpenCV y la propia interfaz de ROS. Permite convertir los mensajes propios de OpenCV en mensajes propios de ROS y viceversa. Esto es fundamental, pues podemos trabajar con la gran cantidad de funciones que nos proporciona OpenCV pero con un tipo de dato que este puente hará compatible con ROS, por lo que podemos publicar resultados interesantes (incluso la imagen tratada) en algún topic como conclusión de un tratamiento de imagen con OpenCV.

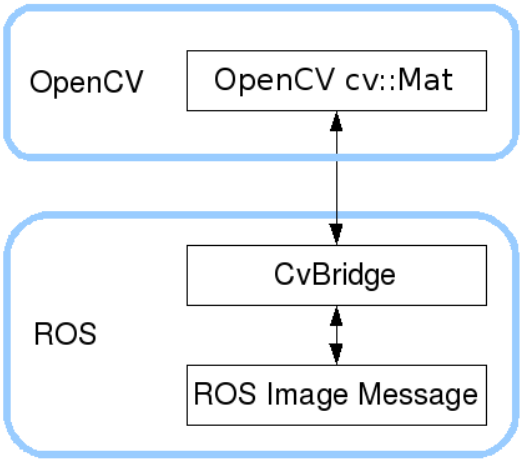

Figura 5-1. Conexión de OpenCV con ROS a través de cv_bridge (Fuente: [33])

5.2 Manejo básico con imágenes y funciones de OpenCV

Como se ha explicado, tomamos información con la cámara en gazebo (por tanto en ROS) y la trataremos con la librería OpenCV. Básicamente, lo que hará un código fuente de OpenCV en ROS como el nuestro será lo siguiente [33]:

- Suscribirse al topic que contiene la información de la cámara (en nuestro caso **summit_xl/camera1/image_raw**). Esta información es un tipo de mensaje de ROS concreto: sensor_msgs/image.
- Convertir esa información a través del puente cv_bridge y volcarlo al formato típico del tratamiento de imágenes: una matriz de píxeles, denominado **cv::Mat**. Para ello hacemos uso de la función **cv_bridge::toCvCopy**, la cual nos devuelve la imagen de ROS en el tipo de dato **cv_bridge::CvImagePtr**, compatible con OpenCV. Ese tipo de dato es una estructura que contiene en el campo *image* la imagen con la que vamos a trabajar en esta librería. El hecho de usar cv_bridge::toCvCopy y no **cv_bridge::toCvShare** es que vamos a modificar la imagen original y, en esos casos, hay que guardar una copia de los datos de imagen; eso lo hace cv_bridge::toCvCopy.

  Es conveniente recordar que las imágenes se representan en digital de manera discreta a través de una matriz de puntos o píxeles [34]. Cada píxel corresponde a una zona de la imagen y tendrá un valor de intensidad que marcará su color. Cada píxel tendrá un único valor de intensidad (imágenes en blanco y negro) o tres (imágenes a color, siendo las intensidades las correspondientes a los colores rojo, verde y azul y cuya combinación da lugar al color final).

  Una forma muy usada de acotar el valor de estas intensidades es usar una representación de 8 bits. Por tanto, podremos obtener hasta 256 valores diferentes de intensidades ($2^8 = 256$). Esto se traduce en que cada uno de nuestros píxeles puede tomar un valor de intensidad comprendido entre 0 y 255 (en imágenes en blanco y negro o en cada canal de color de una imagen a color).
- Realizar el tratamiento de imágenes en sí usando las funcionalidades propias de OpenCV. Se puede hacer uso de métodos de separación de regiones, búsqueda de contornos, reconocimiento de colores o formas, así como técnicas previas al tratamiento en sí que sirven para preparar la imagen. Con estas últimas nos referimos a suavizados, mejoras del brillo o contraste o algún filtro interesante que pueda ser fruto de nuestro conocimiento de la situación. Por ejemplo, en nuestro caso vamos a reconocer agujeros oscuros, por lo que un filtro interesante puede ser separas las regiones de la imagen en dos: zona oscura y zona clara.

  Para el tratamiento de la imagen se han usado las siguientes funciones [35]:

  - **cv::cvtcolor**: permite pasar de una escala de colores a otras. Lo empleamos para pasar la imagen original (en escala RGB: Red, Green, Blue) a escala de grises.
  - **cv::threshold**: con esta función aplicamos el filtro zona clara-zona oscura del que hablamos anteriormente. Se le pasa como argumento el umbral según el cual un píxel pertenecerá a una zona u otra en función de que su intensidad supere o no ese valor. Finalmente, se obtiene una matriz con dos valores de intensidad posible para los píxeles: 0 (que será color negro, para la zona más oscura) o 255 (que será blanco, para la zona más clara).
  - **cv::Canny y cv::findContours**: con la primera obtenemos una imagen de contornos usando el algoritmo de Canny, que sirve para calcular el gradiente de una matriz de imagen para detectar cambios bruscos de intensidad y posibles candidatos a contornos. Para decidir si el gradiente calculado de un píxel con sus vecinos es suficiente para considerarlo parte del contorno, es necesario usar un umbral que conviene ajustar de manera experimental. La segunda función permite rellenar un vector con los puntos (coordenadas de la matriz) que pertenecen al contorno de la matriz obtenida anteriormente.
  - **cv::drawContours**: nos permite dibujar en una matriz de imagen el contorno calculado anteriormente y almacenado en forma de vector de puntos. Una buena forma es crearse una matriz del tamaño de la imagen (mismo número de filas y columnas) cuyos elementos valgan todos 255 (imagen entera en blanco). Sobre esta imagen se dibujan los puntos que pertenecen al contorno con esta función y en el color que queramos: escogemos negro o valor de

intensidad 0.

- **cv:minAreaRect**: a esta función se le pasa el vector de puntos del contorno y te calcula el rectángulo de menor área que encierra a todos esos puntos.
- **cv:fitEllipse**: igual que la anterior, pero calcula la elipse que mejor se ajusta y deja en su interior todos los puntos del contorno. Esta función es muy interesante, pues queremos reconocer agujeros en el suelo que tienen forma circular, los cuales vistos desde la altura del chasis, donde está la cámara, se ven como elipses. De esta manera, podemos englobar el contorno de nuestro obstáculo objetivo de manera muy precisa**.**
- **cv::ellipse**: usamos esta función para dibujar sobre la imagen original la elipse calculada que representa el contorno del obstáculo de nuestra imagen.
- **cv::imshow**: muestra en una ventana nueva la imagen que le pasemos como argumento en forma de matriz. En nuestro caso, representaremos la imagen original con la elipse que representa al contorno, dibujada sobre el obstáculo en caso de que este exista.

- Una vez hemos tratado la imagen, se saca de ella la información que buscábamos y se prepara con los cálculos necesarios. En nuestro caso será estimar la posición y dimensiones del obstáculo en caso de haber reconocido un agujero oscuro en el tratamiento de la imagen. Esto será explicado en el capítulo 6.
- Publicamos dicha información en un topic para que otros nodos del sistema en ROS pueda tener acceso a ella. El topic en el que publicaremos los datos finales será **/Obstacle_data**, y en él informaremos de la posición y tamaño del obstáculo en caso de encontrarlo.

Hay que destacar la potencia de la librería, pues permite realizar todo lo explicado anteriormente en tiempo real. Es decir a cada instante es capaz de analizar la imagen y reconocer formas, aproximándose a lo que haría el ojo humano, lo cual es una ventaja enorme para estudiar el entorno.

6 INTEGRACIÓN DE VISIÓN EN LA NAVEGACIÓN

Como ya se ha comentado, este es el capítulo más importante de la memoria, pues se recoge en él la filosofía seguida a lo largo de todo el trabajo, tanto en el tratamiento de imágenes como en la construcción del mapa de obstáculos. Tras una breve introducción de la cámara que se usa, se pasa al desglosar el esquema de la metodología seguida en el trabajo.

6.1 Descripción de la cámara

Se hará uso de una cámara PTZ P5512, de la empresa Axis Communications. Esta cámara, tal y como se recoge en su hoja de datos [36], es ideal para tareas de vigilancia, tanto en interiores como en exteriores.

Figura 6-1. Cámara PTZ P5512 (Fuente [36])

Estas cámaras tienen una serie de características:

- Ángulo de barrido horizontal de 360º, con posibilidad de barrido automático.
- Ángulo de inclinación en un rango de 180º.
- Zoom 12x con enfoque automático.
- Visión diurna y nocturna (o con escasa iluminación).
- Protección contra polvo y agua.

6.2 Procedimiento seguido en la integración

Llegar a integrar la información de la cámara en el mapa es una tarea que puede dividirse a su vez en varias subtareas, y sobre todo, partiendo de cero en cuanto a la visión. La generación del mapa con el láser si es algo que ya estaba implementado y simplemente se ha lanzado para comprobar que funcionaba correctamente y, más tarde, añadirle la información de la cámara.

La división del objetivo se realiza según lo que se ve en el esquema de la Figura 6-2.

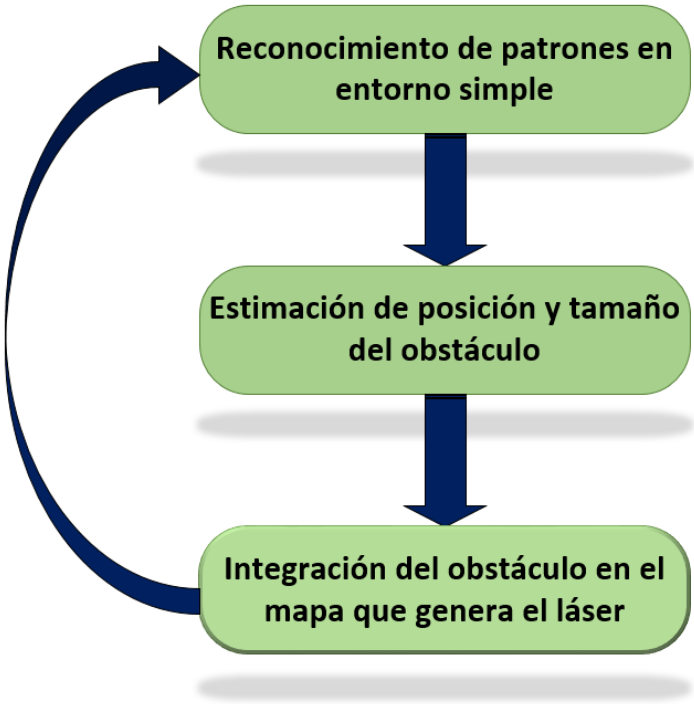

Figura 6-2. Procedimiento de integración de obstáculos en el mapa del láser

Se observan tres pasos claramente diferenciados. Del primero de ellos ya se habló un poco en el capítulo 5. Se trata de leer la imagen del suelo por el que anda el robot en la simulación de gazebo y ser capaz de reconocer agujeros oscuros. Por ello, se estará analizando la retransmisión por cámara del robot continuamente en busca de encontrar un obstáculo. Este será el primer paso y, hasta que no se reconozca algún obstáculo que siga el patrón del agujero, no se saltará a los siguientes pasos.

En caso de reconocer uno, se hará uso del conocimiento de la posición y orientación de la cámara, así como de algunos datos experimentales de calibración, para estimar la posición, orientación y tamaño del obstáculo. Al ser circunferencias, nos basta con calcular 3 datos para definirla: distancias en el eje horizontal y vertical del centro de la circunferencia a la base de la cámara y radio de la circunferencia, tal y como muestra la Figura 6-3.

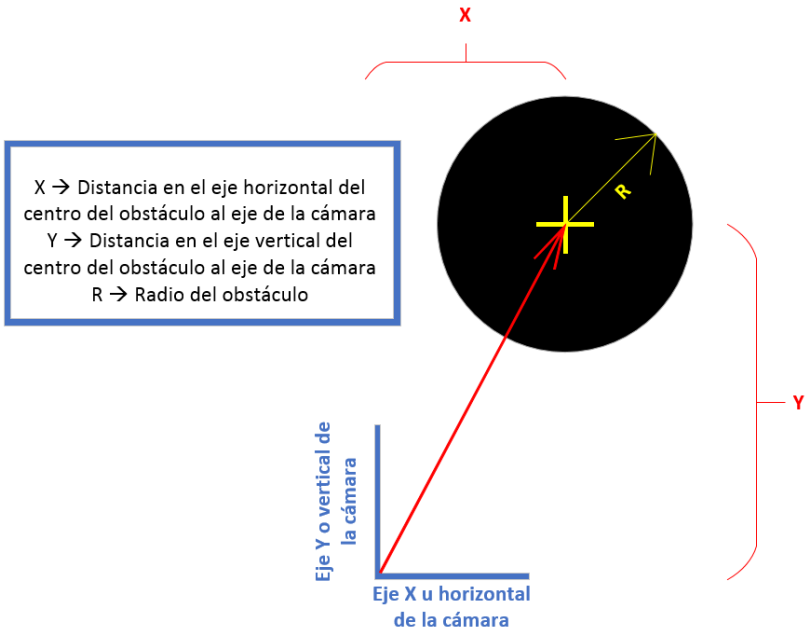

Figura 6-3. Datos relevantes de un obstáculo

Una vez que sepamos esos tres datos, el siguiente paso será publicarlos en un topic para que el nodo que realiza el mapa de obstáculos tenga acceso a ellos. En este paso pues, se realizará el mapa del láser añadiéndole los obstáculos oportunos según mande la cámara, quedando la información mucho más completa.

6.3 Tratamiento de imagen y estimación de posición

Se va a comentar la filosofía seguida en el archivo de nombre image_converter.cpp, que será nuestro código fuente para la cámara. Dicho código se adjunta al final de esta memoria en el Anexo B. Códigos desarrollados.

6.3.1 Reconocimiento de patrones

En primer lugar existe un tratamiento de la imagen que se correspondería con la primera subtarea de este trabajo: reconocimiento de patrones conocidos. Los pasos seguidos se pueden intuir del capítulo anterior y son los siguientes [35] y [37]:

1. Nos suscribimos al topic /summit_xl/camera1/image_raw, donde gazebo publicará la información de la cámara.
2. Creamos una ventana bajo el rótulo “Image window” donde representaremos la imagen tratada. Esta será la original y, en caso de encontrar un agujero, aparecerá dibujada la elipse que mejor se adapte a dicho agujero.
3. Pasamos la imagen recibida a través de la librería puente (*cv_bridge*) del formato a color (RGB) a escala de grises.
4. Convertimos la imagen en escala de grises a blanco y negro. Para ello, establecemos de manera experimental un valor umbral de intensidad. Los píxeles con intensidad comprendida entre 0 y ese umbral, pasarán a valer 0 (es decir, color negro). Por su parte, los píxeles cuyo valor se encuentre entre el umbral y 255, pasarán a ser 255 (color blanco). De esta manera, discriminamos los agujeros oscuros y los diferenciamos claramente del fondo, que se supone que no es de color negro.

   El valor del umbral se puede obtener de manera experimental porque conocemos de antemano la situación, que será sobre un fondo y suelo no oscuros la aparición de posibles obstáculos negros. Se probó un método más flexible pero cuyo resultado era peor que el experimental. Ese era el método de Otsu [34], el cual se aplica a imágenes cuyo histograma presente una clara tendencia bimodal; es decir, haya dos zonas claramente diferenciadas en niveles de intensidad distintos. A través de funciones de probabilidad calcula el valor umbral que separa las dos tendencias de la imagen. En la Figura 6-4 y Figura 6-5 podemos observar la aplicación de este método.

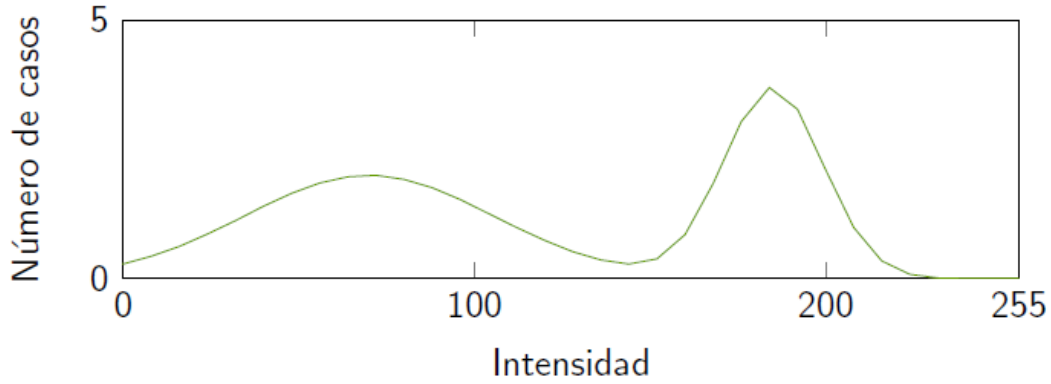

Figura 6-4. Ejemplo de histograma bimodal (Fuente [34])

Se observa claramente dos tendencias de niveles de intensidad en la imagen según el histograma de la Figura 6-4: uno en torno a las intensidades 70-80 y otros en torno a la intensidad de valor 190. Según la Figura 6-5, el valor umbral correcto para separar esta imagen será la intensidad de 149. Como decimos, es un método más flexible, pero buscamos más la precisión que la flexibilidad y eso se consigue con el método experimental.

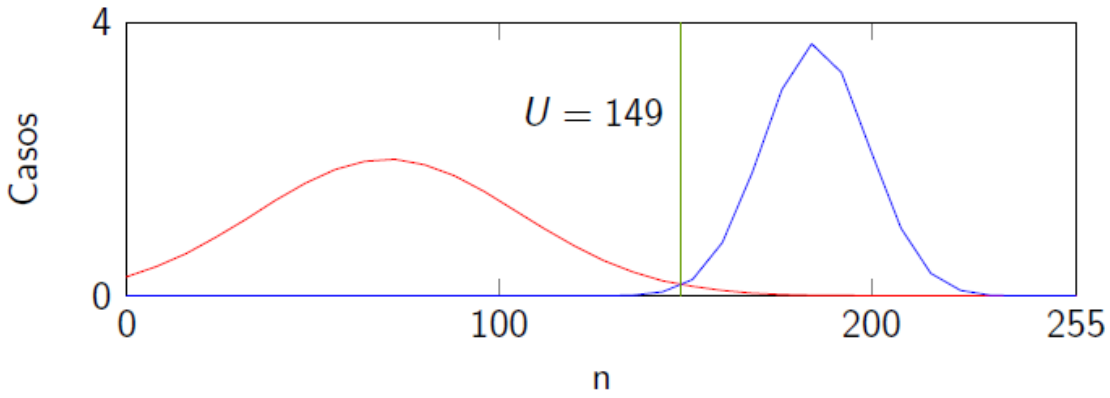

Figura 6-5. Aplicación del Método de Otsu a un histograma bimodal (Fuente [34])

5. A través del método de Canny para calcular el gradiente de una imagen (matriz que nos informa de la variación de cada píxel con sus píxeles vecinos), obtenemos los píxeles candidatos a pertenecer al contorno debido a que superan un determinado umbral de cambio. Así, rellenamos un vector de puntos (posición dentro de la matriz de imagen) con los puntos que pertenecen al contorno según este método.
6. Usamos la información contenida en el vector de puntos del contorno. Para ello, creamos una matriz de imagen cuyo valor de todos los píxeles sea 255 (imagen entera blanca) y le añadimos en intensidad 0 (negro) los puntos del contorno. De esta manera obtenemos una matriz de imagen binaria (valor 0 o 255) de contornos.
7. Obtenemos a partir de esa última imagen el rectángulo y la elipse de menor área que encierran a todos los puntos del contorno y que mejor se adaptan al mismo. El hecho de obtener la elipse se debe a que un círculo en el suelo visto desde una altura (en nuestro caso la cámara está a una altura de 28.75 cm respecto al suelo) se ve como una elipse, por lo que aumentamos así la fiabilidad de nuestro código.
8. Mostramos en la ventana de imagen creada el principio la imagen con la elipse dibujada en el obstáculo en caso de existir.

6.3.2 Estimación de posición y tamaño de obstáculos

Llegados a este punto, lo que tenemos que hacer es interpretar la información de las distancias en píxeles y convertirlas a distancias reales. En la Figura 6-6 podemos verlo más claro. Tendremos una matriz con el origen de píxeles donde se indica y hay que traducir esa información a la imagen real, calculando la distancia en ambos ejes a la cámara en el plano horizontal que contiene a la misma.

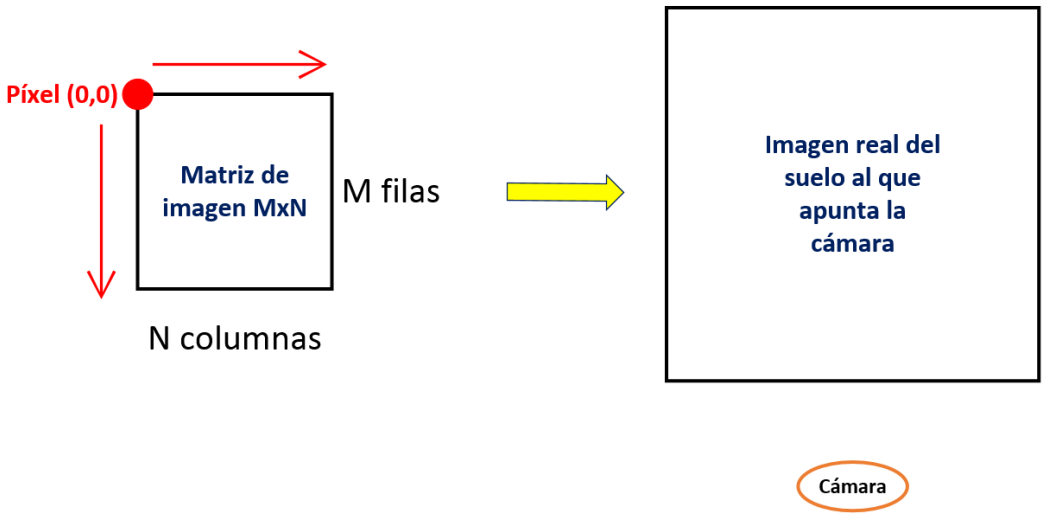

Figura 6-6. Conversión de matriz de imagen a distancias reales

Teniendo en cuenta que conocemos datos como son la altura a la que se encuentra la cámara del suelo (28.75 cm), el ángulo de inclinación de la misma para apuntar al suelo (1.3708 rad ≈ 79º) o el ángulo FOV (Fiel Of View, o campo de visión de la cámara), que es igual a 60º, parece intuitivo que podremos estimar la distancia de cualquier punto que deseemos de la imagen respecto a la cámara.

Esto podría hacerse con cálculos trigonométricos y sacando relaciones en principio lineales. Sin embargo, hay que tener en cuenta otro dato importante: el *far clip*, que es la distancia máxima que se puede ver desde la cámara, y que es igual a 8 m. Si realizamos una simulación en un entorno vacío, simplemente con un suelo y los agujeros de prueba que situemos manualmente, observamos que un obstáculo situado en la distancia máxima que puede percibir la cámara, 8 metros, se corresponde siempre con la fila 230 de la matriz. Otro dato importante es que la matriz tiene 480 filas, que también se comprueba en simulación que dicha fila equivale a una distancia en el eje vertical de la cámara de 0.7 m.

Para completar la información de la distancia en el eje vertical de un obstáculo a la cámara, parece claro que tendremos que calcular una equivalencia entre las filas, comprendidas entre 230 y 480 y las distancias reales, comprendidas entre los 8 y los 0.7 metros, respectivamente.

Para ello, se elabora de manera experimental, calibrando distancias en simulación y comprobando a qué fila de la matriz se corresponde, la Tabla 6-1:

Número de fila en la imagen digital (píxel vertical)	**Distancia vertical en el plano a la altura de cámara (m)**
230	8
235	6.7
240	5.7
245	5.3
250	4.7
255	4.2
260	3.7
265	3.4
270	3
275	2.95
280	2.7
285	2.55
290	2.4
295	2.25
300	2.2
305	2.15
310	2.05
315	1.9
320	1.8
325	1.7
330	1.6
335	1.55
340	1.5
345	1.43
350	1.4
355	1.33
360	1.3
365	1.25
370	1.2
375	1.15
380	1.13
385	1.09

390	1.05
395	1.02
400	1
405	1
410	0.95
415	0.9
420	0.9
425	0.87
430	0.84
435	0.83
440	0.82
445	0.8
450	0.78
455	0.75
460	0.75
465	0.74
470	0.73
475	0.71
480	0.7

Tabla 6-1. Equivalencia fila de la matriz – distancia vertical a la cámara

Esta tabla tendremos que transformarla en una ecuación que nos relacione la fila de la matriz en la que se encuentra el punto en cuestión con la distancia de ese punto en la realidad a la cámara. Tras probar con diferentes curvas (lineales, exponenciales, logarítmicas y polinómicas entre otras), se llega a la conclusión de que la que mejor se adapta a la nube de puntos de la Tabla 6-1 es la de la Figura 6-7, que corresponde a un polinomio de grado 10.

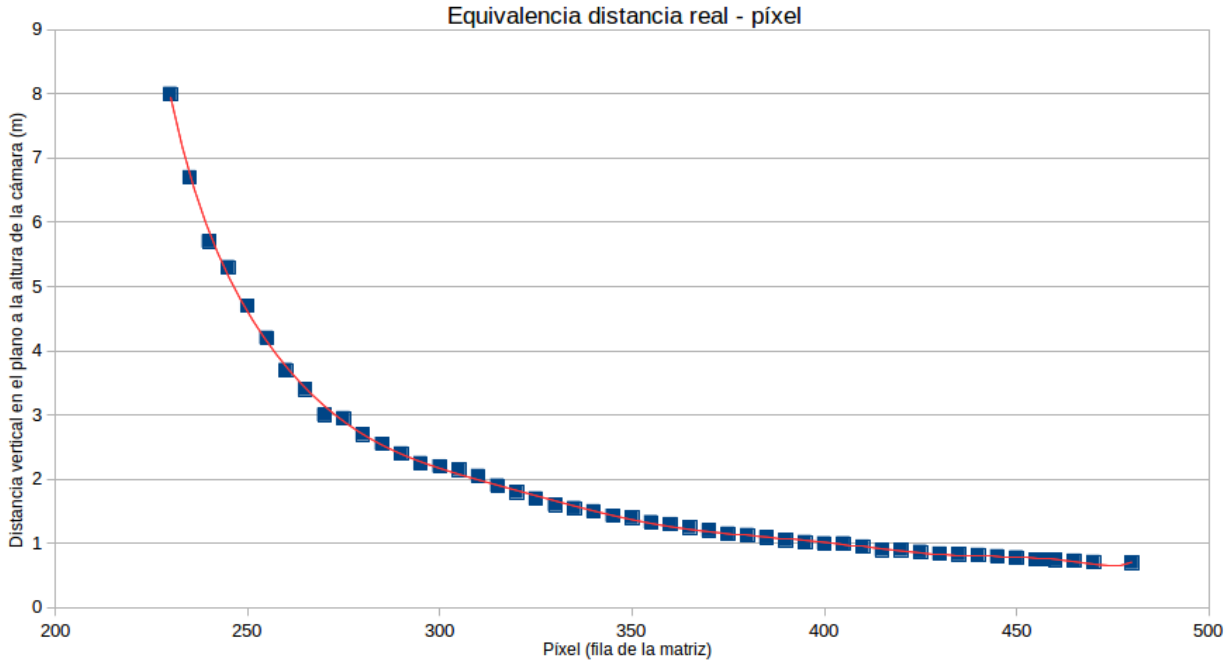

Figura 6-7. Nube de puntos y curva de mejor ajuste para las distancias verticales a la cámara

El polinomio de grado 10 al que corresponde esa curva es el de la ecuación (6–1):

$$f(x) = 1.70221906262318 \cdot 10^{-20}x^{10} - 6.07450388438866 \cdot 10^{-17}x^{9} + 9.6913695011794 \cdot 10^{-14}x^{8} - 9.1020034026841 \cdot 10^{-11}x^{7} + 5.57242765798284 \cdot 10^{-8}x^{6} - 2.32358874787887 \cdot 10^{-5}x^{5} + 0.0066829751x^{4} - 1.3091703182x^{3} + 167.1876205147x^{2} - 12570.5337212198x + 422689.806421953 \quad (6\text{–}1)$$

Siendo *f(x)* la distancia vertical de un punto a la cámara en metros y *x* la fila que ocupa ese punto en la matriz de imagen. Cuando hablamos de distancia vertical, en este caso nos referimos a la distancia "Y" de la Figura 6-8.

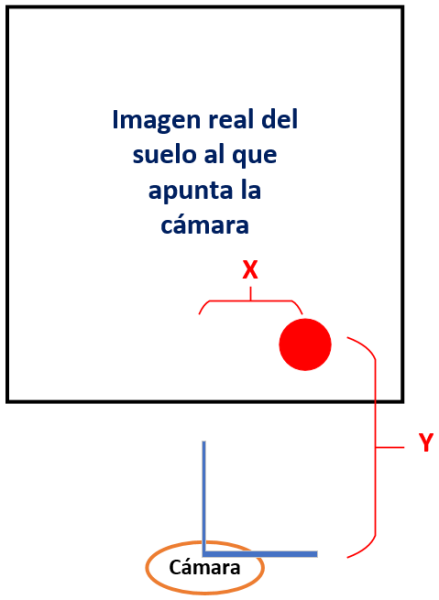

Figura 6-8. Distancias a medir del objeto a la cámara

La otra distancia a calcular será la horizontal o "desviación" a izquierda o derecha respecto a la cámara o centro de la imagen. En la Figura 6-8 nos referimos a la distancia etiquetada como "X". En este caso, de nuevo experimentalmente, observamos que para cada distancia "Y", la anchura de la imagen real es diferente. A mayor distancia de la cámara, se capta mayor anchura en la imagen. Por ejemplo, a una distancia "Y" de 8 metros, la anchura real que la cámara capta es de 8.1 metros y a la distancia "Y" de 1 metro, la anchura será de 1.1 metros.

Por tanto, elaboramos la tabla correspondiente para establecer una relación entre la distancia a la que se encuentra el objeto (comprendida entre 1 y 8 metros) y la anchura que se capta en la imagen (comprendida entre 1.1 y 8.1 metros). La nube de puntos experimental es la de la Tabla 6-1.

Distancia vertical en el plano a la altura de cámara (m)	Anchura máxima real de la imagen (m)
8	8.1
7.5	7.6
7	7.1
6.5	6.6
6	6.1
5.5	5.6
5	5.1
4.5	4.6
4	4.1
3.5	3.6

3	3.1
2.5	2.6
2	2.1
1.5	1.6
1	1.1

Tabla 6-2. Equivalencia distancia a la cámara – anchura real de la imagen

En este caso se observa claramente la relación lineal entre ambos parámetros. La representación de la nube de puntos junto con la recta de mejor ajuste (que se ajusta de manera exacta a todos los puntos) es la de la Figura 6-9.

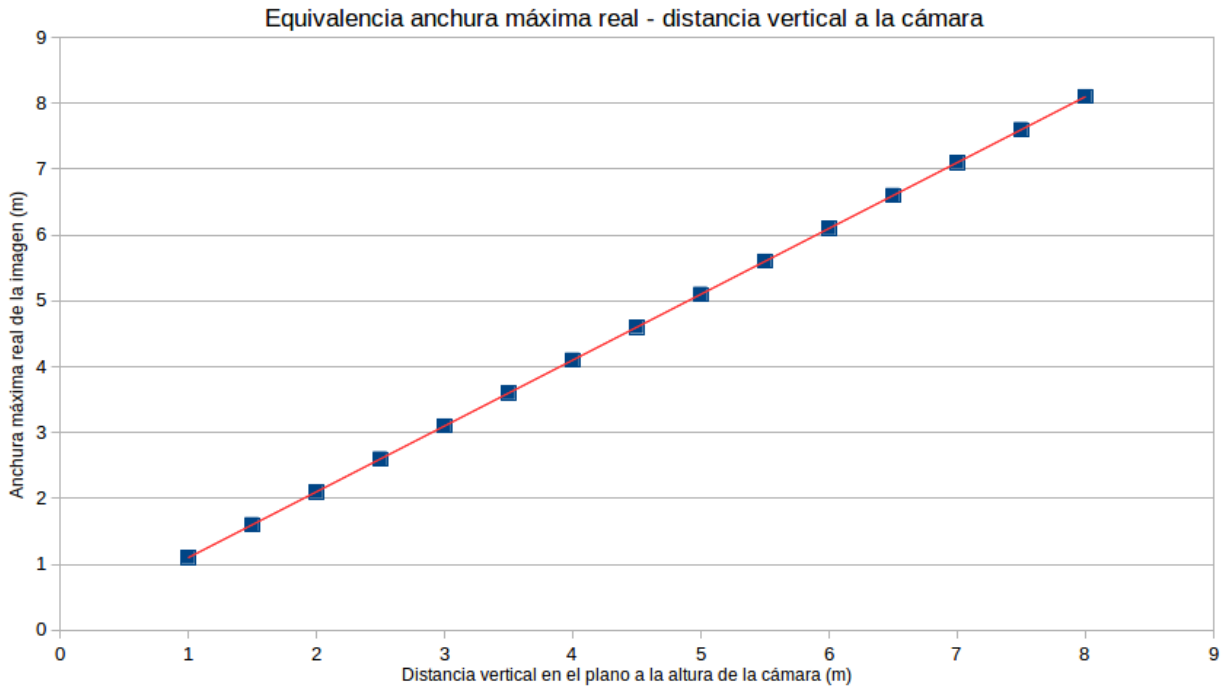

Figura 6-9. Nube de puntos y curva de mejor ajuste para la anchura de la imagen

La ecuación que representa a dicha recta es la (6–2):

$$f(x) = 0.1 + x \tag{6–2}$$

Una vez que se ha explicado el método seguido para establecer la relación entre la matriz de imagen y la imagen real, pasamos a describir brevemente la secuencia de acciones llevadas a cabo en el código fuente image_converter.cpp para calcular la posición. Estas acciones son:

1. Accedemos al vector que contiene la fila y columna de los cuatro vértices del rectángulo calculado al reconocer un agujero. Con el valor de las filas, calculamos la distancia a la cámara (la etiquetada como "Y" en la Figura 6-8) implementando la ecuación (6–1).
2. Con el dato de la distancia real de cada vértice a la cámara calculamos el radio de la circunferencia según la ecuación (6–3) y la distancia real del centro de la misma a la cámara con la ecuación (6–4).

$$radio = \frac{dvmas - dvmenos}{2} \tag{6–3}$$

Siendo *dvmas* la distancia a la cámara del vértice más alejado y *dvmenos* la distancia del vértice menos alejado.

$$dcc = dvmas - radio \tag{6–4}$$

Siendo *dcc* la distancia del centro del agujero a la cámara y *dvmas* y *radio* los explicados anteriormente en la ecuación (6–3).

