

Título del proyecto

Daniel Gómez

27 de mayo de 2022

Resumen

Abstract

Agradecimientos

Índice general

Resumen	III
Abstract	V
Agradecimientos	VII
Lista de figuras	XI
Lista de tablas	XIII
I Introducción	XV
1. Introducción y visión general	1
1.1. La localización como una necesidad	1
1.2. Objetivo	6
1.3. Estructura del documento	7
II Desarrollo	9
2. Herramientas utilizadas	11
2.1. Robotic Operating System	11
2.1.1. Arquitectura de la comunicación mediante ROS	11
2.1.2. ¿Por qué ROS?	13
2.2. Mapas de ocupación	14
2.3. Sensores RGBD	15
2.3.1. Imagen de profundidad	15
2.4. Nubes de puntos	15
2.5. YOLO: You only look once	17
2.5.1. CUDA: Compute Unified Device Architecture	18
3. Diseño e implementación	19
3.1. Vista general del sistema	19
3.2. Fuente de datos	21

ÍNDICE GENERAL

3.3. De imagen de profundidad a barrido láser	22
3.4. Generación del mapa de ocupación	24
3.5. Detección de objetos	25
3.5.1. Descompresión de la imagen de color	25
3.5.2. Ejecución de YOLO	25
3.6. Procesamiento de los objetos detectados	26
3.7. Análisis de los objetos detectados	27
4. Resultados obtenidos	29
4.1. Mapa generado a partir de la imagen de profundidad	29
4.1.1. Generación del láser artificial	30
4.1.2. Modificación de parámetros por defecto	34
4.2. Análisis de la fiabilidad de las detecciones	35
III Conclusiones	39
5. Conclusiones, propuestas y líneas futuras	41
5.1. Conclusiones	41
5.2. Propuestas y líneas futuras	42
IV Apéndices	45
A. Otras herramientas utilizadas	47
A.1. Librerías y paquetes	47
A.1.1. <i>message_filters</i>	47
A.1.2. <i>fstream</i>	47
A.1.3. <i>libfreenect</i>	48
A.2. RVIZ	48
B. Código de los nodos diseñados	49
B.1. Nodo <i>cam_info</i>	49
B.2. Nodo <i>sync_info</i>	51
B.3. Nodo <i>write_objects</i>	52

Índice de figuras

1.1.	Porción del plano del metro de Madrid [2]	2
1.2.	Robot SHAKY, 1968 [4]	3
1.3.	Ejemplos de tipos de mapas	4
1.4.	Mapa generado con el barrido de un láser 2D	5
1.5.	Comparación del resultado de un láser 2D y un sensor RGBD	6
2.1.	Mecanismos de comunicación en ROS	13
2.2.	Ejemplo de mapa de ocupación	14
2.3.	Kinect V1	15
2.4.	Ejemplo de imagen de profundidad	16
2.5.	Ejemplo de nube de puntos	16
2.6.	Comparación con otros modelos	17
2.7.	Procedimiento de YOLO	18
3.1.	Árbol de nodos y topics. Los nodos diseñados para este trabajo son de color azul. Los nodos de terceros cuyo código ha sido modificado son de color naranja. Los nodos descargados que no se han modificado son de color blanco	20
3.2.	Diagrama del nodo cam_info	22
3.3.	Diagrama del nodo sync_info	22
3.4.	Salida en la terminal del nodo darknet_ros	26
4.1.	Mapas generados con láser 2D (a) y con sensor RGBD (b)	30
4.2.	Superposición de los dos mapas	31
4.3.	Muestra de medida 1: diferencia entre las medidas del láser real (blancas) y del láser artificial (rojas)	32
4.4.	Muestra de medida 2: diferencia entre las medidas del láser real (blancas) y del láser artificial (rojas)	32
4.5.	Muestra de medida 3: diferencia entre las medidas del láser real (blancas) y del láser artificial (rojas)	33
4.6.	Muestra de medida 4: diferencia entre las medidas del láser real (blancas) y del láser artificial (rojas)	33
4.7.	Resultados para diferentes parámetros de <code>linearUpdate</code> , <code>angularUpdate</code> y <code>temporalUpdate</code> , respectivamente	34

ÍNDICE DE FIGURAS

4.8. Resultados para distintos valores de distancia máxima	35
4.9. Trayectoria seguida por el robot. El punto rojo indica el inicio de la trayectoria. El punto verde indica el final	36
4.10. Gráfica de objetos detectados. Muestra el número de veces que fue detectado cada uno de los objetos	36
4.11. Momentos en los que se ha detectado la clase ‘oven’. Los puntos azules representan la trayectoria del robot, los rojos las veces que se detectó la clase	37
4.12. Detección de la clase ‘oven’ en la imagen	37
4.13. Momentos en los que se ha detectado la clase ‘refrigerator’. Los puntos azules representan la trayectoria del robot, los rojos las veces que se detectó la clase	38
4.14. Detecciones de la clase ‘refrigerator’ en la imagen	38

Índice de cuadros

3.1. Formato del archivo CSV generado	27
---	----

Parte I

Introducción

Capítulo 1

Introducción y visión general

En este capítulo se hará una breve introducción para entender el contexto de este trabajo y el porqué de la necesidad del mismo. Se comentará la importancia de la localización para los seres humanos y se hará una introducción sobre los mapas utilizados por los robots para localizarse. Finalmente se expondrá la estructura del documento.

1.1. La localización como una necesidad

Que levante la mano quien no se ha perdido alguna vez por los despachos de alguna de las facultades, sobre todo en esos ambientes prácticamente idénticos entre sí. Todo ser humano alguna vez se ha parado en seco, ha mirado a un lado y a otro con cara de desaprobación y se ha preguntado: *¿dónde estoy?* Estas situaciones, a veces incómodas, se deben esencialmente a que todo ser vivo con capacidad para desplazarse por un entorno necesita localizarse y orientarse si quiere alcanzar un objetivo, más aún si dicho entorno es totalmente desconocido.

A lo largo de los años, los seres humanos hemos utilizado multitud de métodos para la orientación, incluso en las situaciones más desfavorables. Un estudio para la revista *The Royal Society* de la Universidad de Rennes sostiene que los vikingos usaban una variedad de calcita para calcular dónde se encontraba el sol utilizando la polarización de la luz dispersada por las nubes. Esta piedra solar se llama “Espato de Islandia” y les permitía orientarse incluso en fechas donde el sol parecía no querer mostrarse, algo que sucede muy frecuentemente en los mares nórdicos [1]. Hoy en día, con el uso de la tecnología GPS y de aplicaciones que la implementan como Google Maps, hemos cambiado esa piedra solar por un dispositivo electrónico que, no solo nos localiza y orienta en prácticamente cualquier situación, sino que lo hace con un error casi despreciable. Pese al avance de las tecnologías de localización como GPS, existen ciertas situaciones en las que es complicado aprovecharse de ellas, como por ejemplo en los entornos indoor. En estos casos, los satélites no son capaces

CAPÍTULO 1. INTRODUCCIÓN Y VISIÓN GENERAL

de comunicarse con nuestros dispositivos, por lo que necesitamos de otro mecanismo para localizarnos.

En la figura 1.1 se muestra el plano de la red de metro de la Comunidad de Madrid. A primera vista, uno se pierde entre tanto entresijo de líneas de colores pero si se establece una estación de origen y otra de destino, este plano facilita, e incluso posibilita, concretar el camino a tomar. Además, en las estaciones suele haber una gran cantidad de señalizaciones que hacen aún más fácil la localización. Entonces, si los humanos, seres inteligentes con percepción visual y capacidad de razonamiento, necesitamos localizarnos para alcanzar un objetivo, ¿porqué no lo iban a necesitar los robots?



Figura 1.1: Porción del plano del metro de Madrid [2]

La robótica móvil ha ido evolucionando a un ritmo vertiginoso desde que comenzó a desarrollarse. En los años sesenta se diseñó el apodado como *SHAKY* o “el primer robot inteligente móvil del mundo” según el IEEE (Instituto de Ingenieros Eléctricos y Electrónicos, del inglés *Institute of Electrical and Electronics Engineers*). Este robot era el primero de su generación capaz de navegar en un entorno no controlado, sirviéndose de múltiples dispositivos que le proporcionaban información sobre la distribución de los elementos que le rodeaban, dándole la posibilidad al robot de evitarlos durante el trayecto [3].

Un robot autónomo móvil (ARM) es un robot capaz de navegar a través de un entorno sin supervisión directa por parte de un operador humano ni la necesidad de disponer de una ruta fijada previamente. Para poder llevarlo a cabo, los robots disponen de multitud de sensores que les permiten percibir e interpretar el entorno, dotando al robot de la capacidad de navegar por el mismo evitando obstáculos, tanto fijos como móviles. Para poder realizar una navegación de alto nivel que les permita ir desde un punto origen a otro punto destino, los robots se sirven de mapas. Un mapa es una representación bidimensional del entorno. Existen dos tipos de mapas

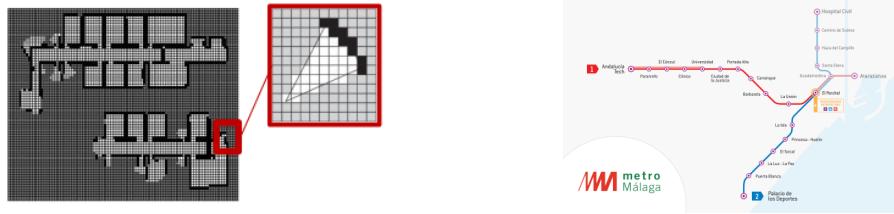
1.1. LA LOCALIZACIÓN COMO UNA NECESIDAD



Figura 1.2: Robot SHAKEY, 1968 [4]

de aplicación a la robótica: los mapas métricos y los mapas topológicos. Los mapas métricos (figura 1.3, izquierda) dividen el entorno en una rejilla donde cada celda representa una probabilidad de estar ocupada o de no estarlo. Si se discretiza esa probabilidad, el resultado es un mapa de ocupación, (también llamado *floorplan* o “plano de suelo”, en un ámbito menos técnico), si no, se obtiene un mapa probabilístico. El mapa métrico más simple es el formado por un conjunto de referencias visuales o *landmarks* de posición conocida que ayudan al robot a estimar su posición. Los mapas topológicos (figura 1.3, derecha) están basados en grafos y representan la conectividad entre cada uno de sus nodos. Existe una tercer tipo de mapa que es un híbrido entre los comentados y es el más utilizado para los robots: utilizan los grafos para la generación de trayectorias (navegación) y el mapa métrico para la localización.

CAPÍTULO 1. INTRODUCCIÓN Y VISIÓN GENERAL



(a) Ejemplo de mapa métrico:
mapa de ocupación

(b) Mapa topológico: líneas
del metro de Málaga [5]

Figura 1.3: Ejemplos de tipos de mapas

Los mapas métricos más utilizados en la robótica móvil son los mapas de ocupación o *floorplans*. Son generados en una primera fase de inspección del entorno, empleando los mismos sensores que luego servirán para la navegación autónoma. Tradicionalmente, estos mapas de ocupación son generados empleando un láser 2D o LIDAR (dado su gran alcance y amplio campo de visión), generando por tanto mapas bidimensionales que especifican aquellas zonas del entorno que están libres de obstáculos y por tanto susceptibles de navegación, y aquellas que no lo están. No obstante, dado que un láser 2D no puede distinguir entre los objetos detectados, los mapas de ocupación bidimensionales generados con este sensor contienen, no sólo los elementos estructurales del entorno como paredes, puertas, columnas, etc., si no que incluyen multitud de objetos como mesas y sillas, camas, cajas e incluso personas si estaban presentes en el momento de generar el mapa.

La inclusión de estos elementos no estructurales en el mapa de ocupación puede ocasionar problemas si se pretende utilizar ese mapa durante un largo período de tiempo. El mapa generado no estaría preparado si el entorno que representa sufre algún tipo de modificación de mobiliario, por lo que sería necesario volver a generar un nuevo mapa, con todo lo que eso conlleva. Sin ir más lejos, en la Escuela de Ingenierías Industriales hay muchos elementos que no pertenecen a la estructura del edificio que, si hoy mismo se utilizara un robot equipado con un láser 2D para generar un mapa, ocasionarían los problemas ya comentados. Estos elementos son, por ejemplo, las impresoras que hay en algunas zonas, las máquinas de refrescos y snacks, papeleras, carteles publicitarios, las nuevas mesas instaladas en la zona central, etc (figura 1.5).

En la figura 1.4 se muestra un ejemplo de un mapa generado de un entorno virtual a través de un láser 2D de 360º. Como se puede observar hay zonas, marcadas con numeración, en las que el mapa no representa fielmente las dimensiones reales de la vivienda. De modo que, si los objetos que no forman parte de la estructura modifican su posición después de la primera fase de análisis, el mapa de ocupación generado no podría ser utilizado por el robot para la navegación.

