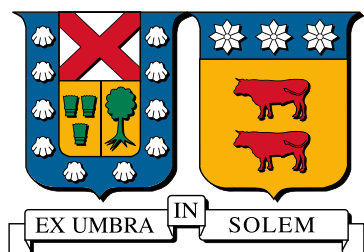


UNIVERSIDAD TÉCNICA FEDERICO SANTA
MARÍA

DEPARTAMENTO DE ELECTRÓNICA

VALPARAÍSO - CHILE



“MAPEO DE ENTORNOS CERRADOS
USANDO CÁMARAS RGB-D MEDIANTE
EL USO DE UN ROBOT CUADRÚPEDO”

NICOLAS IGNACIO JORQUERA MARTINEZ

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL
ELECTRÓNICO

PROFESOR GUIA: DR. FERNANDO AUAT
PROFESOR CORREFERENTE: DR. SAMIR KOURO

MARZO 2024

Mapeo de entornos cerrados usando cámaras RGB-D mediante el uso de un robot cuadrúpedo

Nicolás Ignacio Jorquera Martínez

Memoria para optar al título de Ingeniero Civil Electrónico, mención Computadores,
submención Informática

Universidad Técnica Federico Santa María

Profesor Guía: Dr. Fernando Auat

Profesor Correferente: Dr. Samir Kouro

Marzo 2024

Resumen

En este proyecto se implementó un algoritmo de Matlab en el robot Uni-tree GO 1, el cual se encargó de realizar el mapeo del entorno de lo que el robot observa. Esta implementación utiliza la posición y orientación que entrega el robot junto con las nubes de puntos que se obtienen de la cámara RGB-D que se encuentra en la parte frontal de la cabeza. Los resultados obtenidos muestran que si bien la rapidez con que se procesa la información permite ver una previsualización en tiempo real del mapeo tridimensional, la calidad del mapeo tridimensional no es el óptimo, presenta errores con respecto al posicionamiento de las nubes de puntos debido a la odometría que se obtiene del robot. Además, se propone un posible uso del sistema para inspeccionar infraestructuras como tuberías, líneas eléctricas, o estructuras en construcción, especialmente en áreas que presentan riesgos para los trabajadores.

Mapping closed environments using RGB-D cameras using a quadruped robot

Nicolás Ignacio Jorquera Martínez

Thesis for the fulfillment of the B.S. in Electronic Engineering, major in Computer
Electronics, minor in Informatics

Universidad Técnica Federico Santa María

Profesor Guía: Dr. Fernando Auat

Profesor Correferente: Dr. Samir Kouro

March 2024

Abstract

In this project, a Matlab algorithm was implemented on the Unitree GO 1 robot, which was responsible for mapping the environment that the robot observes. This implementation uses the position and orientation provided by the robot, along with the point clouds obtained from the RGB-D camera located on the front of the head. The results show that while the speed of processing the information allows for a real-time preview of the three-dimensional mapping, the quality of the three-dimensional mapping is not optimal. It presents errors with respect to the positioning of the point clouds due to the odometry obtained from the robot. Additionally, a possible use of the system for inspecting infrastructure such as pipelines, power lines, or structures under construction is proposed, especially in areas that present risks to workers.

Índice de contenidos

1. Introducción	1
1.1. Motivación y contexto	2
1.2. Alcances y Contribuciones	3
1.3. Objetivos del trabajo	4
1.4. Metodología	5
1.4.1. Metodología de investigación	5
1.4.2. Metodología de implementación	5
2. Antecedentes	7
2.1. Robots Cuadrúpedos	7
2.2. Sensores RGB-D	8
2.3. ROS	9
2.4. Estado del arte	10
3. Procedimiento	12
3.1. Simulación	12
3.1.1. Levantar simulación	12
3.1.2. Implementación con RTAB-Map	13
3.2. Implementación en robot real	17
3.2.1. Arquitectura de Go1	18
3.2.2. Implementación con RTAB-Map	19
3.2.3. Implementación con Matlab	24
4. Resultados	32
4.1. Resultados de la simulación	32
4.2. Resultados de las implementación con Matlab	36
4.2.1. Entornos reales y rutas de recorrido	38
4.2.2. Configuración A ($f=1\text{hz}$ y $d=5$)	39

4.2.3. Configuración B ($f=15\text{hz}$ y $d=5$)	40
4.2.4. Configuración C ($f=50\text{hz}$ y $d=5$)	41
4.2.5. Configuración D ($f=1\text{hz}$ y $d=10$)	42
4.2.6. Configuración E ($f=15\text{hz}$ y $d=10$)	43
4.2.7. Configuración F ($f=50\text{hz}$ y $d=10$)	44
4.2.8. Configuración G ($f=1\text{hz}$ y $d=30$)	45
4.2.9. Configuración H ($f=15\text{hz}$ y $d=30$)	46
4.2.10. Configuración I ($f=50\text{hz}$ y $d=30$)	47
4.2.11. Calificaciones de las configuraciones	48
4.2.12. Análisis de mejor configuración	48
5. Conclusiones	52
Referencias	55
6. Anexos	56
6.1. Launch file para simulación	56
6.2. Launch file para implementación RTAB-Map en robot real	58
6.3. Script para implementación de matlab	60

Índice de tablas

3.1. Teclas para controlar al robot durante la simulación.	13
3.2. Placas de procesamiento de Go1	18
4.3. Matriz de evaluación de resultados	37
4.4. Matriz de evaluación de resultados	48

Índice de figuras

3.1. Simulación funcionando	17
---------------------------------------	----

3.2. Robot Unitree Go1	17
3.3. Diagrama del sistema de arquitectura de robot Go1 [1]	18
3.4. Diagrama de flujo del procedimiento	20
3.5. Registro de como llegan los tópicos de color y profundidad	22
3.6. Diagrama de flujo del procedimiento	26
3.7. Árbol de relaciones entre frames	28
3.8. Mapa generado por la implementación en Matlab	31
4.9. Partes de mapeo con RTAB-Map en la simulación	33
4.10. Visualización del mapa creado	34
4.11. Visualización del mapa creado con mas detalle	35
4.12. Mapas generados en entorno plano y entorno con objetos	38
4.13. Mapas generados en entorno plano y entorno con objetos	39
4.14. Mapas generados en entorno plano y entorno con objetos	40
4.15. Mapas generados en entorno plano y entorno con objetos	41
4.16. Mapas generados en entorno plano y entorno con objetos	42
4.17. Mapas generados en entorno plano y entorno con objetos	43
4.18. Mapas generados en entorno plano y entorno con objetos	44
4.19. Mapas generados en entorno plano y entorno con objetos	45
4.20. Mapas generados en entorno plano y entorno con objetos	46
4.21. Mapas generados en entorno plano y entorno con objetos	47
4.22. Entorno plano: real y mapa generado	48
4.23. Mapa generado con más tiempo del entorno plano	49
4.24. Entorno denso: real y mapa generado	50
4.25. Mapa generado con más tiempo del entorno denso	51

1. Introducción

Los robots cuadrúpedos han constituido un campo fascinante de investigación y desarrollo durante numerosos años. Empresas pioneras, como Boston Dynamics, han dedicado esfuerzos significativos al perfeccionamiento de la movilidad y agilidad de estos robots, con el objetivo de que puedan ser eficazmente utilizados tanto en entornos laborales como recreativos. Esta evolución representa un hito en la robótica, abriendo un abanico de posibilidades en diversas aplicaciones.

Por otro lado, los sensores RGB-D, que representan una innovación tecnológica crucial, son dispositivos de detección de profundidad que operan en sinergia con una cámara de sensor RGB (colores rojo, verde y azul). Estos sensores tienen la capacidad única de enriquecer la imagen convencional con valiosa información de profundidad (relacionada con la distancia al sensor) en una base por píxel. Esta característica ha sido un catalizador en el campo de la visión por computadora y los gráficos por computadora, impulsando a estas comunidades científicas y tecnológicas a explorar y desarrollar soluciones innovadoras y avanzadas basadas en imágenes RGB-D.

En los últimos años, la integración y el progreso en el uso de sensores de profundidad han marcado un avance significativo en la robótica y las tecnologías relacionadas, abriendo puertas a nuevas posibilidades en términos de navegación autónoma, mapeo del entorno y la interacción más fluida y eficiente entre robots y su entorno. Estos avances, combinados con la mejora continua en la movilidad de los robots cuadrúpedos, auguran un futuro prometedor y emocionante en el ámbito de la robótica avanzada.

Para la exitosa realización de este proyecto, es esencial contar con una profunda comprensión y conocimientos especializados en campos como el procesamiento de imágenes y la visión por computadora. El procesamiento de imágenes se refiere a un conjunto avanzado de técnicas aplicadas a imágenes digitales, con el propósito primordial de mejorar su calidad o facilitar la extracción de información valiosa. Estas técnicas abarcan desde el refinamiento de imágenes mediante filtrado, para suavizar, eliminar ruido o

acentuar bordes, hasta procedimientos más complejos como la detección de bordes y la segmentación de imágenes.

Por otro lado, la visión por computadora representa un ámbito de estudio fascinante y desafiante que aspira a emular las habilidades visuales de los seres biológicos. Este campo se centra en el análisis exhaustivo de imágenes o videos para obtener una comprensión profunda y detallada de ellos, de manera similar a cómo el cerebro humano procesa la información visual. En esencia, la visión por computadora se dedica a interpretar y comprender el contenido de las imágenes, lo que la convierte en una herramienta crucial para este proyecto.

La integración de estos dos campos, procesamiento de imágenes y visión por computadora, es vital para avanzar en la comprensión y manipulación de datos visuales, lo que resulta esencial para el éxito y la innovación en nuestro proyecto.

1.1. Motivación y contexto

Este proyecto emerge como respuesta a necesidades crecientemente evidentes en los campos de la robótica y la percepción ambiental. Una de estas necesidades es la capacidad de explorar y cartografiar con precisión entornos cerrados, una habilidad que se ha vuelto crucial para aplicaciones como la inspección y vigilancia de espacios inaccesibles o peligrosos para seres humanos. Esto abarca desde la exploración de áreas afectadas por desastres naturales hasta la inspección de infraestructuras industriales y la monitorización de estructuras en desuso.

Adicionalmente, el creciente auge de la robótica de servicio ha generado una demanda significativa de robots capaces de realizar tareas de inspección y mantenimiento en entornos interiores. Esta tendencia representa oportunidades de mercado importantes para soluciones que aseguren tanto la seguridad como la eficiencia en la exploración de espacios confinados.

El contexto tecnológico actual desempeña un papel fundamental en el avance de este

proyecto. Los progresos en sensores de percepción, algoritmos de visión por computadora y capacidades de procesamiento han allanado el camino hacia una percepción del entorno más precisa y accesible. La integración de datos de cámaras RGB-D, que combina información de color con datos de profundidad, permite una representación tridimensional detallada del entorno, facilitando así la navegación autónoma de robots en espacios restringidos.

El proyecto de mapeo de entornos cerrados, utilizando un robot cuadrúpedo equipado con cámaras RGB-D, se sitúa en un escenario donde la exploración segura y eficiente de espacios interiores es esencial. Al aprovechar los avances tecnológicos y las oportunidades de mercado, esta investigación aspira a ampliar las capacidades de los robots en la exploración y cartografía de entornos cerrados, con aplicaciones que van desde la seguridad industrial hasta la respuesta a emergencias. Este enfoque no solo refleja un compromiso con la innovación en la robótica, sino que también subraya la importancia de desarrollar tecnologías que respondan de manera efectiva a desafíos reales y urgentes.

1.2. Alcances y Contribuciones

Se limitan los alcances del presente trabajo bajo las siguientes condiciones:

- El robot cuadrúpedo equipado con cámaras RGB-D permitirá la exploración segura de entornos inaccesibles o peligrosos para los seres humanos. Esto incluye la capacidad de inspeccionar áreas afectadas por desastres naturales, como zonas inundadas o estructuras colapsadas, sin poner en riesgo vidas humanas.
- El proyecto abordará la creciente demanda de robots de servicio para tareas de inspección y mantenimiento en entornos interiores, como plantas industriales, almacenes, hospitales, y edificios comerciales. Los robots cuadrúpedos podrán realizar inspecciones detalladas y llevar a cabo tareas de seguridad y mantenimiento de manera eficiente.
- El enfoque en la navegación autónoma en espacios cerrados permitirá a los robots

cuadrúpedos sortear obstáculos y moverse con agilidad, lo que es esencial para tareas de mapeo y exploración en entornos interiores complejos.

- La fusión de datos de cámaras RGB-D permitirá al robot obtener una percepción tridimensional detallada del entorno, lo que mejorará la precisión de la cartografía y la toma de decisiones autónomas del robot.

A partir de los resultados obtenidos se puede reconocer las siguientes contribuciones:

- Una de las principales contribuciones será mejorar la seguridad y eficiencia en la exploración de entornos cerrados. Esto se traducirá en la reducción de riesgos para los trabajadores y en una mayor eficiencia en tareas de inspección y mantenimiento.
- El proyecto tendrá un impacto significativo en aplicaciones de rescate y respuesta a desastres al permitir la exploración temprana de áreas peligrosas, lo que podría salvar vidas y acelerar las operaciones de socorro.
- En el ámbito industrial, las contribuciones incluirán la capacidad de realizar inspecciones más precisas y regulares, lo que puede llevar a una mayor eficiencia y reducción de costos en la operación y mantenimiento de instalaciones.
- El proyecto contribuirá al avance de la tecnología de percepción del entorno, lo que puede tener aplicaciones en una variedad de campos, desde la robótica autónoma hasta la realidad aumentada y virtual.

1.3. Objetivos del trabajo

El objetivo principal de este proyecto es desarrollar un sistema de mapeo de entornos cerrados utilizando cámaras RGB-D montadas en un robot cuadrúpedo. El sistema permitirá al robot explorar y cartografiar con precisión espacios interiores, con un enfoque en la navegación autónoma en entornos confinados. El objetivo central es mejorar la seguridad, eficiencia y capacidad de exploración en entornos que son inaccesibles o pe-

ligeros para los seres humanos, lo que abarca desde la inspección de áreas afectadas por desastres naturales hasta la monitorización de infraestructuras industriales. Dado este objetivo primario se desprenden los siguientes objetivos específicos:

- Capturar la información proporcionada por el sensor RGB-D. Esto implica utilizar el sensor para recopilar datos sobre el entorno en el que se encuentra el robot cuadrúpedo.
- Implementar un algoritmo para procesar la información capturada por el sensor RGB-D y generar un mapa tridimensional del entorno. Esto implica utilizar técnicas de procesamiento de imágenes y visión por computadora para analizar los datos recopilados por el sensor y crear una representación tridimensional del entorno.
- Entregar la información del mapa tridimensional generado. Esto puede lograrse almacenando el mapa en el mismo robot o enviándolo a algún dispositivo conectado al robot para su posterior visualización.