3. Con el dato de la distancia calculado en el punto 1 y la ecuación (6–2), calculamos la máxima anchura que tendrá la imagen a la altura del centro del rectángulo (la fila del centro del rectángulo se calcula como la media de la fila del vértice más alejado y la del vértice menos alejado).

   Tras eso, y apoyándonos en la Figura 6-10, aplicamos la ecuación (6–5):

$$dvci = (cvr - cci)\frac{am}{Nc} \tag{6–5}$$

   Siendo *dvci* la distancia real del vértice del rectángulo al centro de la imagen, *cvr* la columna del vértice del rectángulo en la matriz de imagen, *cci* la columna del centro de la matriz de imagen, *am* la amplitud máxima calculada con la ecuación (6–2) y *Nc* el número de columnas de la matriz de imagen. Nótese que (*cvr* - *cci*) es lo que aparecen en la Figura 6-10 como "a". Al fin y al cabo no es más que establecer una factor de conversión entre la anchura máxima real de la imagen para cada fila y el número de columnas total de la matriz de imagen para poder calcular longitudes a partir de distancias en píxeles.

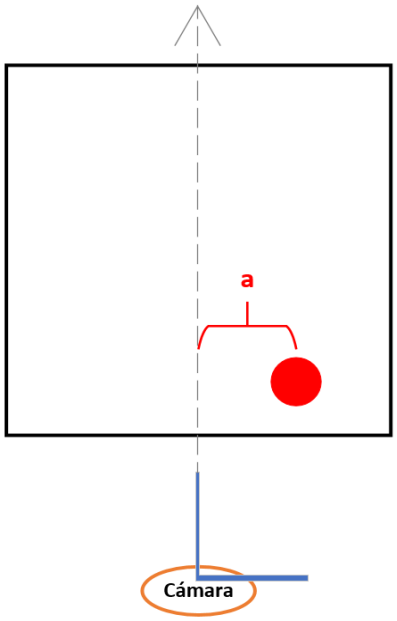

Figura 6-10. Desviación del obstáculo respecto al centro de la imagen

4. Con el dato de la desviación de cada vértice del rectángulo con respecto al centro de la imagen calculamos la posición del centro en ese eje con la ecuación :

$$dcaci = dvmasci - \frac{dvmasci - dvmenosci}{2} \tag{6–6}$$

   Siendo *dcaci* la distancia del centro del agujero al centro de la imagen ("a" en la Figura 6-10), *dvmasci* la distancia del vértice del rectángulo más alejado del centro de la imagen y *dvmenosci* la distancia del vértice del rectángulo menos alejado del centro de la imagen.

5. Como se observa que esta última distancia calculada no es tan exacta como la primera distancia que calculamos (del objeto a la cámara en el eje "Y" de la misma), truncamos el valor de la desviación de los vértices del rectángulo respecto al centro del imagen. Para ello cogemos la distancia *dcaci* calculada anteriormente en la ecuación (6–6) (que nos da la distancia del centro del agujero al centro de la imagen) y le sumamos y restamos el valor del radio calculado en la ecuación (6–3). Obtenemos ahora sí, una medida más creíble de la desviación máxima y mínima del agujero respecto al centro de la imagen. Las ecuaciones empleadas son la (6–7) y la (6–8):

$$drvmasci = dcaci + radio \quad (6–7)$$

$$drvmenosci = dcaci - radio \quad (6–8)$$

Siendo *drvmasci* la distancia real del vértice del rectángulo más alejado al centro de la imagen, *drvmenosci* la distancia real del vértice del rectángulo menos alejado al centro de la imagen, *dcaci* la distancia calculada en la ecuación (6–6) y *radio* el calculado en la ecuación (6–3).

6. Imprimimos por pantalla las distancias calculadas.
7. Todos los pasos anteriores (del 1 al 6) se realizarán siempre que se haya encontrado algún obstáculo y funcionarán correctamente para un solo obstáculo en la misma escena (lo cual será una suposición que tenemos que asumir) y con mayor precisión para distancias a la cámara comprendidas entre los 1.5 y los 4 metros. En caso de no encontrar obstáculo alguno, los 3 datos que definen a la circunferencia se completarán con el número -9999 en señal de que no se ha encontrado nada.

   A pesar de que a distancias mayores de 4 metros comienza a ser algo más inexacta la estimación de la posición, seguimos estimándola e informando de ella, pues es un primer paso para reconocer que por la zona hay un obstáculo. Se supone que en ese caso el robot se acercará más para obtener una información más correcta, incluirla en el mapa y, posteriormente, esquivarla.
8. Por último, el nodo image_converter, cuyo código fuente estamos describiendo, publicará la información que se comentó en el topic de nombre **/Obstacle_data**. Recordamos que la información eran los tres parámetros que aparecen en la Figura 6-3.

Con esto ya hemos completado los dos primeras subtareas de este trabajo y estamos en disposición de desglosar lo realizado con el mapa. Algunos resultados del tratamiento de imágenes y reconocimiento y situación de agujeros se incluyen en el capítulo 7, dedicado a resultados.

6.4 Modificación del mapeado

6.4.1 Método de construcción del mapa

En el Anexo A. Manual de usuario de esta memoria se explica cómo obtener los códigos para realizar el mapeado, cómo ejecutarlos y la configuración necesaria que hay que hacer en RViz para ver correctamente el mapa. Aquí, en cambio, nos centraremos en la filosofía del código fuente que permite construir un mapa de obstáculos a partir de la información del láser.

El mapa que se construye es, al fin y al cabo, una matriz de valores en la que cada elemento corresponde con una celda del escenario de simulación por el que se mueve el robot visto en planta (2 dimensiones) [38] y [39]. En este mapa nos podemos encontrar tres zonas bien diferenciadas según las cuales se le asignará un valor otro a los elementos de la matriz. Estos casos se contemplan en la Tabla 6-3.

Caso	Denominación	Valor del elemento correspondiente de la matriz	Color de visualización de esa celda en el mapa de RViz
La celda pertenece a una zona que el láser aún no ha visto y no tiene información de ella	Zona no visitada	-1	Gris oscuro
La celda pertenece a una zona que el láser ha visto y existe obstáculo	Zona visitada y ocupada	100	Negro
La celda pertenece a una zona que el láser ha visto y no existe obstáculo	Zona visitada y no ocupada	0	Gris claro

Tabla 6-3. Zonas posibles en el mapa de obstáculos

Según esa información, el código del mapeado se dedicará a lo que refleja el diagrama de flujo de la Figura 6-11.

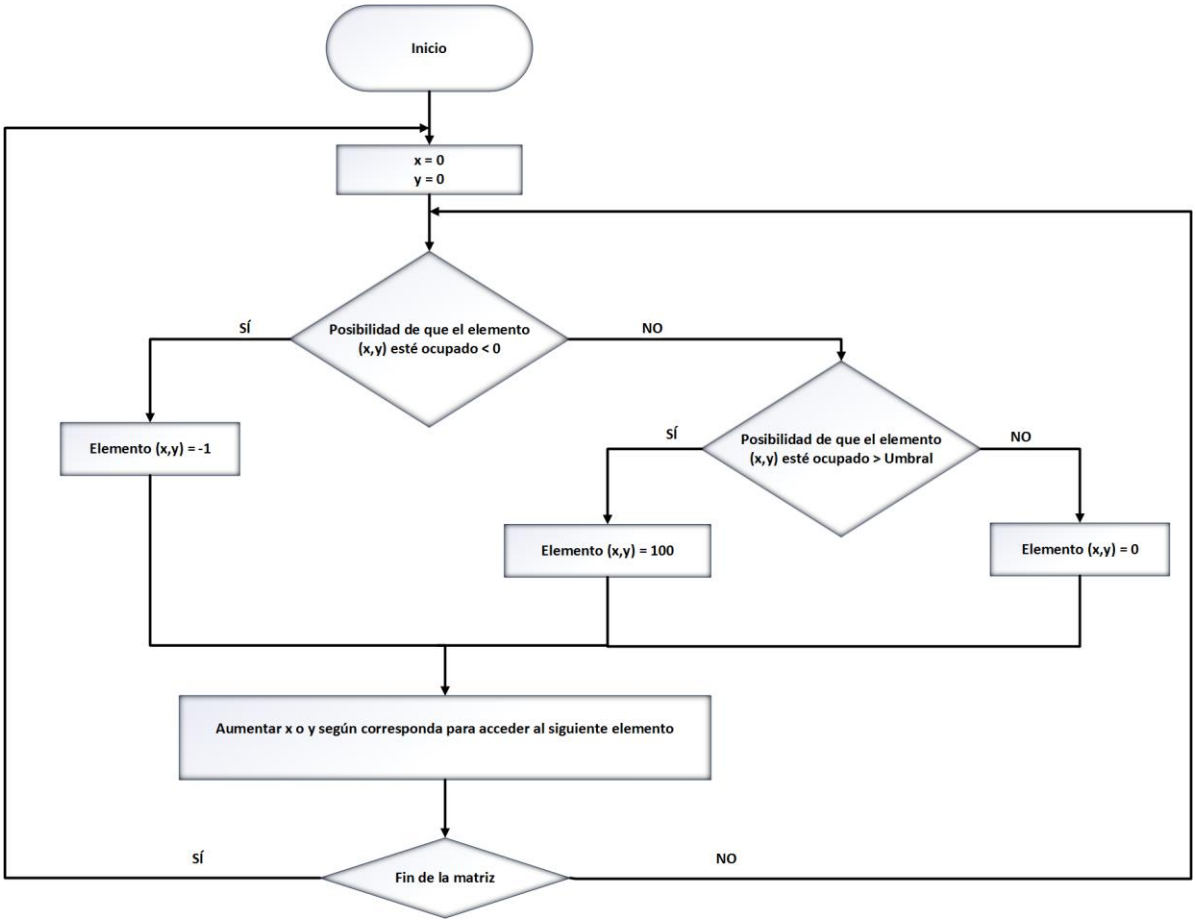

Figura 6-11. Diagrama de flujo del código del mapeado

El diagrama es sencillo, pero la forma de obtener los datos para llevarlo a cabo no es trivial. El código está continuamente tomando información del láser *hokuyo* a través del topic **/hokuyo_base/scan**. Esa información se vuelca en el mapa en función de la posición del robot respecto a la base empleada para realizar el mapa. La base, denominada "map", se sitúa en el centro de la matriz del mapa (ver Figura 6-12, que corresponde a la simulación en gazebo de la Figura 6-13). Por odometría se obtiene la posición del robot respecto a esa base y, en función de la información de distancias que nos ofrezca el sensor láser, situaremos esa información en una celda u otra de la matriz del mapa.

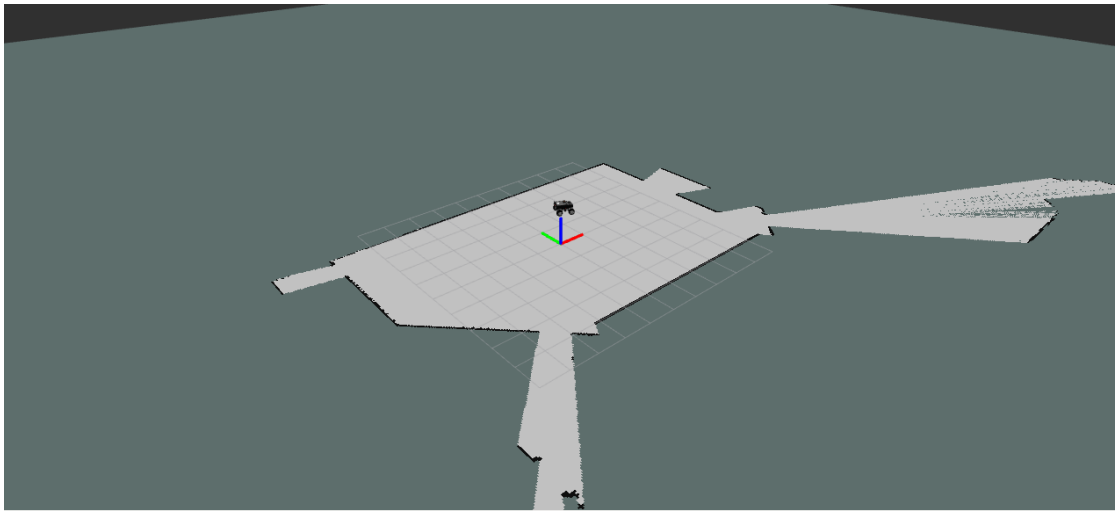

Figura 6-12. Mapa construido en RViz y situación del eje de coordenadas "map"

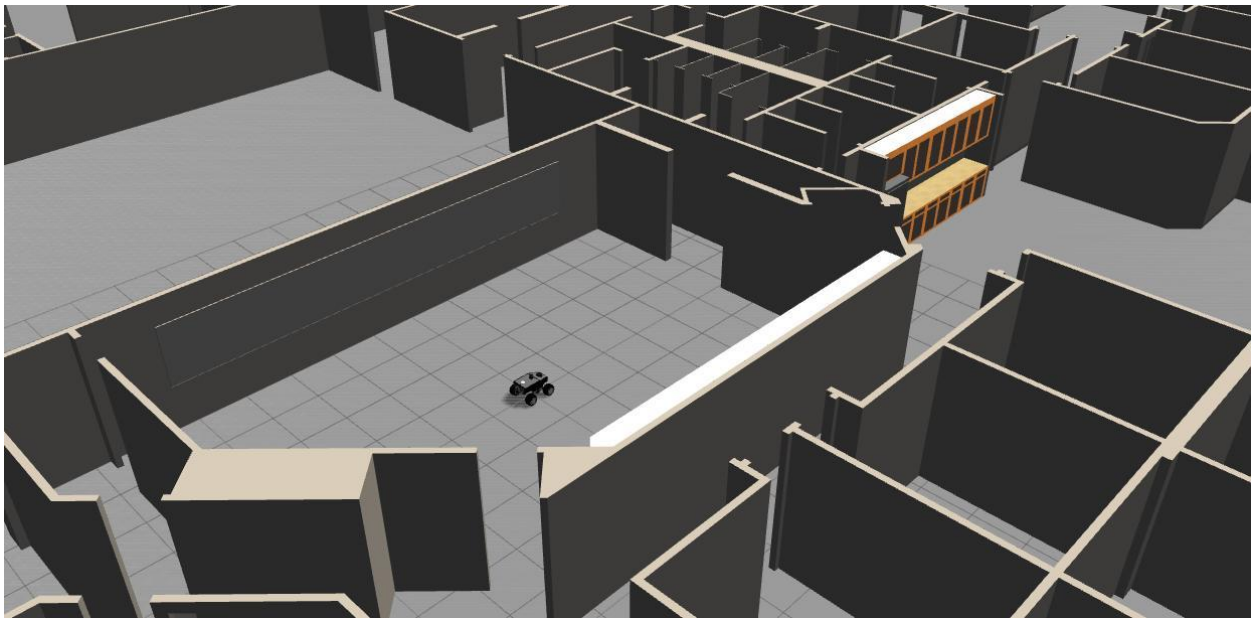

Figura 6-13. Ejemplo de simulación en gazebo

Para ver si una celda tiene posibilidad de estar ocupada o no, llevamos a cabo un recuento con la función **cell**, que nos dará la información para discriminar si un determinado punto está ocupado o no, o incluso si no se ha visitado. Por otra parte, el "**Umbral**" que aparece en el diagrama de flujo de la Figura 6-11 se define al principio del código con el nombre "**occ_thresh_**" y para su configuración nos remitimos al Anexo A. Manual de usuario.

Otra información importante que se define al principio del código es el tamaño del mapa y el número de divisiones o elementos de la matriz. Será un cuadrado de lado 100 metros en el que cada celda tendrá un tamaño de 5 cm. Por tanto, la matriz será cuadrada de dimensiones 2000x2000. Podemos observar el tamaño del mapa en la Figura 6-14.

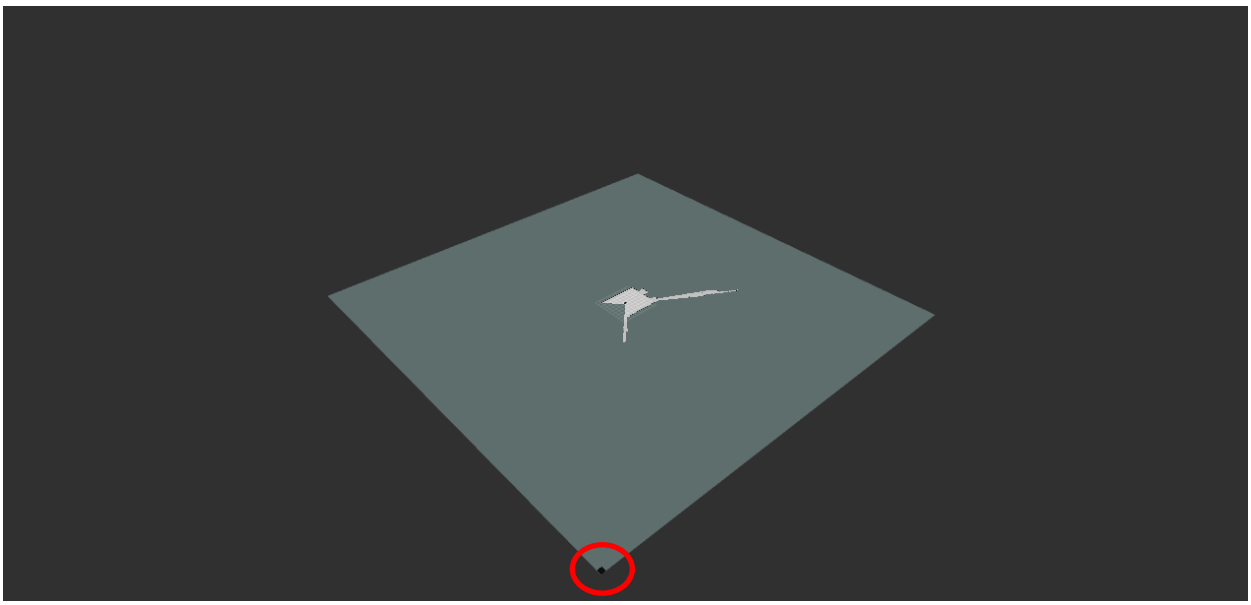

Figura 6-14. Visión general de todo el mapa que se construye

En la imagen se puede apreciar además cómo el robot parte de la zona central (la única por ahora visitada) y el punto inicial de la matriz que representa el mapa, señalado en la imagen de color rojo y que se puso de color negro en la matriz para identificarlo.

Por último, se muestra la Figura 6-15. En ella, se puede apreciar más claramente las tres zonas de las que se hablaba al principio. En gris claro la zona visitada y no ocupada, en negro la zona visitada y ocupada (que se

corresponde en este caso con las paredes de la simulación de la Figura 6-13) y en gris oscuro la zona no visitada y desconocida, que es la mayoría y se debe en este caso bien a que se encuentra a espaldas del robot o bien a que hay objetos de por medio como son las paredes, además de que el tamaño del mapa es grande en comparación con el alcance del láser, que es de 30 metros.

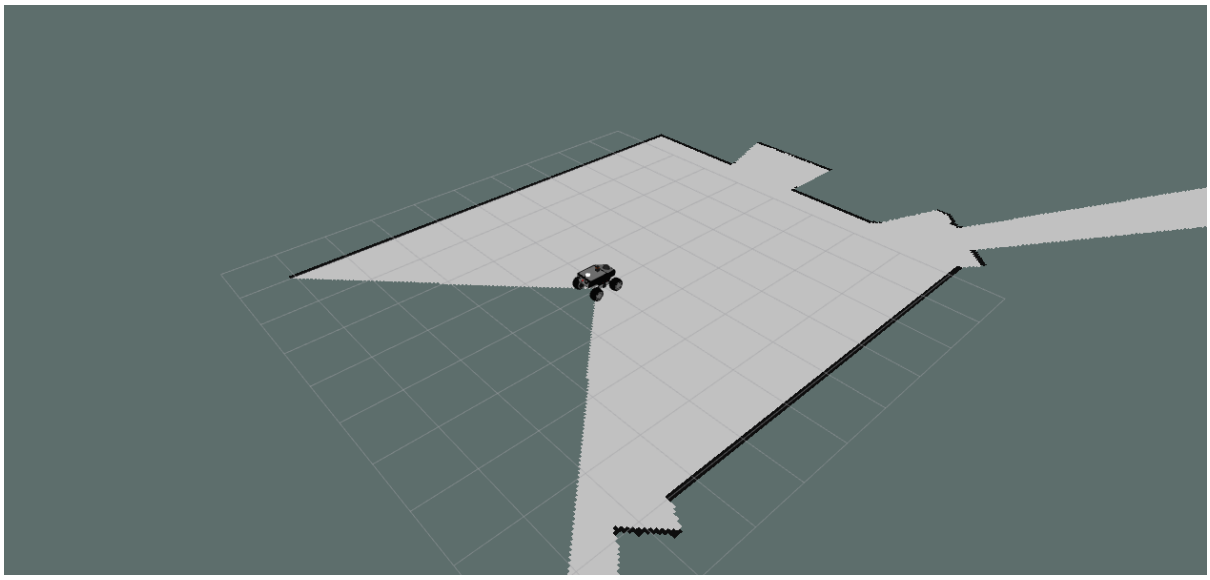

Figura 6-15. Ejemplo de mapeado en RViz

6.4.2 Modificaciones al código del mapeado

Ya se ha explicado en el apartado anterior la filosofía del código descargado para el mapa y cómo funciona la construcción de este. Ahora pasamos a describir lo novedoso para incluir la información de la cámara, todo ello relacionado con el código fuente slam_gmapping.cpp incluido en el Anexo B. Códigos desarrollados.

En primer lugar, tenemos que suscribirnos al topic **/Obstacle_data**, que es donde la cámara deja el mensaje con los datos del obstáculo. Para ello, acudimos a la función "void SlamGMapping::startLiveSlam()" y nos suscribimos a dicho topic con la siguiente línea de código:

```
ros::Subscriber sub = node_.subscribe("Obstacle_data", 1000, clbk);
```

Además, añadimos al final de esa función lo siguiente:

```
ros::spin();
```

Lo que hemos hecho ha sido suscribirnos al topic y, que cada vez que llegue información, llamemos a la función **clbk** (que explicaremos a continuación). La llamada **ros::spin()** mantiene la función clkb activa mientras se el código siga ejecutándose [2].

La función clbk la tenemos que declarar al principio del código (justo detrás de las cabeceras) con la línea:

```
void clbk(const geometry_msgs::Point32 msg);
```

Lo que hace la función (que se puede ver en el Anexo B. Códigos desarrollados) será recibir los datos de posición y tamaño del obstáculo. Con esa información y la de la posición y orientación del robot respecto a la base del mapa, calculamos la posición del centro del agujero respecto a la base del mapa.

Si la información es similar a la de otros obstáculos ya vistos y guardados, se actualiza, reemplazándola. Si no lo es, se almacena como un obstáculo nuevo. Por tanto, estamos almacenando la información en vectores donde cada componente debe ser un obstáculo distinto.

Este vector, al ser global, es accesible desde otras funciones, como la que dibuja el mapa ("void SlamGMapping::updateMap(const sensor_msgs::LaserScan& scan)"). En esa función seguimos básicamente el diagrama de flujo de la Figura 6-16, que es parecido al de la Figura 6-11. Sin embargo, antes de ver si la posibilidad del que el elemento (x, y) de la matriz esté ocupado es negativa, nos preguntamos si dicho elemento coincide con el centro de alguno de los obstáculos ya almacenados. En caso de que coincida,

dibujamos el obstáculo con la información de la posición respecto a la base del mapa y el tamaño del agujero. En caso de no coincidir, es cuando nos preguntamos si la posibilidad de que el elemento (x, y) esté ocupado es menor que cero y procedemos como anteriormente.

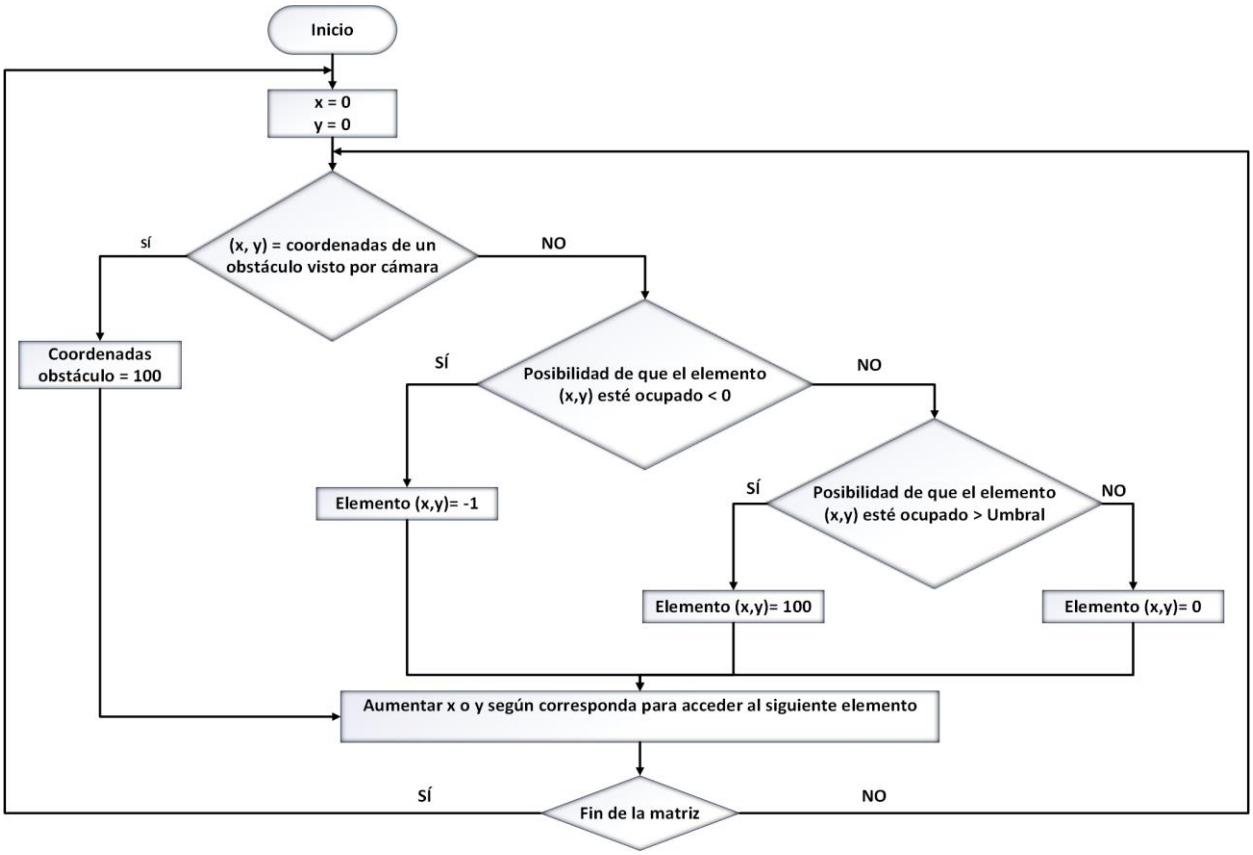

Figura 6-16. Diagrama de flujo final del código del mapeado

En esta explicación general de la modificación al código, hay dos aspectos que deben ser tratados aparte. En primer lugar, la información que obtenemos de la cámara y almacenamos como obstáculos en la función clbk viene en unidades de longitud, concretamente en metros. Sin embargo, al recorrer la matriz del mapa estamos ante celdas o elementos, por lo que tendremos que realizar los cálculos de las ecuaciones (6–9), (6–10) y (6–11) a los datos que definen el obstáculo:

$$fila_{obstaculo} = \frac{obstaculo_x}{delta} \tag{6–9}$$

$$columna_{obstaculo} = \frac{obstaculo_y}{delta} \tag{6–10}$$

$$radio_{obstaculo} = \frac{obstaculo_z}{delta} \tag{6–11}$$

Donde $fila_{obstáculo}$ y $columna_{obstáculo}$ son la fila y columna que ocupa el centro del obstáculo en la matriz del mapa respectivamente, medidas ambas desde el centro del mismo. Las variables $obstaculo_x$, $obstaculo_y$ y $obstaculo_z$ son los datos del agujero medidos respecto al eje de la cámara en unidades de longitud: distancia a la cámara, desviación respecto al centro de la imagen y radio del agujero. Por su parte, *delta* nos da la relación entre longitudes y elementos de la matriz del mapa. Su valor está fijado en 0.05 m/elemento.

El otro aspecto importante es la obtención de las coordenadas del centro del obstáculo respecto a la base del mapa y no como nos la ofrece la cámara respecto a su base. Para ello tenemos que recordar cómo representar la posición y orientación de puntos en el plano cuando tenemos un sistema de referencia fijo (el del mapa) y otro que se traslada y rota (el de la cámara, que se mueve según el robot) [13]. Nos apoyamos en la Figura 6-17.

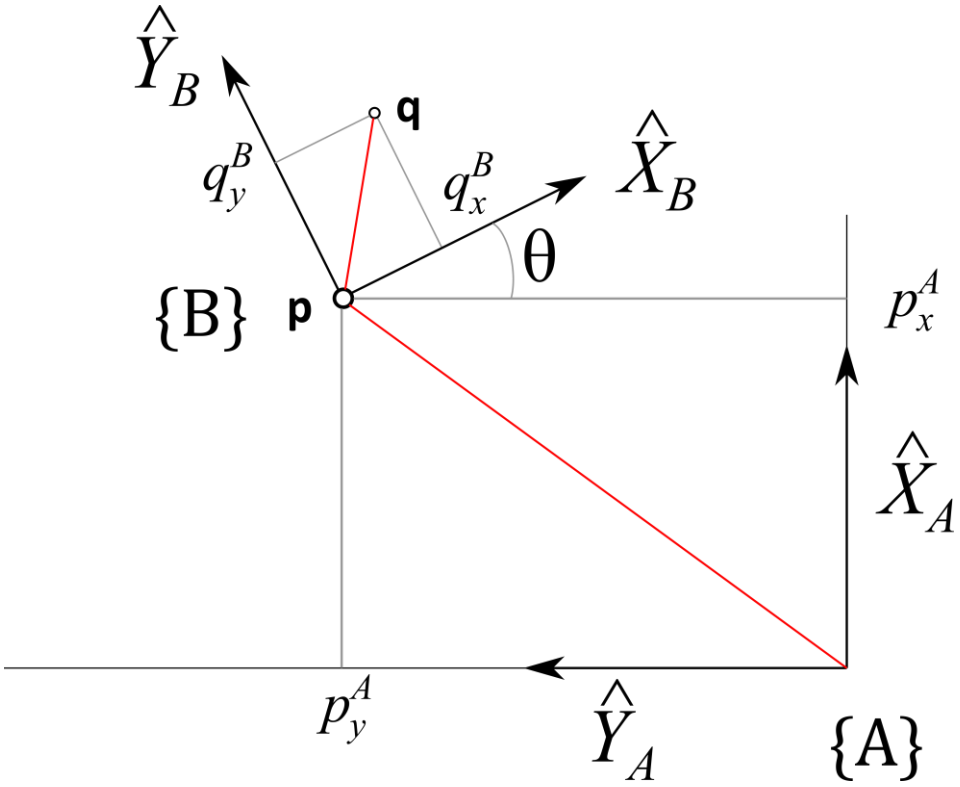

Figura 6-17. Sistemas de referencia del mapa y la cámara

En él observamos el sistema de referencia fijo {A}, que corresponde al mapa, y el sistema de referencia móvil {B}, que corresponde a la cámara. Se observa que para situar un punto del sistema {B} en el {A} hay que realizar una traslación del sistema {B} sobre el {A} y, a su vez, una rotación del sistema {B} un ángulo θ.