1.1. LA LOCALIZACIÓN COMO UNA NECESIDAD

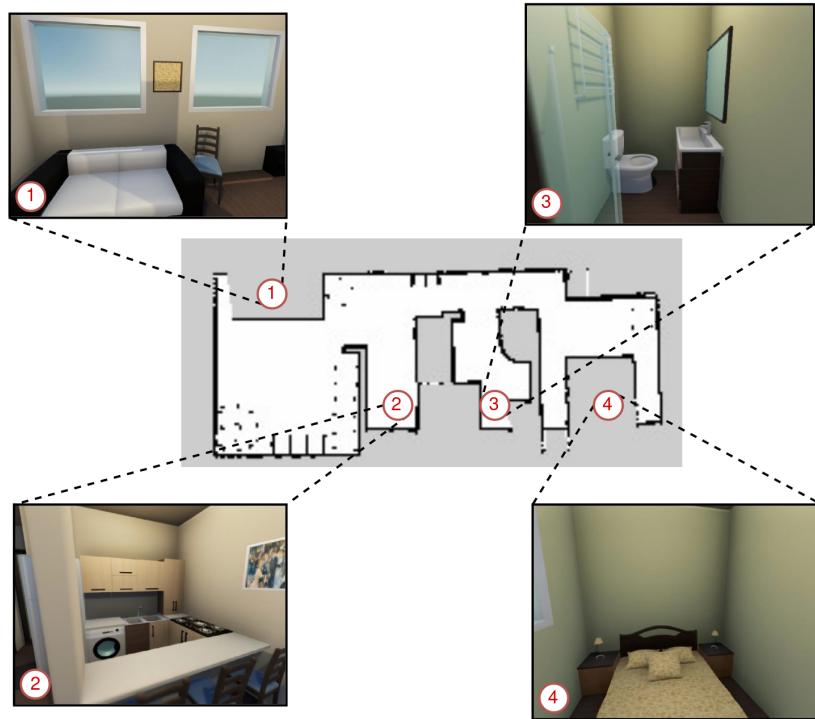


Figura 1.4: Mapa generado con el barrido de un láser 2D

Dado que estos métodos de generación de mapas de ocupación mediante láser 2D sufren frente a la variabilidad en la distribución, surge la idea de utilizar otros sensores que permitan discernir entre los distintos elementos que pudieran formar parte del entorno. Los sensores RGBD son cámaras que, además de proporcionar información sobre el color, ofrecen la distancia a la que se encuentra cada píxel captado en la imagen con respecto a la cámara. Por lo tanto, ofrecen información en 3D del dominio, lo que abre más posibilidades a la hora de percibir e interpretar la información captada mediante los sensores. En la figura 1.5 se muestra un ejemplo de lo que se quiere conseguir aprovechando las ventajas de los sensores RGBD frente a los láseres 2D.

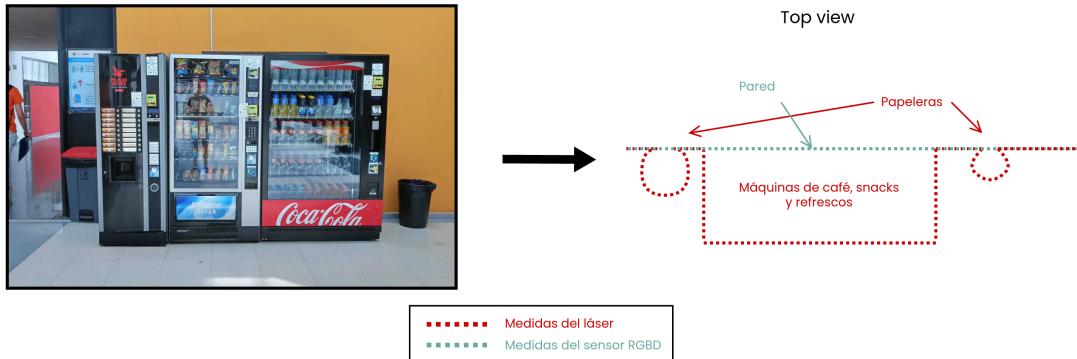


Figura 1.5: Comparación del resultado de un láser 2D y un sensor RGBD

En este proyecto se pretende analizar una posible solución al problema de fiabilidad en la representación de los mapas de ocupación realizados con sensores LIDAR mediante el uso de sensores RGBD para generar *floorplans* fieles a la estructura invariable del entorno. Entre las utilidades de conseguir un mapa sin los objetos móviles, las más destacadas son:

- La más importante es la posibilidad de tener un mapa que permita al robot localizarse aún con los cambios que puedan surgir a lo largo del tiempo en los elementos que conforman el entorno. Así se evita tener que generar un nuevo mapa cada vez que surja algún cambio significativo.
- En aplicaciones de realidad aumentada para el diseño de interiores esto permitiría obtener las dimensiones reales de la vivienda y así tener una idea en primera persona del cambio de mobiliario o de una nueva organización del mismo.
- Normalmente existe una clara disparidad entre el plano de una vivienda o de un edificio diseñado por el arquitecto y el producto final. Con este mapa generado sería posible comparar y obtener un plano fiel a la realidad y así validar las dimensiones reales de la infraestructura.

1.2. Objetivo

Dado que los robots utilizan los mapas de ocupación o *floorplans* para localizarse y navegar en un entorno, es necesario que estos sean lo más fiel posible a la estructura real del edificio o de la vivienda. Esto resulta imposible si durante la fase de obtención del mapa existen elementos que no pertenecen a la infraestructura, ya que provocan que el mapa resultante no sea fiel las dimensiones reales del entorno.

1.3. ESTRUCTURA DEL DOCUMENTO

Debido a la necesidad y utilidad de obtener mapas de ocupación sin elementos cambiantes, el objetivo de este trabajo es utilizar la información proporcionada por sensores RGBD para la generación de *floorplans*, solucionando problemas de variabilidad en el entorno que perjudican seriamente la fiabilidad del mapeado mediante sensores láser.

Se obtendrá información acerca de la profundidad de los píxeles captados y se escogerán aquellos puntos menos restrictivos (más alejados) para generar un láser artificial que permita la creación del mapa de ocupación mediante algoritmos de mapeado.

Paralelamente, se utilizará un modelo entrenado de inteligencia artifical, YOLO, para detectar los elementos de la imagen de color y así llevar un registro de los objetos que se ha encontrado el robot en su trayectoria.

Finalmente, se hará una comparativa de los resultados obtenidos únicamente con el LIDAR con los obtenidos mediante el procesamiento de la información de la cámara. Además se hará un estudio de los objetos detectados durante el mapeado y de la fiabilidad del modelo de detección.

1.3. Estructura del documento

Este documento está dividio en tres[cuatro?] partes. En la primera parte se hará una introducción al contexto del proyecto y se explicará por qué se ha desarrollado.

La segunda parte consta de tres capítulos. En el primero de ellos se expondrán las herramientas utilizadas y se explicarán brevemente cómo se utilizan y porqué se han escogido. En un segundo capítulo, el más importante, se desarrollará el trabajo y se explicará cómo se ha llevado a cabo. En el tercer y último capítulo se expondrán los resultados y se compararán con los ideales, comentando sus diferencias y problemas encontrados.

En la tercera [y última?] parte se mostrarán las conclusiones obtenidas tras la consecución del proyecto y se comentarán algunas posibles líneas de trabajo abiertas de cara al futuro, así como posibles mejorías que se pudieran implementar.

[Cuarta parte. ANEXOS. No sé si me dará tiempo a una cuarta parte de Anexos]

Parte II

Desarrollo

Capítulo 2

Herramientas utilizadas

2.1. Robotic Operating System

Robot Operating System (ROS) es un framework para el desarrollo de software de robots. Se desarrolló originariamente en 2007 por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte a su proyecto STAIR¹ [7]. Es software libre bajo términos de licencia BSD² que, pese a no ser un sistema operativo, provee los servicios estándar de éstos entre los que se incluyen la abstracción del hardware, el control de dispositivos de bajo nivel, la implementación de funcionalidad de uso común, el paso de mensajes entre procesos y el mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros.

2.1.1. Arquitectura de la comunicación mediante ROS

La unidad básica de una comunicación en ROS es el *nodo*. Un nodo es un programa diseñado en Python o C++ que se comunica con otros nodos principalmente mediante *topics*. Cuando varios nodos se complementan para cumplir una determinada función, se suelen agrupar en paquetes. Un paquete contiene, entre otras cosas, el nodo o los nodos que lo conforman, las dependencias necesarias para la ejecución de alguno de sus nodos, información acerca del propio paquete, etc. ROS tiene sus propios paquetes incluidos de forma estándar con la instalación pero además se pue-

¹STAIR (Stanford Artificial Intelligence Robot) es un proyecto llevado a cabo por la Universidad de Stanford para desarrollar robots capaces de navegar en entornos indoor e interactuar con objetos y personas mediante inteligencia artificial. Para su versión 2.0 desarrollaron el framework de ROS [6]

²La licencia BSD es la licencia de software libre otorgada principalmente para los sistemas BSD (Berkeley Software Distribution). Es una licencia permisiva que permite la redistribución libre o privativa [8]

CAPÍTULO 2. HERRAMIENTAS UTILIZADAS

den instalar paquetes de terceros, desarrollados por otros usuarios.

Existe un nodo especial, llamado nodo *master*, que se encarga de registrar las direcciones de cada nodo y qué y dónde publica o se suscribe. Además, se encarga de alojar el servidor de parámetros, accesibles y modificables en cualquier momento. Es necesario lanzar este nodo previamente al lanzamiento de cualquier otro.

Los topics son el método más común de comunicación entre nodos. Está basado en el modelo de publicador/suscriptor. Los nodos pueden publicar (Publisher) o recibir (Subscriber) mensajes de un único tipo, establecidos por el nodo publicador. En un mismo topic pueden publicar y suscribirse tantos nodos se necesiten³.

Otro método de comunicación son los servicios o *services*. Están basados en un modelo de llamada/respuesta. Mientras que los topics permiten a los nodos obtener continuas actualizaciones de datos, los servicios solo ofrecen respuesta tras ser llamados. Permiten la inclusión de parámetros tanto en la solicitud como en la respuesta al servicio. Este método es bloqueante, es decir, el nodo solicitante queda bloqueado al momento de enviar la solicitud hasta que reciba una respuesta, por lo que están pensados para ser utilizados en tareas que no ocupen mucho tiempo.

El otro método de comunicación que existe en ROS son las acciones o *actions*, pensadas para tareas que necesiten una gran cantidad de tiempo para poder ser realizadas. Las acciones funcionan con un modelo cliente/servidor. A diferencia que los servicios, las acciones no son bloqueantes y permiten ser canceladas mientras se están ejecutando.

Como se ha comentado, el método más usual de comunicación es a través de topics, donde los nodos envían y reciben información continuamente, sin peticiones ni esperas. Los servicios y acciones se utilizan en función de los requerimientos del software a desarrollar.

³Realmente, ROS especifica que no existe límite de nodos suscritos a un mismo topic. Sin embargo, se ha demostrado que a partir de 10 nodos, la fiabilidad se ve reducida y la latencia aumentada de forma significativa, pudiendo incluso ocasionar que algunos mensajes no sean recibidos por algunos suscriptores [9]

2.1. ROBOTIC OPERATING SYSTEM

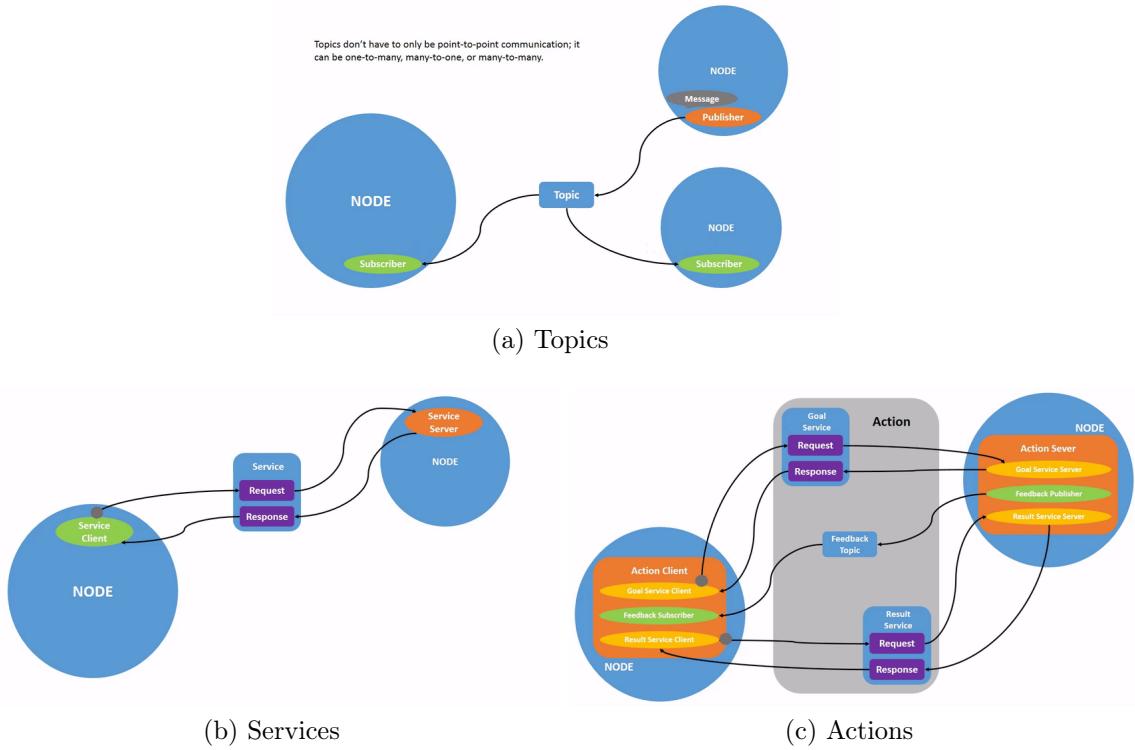


Figura 2.1: Mecanismos de comunicación en ROS

ROS también implementa una herramienta muy útil a la hora de probar los programas que se desarrollan. Un *rosbag* de ROS es una herramienta que permite capturar todos los mensajes que se publican por determinados topics para luego reproducirlos. Esto permite ejecutar el robot una única vez capturando todos los topics y luego reproducir los mensajes de estos topics cuantas veces queramos, permitiendo probar los nodos que se estén desarrollando sin tener que hacerlo directamente sobre el robot. Toda esta información se guarda en un archivo .bag.

2.1.2. ¿Por qué ROS?

La comunidad robótica ha evolucionado considerablemente a lo largo de los años. Pese a este rápido progreso, los robots aún presentan auténticos desafíos para los desarrolladores de software. ROS nace precisamente para facilitar muchas de las dificultades que surgen en la comunicación entre los distintos elementos que conforman un robot.

Muchos de los sistemas robóticos modernos necesitan de un sistema de comunicaciones que sirva de enlace entre los diferentes procesos. Estos procesos suelen dividirse en varias computadoras, lo que complica aún más la comunicación. Las diferentes vías de comunicación que ROS pone a disposición brindan a los desarro-

CAPÍTULO 2. HERRAMIENTAS UTILIZADAS

lladores infinitas de posibilidades.