1.4. Metodología

1.4.1. Metodología de investigación

Los pasos necesarios para la realización de este trabajo se exponen a continuación:

- Estudiar el estado del arte, mediante publicaciones, revistas y tesis especializadas en el área.
- Investigar librerías en ROS que puedan hacer lo que se requiere para el proyecto (SLAM con cámaras RGBD).

1.4.2. Metodología de implementación

Los pasos para la implementación que se propone en este trabajo son:

- Configuración del ambiente necesario para trabajar con el robot (Ubuntu, ROS,

paquetes, librerías, etc...)

- Simular el modelo utilizando los paquetes propios del robot, junto con proporcionarle un entorno en el que pueda moverse el robot.
- Implementación de la librería de ROS seleccionada para realizar el SLAM.
- Simulación del modelo junto con la librería para SLAM, y verificar el funcionamiento.
- Implementación en el robot real de lo que fue simulado.
- Verificar el correcto funcionamiento de la implementación física del modelo.

2. Antecedentes

2.1. Robots Cuadrúpedos

Un robot cuadrúpedo se define como un robot equipado con cuatro patas, lo que le permite adoptar patrones de locomoción sofisticados y navegar por superficies donde los robots con ruedas enfrentarían dificultades. Estos robots son notablemente superiores a los de ruedas y orugas en la mayoría de los contextos, gracias a su capacidad para maniobrar en una amplia gama de terrenos, de manera similar a humanos y animales. Sin embargo, existen situaciones específicas donde los robots diseñados para condiciones particulares pueden ser la mejor opción.

La construcción de un robot cuadrúpedo involucra tanto componentes mecánicos como eléctricos. Por ejemplo, un modelo típico puede estar equipado con tan solo cuatro servomotores, uno para cada pata, y los movimientos necesarios para la locomoción se logran a través del uso de mecanismos de enlace. Estos mecanismos, que pueden ser de naturaleza física o lógica, generalmente se implementan como conexiones físicas en este tipo de aplicaciones para asegurar mayor robustez.

Los avances en la movilidad y agilidad de los robots cuadrúpedos han sido notables en los últimos años. Mientras que los primeros modelos se enfocaban en mantener el equilibrio, las versiones más recientes han alcanzado un rendimiento dinámico y una adaptabilidad ambiental sin precedentes. Estos avances han permitido su aplicación en áreas como la respuesta a emergencias, reconocimiento militar y construcción de infraestructuras.

Se ha invertido considerablemente en el perfeccionamiento de la estructura biónica y los métodos de control de movimiento de los robots cuadrúpedos para mejorar su habilidad de cambio de marcha, adaptación a terrenos variados y resistencia a perturbaciones. Adicionalmente, se han desarrollado modelos con brazos para asistir en tareas móviles, así como métodos de control orientados a múltiples tareas que abarcan todo el cuerpo del robot. Estos desarrollos subrayan el creciente potencial y versatilidad de los robots

cuadrúpedos en una variedad de aplicaciones prácticas.

2.2. Sensores RGB-D

Los sensores RGB-D son dispositivos especializados en la detección de profundidad, que funcionan en conjunto con una cámara de sensor RGB (rojo, verde y azul). Estos sensores enriquecen las imágenes convencionales con información detallada de profundidad, relacionada con la distancia al sensor, en una base por píxel, ofreciendo así una visión más completa del entorno.

Proporcionando mediciones densas y en tiempo real de superficies 3D, los sensores RGB-D emiten señales de 4 canales. Los tres canales de color RGB describen la apariencia visual de la superficie, mientras que un cuarto canal añade mediciones geométricas precisas. Desde su introducción hace más de una década, estos sensores se han convertido en un componente esencial en el desarrollo de tecnologías avanzadas de mapeo y reconstrucción 3D.

En los últimos años, la tecnología de mapeo y reconstrucción 3D empleando sensores RGB-D ha experimentado avances significativos. Algoritmos innovadores para la reconstrucción y mapeo 3D basados en RGB-D han sido desarrollados, marcando el estado del arte en este campo.

En el contexto de SLAM (Simultaneous Localization and Mapping), un proceso donde un robot o dispositivo móvil construye un mapa de su entorno y rastrea su propia ubicación dentro de este, los sensores RGB-D son herramientas invaluableles. Proporcionan información esencial de profundidad junto con datos visuales, permitiendo al robot estimar su posición y orientación con respecto a su entorno.

Los algoritmos SLAM basados en RGB-D han demostrado ser altamente efectivos para la reconstrucción y mapeo 3D en tiempo real. Estos aprovechan la información de profundidad de los sensores RGB-D para calcular la posición y orientación del robot, y así construir un mapa 3D detallado del entorno a medida que el robot se desplaza.

Estos avances representan un paso significativo en la robótica y la percepción ambiental, abriendo nuevas posibilidades en una variedad de aplicaciones prácticas.

2.3. ROS

El Robot Operating System (ROS) desempeña un papel crucial en el manejo de los distintos sensores y actuadores del robot Unitree Go 1, abarcando componentes tanto de alto como de bajo nivel.

ROS es un marco de desarrollo de código abierto, diseñado para facilitar significativamente la programación de robots. A pesar de no ser un sistema operativo en el sentido tradicional, proporciona una colección integral de bibliotecas, herramientas y convenciones que optimizan el desarrollo de software modular para robots.

La arquitectura modular de ROS permite a los desarrolladores construir software en forma de "nodos", cada uno realizando funciones específicas y comunicándose a través de un sistema de mensajería eficiente. Esta estructura modular es esencial para el desarrollo de sistemas robóticos complejos y distribuidos, permitiendo una mayor flexibilidad y escalabilidad.

Uno de los beneficios clave de ROS es la reutilización de código. Con una extensa gama de paquetes y bibliotecas a su disposición, los desarrolladores pueden implementar soluciones existentes para problemas comunes en robótica, aumentando la eficiencia y reduciendo el tiempo de desarrollo.

Además, ROS simplifica enormemente la gestión de hardware, ofreciendo controladores y bibliotecas para una amplia gama de dispositivos, desde sensores y actuadores hasta plataformas de locomoción. Esto resulta en una integración más fluida de diversos componentes de hardware en el sistema robótico.

Herramientas de desarrollo como Rviz son fundamentales en ROS, permitiendo a los desarrolladores visualizar datos sensoriales y modelos de robots en tiempo real, lo que facilita enormemente la depuración y optimización del software.

La amplia comunidad de desarrolladores y usuarios de ROS es un recurso inestimable, proporcionando acceso a tutoriales, documentación y foros de discusión. Este soporte comunitario es vital para el aprendizaje continuo y la solución efectiva de problemas en el campo del desarrollo de software robótico.

2.4. Estado del arte

En esta sección se resumen brevemente algunos de los artículos, de entre los que fueron estudiados durante el proceso de investigación del presente trabajo, que resultaron de utilidad para el desarrollo del sistema propuesto.

Generar mapas tridimensionales a partir de sensores RGB-D ha sido estudiado ampliamente desde su masificación con la creación del sensor Kinect de Microsoft en el año 2010. Sin embargo, la idea de crear mapas tridimensionales a partir de datos no es nueva y ha sido un área de investigación activa en robótica y visión por computadora desde al menos las décadas de 1980 y 1990.

Una de las razones del triunfo o masificación de los sensores RGB-D se debe a que son mucho mas baratos que sensores que cumplen funciones similares, como los sensores Lidar que cuestan en ocasiones mas de 10 veces lo que cuesta una cámara RGB-D. En esta misma linea de abaratar costos al momento de generar un mapa tridimensional se creo una forma de obtener odometría (posición y orientación) a partir de las imágenes que se obtienen del sensor RGB-D, en vez de utilizar un sensor que obtenga esta posición y orientación específicamente. A este procedimiento se le llama odometría visual, ya que obtiene posición y orientación de diferentes referencias visuales que se obtienen de la cámara RGB-D.

- **3-D Mapping With an RGB-D Camera**

En [2], los autores introducen un innovador sistema de mapeo capaz de generar mapas 3D de alta precisión de forma robusta, utilizando únicamente una cámara RGB-D. Lo destacable de su enfoque es que prescinde completamente de sensores

adicionales o de sistemas de odometría. Gracias a la disponibilidad de sensores RGB-D económicos y ligeros, como el Microsoft Kinect, su sistema es adecuado tanto para pequeños robots domésticos, como las aspiradoras, como para drones, como los cuadricópteros. Además, el sistema que presentan no se limita a aplicaciones robóticas; también facilita la reconstrucción manual de modelos 3D detallados.

- **Robust real-time visual odometry for dense RGB-D mapping**

En [3] se detallan mejoras en el algoritmo Kintinuous, ampliando su capacidad para crear modelos 3D detallados y coloreados de entornos grandes. Se logra mediante la incorporación de métodos avanzados de seguimiento de cámara, una implementación eficiente en GPU para la odometría visual RGB-D, y técnicas de coloración de superficies en tiempo real. Los autores validan estas innovaciones con experimentos que demuestran su eficacia en aplicaciones de robótica y realidad virtual, ofreciendo soluciones robustas para mapear entornos complejos.

- **Actively Mapping Industrial Structures with Information Gain-Based Planning on a Quadruped Robot**

En [4], los autores presentaron un sistema de mapeo activo diseñado para que un robot cuadrúpedo realice reconocimientos autónomos de grandes estructuras. Este sistema, que no requiere de modelos previos, utiliza representaciones en vóxeles y determina iterativamente el mejor punto de vista para la expansión del mapa, considerando tanto la reconstrucción como la evitación de obstáculos. A través de un cálculo que evalúa la ganancia de información y el costo de acceso a puntos candidatos, el robot selecciona su próxima ubicación de exploración y planifica la ruta óptima. Los ensayos demostraron la eficacia del sistema en entornos tanto simulados como reales. Como prueba de su funcionamiento, se realizó una demostración con el robot ANYbotics ANYmal, que logró reconstruir autónomamente fachadas de edificios e infraestructuras industriales.

3. Procedimiento

En el capítulo anterior se abordaron distintos temas como los Robots cuadrúpedos, los sensores RGB-D y ROS. Además, se mencionaron algunos de los topics relacionados, como la adaptabilidad de los robots cuadrupedos en terrenos irregulares y el Simultaneous Localization And Mapping (SLAM). Para este robot en específico se facilita el trabajo al utilizar ROS, ya que este se encarga de manejar los distintos sensores y actuadores del robot, aparte se puede trabajar sobre paquetes (hechos por la comunidad) con distintas funcionalidades.

El robot Unitree Go 1 cuenta con un modelo para gazebo, en donde se puede simular el robot junto con todos sus nodos de ROS.

3.1. Simulación

Debido al riesgo que conlleva trabajar directo con un robot que utiliza ROS sin haberlo utilizado antes, se decidió simular al robot y lograr que funcione el mapeo antes de trabajar con el robot real (aprovechando que Unitree facilita este modelo).

3.1.1. Levantar simulación

Para poder levantar el modelo del robot Unitree Go 1 en Gazebo es crucial seguir el paso a paso publicado por Unitree en Github [5].

Primero se recomienda correr este proyecto en el ambiente de Ubuntu 18.04 y ROS melodic. Luego, simplemente basta con poner tres dependencias en una carpeta src en el workspace de ROS, las dependencias en cuestión son:

- unitree_guide
- unitree_ros
- unitree_legged_msgs

Para el build se pide abrir un terminal, moverte al workspace que alberga a unitree_guide y correr el comando:

```
1 catkin_make
```

Finalmente, para correr la simulación se pide abrir dos terminales en el workspace que alberga a unitree_guide (una para correr la simulación y otra para levantar el control del robot).

En la primera terminal correr los siguientes comandos:

```
source ../devel/setup.bash
roslaunch unitree_guide gazeboSim.launch
```

Mientras que en la segunda terminal, el siguiente comando:

```
../devel/lib/unitree_guide/junior_ctrl
```

Para terminar, se indican las teclas designadas para controlar al robot:

Tabla 3.1: Teclas para controlar al robot durante la simulación.

Comando	Función
'1'	Pasar a posición pasiva (estado inicial)
'2'	Pasar a posición fijada (parado)
'4'	Pasar a posición de trote (para moverse)
'w','a','s','d'	Mover robot cuando esta en posición de trote
'j','l'	Girar robot cuando esta en posición de trote

3.1.2. Implementación con RTAB-Map

RTAB-Map, o Real-Time Appearance-Based Mapping, es un sistema utilizado en robots para crear mapas 3D en tiempo real mientras determina la posición del robot. Basándose en información visual y sensorial, es especialmente útil en entornos desconocidos, siendo adaptable a cambios y condiciones diversas. Su integración con ROS facilita su uso en

sistemas robóticos más amplios. Una característica destacada es su integración con el Robot Operating System (ROS), lo que facilita su incorporación en sistemas robóticos más amplios y su combinación con otras herramientas y bibliotecas disponibles en el ecosistema ROS.

RTAB-Map es el paquete seleccionado para encargarse de procesar la información de las cámaras, en específico necesita los tópicos de RGB (color), profundidad y las configuraciones de la cámara. Solo se encontró este paquete que pueda realizar el mapeo tridimensional con cámaras RGB-D (se encontraron otras pero solo lo hacían con sensores lydar).

Para la instalación de este paquete se necesita de dos paquetes realmente, el paquete rtabmap como tal, y el de rtabmap_ros. Este ultimo se encarga de permitir trabajar con rtabmap en el ambiente de ros.

El paso a paso para la instalación de ambos paquetes se puede encontrar en el github de introlab, en el repositorio rtabmap_ros [6].

Una vez habiendo instalado estos paquetes se creó un launch file, en donde se mapean los tópicos necesarios para que funcione el paquete. Por cada cámara se necesitan los tópicos de /color/image_raw, /depth/image_raw y /color/camera_info, es decir, se necesita la información de los colores, la profundidad y la información de la cámara.

Lo anterior se realiza mediante la siguiente configuración en el launch file (por cada cámara, esta es solo para la de adelante):

```
4 <group ns="camera_face">
  <node pkg="rtabmap_sync" type="rgbd_sync" name="rgbd_sync">
    <remap from="rgb/image" to="color/image_raw"/>
    <remap from="depth/image" to="depth/image_raw"/>
    <remap from="rgb/camera_info" to="color/camera_info"/>
    <param name="approx_sync" value="false"/>
  </node>
</group>
```

Luego de haber obtenido la información de las cámaras se necesita levantar 3 nodos

que se encargan de realizar el mapeo y la visualización.