El punto *p* representa el origen del sistema de coordenadas {B} y el punto *q* el centro del obstáculo encontrado. La posición del punto *p* respecto al sistema de referencia {A} es la de la ecuación (6–12):

$$\boldsymbol{p^A} = \begin{bmatrix} p_y^A \\ p_x^A \end{bmatrix} \tag{6–12}$$

Por su parte, la posición del punto *q* en el sistema de referencia {B} es la de la ecuación (6–13):

$$\boldsymbol{q^B} = \begin{bmatrix} q_x^B \\ q_y^B \end{bmatrix} \tag{6–13}$$

Y la matriz de rotación para alinear el sistema {B} con el sistema {A} es la de la ecuación (6–14):

$$\boldsymbol{{}_B^A R} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \tag{6–14}$$

Por tanto, para calcular la posición del punto *q* respecto al sistema {A}, realizamos la composición de traslación más rotación según la ecuación (6–15):

$$\boldsymbol{q^A} = \begin{bmatrix} q_y^A \\ q_x^A \end{bmatrix} = \boldsymbol{p^A} + \boldsymbol{{}_B^A R q^B} = \begin{bmatrix} p_y^A \\ p_x^A \end{bmatrix} + \begin{bmatrix} -(cos\theta) & -(-sin\theta) \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} q_x^B \\ q_y^B \end{bmatrix} \tag{6–15}$$

Hay dos detalles importantes. El primero es que las coordenadas x e y en el punto *q* aparecen al contrario que en el punto *p*. Se debe a que los ejes x e y del sistema {B} están al contrario que los del sistema {A}. Y el segundo es la aparición del signo negativo en los términos de la primera fila de la matriz de rotación. La causa de esto es que el eje horizontal de la cámara (X del sistema {B}) tiene el sentido positivo hacia el lado negativo del eje horizontal del mapa (eje Y del sistema {A}).

7 RESULTADOS

Este capítulo tiene el objetivo de mostrar que los métodos que se han ido describiendo a lo largo de la memoria funcionan y hacen posible que se cumplan los objetivos marcados. Lo dividiremos en dos: los resultados de la parte de visión y los resultados finales de la construcción del mapa.

7.1 Resultados de vision

Ya se comentó que la cámara es capaz de ver obstáculos que se encuentren en una distancia comprendida entre los 0.7 y los 8 metros debido a limitaciones de la misma y a la orientación que toma. Además, reconocerá fundamentalmente obstáculos en el suelo, debido a que apunta hacia el mismo. Y la anchura que se puede captar en una imagen variará linealmente desde una anchura de 1 metro aproximadamente (a la distancia más cercana que ve la cámara) hasta unos 8 metros (a la máxima distancia que ve la cámara).

En ese rango se hacen pruebas con agujeros de distinto tamaño y situados en posiciones más alejadas o más cercanas, desviados del eje de la cámara o centrados. Se usan para ello agujeros de radio 15 cm, 30 cm, 50 cm y 80 cm. Los resultados de las simulaciones los podemos ver en las figuras siguientes. En ellas podemos observar cuatro ventanas: la simulación en gazebo de fondo, abajo a la izquierda la terminal que nos da la posición (tanto en la matriz como en la realidad respecto a la cámara) de los vértices del rectángulo que engloban al obstáculo, arriba a la derecha la ventana con la cámara donde se dibujará la elipse que mejor se adapte al obstáculo y abajo a la derecha, casi escondido, la terminal de teleoperación del robot.

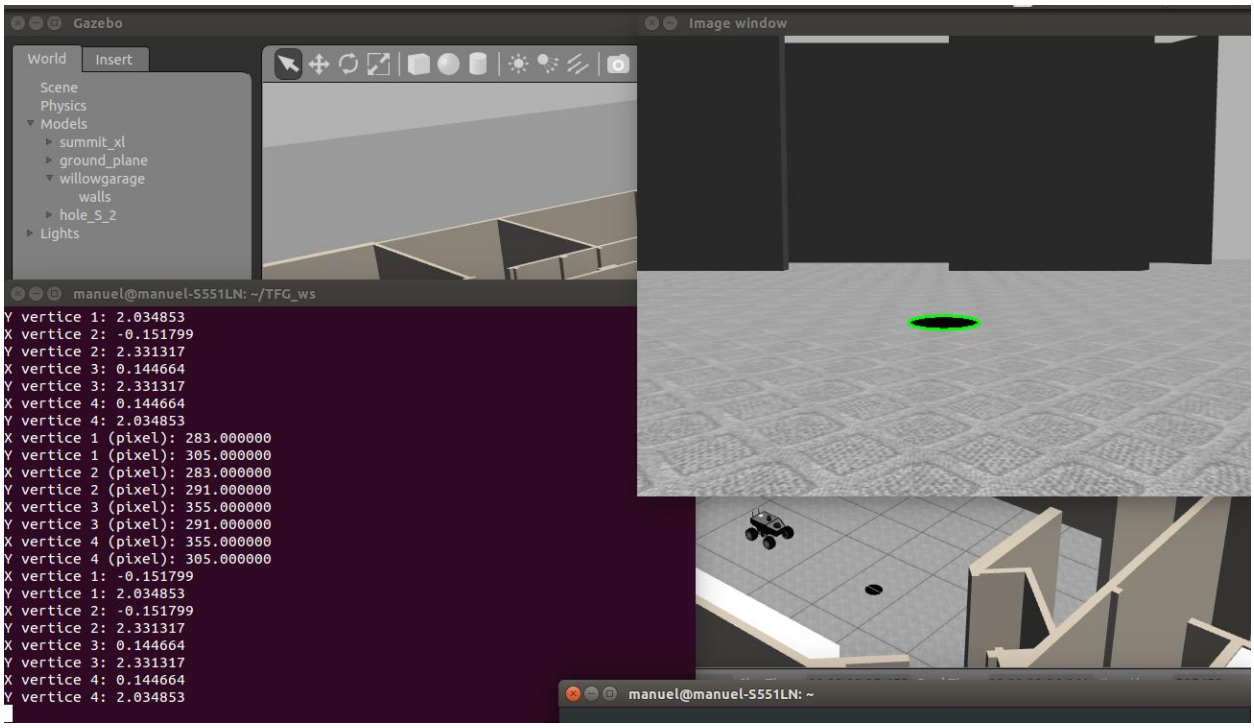

Figura 7-1. Simulación 1: agujero de radio 15 cm, centrado y a unos 2.2 m de distancia

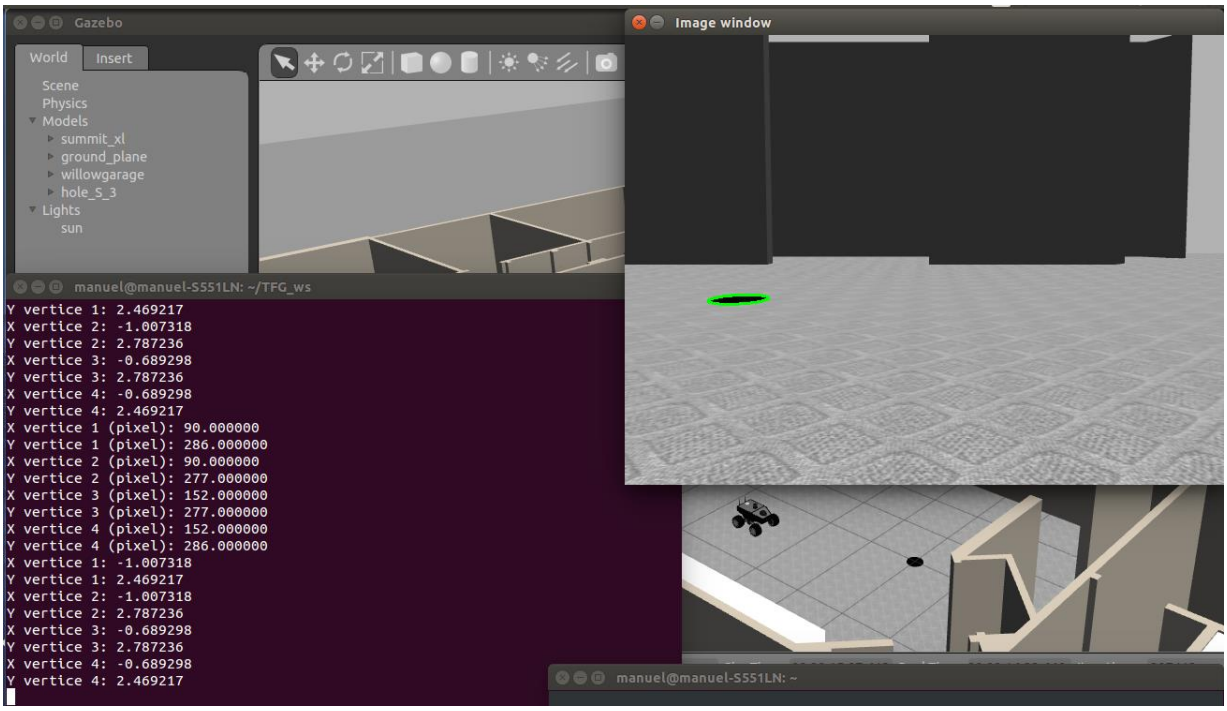

Figura 7-2. Simulación 2: agujero de radio 15 cm, no centrado y a unos 2.7 m de distancia

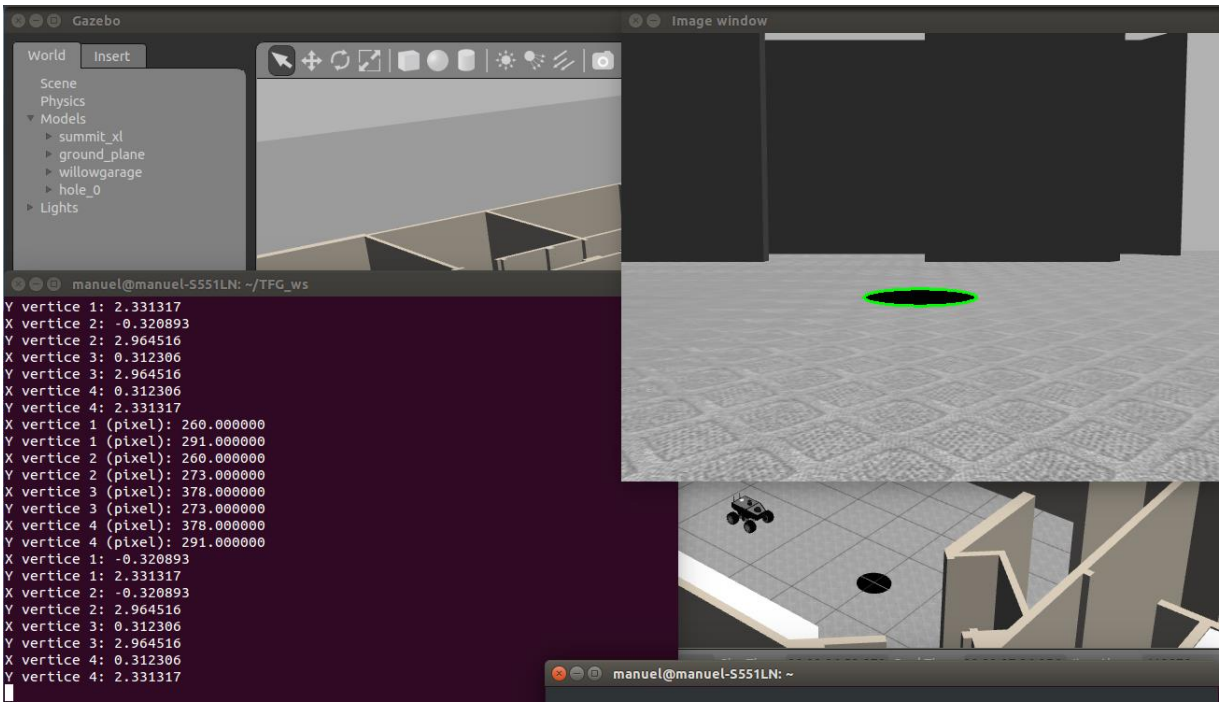

Figura 7-3. Simulación 3: agujero de radio 30 cm, centrado y a unos 2.7 m de distancia

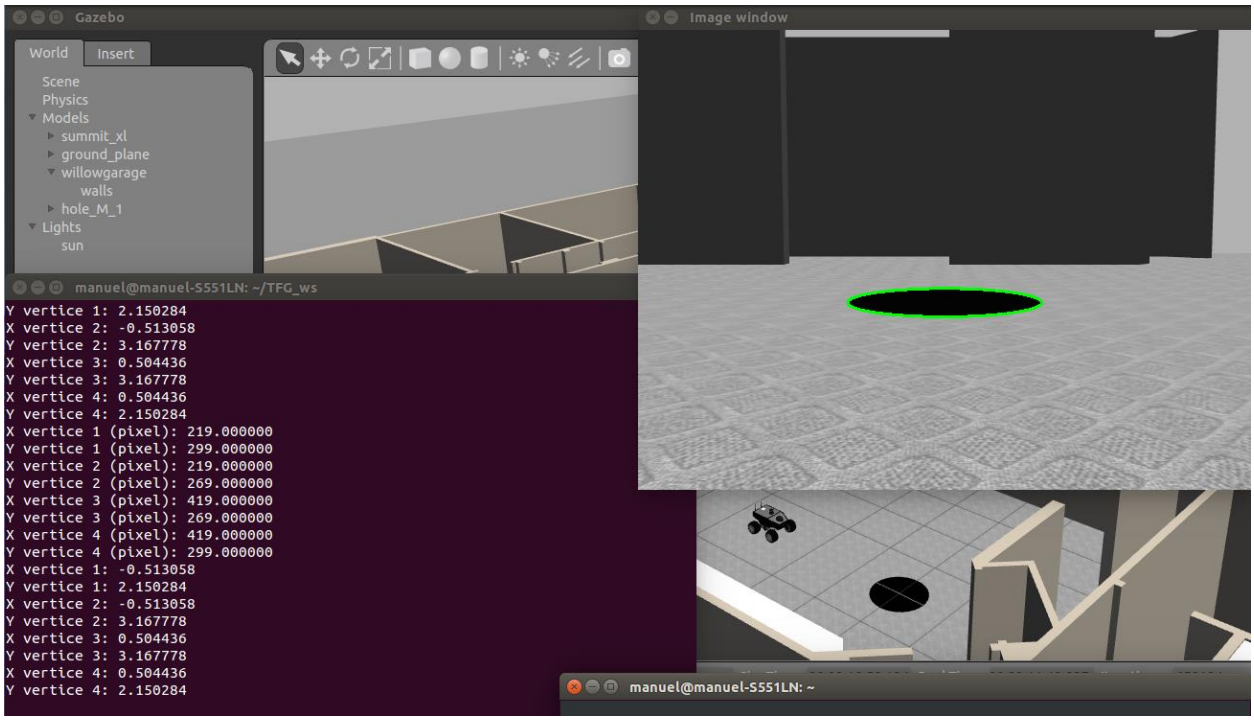

Figura 7-4. Simulación 4: agujero de radio 50 cm, centrado y a unos 2.7 m de distancia

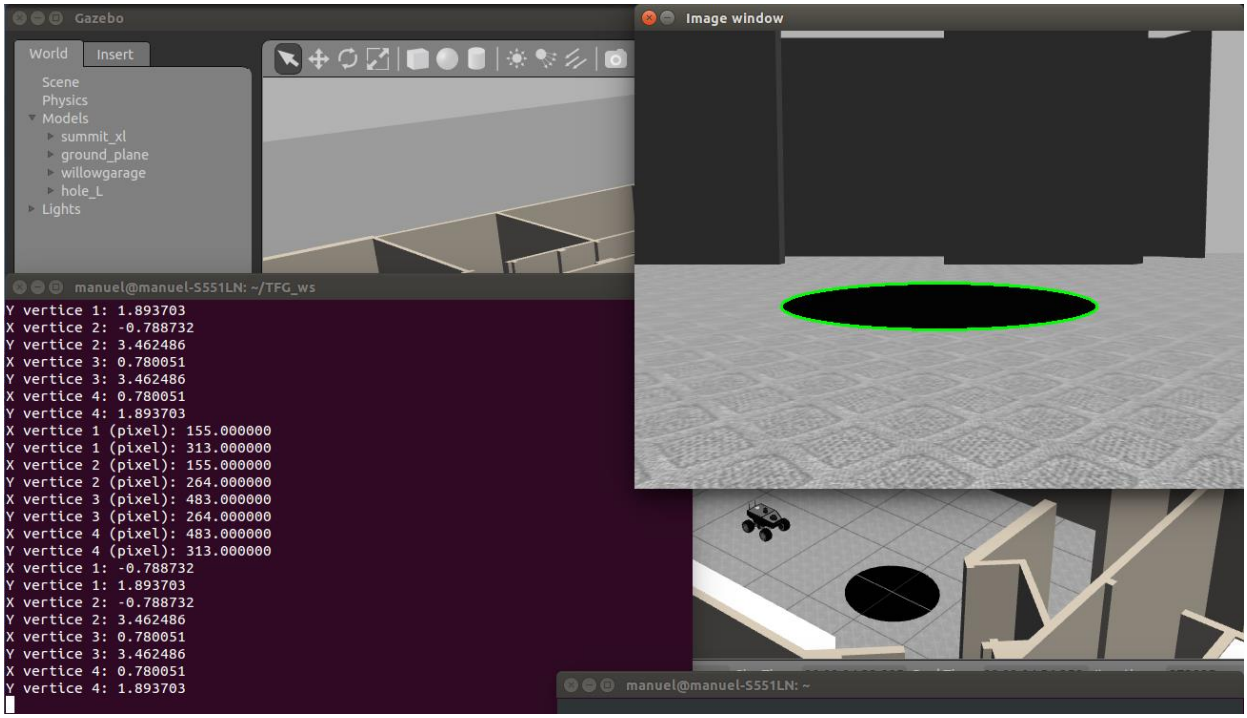

Figura 7-5. Simulación 5: agujero de radio 80 cm, centrado y a unos 2.7 m de distancia

Realmente el único agujero que se ha puesto desviado del centro de la imagen ha sido el de tamaño más pequeño. Esto simplemente se debe a que era el que cabía mejor en la imagen y que fuera la cámara capaz de cogerlo entero, pero no quiere decir que sea el único que se puede reconocer estando desviado.

Para descifrar esos resultados, es necesario aclarar cuál es cada vértice del rectángulo y cuáles son los ejes de la cámara respecto a la que se miden las distancias. Para eso tenemos la Figura 7-6.

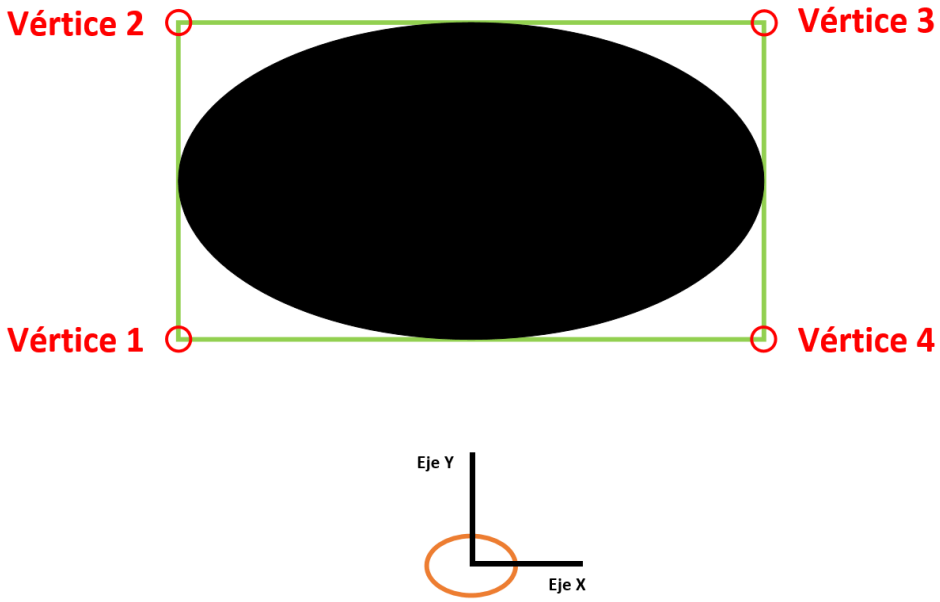

Figura 7-6. Ejes de la cámara y vértices del rectángulo

De esta manera, en la Figura 7-6, los vértices 1 y 2 tienen coordenada X negativa y los vértices 3 y 4, positiva. Todos los vértices de la imagen tienen coordenada Y positiva.

En todos los casos expuestos podemos calcular el radio, por ejemplo usando la ecuación (7–1):

$$radio_{agujero} = \frac{x_{vertice3} - x_{vertice1}}{2} \tag{7–1}$$

Y la distancia del centro del agujero a la cámara con la ecuación (7–2):

$$distancia_{camara} = y_{vertice1} + radio_{agujero} \tag{7–2}$$

La distancia en el eje X de la cámara del centro del agujero al centro de la imagen se calcula con la ecuación (7–3):

$$distancia_{centro} = \frac{x_{vertice4} + x_{vertice1}}{2} \tag{7–3}$$

Nótese que existen otras combinaciones posibles para calcular esos datos y que se pueden usar como comprobación. Los resultados de las cinco simulaciones mostradas anteriormente pueden resumirse en la Tabla 7-1:

La ecuación para calcular el error relativo es la (7–4), el cual no se calcula para la posición del centro del agujero en el eje X de la cámara por tener un cero en el denominador, dando lugar a una indeterminación.

$$Error_{relativo}(\%) = \frac{|Valor_{real} - Valor_{medido}|}{Valor_{real}} x100 \tag{7–4}$$

Simulación	Radio del agujero			Coordenada X del centro del agujero en el eje de la cámara		Coordenada Y del centro del agujero en el eje de la cámara		
	Real (cm)	Medido (ecuación (7–1)) (cm)	Error (ecuación (7–4)) (cm)	Real (cm)	Medido (ecuación (7–3)) (cm)	Real (m)	Medido (ecuación (7–2)) (m)	Error (ecuación (7–4)) (%)
1	15	14.82	1.2%	0	-0.36	2.2	2.18	0.91%
2	15	15.90	6%	-100	-84.83	2.7	2.63	2.59%
3	30	31.66	5.53%	0	-0.43	2.7	2.65	1.85%
4	50	50.87	1.74%	0	-0.43	2.7	2.66	1.48%
5	80	78.44	1.95%	0	-0.43	2.7	2.68	0.74%

Tabla 7-1. Resultados de visión

Hay que destacar que el robot es capaz de reconocer más de un agujero en la misma imagen, tal y como se muestra en la Figura 7-7. Sin embargo, por simplicidad en el código, los resultados de estimación sólo son válidos cuando hay un objeto en la misma escena (o ninguno). Además, se busca simplificar ya que por regla general los agujeros los encontraremos dispersos en entornos simples, tal y como marcaban los objetivos del trabajo.

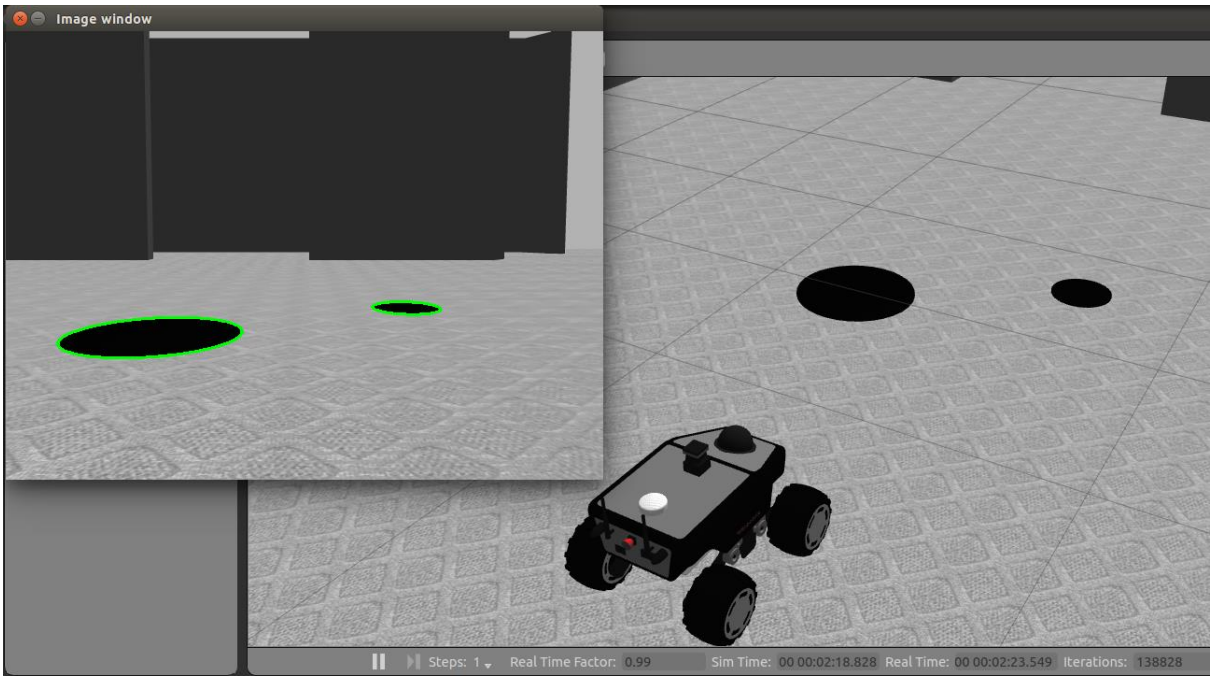

Figura 7-7. Reconocimiento de dos agujeros en la misma escena

Por último, se pueden reconocer con esta aplicación otro tipo de obstáculos que no sean agujeros, aunque el objetivo era para agujeros. Para ello no deben ser muy altos y tienen que tener un color negro o similar. Por ejemplo, se reconoce la cámara *kinect* en la Figura 7-8.

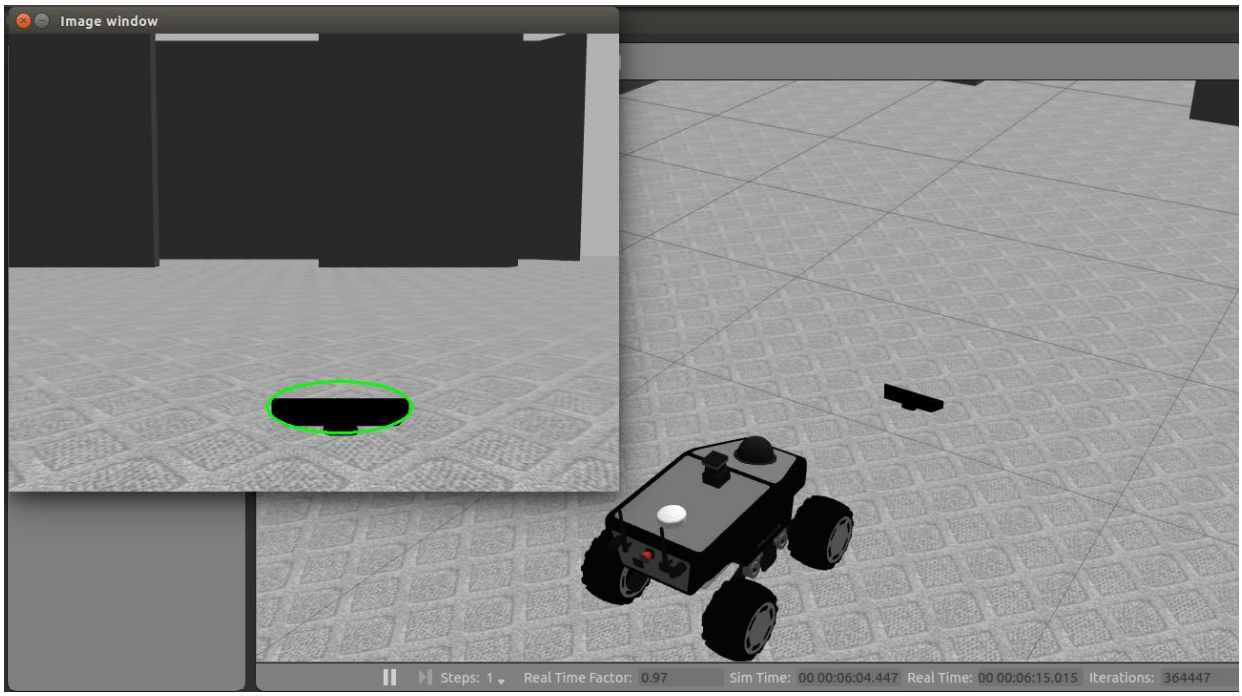

Figura 7-8. Reconocimiento de obstáculo distinto a agujero

7.2 Resultados de la construcción del mapa

Se muestra a continuación el mapa resultante de varias simulaciones. En ellas se prueban agujeros de diferentes tamaños y situados en diferentes puntos del entorno, para comprobar que el cálculo de la posición es correcto en cada uno de los cuatro cuadrantes en los que el origen de la base "map" divide la matriz del mapa.