Pero la que es quizás la mayor ventaja de ROS frente a otros frameworks y software especializado en robótica es la gran comunidad que tiene detrás. Existen miles de usuarios que contribuyen continuamente con nuevos paquetes que implementan funcionalidades de todas las ramas de la robótica, lo que abre un abanico de opciones para el desarrollador, reduciendo en gran medida la curva de aprendizaje.

2.2. Mapas de ocupación

Un mapa o rejilla de ocupación es una representación bidimensional del entorno que se almacena en el robot para realizar tareas de navegación. Está formado por celdas que discretizan el entorno y determinan si una porción del espacio está ocupado o no. Generalmente, las celdas ocupadas se representan con el color negro, y las celdas libres con el blanco.

Para generar estos mapas, se utiliza la información obtenida de los diferentes sensores que conforman el robot, como pueden ser infrarrojos, cámaras RGBD, sónares o, los más utilizados, láseres 2D. Independientemente de la fiabilidad del sensor, no es posible establecer con certeza si una celda está ocupada o no. En la generación de estos mapas, típicamente se establecen tres suposiciones: una celda puede estar libre u ocupada, cada celda es independiente a las demás y el entorno es estático. Para estimar las distribuciones de probabilidad de cada celda se utiliza generalmente un Filtro de Ocupación Bayesiano (BOF) [10].

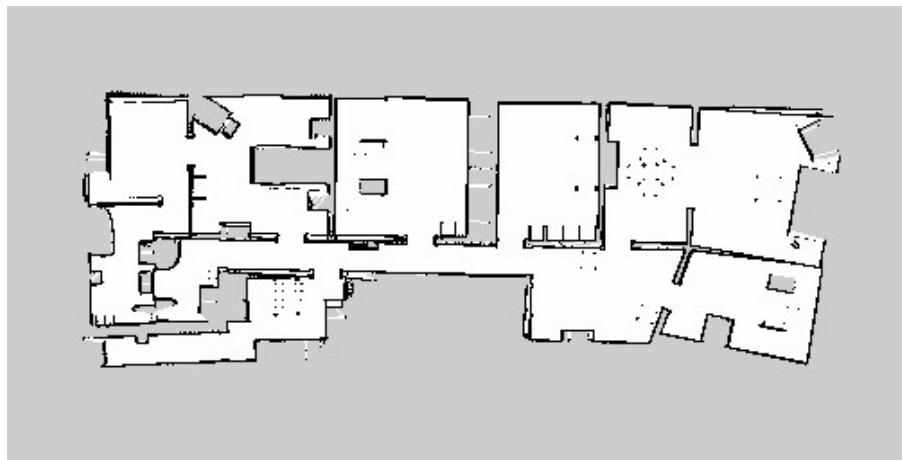


Figura 2.2: Ejemplo de mapa de ocupación

2.3. Sensores RGBD

Las cámaras RGBD son sensores visuales que, además de ofrecer información de color del entorno, proporcionan la distancia a la que se encuentra cada píxel de la imagen. Son cámaras a color comunes a las que se le añade un sensor de profundidad (normalmente, sensores infrarrojos) junto con un procesamiento que les permite percibir e interpretar la distancia a la que se sitúan los diferentes elementos del entorno que estén dentro de su campo de visión.



Figura 2.3: Kinect V1

Algunos de los fabricantes más importantes de este tipo de sensores son Microsoft, con la conocida Kinect; ASUS, IFM, StereoLabs, Intel, Orbbec, etc.

2.3.1. Imagen de profundidad

La imagen de profundidad, al igual que cada uno de los canales de color RGB, es una matriz donde cada valor representa un nivel de gris asociado a una distancia y está referido a cada uno de los píxeles que componen la imagen. Las dimensiones de esta imagen vendrán determinadas por la resolución de la cámara, es decir, si por ejemplo la resolución de la cámara es de 640x480 píxeles, la matriz tendrá 480 filas y 640 columnas.

El valor de distancia para cada píxel establece un nivel de gris. Generalmente, los objetos situados más alejados del sensor tendrán valores más claros, mientras que a los más cercanos se le asignan valores más oscuros. Esta información se decodifica para obtener así la información absoluta en distancia, típicamente en metros o milímetros, dependerá de la codificación con la que la cámara envía la información.

2.4. Nubes de puntos

Una nube de puntos o pointcloud es un conjunto de vértices en un sistema de coordenadas tridimensional. Estos vértices se identifican habitualmente como coordenadas X, Y y Z y son representaciones de la superficie externa de un objeto. Se

CAPÍTULO 2. HERRAMIENTAS UTILIZADAS



Figura 2.4: Ejemplo de imagen de profundidad

crean a partir de escáneres láser tridimensionales o imágenes de profundidad.

Las nubes de puntos tienen infinidad de aplicaciones como la elaboración de modelos 3D en CAD de piezas fabricadas, la inspección de calidad en metrología y muchas otras en el ámbito de la visualización, animación, texturización y aplicaciones de personalización masiva.

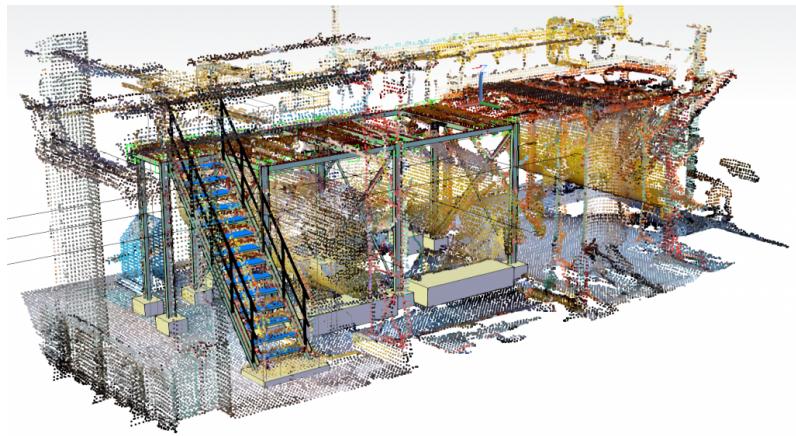


Figura 2.5: Ejemplo de nube de puntos

2.5. YOLO: You only look once

You only look once (YOLO) es un sistema en tiempo real de detección de objetos basado en rede neuronales convolucionales y deep learning. Su principal característica y la que le distingue de otros modelos para la detección de objetos es que solo requiere visualizar una única vez la imagen (haciendo honores a su acrónimo). Esta propiedad le permite ser más rápido que los modelos competidores permitiendo la detección en tiempo real en vídeos de hasta 30 FPS. Según comentan sus creadores, es 1000 veces más rápido que R-CNN y 100 veces más rápido que Fast R-CNN.

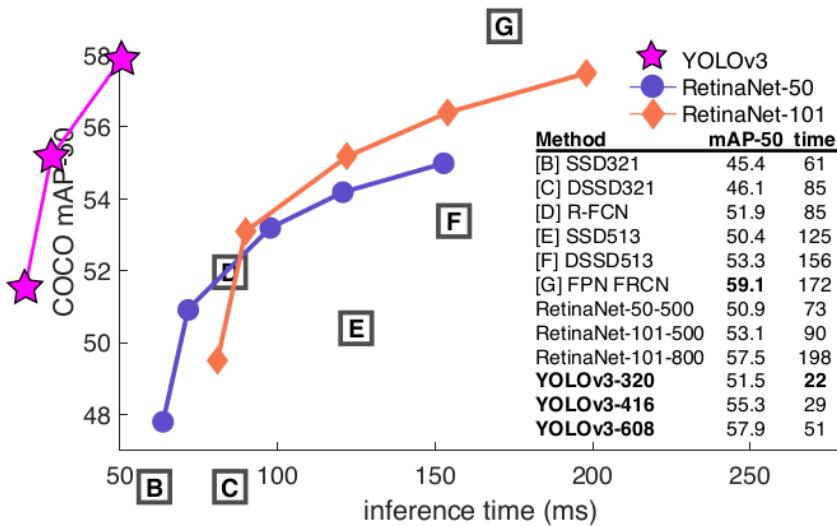


Figura 2.6: Comparación con otros modelos

El procedimiento llevado a cabo por YOLO es sencillo. Primero divide la imagen en una cuadrícula de $S \times S$ y en cada una de estas celdas predice N posibles bounding boxes y calcula su probabilidad. Tras el cálculo, se eliminan aquellas que no superen un umbral de probabilidad. A las bounding boxes restantes, se les somete a un proceso de supresión de no máximos con el objetivo de eliminar los objetos que fueron detectados por duplicado, obteniéndose el resultado de la figura 2.7.

YOLO es fácilmente implementable en ROS gracias al paquete darknet_ros, que permite utilizarlo tanto en GPU como en CPU. La ventaja de usar la GPU para lanzar el modelo es que es 500 veces más rápido que utilizando la CPU. Para utilizar la GPU es necesario instalar el software CUDA de Nvidia [11].

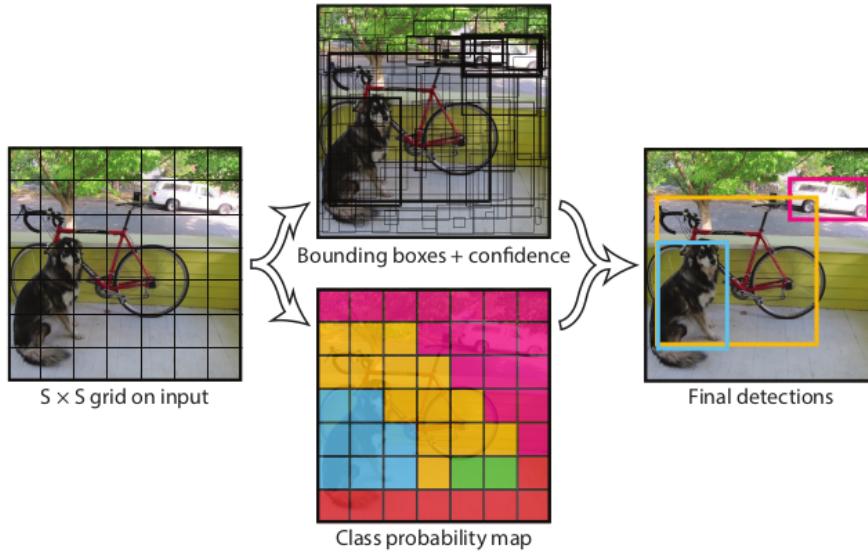


Figura 2.7: Procedimiento de YOLO

2.5.1. CUDA: Compute Unified Device Architecture

CUDA es un conjunto de herramientas de desarrollo creadas por Nvidia que permiten a los programadores usar una variación del lenguaje de programación C (CUDA C) para codificar algoritmos GPU de Nvidia. Tiene como objetivo explotar las ventajas de las GPU frente a las CPU de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten lanzar un altísimo número de procesos simultáneos [12].

Las principales ventajas de este sistema de computación son:

- Lecturas dispersas. Se puede consultar cualquier posición en memoria.
- Memoria compartida. CUDA pone a disposición un área de memoria de entre 16KB y 48KB que se compartirá entre hilos del mismo bloque, pudiéndose utilizar como memoria caché.
- Lecturas más rápidas de y hacia la GPU.
- Soporte para enteros y operadores a nivel de bit.

Capítulo 3

Diseño e implementación

Para implementar el sistema que nos permita obtener estos mapas de ocupación a partir de la imagen de profundidad generada por una cámara RGB-D, se utilizarán una serie de nodos de ROS. Algunos de estos paquetes y nodos son proporcionados por la comunidad de usuarios de ROS. Otros son diseñados expresamente para este proyecto.

Primeramente se realizará una visión general de todo el conjunto de nodos y paquetes. Se explicará el camino que llevará la información y las formas que tienen los nodos de comunicarse.

Sobre los paquetes de terceros, se hará una breve explicación sobre la función que tienen de fábrica y la funcionalidad que se le han dado en este trabajo, así como el modo de comunicación con todo el árbol de nodos.

3.1. Vista general del sistema

Como se ha comentado, el objetivo es transformar la imagen de profundidad que se obtiene de un sensor RGBD en un láser artificial para generar un mapa de ocupación. En otro proceso, simultáneamente o de forma asíncrona, se generará un archivo de texto o CVS con los elementos detectado por YOLO de la imagen de color para tener un registro y poder analizarlo. En la figura 3.1 se muestra el árbol de nodos y topics de todo el sistema. Los nodos que han sido diseñados para este proyecto se muestran en color azul. Los nodos que han sido obtenidos de otras fuentes pero han necesitado de alguna modificación en su código se muestran en color naranja. Los nodos que no se han modificado se muestran en blanco. Los topics se representan con una caja de color blanco.

A continuación se enumeran todos los nodos que se van a utilizar en este proyecto:

CAPÍTULO 3. DISEÑO E IMPLEMENTACIÓN

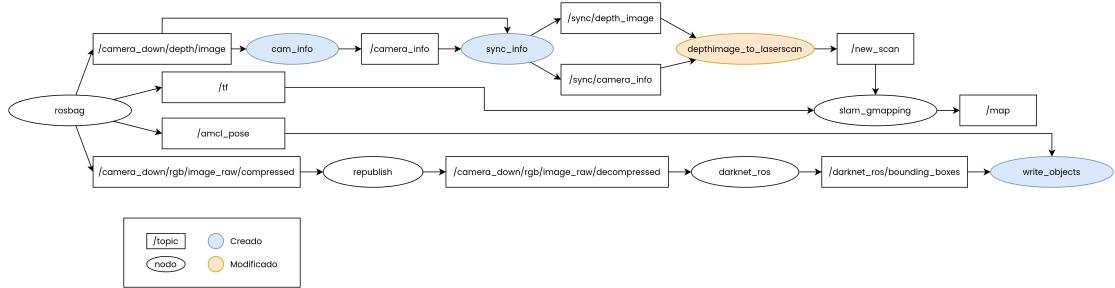


Figura 3.1: Árbol de nodos y topics. Los nodos diseñados para este trabajo son de color azul. Los nodos de terceros cuyo código ha sido modificado son de color naranja. Los nodos descargados que no se han modificado son de color blanco

- El nodo **cam_info** es un nodo diseñado expresamente para este trabajo. Se encarga de recibir la cabecera de la imagen de profundidad para generar un mensaje con información de la cámara y publicarlo.
- El nodo **sync_info** es otro de los que han sido diseñados para el proyecto. Su función es sincronizar los mensajes de la imagen de profundidad y de la información de la cámara.
- **depthimage_to_laserscan** es un nodo de un paquete homónimo diseñado por Chad Rocke [13] que se encarga, principalmente, de transformar una imagen de profundidad en un barrido láser en función de unos parámetros. Para implementar el nodo en este trabajo modificaremos algunos fragmentos de su código.
- El nodo **slam_gmapping** es un nodo del paquete **gmapping** diseñado por Brian Gerkey [14]. Su función es la de obtener la localización del robot generando el mapa de ocupación.
- **republish** es un nodo que pertenece al paquete **image_transport** diseñado por Patrick Mihelich [15]. Se encarga de descomprimir las imágenes de entrada y publicarlas en formato descomprimido.
- El nodo **darknet_ros** es un nodo de un paquete con el mismo nombre diseñado por Marko Bjelonic [16]. Implementa YOLO en ROS.
- El nodo **write_objects** es un nodo diseñado para este proyecto con el fin de procesar los objetos detectados y escribirlos en un archivo de texto para su posterior análisis.