- El primer nodo que se levanta es el de odometría, el cual en este caso necesita de las imágenes de las cámaras para obtener la posición y orientación a través del método de odometría visual. Lo anterior se realiza por el siguiente código:

```

2      <!-- Odometry -->
      <node pkg="rtabmap_odom" type="rgbd_odometry" name="rgbd_odometry" output
        ="screen">
          <remap from="imu" to="/trunk_imu" />
          <remap from="rgbd_image" to="/camera_face/rgbd_image" />
          <param name="subscribe_rgbd" type="bool" value="true" />
          <param name="frame_id" type="string" value="base" />
7      <param name="rgbd_cameras" type="int" value="1" />
          <param name="wait_for_imu_to_init" type="bool" value="true" />
      </node>

```

- El segundo nodo que se levanta es el que se encarga de crear el mapa, en el cual se tiene que indicar que se trata de una implementación con cámaras RGB-De indicar cual es el frame en el árbol de tf que corresponde al robot en si (en este caso es el "base"). Esto se realiza por el siguiente código:

```

1      <!-- Visual SLAM -->
      <node name="rtabmap" pkg="rtabmap_slam" type="rtabmap" output="screen"
        args="$(arg rtabmap_args)">
          <remap from="imu" to="/trunk_imu" />
          <remap from="rgbd_image" to="/camera_face/rgbd_image" />
6      <param name="subscribe_depth" type="bool" value="false" />
          <param name="subscribe_rgbd" type="bool" value="true" />
          <param name="rgbd_cameras" type="int" value="0" />
          <param name="frame_id" type="string" value="base" />
11     <param name="Grid/RangeMin" type="string" value="0.1" /> <!-- to avoid
adding legs as obstacle
          in occupancy grid map -->
          <param name="Grid/MaxObstacleHeight" type="string" value="1" />
      </node>

```

- El tercer y ultimo nodo se encarga de abrir una ventana en la que se puede

visualizar el mapa tridimensional, el plano, entre otras cosas a medida que se van generando.

```
1      <!-- Visualisation RTAB-Map -->
      <node pkg="rtabmap_viz" type="rtabmap_viz" name="rtabmap_viz"
        args="-d $(find rtabmap_demos)/launch/config/rgbd_gui.ini" output="
screen">
      <remap from="rgbd_image" to="/camera_face/rgbd_image" />
      <param name="subscribe_depth" type="bool" value="false" />
6      <param name="subscribe_rgbd" type="bool" value="true" />
      <param name="subscribe_odom_info" type="bool" value="true" />
      <param name="frame_id" type="string" value="base" />
      <param name="rgbd_cameras" type="int" value="1" />
      </node>
11
```

Para poder levantar este launch file que se encarga del mapeo del entorno, se necesita abrir un tercer terminal (uno mas que en el levantamiento de la simulación). Para poder correr este launch file se necesita el siguiente comando:

```
$ roslaunch rtabmap_demos demo_unitree_quadraped_robot.launch
```

Una vez finalizada la configuración del launch file se necesita el uso de un Digital Twin (un escaneo de un ambiente real). Esto debido a que RTAB-Map si no se le otorga una audiometría explícita del robot, este puede obtenerlo de puntos visuales alrededor del robot. Como el ambiente que trae el modelo de Unitree solo contempla el resto del espacio esta vacío, se utiliza un Digital Twin obtenido en el github de Introlab.

Finalmente, se logró generar un mapeo tridimensional de lo que ve el robot a medida que va caminando utilizando las cámaras RGB-D.

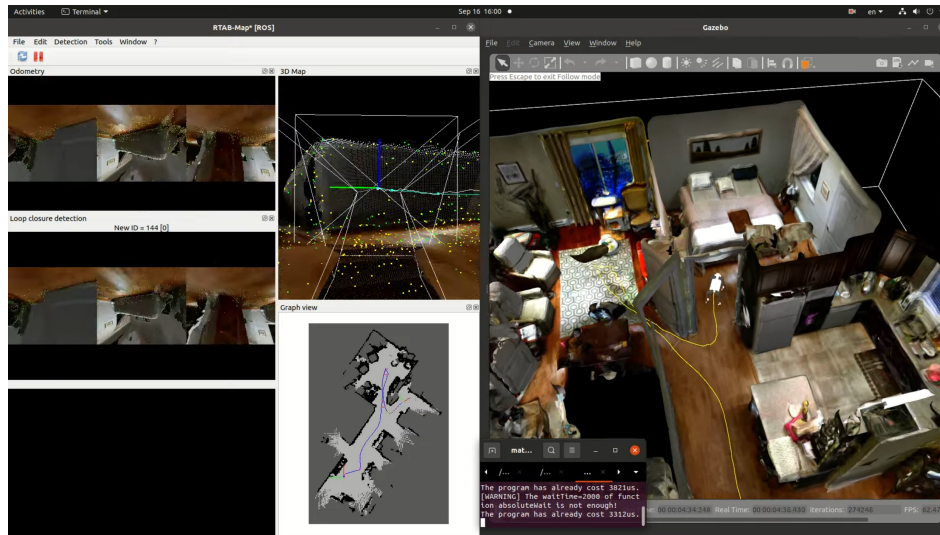


Figura 3.1: Simulación funcionando

3.2. Implementación en robot real

Habiendo trabajado en la simulación, ya se puede trabajar en el robot real con un mínimo riesgo de que se pase a llevar elementos o paquetes del robot por equivocación. También se logró encontrar el paquete adecuado para este proyecto.



Figura 3.2: Robot Unitree Go1

3.2.1. Arquitectura de Go1

Para poder trabajar correctamente con el robot real es crucial conocer su arquitectura. La cual se describe en el siguiente diagrama:

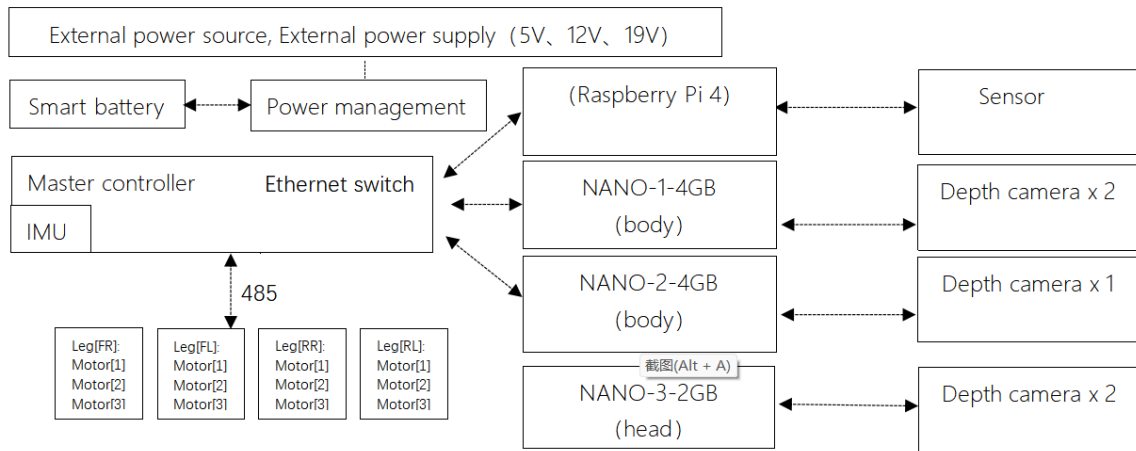


Figura 3.3: Diagrama del sistema de arquitectura de robot Go1 [1]

En este caso tenemos 5 placas de procesamiento:

Placa	Función	IP
MCU	Placa principal de movimiento	192.168.123.10
Raspberry Pi 4	Placa principal de sensores	192.168.12.1
NANO-1-4GB	Cámara de profundidad x 2	192.168.123.14
NANO-2-4GB	Cámara de profundidad x 1	192.168.123.15
NANO-3-2GB	Cámara de profundidad x 2	192.168.123.13

Tabla 3.2: Placas de procesamiento de Go1

La placa MCU se encarga de los movimientos del robot, y debido a que este proyecto se considera mover al robot con el control remoto (concentrarse solo en el mapeo) no se utilizó.

La placa principal de los sensores corresponde a la Raspberry Pi 4, mientras que el resto de placas se centran específicamente en las cámaras. Cada una de estas tarjetas son accesibles a través de SSH, utilizando la IP correspondiente como se muestra en la tabla 3.2, cabe destacar que el usuario para la Raspberry Pi 4 corresponde a *pi*, mientras que para las NANO es *unitree*.

3.2.2. Implementación con RTAB-Map

La idea inicial era que el robot se encargue de la obtención de la información de las cámaras, procesar esa información y enviar el mapeo tridimensional de la habitación, al computador externo (el computador externo solo se encarga de recibir el mapeo y mostrarlo), pero debido a que la placa Raspberry Pi 4 muestra un error sobre de que no encuentra el paquete en las librerías de ros al intentar instalarlo se decidió enviar la información de las cámaras al computador externo.

Con respecto a la implementación de RTAB-Map en si en el computador externo es similar a la implementación que se explico para la simulación, con la diferencia que al momento de obtener los tópicos de la cámara se hace de la siguiente forma:

```
4  <group ns="camera_face">
    <node pkg="rtabmap_sync" type="rgbd_sync" name="rgbd_sync">
      <remap from="rgb/image" to="color" />
      <remap from="depth/image" to="depth" />
      <remap from="rgb/camera_info" to="camera_info" />
      <param name="approx_sync" value="false" />
    </node>
  </group>
```

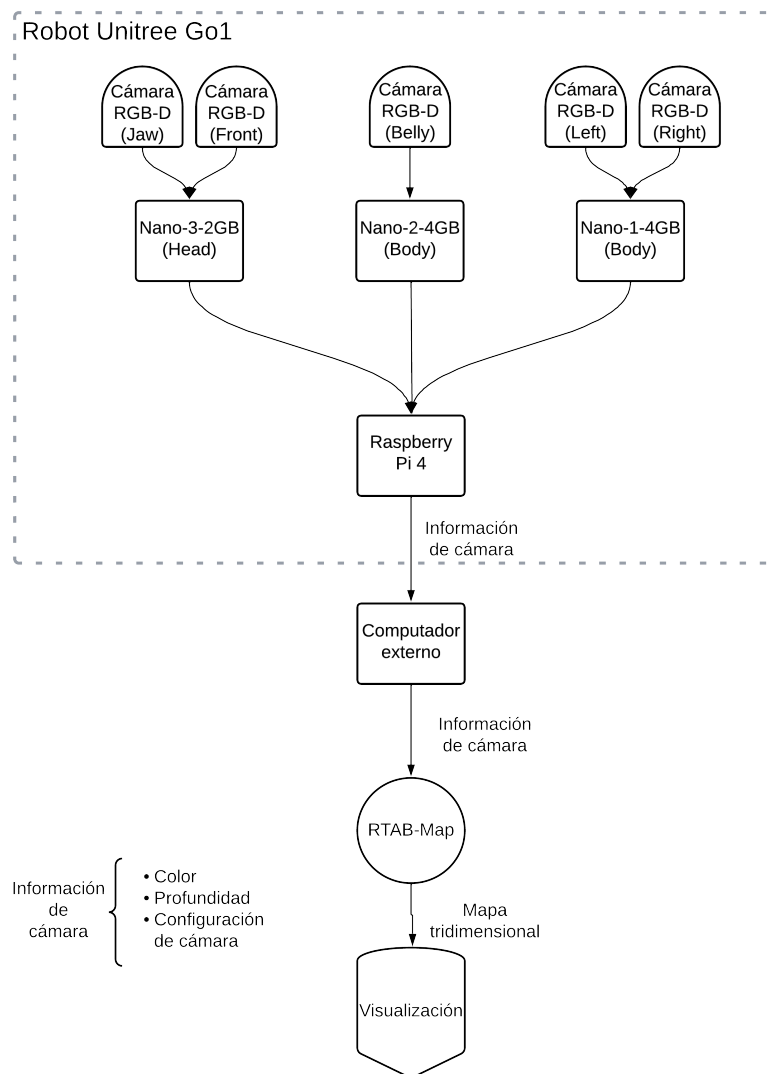


Figura 3.4: Diagrama de flujo del procedimiento

Si bien se ve como si las placas nano no hicieran mucho, estas son las encargadas de publicar la información de las cámaras en ROS, creando nodos y tópicos con esta información. Esto se hizo mediante un wrapper que se encarga de hacer toda la configuración para poder conectarse a la Raspberry pi 4 y publicarla en esta. Como se puede apreciar, la información de la cámara no se procesa hasta que llega al computador externo, en donde se utiliza RTAB-Map, el cual se encarga de generar el mapa. Finalmente, el mapa tridimensional se visualiza a través del mismo computador externo. Debido al tiempo

se decidió trabajar solo con la cámara de la cámara RGBD Front, ya que es la mínima necesaria para hacer el mapeo. La razón por la que se mandan a la Raspberry pi 4 y no directamente desde las Nano al computador externo, es porque la Raspberry pi 4 es la placa principal de los sensores del robot, por lo que si se quiere trabajar con las otras cámaras se necesitaría llevarlas a esta placa principal, haciendo más simple la obtención de la información para el computador externo, ya que solo se tendría que conectar a una placa.

Como se mencionó antes, se trabajará solo con la cámara de la cabeza, por lo que este procedimiento se llevo a cabo en la placa encargada de los sensores de la cabeza (ip terminada en 13). En la etapa de simulación, al levantar el modelo del robot automáticamente se creaban los nodos y tópicos respectivos para cada cámara RGBD, por lo que no era necesario hacer este handler para publicar la información en ROS. Con el robot real la información de las cámaras no se publica automáticamente, por lo que se debe crear un script que se encargue de esto. Este script se baso en 3 ejemplos, un tutorial de ROS sobre la publicación de nodos [7] y dos ejemplos de UnitreecameraSDK sobre como obtener la información de la imagen rectificada [8] y profundidad [9].