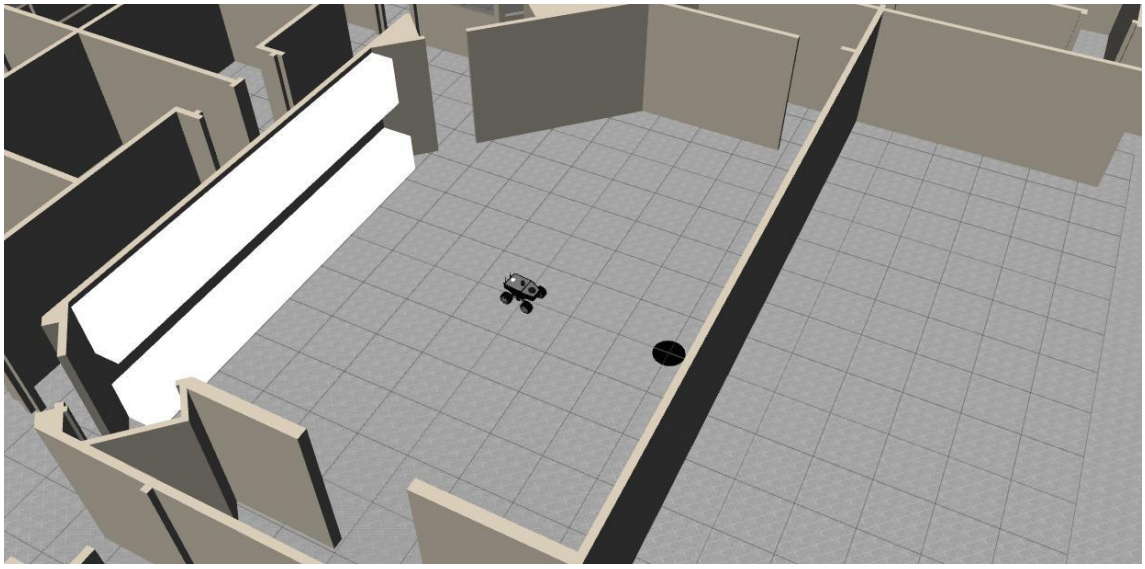

Figura 7-9. Simulación 1 para mapeado en gazebo

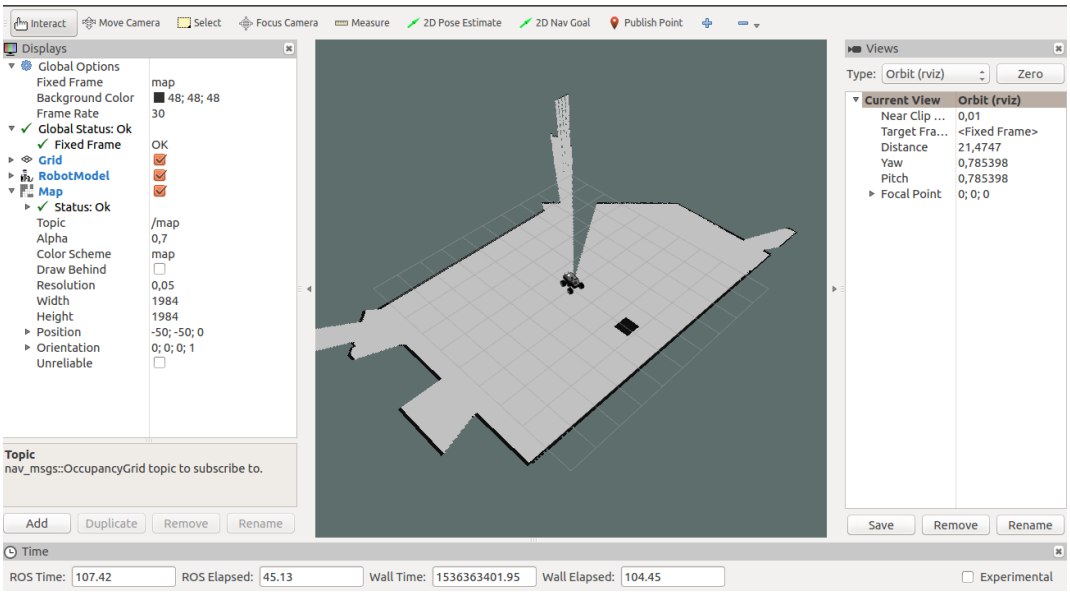

Figura 7-10. Mapa obtenido en RViz para la simulación 1

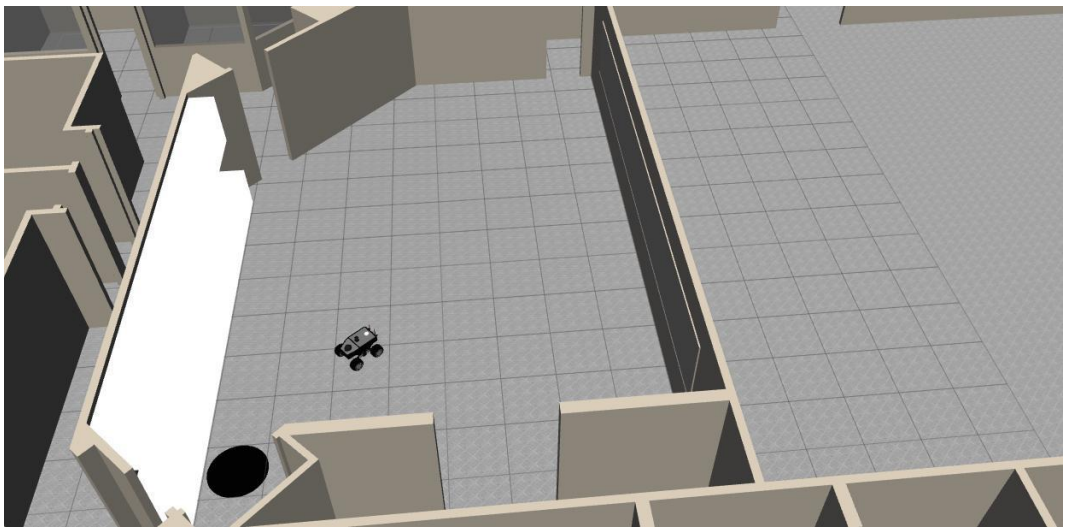

Figura 7-11. Simulación 2 para mapeado en gazebo

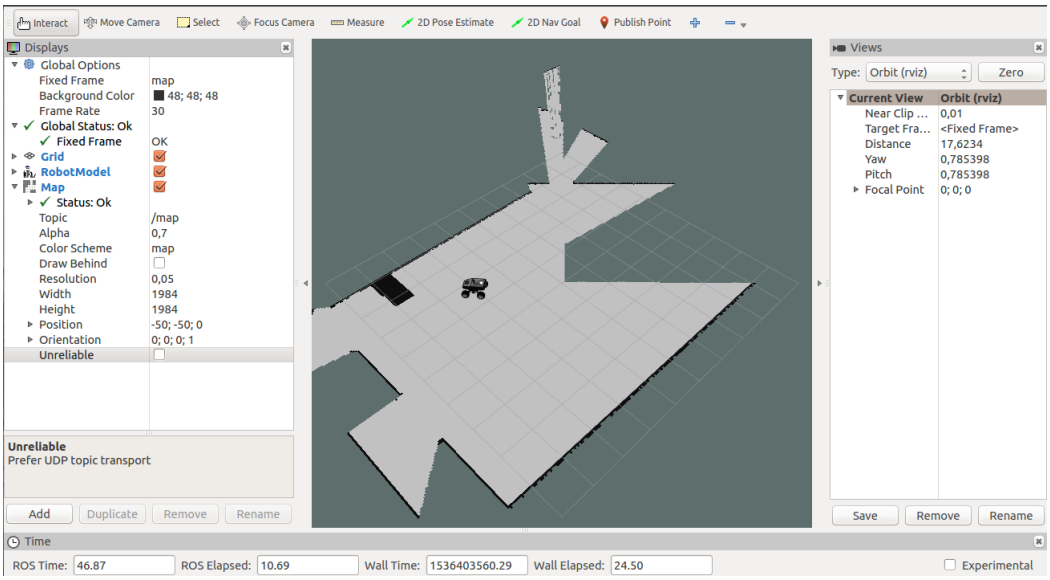

Figura 7-12. Mapa obtenido en RViz para la simulación 2

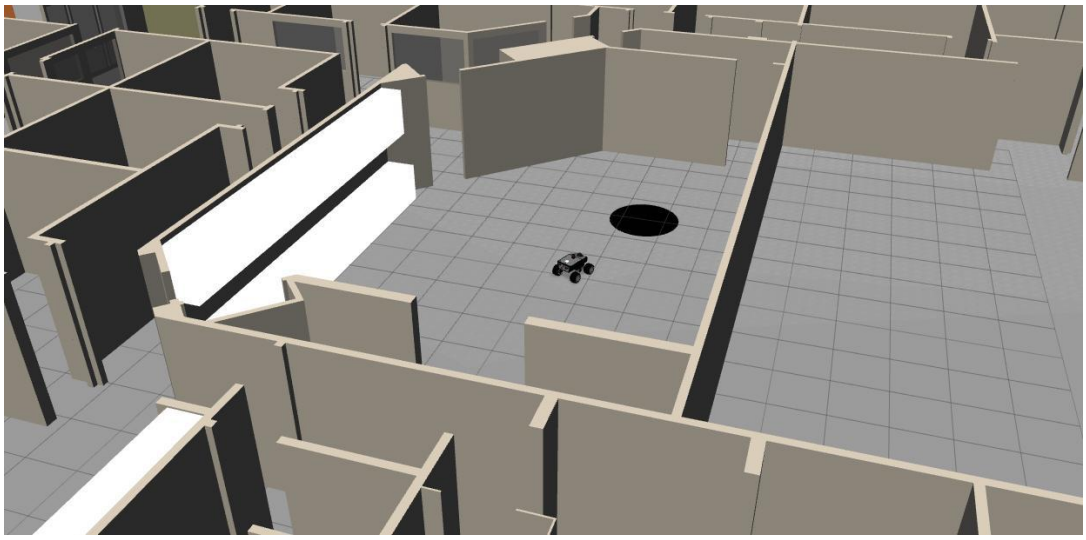

Figura 7-13. Simulación 3 para mapeado en gazebo

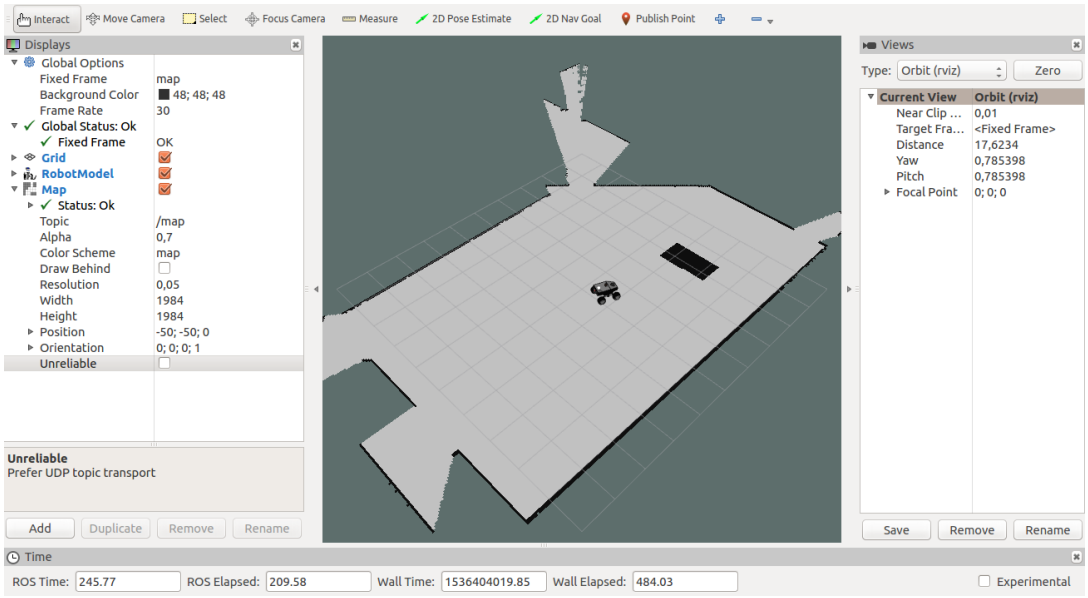

Figura 7-14. Mapa obtenido en RViz para la simulación 3

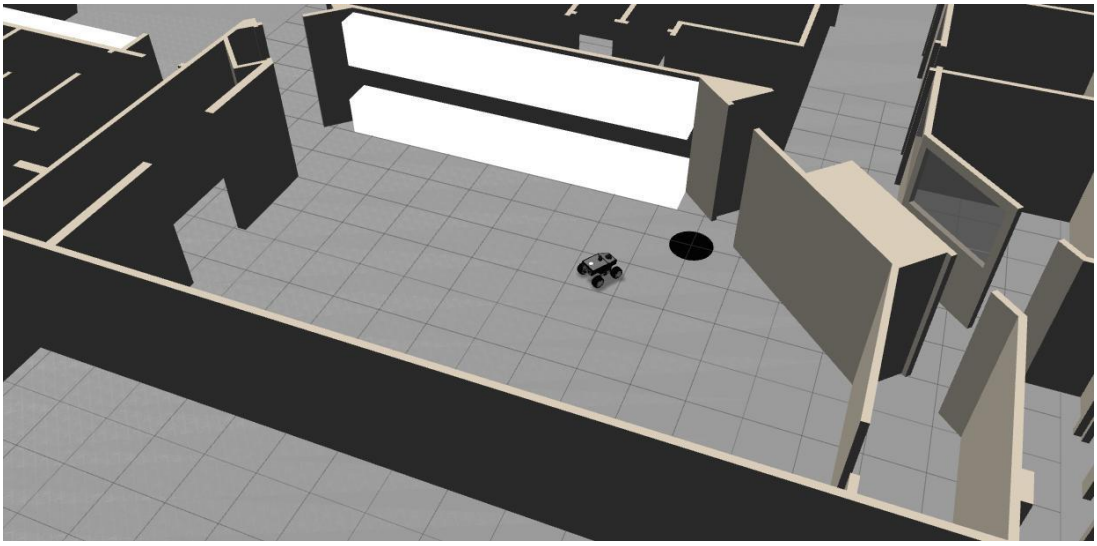

Figura 7-15. Simulación 4 para mapeado en gazebo

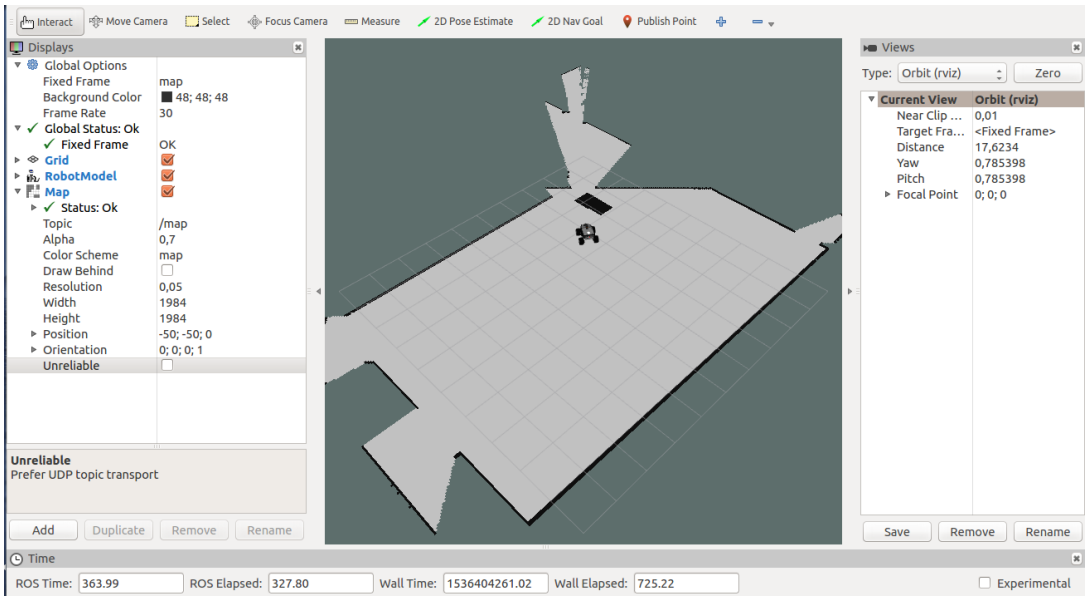

Figura 7-16. Mapa obtenido en RViz para la simulación 4

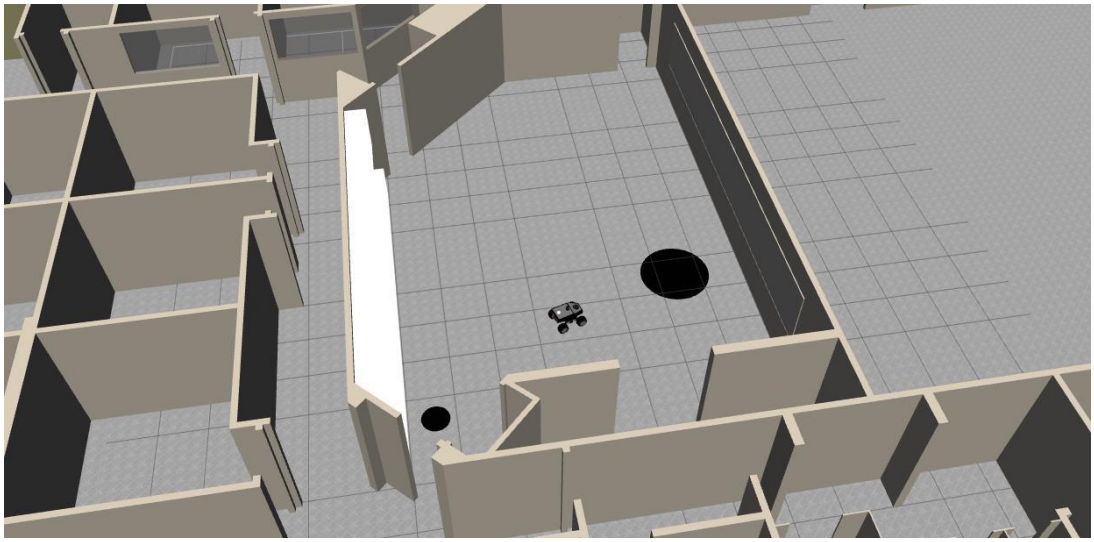

Figura 7-17. Simulación 5 para mapeado en gazebo

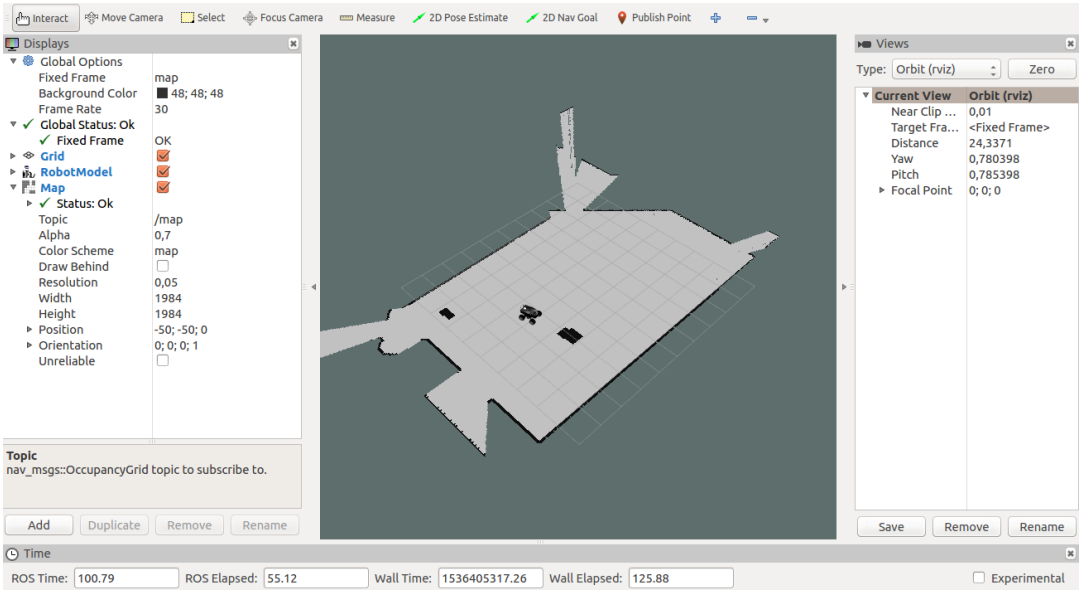

Figura 7-18. Mapa obtenido en RViz para la simulación

La última simulación incluye incluso más de un obstáculo en todo el entorno para mostrar que, aunque no se esté viendo alguno de ellos, la información queda retenida para la veracidad del mapa. Hay que señalar también que las simulaciones en las que no aparece el mapa de la habitación completo es porque el Summit-XL aún no ha apuntado hacia esta zona, por lo que no tiene la información necesaria para construir el mapa.

8 CONCLUSIONES

Tras el estudio de los resultados del capítulo anterior, nos dedicamos, por último, a valorar la realización del trabajo y, además, comentar las limitaciones a las que estamos sujetos, así como posibles mejoras para un futuro.

8.1 Conclusiones

En primer lugar, hay que ser consciente de la gran comunidad que es ROS. Este metasistema operativo permite aplicaciones de alto nivel en el campo de la robótica. Además, su cualidad de colaboración entre usuarios lo hace crecer a gran escala, pues es un trabajo de cooperación mutua. Aprender a programar en ROS es, por tanto, el objetivo personal más valioso de este trabajo.

En cuanto a los objetivos físicos o materiales, no se puede negar que el uso de la cámara mejora la percepción del entorno que puede captar el robot. Si en trabajos anteriores el Summit-XL ya fue capaz de navegar de manera autónoma con la información que la proporcionaba únicamente el láser, ahora aumentamos la fiabilidad, pues somos capaces de reconocer más objetos.

A pesar de que al campo de la visión en la robótica le queda aún por recorrer, observamos la potencia que tiene una librería como es OpenCV. Se pueden realizar tratamientos de imagen muy sofisticados con tan sólo llamar a unas simples funciones. Al ser una librería de funciones que permite trabajar en tiempo real, la hace más versátil.

Además, la orientación que le damos a la cámara es la correcta para esta aplicación si tenemos en cuenta que lo que no se podía divisar antes era el suelo y lo que esperamos encontrar en estas situaciones son los agujeros situados en el mismo. Para obstáculos de mayor altura, aunque se podría usar la cámara, ya tenemos el láser.

Sin embargo, la cámara podría ser aprovechada de otra manera: rotando. Conocemos que la cámara tiene un ángulo de barrido de 360º y de inclinación de 180º. Se podría, por ejemplo, hacer que la cámara, además de apuntar al suelo que tiene el robot de frente, girase 180º y observará lo que el robot tiene por detrás. De esta manera se construiría antes el mapa, pues con el láser no podemos acceder a la parte trasera del robot a no ser que nos movamos y giremos el robot por completo.

Por otra parte, hay que recordar que trabajamos con un robot que se emplea en tareas de vigilancia y se espera que trabaje de manera autónoma. Pero para ello necesita que los pasos previos a la navegación autónoma estén bien depurados. Mención especial para la construcción del mapa, sin lo cual la navegación autónoma no sería posible (a menos que conozcamos el entorno y el robot disponga de un mapa del mismo, situación poco flexible para entornos que puedan ser cambiantes). La construcción de un mapa de obstáculos completo evitará futuros daños en un robot que funcione de manera autónoma. Si antes de incluir la cámara el robot se hubiera encontrado con un agujero, probablemente habría metido la rueda o se habría metido entero en el agujero, lo que supondría un bloqueo y un posible daño al mismo.

Por último, en el mapa construido se observa que los obstáculos que añadimos no tienen la forma circular de los agujeros e incluso pueden variar un poco en tamaño. Esto principalmente se debe a errores de aproximaciones en los cálculos y, en gran parte, a la discretización del entorno, dividido en celdas para la construcción del mapa. Sin embargo no es un tema que nos preocupe demasiado, pues en líneas generales se obtiene una representación del agujero cercana a la realidad. Lo interesante es tener una idea de donde se sitúa el obstáculo y su tamaño para que, en futuros algoritmos de planificación de trayectorias se evite esa zona, lo cual siempre que se pueda se hará con un margen de seguridad para salvar los posibles errores comentados.

Respecto a estos errores de estimación de posición y distancia podemos poner un ejemplo de una de las simulaciones realizadas en este trabajo. Se corresponde con la última simulación de construcción del mapa

realizada, en la cual había dos obstáculos. Tras observar uno de los obstáculos desde varios puntos de vista, nos encontramos que el mapa construido queda como el de la Figura 8-1.

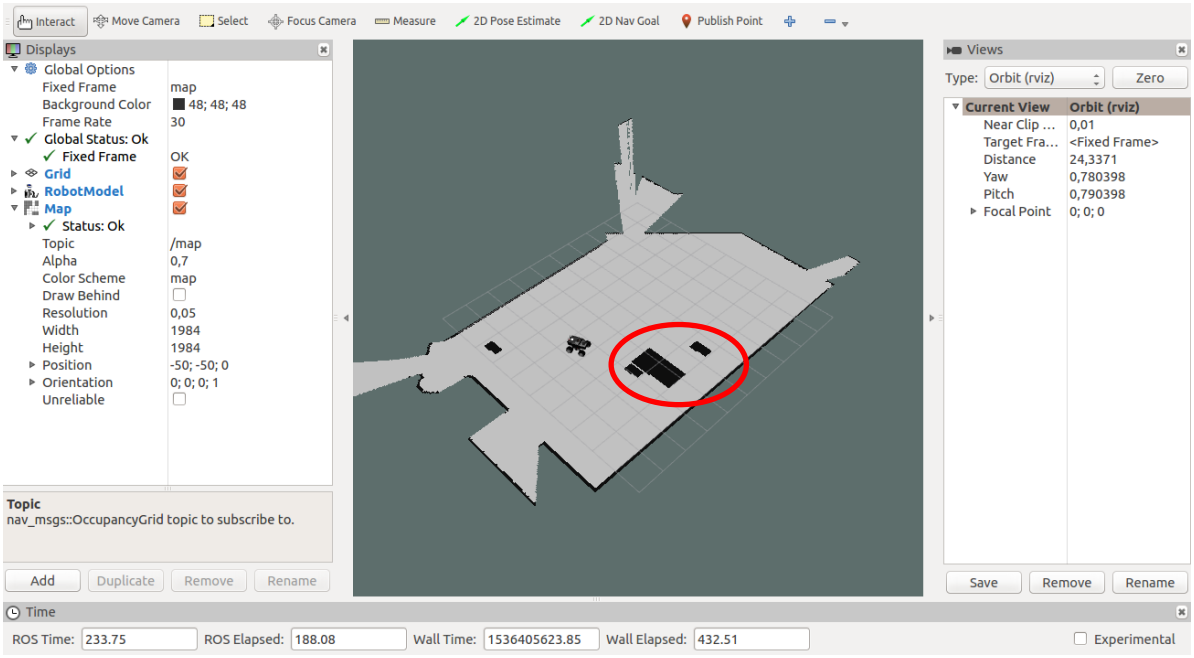

Figura 8-1. Construcción del mapa con leve error

Aparece un error en la zona señalada en rojo en la imagen. El mapa muestra como si hubiera dos obstáculos diferentes que se encuentran muy cerca el uno del otro. Este error podría deberse perfectamente a la interpretación de la escena por parte de la cámara, ya que es conocido que temas como la iluminación o el movimiento o ruido de la escena alteran la visión por computador.

Es obvio que el mapa de la Figura 8-1 no es correcto, pero como hemos comentado, al final crea una zona en torno al obstáculo por la que no debemos de pasar, evitando así caer en él. Esto es posible aceptarlo porque el Summit-XL no navegará por entornos de espacios reducidos y tendrá la posibilidad de esquivar zonas conflictivas.

8.2 Limitaciones y mejoras futuras

Existen dos limitaciones en este trabajo que son obvias. La primera es el hecho de que estamos trabajando con simulaciones y no con el robot real. Será objeto de estudio en un futuro volcar esta aplicación en el robot real, para lo cual será necesario conocer más el robot físicamente y cómo conectarse a él. Además, quizás haya que depurar ciertos aspectos del código, pues por muy buena que sea una simulación como la de gazebo, el mundo real es más caprichoso y cambiante, provocando situaciones inesperadas a veces.

La otra limitación que salta a la vista tiene que ver con la forma de navegar del robot. En nuestro caso se teleopera, lo cual es una opción válida. Sin embargo, se pretenderá más adelante que trabajos como este puedan servir a una navegación autónoma, sin necesidad de participación humana. Esto significaría un aumento de la comodidad en el uso del robot.

En el campo de la visión también nos encontramos con situaciones que se podrían mejorar. En las simulaciones se ha observado que las mediciones que proporciona la cámara de los agujeros que se encuentra son más fiables si nos encontramos a una distancia del agujero comprendida entre los 1.5 y 4 metros. Esto es lógico, pues igual que el ojo humano, una cámara es capaz de interpretar mejor una escena cuanto más cerca está de ella. Aunque a mayores distancias (hasta 8 metros) se pueden reconocer objetos y estimar su posición y tamaño, trataremos siempre de acercarnos al obstáculo para obtener unas medidas más fiables.

Otro aspecto que ya se comentó a lo largo de la memoria es que, además de poder reconocer más de un obstáculo con la cámara en la misma escena, se pueda estimar la posición y tamaño de ambos. En nuestro

caso, como ya se explicó, nuestros objetivos fijaban que nos encontramos en un entorno simple y, además es cierto que la probabilidad de encontrar agujeros tan cercanos es más reducida, por lo que nos limitamos a un único obstáculo por escena. En caso improbable de que nos encontráramos más de un agujero en la misma escena, sólo uno de ellos sería incluido en el mapa de obstáculos, por lo que la información del entorno no quedaría completa.

De los resultados dados en el capítulo 7, deducimos claramente que los obstáculos se reconocen mejor cuando están centrados en la imagen. Esto se podría intentar mejorar, pero en cualquier caso, el robot irá encontrando agujeros prácticamente de frente, pues para construir el mapa investigará la zona minuciosamente.

ANEXO A. MANUAL DE USUARIO

A. 1. Instalación de Ubuntu junto a Windows 8.1

Este apartado no pretende ser una guía exhaustiva para la instalación de una partición de Ubuntu junto a Windows. Se darán los pasos generales que hay que realizar y los enlaces web usados como apoyo. Sin embargo, estos enlaces pueden cambiar de una versión a otra del PC, del sistema operativo que se tiene y del que se quiere instalar, aunque grosso modo, hay que seguir siempre la misma secuencia de acciones. Los datos para los que se ha probado la correcta instalación son los siguientes:

- PC: ASUS VivoBook S551LN
- OS instalado: Windows 8.1
- OS a instalar: Ubuntu 14.04 LTS
- Procesador: Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz 2.40 GHz
- RAM instalado: 8.00 GB (7.89 GB utilizable)
- Tipo de sistema: Sistema operativo de 64 bits, procesador x64

Como referencia principal se tendrán los siguientes enlaces:

- https://bytelix.com/guias/como-instalar-ubuntu-junto-a-windows-paso-a-paso/
- https://www.redeszone.net/2013/03/10/manual-para-instalar-ubuntu-junto-a-windows-8-en-el-mismo-disco-duro/

Los pasos realizados, de manera general, son (se incluyen los enlaces de apoyo para cada uno):

1. Reducir volumen del disco duro (:C) 40 GB siguiendo las recomendaciones del enlace: http://elblogdeliher.com/mi-recomendacion-para-hacer-las-particiones-para-instalar-ubuntu/

2. Descargar Ubuntu (archivo ISO) del enlace: https://www.ubuntu.com/download/alternative-downloads y montarlo en un pen drive usando el programa Rufus recomendado en el primer enlace principal.

Figura A-0-1. Logo del programa Rufus

3. Configurar la Bios del PC para seleccionar las prioridades en el arranque. Puede que haya que configurar la UEFI según el PC. Este paso se realiza junto al paso 4 en función de las necesidades de cada ordenador, pues puede que no haya que configurar nada. El acceso a la Bios de Windows en el arranque se puede

realizar de diferentes maneras que también varían según el PC, por lo que lo más recomendable es buscar en internet según nuestras necesidades.

4. Reiniciar el ordenador con el pen drive (que contiene la ISO montada) puesto para acceder a la versión LiveCD de Ubuntu que te permite modo prueba y modo instalación. A la hora de elegir el pen drive en la pantalla de inicio aparecen dos nombres; IMPORTANTE: se escoge el del pen drive que empieza con la palabra UEFI (a diferencia de lo que dice en el vídeo siguiente). Para conseguir que el ordenador arranque con el pen drive se han seguido los pasos de este vídeo: https://www.youtube.com/watch?v=QvVEP6QBcCE

5. Instalar Ubuntu desde la versión LiveCD siguiendo los pasos de los dos enlaces principales (similares) y con las recomendaciones para las particiones del enlace ya citado: http://elblogdeliher.com/mi-recomendacion-para-hacer-las-particiones-para-instalar-ubuntu/.

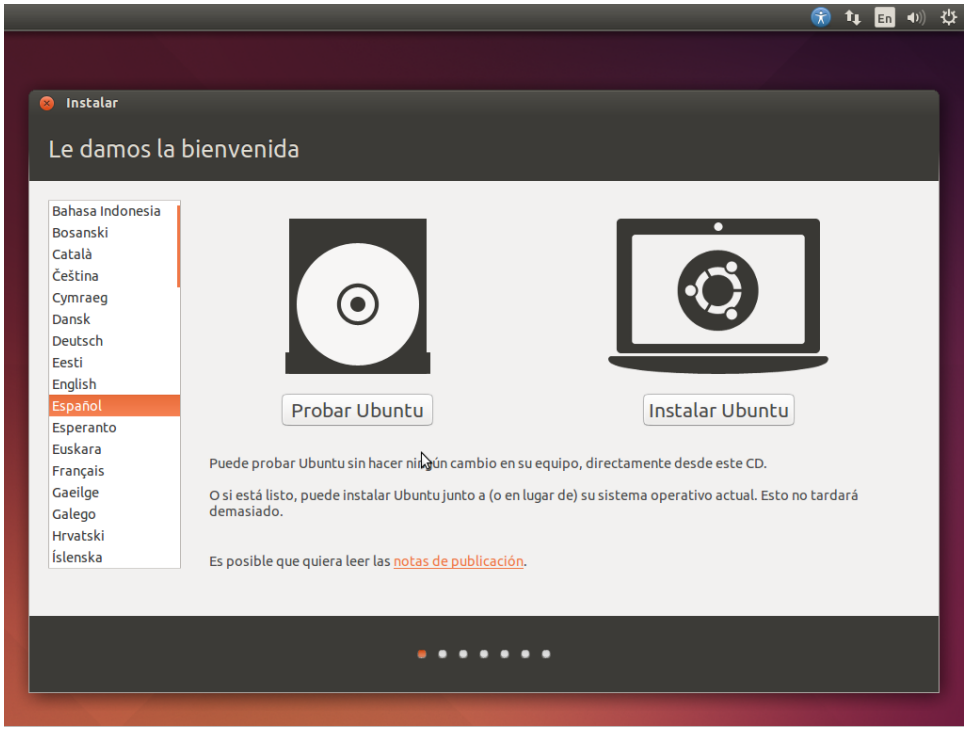

Figura A-0-2. Versión LiveCD de Ubuntu

6. Una vez terminada la instalación, reiniciar y ya te aparece la selección del sistema operativo (con el Grub de Ubuntu): Ubuntu por defecto o Windows. También permite acceder a la Bios (última opción: System setup). Si esta pantalla (ver Figura A-0-3) o una similar no apareciera, será necesario configurar la Bios del PC.