3.2. Fuente de datos

La mejor manera de comprobar el correcto funcionamiento de todo el conjunto de nodos sería hacerlo directamente sobre el robot, con información real de sensores y del entorno. Sin embargo, debido a la dificultad que supondría utilizar este sistema sobre el robot, se decidió utilizar la herramienta que ROS pone a disposición precisamente para lidiar con este inconveniente: los rosbags.

Para este proyecto se va a utilizar un rosbag generado por uno de los robots del laboratorio de Automática de la Escuela Técnica Superior de Informática. Esta grabación recoge la información de la pose del robot (posición X e Y y el ángulo de orientación), los datos recogidos por una cámara RGB-D (color y profundidad), el barrido de un láser y otra información que típicamente se utiliza en los robots como el árbol de transformadas.

Sin embargo, pese a que este rosbag ha sido el utilizado para comprobar el funcionamiento del algoritmo diseñado, ha sido necesario hacer algunas modificaciones para que se pueda procesar la información correctamente.

Normalmente, cuando se utiliza una cámara RGBD en ROS, el paquete encargado de controlar la cámara envía tres tipos de mensajes pertenecientes al paquete `sensor_msgs`. Estos mensajes son:

- Imagen a color. Información de color de la imagen capturada con un mensaje de tipo `Image`. La mayoría de cámaras ofrece además la imagen rectificada, sin distorsiones.
- Imagen de profundidad. Información sobre la profundidad de la imagen capturada con un mensaje de tipo `Image`. Al igual que con la de color, también se ofrece la imagen rectificada.
- Parámetros de la cámara. Información sobre las propiedades de la cámara con un mensaje de tipo `CameraInfo`. Contiene datos sobre las propiedades de la imagen (ancho y alto) y parámetros de calibración de la cámara.

Debido a algunos problemas que se tuvieron durante la captura de los datos, este último topic no se ofrece en el rosbag, por lo que es necesario generarla externo al rosbag. Para ello, se ha diseñado un nodo llamado `cam_info` que se encarga de generar este mensaje que más tarde será utilizado por otros nodos.

Este nodo, programado en C++, recibe el mensaje de la imagen de profundidad ofrecido por el rosbag, copia su encabezado, crea el mensaje de tipo `sensor_msgs/CameraInfo` con los parámetros de la cámara y lo envía por el topic `/camera_info`.

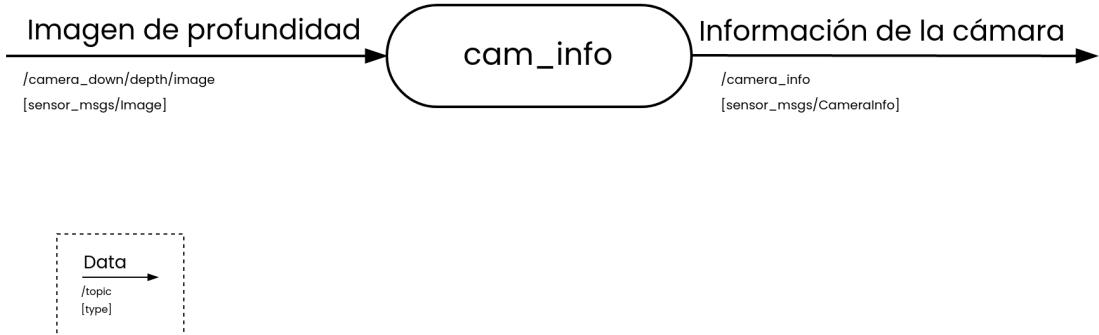


Figura 3.2: Diagrama del nodo `cam_info`

Finalmente, una vez generado el mensaje con la información de los parámetros de la cámara, es necesario sincronizar estos topics para que puedan ser utilizados por otros nodos. Para ello se ha diseñado un nodo en Python llamado `sync_info` que se sirve del paquete `message_filter` para realizar este proceso. Se ha programado en Python porque resulta más fácil utilizar esta librería en este lenguaje que en C++.

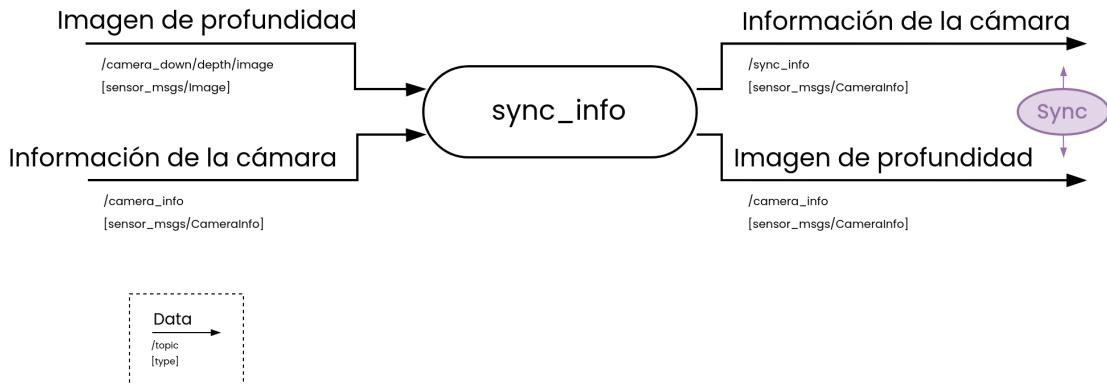


Figura 3.3: Diagrama del nodo `sync_info`

3.3. De imagen de profundidad a barrido láser

Una vez se tiene toda la información ofrecida por el robot y la cámara correctamente sincronizada y configurada, es momento de realizar el procedimiento principal de este proyecto: transformar la imagen de profundidad obtenida mediante la cámara RGB-D en un barrido láser equivalente, cogiendo la distancias más alejadas captadas en la imagen. De esta forma, si tenemos varios objetos delante de una pared, con el

3.3. DE IMAGEN DE PROFUNDIDAD A BARRIDO LÁSER

algoritmo diseñado no se tendrán en cuenta estos objetos y se seleccionarán siempre las distancias menos restrictivas para generar el láser. Es necesario generar un láser artificial porque los algoritmos que se utilizan para generar el mapa de ocupación utilizan este tipo de datos.

El algoritmo utilizado está basado en el paquete de ROS `depthimage_to_laserscan`. Este paquete se encarga de transformar una imagen de profundidad en un barrido láser a partir de los parámetros de la cámara y de una serie de parámetros de entrada configurables. Estos parámetros son:

- `scan_height`. Establece la cantidad de filas que se quieren procesar para generar el láser.
- `scan_time`. Establece el tiempo de actualización entre escaneos. Por defecto está a 0.033 (30 FPS).
- `range_min`. Rango de distancia mínima. Valores medidos menores que este valor se tomarán como -Inf. Por defecto está a 0.45 (metros).
- `range_max`. Rango de distancia máxima. Valores medidos mayores que este valor se tomarán como +Inf. Por defecto está a 10 (metros).
- `output_frame_id`. Establece el id del eje de coordenadas (frame) de salida. Se indica el id del frame del láser.

Este nodo primeramente recibe la imagen de profundidad y la información de la cámara. Se recuerda que estos mensajes los debe recibir a la vez y por eso requería de la sincronización. Una vez recibidos estos mensajes, evalúa la codificación de la imagen. Si la codificación es correcta (admite codificación 16UC1 o 32FC1)¹, procede a convertir la imagen de profundidad a un mensaje de tipo LaserScan.

Un mensaje de tipo LaserScan se forma a partir de las siguientes características:

- `header`. Encabezado del mensaje. Propiedad que tienen todos los mensajes.
- `angle_min`. Ángulo mínimo que abarca el láser en radianes.
- `angle_max`. Ángulo máximo en radianes.
- `angle_increment`. Ángulo entre medidas en radianes.
- `time_increment`. Tiempo entre medidas en segundos.
- `scan_time`. Tiempo entre escaneos en segundos.

¹El paquete `sensor_msgs` prove de multitud de codificaciones para las imágenes. Este es un parámetro de la imagen y viene dado por una cadena de caracteres [17]. El paquete `depthimage_to_laserscan` solo admite las dos codificaciones comentadas

- **range_min.** Rango mínimo del sensor en metros.
- **range_max.** rango máximo del sensor en metros.
- **ranges.** Vector con las medidas tomadas en metros.
- **intensities.** Vector con las intensidades de las medidas. Lo tomaremos como un array vacío porque no es necesario.

El algoritmo se encarga de analizar tantas filas como se establezcan en el parámetro **scan_height** del nodo **depthimage_to_laserscan** y transformar la distancia dada por la imagen de profundidad en una distancia real al sensor. Una vez transformada, se compara con la distancia de las otras filas medidas en la misma columna y, si la distancia es mayor, se añade al vector ranges. Una vez completado el análisis de todas las filas de la imagen, se publica el mensaje generado de tipo **sensor_msgs/LaserScan** en el topic **/new_scan**.

3.4. Generación del mapa de ocupación

Una vez obtenido el láser artificial a partir de la imagen de profundidad, es momento de generar el mapa o rejilla de ocupación. Para este procedimiento se utilizará el paquete **gmapping**.

Este paquete ofrece un sistema de Localización y Mapeado Simultáneo o SLAM (del inglés, *Simultaneous Location and Mapping*) a través de un nodo llamado **slam_gmapping**. Este nodo recibe el láser artificial (por el topic **/new_scan**) y el árbol de transformadas (por el topic **/tf**).

Al lanzar este nodo, es posible establecer multitud de parámetros que determinan el modo de operación. Entre otros, los parámetros que se tendrán en cuenta para ajustar el resultado acorde al objetivo son:

- **linearUpdate.** El nodo procesará un escaneo cada vez que se alcance la distancia lineal determinada por este parámetro. Su valor por defecto es 1,0. El valor óptimo encontrado tras múltiples pruebas es 0,3.
- **angularUpdate.** Determina el incremento de ángulo por el cuál el robot procesará otro escaneo. Su valor por defecto es 0,5. El valor óptimo es 0,7.
- **temporalUpdate.** Establece cada cuánto se procesa un nuevo escaneo. Su valor por defecto y óptimo es 3,0.

Como se comenta en el listado de parámetros, tienen unos valores por defecto que se han ido modificando para obtener un mapa con más fiable. Esto se comentará

más en profundidad en la sección de resultados.

El nodo `slam_gmapping` envía por el topic `/map` mensajes de tipo `nav_msgs-/OccupancyGrid` con información sobre el mapa de forma que, tras finalizar la reproducción del rosbag, el último mensaje enviado por este topic contendrá el mapa completo que se ha generado. Para guardarlo en un formato con el que poder trabajar, se utiliza el nodo `map_saver` del paquete `map_server`, diseñado por Brian Gerkey y Tony Pratkanis. Simplemente se lanza el nodo dándole como parámetro el nombre con el que se va a guardar el mapa. Finalmente se obtienen dos archivos: un `.yaml`, para la configuración, y otro en formato `.pgm` para la imagen.

3.5. Detección de objetos

Paralelamente, se ha diseñado un sistema capaz de interpretar la imagen a color proporcionada por el sensor RGBD para detectar los objetos y tener un registro de los mismos. Este sistema, en principio, se puede lanzar simultáneamente junto con el sistema de generación de mapas de ocupación, sin embargo, debido a la potencia necesaria para ejecutar la red neuronal por GPU, resulta tedioso e incluso problemático con algunos ordenadores. Es por ello que se ha diseñado para que pueda ser lanzado de forma asíncrona con respecto al otro sistema.

3.5.1. Descompresión de la imagen de color

En muchas ocasiones, los paquetes de ROS que sirven para utilizar determinadas marcas de sensores RGBD proporcionan las imágenes en un formato comprimido para ahorrar potencia. Con el rosbag que se ha utilizado como fuente de datos ocurre lo mismo, por lo que es necesario un proceso de descompresión.

Para este procedimiento, se ha utilizado un nodo llamado `republish` del paquete `image_transport`. Este nodo se recibe un mensaje en formato `sensor_msgs-/CompressedImage` y lo convierte en un mensaje de tipo `sensor_msgs/Image` para publicarlo.

3.5.2. Ejecución de YOLO

Una vez la imagen está descomprimida, ya puede ser recibida por la red neuronal. Este nodo recibe la imagen de color en formato `sensor_msgs/Image` y publica en tres topics:

- `object_detector`. Mensaje de tipo `std_msgs/Int8` que indica el número de objetos detectados.

- **bounding_boxes**. Mensaje de tipo `darknet_ros_msgs/BoundingBoxes` que representa un array que proporciona información sobre la posición y el tamaño de los bounding boxes en píxeles.
- **detection_image**. Mensaje de tipo `sensor_msgs/Image` que ofrece las bounding boxes detectadas sobre la imagen a color que ha sido procesada.

Las bounding boxes son unas “cajas” que señalizan al elemento detectado. Un mensaje de tipo `darknet_ros_msgs/BoundingBoxes` es un array de elementos con formato `darknet_ros_msgs/BoundingBox`.