Como el paquete de RTAB-Map necesita de 3 tópicos (color, profundidad y la información de la cámara), se decide crear un solo nodo en el cual se publique esta información en 3 tópicos, esto se hace mediante el siguiente código:

```
2  ros::init(argc, argv, "camera_face");  
    ros::NodeHandle nh("~/");  
    image_transport::ImageTransport it(nh);  
    image_transport::Publisher camera1color = it.advertise("color/", 1);  
    image_transport::Publisher camera1depth = it.advertise("depth/", 1);  
    ros::Publisher camera1info = nh.advertise<sensor_msgs::CameraInfo>("camera_info/",  
        1);
```

Para lograr que se este publicando en loop se crea un while que sera valido mientras la cámara este abierta, lo cual sucede siempre mientras este corriendo este script. Dentro de este while se obtiene la información como muestran los ejemplos, y al final de cada iteración se publica la información obtenida en los tópicos correspondientes de la

siguiente forma:

```
camera1color.publish(msgColor);  
camera1depth.publish(msgDepth);  
camera1info.publish(camera_info_msg);
```

Para poder visualizar estas imágenes en el computador externo se necesita conectar a la raspberry pi, y esto se hace al definir la variable ROS_MASTER_URI en el terminal de linux e igualándola a la ip de la raspberry pi:

```
$ ROS_MASTER_URI=http://192.168.123.161:11311
```

Luego de esto se debe levantar rviz, la siguiente imagen muestra la visualización de ambos tópicos en el computador externo

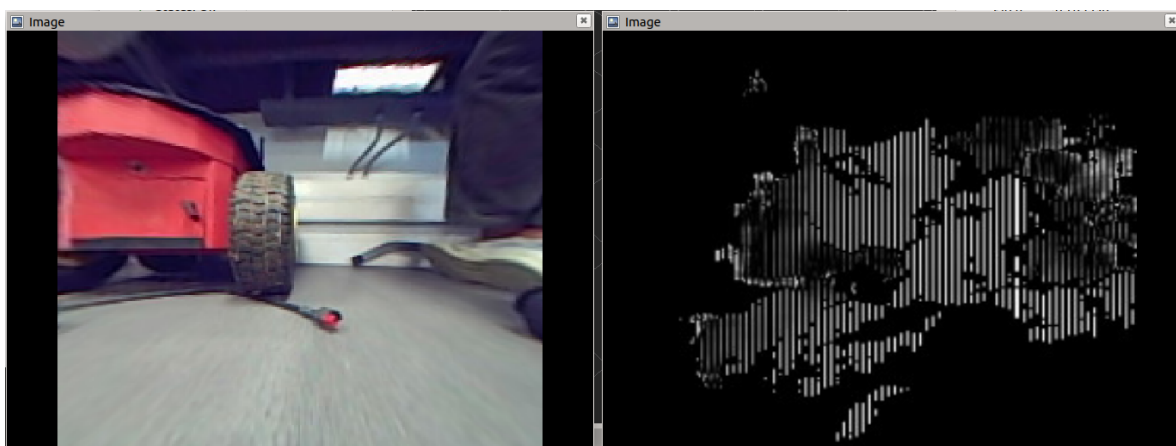


Figura 3.5: Registro de como llegan los tópicos de color y profundidad

Para poder visualizar el topico de "camera_info_msg" se necesita realizar el siguiente comando en la terminal:

```
$ rostopic echo /camera_face/camera_info_msg
```

A lo que se tendria una respuesta con la siguiente estructura:

```
header:  
  seq: 163  
  stamp:
```

```
4      secs: 1636360011
      nsecs: 785186573
      frame_id: "camera_face"
      height: 400
      width: 464
9      distortion_model: "narrow_stereo"
      D: [-0.0850522996852782, 0.18483663877205245, -0.0008254170617288337,
          0.0028799372589857638, 0.0]
      K: [2580.0868051856605, 0.0, 1347.4193338392204, 0.0, 2581.2289148329596,
          1034.4539881461978, 0.0, 0.0, 1.0]
      R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
      P: [2572.388916015625, 0.0, 1352.2652257760637, 0.0, 0.0, 2575.5068359375,
          1032.8075442280096, 0.0, 0.0, 0.0, 1.0, 0.0]
```

Cabe destacar que RTAB-Map no necesita de odometria ya que este paquete realiza una odometria visual con respecto a las imágenes que recibe pero para hacer esta odometria visual se necesita de la configuración de la cámara. Si bien este script logro su objetivo (publicar la información en un nodo y recibirla en el computador externo), el tópico de configuración de la cámara no se logro obtener, este también es un tópico que en la simulación se levantaba solo, pero en la implementación real no. Para poder obtener este tópico se necesitan, entre otras cosas, las matrices de distorsión, intrínseca, proyección y la de rectificación. [10]

- La matriz Intrínseca (K) corresponde a la matriz de la imagen en bruto (raw image).
- La matriz de distorsión (D) corresponde a 5 parametros (depende de el modelo de distorsión) para el modelo "plumb_bob", el cual es de los modelos mas comunes.
- La matriz de proyección (P) corresponde a la matriz de la imagen procesada o rectificada.
- La matriz de rectificación (R) se encarga de alinear el sistema de coordenadas de la cámara con el plano de imagen estéreo ideal para que las líneas epipolares en ambas imágenes estéreo sean paralelas.

Estas matrices se intentaron obtener de diferentes formas, una de ellas fue la de utilizar otro de los ejemplos que se encuentran en el repositorio UnitreecameraSDK de Unitree (junto con los ejemplos de obtener profundidad y color), el ejemplo en cuestión se trata del `getCalibParamsFile`, el cual entrega 6 matrices principalmente, la matriz K (intrínseca de imagen en bruto), R (rotación) y D (distorsión), pero tanto para el lado izquierdo como para el derecho. Al poner estas matrices en RTAB-Map no se generó odometría visual.

Otra estrategia para obtener estas matrices fue la de usar `opencv`, en específico la función `stereoRectify()`, la cual necesita las matrices K y D para obtener las matrices P y R . Pero debido a que esta función está orientada para cámaras stereo (ambas cámaras), se necesitan las 2 matrices K , las 2 matrices D para obtener 2 matrices P y 2 matrices R . El problema es que para el paquete de RTAB-Map solo se necesita una matriz de cada una.

Finalmente se decidió dejar de intentar de esta forma debido a que el robot automáticamente levanta 2 tópicos en ROS (nube de puntos y odometría) con los cuales se puede hacer un mapa sin necesidad de recurrir a la odometría visual y a la configuración de la cámara (matrices K , D , P y R).

3.2.3. Implementación con Matlab

Debido a las dificultades que se tuvieron en la implementación con RTAB-Map se decidió simplificar el concepto para el mapeo, debido a que solo se necesita profundidad/nube de puntos y la odometría del robot, los cuales se levantan automáticamente por el robot. Si bien en los tutoriales de ROS se habla de python y c++ como los lenguajes principales para trabajar en esta plataforma, Matlab también representa un buen lenguaje para comunicarse con robots por ROS. Se decidió trabajar con Matlab debido a que no solo representa un lenguaje consolidado de programación como python o c++, sino que también representa un ambiente de trabajo que permite agilizar el desarrollo de esta implementación.

Otra cosa que cabe destacar es que para poder realizar esta implementación se tuvo que llevar a cabo un formateo o reseteo al estado de fabrica del robot, debido a que este ya no estaba levantando todos los tópicos que debiese levantar automáticamente (solo publicaba los tópicos de nube de puntos de las cámaras frontal y laterales). Este formateo al estado de fabrica se logro tras flashear la memoria SD de la raspberry pi del robot con la imagen que se obtuvo al comunicarse con Unitree directamente, para poder acceder a esta memoria SD se tuvo que abrir al robot.

Esta implementación tiene un diagrama de flujo muy parecido al que se utilizo para trabajar con RTAB-Map, con la gran diferencia de que la información de la cámara en vez de representar 3 tópicos (color, profundidad y configuración de la cámara), solo necesita 2 (profundidad/nube de puntos y odometría).

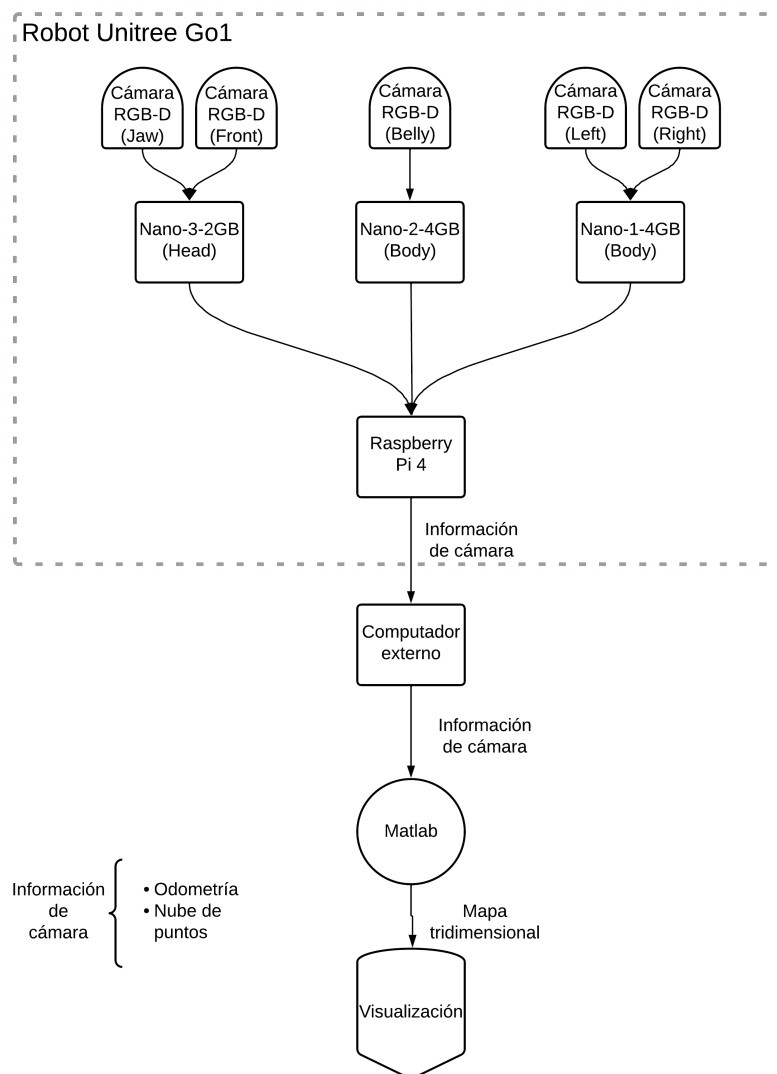


Figura 3.6: Diagrama de flujo del procedimiento

La idea principal de esta implementación es generar una grilla tridimensional que permita posicionar nubes de puntos, y con la odometría del robot (posición y orientación) ir calculando en donde se debería posicionar esta nube de puntos.

En el script de Matlab que se implementó, la conexión con la raspberry pi consta de las siguientes 4 líneas de código, en donde se conecta a la ip de la raspberry pi del robot (vía ethernet) y se define variables que recibirán la información de esos tópicos.

```
rosshutdown;
```

```
2    rosinit('http://192.168.123.161:11311');  
  
    go1_odom = rossubscriber('/tf');  
    go1_pc_face = rossubscriber('/camera1/point_cloud_face');
```

Luego de esto se realizan la creación de variables y una primera obtención de la nube puntos y odometría.

```
5    tStartOdom = tic;  
    [odomTF, status1, statusText1] = receive(go1_odom);  
    [pc_face, status2, statusText2] = receive(go1_pc_face);  
  
    fqGet = 1; %frequency for getting topics  
    maxRange = 5;  
    map3D = occupancyMap3D(50);
```

En el segmento de código anterior se define una variable que se utilizara para manejar la frecuencia de escritura en el mapa, se reciben los primeros datos de nube de puntos y odometría, y finalmente se definen 3 variables, frecuencia de escritura en mapa, rango máximo de la nube de puntos y la definición de un mapa de ocupación con un valor de resolución de la grilla de 50.

Para generar un bucle en el que se obtienen los tópicos del robot en cada iteración se define un while 1, el cual se ejecutara hasta que el script se detenga (con la interfaz de matlab). La idea general de lo que va dentro de este while es obtener la posición y orientación a través del tópico de tf, y asociarlo a una variable que representara la odometría que se utilizara para generar el mapa (hay que filtrar de que sea el frame "trunk") y finalmente graficar en el mapa si ha pasado el tiempo suficiente para cumplir con la restricción de frecuencia.

El tópico de "/tf" corresponde a las distintas transformadas entre frames o partes del robot, la siguiente imagen describe como se relacionan los distintos frames del robot:

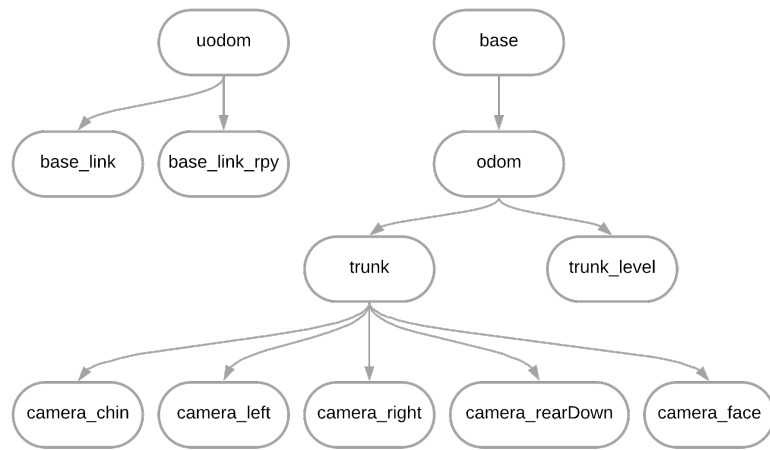


Figura 3.7: Árbol de relaciones entre frames

De este diagrama se puede observar que los frames `uodom`, `base_link` y `base_link_rpy` no se encuentran relacionados con los frames de cámaras, y de hecho no están conectados para nada con el sistema central del robot, debido a esto las transformadas de estos frames son ceros en todas las coordenadas y ángulos. Gracias a observar transformada por transformada y verificando que llega por cada una, se llega a la conclusión de que el frame que almacena las transformadas con respecto a un punto de origen (odometría), es la que tiene la propiedad de `ChildFrameId` igual "trunk". Por lo que después de haber definido un "while 1" para garantizar un loop que lea los mensajes de estos tópicos, se crea un filtro que solo permite utilizar las transformadas con la propiedad de `ChildFrameId` igual a "trunk", esto se hace mediante el siguiente código.

```

[go1_test_msg, status1, statusText1] = receive(go1_odom);
2
if strcmp(go1_test_msg.Transforms.ChildFrameId, 'trunk')
    odomTF = go1_test_msg;
end

```

Para retener la escritura en el mapa hasta que se cumpla la restricción de frecuencia definida al comienzo se hace el siguiente if:

```

tEndOdom = toc(tStartOdom);

```

```
if round(1/tEndOdom) <= fqGet
```

En cada iteración se hace el tic (comienza el conteo de tiempo hasta el toc) al final del if, después de haber dibujado en el mapa), por lo que mientras el tiempo desde la última vez que se escribió en el mapa genere una frecuencia menor que la definida al comienzo del script se debe volver a dibujar en el mapa. Finalmente la frecuencia definida al comienzo sirve de frecuencia máxima, se hizo de esta forma debido a que se perdía demasiada información si se pide que se escriba solo cuando el tiempo genere una frecuencia exacta a la descrita al comienzo, esto debido a que el toc puede caer justo en un tiempo en el que genera una mayor frecuencia, provocando que no se vuelva entrar al if ya que la frecuencia generada en el toc será siempre mayor a la frecuencia definida al comienzo.