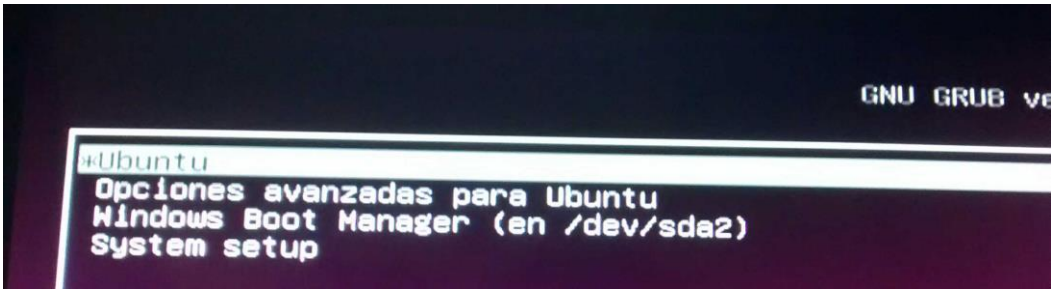

Figura A-0-3. Grub que aparece al iniciar el PC tras la instalación de la partición

A. 2. Instalación de ROS Indigo Igloo

Se va a desarrollar cómo instalar ROS Indigo Igloo paso a paso en nuestra sesión de Ubuntu 14.04 LTS. Para ello seguimos las instrucciones de la web de ROS [40]:

1. Abrimos una terminal de Ubuntu (Ctrl+Alt+T) para poder escribir los comandos necesarios.

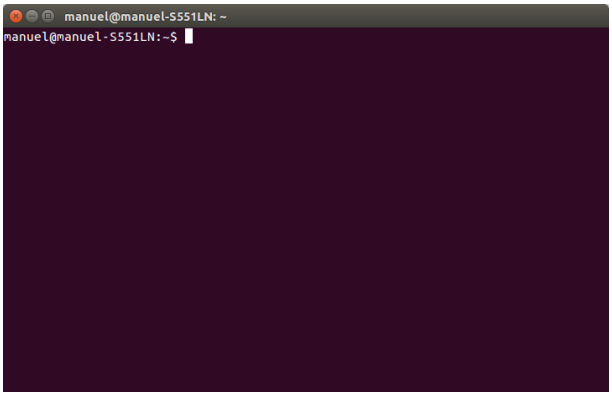

Figura A-0-4. Apariencia de una terminal o intérprete de comandos

2. Configuramos nuestro PC para aceptar software de packages.ros.org, escribiendo en la línea de comandos:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc
) main" > /etc/apt/sources.list.d/ros-latest.list'
```

3. Configuramos nuestros keys:

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-ke
y 421C365BD9FF1F717815A3895523BAEEB01FA116
```

4. Nos aseguramos de que nuestro paquete Debian está actualizado:

```
sudo apt-get update
```

5. Hay muchas librerías y herramientas en ROS, por lo que se pueden instalar más adelante. Sin embargo, seguimos la opción recomendada de instalar el paquete completo:

```
sudo apt-get install ros-indigo-desktop-full
```

6. Antes de usar ROS, hay que inicializar rosdep. rosdep permite instalar fácilmente las dependencias del sistema para la fuente que desea compilar y es necesario para ejecutar algunos componentes principales en ROS:

```
sudo rosdep init

rosdep update
```

7. Configuración del environment (entorno de trabajo). Se vuelcan las variables de entorno a un archivo .bashrc cada vez que se abra una nueva terminal para facilitar el trabajo y se ejecuta dicho archivo:

```
echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

8. Instalamos la herramienta rosinstall que permite desde la línea de comandos descargar muchos packages de ROS con un solo comando:

```
sudo apt-get install python-rosinstall
```

A. 3. Comandos útiles para ROS

Para muchos de los que se inician en ROS puede ser novedoso trabajar sobre una terminal o "shell", escribiendo órdenes en su línea de comandos. Un buen uso de la terminal nos facilitará la tarea de trabajar con ROS. En primer lugar mostramos una serie de comandos básicos comunes a cualquier sistema basado en Unix [41]:

```
mkdir -p [ruta]
```

Ejemplo: mkdir –p ~/catkin_ws/src

Crea la carpeta de nombre *catkin_ws* dentro de nuestra carpeta personal y, dentro de ella, la carpeta *src*. El argumento *–p* crea los directorios padres necesarios en caso de no existir.

```
cd [ruta]
```

Ejemplo: cd ~/catkin_ws

Nos movemos a la carpeta de ruta Carpeta personal (~) → *catkin_ws*.

```
cd ..
```

Vuelta al directorio que está justo por encima del actual.

```
cd ../..
```

Nos movemos dos directorios arriba del que estábamos actualmente.

```
cd -
```

Se vuelve al directorio en el que trabajábamos justo antes.

```
pwd
```

Muestra la ruta del directorio en el que estamos trabajando actualmente.

```
ls
```

Muestra el contenido de la carpeta de trabajo actual.

```
cat [fichero]
```

Ejemplo: cat package.xml

Muestra el fichero por pantalla.

Tras esto, es conveniente ampliarlo a comandos de ROS basándonos en los tutoriales [2]:

```
rospack find [package_name]
```

Ejemplo: rospack find key_teleop

Nos muestra el directorio en que se encuentra el package.

```
roscd [package_name]
```

Nos lleva al directorio de ese package.

```
rosls [package_name]
```

Muestra el contenido de ese package.

```
source /opt/ros/indigo/setup.bash
```

Comando que debemos ejecutar siempre que abramos una nueva terminal para acceder a los comandos de ROS. Sustituir la palabra *indigo* por la versión de ROS que se tenga instalada.

```
catkin_make
```

Se ejecuta este comando trabajando en un workspace y sirve para generar los cambios, configurar y compilar los packages.

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

Una vez ejecutado *catkin_make* con éxito, cargamos el workspace con este comando.

```
catkin_create_pkg [package_name] [dependencies]
```

Ejemplo: catkin_create_pkg vision_summit_xl sensor_msgs cv_bridge roscpp std_msgs image_transport

En el ejemplo se crea el package de nombre *visión_summit_xl*, el cual depende de *sensor_msgs*, *cv_bridge*, *roscpp*, *std_msgs* e *image_transport*.

```
rospack depends1 [package_name]
rospack depends [package_name]
```

La primera forma muestra solo las dependencias directas del package. La segunda muestra todas las dependencias.

```
roscore
```

Se ejecuta este comando siempre al principio de trabajar con ROS en una terminal aparte, a menos que se ejecute un lanzador (archivo .launch), pues este lanzará roscore de manera automática si no lo está. roscore iniciará:

- El ROS Master
- El ROS Parameter Server
- El nodo rosout, que es el equivalente en ROS a stdout/stderror

```
rosnode list
```

Muestra una lista en tiempo real de los nodos en ejecución.

```
rosnode info [node_name]
```

Ejemplo: rosnode /hokuyo_base/scan

Muestra información sobre el nodo.

```
rosrun [package_name] [node_name]
```

Ejemplo: rosrun gmapping slam_gmapping

Ejecuta el nodo de nombre *node_name*, perteneciente al package de nombre *package_name*.

```
sudo apt-get install ros-indigo-[package_name]
```

Ejemplos: sudo apt-get install ros-indigo-rqt

sudo apt-get install ros-indigo-rqt-common-plugins

Instala el package de nombre *package_name*. Sustituir *indigo* por la versión de ROS instalada.

```
rosrun rqt_graph rqt_graph

rqt_graph
```

Un comando muy útil es ejecutar *rqt_graph*, que nos muestra un gráfico de los nodos en ejecución y sus conexiones por medio de los topics. Se puede ejecutar de ambas formas y hay que instalarlo previamente con los dos comandos del ejemplo anterior. El resultado es un gráfico como el de la Figura A-0-5.

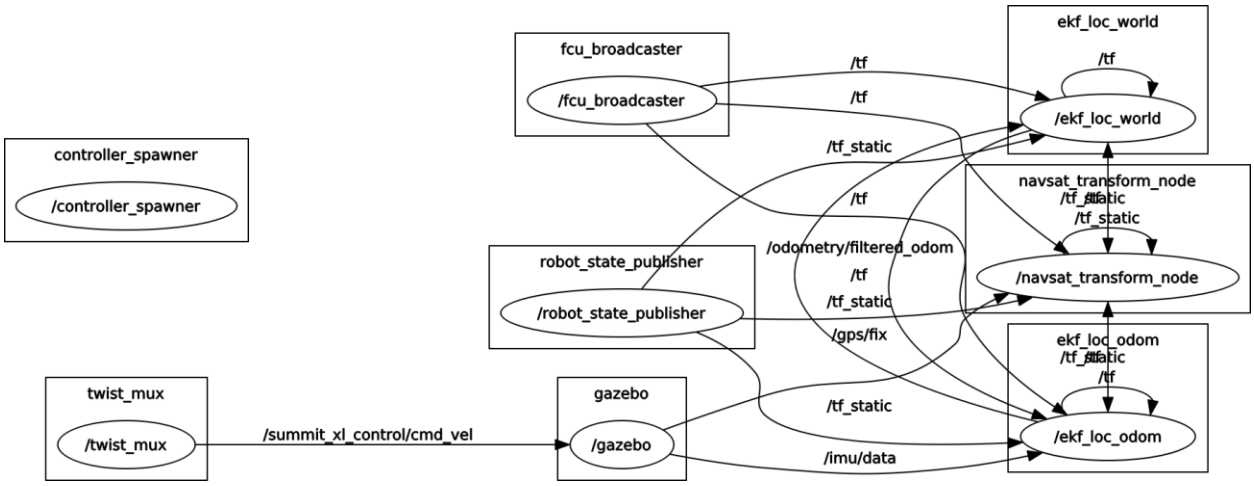

Figura A-0-5. Ejemplo de grafo de nodos de un sistema ROS

```
rostopic echo [topic_name]
```

Ejemplo: rostopic echo Obstacle_data

Muestra los datos publicados en el topic.

```
rostopic list -v
```

Nos muestra una lista completa de publicadores y suscriptores a los topics.

```
rostopic type [topic_name]
```

Te devuelve el tipo de mensaje que se publica en ese topic.

```
rostopic type [topic_name] | rosmsg show
```

Te da información sobre el tipo de mensaje que se publica en el topic y su estructura.

```
rostopic pub [topic_name] [message_type] [args]
```

Sirve para publicar un mensaje del tipo especificado en el topic que deseemos y con los argumentos dentro del mensaje indicados.

```
rosrun rqt_plot rqt_plot
```

Ejecuta un gráfico temporal donde podemos ver todos los datos publicados en cualquiera de los topics. La apariencia de la pestaña del gráfico es la de la Figura A-0-6. Para añadir topics a ver en el gráfico hay que escribir el nombre del mismo en la barra superior izquierda.

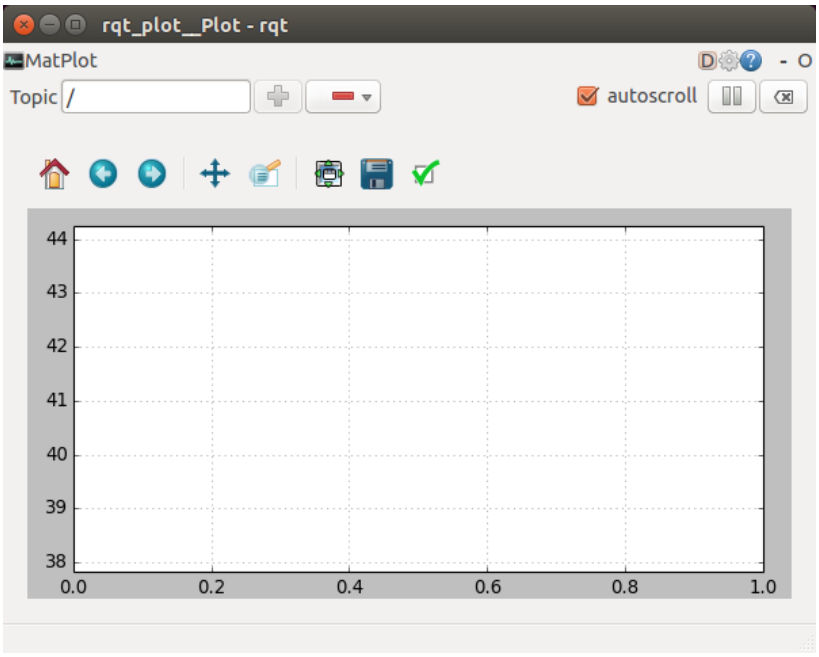

Figura A-0-6. Ventana gráfica de rqt_plot

```
rosservice list
```

Nos muestra una lista de los servicios.

```
rosservice type [service_name]
```

Te devuelve el tipo de ese servicio.

```
rossrv show [service_type]
```

Te da información sobre ese servicio: argumentos de la pregunta y la respuesta.

```
rosservice call [service_type][args]
```

Se llama a un servicio como cliente con los argumentos de la pregunta.

```
roslaunch [package_name] [filename.launch]
```

Ejemplo: roslaunch summit_xl_sim_bringup summit_xl_complete.launch

Se llama a un archivo de lanzamiento de nodos (extensión .launch) que se encuentra dentro del package citado.

```
rosmsg show [message_type]
```

Ejemplo: rosmsg show geometry_msgs/Point32

Nos muestra la estructura de ese tipo de mensaje, con los tipos de datos que lo componen. Si no nos acordamos del nombre del package donde se encuentra el mensaje, se puede omitir. En el ejemplo sería: *rosmsg show Point32*. En ese caso, además de la estructura del mensaje, se nos devuelve el package al que pertenece.

```
chmod +x codigo.py
```

Comando para generar un ejecutable a partir de un código escrito en python (extensión .py).

```
mkdir ~/bagfiles
cd ~/bagfiles
rosbag record -a
```

Con estos comandos grabamos los mensajes publicados por todos los topics activos en una carpeta que acabamos de crear llamada bagfiles. Se guardarán con un nombre automático y extensión .bag. Para parar de grabar los datos basta con teclear Ctrl + C.

```
rosbag record –O [nombre_archivo] [topic_names]
```

Igual que el comando anterior pero con la opción –O, que nos permite guardar el archivo con el nombre señalado en *nombre_archivo* y grabar sólo los topic que se especifiquen.

```
rosbag info <your bagfile>
```

Ejemplo: rosbag info ejemplo.bag

Te muestra información del archivo .bag, como puede ser los topics que se han grabado y el tipo de mensaje que contienen.

```
rosbag play <your bagfile>
```

Ejecuta el contenido de los datos guardados en ese archivo .bag. Sirve para repetir acciones guardadas. Por ejemplo, si grabamos la trayectoria de un robot móvil, y la ejecutamos con este comando, el robot repetirá la misma trayectoria.

```
roscd
roswtf
```

Roswtf es un analizador de errores y hace un chequeo del sistema. Te muestra por pantalla si encuentra errores, advertencias o algo sospechoso. (Nota: *roscd* te lleva a la dirección */opt/ros/indigo*, entendiendo *ingido* como la versión de ROS que se tenga instalada).

```
[comando_ROS] -h
```

Ejemplo: rosmsg show –h.

Si escribimos cualquier comando de ROS seguido del argumento *–h*, nos aparecerá la ayuda desglosando las posibles opciones que tenemos con ese comando, como pueden ser otros argumentos.

A. 4. Descarga de packages de partida

Es necesario acudir al repositorio web donde se encuentran los códigos de la empresa Robotnik y descargarse los siguientes packages:

- summit_xl_common: accedemos al enlace [25] y hacemos click en Clone or download → Download ZIP (ver Figura A-0-7), asegurándonos de escoger bien en la pestaña “Branch” (arriba a la izquierda) nuestra versión: indigo-multirobot-devel.

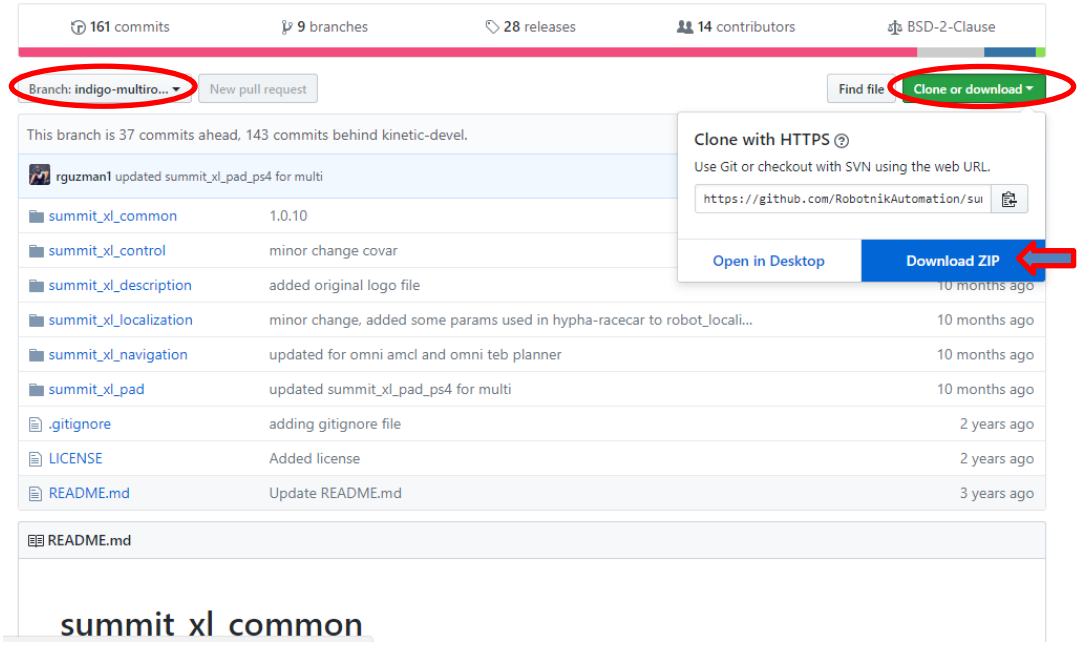

Figura A-0-7. Descarga del package summit_xl_common

- summit_xl_sim: siguiendo los pasos del package anterior, descargamos este package del enlace [26], tal como indica la Figura A-0-8.

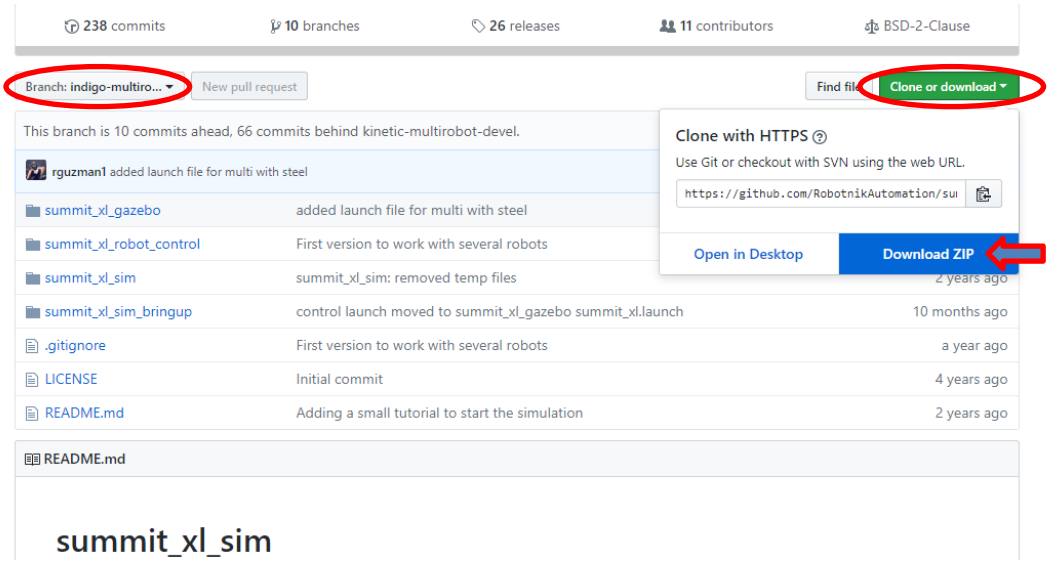

Figura A-0-8. Descarga del package summit_xl_sim

- robotnik_msgs: igual que los anteriores, pero seleccionando en la pestaña “Branch” la versión master. El enlace es [27] y se representa en la Figura A-0-9.

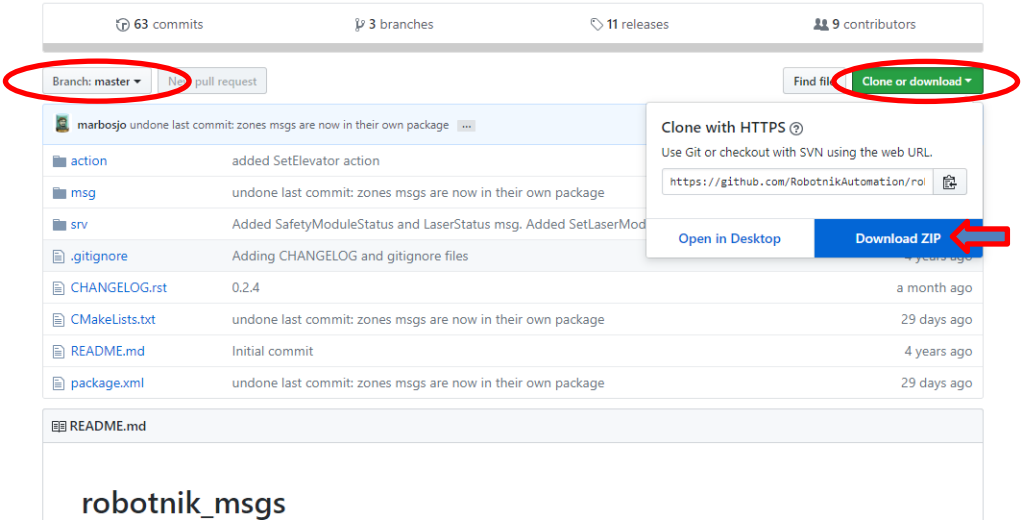

Figura A-0-9. Descarga del package robotnik_msgs

- robotnik_sensors: entrando en el enlace [28] y siguiendo las instrucciones de los dos primeros packages, descargamos este tal y como se observa en la Figura A-0-10.

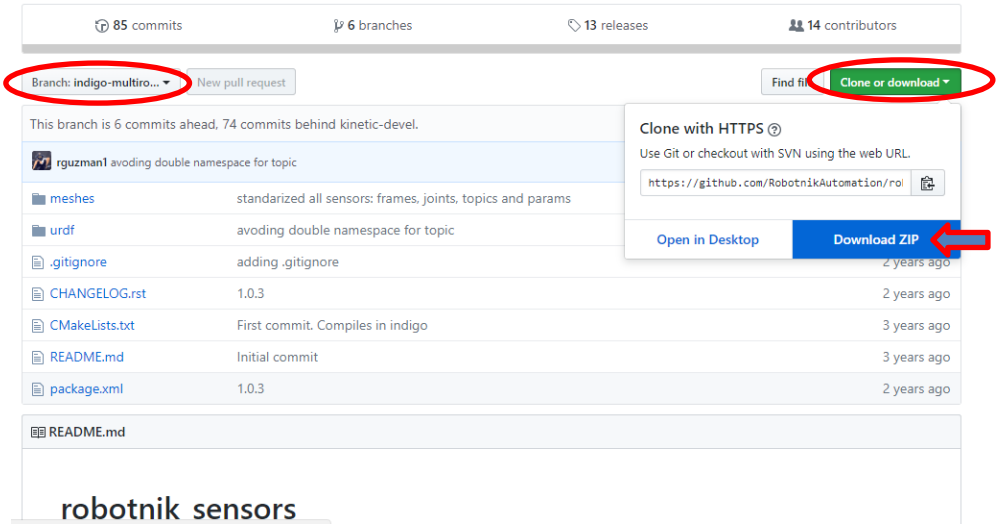

Figura A-0-10. Descarga del package robotnik_sensors

- robotnik_trajectory_suite: a partir del enlace [29], seleccionando en la pestaña "Branch" la versión indigo-devel y siguiendo las indicaciones de la Figura A-0-11, nos descargamos este package (recordamos que era opcional).

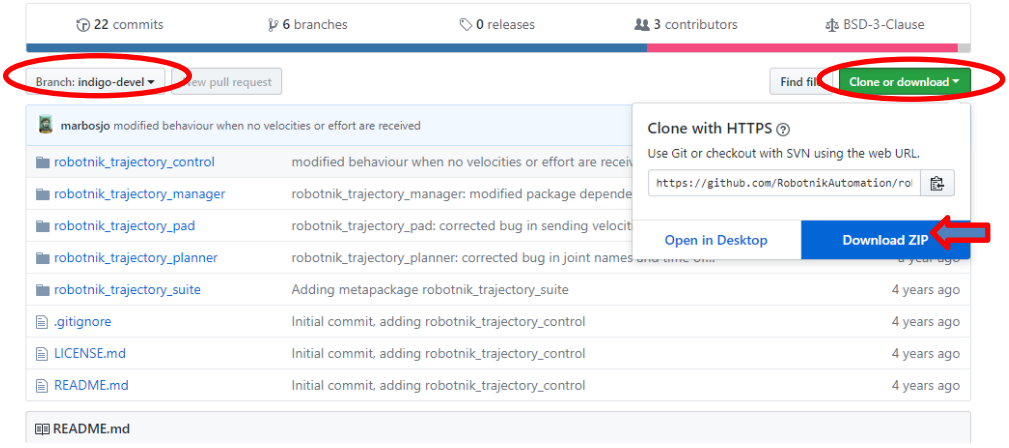

Figura A-0-11. Descarga del package robotnik_trajectory_suite

A. 5. Creación de nuestro workspace

Una vez tenemos instalado ROS y descargado los packages, el siguiente paso será lanzar una simulación. Para ello es necesario un paso previo de organización de archivos y creación de un workspace desde el que trabajar con nuestro sistema. Para ello hay que seguir los siguientes pasos [2]:

1. Abrir una terminal (Ctrl+Alt+T) y escribir los siguientes comandos (para más información sobre los mismos nos remitimos al apartado 3. Comandos útiles para ROS de este anexo):

```
source /opt/ros/indigo/setup.bash

mkdir –p ~/catkin_ws/src

cd ~/catkin_ws

catkin_make

source devel/setup.bash
```

2. Comprobamos si el workspace se ha cargado correctamente (este paso es opcional, pero sirve para asegurarnos de que lo que vamos haciendo funciona):

```
echo $ROS_PACKAGE_PATH
```

   Tras ejecutar ese comando nos debe aparecer lo siguiente:

```
/home/youruser/catkin_ws/src:/opt/ros/indigo/share:/opt/ros/indigo/stac
ks
```

   Nota: en "youruser" debe aparecer tu nombre de usuario y en "indigo", la versión de ROS instalada.

3. Organizamos los archivos en nuestro workspace, de nombre *catkin_ws*. Para ello:
   a. Nos vamos al explorador de archivos de Ubuntu y entramos en la carpeta "Descargas". Ahí extraemos los 5 archivos .zip descargados anteriormente como packages. Se observa que los packages están nombrados como los conocemos, seguido de un guión con la versión del package descargada. Por ejemplo, el package *summit_xl_sim* ahora pasa a llamarse *summit_xl_sim-indigo-multirobot-devel*.
   b. Copiamos los 5 archivos ya extraídos en la carpeta "src" de nuestro workspace.

A. 6. Lanzamiento de una simulación completa

Tras completar el apartado anterior, estamos en disposición de ejecutar una simulación. Hay que seguir los siguientes pasos:

1. Instalar una serie de dependencias necesarias para compilar los packages [42] y [17]:

```
sudo apt-get install ros-indigo-moveit-core

sudo apt-get install ros-indigo-mavlink ros-indigo-mavros ros-indigo-re
altime-tools ros-indigo-robot-localization ros-indigo-controller-manage
r ros-indigo-transmission-interface ros-indigo-joint-limits-interface r
os-indigo-control-toolbox ros-indigo-twist-mux ros-indigo-slam-gmapping
ros-indigo-gazebo-ros-pkgs ros-indigo-robotnik-msgs ros-indigo-robotnik
-sensors ros-indigo-uuid-msgs ros-indigo-geographic-msgs
```

2. Compilamos y construimos nuestro workspace [2]:

```
catkin_make
```

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

Si al ejecutar catkin_make aparecen errores de que no se puedan encontrar archivos de configuración de algún package, aun habiendo ejecutado el paso número 1, ejecutamos lo siguiente [43]:

```
cd ~/catkin_ws
rosdep install --from-paths./src --ignore-src --rosdistro indigo -y
```

Esto instalará todas las dependencias necesarias de los packages para construir nuestro código fuente en el workspace. Si hemos tenido que realizar esto, hay que volver a compilar y construir con los dos primeros comandos de este paso.

3. Lanzamos la simulación completa con:

```
roslaunch summit_xl_sim_bringup summit_xl_complete.launch
```

Se observa que aparece el Summit-XL pero sin dos sensores fundamentales: el sensor láser y la cámara. Para arreglar esto, acudimos a la carpeta **summit_xl_common-indigo-multirobot-devel → summit_xl_description → robots** y abrimos el código de nombre **summit_xl.urdf.xacro**. Al final de ese código aparecen los sensores a añadir al robot. Descomentamos lo siguiente:

- Líneas 77-79, correspondiente al sensor láser. Debe quedar así:

```
<xacro:sensor_hokuyo_utm30lx name="hokuyo_base" parent="base_link">
        <origin xyz="0.0 0.0 0.33" rpy="0 0 0"/>
</xacro:sensor_hokuyo_utm30lx>
```

- Líneas 91-93, correspondiente a la cámara. Debe quedar así:

```
<xacro:sensor_axis name="camera" parent="base_link">
        <origin xyz="0.19 0 0.17" rpy="0 ${-75*PI/180} 0"/>
</xacro:sensor_axis>
```

Repitiendo los pasos 2 (sin la instalación de dependencias) y 3 anteriores, lanzamos una simulación en la que ahora sí aparecen la cámara y el sensor láser.

El entorno de gazebo está vacío, por lo que no hay obstáculos, algo poco realista. Para no tener que añadir obstáculos manualmente, podemos hacer uso de otro world que trae el Summit-XL, en el cual nos situaremos en un recinto interior con muchas habitaciones y pasillos. Para que este entorno aparezca en la simulación tenemos que entrar en la carpeta **summit_xl_sim-indigo-multirobot-devel → summit_xl_gazebo → launch** y modificar el código del archivo **summit_xl.launch**. La modificación será:

- Cambiar la línea 7, que era:

```
<arg name="world" default="summit_xl.world"/>
```

Por lo siguiente:

```
<arg name="world" default="summit_xl_office.world"/>
```

Ahora sí, nos aparecerá nuestro robot con todos sus accesorios en un entorno de simulación con obstáculos para poder hacer pruebas. Puede verse un ejemplo de la simulación en el apartado 4.4.2 de esta memoria (Figura 4-8).

A. 7. Añadir teleoperación a la simulación

El robot se moverá tras recibir órdenes en el topic /cmd_vel. Pero al lanzar una simulación se observa lo siguiente:

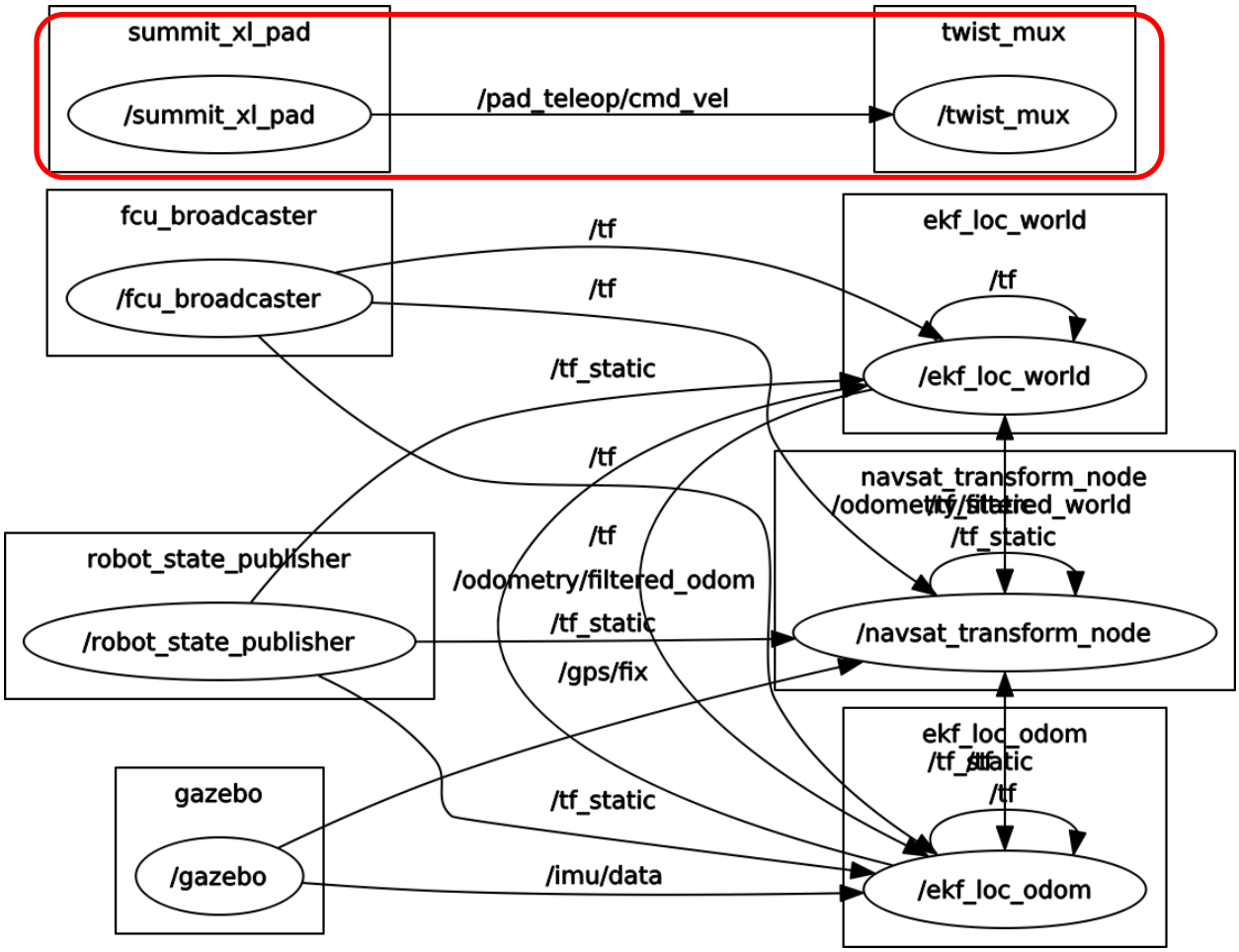

Figura A-0-12. Resultado de ejecutar rqt_graph con una simulación completa

Se observa que es el nodo el summit_xl_pad el que manda velocidades al nodo twist_mux y este, no lo comunica a nadie. Como el nodo summit_xl_pad será para teleoperaciones con mandos de videoconsolas (mandos de PS3, PS4, Xbox o Logitech), buscamos reemplazarlo por el nodo que teleopere desde el teclado. Para ello hay que seguir los siguientes pasos [30]:

1) Descargamos un package para teleoperación con las flechas del teclado:

```
sudo apt-get install ros-indigo-key-teleop
```

2) Ejecutamos el nodo de teleoperación recién descargado con el comando siguiente, que redirige su salida de key_teleop en key_vel al topic /cmd_vel:

```
rosrun key_teleop key_teleop.py key_vel:=cmd_vel
```

El resultado es el de la Figura A-0-13. Ya tenemos una teleoperación por teclado (nodo key_teleop) que manda órdenes al nodo twist_mux. Sin embargo, sigue apareciendo un problema: el nodo twist_mux no conecta con nadie y debería hacerlo con gazebo para enviar órdenes al robot. De hecho, si intentamos mandar órdenes con la simulación abierta, el robot no se mueve (IMPORTANTE: para que al pulsar las flechas se manden comandos, la terminal de teleoperación debe ser la ventana activa en ese momento). Otra cosa conveniente sería quitar el nodo summit_xl_pad ya que no lo vamos a usar.

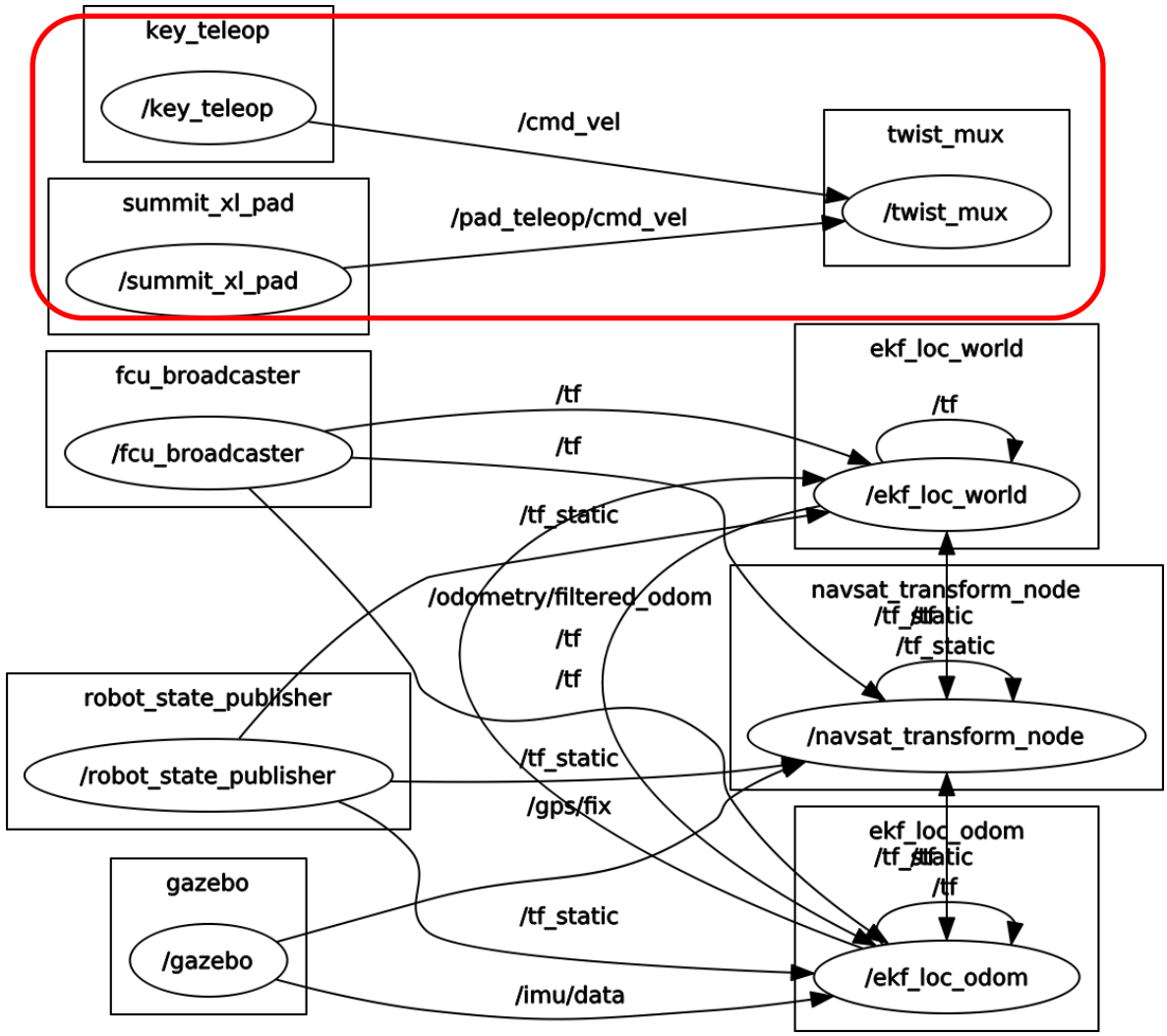

Figura A-0-13. Resultado de ejecutar rqt_graph tras llamar a la teleoperación

3) Dejamos de llamar al nodo summit_xl_pad para evitar usar recursos innecesarios. Para ello entramos en la carpeta **summit_xl_sim-indigo-multirobot-devel → summit_xl_gazebo → launch** y comentamos la línea 33 del código del lanzador **summit_xl.launch**, quedando ahora de la siguiente manera:

```
<!--<include file="$(find summit_xl_pad)/launch/summit_xl_pad.launch"
/>-->
```

4) Añadimos el plugin de gazebo para la configuración skid-steering, haciendo que la teleoperación acabe conectándose con gazebo definitivamente. Hay que acceder a la carpeta **summit_xl_common-indigo-multirobot-devel → summit_xl_description → robots** y modificar el código del archivo **summit_xl.urdf.xacro**. Lo que se hace es descomentar las líneas 49 y 50, lo cual permite que se conecten el nodo twist_mux con el nodo gazebo a través del topic /summit_xl_control/cmd_vel. Estas líneas quedan de la siguiente forma:

```
<xacro:ros_force_based_move broadcastOdomTF="0"/>
<xacro:skid_steering broadcastOdomTF="0"/>
```

El resultado de aplicar las modificaciones 3 y 4 es el grafo de la Figura A-0-14 . Se observa perfectamente cómo el nodo de teleoperación manda órdenes de velocidad al nodo twist_mux y este, con las modificaciones necesarias para conseguir mover una configuración skid-steering, las envía a gazebo. Si lo probamos en simulación (recordar tener la ventana de teleoperación activa), vemos cómo el robot se mueve y responde a las órdenes que le mandamos. La apariencia que tiene junto con la simulación la podemos ver en la Figura 4-10.

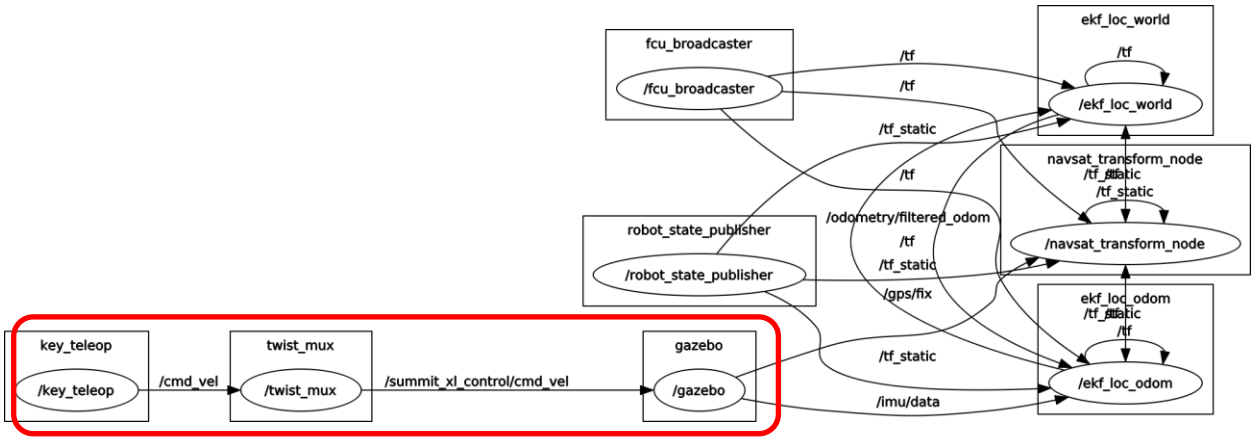

Figura A-0-14. Resultado definitivo en rqt_graph tras aplicar la teleoperación

A. 8. Conexión de OpenCV con ROS: lanzamiento de la cámara

Como ya se ha comentado, el stack de nombre visión_opencv tiene que estar incluido en los packages instalados de ROS si se han seguido los pasos de este manual, pues se eligió la instalación de ROS completa. En caso poco probable de no estarlo habrá que instalarlo manualmente.

Para el lanzamiento de la cámara habrá que realizar un paso previo. Tenemos que asignar un nombre para nuestra cámara y otro para el topic en el que esta va a publicar la información [44] y [45]. Para ello tenemos que modificar el archivo de nombre **axis.urdf.xacro**, situado en: **robotnik_sensors-indigo-multirobot-devel → urdf**. Se cambiarán las líneas de código 156 y 157, las cuales se encuentran en la parte del mismo que añade el plugin de la cámara a gazebo. En la primera de ellas asignamos el nombre summit_xl/camera1 a la cámara y en la segunda, image_raw para el topic en el que publica información. De esta manera, si queremos obtener la información de la cámara, debemos suscribirnos al topic de nombre **/summit_xl/camera1/image_raw**. Estas dos líneas de código quedan de la siguiente manera:

```
<cameraName>summit_xl/camera1</cameraName>
<imageTopicName>image_raw</imageTopicName>
```

Tras esto, tenemos que adaptar la cámara para que apunte al suelo como queremos. Esto hace necesario modificar la configuración que trae la descripción de la cámara por defecto para los ángulos de barrido o *pan* y de inclinación o *tilt* [44], [45] y [46]. Se observa que al lanzar la cámara con la configuración por defecto, tanto el ángulo de barrido como el de inclinación están en continuo movimiento.

El ángulo de barrido queremos que se quede fijo sobre el eje simetría del robot y apunte al frente y no hacia los lados. Para ello, entramos de nuevo en el código axis.urdf.xacro y modificamos la línea 43 definiendo la articulación de barrido o *pan* como fija. Lo que se hace es cambiar la palabra "revolute" por "fixed". Queda de la siguiente forma:

```
<joint name="joint_${name}_pan" type="fixed">
```

Además, se elimina la transmisión de fuerza a la articulación de revolución del ángulo de barrido. Para ello, se comentan las líneas de la 61 a la 70 del código en cuestión, quedando así:

```
<!--<transmission name="${name}_pan_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="joint_${name}_pan">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="pan_motor">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
<mechanicalReduction>${ptz_mechanical_reduction}</mechanicalReduction>
  </actuator>
</transmission>-->
```

La manera de proceder con el ángulo de inclinación es diferente. Ahora queremos forzar su posición para que

apunte al suelo sin que la imagen interfiera con el chasis del robot. Tras varias comprobaciones experimentales, se llega a la conclusión de que la articulación *tilt* debe estar inclinada hacia el suelo un ángulo de 1.3708 rad (lo que equivale aproximadamente a unos 79º) medidos desde la horizontal de la cámara.

Por tanto, en el código axis.urdf.xacro dejaremos en la línea 72 la articulación tilt como "revolute" o revolución y lo que modificaremos será la línea 77, donde configuraremos el ángulo de inclinación al valor que ya hemos comentado. Esta línea queda de la siguiente manera:

```
<limit effort="${ptz_joint_effort_limit}"
velocity="${ptz_joint_velocity_limit}" lower="-1.3708" upper="-1.3708"/>
```

Además, al igual que se hacía con el ángulo de barrido, eliminamos la transmisión para evitar que se manden órdenes de movimiento a la articulación de inclinación. Se comentan, por tanto, las líneas de código de la 90 a las 99:

```
<!--<transmission name="${name}_tilt_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="joint_${name}_tilt">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="tilt_motor">
    <hardwareInterface>PositionJointInterface</hardwareInterface>
<mechanicalReduction>${ptz_mechanical_reduction}</mechanicalReduction>
  </actuator>
</transmission>-->
```

Una vez hemos preparado la cámara para su uso, estamos en disposición de lanzarla para poder representar la escena y analizarla usando la librería OpenCV. Para ello, hay que seguir una serie de pasos [33]:

1. Abrir una terminal (Ctrl+Alt+T) y crear dentro de nuestro workspace un package para el tratamiento de imágenes. Este package se llamará, por ejemplo, **visión_summit_xl** y dependerá de: sensor_msgs, cv_bridge, roscpp, std_msgs y de image_transport. Este paso se realiza copiando lo siguiente en la línea de comandos:

```
source /opt/ros/indigo/setup.bash

cd ~/catkin_ws/src

catkin_create_pkg vision_summit_xl sensor_msgs cv_bridge roscpp std_msg
s image_transport
```

2. Creamos la carpeta de nombre **src** dentro del package vision_summit_xl ya creado. En esa carpeta creamos un nuevo archivo con el nombre **image_converter.cpp**, que será nuestro código fuente para la cámara. Abrimos ese archivo y copiamos en él nuestro código para la cámara. En el caso de este trabajo será el que se encuentra en el Anexo B. Códigos desarrollados. Si se quiere partir de cero, en el tutorial de [33] encontramos una plantilla para trabajar nuestro código. Una vez lo tengamos copiado, guardamos y cerramos.

3. Accedemos al archivo de nombre CMakeLists.txt, situado dentro del package visión_summit_xl. Al final de este archivo de texto incluimos las dos líneas siguientes:

```
add_executable(image_converter src/image_converter.cpp)

target_link_libraries(image_converter ${catkin_LIBRARIES})
```

   Esto nos permite crear el ejecutable de nombre **image_converter** a partir del código fuente image_converter.cpp una vez compilemos y se construya el código. Este ejecutable lo podremos encontrar en la carpeta: Carpeta personal → catkin_ws → devel → lib → visión_summit_xl.

4. En una nueva terminal, llevamos a cabo una simulación completa tal y como lo hemos hecho otras veces escribiendo en la línea de comandos:

```
source /opt/ros/indigo/setup.bash
```

```
cd ~/catkin_ws
catkin_make
source devel/setup.bash
roslaunch summit_xl_sim_bringup summit_xl_complete.launch
```

Y en otra terminal, lanzamos el ejecutable de la cámara copiando lo siguiente:

```
source /opt/ros/indigo/setup.bash
cd ~/catkin_ws
catkin_make
source devel/setup.bash
rosrun vision_summit_xl image_converter
```

Tras ejecutar esto, nos aparece una nueva ventana en la que se retransmite en tiempo real la escena de la simulación de gazebo (ver Figura A-0-15). También se podría retransmitir otras imágenes fruto del tratamiento de la escena e incluso lanzar varias ventanas de imagen para ver la evolución del tratamiento.

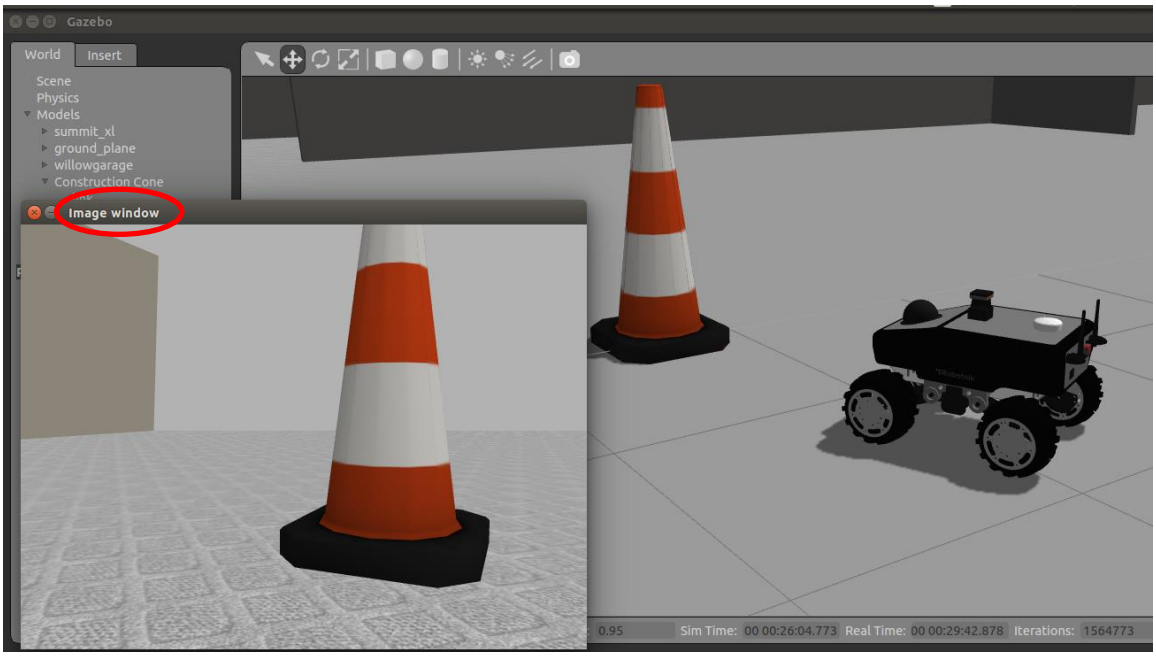

Figura A-0-15. Ejemplo de lanzamiento de la cámara

Si abrimos otra terminal y ejecutamos el comando **rqt_graph**, obtendremos el grafo de nodos y topics resultante (Figura A-0-16). Se observa en la parte inferior cómo el nodo de gazebo publica información en el topic /summit_xl/camera1/image_raw, siendo el nodo image_converter el que se suscribe al mismo, conectando así el mundo de ROS y gazebo con la librería de tratamiento de imágenes OpenCV.

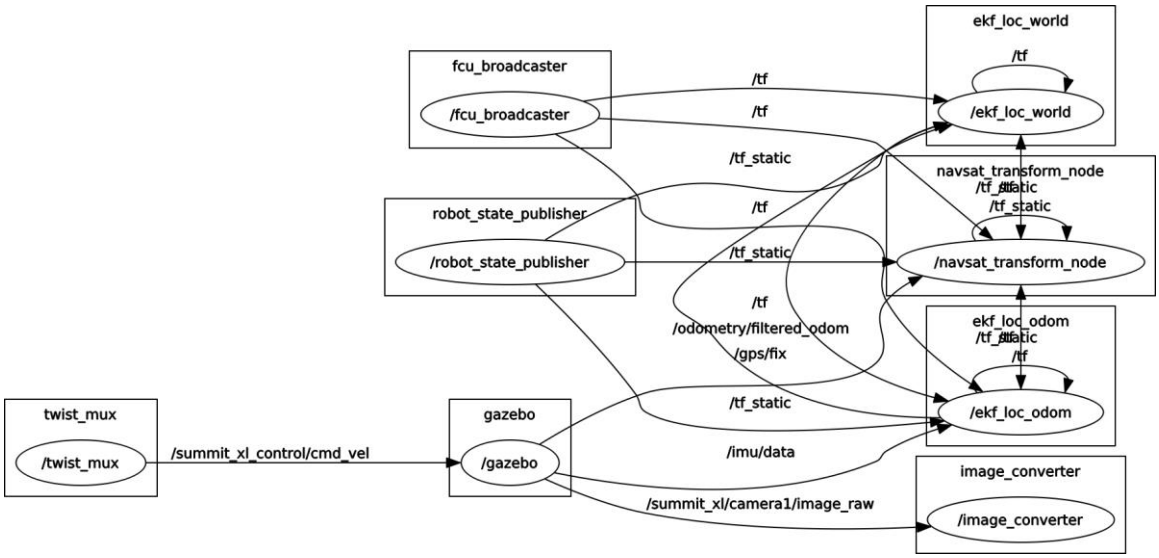

Figura A-0-16. Resultado de ejecutar rqt_graph tras lanzar la cámara

A. 9. Creación de objetos en gazebo

Gazebo permite crear modelos de objetos que podrán ser usados como robots, escenarios o como obstáculos, por ejemplo. En el caso de este trabajo se necesita algún obstáculo que se asemeje a un agujero oscuro para incluirlo en la simulación y ser capaz de reconocerlo. La idea será crear un cilindro de color negro, de altura despreciable y radios variados para hacer diferentes pruebas. Una vez creados se repartirán por el suelo de la simulación, representando esos agujeros.

Para crearlos seguimos el tutorial de [47]. Según este serán necesarios los siguientes pasos:

1. Dentro de los modelos que incluye gazebo hay que incluir una carpeta con el nombre que tendrá nuestro modelo. Abrimos una terminal y escribimos:

   ```
   mkdir -p ~/.gazebo/models/hole
   ```

   En nuestro caso, el modelo se llamará “hole”.

2. Se crea el archivo de configuración del modelo escribiendo lo siguiente en la línea de comandos: :

   ```
   gedit ~/.gazebo/models/hole/model.config
   ```

   Esto abre un editor de texto para que lo completemos con dicha configuración. Para el agujero escribimos lo siguiente, guardamos y cerramos:

```
<?xml version="1.0"?>
<model>
  <name>Hole</name>
  <version>1.0</version>
  <sdf version='1.4'>model.sdf</sdf>

  <author>
    <name>My Name</name>
    <email>me@my.email</email>
  </author>

  <description>
    My awesome robot.
  </description>
</model>
```

3. Creamos el archivo con el que definiremos el objeto. Para ello escribimos en la terminal:

```
gedit ~/.gazebo/models/hole/model.sdf
```

Nuevamente se abre un editor de texto, el cual completamos con el siguiente código:

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <model name="hole">
    <link name="black_hole">
      <pose>0 0 0 0 0 0</pose>
      <collision name="collision">
        <geometry>
          <cylinder>
            <radius>0.3</radius>
            <length>0.01</length>
          </cylinder>
        </geometry>
      </collision>
      <visual name="visual">
        <geometry>
          <cylinder>
            <radius>0.3</radius>
            <length>0.01</length>
          </cylinder>
        </geometry>
        <material>
          <script>
            <uri>file://media/materials/scripts/gazebo.material</uri>
            <name>Gazebo/Black</name>
          </script>
        </material>
      </visual>
    </link>
  </model>
</sdf>
```

Para crear cilindros de diferentes tamaños repetimos estos pasos para crear un modelo nuevo y modificamos las dos líneas de código en las que se escribe el radio (en este ejemplo, es un cilindro de radio 0.3 m). La altura, para que sea despreciable se deja en las dos líneas dedicadas a ello en 0.01 m. Otros aspectos importantes son la línea de posición (<pose>0 0 0 0 0 0</pose>), que debe estar entera a cero para que el agujero aparezca orientado correctamente, y las dos líneas para el color (<uri>file://media/materials/scripts/gazebo.material</uri> y <name>Gazebo/Black</name>), que hace que el agujero sea de color negro.

4. Finalmente, guardamos y cerramos el editor de texto y abrimos gazebo. En la pestaña "Insert" de la parte izquierda de la ventana de gazebo ya aparece el modelo que hemos creado. En nuestro caso aparece con el nombre de "Hole" (nombre proporcionado en el archivo hole/model.config). Podemos seleccionarlo y arrastrarlos hasta el lugar que queramos de la simulación.

A. 10. Obtención del código del mapeado, lanzamiento y configuración en RViz

Para realizar esta tarea tenemos que lanzar en tres terminales distintos las siguientes tareas en este orden:

1) Simulación en gazebo completa tal y como venimos haciendo
2) Ejecutable que realiza el mapeo a través del sensor láser
3) Simulación en RViz para obtener datos de los sensores y, en particular, ver el mapeado que realiza la tarea anterior

Los dos últimos son novedosos hasta ahora. Sin embargo, el Summit-XL trae archivos dedicados a ello. Para lanzar la primera tarea recordamos que hay que escribir en la terminal:

```
source /opt/ros/indigo/setup.bash
cd ~/catkin_ws
catkin_make
source devel/setup.bash
roslaunch summit_xl_sim_bringup summit_xl_complete.launch
```

La segunda tarea se ejecutará con estos comandos:

```
source /opt/ros/indigo/setup.bash
cd ~/catkin_ws
catkin_make
source devel/setup.bash
roslaunch summit_xl_navigation summit_xl_gmapping.launch
```

Para que esa tarea funcione tenemos que acudir al archivo **summit_xl_gmapping.launch**, situado en la carpeta **summit_xl_common-indigo-multirobot-devel → summit_xl_navigation → launch**, descomentar la línea 7 y comentar la línea 8 para establecer el topic del láser del que se obtiene la información, que será el *hokuyo*. Ambas líneas quedan de la siguiente manera:

```
<remap from="scan" to ="/hokuyo_base/scan"/>
<!--<remap from="scan" to="/laser_front/scan"/-->
```

Lo último, lanzar RViz, se consigue con:

```
source /opt/ros/indigo/setup.bash
cd ~/catkin_ws
catkin_make
source devel/setup.bash
roslaunch summit_xl_description summit_xl_rviz.launch
```

Con esto, tal y como vamos a explicar a continuación acerca de la configuración en RViz para ver el movimiento del robot y la construcción del mapa, ya tendríamos nuestro objetivo. Sin embargo, la tarea del mapeado requiere de un ejecutable de ROS a cuyo código fuente no tenemos acceso. Como lo que se pretenderá será modificar ese código fuente acudimos a otra alternativa: descargar el código del mapeado a través de un package que incluiremos en nuestro workspace.

Para descargar el código acudimos el enlace de [39] y nos aparecerá una ventana como la de la Figura A-0-17. En la pestaña superior izquierda, denominada “Branch” elegimos la versión hydro-devel. Sólo existe este package para dos distribuciones de ROS pero ambas son anteriores a la nuestra (indigo), por lo que son igualmente válidas. Finalmente, en la parte superior derecha hacemos click para descargar tal y como vemos en la Figura A-0-17.

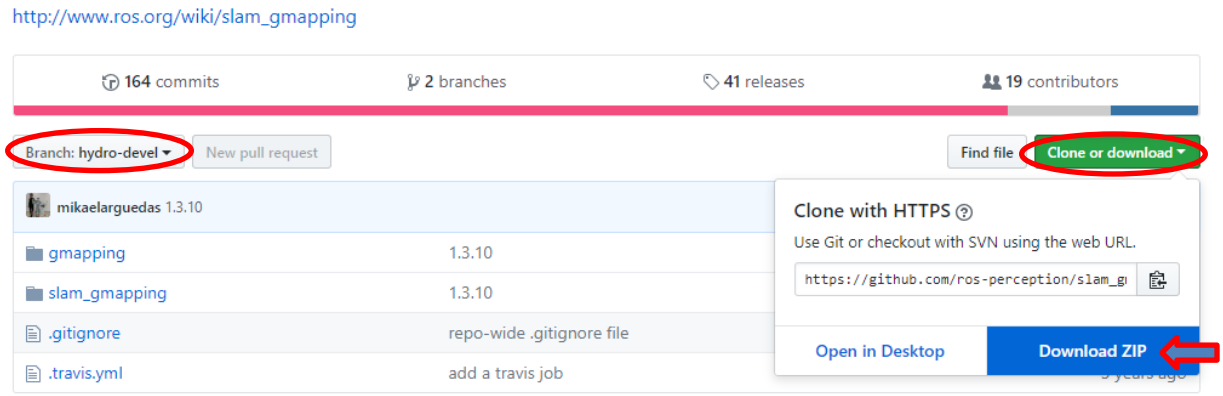

Figura A-0-17. Descarga del package slam_gmapping

Tras esto, acudimos a nuestra carpeta de descarga y descomprimimos el package, cuyo nombre será **slam_gmapping-hydro-devel**. Una vez extraído, lo copiamos dentro de la carpeta src de nuestro workspace (catkin_ws), tal y como hicimos con otros packages.

Antes de lanzar el código del mapeo con este nuevo package, tenemos que modificar algunos parámetros en el código fuente del mismo. Este código se denomina **slam_gmapping.cpp** y se encuentra en la carpeta: **Carpeta personal → catkin_ws → src → slam_gmapping-hydro-devel → gmapping → src**. Las modificaciones serán [38]:

1. Configurar los parámetros para realizar el mapeado tal y como vienen configurados en el código del archivo **summit_xl_gmapping.launch**, situado en la carpeta **summit_xl_common-indigo-multirobot-devel → summit_xl_navigation → launch**. Esto supone variar, en caso de que no coincidan, los valores de los parámetros que se encuentran entre las líneas 175 y 250 del código fuente slam_gmapping.cpp. Estas líneas quedan de la siguiente forma:

```
  // Parameters used by our GMapping wrapper
  if(!private_nh_.getParam("throttle_scans", throttle_scans_))
    throttle_scans_ = 1;
  if(!private_nh_.getParam("base_frame", base_frame_))
    base_frame_ = "base_link";
  if(!private_nh_.getParam("map_frame", map_frame_))
    map_frame_ = "map";
  if(!private_nh_.getParam("odom_frame", odom_frame_))
    odom_frame_ = "odom";

  private_nh_.param("transform_publish_period",
transform_publish_period_, 0.05);

  double tmp;
  if(!private_nh_.getParam("map_update_interval", tmp))
    tmp = 2.0; //tmp = 5.0;
  map_update_interval_.fromSec(tmp);

  // Parameters used by GMapping itself
  maxUrange_ = 16.0; //maxUrange_ = 0.0;
  maxRange_ = 0.0; // preliminary default, will be set in initMapper()
  if(!private_nh_.getParam("minimumScore", minimum_score_))
    minimum_score_ = 0;
  if(!private_nh_.getParam("sigma", sigma_))
    sigma_ = 0.05;
  if(!private_nh_.getParam("kernelSize", kernelSize_))
    kernelSize_ = 1;
  if(!private_nh_.getParam("lstep", lstep_))
    lstep_ = 0.05;
  if(!private_nh_.getParam("astep", astep_))
```

```
    astep_ = 0.05;
  if(!private_nh_.getParam("iterations", iterations_))
    iterations_ = 5;
  if(!private_nh_.getParam("lsigma", lsigma_))
    lsigma_ = 0.075;
  if(!private_nh_.getParam("ogain", ogain_))
    ogain_ = 3.0;
  if(!private_nh_.getParam("lskip", lskip_))
    lskip_ = 0;
  if(!private_nh_.getParam("srr", srr_))
    srr_ = 0.1;
  if(!private_nh_.getParam("srt", srt_))
    srt_ = 0.2;
  if(!private_nh_.getParam("str", str_))
    str_ = 0.1;
  if(!private_nh_.getParam("stt", stt_))
    stt_ = 0.2;
  if(!private_nh_.getParam("linearUpdate", linearUpdate_))
    linearUpdate_ = 0.2; //linearUpdate_ = 1.0;
  if(!private_nh_.getParam("angularUpdate", angularUpdate_))
    angularUpdate_ = 0.1; //angularUpdate_ = 0.5;
  if(!private_nh_.getParam("temporalUpdate", temporalUpdate_))
    temporalUpdate_ = 3.0; //temporalUpdate_ = -1.0;
  if(!private_nh_.getParam("resampleThreshold", resampleThreshold_))
    resampleThreshold_ = 0.5;
  if(!private_nh_.getParam("particles", particles_))
    particles_ = 100; //particles_ = 30;
  if(!private_nh_.getParam("xmin", xmin_))
    xmin_ = -50.0; //xmin_ = -100.0;
  if(!private_nh_.getParam("ymin", ymin_))
    ymin_ = -50.0; //ymin_ = -100.0;
  if(!private_nh_.getParam("xmax", xmax_))
    xmax_ = 50.0; //xmax_ = 100.0;
  if(!private_nh_.getParam("ymax", ymax_))
    ymax_ = 50.0; //ymax_ = 100.0;
  if(!private_nh_.getParam("delta", delta_))
    delta_ = 0.05;
  if(!private_nh_.getParam("occ_thresh", occ_thresh_))
    occ_thresh_ = 0.25;
  if(!private_nh_.getParam("llsamplerange", llsamplerange_))
    llsamplerange_ = 0.01;
  if(!private_nh_.getParam("llsamplestep", llsamplestep_))
    llsamplestep_ = 0.01;
  if(!private_nh_.getParam("lasamplerange", lasamplerange_))
    lasamplerange_ = 0.005;
  if(!private_nh_.getParam("lasamplestep", lasamplestep_))
    lasamplestep_ = 0.005;
```

2. Configurar el topic del láser *hokuyo*, pues será el que usaremos. Para ello, en la línea 264, cambiamos el topic "scan" por el topic "**/hokuyo_base/scan**", quedando así:

```
scan_filter_sub_ = new
message_filters::Subscriber<sensor_msgs::LaserScan>(node_,
"/hokuyo_base/scan", 5);
```

Ya estamos en disposición de lanzar el mapeado. Se lanzan en el mismo orden que antes: gazebo primero, mapeado después y RViz por último. Las tareas de simulación en gazebo y RViz se lanzan igual. La de mapeado se ejecuta con las siguientes órdenes:

```
source /opt/ros/indigo/setup.bash

cd ~/catkin_ws
```

```
catkin_make

source devel/setup.bash

rosrun gmapping slam_gmapping
```

Por último, queda configurar lo que queremos visualizar en RViz [48], que se nos abrirá en una pestaña a parte al ejecutar los comandos ya indicados (ver Figura A-0-18). En la parte izquierda de la pantalla de RViz nos aparecerá la ventana de "Displays" (marcada en rojo).