Además, este nodo ofrece una salida en la terminal indicando los frames por segundo (FPS) a los que se está ejecutando y un listado de los objetos encontrados junto con sus probabilidades, como se muestra en la figura 3.4. YOLO tiene un umbral de probabilidad establecido por defecto en 0,7 para aceptar una detección como válida o no.

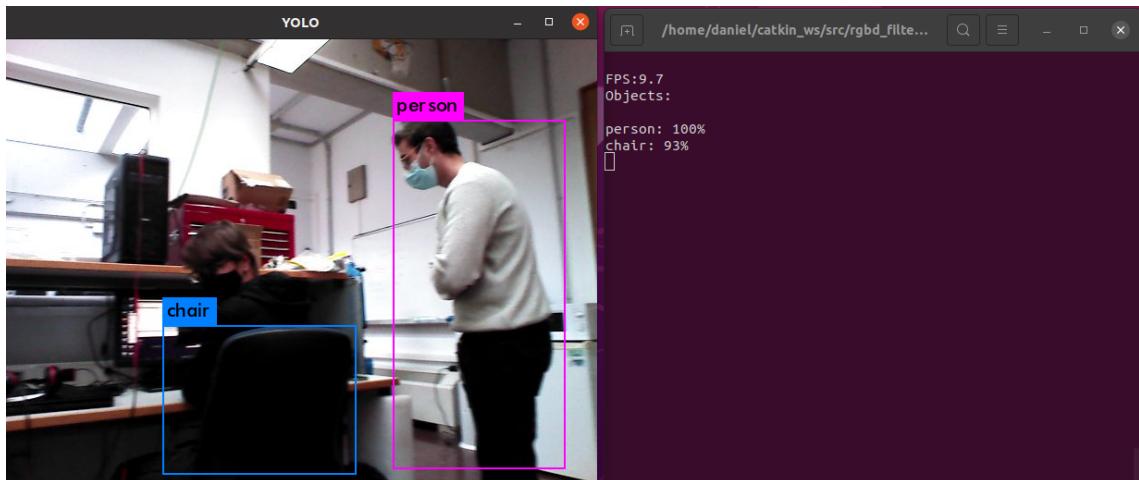


Figura 3.4: Salida en la terminal del nodo `darknet_ros`

[Cabe mencionar que para utilizar YOLO con CUDA correctamente ha sido necesario un largo y tedioso proceso de configuración que se explicará en el Anexo X.]

3.6. Procesamiento de los objetos detectados

Como se ha comentado, a partir del nodo que implementa YOLO obtenemos un array de bounding boxes con información sobre la posición y el tamaño en píxeles. Para el desarrollo de este proyecto se ha visto conveniente procesar esa información para realizar un posterior estudio de los objetos que han sido detectados y analizar

3.7. ANÁLISIS DE LOS OBJETOS DETECTADOS

la fiabilidad de la red neuronal así como la calidad de los datos proporcionado por el rosbag.

Para este procesamiento se ha diseñado un nodo llamado `write_objects` que se encarga de recibir los mensajes que se publican en `/darknet_ros/bounding_boxes` y en `/amcl_pose` para generar un archivo CSV donde cada fila representa la información con el formato mostrado en el cuadro 3.1. Cada columna utilizará el delimitador de punto y coma (“ ; ”).

Objetos	Probs	Posición	Orientación	Tiempo
obj1:obj2:...:objN	prob1:prob2:...:probN	pX:pY:pZ	oW:oX:oY:oZ	(seg)

Cuadro 3.1: Formato del archivo CSV generado

Toda esta información es obtenida a partir del formato de los mensajes de las bounding boxes y de la pose. Para las bounding boxes es necesario acceder a cada una de ellas y obtener sus propiedades `Class` y `probability`. Para la pose es necesario acceder a su propiedades `pose.pose.position` y `pose.pose.orientation`.

El formato del mensaje de una bounding box tiene las siguientes propiedades:

- **probability**. Probabilidad de que el elemento detectado sea realmente de la clase que se le atribuye.
- **xmin**. Coordenada X mínima de la bounding box en píxeles.
- **ymin**. Coordenada Y mínima de la bounding box en píxeles.
- **xmax**. Coordenada X máxima de la bounding box en píxeles.
- **ymax**. Coordenada Y máxima de la bounding box en píxeles.
- **id**. Identificador único para cada bounding box.
- **Class**. Clase o etiqueta del elemento detectado.

3.7. Análisis de los objetos detectados

Una vez obtenido el registro de los objetos en formato CSV es posible procesar esta información para realizar un análisis de la red y del entorno. Para ello se utilizará el entorno de programación en Python de *Google Colab*, junto con las librerías *Pandas* y *Matplotlib*. La información sobre las gráficas y los resultados obtenidos se exponen en el capítulo siguiente.

Capítulo 4

Resultados obtenidos

En este capítulo se mostrarán los resultados que se han obtenido tras implementar el sistema. Primeramente se expondrá el mapa generado con la imagen de profundidad y los mapas obtenidos utilizando diferentes parámetros. Luego se hará una comparativa con el resultado obtenido a partir del láser 2D incorporado en el robot. En una segunda parte se mostrarán algunas gráficas realizadas con el registro de objetos detectados y se expondrán los posibles errores de la red neuronal en el proceso de la detección.

4.1. Mapa generado a partir de la imagen de profundidad

En la figura 4.1 se muestra el resultado de generar el mapa a partir del láser 2D (izquierda) y a partir de la imagen de profundidad captada por el sensor RGBD (derecha).

En el mapa generado con el láser se pueden ver los entrantes y salientes se forman debido a la presencia de mobiliario delante de las paredes que impide obtener una medida real de la estructura. Además, en zonas interiores existen muchos grupos pequeños de celdas ocupadas causadas por la presencia de patas de mesas, sillas y otros elementos de menor tamaño. Sin olvidar que la grabación cuenta con la presencia de personas que modifican su posición, complicando aún más la generación del mapa.

Con el mapa obtenido gracias al procesamiento de la imagen de profundidad, se puede comprobar como los elementos situados entre el sensor y las paredes no se tienen en cuenta, quedando el interior del mapa totalmente “limpio”. Además, las zonas donde se sitúa el mobiliario quedan mucho más suavizadas, mostrándose únicamente la pared situada tras el mismo. En la figura 4.2 se muestran los dos mapas

CAPÍTULO 4. RESULTADOS OBTENIDOS

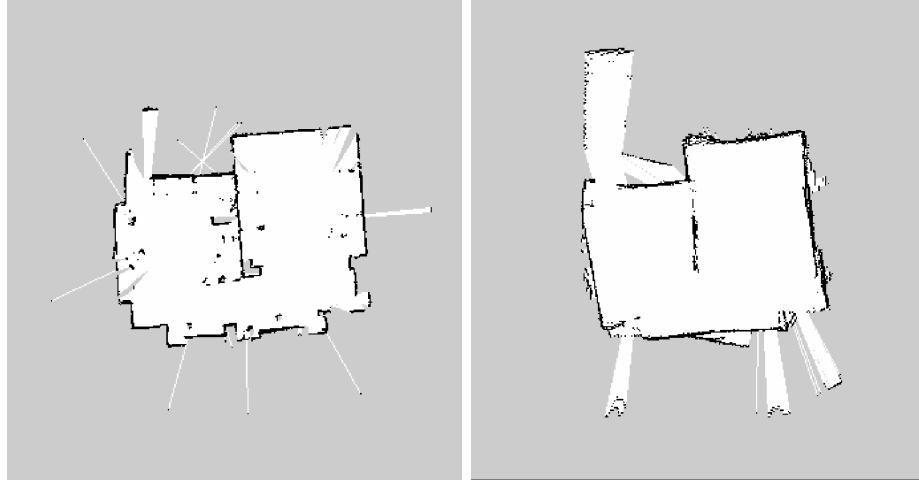


Figura 4.1: Mapas generados con láser 2D (a) y con sensor RGBD (b)

superpuestos para poder ver mejor la diferencia entre sendos mapas.

Pese a que mejora al láser en muchos aspectos, el mapa generado con la cámara tiene numerosos problemas. Como se puede observar, las paredes no son totalmente rectas sino que tienen curvatura, lo que provoca que las esquinas no formen 90°. Incluso hay paredes, como la que está situada más a la izquierda, que parecen desviarse con respecto al mapa global.

Estas curvaturas obtenidas en la medición puede ser causada por la calibración intrínseca de la cámara. Es posible que algunos parámetros de la cámara no estén correctamente determinados o incluso que la transformada entre la base del robot y la cámara no esté correctamente establecida. Por la curvatura en las esquinas se intuye que la causa más probable sea una mala calibración de los parámetros de distorsión de la cámara.

Este resultado ya se podía percibir observando el láser generado a partir del sensor RGBD en comparación con el del láser 2D durante el proceso de construcción del mapa.

4.1.1. Generación del láser artificial

Como se ha comentado previamente, el láser que se genera a partir de la imagen de profundidad tiene unas curvaturas que ya se podían percibir claramente durante el proceso de generación del mapa. En muchas ocasiones, el láser artifical generado no se superpone con el láser real en las situaciones en las que debería hacerlo, como

4.1. MAPA GENERADO A PARTIR DE LA IMAGEN DE PROFUNDIDAD

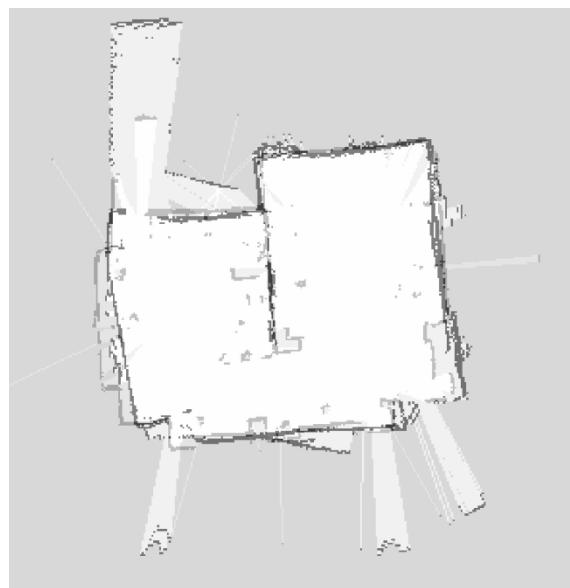


Figura 4.2: Superposición de los dos mapas

las de las figuras 4.3 y 4.4.

En otras muchas ocasiones, como en las de las figuras 4.5 e 4.6, se puede ver como el procesamiento realiza correctamente su función, cumpliendo el objetivo inicialmente establecido de “eliminar” los objetos frente a las paredes.

CAPÍTULO 4. RESULTADOS OBTENIDOS

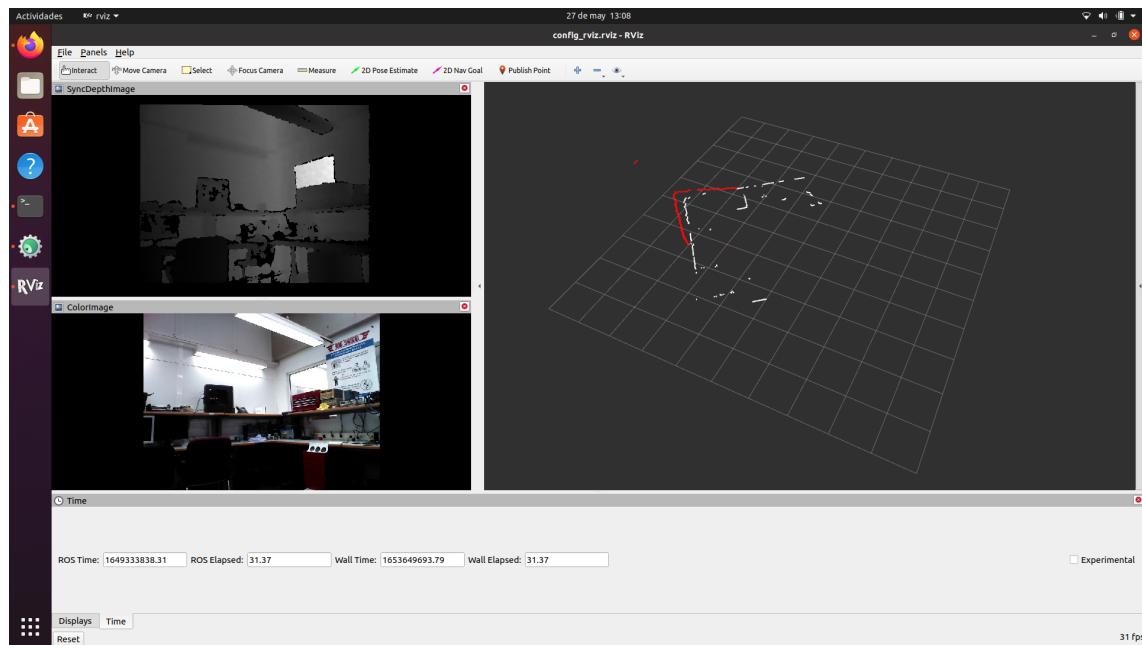


Figura 4.3: Muestra de medida 1: diferencia entre las medidas del láser real (blancas) y del láser artificial (rojas)