Con respecto a la nube de puntos generada por el robot, la información que llega en el tópico de `"/camera1/point_cloud_face"` se encuentra en el formato de `"PointCloud2"`, el cual está presente en el paquete de matlab `"ROS Toolbox"`. Si bien matlab no tiene problemas para trabajar con esta estructura, la función `insertPointCloud()` necesita el formato de `"pointCloud"`. Algo parecido sucede con la odometría, esta función necesita los valores de posición y rotación en cierto orden (la cual no es igual al orden en el que llegan en el tópico `"/tf"`), por lo tanto la conversión de la nube de puntos y la odometría que entrega el robot a la que necesita esta función se hace de la siguiente forma:

```
2      [pc_face, status2, statusText2] = receive(go1_pc_face);

      xPos = odomTF.Transformations.Transform.Translation.X;
      yPos = odomTF.Transformations.Transform.Translation.Y;
      zPos = odomTF.Transformations.Transform.Translation.Z;

7      xRot = odomTF.Transformations.Transform.Rotation.X;
      yRot = odomTF.Transformations.Transform.Rotation.Y;
      zRot = odomTF.Transformations.Transform.Rotation.Z;
      wRot = odomTF.Transformations.Transform.Rotation.W;

12     pose = [ xPos yPos zPos wRot xRot yRot zRot];
      points = readXYZ(pc_face);
```

```
pc = pointCloud(points);  
  
pc.Color = pc_face.readRGB;  
  
insertPointCloud(map3D,pose,pc,maxRange);
```

Primero se obtiene el mensaje desde el robot en la variable "pc_face", luego se extraen 7 datos de esta variable, 3 relacionados con la posición del robot (X-Y-Z) y 4 relacionados con la rotación del mismo (X-Y-Z-W), estos datos se ordenan y guardan en la variable "pose", la cual se utilizara al insertar esta nube de puntos. Para convertir la nube de puntos se utiliza la característica de matlab que permite crear una nube de puntos con la estructura "pointCloud" a partir de una serie de puntos, por lo que se extraen solo los puntos de la variable "pc_face" y se ocupa la función pointCloud() que se encarga de convertir estos puntos en una nube de puntos, para agregar los colores asociados a cada punto se obtienen de pc_face con ".readRGB" y se asocian al objeto Color dentro de la nube de puntos que se acaba de crear. Finalmente se inserta la nube de puntos en el mapa de ocupación "map3D".

Habiendo descrito el script en matlab que se encarga de generar el mapa del entorno, la siguiente imagen es uno de los mapas generados en uno de los entornos de prueba:

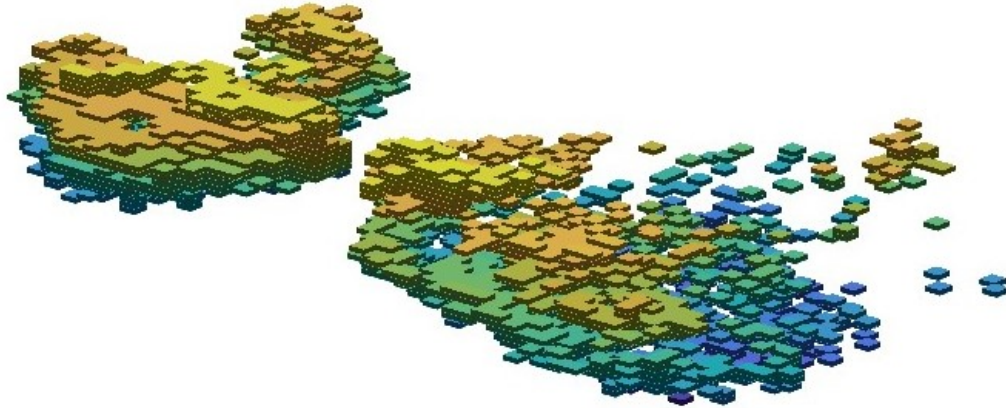


Figura 3.8: Mapa generado por la implementación en Matlab

Parte de las debilidades que se pueden ver en esta implementación son que no se asigna el color obtenido a cada punto y saltos en la generación del mapa.

- Color del mapa: El mapa que se genera ignora los colores asignados para cada punto de la nube de puntos y le asigna colores en función de uno de los ejes, después de probar moviendo al robot se llegó a la conclusión de que es el eje asignado a la altura, es decir, mientras mas alto este el punto se le asignara un color mas cálido (amarillo) pero si es un punto bajo (normalmente el piso) se le asignara un color mas frío (azul).
- Saltos del mapa: Al comenzar a mover el robot se pueden visualizar saltos en la generación del mapa, esto debido a problemas con la actualización de la odometría. Estos problemas son 2 principalmente, la frecuencia con la que se actualiza la odometría no es constante (entre el rango de 0.5 a 5 hz) y la odometría es inexacta, es decir, tiene errores que rondan entre los 5 a 50 cm.

4. Resultados

En esta sección presentamos los resultados obtenidos de la implementación y evaluación de los sistema de mapeo tridimensional utilizando un robot cuadrúpedo equipado con cámaras RGB-D. Tras un exhaustivo proceso de simulación y pruebas en entornos controlados, avanzamos hacia la implementación en el robot real, detallando las capacidades de mapeo, la precisión de la reconstrucción tridimensional, y los desafíos enfrentados. Esta sección se dividirá entre las implementaciones en el simulador y el robot real.

4.1. Resultados de la simulación

La implementación de RTAB-Map en el simulador si bien contemplo algunas dificultades, debido a que RTAB-Map es un paquete que solo recibe las entradas de color, profundidad y configuración de la cámara, no permite cambiar la configuración con la que realiza este mapeo tridimensional. Otro punto que cabe mencionar es que debido a que la simulación se utilizo como una prueba de concepto de lo que se quiere lograr, no se intento implementar mas que un ambiente de pruebas, el cual es un "digital twin" obtenido de uno de los mantenedores y creadores del paquete RTAB-Map Mathiue Labbe, el cual pone a disposición de la comunidad estos escaneos [11]. En este caso se utiliza el archivo llamado "apt.world", el cual corresponde a un escaneo tridimensional del interior de un departamento.

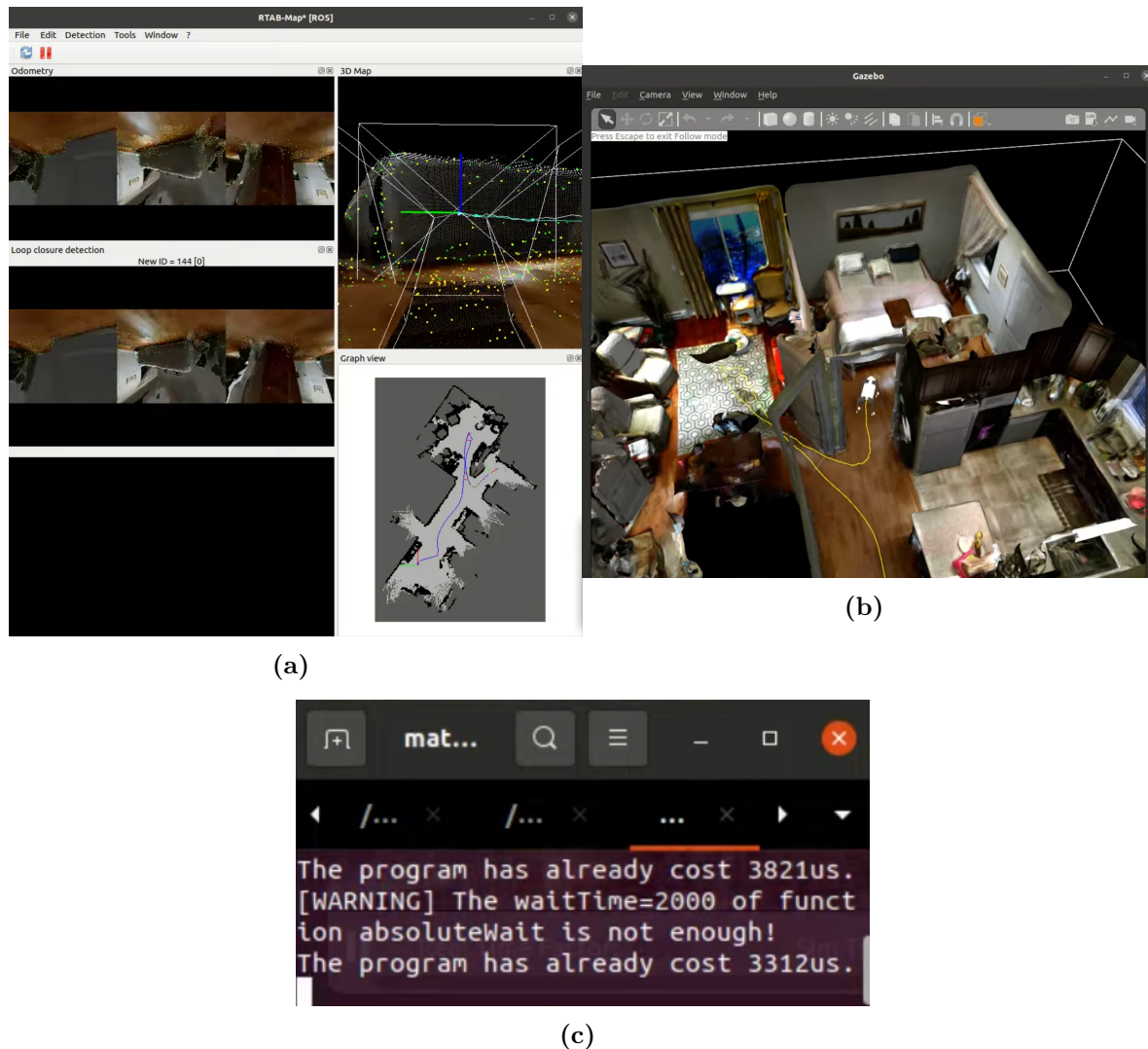


Figura 4.9: Partes de mapeo con RTAB-Map en la simulación

El mapeo con RTAB-Map en el ambiente simulado para el robot Unitree Go 1 se compone de 3 partes. (a) corresponde a la ventana que se abre de RTAB-Map para poder visualizar el mapa que se va generando, (b) es la ventana de gazebo que muestra el ambiente simulado al cual se le quiere obtener el mapa por RTAB-map (corresponde al "apt.world" que se menciono previamente) y finalmente (c) es la terminal de linux que se destina para controlar al robot en el ambiente simulado.

Para poder visualizar el mapa creado después de haber hecho el recorrido con el robot

se debe abrir un archivo en el cual RTAB-Map se encarga de guardar esta información. Para poder abrir este archivo se debe correr el siguiente comando:

```
$ rtabmap ~/.ros/rtabmap.db
```

Después de este comando se abre una ventana en la que se puede recorrer el mapa generado

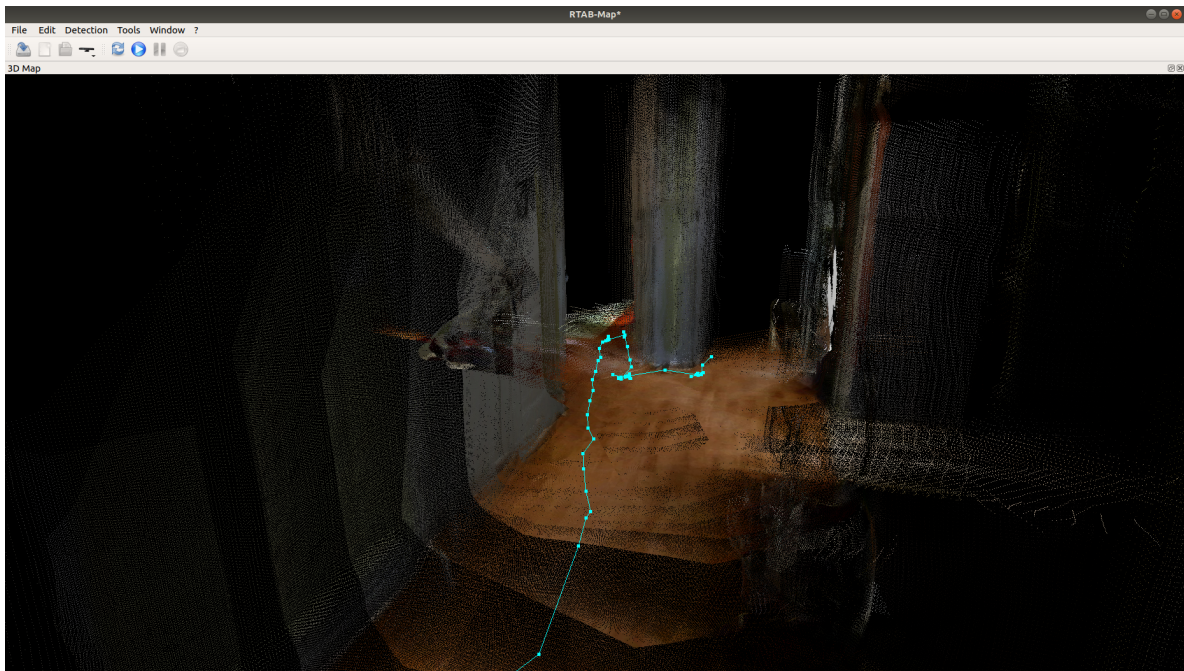


Figura 4.10: Visualización del mapa creado

La calidad de mapa que se genera es bastante buena, considerando la calidad de componentes que tiene el robot, y la odometría visual que se genera RTAB-Map a partir de las referencias visuales que obtiene del robot es lo suficientemente buena para lograr un mapa sin fallos con respecto a la posición de las nubes de puntos. Este mapa creado fue el resultado de un recorrido de 3 minutos, en el que se alcanzó a recorrer un pasillo y entrar a dos habitaciones del departamento, este recorrido se puede observar al mirar la línea celeste. Gracias a este mapa se puede saber que hay al menos dos habitaciones en el departamento, los cuales los dos dan hacia un pasillo, junto con esto también se puede visualizar (recorriendo el mapa generado) que al lado derecho del pasillo está la

cocina, y de hecho se puede saber que hay un refrigerador, el cual se puede visualizar al lado derecho de la figura 4.10.