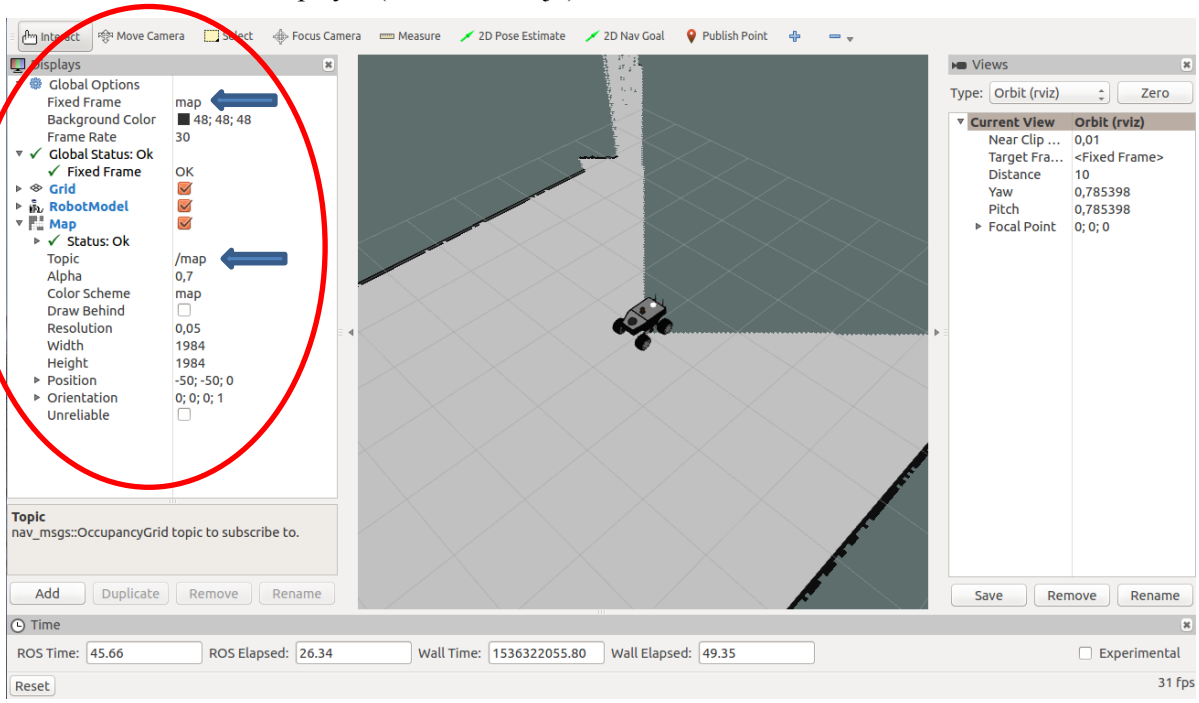

Figura A-0-18. Ventana resultante tras configurar RViz para ver el mapeado

Desplegando la opción "**Global Options**", elegimos en "**Fixed Frame**" la opción "**map**". Este paso es muy importante para poder visualizar el mapa correctamente. También añadimos el modelo del robot a la simulación para poder observar por donde se mueve. Para ello, hacemos click en la esquina superior izquierda ("Add") y seleccionamos la opción "**RobotModel**", con lo que nos debe aparecer el robot en la ventana central.

Por último y más importante, volvemos al botón de "Add" y añadimos el display del mapa: "**Map**". Lo desplegamos y en "**Topic**" seleccionamos la opción "**/map**" que según nuestro código fuente es donde se publica la información del láser para la construcción del mapa.

A partir de aquí ya se deja libertad para teleoperar el robot como sabemos y construir mapas de las zonas que queramos incluyéndole si es necesario obstáculos en la simulación de gazebo.

El resultado de ejecutar en una terminal distinta el comando **rqt_graph** tras realizar los pasos de este apartado es el de la Figura A-0-19. Se observa en la parte inferior de la imagen cómo el nodo de gazebo publica información del láser en el topic de nombre /hokuyo_base/scan. A ese topic se suscribe el nodo del mapeado, de nombre /slam_gmapping, que a su vez publica la información del mapa en el topic /map. Nótese que la cámara no se ha lanzado, por lo que no aparece en el grafo de nodos y topics.

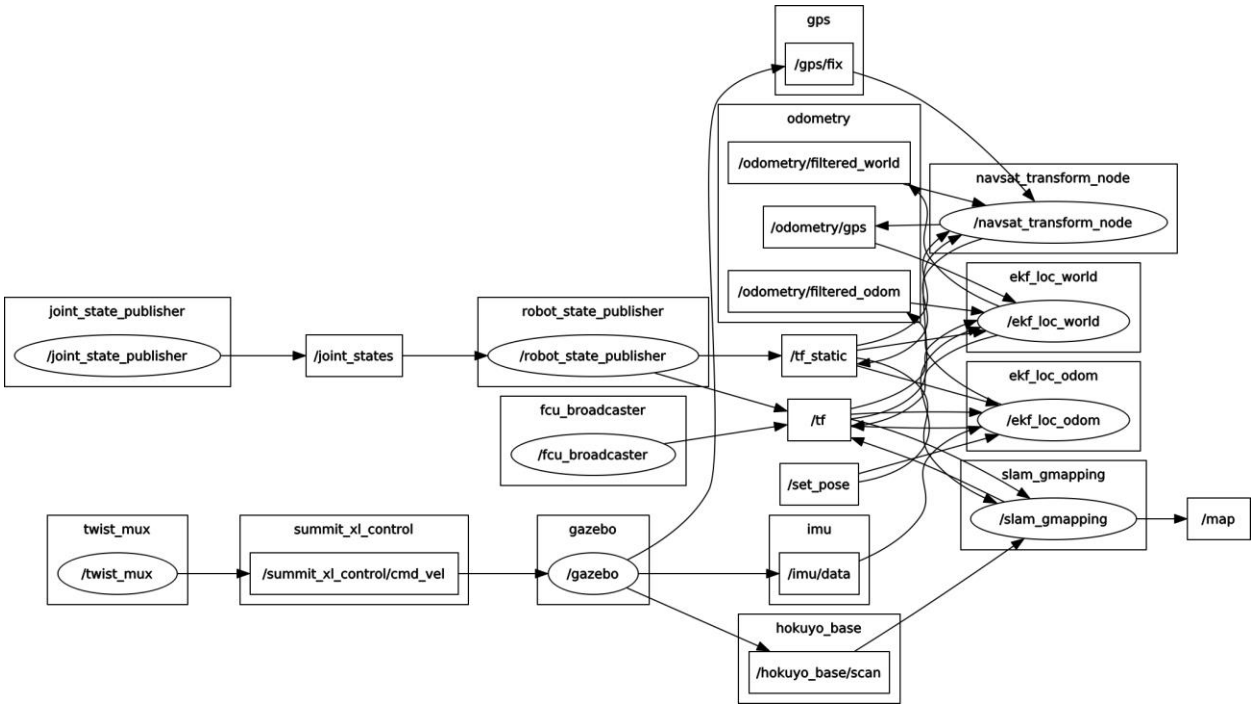

Figura A-0-19. Resultado de ejecutar rqt_graph tras lanzar un mapeo

A. 11. Lanzamiento del proyecto completo

Por último, si queremos probar la aplicación de este proyecto tendremos que lanzar, en el siguiente orden: simulación completa en gazebo, visión, mapeado y RViz.

Para ello:

1) En una terminal:

```
source /opt/ros/indigo/setup.bash
cd ~/catkin_ws
catkin_make
source devel/setup.bash
roslaunch summit_xl_sim_bringup summit_xl_complete.launch
```

2) En otra:

```
source /opt/ros/indigo/setup.bash
cd ~/catkin_ws
catkin_make
source devel/setup.bash
rosrun vision_summit_xl image_converter
```

3) En otra:

```
source /opt/ros/indigo/setup.bash
cd ~/catkin_ws
```

```
catkin_make
source devel/setup.bash
rosrun gmapping slam_gmapping
```

4) Y, por último, en otra terminal:

```
source /opt/ros/indigo/setup.bash
cd ~/catkin_ws
catkin_make
source devel/setup.bash
roslaunch summit_xl_description summit_xl_rviz.launch
```

5) Si, además, queremos teleoperarlo, abrimos otra terminal para escribir

```
rosrun key_teleop key_teleop.py key_vel:=cmd_vel
```

Si ejecutamos en este caso el comando **rqt_graph**, obtenemos el grafo de la Figura A-0-20. La zona superior de la imagen es la que nos debe de llamar la atención, pues vemos cómo primero teleoperamos a gazebo para que este mande después información de la cámara y del láser. La del láser llega directamente al nodo /slam_gmapping, y la de la cámara pasa primero por el tratamiento previo del nodo /image_converter. No es más que un buen resumen de los objetivos marcados para este trabajo.

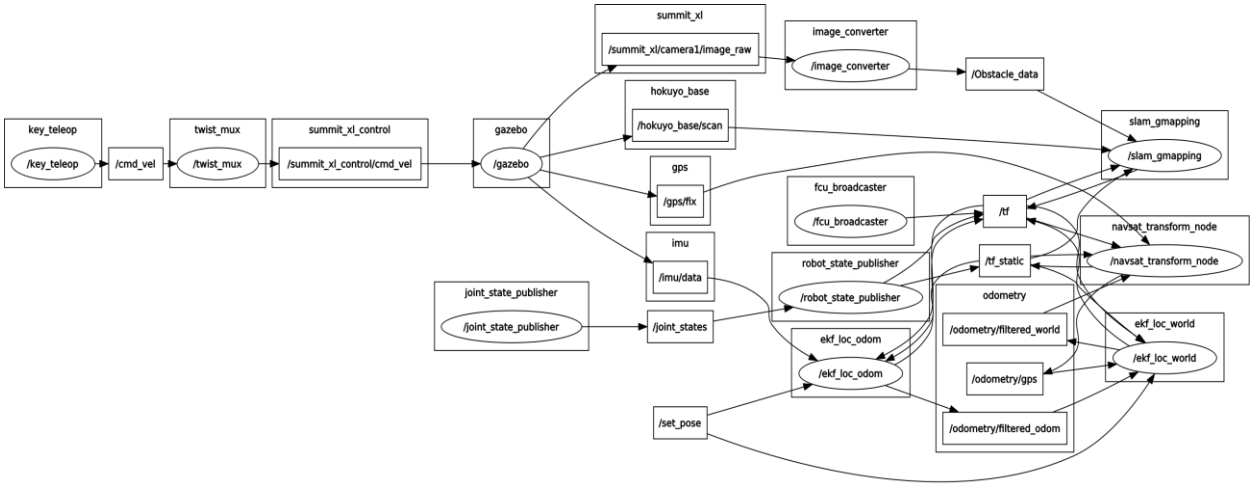

Figura A-0-20. Resultado de ejecutar rqt_graph tras lanzar el proyecto completo

ANEXO B. CÓDIGOS DESARROLLADOS

B. 1. Código de la cámara: *image_converter.cpp*

Partiendo de la plantilla que se proporciona en [33], se desarrolla el siguiente código para la cámara:

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <cmath>
#include <stdio.h>
#include "geometry_msgs/Point32.h"

static const std::string OPENCV_WINDOW = "Image window";
int thresh = 100; //Umbral para decidir por el metodo de Canny si pertenece
            //al contorno o no (gradiente por Canny mayor que tresh)
static float PI = 3.151593;
float cent_vert;
float cent_hor;
float radius_vert;

class ImageConverter
{
 ros::NodeHandle nh_;
 image_transport::ImageTransport it_;
 image_transport::Subscriber image_sub_;
 cv::Mat gray_image;
 cv::Mat bw_image;
 cv::Mat canny_output;
 cv::vector<cv::vector<cv::Point> > contours;
 cv::vector<cv::Vec4i> hierarchy;

public:
 ImageConverter()
  : it_(nh_)
 {
  //Subscrive to input video feed and publish output video feed
  image_sub_ = it_.subscribe("/summit_xl/camera1/image_raw", 1,
    &ImageConverter::imageCb, this);
  cv::namedWindow(OPENCV_WINDOW);
 }


 ~ImageConverter()
 {
  cv::destroyWindow(OPENCV_WINDOW);
 }

 void imageCb(const sensor_msgs::ImageConstPtr& msg)
 {
//------------------TRATAMIENTO DE LA IMAGEN------------------//
  cv_bridge::CvImagePtr cv_ptr;
  try
```

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

  //PASAMOS LA IMAGEN ORIGINAL RECIBIDA A ESCALA DE GRISES
  cv::cvtColor(cv_ptr->image,gray_image,CV_BGR2GRAY);

  //LA PASAMOS A BLANCO Y NEGRO. PRIMER METODO: EXPERIMENTAL; SEGUNDO METODO: OTSU. SALE MEJOR CON EL PRIMERO
  cv::threshold(gray_image,bw_image,10,255,0);
  //cv::threshold(gray_image,bw_image,0,255,CV_THRESH_BINARY|CV_THRESH_OTSU);

  //BUSCAMOS CONTORNOS
  cv::Canny(bw_image,canny_output,thresh,thresh*2,3);
  cv::findContours(canny_output,contours,hierarchy,CV_RETR_TREE,CV_CHAIN_APPROX_SIMPLE,cv::Point(0, 0));

  //CREAMOS MATRIZ BINARIA DE CONTORNOS
  cv::Mat contour_image = 255*cv::Mat::ones(canny_output.size(),CV_8UC1);
  for(size_t i=0;i<contours.size();i++)
  {
    cv::Scalar color = cv::Scalar(0,0,0);
    cv::drawContours(contour_image,contours,(int)i,color,2,8,hierarchy,0,cv::Point());
  }

  //RECTANGULO Y ELIPSE QUE MEJOR SE ADAPTE
  cv::vector<cv::RotatedRect> minRect(contours.size());
  cv::vector<cv::RotatedRect> minEllipse(contours.size());
  for(size_t i=0;i<contours.size();i++)
  {
    minRect[i] = cv::minAreaRect(contours[i]);
    if(contours[i].size()>5)
    {
      minEllipse[i] = cv::fitEllipse(contours[i]);
    }
  }
  for(size_t i=0;i<contours.size();i++)
  {
    cv::Scalar color = cv::Scalar(0,255,0);
    cv::Scalar color2 = cv::Scalar(0,0,255);
    //Contour
    //cv::drawContours(result,contours,(int)i,color,1,8,cv::vector<cv::Vec4i>(),0,cv::Point());
    //Ellipse
    cv::ellipse(cv_ptr->image,minEllipse[i],color,2,8);
    //Rotated rectangle
    /*cv::Point2f rect_points[4];
    minRect[i].points(rect_points);
    for(int j=0;j<4;j++)
      cv::line(cv_ptr->image,rect_points[j],rect_points[(j+1)%4],color,1,8);*/
  }

  //MOSTRAMOS CAMARA Y RESULTADO EN UNA VENTANA
  //Update GUI Window
  cv::imshow(OPENCV_WINDOW,cv_ptr->image);
  cv::waitKey(3);

//--------------------CALCULO DISTANCIAS--------------------//
  //TOMAMOS COMO SISTEMA DE REFERENCIA EL DE LA CAMARA
  cv::Point2f corner_rect[4];
  float pos_x_corner[4]; //Valor de la coordenada x de los 4 vertices del rectangulo
  float pos_y_corner[4]; //Valor de la coordenada y de los 4 vertices del rectangulo
```

```
  if (minRect.size()>0)
  {
   //CALCULAMOS LAS DISTANCIAS VERTICALES REALES
   minRect[0].points(corner_rect); //Se supone un solo obstaculo en la misma imagen
   float max_width;
   int row;
   for(int k=0;k<4;k++)
   {
    row = corner_rect[k].y;
    pos_y_corner[k] = 1.70221906262318E-20 * pow(row,10) -


6.07450388438866E-17 * pow(row,9) +


9.6913695011794E-14 * pow(row,8) -


9.1020034026841E-11 * pow(row,7) +


5.57242765798284E-8 * pow(row,6) -


2.323588747878887E-5 * pow(row,5) +


0.0066829751      * pow(row,4) -


1.3091703182      * pow(row,3) +


167.1876205147    * pow(row,2) -


12570.5337212198  * row      +


422689.806421953;
   }

   //CALCULAMOS LAS DISTANCIAS HORIZONTALES TEORICAS
   float max_vert = cv::max(cv::max(pos_y_corner[0],pos_y_corner[1]),cv::max(pos_y_corner[2],pos_y_corner[3]));
   float min_vert = cv::min(cv::min(pos_y_corner[0],pos_y_corner[1]),cv::min(pos_y_corner[2],pos_y_corner[3]));
   cent_vert = max_vert - (max_vert - min_vert)/2;
   radius_vert = (max_vert - min_vert)/2;
   for(int l=0;l<4;l++)
   {
    for(int m=0;m<4;m++)
    {
     if (l!=m && corner_rect[l].x==corner_rect[m].x)
                                                                                                {
      max_width = 0.1 + ((pos_y_corner[l] + pos_y_corner[m])/2);
      pos_x_corner[l] = (corner_rect[l].x - floor(bw_image.cols/2)) * (max_width / bw_image.cols);
                                                                                                }
    }
   }

   //TRUNCAMOS CON EL VALOR DEL RADIO CONOCIDO
   float max_hor = cv::max(cv::max(pos_x_corner[0],pos_x_corner[1]),cv::max(pos_x_corner[2],pos_x_corner[3]));
   float min_hor = cv::min(cv::min(pos_x_corner[0],pos_x_corner[1]),cv::min(pos_x_corner[2],pos_x_corner[3]));
   cent_hor = max_hor - (max_hor - min_hor)/2;
   for(int n=0;n<4;n++)
```

```
    {
     if(pos_x_corner[n]==min_hor)
                                                                                {

pos_x_corner[n] = cent_hor - radius_vert;
                                                                                }
                                                                                else
                                                                                {

pos_x_corner[n] = cent_hor + radius_vert;
                                                                                }
    }

    //IMPRIMIMOS POR PANTALLA Y PASAMOS RESULTADOS
    if (pos_y_corner[0]<4 && pos_y_corner[1]<4 && pos_y_corner[2]<4 && pos_y_corner[3]<4 && pos_y_corner[0]>0.7 &&
pos_y_corner[1]>0.7 && pos_y_corner[2]>0.7 && pos_y_corner[3]>0.7) //
    {//

                                                                                printf("
X vertice 1 (pixel): %f \n", corner_rect[0].x);
      printf("Y vertice 1 (pixel): %f \n", corner_rect[0].y);
      printf("X vertice 2 (pixel): %f \n", corner_rect[1].x);
      printf("Y vertice 2 (pixel): %f \n", corner_rect[1].y);
      printf("X vertice 3 (pixel): %f \n", corner_rect[2].x);
      printf("Y vertice 3 (pixel): %f \n", corner_rect[2].y);
      printf("X vertice 4 (pixel): %f \n", corner_rect[3].x);
      printf("Y vertice 4 (pixel): %f \n", corner_rect[3].y);
      printf("X vertice 1: %f \n", pos_x_corner[0]);
      printf("Y vertice 1: %f \n", pos_y_corner[0]);
      printf("X vertice 2: %f \n", pos_x_corner[1]);
      printf("Y vertice 2: %f \n", pos_y_corner[1]);
      printf("X vertice 3: %f \n", pos_x_corner[2]);
      printf("Y vertice 3: %f \n", pos_y_corner[2]);
      printf("X vertice 4: %f \n", pos_x_corner[3]);
      printf("Y vertice 4: %f \n", pos_y_corner[3]);
    }
   }
   else
   {
    cent_hor = -9999;
    cent_vert = -9999;
    radius_vert = -9999;
    for(int z=0;z<4;z++)
    {
     pos_x_corner[z] = -9999;
                                                                                pos_y_
corner[z] = -9999;
    }
   }
  }
};

int main(int argc, char** argv)
{
 ros::init(argc, argv, "image_converter");
 ImageConverter ic;
 ros::NodeHandle n;
 ros::Publisher pub = n.advertise<geometry_msgs::Point32>("Obstacle_data", 1000);
 ros::Rate loop_rate(10);
 geometry_msgs::Point32 msg;
 while (ros::ok())
 {
  msg.x = cent_hor; //Distancia en el eje x (horizontal) del centro del obstaculo
  msg.y = cent_vert; //Distancia en el eje y (vertical) del centro del obstaculo
```

```
    msg.z = radius_vert; //Radio del obstaculo
    pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

B. 2. Código del mapa: *slam_gmapping.cpp*

Se muestra a continuación el código del mapa, el cual es prácticamente igual que el obtenido de [39], con las modificaciones comentadas a lo largo de esta memoria.

```
 /*
 * slam_gmapping
 * Copyright (c) 2008, Willow Garage, Inc.
 *
 * THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE
 * COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY
 * COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS
 * AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.
 *
 * BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO
 * BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS
 * CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND
 * CONDITIONS.
 *
 */

/* Author: Brian Gerkey */
/* Modified by: Charles DuHadway */


/**

@mainpage slam_gmapping

@htmlinclude manifest.html

@b slam_gmapping is a wrapper around the GMapping SLAM library. It reads laser
scans and odometry and computes a map. This map can be
written to a file using e.g.

  "rosrun map_server map_saver static_map:=dynamic_map"

<hr>

@section topic ROS topics

Subscribes to (name/type):
- @b "scan"/<a href="../../sensor_msgs/html/classstd__msgs_1_1LaserScan.html">sensor_msgs/LaserScan</a> : data from a laser
range scanner
- @b "/tf": odometry from the robot


Publishes to (name/type):
- @b "/tf"/tf/tfMessage: position relative to the map


@section services
 - @b "~dynamic_map" : returns the map
```

@section parameters ROS parameters

Reads the following parameters from the parameter server

Parameters used by our GMapping wrapper:

- @b "~throttle_scans": @b [int] throw away every nth laser scan
- @b "~base_frame": @b [string] the tf frame_id to use for the robot base pose
- @b "~map_frame": @b [string] the tf frame_id where the robot pose on the map is published
- @b "~odom_frame": @b [string] the tf frame_id from which odometry is read
- @b "~map_update_interval": @b [double] time in seconds between two recalculations of the map

Parameters used by GMapping itself:

Laser Parameters:
- @b "~/maxRange" @b [double] maximum range of the laser scans. Rays beyond this range get discarded completely. (default: maximum laser range minus 1 cm, as received in the the first LaserScan message)
- @b "~/maxUrange" @b [double] maximum range of the laser scanner that is used for map building (default: same as maxRange)

- @b "~/sigma" @b [double] standard deviation for the scan matching process (cell)

- @b "~/kernelSize" @b [int] search window for the scan matching process

- @b "~/lstep" @b [double] initial search step for scan matching (linear)

- @b "~/astep" @b [double] initial search step for scan matching (angular)

*- @b "~/iterations" @b [int] number of refinement steps in the scan matching. The final "precision" for the match is lstep*2^(-iterations) or astep*2^(-iterations), respectively.*

- @b "~/lsigma" @b [double] standard deviation for the scan matching process (single laser beam)

- @b "~/ogain" @b [double] gain for smoothing the likelihood

- @b "~/lskip" @b [int] take only every (n+1)th laser ray for computing a match (0 = take all rays)

- @b "~/minimumScore" @b [double] minimum score for considering the outcome of the scanmatching good. Can avoid 'jumping' pose estimates in large open spaces when using laser scanners with limited range (e.g. 5m). (0 = default. Scores go up to 600+, try 50 for example when experiencing 'jumping' estimate issues)

Motion Model Parameters (all standard deviations of a gaussian noise model)

- @b "~/srr" @b [double] linear noise component (x and y)

- @b "~/stt" @b [double] angular noise component (theta)

- @b "~/srt" @b [double] linear -> angular noise component

- @b "~/str" @b [double] angular -> linear noise component

Others:

- @b "~/linearUpdate" @b [double] the robot only processes new measurements if the robot has moved at least this many meters

- @b "~/angularUpdate" @b [double] the robot only processes new measurements if the robot has turned at least this many rads

- @b "~/resampleThreshold" @b [double] threshold at which the particles get resampled. Higher means more frequent resampling.

- @b "~/particles" @b [int] (fixed) number of particles. Each particle represents a possible trajectory that the robot has traveled

Likelihood sampling (used in scan matching)

- @b "~/llsamplerange" @b [double] linear range

- @b "~/lasamplerange" @b [double] linear step size

```
- @b "~/llsamplestep" @b [double] linear range

- @b "~/lasamplestep" @b [double] angular step size


Initial map dimensions and resolution:

- @b "~/xmin" @b [double] minimum x position in the map [m]

- @b "~/ymin" @b [double] minimum y position in the map [m]

- @b "~/xmax" @b [double] maximum x position in the map [m]
- @b "~/ymax" @b [double] maximum y position in the map [m]
- @b "~/delta" @b [double] size of one pixel [m]

*/


#include <math.h>
#include "slam_gmapping.h"

#include <iostream>

#include <time.h>

#include "ros/ros.h"
#include "ros/console.h"
#include "nav_msgs/MapMetaData.h"

#include "gmapping/sensor/sensor_range/rangesensor.h"
#include "gmapping/sensor/sensor_odometry/odometrysensor.h"

#include <rosbag/bag.h>
#include <rosbag/view.h>
#include <boost/foreach.hpp>

#define foreach BOOST_FOREACH

void clbk(const geometry_msgs::Point32 msg);

double pos_x;
double pos_y;
double pos_th;
#define ON 10
float camera_x[ON];
float camera_y[ON];
float camera_z[ON];
int count = 0;

// compute linear index for given map coords
#define MAP_IDX(sx, i, j) ((sx) * (j) + (i))

SlamGMapping::SlamGMapping():
  map_to_odom_(tf::Transform(tf::createQuaternionFromRPY( 0, 0, 0 ), tf::Point(0, 0, 0 ))),
  laser_count_(0), private_nh_("~"), scan_filter_sub_(NULL), scan_filter_(NULL), transform_thread_(NULL)
{
  seed_ = time(NULL);
  init();
}

SlamGMapping::SlamGMapping(ros::NodeHandle& nh, ros::NodeHandle& pnh):
  map_to_odom_(tf::Transform(tf::createQuaternionFromRPY( 0, 0, 0 ), tf::Point(0, 0, 0 ))),
  laser_count_(0),node_(nh), private_nh_(pnh), scan_filter_sub_(NULL), scan_filter_(NULL), transform_thread_(NULL)
{
  seed_ = time(NULL);
```

```
  init();
}

SlamGMapping::SlamGMapping(long unsigned int seed, long unsigned int max_duration_buffer):
  map_to_odom_(tf::Transform(tf::createQuaternionFromRPY( 0, 0, 0 ), tf::Point(0, 0, 0 ))),
  laser_count_(0), private_nh_("~"), scan_filter_sub_(NULL), scan_filter_(NULL), transform_thread_(NULL),
  seed_(seed), tf_(ros::Duration(max_duration_buffer))
{
  init();
}


//-----------------------------------------------------------------------------------------------------------------//
void SlamGMapping::init()
{
  // log4cxx::Logger::getLogger(ROSCONSOLE_DEFAULT_NAME)->setLevel(ros::console::g_level_lookup[ros::console::levels::Debug]);

  // The library is pretty chatty
  //gsp_ = new GMapping::GridSlamProcessor(std::cerr);
  gsp_ = new GMapping::GridSlamProcessor();
  ROS_ASSERT(gsp_);

  tfB_ = new tf::TransformBroadcaster();
  ROS_ASSERT(tfB_);

  gsp_laser_ = NULL;
  gsp_odom_ = NULL;

  got_first_scan_ = false;
  got_map_ = false;


  // Parameters used by our GMapping wrapper
  if(!private_nh_.getParam("throttle_scans", throttle_scans_))
    throttle_scans_ = 1;
  if(!private_nh_.getParam("base_frame", base_frame_))
    base_frame_ = "base_link";
  if(!private_nh_.getParam("map_frame", map_frame_))
    map_frame_ = "map";
  if(!private_nh_.getParam("odom_frame", odom_frame_))
    odom_frame_ = "odom";

  private_nh_.param("transform_publish_period", transform_publish_period_, 0.05);

  double tmp;
  if(!private_nh_.getParam("map_update_interval", tmp))
    tmp = 2.0; //tmp = 5.0;
  map_update_interval_.fromSec(tmp);

  // Parameters used by GMapping itself
  maxUrange_ = 16.0; //maxUrange_ = 0.0;
  maxRange_ = 0.0; // preliminary default, will be set in initMapper()
  if(!private_nh_.getParam("minimumScore", minimum_score_))
    minimum_score_ = 0;
  if(!private_nh_.getParam("sigma", sigma_))
    sigma_ = 0.05;
  if(!private_nh_.getParam("kernelSize", kernelSize_))
    kernelSize_ = 1;
  if(!private_nh_.getParam("lstep", lstep_))
    lstep_ = 0.05;
  if(!private_nh_.getParam("astep", astep_))
    astep_ = 0.05;
  if(!private_nh_.getParam("iterations", iterations_))
    iterations_ = 5;
```

```
  if(!private_nh_.getParam("lsigma", lsigma_))
    lsigma_ = 0.075;
  if(!private_nh_.getParam("ogain", ogain_))
    ogain_ = 3.0;
  if(!private_nh_.getParam("lskip", lskip_))
    lskip_ = 0;
  if(!private_nh_.getParam("srr", srr_))
    srr_ = 0.1;
  if(!private_nh_.getParam("srt", srt_))
    srt_ = 0.2;
  if(!private_nh_.getParam("str", str_))
    str_ = 0.1;
  if(!private_nh_.getParam("stt", stt_))
    stt_ = 0.2;
  if(!private_nh_.getParam("linearUpdate", linearUpdate_))
    linearUpdate_ = 0.2; //linearUpdate_ = 1.0;
  if(!private_nh_.getParam("angularUpdate", angularUpdate_))
    angularUpdate_ = 0.1; //angularUpdate_ = 0.5;
  if(!private_nh_.getParam("temporalUpdate", temporalUpdate_))
    temporalUpdate_ = 3.0; //temporalUpdate_ = -1.0;
  if(!private_nh_.getParam("resampleThreshold", resampleThreshold_))
    resampleThreshold_ = 0.5;
  if(!private_nh_.getParam("particles", particles_))
    particles_ = 100; //particles_ = 30;
  if(!private_nh_.getParam("xmin", xmin_))
    xmin_ = -50.0; //xmin_ = -100.0;
  if(!private_nh_.getParam("ymin", ymin_))
    ymin_ = -50.0; //ymin_ = -100.0;
  if(!private_nh_.getParam("xmax", xmax_))
    xmax_ = 50.0; //xmax_ = 100.0;
  if(!private_nh_.getParam("ymax", ymax_))
    ymax_ = 50.0; //ymax_ = 100.0;
  if(!private_nh_.getParam("delta", delta_))
    delta_ = 0.05;
  if(!private_nh_.getParam("occ_thresh", occ_thresh_))
    occ_thresh_ = 0.25;
  if(!private_nh_.getParam("llsamplerange", llsamplerange_))
    llsamplerange_ = 0.01;
  if(!private_nh_.getParam("llsamplestep", llsamplestep_))
    llsamplestep_ = 0.01;
  if(!private_nh_.getParam("lasamplerange", lasamplerange_))
    lasamplerange_ = 0.005;
  if(!private_nh_.getParam("lasamplestep", lasamplestep_))
    lasamplestep_ = 0.005;

  if(!private_nh_.getParam("tf_delay", tf_delay_))
    tf_delay_ = transform_publish_period_;

}

//-----------------------------------------------------------------------------------------------------------------//
void SlamGMapping::startLiveSlam()
{
  entropy_publisher_ = private_nh_.advertise<std_msgs::Float64>("entropy", 1, true);
  sst_ = node_.advertise<nav_msgs::OccupancyGrid>("map", 1, true);
  sstm_ = node_.advertise<nav_msgs::MapMetaData>("map_metadata", 1, true);
  ss_ = node_.advertiseService("dynamic_map", &SlamGMapping::mapCallback, this);
  scan_filter_sub_ = new message_filters::Subscriber<sensor_msgs::LaserScan>(node_, "/hokuyo_base/scan", 5); //scan_filter_sub_ =
new message_filters::Subscriber<sensor_msgs::LaserScan>(node_, "scan", 5);
  scan_filter_ = new tf::MessageFilter<sensor_msgs::LaserScan>(*scan_filter_sub_, tf_, odom_frame_, 5);
  scan_filter_->registerCallback(boost::bind(&SlamGMapping::laserCallback, this, _1));
  ros::Subscriber sub = node_.subscribe("Obstacle_data", 1000, clbk); //
  transform_thread_ = new boost::thread(boost::bind(&SlamGMapping::publishLoop, this, transform_publish_period_));
```

```
  ros::spin(); //
}

//-----------------------------------------------------------------------------------------------------------------------//
void SlamGMapping::startReplay(const std::string & bag_fname, std::string scan_topic)
{
 double transform_publish_period;
 ros::NodeHandle private_nh_("~");
 entropy_publisher_ = private_nh_.advertise<std_msgs::Float64>("entropy", 1, true);
 sst_ = node_.advertise<nav_msgs::OccupancyGrid>("map", 1, true);
 sstm_ = node_.advertise<nav_msgs::MapMetaData>("map_metadata", 1, true);
 ss_ = node_.advertiseService("dynamic_map", &SlamGMapping::mapCallback, this);

 rosbag::Bag bag;
 bag.open(bag_fname, rosbag::bagmode::Read);

 std::vector<std::string> topics;
 topics.push_back(std::string("/tf"));
 topics.push_back(scan_topic);
 rosbag::View viewall(bag, rosbag::TopicQuery(topics));