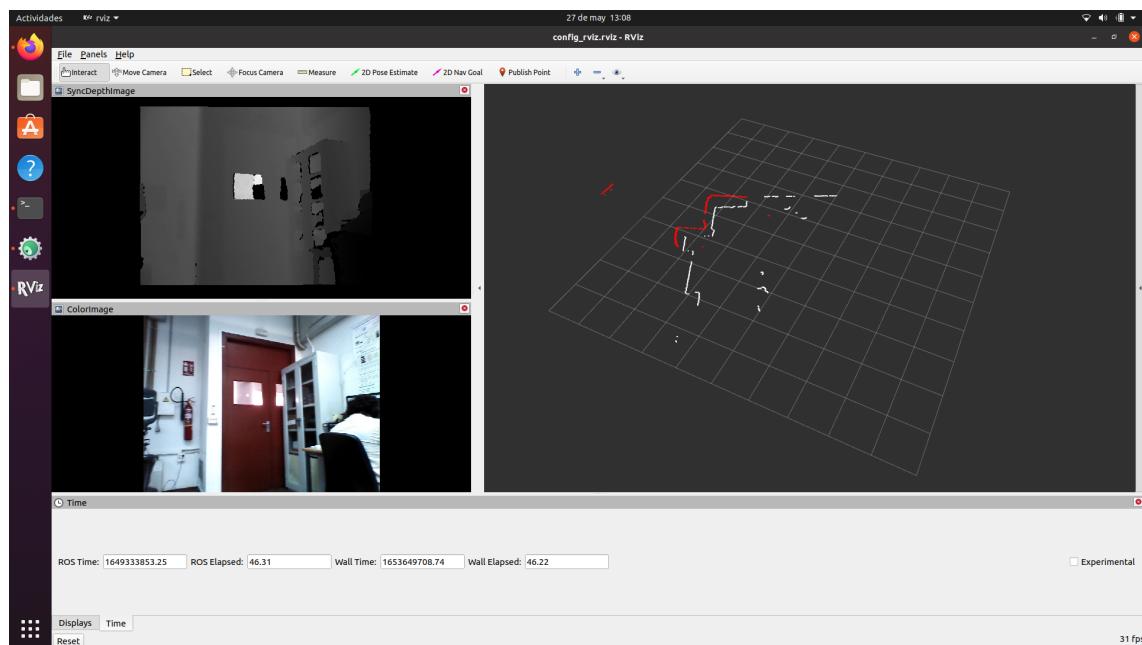


Figura 4.4: Muestra de medida 2: diferencia entre las medidas del láser real (blancas) y del láser artificial (rojas)

4.1. MAPA GENERADO A PARTIR DE LA IMAGEN DE PROFUNDIDAD

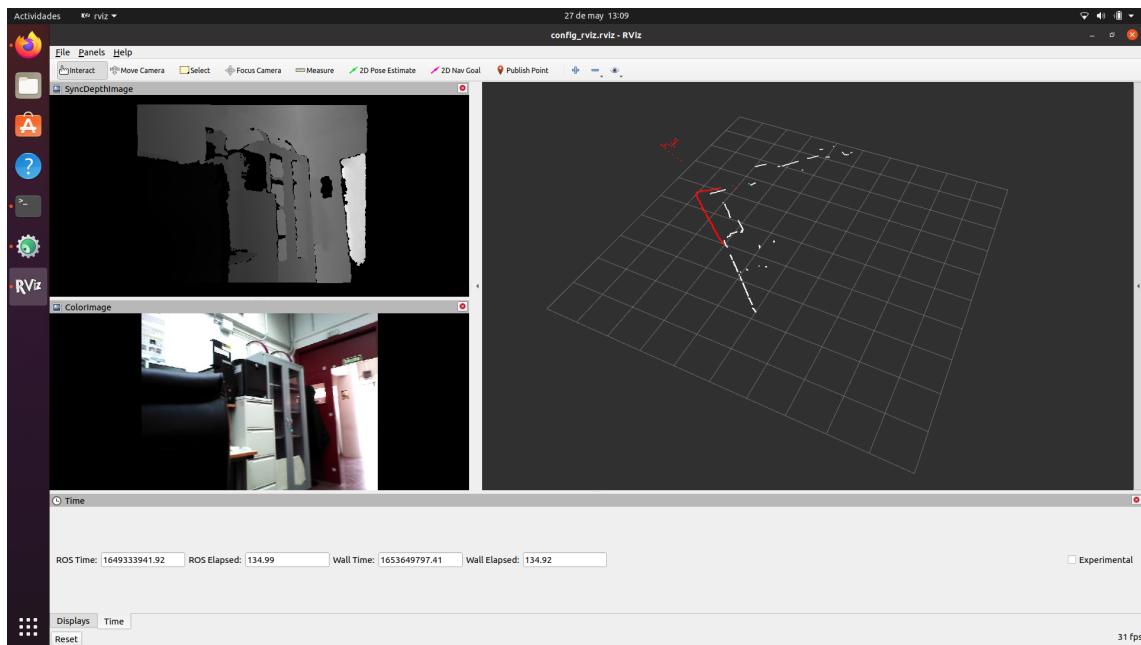


Figura 4.5: Muestra de medida 3: diferencia entre las medidas del láser real (blancas) y del láser artificial (rojas)

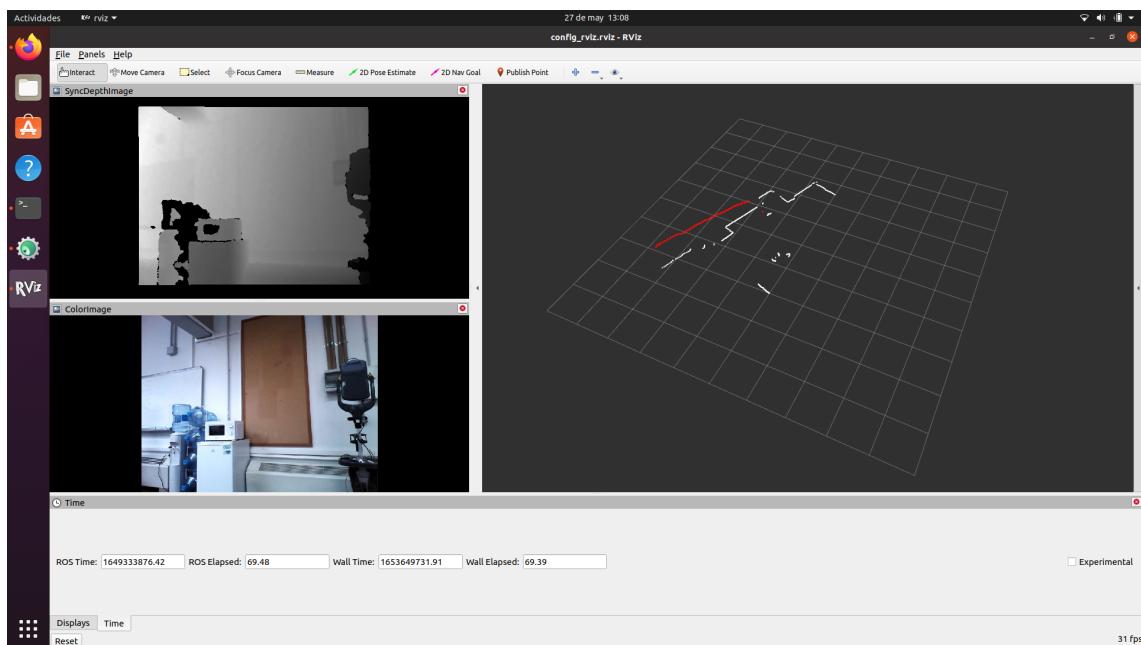


Figura 4.6: Muestra de medida 4: diferencia entre las medidas del láser real (blancas) y del láser artificial (rojas)

4.1.2. Modificación de parámetros por defecto

Como se comentó en la sección 3.4, el nodo `slam_gmapping` tiene una serie de parámetros que permiten ajustar cada cuántos metros, ángulos o segundos se procesa la información del láser.

Tras una primera ejecución del procedimiento diseñado, el resultado obtenido no fue el esperado. Es por ello que se decidió variar estos parámetros hasta obtener un resultado satisfactorio. Los mapas generados para diferentes valores de estos tres parámetros se muestran en la figura 4.7.

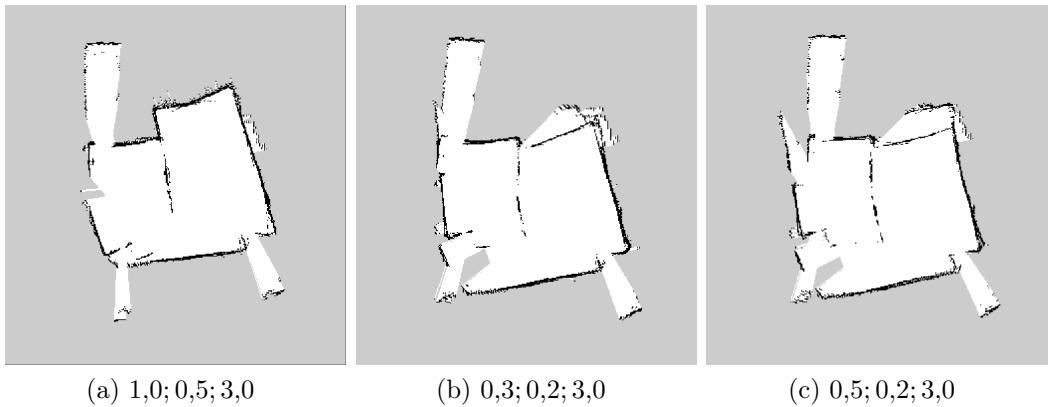
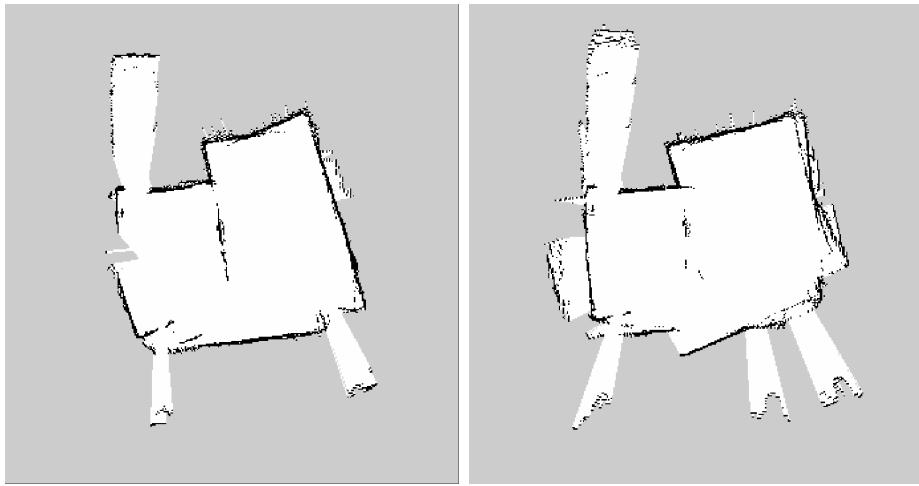


Figura 4.7: Resultados para diferentes parámetros de `linearUpdate`, `angularUpdate` y `temporalUpdate`, respectivamente

Otro parámetro que se ha cambiado para obtener mejores resultados es la distancia máxima de profundidad, `range_max`, del nodo `depthimage_to_laserscan` comentada en el apartado 3.3. Este valor está por defecto a 10 metros. Sin embargo, debido a la existencia de amplias ventanas y puertas, el resultado se veía muy influenciado por las distancias alejadas captadas a través de estas. El valor que ha originado mejores resultados es 7 metros.

4.2. ANÁLISIS DE LA FIABILIDAD DE LAS DETECCIONES



(a) `range_max` a 7 metros

(b) `range_max` a 8 metros

Figura 4.8: Resultados para distintos valores de distancia máxima

4.2. Análisis de la fiabilidad de las detecciones

YOLO es una de las redes neuronales más rápidas en el ámbito de la detección y localización de objetos en una imagen. Sin embargo, esta velocidad tan característica le hace sacrificar algo de exactitud en sus medidas, convirtiéndola en una red no del todo fiable. Aún así, continúa siendo de las mejores en este aspecto.

Con el nodo `write_objects` obteníamos un archivo de texto con los datos en formato CSV, un formato que permite procesar la información de manera sencilla. Este archivo contiene información sobre los objetos o clases detectadas, sus probabilidades, la pose del robot (posición y orientación) y el tiempo en el instante en que se detectaron dichos objetos. Para entender la trayectoria seguida por el robot, se propone la gráfica de la figura 4.9. Destacar que no comienza en el origen de coordenadas porque existe cierto desfase entre la ejecución de YOLO y los demás nodos, debido a la potencia necesaria para lanzarlo.

En la figura 4.10 se muestra una gráfica de barras con el número de ocasiones que fueron detectadas cada una de las clases. Las clases ‘tvmonitor’ (monitor de TV) y ‘chair’ (silla) son las que más han sido detectadas, algo coherente con la distribución del laboratorio. La tercera clase más detectada, ‘person’ (persona), tampoco se escapa de la normalidad ya que durante la grabación había personas en el laboratorio. Las clases ‘mouse’ (ratón) y ‘keyboard’ (teclado), en principio, deberían haber sido tan frecuentes como las otras. Sin embargo, la cámara está prácticamente al mismo nivel que las mesas, lo que dificulta en gran medida el reconocimiento de estos objetos.

CAPÍTULO 4. RESULTADOS OBTENIDOS

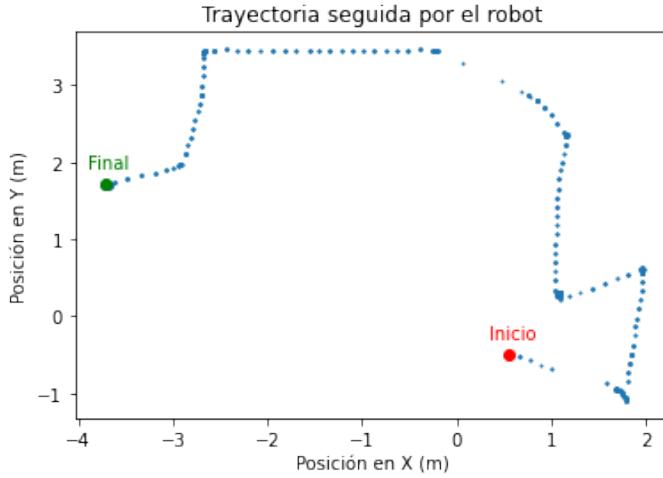


Figura 4.9: Trayectoria seguida por el robot. El punto rojo indica el inicio de la trayectoria. El punto verde indica el final

De todos los objetos que se han detectado, hay tres que no parecen pertenecer al entorno habitual de un laboratorio. Es el caso de ‘oven’ (hornos), ‘refrigerator’ (frigorífico) y ‘microwave’ (microondas). Este último es coherente ya que en el laboratorio hay uno y las detecciones las hace correctamente. Los otros son casos especiales que se explicarán a continuación.