En esta siguiente imagen podemos ver como la imagen mejora significativamente cuando el recorrido toma mas tiempo para escanear los diferente ángulos de los objetos y habitaciones.



Figura 4.11: Visualización del mapa creado con mas detalle

Al tomar mas tiempo en la toma de datos se pueden reconocer muchos mas detalles del ambiente de simulación, ahora se puede visualizar claramente el piso y las muchas otras habitaciones y puertas que tiene este pasillo. Si bien el recorrido fue parecido, se puede ver como la linea del recorrido (linea celeste) es algo diferente, en esta ocasión tiene unos cúmulos de puntos cada ciertos segmentos, esto es debido a que se realizaron ciertas pausas en el recorrido para dar una vuelta en 360 grados y así poder captar imágenes de los objetos, paredes y piso desde diferentes ángulos, proporcionando así mas detalle a cada uno de estos.

4.2. Resultados de las implementación con Matlab

Como se plasmo en la sección 3 se implementaron dos formas de mapear, RTAB-Map y por Matlab. Debido a que la implementación por RTAB-Map no se logro, no tiene cabida en esta sección.

Para lograr evaluar los resultados de esta implementación se considero dos tipos de entorno para recorrer y mapear con el robot Unitree Go 1. Uno es de un entorno plano y el otro de un entorno con muchos objetos como sillas, mesas, entre otras cosas.

El entorno plano se usara para observar si esta implementación logra generar un mapa plano, y en caso de que genere secciones del mapa que no sean planas poder observar que tanto se repiten estos errores. El entorno lleno de objetos se usara para observar que tan bien genera un mapa mas complicado y poder contrastarlo con los resultados obtenidos por la simulación.

El script de matlab se puede configurar en base a dos variables principalmente, la frecuencia de mapeo y el detalle de los puntos en la grilla del mapa. La variable de frecuencia de mapeo se hará variar con los valores de 1, 15 y 50 hz, mientras que el valor de detalle del mapa se harán con los valores de 5, 10 y 30 (este valor es directamente proporcional al detalle del mapa, es decir, mientras mayor sea este valor mayor sera el detalle del mapa y si es menor , menor sera el detalle del mapa).

La siguiente matriz se crea con la intención de ordenar la evaluación de los resultados de esta implementación con Matlab, y como varia el mapa generado dependiendo de estas dos variables.

Tabla 4.3: Matriz de evaluación de resultados

Resolución	Frecuencia [Hz]			
	Valores	f=1	f=15	f=50
	d=5	A	B	C
	d=10	D	E	F
	d=30	G	H	I

A continuación se realizara una evaluación de cada combinación de estos valores, en donde los puntos a evaluar serán los siguientes:

- Precisión y fidelidad: Evaluar cuán precisamente el mapa refleja las dimensiones y características del entorno real. Esto puede incluir la comparación con mapas de referencia o medidas directas del entorno.
- Resolución: La granularidad de los detalles capturados en el mapa, lo que afecta la capacidad para discernir y navegar a través de pequeños obstáculos o características.
- Cobertura: Qué tan completo es el mapa, es decir, si hay áreas sin mapear o si el robot pudo acceder a todas las áreas de interés.

Cada uno de estos aspectos se calificaran del 1 al 5, en donde 1 significa que esta configuración se desempeño mal en este apartado y un 5 significa que logro completamente este aspecto.

4.2.1. Entornos reales y rutas de recorrido

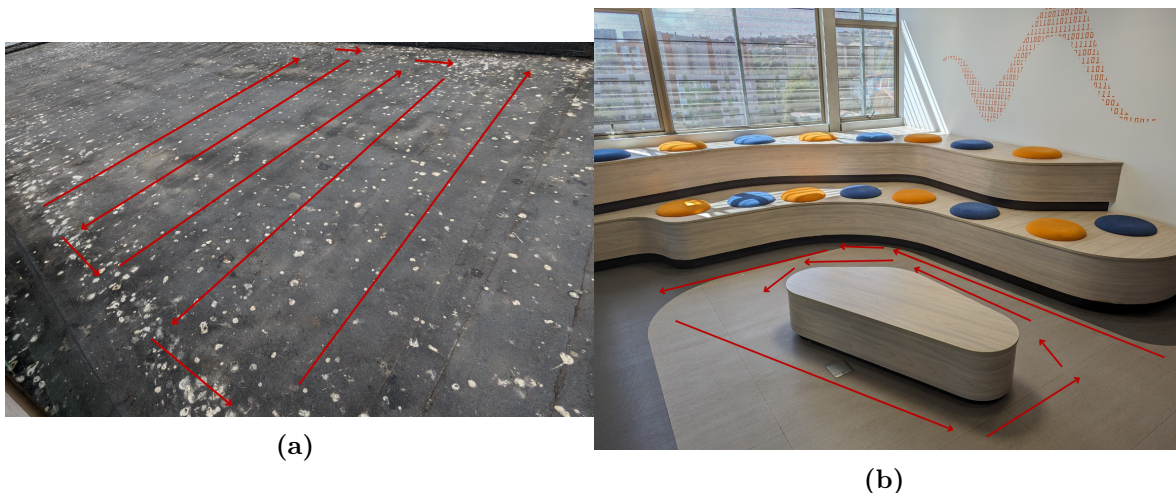


Figura 4.12: Mapas generados en entorno plano y entorno con objetos

Los entornos reales corresponden a una azotea para el entorno plano (no se logro encontrar un entorno cerrado con estas características) y una oficina para el entorno con mas objetos.

Para el entorno plano se decidió hacer una ruta en zigzag como muestra la imagen, esta ruta contempla al rededor de 2 minutos haciendo caminar al robot a un paso lento.

El entorno con mas objetos tiene una ruta que busca recorrer el contorno de los diferentes muebles del entorno. Son dos vueltas, en la primera se recorre el contorno del mueble curvo exterior y en la segunda se recorre el mueble interior. Esta ruta también toma al rededor de 2 minutos.

4.2.2. Configuración A ($f=1\text{hz}$ y $d=5$)

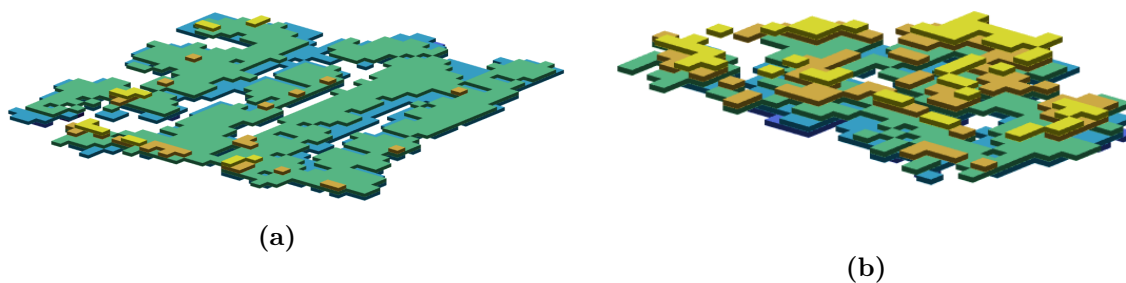


Figura 4.13: Mapas generados en entorno plano y entorno con objetos

- Precisión y fidelidad: Nota 4, los mapas generados son bastante parecidos al entorno real, pero en específico para el entorno mas denso tuvo problemas para posicionar correctamente los objetos de la habitación.
- Resolución: Nota 2, en el entorno plano si bien no se necesita mucha resolución, en el entorno de oficina se nota la falta de detalle.
- Cobertura: Nota 4, tiene pequeños espacios en los que no se mapeo.

4.2.3. Configuración B ($f=15\text{hz}$ y $d=5$)

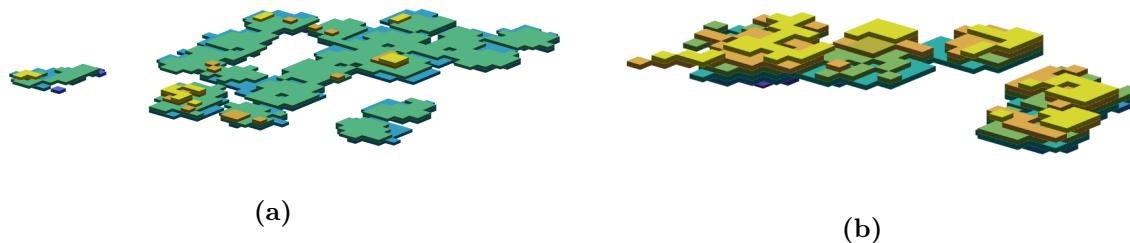


Figura 4.14: Mapas generados en entorno plano y entorno con objetos

- Precisión y fidelidad: Nota 3, los mapas se asemejan al entorno real pero no completamente.
- Resolución: Nota 2, debido a la variable de resolución, para este aspecto las configuraciones A, B y C son bastantes parecidas.
- Cobertura: Nota 2, se encuentran grandes lugares del mapa sin nube de puntos, lugares por los que se recorrió y debiesen tener una nube de puntos asociadas.

4.2.4. Configuración C ($f=50\text{hz}$ y $d=5$)

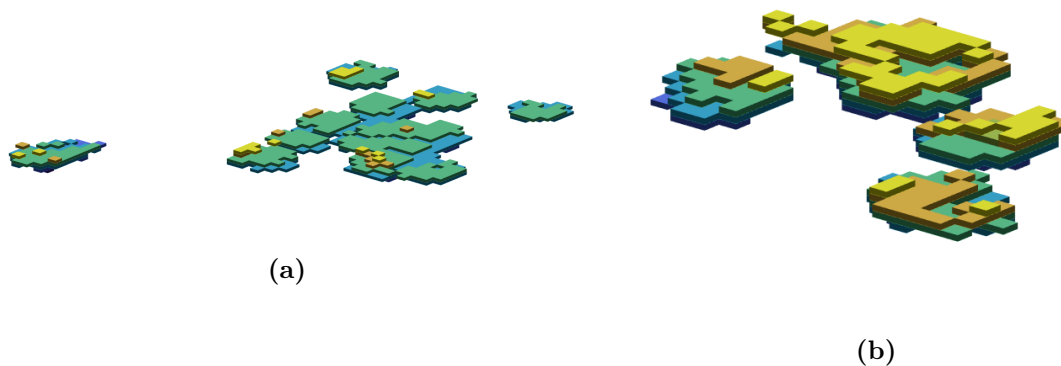


Figura 4.15: Mapas generados en entorno plano y entorno con objetos

- Precisión y fidelidad: Nota 2, los mapas se asemejan pero en el caso del entorno mas denso fallo en representar las características principales.
- Resolución: Nota 2, muy parecida la resolución a la de la configuración A y B.
- Cobertura: Nota 1, además de tener saltos enormes en el mapa, hay una nube de puntos errónea (entorno plano) que se genero al comienzo de la ruta (la nube de puntos que se encuentra mas alejada).

4.2.5. Configuración D ($f=1\text{hz}$ y $d=10$)

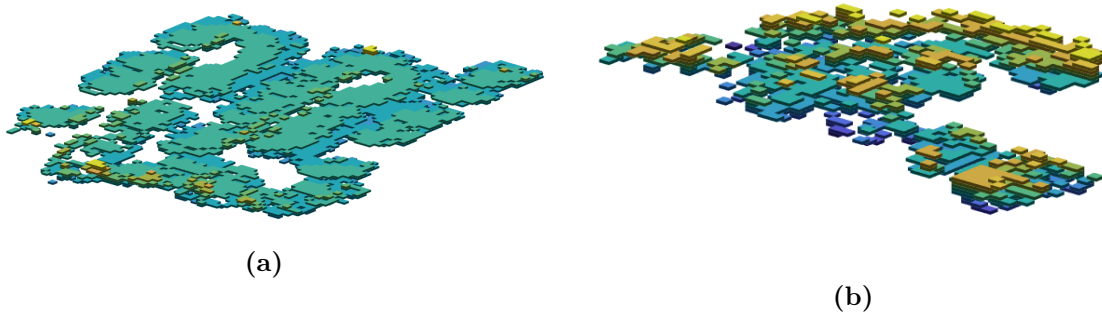


Figura 4.16: Mapas generados en entorno plano y entorno con objetos

- Precisión y fidelidad: Nota 4, las características principales del entorno real se ven plasmadas en los mapas pero hubo problemas con el mueble central del entorno denso.
- Resolución: Nota 3, como la variable de resolución aumento se ve reflejado en este apartado para las configuraciones D, E y F.
- Cobertura: Nota 4, debido a que se encuentran pequeños espacios vacíos en el mapa donde debiesen haber nubes de puntos ya que se paso por esa zona en la ruta.

4.2.6. Configuración E ($f=15\text{hz}$ y $d=10$)

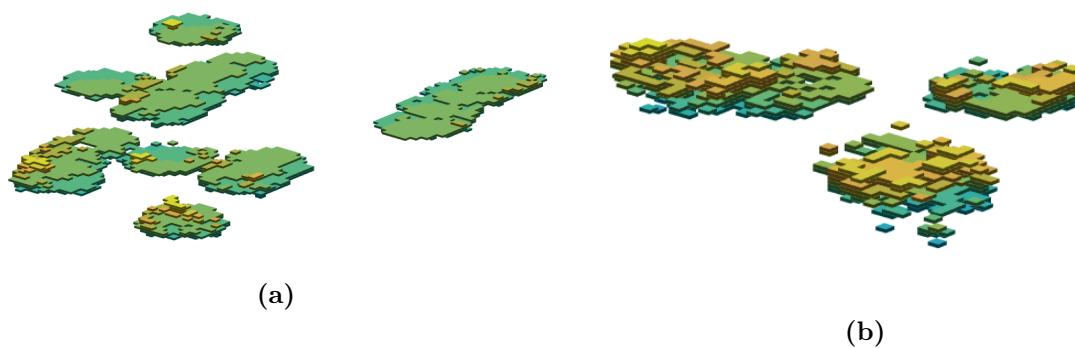


Figura 4.17: Mapas generados en entorno plano y entorno con objetos

- Precisión y fidelidad: Nota 2, se logro capturar algunas de las características principales del entorno.
- Resolución: Nota 3, al igual que en las configuraciones D y F.
- Cobertura: Nota 1, grandes espacios vacíos en el mapa que si debiesen tener asociados una nube de puntos ya que se paso por ahí en la ruta.