 // Store up to 5 messages and there error message (if they cannot be processed right away)
 std::queue<std::pair<sensor_msgs::LaserScan::ConstPtr, std::string> > s_queue;
 foreach(rosbag::MessageInstance const m, viewall)
 {
  tf::tfMessage::ConstPtr cur_tf = m.instantiate<tf::tfMessage>();
  if (cur_tf != NULL) {
   for (size_t i = 0; i < cur_tf->transforms.size(); ++i)
        {
    geometry_msgs::TransformStamped transformStamped;
    tf::StampedTransform stampedTf;
    transformStamped = cur_tf->transforms[i];
    tf::transformStampedMsgToTF(transformStamped, stampedTf);
    tf_.setTransform(stampedTf);
   }
  }

  sensor_msgs::LaserScan::ConstPtr s = m.instantiate<sensor_msgs::LaserScan>();
  if (s != NULL) {
   if (!(ros::Time(s->header.stamp)).is_zero())
   {
    s_queue.push(std::make_pair(s, ""));
   }
   // Just like in live processing, only process the latest 5 scans
   if (s_queue.size() > 5) {
    ROS_WARN_STREAM("Dropping old scan: " << s_queue.front().second);
    s_queue.pop();
   }
   // ignoring un-timestamped tf data
  }

  // Only process a scan if it has tf data
  while (!s_queue.empty())
  {
   try
   {
    tf::StampedTransform t;
    tf_.lookupTransform(s_queue.front().first->header.frame_id, odom_frame_, s_queue.front().first->header.stamp, t);
    this->laserCallback(s_queue.front().first);
    s_queue.pop();
   }
   // If tf does not have the data yet
   catch(tf2::TransformException& e)
   {
```

```
      // Store the error to display it if we cannot process the data after some time
      s_queue.front().second = std::string(e.what());
      break;
     }
    }
   }

  bag.close();
}

//-------------------------------------------------------------------------------------------------------------------//
void SlamGMapping::publishLoop(double transform_publish_period)
{
  if(transform_publish_period == 0)
    return;

  ros::Rate r(1.0 / transform_publish_period);
  while(ros::ok()){
    publishTransform();
    r.sleep();
  }
}

//-------------------------------------------------------------------------------------------------------------------//
SlamGMapping::~SlamGMapping()
{
  if(transform_thread_){
    transform_thread_->join();
    delete transform_thread_;
  }

  delete gsp_;
  if(gsp_laser_)
    delete gsp_laser_;
  if(gsp_odom_)
    delete gsp_odom_;
  if (scan_filter_)
    delete scan_filter_;
  if (scan_filter_sub_)
    delete scan_filter_sub_;
}

//-------------------------------------------------------------------------------------------------------------------//
bool SlamGMapping::getOdomPose(GMapping::OrientedPoint& gmap_pose, const ros::Time& t)
{
  // Get the pose of the centered laser at the right time
  centered_laser_pose_.stamp_ = t;
  // Get the laser's pose that is centered
  tf::Stamped<tf::Transform> odom_pose;
  try
  {
    tf_.transformPose(odom_frame_, centered_laser_pose_, odom_pose);
  }
  catch(tf::TransformException e)
  {
    ROS_WARN("Failed to compute odom pose, skipping scan (%s)", e.what());
    return false;
  }
  double yaw = tf::getYaw(odom_pose.getRotation());

  gmap_pose = GMapping::OrientedPoint(odom_pose.getOrigin().x(),
                                      odom_pose.getOrigin().y(),
                                      yaw);
  return true;
```

```
}

//-----------------------------------------------------------------------------------------------------------//
bool SlamGMapping::initMapper(const sensor_msgs::LaserScan& scan)
{
 laser_frame_ = scan.header.frame_id;
 // Get the laser's pose, relative to base.
 tf::Stamped<tf::Pose> ident;
 tf::Stamped<tf::Transform> laser_pose;
 ident.setIdentity();
 ident.frame_id_ = laser_frame_;
 ident.stamp_ = scan.header.stamp;
 try
 {
  tf_.transformPose(base_frame_, ident, laser_pose);
 }
 catch(tf::TransformException e)
 {
  ROS_WARN("Failed to compute laser pose, aborting initialization (%s)",
        e.what());
  return false;
 }

 // create a point 1m above the laser position and transform it into the laser-frame
 tf::Vector3 v;
 v.setValue(0, 0, 1 + laser_pose.getOrigin().z());
 tf::Stamped<tf::Vector3> up(v, scan.header.stamp,
                             base_frame_);
 try
 {
  tf_.transformPoint(laser_frame_, up, up);
  ROS_DEBUG("Z-Axis in sensor frame: %.3f", up.z());
 }
 catch(tf::TransformException& e)
 {
  ROS_WARN("Unable to determine orientation of laser: %s",
        e.what());
  return false;
 }

 // gmapping doesnt take roll or pitch into account. So check for correct sensor alignment.
 if (fabs(fabs(up.z()) - 1) > 0.001)
 {
  ROS_WARN("Laser has to be mounted planar! Z-coordinate has to be 1 or -1, but gave: %.5f",
          up.z());
  return false;
 }

 gsp_laser_beam_count_ = scan.ranges.size();

 double angle_center = (scan.angle_min + scan.angle_max)/2;

 if (up.z() > 0)
 {
  do_reverse_range_ = scan.angle_min > scan.angle_max;
  centered_laser_pose_ = tf::Stamped<tf::Pose>(tf::Transform(tf::createQuaternionFromRPY(0,0,angle_center),
                                       tf::Vector3(0,0,0)), ros::Time::now(), laser_frame_);
  ROS_INFO("Laser is mounted upwards.");
 }
 else
 {
  do_reverse_range_ = scan.angle_min < scan.angle_max;
  centered_laser_pose_ = tf::Stamped<tf::Pose>(tf::Transform(tf::createQuaternionFromRPY(M_PI,0,-angle_center),
                                       tf::Vector3(0,0,0)), ros::Time::now(), laser_frame_);
```

```
  ROS_INFO("Laser is mounted upside down.");
}

// Compute the angles of the laser from -x to x, basically symmetric and in increasing order
laser_angles_.resize(scan.ranges.size());
// Make sure angles are started so that they are centered
double theta = - std::fabs(scan.angle_min - scan.angle_max)/2;
for(unsigned int i=0; i<scan.ranges.size(); ++i)
{
  laser_angles_[i]=theta;
  theta += std::fabs(scan.angle_increment);
}

ROS_DEBUG("Laser angles in laser-frame: min: %.3f max: %.3f inc: %.3f", scan.angle_min, scan.angle_max,
       scan.angle_increment);
ROS_DEBUG("Laser angles in top-down centered laser-frame: min: %.3f max: %.3f inc: %.3f", laser_angles_.front(),
       laser_angles_.back(), std::fabs(scan.angle_increment));

GMapping::OrientedPoint gmap_pose(0, 0, 0);

// setting maxRange and maxUrange here so we can set a reasonable default
ros::NodeHandle private_nh_("~");
if(!private_nh_.getParam("maxRange", maxRange_))
  maxRange_ = scan.range_max - 0.01;
if(!private_nh_.getParam("maxUrange", maxUrange_))
  maxUrange_ = maxRange_;

// The laser must be called "FLASER".
// We pass in the absolute value of the computed angle increment, on the
// assumption that GMapping requires a positive angle increment.  If the
// actual increment is negative, we'll swap the order of ranges before
// feeding each scan to GMapping.
gsp_laser_ = new GMapping::RangeSensor("FLASER",
                         gsp_laser_beam_count_,
                         fabs(scan.angle_increment),
                         gmap_pose,
                         0.0,
                         maxRange_);
ROS_ASSERT(gsp_laser_);

GMapping::SensorMap smap;
smap.insert(make_pair(gsp_laser_->getName(), gsp_laser_));
gsp_->setSensorMap(smap);

gsp_odom_ = new GMapping::OdometrySensor(odom_frame_);
ROS_ASSERT(gsp_odom_);


/// @todo Expose setting an initial pose
GMapping::OrientedPoint initialPose;
if(!getOdomPose(initialPose, scan.header.stamp))
{
  ROS_WARN("Unable to determine inital pose of laser! Starting point will be set to zero.");
  initialPose = GMapping::OrientedPoint(0.0, 0.0, 0.0);
}

gsp_->setMatchingParameters(maxUrange_, maxRange_, sigma_,
                  kernelSize_, lstep_, astep_, iterations_,
                  lsigma_, ogain_, lskip_);

gsp_->setMotionModelParameters(srr_, srt_, str_, stt_);
gsp_->setUpdateDistances(linearUpdate_, angularUpdate_, resampleThreshold_);
gsp_->setUpdatePeriod(temporalUpdate_);
gsp_->setgenerateMap(false);
```

```
  gsp_->GridSlamProcessor::init(particles_, xmin_, ymin_, xmax_, ymax_,
                                delta_, initialPose);
  gsp_->setllsamplerange(llsamplerange_);
  gsp_->setllsamplestep(llsamplestep_);
  /// @todo Check these calls; in the gmapping gui, they use
  /// llsamplestep and llsamplerange intead of lasamplestep and
  /// lasamplerange.  It was probably a typo, but who knows.
  gsp_->setlasamplerange(lasamplerange_);
  gsp_->setlasamplestep(lasamplestep_);
  gsp_->setminimumScore(minimum_score_);

  // Call the sampling function once to set the seed.
  GMapping::sampleGaussian(1,seed_);

  ROS_INFO("Initialization complete");

  return true;
}

//-----------------------------------------------------------------------------------------------------------------//
bool SlamGMapping::addScan(const sensor_msgs::LaserScan& scan, GMapping::OrientedPoint& gmap_pose)
{
  if(!getOdomPose(gmap_pose, scan.header.stamp))
     return false;

  if(scan.ranges.size() != gsp_laser_beam_count_)
    return false;

  // GMapping wants an array of doubles...
  double* ranges_double = new double[scan.ranges.size()];
  // If the angle increment is negative, we have to invert the order of the readings.
  if (do_reverse_range_)
  {
    ROS_DEBUG("Inverting scan");
    int num_ranges = scan.ranges.size();
    for(int i=0; i < num_ranges; i++)
    {
      // Must filter out short readings, because the mapper won't
      if(scan.ranges[num_ranges - i - 1] < scan.range_min)
        ranges_double[i] = (double)scan.range_max;
      else
        ranges_double[i] = (double)scan.ranges[num_ranges - i - 1];
    }
  } else
  {
    for(unsigned int i=0; i < scan.ranges.size(); i++)
    {
      // Must filter out short readings, because the mapper won't
      if(scan.ranges[i] < scan.range_min)
        ranges_double[i] = (double)scan.range_max;
      else
        ranges_double[i] = (double)scan.ranges[i];
    }
  }

  GMapping::RangeReading reading(scan.ranges.size(),
                                 ranges_double,
                                 gsp_laser_,
                                 scan.header.stamp.toSec());

  // ...but it deep copies them in RangeReading constructor, so we don't
  // need to keep our array around.
  delete[] ranges_double;
```

```
  reading.setPose(gmap_pose);

  /*
  ROS_DEBUG("scanpose (%.3f): %.3f %.3f %.3f\n",
         scan.header.stamp.toSec(),
         gmap_pose.x,
         gmap_pose.y,
         gmap_pose.theta);
         */
  ROS_DEBUG("processing scan");

  return gsp_->processScan(reading);
}

//-------------------------------------------------------------------------------------------------------------------//
void SlamGMapping::laserCallback(const sensor_msgs::LaserScan::ConstPtr& scan)
{
  laser_count_++;
  if ((laser_count_ % throttle_scans_) != 0)
    return;

  static ros::Time last_map_update(0,0);

  // We can't initialize the mapper until we've got the first scan
  if(!got_first_scan_)
  {
    if(!initMapper(*scan))
      return;
    got_first_scan_ = true;
  }

  GMapping::OrientedPoint odom_pose;

  if(addScan(*scan, odom_pose))
  {
    ROS_DEBUG("scan processed");

    GMapping::OrientedPoint mpose = gsp_->getParticles()[gsp_->getBestParticleIndex()].pose;
    ROS_DEBUG("new best pose: %.3f %.3f %.3f", mpose.x, mpose.y, mpose.theta);
    ROS_DEBUG("odom pose: %.3f %.3f %.3f", odom_pose.x, odom_pose.y, odom_pose.theta);
    ROS_DEBUG("correction: %.3f %.3f %.3f", mpose.x - odom_pose.x, mpose.y - odom_pose.y, mpose.theta - odom_pose.theta);

    tf::Transform laser_to_map = tf::Transform(tf::createQuaternionFromRPY(0, 0, mpose.theta), tf::Vector3(mpose.x, mpose.y,
0.0)).inverse();
    tf::Transform odom_to_laser = tf::Transform(tf::createQuaternionFromRPY(0, 0, odom_pose.theta), tf::Vector3(odom_pose.x,
odom_pose.y, 0.0));

    map_to_odom_mutex_.lock();
    map_to_odom_ = (odom_to_laser * laser_to_map).inverse();
    map_to_odom_mutex_.unlock();

    if(!got_map_ || (scan->header.stamp - last_map_update) > map_update_interval_)
    {
      updateMap(*scan);
      last_map_update = scan->header.stamp;
      ROS_DEBUG("Updated the map");
    }
  } else
    ROS_DEBUG("cannot process scan");
}

//-------------------------------------------------------------------------------------------------------------------//
double SlamGMapping::computePoseEntropy()
{
```

```
  double weight_total=0.0;
  for(std::vector<GMapping::GridSlamProcessor::Particle>::const_iterator it = gsp_->getParticles().begin();
     it != gsp_->getParticles().end();
     ++it)
  {
    weight_total += it->weight;
  }
  double entropy = 0.0;
  for(std::vector<GMapping::GridSlamProcessor::Particle>::const_iterator it = gsp_->getParticles().begin();
     it != gsp_->getParticles().end();
     ++it)
  {
    if(it->weight/weight_total > 0.0)
     entropy += it->weight/weight_total * log(it->weight/weight_total);
  }
  return -entropy;
}

//-------------------------------------------------------------------------------------------------------------//
void SlamGMapping::updateMap(const sensor_msgs::LaserScan& scan)
{
  float map_x[ON];
  float map_y[ON];
  float map_z[ON];
  for(int dd=0; dd<ON; dd++)
  {
    map_x[dd] = -9999;
    map_y[dd] = -9999;
    map_z[dd] = -9999;
  }

  ROS_DEBUG("Update map");
  boost::mutex::scoped_lock map_lock (map_mutex_);
  GMapping::ScanMatcher matcher;

  matcher.setLaserParameters(scan.ranges.size(), &(laser_angles_[0]),
                             gsp_laser_->getPose());

  matcher.setlaserMaxRange(maxRange_);
  matcher.setusableRange(maxUrange_);
  matcher.setgenerateMap(true);

  GMapping::GridSlamProcessor::Particle best =
        gsp_->getParticles()[gsp_->getBestParticleIndex()];
  std_msgs::Float64 entropy;
  entropy.data = computePoseEntropy();
  if(entropy.data > 0.0)
    entropy_publisher_.publish(entropy);

  if(!got_map_) {
    map_.map.info.resolution = delta_;
    map_.map.info.origin.position.x = 0.0;
    map_.map.info.origin.position.y = 0.0;
    map_.map.info.origin.position.z = 0.0;
    map_.map.info.origin.orientation.x = 0.0;
    map_.map.info.origin.orientation.y = 0.0;
    map_.map.info.origin.orientation.z = 0.0;
    map_.map.info.origin.orientation.w = 1.0;
  }

  GMapping::Point center;
  center.x=(xmin_ + xmax_) / 2.0;
  center.y=(ymin_ + ymax_) / 2.0;
```

```
GMapping::ScanMatcherMap smap(center, xmin_, ymin_, xmax_, ymax_,
                              delta_);

ROS_DEBUG("Trajectory tree:");
int cnt = 0;
for(GMapping::GridSlamProcessor::TNode* n = best.node;
   n;
   n = n->parent)
{
 ROS_DEBUG("  %.3f %.3f %.3f",
        n->pose.x,
        n->pose.y,
        n->pose.theta);
 if (cnt==0)
 {
  pos_x = n->pose.x;
  pos_y = n->pose.y;
  pos_th = n->pose.theta;
 }
 if(!n->reading)
 {
  ROS_DEBUG("Reading is NULL");
  continue;
 }
 matcher.invalidateActiveArea();
 matcher.computeActiveArea(smap, n->pose, &((*n->reading)[0]));
 matcher.registerScan(smap, n->pose, &((*n->reading)[0]));

 cnt = cnt + 1;
}

ROS_INFO("AAAA  %.3f %.3f %.3f", pos_x, pos_y, pos_th);

// the map may have expanded, so resize ros message as well
if(map_.map.info.width != (unsigned int) smap.getMapSizeX() || map_.map.info.height != (unsigned int) smap.getMapSizeY()) {

 // NOTE: The results of ScanMatcherMap::getSize() are different from the parameters given to the constructor
 //       so we must obtain the bounding box in a different way
 GMapping::Point wmin = smap.map2world(GMapping::IntPoint(0, 0));
 GMapping::Point wmax = smap.map2world(GMapping::IntPoint(smap.getMapSizeX(), smap.getMapSizeY()));
 xmin_ = wmin.x; ymin_ = wmin.y;
 xmax_ = wmax.x; ymax_ = wmax.y;

 ROS_INFO("map size is now %dx%d pixels (%f,%f)-(%f, %f)", smap.getMapSizeX(), smap.getMapSizeY(),
        xmin_, ymin_, xmax_, ymax_);

 map_.map.info.width = smap.getMapSizeX();
 map_.map.info.height = smap.getMapSizeY();
 map_.map.info.origin.position.x = xmin_;
 map_.map.info.origin.position.y = ymin_;
 map_.map.data.resize(map_.map.info.width * map_.map.info.height);

 ROS_DEBUG("map origin: (%f, %f)", map_.map.info.origin.position.x, map_.map.info.origin.position.y);
}

GMapping::Point center2;
center2.x = (smap.getMapSizeX() + 16) / 2.0;
center2.y = (smap.getMapSizeY() + 16) / 2.0;
for(int x=0; x < smap.getMapSizeX(); x++)
{
 for(int y=0; y < smap.getMapSizeY(); y++)
 {
  /// @todo Sort out the unknown vs. free vs. obstacle thresholding
  GMapping::IntPoint p(x, y);
```

```
    double occ=smap.cell(p);
    assert(occ <= 1.0);
    for(int ee=0; ee<ON; ee++)
    {
     map_x[ee] = trunc(camera_x[ee]/delta_);
     map_y[ee] = trunc(camera_y[ee]/delta_);
     map_z[ee] = ceil(camera_z[ee]/delta_);
     if(x==(center2.x+map_x[ee]) && y==(center2.y+map_y[ee]) && camera_z[ee]>0.1)
     {
      for(int k=(center2.x+map_x[ee]-map_z[ee]); k<(center2.x+map_x[ee]+map_z[ee]); k++)
      {
       for (int l=(center2.y+map_y[ee]-map_z[ee]); l<(center2.y+map_y[ee]+map_z[ee]); l++)
       {
        map_.map.data[MAP_IDX(map_.map.info.width, k, l)] = 100;
                                                                                                                    }
       }
      }
      else
      {
       if(occ < 0)
        map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = -1;
       else if(occ > occ_thresh_)
       {
        //map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = (int)round(occ*100.0);
        map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = 100;
       }
       else
        map_.map.data[MAP_IDX(map_.map.info.width, x, y)] = 0;
      }
     }
    }
   }
   got_map_ = true;

   //make sure to set the header information on the map
   map_.map.header.stamp = ros::Time::now();
   map_.map.header.frame_id = tf_.resolve( map_frame_ );

   sst_.publish(map_.map);
   sstm_.publish(map_.map.info);
}

//-----------------------------------------------------------------------------------------------------------------------//
bool SlamGMapping::mapCallback(nav_msgs::GetMap::Request  &req,
                               nav_msgs::GetMap::Response &res)
{
  boost::mutex::scoped_lock map_lock (map_mutex_);
  if(got_map_ && map_.map.info.width && map_.map.info.height)
  {
    res = map_;
    return true;
  }
  else
    return false;
}

//-----------------------------------------------------------------------------------------------------------------------//
void SlamGMapping::publishTransform()
{
  map_to_odom_mutex_.lock();
  ros::Time tf_expiration = ros::Time::now() + ros::Duration(tf_delay_);
  tfB_->sendTransform( tf::StampedTransform (map_to_odom_, tf_expiration, map_frame_, odom_frame_));
  map_to_odom_mutex_.unlock();
}
```

```
//-------------------------------------------------------------------------------------------------------------------//
void clbk(const geometry_msgs::Point32 msg)
{
 //ROS_INFO("AAAAAAAA: %f, %f, %f", msg.x, msg.y, msg.z);
 if(count==0)
 {
  for(int aa=0; aa<ON; aa++)
  {
   camera_x[aa] = -9999;
   camera_y[aa] = -9999;
   camera_z[aa] = -9999;
  }
 }
 if(msg.z>0.1 && msg.x>-9999 && msg.y>-9999)
 {
  float rotBA[2][2] = {{cos(pos_th), -sin(pos_th)}, {sin(pos_th), cos(pos_th)}};
  float posB[2][1] = {{msg.x}, {msg.y}};
  float posA[2][1] = {{(rotBA[0][0]*posB[0][0]) + (rotBA[0][1]*posB[1][0])}, {(rotBA[1][0]*posB[0][0]) + (rotBA[1][1]*posB[1][0])}};
  float aux_x = pos_x + posA[1][0];
  float aux_y = pos_y - posA[0][0];
  int flag = 0;
  for(int bb=0; bb<ON && flag==0; bb++)
  {
   ROS_INFO("%d: %f, %f, %f", bb, camera_x[bb], camera_y[bb], camera_z[bb]);
   if(aux_x>camera_x[bb]-3 && aux_x<camera_x[bb]+3 && aux_y>camera_y[bb]-3 && aux_y<camera_y[bb]+3)
   {
    camera_x[bb] = aux_x;
    camera_y[bb] = aux_y;
    camera_z[bb] = msg.z;
                                                                                                    flag = 1;
                                                                                                    ROS_IN
FO("REPITE");
   }
   else
   {
    if(count<ON)
                                                                                                    {

camera_x[count] = aux_x;
      camera_y[count] = aux_y;
      camera_z[count] = msg.z;
      count = count + 1;
                                                                                                     flag =
1;
                                                                                                    }
                                                                                                    else
                                                                                                    {

ROS_INFO("LLENO");
                                                                                                    }
   }
  }
 }
}
```

BIBLIOGRAFÍA

[1] Robotnik. (2018) Robotnik Automation. [Online]. https://github.com/RobotnikAutomation

[2] (2018) ROS Tutorials. [Online]. http://wiki.ros.org/ROS/Tutorials

[3] ROS Robots. Summit-XL. [Online]. https://robots.ros.org/summit-xl/

[4] Robotnik. Summit-XL. [Online]. https://www.robotnik.eu/mobile-robots/summit-xl/

[5] (2016) summit_xl_common. [Online]. http://wiki.ros.org/summit_xl_common

[6] (2016) summit_xl_sim. [Online]. http://wiki.ros.org/summit_xl_sim

[7] Vinssa. (2017, febrero) La historia y evolución de los robots móviles. [Online]. https://vinssa.com/news/la-historia-y-evolucion-de-los-robots-moviles.html

[8] (2009, diciembre) John Hopkins University. [Online]. http://cyberneticzoo.com/tag/johns-hopkins-university/

[9] Jonathan Polo. Una visión del campo de la robótica. [Online]. https://www.monografias.com/trabajos82/vision-del-campo-robotica/vision-del-campo-robotica2.shtml

[10] Missions. Mars Pathfinder. [Online]. https://rps.nasa.gov/missions/5/mars-pathfinder/

[11] (2010, febrero) Aplican la visión artificial al guiado de robots industriales. [Online]. https://www.agenciasinc.es/Noticias/Aplican-la-vision-artificial-al-guiado-de-robots-industriales

[12] (2016, junio) Adjunto 'SUMMIT_XL_Datasheet_2016_e.pdf'. [Online]. http://wiki.ros.org/Robots/SummitXL?action=AttachFile&do=view&target=SUMMIT_XL_Datasheet_2016_e.pdf

[13] Aníbal Ollero, *Robótica. Manipuladores y robots móviles*. Barcelona: Marcombo, 2001.

[14] (2018) [Online]. https://mongoose.ubuntu.com/?utm_expid=.92kIXiqDSpuSF35lE8FS1Q.1&utm_referrer=https%3A%2F%2Fwww.google.es%2F

[15] (2018) ROS Installation Options. [Online]. http://wiki.ros.org/ROS/Installation

[16] Luis García, "Navegación sin mapa y mapeado en robótica móvil para entornos no estructurados (Trabajo Fin de Grado)," Sevilla, 2017.

[17] Víctor Cazalla, "Development of teleoperation and monitoring tools for mobile robots (Trabajo Fin de Grado)," Sevilla, 2017.

[18] About ROS. [Online]. http://www.ros.org/about-ros/

[19] (2014, junio) ROS Concepts. [Online]. http://wiki.ros.org/ROS/Concepts

[20] Guillermo Heredia and Joaquín Ferruz, "Apuntes de la asignatura Robótica Avanzada," Sevilla, 2018.

[21] (2018) Documentation. [Online]. http://wiki.ros.org/Documentation

[22] (2018) ROS Answers. [Online]. https://answers.ros.org/questions/

[23] (2014) Gazebo Tutorials. [Online]. http://gazebosim.org/tutorials

[24] Ricardo Ragel de la Torre, "Modelado, control y simulación de un quadrotor equipado con un brazo manipulador robótico (Trabajo Fin de Carrera)," Sevilla, 2012.

[25] (2018) [Online]. https://github.com/RobotnikAutomation/summit_xl_common/tree/indigo-multirobot-devel

[26] (2018) [Online]. https://github.com/RobotnikAutomation/summit_xl_sim/tree/indigo-multirobot-devel

[27] (2018) [Online]. https://github.com/RobotnikAutomation/robotnik_msgs/tree/master

[28] (2018) [Online]. https://github.com/RobotnikAutomation/robotnik_sensors/tree/indigo-multirobot-devel

[29] (2018) [Online]. https://github.com/RobotnikAutomation/robotnik_trajectory_suite

[30] (2016, abril) [Online]. https://answers.ros.org/question/231265/using-keyboard-to-control-the-robot-in-gazebo/

[31] (2018) OpenCV. [Online]. https://opencv.org/

[32] (2016, mayo) vision_opencv. [Online]. http://wiki.ros.org/vision_opencv

[33] (2017, abril) cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages. [Online]. http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages

[34] "Apuntes de la asignatura Sistemas de Percepción," Sevilla, 2018.

[35] (2018) OpenCV Tutorials. [Online]. https://docs.opencv.org/master/d9/df8/tutorial_root.html

[36] Axis Communcations, "Hoja de datos de Cámaras domo de red AXIS P5512/-E PTZ," 2012.

[37] (2017, junio) Sistemas de Percepción en ROS. Detección y localización de objetivos. (Parte I). [Online]. https://cosasdeingenierossite.wordpress.com/2017/06/10/sistemas-de-percepcion-en-ros-deteccion-y-localizacion-de-objetivos-parte-i/

[38] (2018, febrero) gmapping. [Online]. http://wiki.ros.org/gmapping

[39] (2018) ros-perception/slam_gmapping. [Online]. https://github.com/ros-perception/slam_gmapping

[40] (2017, agosto) Ubuntu install of ROS Indigo. [Online]. http://wiki.ros.org/indigo/Installation/Ubuntu

[41] Joaquín Ferruz, "Resumen de órdenes del intérprete de comandos (Apuntes de la asignatura Informática Industrial)," Sevilla, 2018.

[42] (2014, noviembre) [Online]. https://answers.ros.org/question/196428/object-recognition-kitchen-cmake-fail/

[43] (2014, marzo) [Online]. https://answers.ros.org/question/145881/transferred-code-to-new-computer-and-catkin_make-not-working/

[44] (2017) Robotic simulation scenarios with Gazebo and ROS. [Online]. https://www.generationrobots.com/blog/en/robotic-simulation-scenarios-with-gazebo-and-ros/

[45] (2014) Tutorial: Using Gazebo plugins with ROS. [Online]. http://gazebosim.org/tutorials?tut=ros_gzplugins#RunningtheRRBotExample

[46] (2014) Tutorial: Using a URDF in Gazebo. [Online]. http://gazebosim.org/tutorials/?tut=ros_urdf

[47] (2014) Make a Mobile Robot. [Online]. http://gazebosim.org/tutorials?tut=build_robot&cat=build_robot

[48] (2015, agosto) rviz/UserGuide. [Online]. http://wiki.ros.org/rviz/UserGuide

[49] (2018) Software Framework. [Online]. https://www.techopedia.com/definition/14384/software-framework

GLOSARIO

INTA: Instituto Nacional de Técnica Aeroespacial. Dueño del robot que posee el departamento de Ingeniería de Sistemas y Automática para investigación.

Summit-XL: Plataforma robótica móvil con la que se trabaja en este proyecto.

Robotnik: Empresa encargada de la fabricación y lanzamiento del robot.

ROS: Robot Operating System. En español, Sistema Operativo Robótico. Framework en el que se programa el robot.

Framework [49]: Plataforma software para el desarrollo de código sobre temas similares. A partir de lo genérico, se busca llevar el código a aplicaciones concretas. Permite versatilidad en el uso de códigos de distinta procedencia.

SI: Sistema Internacional de unidades.

TFG: Trabajo Fin de Grado.

Ubuntu: Distribución del Sistema Operativo Linux.

Linux: Sistema Operativo.

PC: Personal Computer. En español, Computadora Personal.

OpenCV: Open Source Computer Vision Library. En español, Librería de Visión por Computador de Código Abierto.

Gazebo: Simulador del framework ROS.

RViz: Visualizador de datos de ROS.

NASA: National Aeronautics and Space Administration. En español, Administración Nacional de la Aeronáutica y del Espacio.

PTZ: Pan-Tilt-Zoom de una cámara, que son sus ángulos de barrido e inclinación y el zoom, respectivamente.

Hokuyo: Sensor láser que posee el Summit-XL.

RTK-DGPS: Real-Time Kinematics and Differential GPS. En español, Posicionamiento Cinemático en Tiempo Real y GPS Diferencial.

GPS: Global Positioning System. En español, Sistema de Posicionamiento Global.

IMU: Inertial Measurement Unit. En español, Unidad de Medición Inercial.

3D: Tres dimensiones.

Odometría: Sistema de estimación de la posición de un robot móvil.

USB: Universal Serial Bus. En español, Bus Serie Universal.

RS232: Protocolo de transmisión de datos por bus serie.

GPIO: General Purpose Input/Output. En español, Entrada/Salida de Propósito General.

RJ45: Conexión de la mayoría de las tarjetas de red Ethernet.

CPU: Central Processing Unit. En español, Unidad Central de Procesamiento.

WiFi: Tecnología de comunicación para conectar dispositivos electrónicos a internet de manera inalámbrica.

Windows: Sistema Operativo de Microsoft Corporation.

Unix: Sistema Operativo en el que se basa Linux.

C++: Lenguaje de programación similar a C pero orientado a objetos.

SLAM: Simultaneous Localization and Mapping. En español, Localización y Construcción del Mapa Simultánea.

EKF: Extended Kalman Filter. En español, Filtro de Kalman extendido.

PS3: PlayStation 3. Videoconsola.

BSD: Licencia software para los sistemas operativos tipo BSD (Berkeley Software Distribution).

xml: eXtensible Markup Language o Lenguaje de Marcado Extensible.

Python: Lenguaje de programación.

TCP/IP: Transmission Control Protocol/Internet Protocol. En español, Protocolo de Transmisión de Control/Protocolo de Internet.

URDF: Unified Robot Description Format. En español, Formato Unificado de Descripción de Robot.

API: Aplication Programming Interface. En español, Interfaz de Programación de Aplicaciones.

ACML: Adaptive Monte Carlo Localization. En español, Adaptación de la Localización de Monte Carlo.

Java: Lenguaje de programación.

RGB: Red, Green, Blue. En español, RVA (Rojo, Verde y Azul).

FOV: Field Of View. En español, Campo de Visión.

Matriz cuadrada: Matriz con el mismo número de filas que de columnas.

Kinect: Tipo de cámara comercial.

OS: Sistema Operativo.

RAM: Random Access Memory. En español, Memoria de Acceso Aleatorio.

ISO: Tipo de archivo en un computador.

Rufus: Programa para crear un USB de arranque.

Bios: Software de la placa base del PC necesario para el arranque del mismo y la carga del OS.

UEFI: Unified Extensible Firmware Interface. En español, Interfaz de Firmware Extensible Unificada.

LiveCD: Versión de prueba de un sistema operativo que no está instalado en el PC y al cual se accede desde un dispositivo externo.

Grub: Administrador de arranque que nos permite elegir entre los sistemas operativos instalados en un PC.

Shell: Terminal, intérprete de comandos.

PS4: PlayStation 4. Videoconsola.

Xbox: Videoconsola.

Logitech: Empresa fabricante de dispositivos electrónicos.