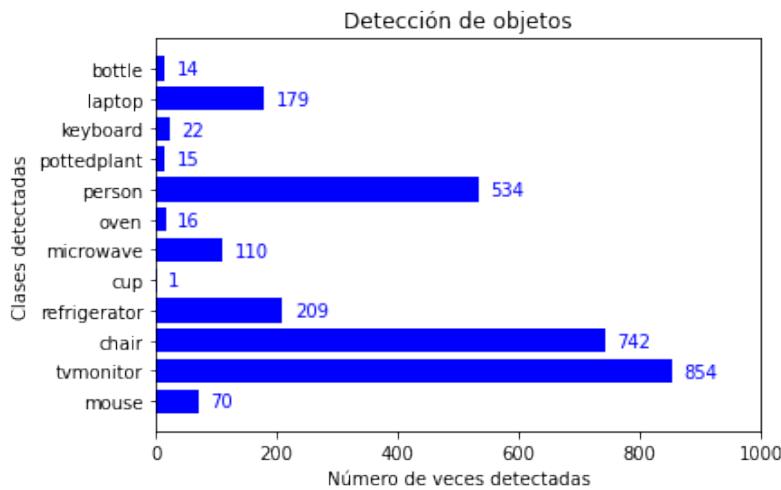


Figura 4.10: Gráfica de objetos detectados. Muestra el número de veces que fue detectado cada uno de los objetos

En este laboratorio no hay ningún horno, por lo que la detección de esta clase es errónea. Como se muestra en la figura 4.11, ‘oven’ solo se detecta en un momento durante la trayectoria. Revisando las imágenes del dataset, se puede ver lo que YO-

4.2. ANÁLISIS DE LA FIABILIDAD DE LAS DETECCIONES

YOLO confunde con un horno, mostrado en la figura 4.12. Ciertamente, el objeto que detecta es muy parecido a los diales de control de un horno, por lo que es razonable la ocnfusion.

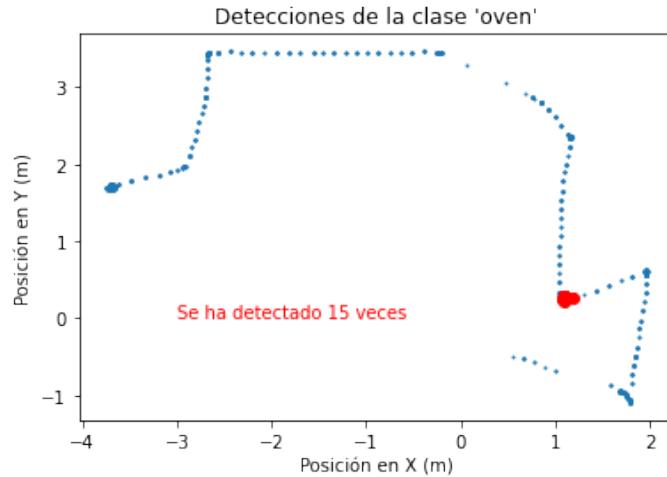


Figura 4.11: Momentos en los que se ha detectado la clase ‘oven’. Los puntos azules representan la trayectoria del robot, los rojos las veces que se detectó la clase

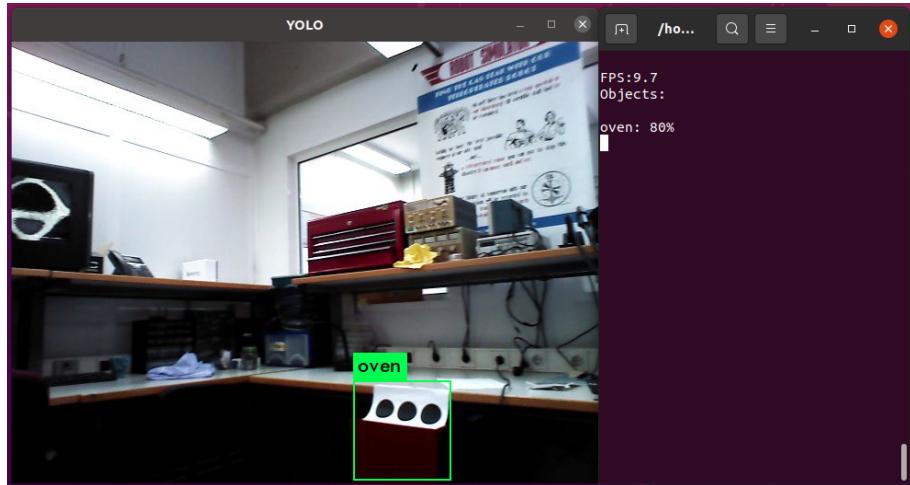


Figura 4.12: Detección de la clase ‘oven’ en la imagen

En cambio, en el laboratorio sí que hay un frigorífico. Concretamente, un frigorífico compacto de una puerta. Sin embargo, pese a solamente haber uno, aparece en muchas ocasiones. Si obtenemos las posiciones en las que se detecta (figura 4.13), se puede ver como lo hace en posiciones muy distintas. Analizando las imágenes del dataset, se puede ver como, efectivamente, YOLO hace detecciones erróneas de la clase ‘refrigerator’. En la figura 4.14 se muestran dos detecciones erróneas y una

CAPÍTULO 4. RESULTADOS OBTENIDOS

correcta. Se puede ver como, en las detecciones incorrectas, los objetos detectados son similares a un frigorífico convencional.

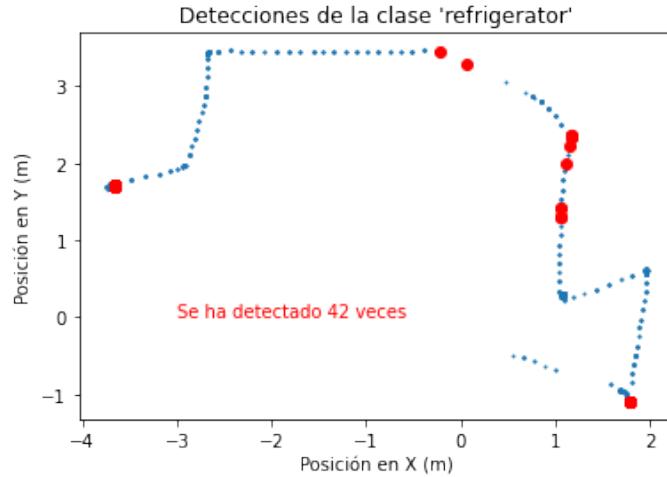


Figura 4.13: Momentos en los que se ha detectado la clase 'refrigerator'. Los puntos azules representan la trayectoria del robot, los rojos las veces que se detectó la clase

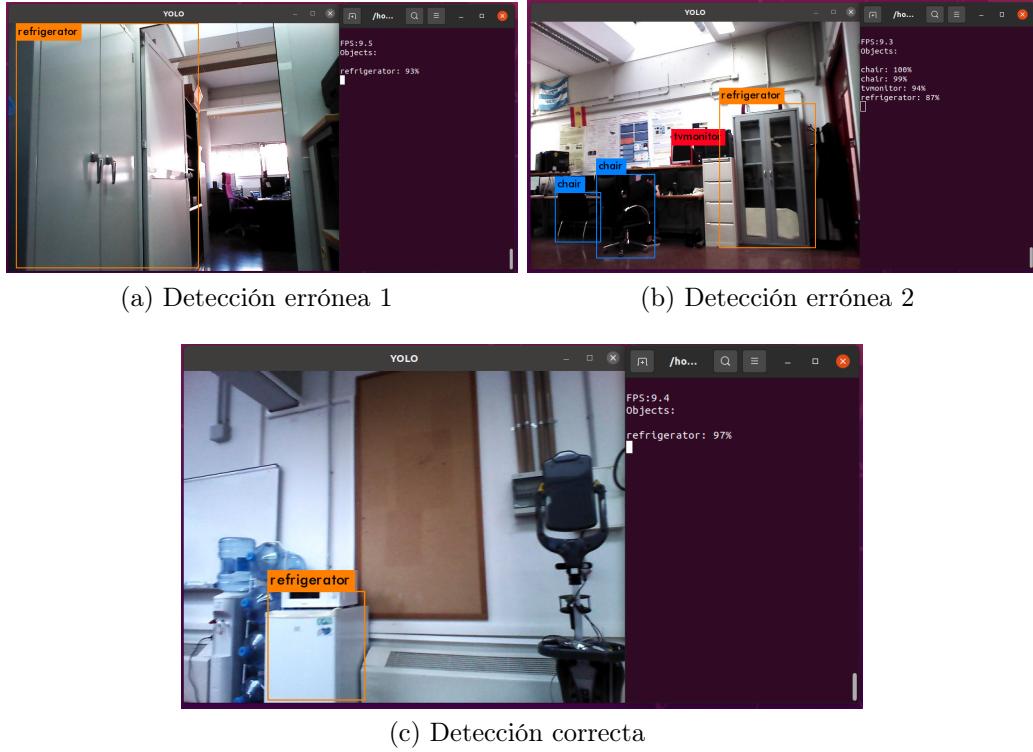


Figura 4.14: Detecciones de la clase 'refrigerator' en la imagen

Parte III

Conclusiones

Capítulo 5

Conclusiones, propuestas y líneas futuras

En este capítulo se pretende, a partir de los resultados obtenidos y de todo el proceso llevado a cabo para conseguirlos, exponer las conclusiones a las que se han llegado. Además se expondrán algunas propuestas de cara a mejorar el procedimiento. Finalmente se trazarán posibles líneas de trabajo de cara al desarrollo futuro de este trabajo.

5.1. Conclusiones

Los láseres 2D son unos sensores que se caracterizan principalmente por su gran precisión y amplio campo de visión. Sin embargo, son elementos que encarecen en gran medida el presupuesto a dedicar para el diseño de un robot. Las cámaras RGBD, en cambio, son sensores mucho más asequibles, por lo que se deberían de tener en cuenta a la hora de generar *floorplans* de la estructura del edificio o vivienda.

Como se ha mostrado en el capítulo 4, el resultado obtenido queda lejos lo ideal. El mapa obtenido mejora al láser en cuanto a la eliminación de los elementos no estructurales pero perjudica seriamente la rectitud de las líneas, provocando que el mapa pierda fiabilidad. Esto es debido a que los sensores RGBD son mucho menos precisos a la hora de calcular distancias, de ahí su gran diferencia de precio. Además, el dataset utilizado para comprobar el funcionamiento (el laboratorio de automática) no es el más amigable para este procedimiento debido a la gran cantidad de elementos que perjudican la medición a través de la cámara. Aún así, el resultado deja entrever que, con una correcta configuración y unas mejores condiciones del entorno, sería posible obtener un correcto *floormap* estructural.

Por tanto, las conclusiones extraídas del desarrollo de este proyecto son las siguientes.

- Las cámaras RGBD pueden llegar a ser un sustituto válido y asequible a la hora de generar `floorplans` que representen exclusivamente la estructura del edificio, sin elementos cambiantes como personas, muebles, etc.
- El entorno es el factor más determinante de cara a la efectividad del método. Un entorno con muchos elementos transparentes o con entrantes y salientes perjudican en gran medida los resultados obtenidos.
- La calibración de la cámara es muy importante si se quieren obtener resultados que puedan ser utilizados en la realidad.
- La comunidad de ROS es una de las más activas que se pueden encontrar. Existen miles de usuarios que aportan paquetes con funciones que abarcan prácticamente todos los ámbitos de la robótica. Por lo tanto, antes de comenzar a desarrollar alguna función, mejor comprobar si alguien ya lo ha hecho, algo que, vista la inmensitud de su comunidad, será lo más probable. Además, ROS cuenta con su propio foro donde compartir problemas y sus soluciones, ayudando en gran medida a usuarios novatos.
- Las redes neuronales como YOLO, entrenadas para la detección de objetos, pese a no ser del todo fiable, cumplen con creces su función. Sin embargo, muchos de los elementos del dataset no ha sido capaz de detectarlos.
- El manejo de datos con *Python*, *Pandas* y *Numpy* resulta realmente sencillo. Complementándolo con la librería *Matplotlib* se convierte en una poderosa herramienta para el análisis y la representación de los datos.
- *Linux*, pese a resultar inicialmente más complicado de comprender que otros sistemas operativos como *Windows* o *MacOS*, tiene numerosas características que lo convierten en una herramienta muy potente para el desarrollo, como la instalación por paquetes o el debugging.

5.2. Propuestas y líneas futuras

Hay aspectos de este trabajo que, una vez analizado de principio a fin, pudieran haberse hecho de otra forma y así, quizá, haber obtenido mejor resultado.

El dataset de datos (rosbag del laboratorio) no es el idóneo para esta tarea. Debido a todos sus entrantes y salientes y a las ventanas en la pared divisoria entre las dos estancias, el resultado obtenido no es del todo correcto. Lo mejor hubiera sido grabar al robot en un ambiente de vivienda convencional, con habitaciones, baños y cocinas. Además, la posición y orientación de la cámara no es la idónea. Dado que los elementos se encuentran mayoritariamente en la parte inferior de las paredes, lo ideal sería situar la cámara en una posición más alta que la del dataset, con una

5.2. PROPUESTAS Y LÍNEAS FUTURAS

orientación paralela al plano del suelo.

YOLO utiliza una serie de pesos para detectar los objetos. Una interesante línea futura para este proyecto sería investigar los distintos datasets de pesos disponibles para YOLO y cuáles convendría utilizar en un entorno de laboratorio de ingenierías como con el que se ha trabajado.

Dado que el principal problema de los resultados obtenidos es la curvatura acaecida en las paredes medidas, sería conveniente buscar los parámetros que se ajustan concretamente a la cámara utilizada para generar el dataset. Posiblemente, una correcta configuración de los parámetros intrínsecos de la cámara, generará una mejoría en los resultados.

Parte IV

Apéndices

Apéndice A

Otras herramientas utilizadas

En este apéndice se expondrán las herramientas que se han utilizado para desarrollar el trabajo que no se explican en los capítulos anteriores pero que han sido necesarias poder alcanzar el objetivo. Principalmente se explicarán brevemente las librerías que se han utilizado en C++ y Python.