4.2.7. Configuración F ($f=50\text{hz}$ y $d=10$)

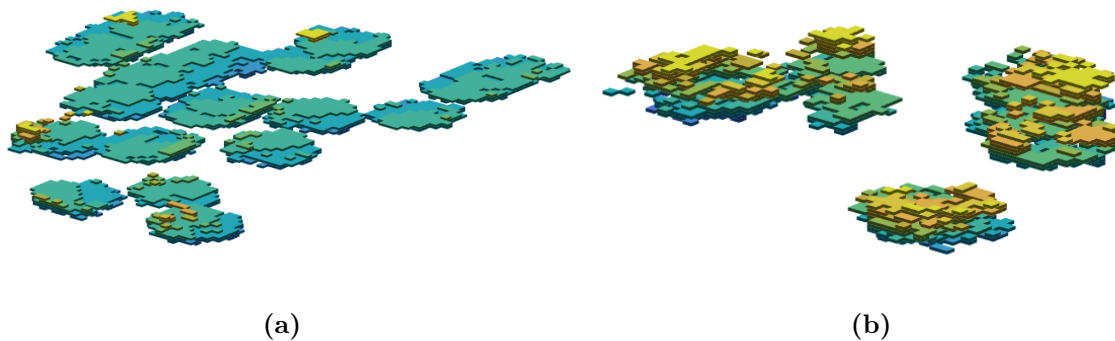


Figura 4.18: Mapas generados en entorno plano y entorno con objetos

- Precisión y fidelidad: Nota 2, tuvo dificultades en posicionar correctamente los muebles en el entorno denso.
- Resolución: Nota 3, al igual que en las configuraciones E y F.
- Cobertura: Nota 1, grandes espacios vacíos en el mapa que si debiesen tener asociados una nube de puntos ya que se paso por ahí en la ruta.

4.2.8. Configuración G ($f=1\text{hz}$ y $d=30$)

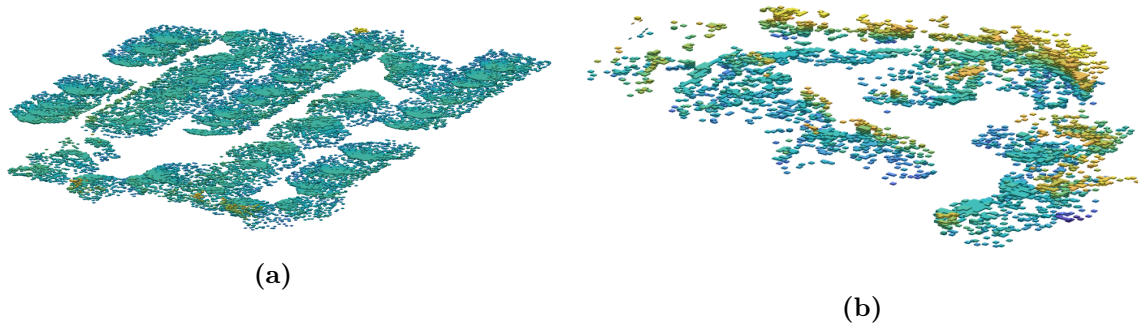


Figura 4.19: Mapas generados en entorno plano y entorno con objetos

- Precisión y fidelidad: Nota 4, de las configuraciones que mejor capturó la curva del mueble en el entorno denso pero de igual forma posiciono incorrectamente un par de nubes de puntos (sillón en el entorno denso).
- Resolución: Nota 4, como la variable de resolución aumento se ve reflejado en este apartado para las configuraciones G, H e I.
- Cobertura: Nota 4, pequeñas zonas en donde no se mapeo. En el entorno denso se ven algo mas de espacios pero la ruta no recorrió todas estas zonas.

4.2.9. Configuración H ($f=15\text{hz}$ y $d=30$)

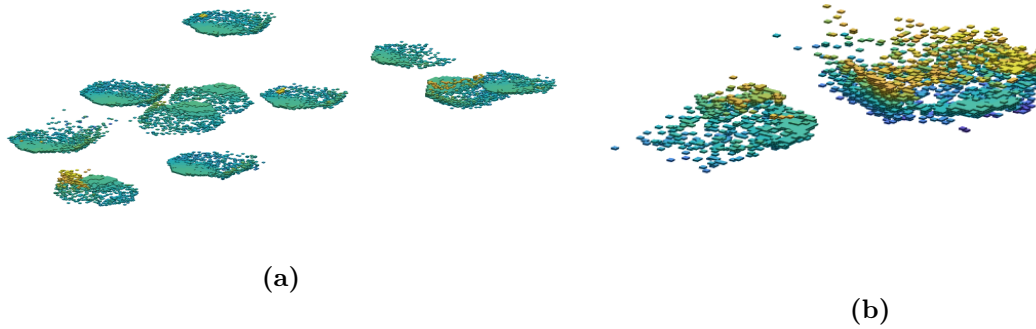


Figura 4.20: Mapas generados en entorno plano y entorno con objetos

- Precisión y fidelidad: Nota 2, si bien el entorno plano se logra vislumbrar, en el entorno denso fracaso completamente en mostrar una representación del entorno.
- Resolución: Nota 4, si bien se mapeo poco, lo poco que se ve tiene buena resolución.
- Cobertura: Nota 1, grandes zonas sin gráficar en el entorno plano y en el entorno mas denso se fracaso rotundamente en este apartado.

4.2.10. Configuración I ($f=50\text{hz}$ y $d=30$)

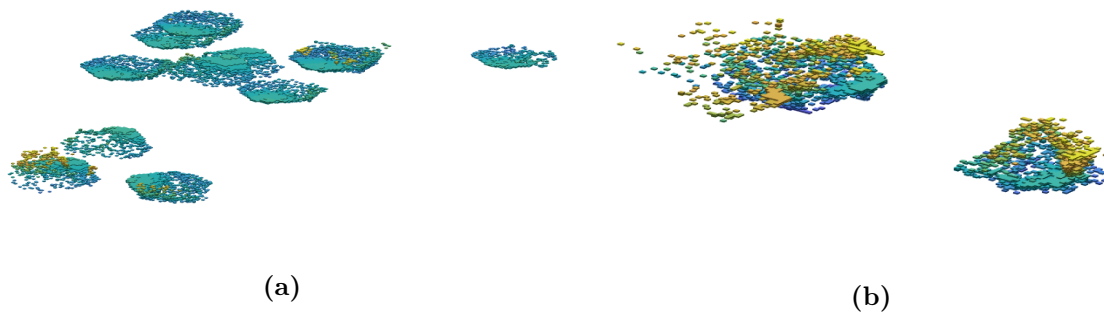


Figura 4.21: Mapas generados en entorno plano y entorno con objetos

- Precisión y fidelidad: Nota 2, muy parecido a la configuración H en este sentido.
- Resolución: Nota 4, lo poco que se ve tiene buena resolución.
- Cobertura: Nota 1, de toda la ruta en el entorno denso solo se graficaron 2 zonas, muy parecido a la configuración H.

4.2.11. Calificaciones de las configuraciones

Ya habiendo revisado cada una de las configuraciones, se promediarían las puntuaciones de cada característica evaluada por configuración, para así obtener una única nota del 1 al 5. Junto con lo anterior:

Tabla 4.4: Matriz de evaluación de resultados

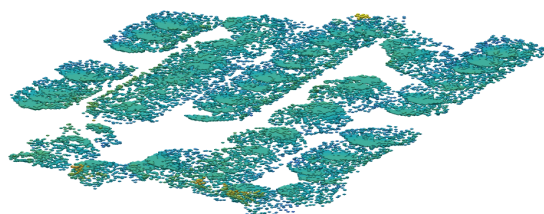
	Frecuencia [Hz]			
	Valores	f=1	f=15	f=50
	Resolución			
	d=5	3.3	2.3	1.7
	d=10	3.7	2	2
	d=30	4	2.3	2.3

Como podemos observar las configuraciones que tienen el valor de frecuencia en 1 hz obtienen los mejores resultados, y a medida que se va aumentando el valor de la resolución los resultados también mejoran, en gran parte debido a este apartado.

4.2.12. Análisis de mejor configuración



(a)



(b)

Figura 4.22: Entorno plano: real y mapa generado

La ruta en el entorno real se trata de un zigzag como se puede ver representado por las flechas rojas en la imagen de la izquierda (a), este recorrido se puede ver representado

en el mapa generado debido a que se ven pequeños espacios entre los diferentes caminos paralelos de la ruta. Las dimensiones del entorno real son de 7.7 x 5 metros, y el mapa generado mostró dimensiones de 8.1 x 5.9 metros, por lo que las dimensiones representadas por el mapa son bastante parecidas al entorno real. Esta diferencia se debe principalmente a errores relacionas con la odometría obtenida del robot, la cual presenta errores de posicionamiento de hasta 1 metro.

Los espacios vacíos que se encuentran por la ruta se deben a que la odometría o el tópicos de /tf (relaciones entre "frames" o partes del robot) no logra mantener una comunicación constante, es decir, la frecuencia con la que envía la posición y orientación (en específico los correspondientes al robot en si "trunk") no se mantiene en un valor constante. Lo anterior trae como resultado el hecho de que a pesar de que se esta moviendo al robot, la odometría no se actualiza necesariamente. Finalmente cuando el robot logra transmitir odometría correctamente el robot ya se salto algunas nubes de puntos.

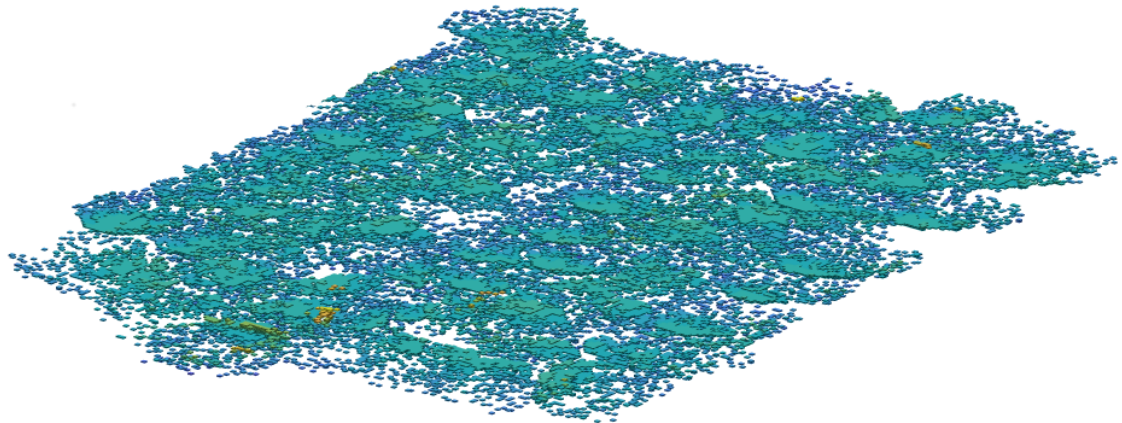


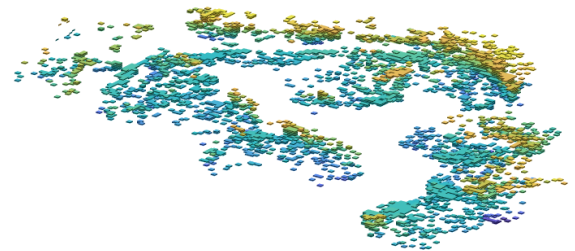
Figura 4.23: Mapa generado con más tiempo del entorno plano

La figura anterior corresponde a la misma configuración pero tomándose mas tiempo de recorrer mas veces el entorno para que de esta forma se pueda generar un mapa mucho mas completo, en específico la ruta de este mapa tomo cerca de 5 minutos (al

rededor del doble que de la muestra de la figura 4.22). Este mapa generado es mucho mejor debido a que se logra evidenciar en su totalidad la superficie plana del entorno.



(a)



(b)

Figura 4.24: Entorno denso: real y mapa generado

La ruta en el entorno denso busca el contorno de los muebles, básicamente consta de 2 vueltas, la primera vuelta recorre el mueble curvo exterior y en la segunda se busca recorrer el contorno del mueble central. Una de las características que mejor se ven representadas es la curva del mueble exterior, el cual en el entorno real mide al rededor de x metros, y en el mapa generado 5.5 metros. Otro mueble importante es el central, el cual si bien no se gráfica totalmente se puede visualizar uno de los segmentos, el cual en el mapa generado mide 2.23 metros, mientras que en el entorno real mide 2 metros. De igual forma que en el entorno plano las dimensiones representadas en el mapa son bastante parecidas al entorno real.

En este entorno se perciben muchos mas espacios vacíos que en el entorno plano, esta diferencia se debe en gran medida a que el entorno presenta problemas para hacer una ruta que abarque los diferentes ángulos y segmentos de los muebles, por lo que esto sumado a la razón descrita para el entorno plano genera espacios mas grandes sin mapear.

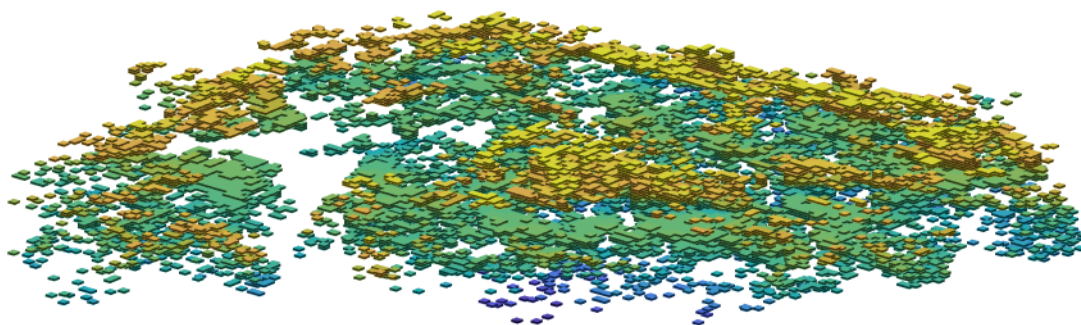


Figura 4.25: Mapa generado con más tiempo del entorno denso

Debido a la complejidad del entorno siguen habiendo espacios sin mapear, a pesar de recorrer por mas tiempo el entorno. Aunque de todas formas se logra visualizar de mejor forma los muebles, es decir, la completitud del mueble curvo se visualiza en el mapa y se logra distinguir un objeto triangular en medio del entorno.