A.1. Librerías y paquetes

A.1.1. *message_filters*

Es un paquete de ROS que proporciona una serie de filtros capaces de recibir mensajes y enviarlos pasado un tiempo determinado en función de la necesidad dictaminada por el filtro. Está diseñado por Josh Faust, Vijay Pradeep y Dirk Thomas.

Para este proyecto se ha utilizado el filtro de sincronización o *time synchronizer*. Este filtro se encarga de recibir mensajes de distintos topics y enviarlos sólo cuando se haya recibido un mensaje de cada uno de los topics con la misma marca temporal (*timestamp*). Concretamente se utiliza en el nodo `sync_info`, para sincronizar los mensajes de la imagen de profundidad y de los parámetros de la cámara.

A.1.2. *fstream*

Es una librería de C++ que permite la escritura en archivos. Se ha utilizado en el nodo `write_objects` para escribir la información recibida de la pose y de la detección de objetos.

A.1.3. *libfreenect*

libfreenect es una librería que permite el acceso USB a la cámara Kinect de Microsoft. Es parte del proyecto *OpenKinect*, una comunidad abierta que trabaja para implementar la posibilidad de uso de este dispositivo en todos los ordeanadores. Concretamente, se ha utilizado el paquete `freenect_launch` que permite lanzar la cámara desde ROS.

Este paquete sirvió para tener una primera toma de contacto con los sensores RGBD, las imágenes de profundidad y las nubes de puntos.

A.2. RVIZ

RVIZ (ROS Visualization) es un visualizador 3D para ROS. Es un software que se instala junto con el framework. Su uso está prácticamente estandarizado en toda la comunidad de ROS.

Se utiliza para visualizar los mensajes que llegan a los topics. Permite visualizar poses, imágenes (en diferentes codificaciones), mapas de ocupación, orientaciones, etc. e incluso permite incluir modelos 3D personalizados.

A.3. Google Colab

Colab, también conocido como “Colaboratory”, es una herramienta creada por Google que permite a los usuarios programar y ejecutar código Python en el navegador, utilizando servicios alojados en la nube. Esta herramienta se ha utilizado para el análisis de los objetos detectados con la red neuronal. Las ventajas de utilizar esta plataforma son:

- No requiere configuración
- Da acceso gratuito a GPUs
- Permite compartir contenido fácilmente

Apéndice B

Código de los nodos diseñados

En este apéndice se expone el código de los nodos diseñados para este proyecto. Estos nodos son: `cam_info`, `sync_info` y `write_objects`.

B.1. Nodo `cam_info`

`cam_info.hpp`

```
1 #include "ros/ros.h"
2 #include "sensor_msgs/Image.h"
3 #include "sensor_msgs/CameraInfo.h"
4
5 class CamInfo{
6
7 public:
8     CamInfo();
10    void callback(const sensor_msgs::Image::ConstPtr & msg);
12
13 private:
14     ros::NodeHandle nh;
15     ros::Publisher pub;
16     ros::Subscriber sub;
18
19 };
```

Listing B.1: Código del archivo `cam_info.hpp`

`cam_info.cpp`

APÉNDICE B. CÓDIGO DE LOS NODOS DISEÑADOS

```
1 #include "rgbd_filter/cam_info.hpp"
2
3 CamInfo::CamInfo(){
4
5     sub = nh.subscribe("camera_down/depth/image", 1000, &CamInfo::
6         callback, this);
7     pub = nh.advertise<sensor_msgs::CameraInfo>("camera_info", 1000)
8     ;
9 }
10 void CamInfo::callback(const sensor_msgs::Image::ConstPtr& msg){
11
12     sensor_msgs::CameraInfo cam_msg;
13
14     cam_msg.header = msg->header;
15     cam_msg.height = msg->height;
16     cam_msg.width = msg->width;
17     cam_msg.distortion_model = "plumb_bob";
18     cam_msg.D = {0.0, 0.0, 0.0, 0.0, 0.0};
19     cam_msg.K = {579.385009765625, 0.0, 318.2640075683594, 0.0,
20     579.385009765625, 234.93800354003906, 0.0, 0.0, 1.0};
21     cam_msg.R = {1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0};
22     cam_msg.P = {579.385009765625, 0.0, 318.2640075683594, 0.0,
23     0.0, 579.385009765625, 234.93800354003906, 0.0, 0.0, 0.0, 1.0,
24     0.0};
25     cam_msg.binning_x = 0;
26     cam_msg.binning_y = 0;
27     cam_msg.roi.do_rectify = false;
28     cam_msg.roi.x_offset = 0;
29     cam_msg.roi.y_offset = 0;
30     cam_msg.roi.height = 0;
31     cam_msg.roi.width = 0;
32 }
33 }
```

Listing B.2: Código del archivo `cam_info.cpp`

main_cam_info.cpp

```
1 #include "rgbd_filter/cam_info.hpp"
2
3 int main(int argc, char** argv){
4
5     ros::init(argc, argv, "cam_info");
6
7     CamInfo cam_info;
8
9     ros::spin();
```

```
10 }
11 }
```

Listing B.3: Código del archivo main_am_info.cpp

B.2. Nodo sync_info

```
1 #!/usr/bin/env python
2
3 import rospy
4 import message_filters
5 from sensor_msgs.msg import Image, CameraInfo
6
7 class SyncInfo:
8
9     def __init__(self):
10
11         self.depth_topic = "camera_down/depth/image"
12         self.info_topic = "camera_info"
13
14         self.pub_depth = rospy.Publisher('sync/depth_image', Image,
15                                         queue_size=10)
15         self.pub_info = rospy.Publisher('sync/camera_info',
16                                         CameraInfo, queue_size=10)
16
17         sub_depth = message_filters.Subscriber(self.depth_topic,
18                                               Image)
18         sub_info = message_filters.Subscriber(self.info_topic,
19                                               CameraInfo)
20
20         message_filter = message_filters.
21         ApproximateTimeSynchronizer([sub_depth, sub_info], 10, 0.5)
21         message_filter.registerCallback(self.callback)
22
23
24     def callback(self, depth_msg, info_msg):
25
26         self.pub_depth.publish(depth_msg)
27         self.pub_info.publish(info_msg)
28
29
30 if __name__ == '__main__':
31     rospy.init_node('sync_info', anonymous=True)
32     myNode = SyncInfo()
33     rospy.spin()
```

Listing B.4: Código del archivo sync_info.py

B.3. Nodo write_objects

write_objects.hpp

```
1 #include <ros/ros.h>
2
3 #include <string>
4 #include <iostream>
5 #include <fstream>
6
7 #include <sensor_msgs/Image.h>
8 #include <geometry_msgs/PoseWithCovarianceStamped.h>
9 #include <geometry_msgs/PoseWithCovariance.h>
10 #include <geometry_msgs/Pose.h>
11 #include <geometry_msgs/Point.h>
12 #include <geometry_msgs/Quaternion.h>
13 #include <darknet_ros_msgs/BoundingBoxes.h>
14 #include <darknet_ros_msgs/BoundingBox.h>
15
16 using namespace sensor_msgs;
17 using namespace darknet_ros_msgs;
18
19 class WriteObjects{
20
21 public:
22
23     WriteObjects();
24
25     void cb_bboxes(const BoundingBoxes::ConstPtr& msg);
26     void cb_pose(const geometry_msgs::
27         PoseWithCovarianceStampedConstPtr & msg);
28
29
30
31 private:
32
33     ros::NodeHandle nh;
34     ros::Subscriber sub_bboxes;
35     ros::Subscriber sub_pose;
36
37     std::string position;
38     std::string orientation;
39
40     std::string str_objetos;
41     std::string str_probs;
42
43     double start_time;
44
```

```
46 };
```

Listing B.5: Código del archivo write_objects.hpp

write_objects.cpp

```

1 #include "rgbd_filter/write_objects.hpp"
2
3 using namespace sensor_msgs;
4 using namespace darknet_ros_msgs;
5
6 WriteObjects::WriteObjects(){
7
8     sub_bboxes = nh.subscribe("/darknet_ros/bounding_boxes", 1, &
9     WriteObjects::cb_bboxes, this);
10    sub_pose = nh.subscribe("/amcl_pose", 1, &WriteObjects::cb_pose
11    , this);
12
13    position = "";
14    orientation = "";
15    str_objetos = "";
16    str_probs = "";
17
18 }
19
20
21 void WriteObjects::cb_bboxes(const BoundingBoxes::ConstPtr& msg){
22
23     std::vector<BoundingBox> v = msg->bounding_boxes;
24     int n = v.size();
25
26     str_objetos = "";
27     str_probs = "";
28
29     // Si hay objetos detectados ->
30     if(n != 0){
31
32         // Iteramos en cada uno de los objetos y lo anadimos a un
33         // array junto con su probabilidad
34         for(int i=0; i<n; i++){
35             str_objetos.append(v[i].Class);
36             str_objetos.append(":");
37             str_probs.append(std::to_string(v[i].probability));
38             str_probs.append(":");
39         }
40
41         // Eliminamos la ultima ","
42         str_objetos = str_objetos.substr(0, str_objetos.size()-1);
43         str_probs = str_probs.substr(0, str_probs.size()-1);

```

APÉNDICE B. CÓDIGO DE LOS NODOS DISEÑADOS

```
43     // Obtenemos el tiempo de simulacion
44     double time = ros::Time::now().toSec() - start_time;
45
46     // Escribimos en el archivo
47     std::ofstream myfile;
48     myfile.open("/home/daniel/Documentos/objetos.txt", std::ios
49 ::ate | std::ios::app);
50     myfile << str_objetos << ";" << str_probs << ";" <<
position << ";" << orientation << ";" << std::to_string(time) <<
"\n";
51     myfile.close();
52 }
53
54 }
55 }
56
57 void WriteObjects::cb_pose(const geometry_msgs::
58 PoseWithCovarianceStampedConstPtr & msg){
59
60     geometry_msgs::Point position_ = msg->pose.pose.position;
61     geometry_msgs::Quaternion orientation_ = msg->pose.pose.
62 orientation;
63
64     std::string pos_x = std::to_string(position_.x);
65     std::string pos_y = std::to_string(position_.y);
66     std::string pos_z = std::to_string(position_.z);
67
68     std::string or_w = std::to_string(orientation_.w);
69     std::string or_x = std::to_string(orientation_.x);
70     std::string or_y = std::to_string(orientation_.y);
71     std::string or_z = std::to_string(orientation_.z);
72
73     position = pos_x + ':' + pos_y + ':' + pos_z;
74     orientation = or_w + ':' + or_x + ':' + or_y + ':' + or_z;
75 }
```

Listing B.6: Código del archivo write_objects.cpp

main_write_objects.cpp

```
1 #include "rgbd_filter/write_objects.hpp"
2
3 int main(int argc, char** argv){
4
5     ros::init(argc, argv, "write_objects");
6
7     WriteObjects write_objects;
8
9     ros::spin();
```

B.3. NODO WRITE_OBJECTS

```
10  
11 }
```

Listing B.7: Código del archivo `main_write_objects.hpp`

Bibliografía

- [1] Teresa Guerrero. *El 'GPS' de los vikingos*. Noviembre de 2011. URL: <https://www.elmundo.es/elmundo/2011/11/07/ciencia/1320663197.html>. (acc: 25.05.2022).
- [2] MetroMadrid. *Metro de Madrid*. URL: <https://www.metromadrid.es/es/viaja-en-metro/plano-de-metro-de-madrid>. (acc: 25.05.2022).
- [3] Cristina Sánchez. *Shakey, El Primer Robot Inteligente de la Historia y el abuelo del Coche Autónomo*. Abril de 2017. URL: https://www.eldiario.es/hojaderouter/tecnologia/shakey-robot-inteligencia-artificial-coche-autonomo_1_3466717.html. (acc: 25.05.2022).
- [4] David Cassel. *Remembering shakey, the First Intelligent Robot*. Diciembre de 2021. URL: <https://thenewstack.io/remembering-shakey-first-intelligent-robot/>. (acc: 25.05.2022).
- [5] Wikipedia. *Metro de Málaga*. URL: https://es.wikipedia.org/wiki/Metro_de_M%C3%A1laga. (acc: 25.05.2022).
- [6] Universidad de Stanford. *STAIR*. URL: <http://stair.stanford.edu/index.php>. (acc: 25.05.2022).
- [7] Morgan Quigley, Eric Berger y Andrew Y. Ng. “STAIR: hardware and software architecture”. En: (2007).
- [8] Wikipedia. *Licencia permisiva*. Marzo de 2022. URL: https://es.wikipedia.org/wiki/Licencia_permisiva. (acc: 25.05.2022).
- [9] Github. *Maximum subscribers of a topic*. URL: https://github.com/ros2/rmw%5C_fastrtps/issues/249. (acc: 25.05.2022).
- [10] Chiara Fulgenzi, Anne Spalanzani y Christian Laugier. “Dynamic Obstacle Avoidance in uncertain environment combining PVOs and Occupancy Grid”. En: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. 2007, págs. 1610-1616. DOI: [10.1109/ROBOT.2007.363554](https://doi.org/10.1109/ROBOT.2007.363554).
- [11] Joseph Redmon y col. *You Only Look Once: Unified, Real-Time Object Detection*. 2015. eprint: [arXiv:1506.02640](https://arxiv.org/abs/1506.02640).
- [12] Wikipedia. *Compute Unified Device Architecture*. URL: <https://es.wikipedia.org/wiki/CUDA>. (acc: 26.05.2022).

BIBLIOGRAFÍA

- [13] Chad Rockey. *ROS WIKI: depthimage to laserscan*. URL: http://wiki.ros.org/depthimage_to_laserscan. (acc: 26.05.2022).
- [14] Brian Gerkey. *ROS WIKI: gmapping*. URL: <http://wiki.ros.org/gmapping?distro=noetic>. (acc: 26.05.2022).
- [15] Patrick Mihelich. *ROS WIKI: image transport*. URL: http://wiki.ros.org/image_transport. (acc: 26.05.2022).
- [16] Marko Bjelonic. *ROS WIKI: darknet ros*. URL: http://wiki.ros.org/darknet_ros. (acc: 26.05.2022).
- [17] ROS. *image-encodings.h File Reference*. URL: http://docs.ros.org/en/jade/api/sensor_msgs/html/image__encodings_8h.html. (acc: 26.05.2022).