5. Conclusiones

En este trabajo se aborda el mapeo de entornos cerrados usando cámaras RGB-D mediante el uso de un robot cuadrúpedo. Debido a que las características del proyecto necesitaban hacer una prueba de concepto en un entorno simulado se decidió implementar una solución por medio de un paquete de ROS que se encargue de realizar el mapeo con cámaras RGB-D, de esta forma se lograría trabajar con una "caja negra" que como input reciba los tópicos que sean necesarios del robot y entregue como salida el mapa del entorno. Si bien esta prueba de concepto resultó funcionar excelente con el entorno simulado y RTAB-Map, al momento de intentar implementarlo en el robot real se tuvieron demasiadas complicaciones, pero debido a que se han logrado resultados increíbles en el entorno simulado, la idea de que se puede lograr esta implementación con RTAB-Map sigue en pie, y debido a que es un paquete que realiza SLAM se hubiese podido lograr mas que un mapa.

Tras haber fracasado en la implementación de RTAB-Map con el robot real se decidió simplificar la idea y utilizar solo los tópicos de nube de puntos y odometría, los cuales son tópicos que normalmente el robot levanta automáticamente pero el robot estaba teniendo problemas para levantarlos, por lo que se tuvo que formatear al estado de fabrica, lo cual se logro de mano con Unitree, quienes proporcionaron una imagen de la raspberry pi que trae de fabrica el robot. Tras devolverlo al estado de fabrica y obtener los tópicos necesarios, se implementó rápidamente un script en Matlab con el cual se logró finalmente mapear el entorno del robot a medida que recorre un entorno.

Como se pudo ver en la sección de resultados, las configuraciones que lograron un mejor mapa en general fueron las que tienen un valor de frecuencia de 1 hz. A medida que se iba aumentando la frecuencia, el aspecto que mas se ve perjudicado es el de la cobertura, lo cual se debe a que el script de matlab utiliza ese valor de frecuencia para definir cada cuanto se gráfica en el mapa las nubes de puntos (las cuales llegan a una frecuencia mas alta que la odometría), al aumentar la frecuencia de escritura en el mapa se pone mas atención en cada iteración a la escritura en el mapa que a recibir la odometría, por lo

que esto se traduce en que si bien se escribe a mas frecuencia en el mapa, a la vez se consulta con menos frecuencia por la odometría. Debido a lo anterior se generan esas lagunas vacías en el mapa generado.

Si bien la configuración que obtuvo un mejor resultado plasma las características principales de los entornos, de todas formas sigue teniendo grandes debilidades al momento de intentar utilizarlo en las aplicaciones que se tenían pensado en un principio. Al tener problemas para cubrir espacios y posicionar correctamente objetos en ocasiones, no se podría utilizar esta implementación para situaciones de delicadas o de riesgo en donde se necesite una alta precisión para poder lograr tareas complejas. Aunque para situaciones en donde se necesite una representación a grandes rasgos del entorno podría servir. Otra debilidad que tiene esta implementación es la falta de color en el mapa, ya que si se tuviera el color de cada punto que ven los sensores del robot se podría sobrellevar de mejor manera el hecho de que tenga pequeños fallos en el posicionamiento de las nubes de puntos, debido a que en el caso de que se vean puntos de un mismo color se asumiría que ese grupo de puntos representan un mismo objeto.

Considerando las limitaciones y fortalezas de la implementación descrita para el mapeo de entornos con un robot cuadrúpedo usando cámaras RGB-D y el script en Matlab, es posible sugerir aplicaciones prácticas donde estas características puedan ser aprovechadas adecuadamente, aun con las debilidades mencionadas. Una posible aplicación de esta tecnología sería en la exploración preliminar y mapeo de áreas grandes donde la precisión absoluta en la ubicación de objetos no es crítica, pero sí lo es obtener una visión general del área. Esto podría incluir, por ejemplo, la exploración de zonas de construcción antes de iniciar un proyecto, o la inspección de áreas naturales para la planificación de actividades de conservación ambiental. En estos casos, la capacidad de generar rápidamente un mapa que refleje las características principales del terreno puede ser más valiosa que la precisión detallada en la ubicación de cada objeto.

Otra aplicación podría ser en el ámbito de la seguridad y vigilancia, donde el robot podría patrullar áreas predeterminadas y proporcionar actualizaciones periódicas del

mapeo del entorno. Aunque la falta de precisión en la ubicación exacta de objetos podría ser una limitación, la capacidad de actualizar constantemente un mapa del área podría ser útil para detectar cambios o intrusiones en entornos de baja complejidad.

Finalmente, este sistema también podría ser útil en contextos educativos o de investigación, donde los estudiantes o investigadores necesiten familiarizarse con tecnologías de mapeo y robótica sin requerir la precisión que sería necesaria en aplicaciones industriales o de alta seguridad. En estos entornos, la implementación podría servir como una herramienta valiosa para el aprendizaje y experimentación con tecnologías de mapeo y navegación autónoma.

Referencias

- [1] Unitree, “Unitree go 1 edu guide.” [Online]. Available: https://unitree-docs.readthedocs.io/en/latest/get_started/Go1_Edu.html
- [2] J. S. D. C. W. B. Felix Endres, Jürgen Hess, “3-d mapping with an rgb-d camera,” *IEEE Transactions on Robotics*, vol. 30, pp. 177–187, 2014.
- [3] M. K. J. J. L. J. M. Thomas Whelan, Hordur Johannsson, “Robust real-time visual odometry for dense rgb-d mapping,” *2013 IEEE International Conference on Robotics and Automation*, 2013.
- [4] M. F. Yiduo Wang, Milad Ramezani, “Actively mapping industrial structures with information gain-based planning on a quadruped robot,” *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020.
- [5] Unitree, “Guide to use the gazebo model.” [Online]. Available: https://github.com/unitreerobotics/unitree_guide
- [6] Introlab, “Installation of rtabmap and rtabmap_ros.” [Online]. Available: https://github.com/introlab/rtabmap_ros/tree/melodic-devel
- [7] ROS, “Writing the publisher node.” [Online]. Available: <https://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29>
- [8] unitreerobotics, “Get rectify frame.” [Online]. Available: https://github.com/unitreerobotics/UnitreecameraSDK/blob/main/examples/example_getRectFrame.cc
- [9] —, “Get depth frame.” [Online]. Available: https://github.com/unitreerobotics/UnitreecameraSDK/blob/main/examples/example_getDepthFrame.cc
- [10] ROS, “Ros camera_info.” [Online]. Available: https://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/CameraInfo.html
- [11] M. Labbe, “Rtab-map maps.” [Online]. Available: https://github.com/matlabbe/rtabmap_drone_example/tree/master

6. Anexos

6.1. Launch file para simulación

```

4  <?xml version="1.0"?>

  <launch>

    <!-- See https://github.com/unitreerobotics/unitree_guide to bringup simulation.
      Fix Cx/Cy of the cameras by setting them to 464 and 400 respectively in
9   unitree_ros/robots/go1_description/xacro/depthCamera.xacro
      Ideally, build rtabmap with OpenGV support.
      Would work better if simulated environment has a lot of visual texture, see
      https://github.com/introlab/rtabmap_ros/issues/1031#issuecomment-1722322305

14   Launch:
      $ source ./devel/setup.bash
      $ roslaunch unitree_guide gazeboSim.launch wname:=apt
      $ roslaunch unitree_gazebo unitree_quadruped_robot.launch
      $ ~/Memoria/devel/lib/unitree_guide/junior_ctrl
19   Press 2 to get up, press 4 to move (w,a,s,d) and rotate (j,l)
      -->

    <arg name="localization" default="false" />
    <arg if="$(arg localization)" name="rtabmap_args" default="--Mem/IncrementalMemory
      false" />
24   <arg unless="$(arg localization)" name="rtabmap_args"
      default="--Mem/IncrementalMemory true --delete_db_on_start" />

    <!-- sync rgb/depth images and camera info per camera -->
    <group ns="camera_face">
29   <node pkg="rtabmap_sync" type="rgbd_sync" name="rgbd_sync">
      <remap from="rgb/image" to="color/image_raw" />
      <remap from="depth/image" to="depth/image_raw" />
      <remap from="rgb/camera_info" to="color/camera_info" />
      <param name="approx_sync" value="false" />
34   </node>
    </group>

    <group ns="rtabmap">

```

```

39  <!-- Odometry -->
    <node pkg="rtabmap_odom" type="rgbd_odometry" name="rgbd_odometry" output="screen"
    >
        <remap from="imu" to="/trunk_imu" />
        <remap from="rgbd_image" to="/camera_face/rgbd_image" />
        <param name="subscribe_rgbd" type="bool" value="true" />
44  <param name="frame_id" type="string" value="base" />
        <param name="rgbd_cameras" type="int" value="1" />
        <param name="wait_for_imu_to_init" type="bool" value="true" />
    </node>

49  <!-- Visual SLAM -->
    <node name="rtabmap" pkg="rtabmap_slam" type="rtabmap" output="screen"
        args="$(arg rtabmap_args)">
        <remap from="imu" to="/trunk_imu" />
        <remap from="rgbd_image" to="/camera_face/rgbd_image" />
54  <param name="subscribe_depth" type="bool" value="false" />
        <param name="subscribe_rgbd" type="bool" value="true" />
        <param name="rgbd_cameras" type="int" value="0" />
        <param name="frame_id" type="string" value="base" />

59  <param name="Grid/RangeMin" type="string" value="0.1" /> <!-- to avoid adding
    legs as obstacle
    in occupancy grid map -->
        <param name="Grid/MaxObstacleHeight" type="string" value="1" />
    </node>

64  <!-- Visualisation RTAB-Map -->
    <node pkg="rtabmap_viz" type="rtabmap_viz" name="rtabmap_viz"
        args="-d $(find rtabmap_demos)/launch/config/rgbd_gui.ini" output="screen">
        <remap from="rgbd_image" to="/camera_face/rgbd_image" />
        <param name="subscribe_depth" type="bool" value="false" />
69  <param name="subscribe_rgbd" type="bool" value="true" />
        <param name="subscribe_odom_info" type="bool" value="true" />
        <param name="frame_id" type="string" value="base" />
        <param name="rgbd_cameras" type="int" value="1" />
    </node>

74  </group>

</launch>

```

6.2. Launch file para implementación RTAB-Map en robot real

```

1
<launch>

  <!-- See https://github.com/unitreerobotics/unitree_guide to bringup simulation.
    Fix Cx/Cy of the cameras by setting them to 464 and 400 respectively in
6 unitree_ros/robots/go1_description/xacro/depthCamera.xacro
    Ideally, build rtabmap with OpenGV support.
    Would work better if simulated environment has a lot of visual texture, see
    https://github.com/introlab/rtabmap_ros/issues/1031#issuecomment-1722322305

11    Launch:
        $ source ./devel/setup.bash
        $ roslaunch unitree_guide gazeboSim.launch wname:=apt
        $ roslaunch unitree_gazebo unitree_quadruped_robot.launch
        $ ~/Memoria/devel/lib/unitree_guide/junior_ctrl
16    Press 2 to get up, press 4 to move (w,a,s,d) and rotate (j,l)
        -->

    <arg name="localization" default="false" />
    <arg if="$(arg localization)" name="rtabmap_args" default="--Mem/IncrementalMemory
      false" />
21    <arg unless="$(arg localization)" name="rtabmap_args"
      default="--Mem/IncrementalMemory true --delete_db_on_start" />

    <!-- sync rgb/depth images and camera info per camera -->
    <group ns="camera_face">
26      <node pkg="rtabmap_sync" type="rgbd_sync" name="rgbd_sync">
        <remap from="rgb/image" to="color" />
        <remap from="depth/image" to="depth" />
        <remap from="rgb/camera_info" to="camera_info" />
        <param name="approx_sync" value="false" />
31      </node>
    </group>

    <group ns="rtabmap">
36      <!-- Odometry -->
      <node pkg="rtabmap_odom" type="rgbd_odometry" name="rgbd_odometry" output="screen"
        >
        <remap from="imu" to="/trunk_imu" />
        <remap from="rgbd_image" to="/camera_face/rgbd_image" />

```

```
41     <param name="subscribe_rgbd" type="bool" value="true" />
    <param name="frame_id" type="string" value="base" />
    <param name="rgbd_cameras" type="int" value="1" />
    <param name="wait_for_imu_to_init" type="bool" value="true" />
  </node>

46

  <!-- Visual SLAM -->
  <node name="rtabmap" pkg="rtabmap_slam" type="rtabmap" output="screen"
    args="$(arg rtabmap_args)">
    <remap from="imu" to="/trunk_imu" />
51    <remap from="rgbd_image" to="/camera_face/rgbd_image" />
    <param name="subscribe_depth" type="bool" value="false" />
    <param name="subscribe_rgbd" type="bool" value="true" />
    <param name="rgbd_cameras" type="int" value="0" />
    <param name="frame_id" type="string" value="base" />

56
    <param name="Grid/RangeMin" type="string" value="0.1" /> <!-- to avoid adding
    legs as obstacle
    in occupancy grid map -->
    <param name="Grid/MaxObstacleHeight" type="string" value="1" />
  </node>

61

  <!-- Visualisation RTAB-Map -->
  <node pkg="rtabmap_viz" type="rtabmap_viz" name="rtabmap_viz"
    args="-d $(find rtabmap_demos)/launch/config/rgbd_gui.ini" output="screen">
    <remap from="rgbd_image" to="/camera_face/rgbd_image" />
66    <param name="subscribe_depth" type="bool" value="false" />
    <param name="subscribe_rgbd" type="bool" value="true" />
    <param name="subscribe_odom_info" type="bool" value="true" />
    <param name="frame_id" type="string" value="base" />
    <param name="rgbd_cameras" type="int" value="1" />
71  </node>

  </group>

76 </launch>
```

6.3. Script para implementación de matlab

```

rosshutdown;
rosinit('http://192.168.123.161:11311');

4   go1_odom = rossubscriber('/tf');
    fqGet = 1000; %frequency for getting topics
    maxRange = 5;

9   go1_pc_face = rossubscriber('/camera1/point_cloud_face');

    tStartOdom = tic;
    [odomTF, status1, statusText1] = receive(go1_odom);
    [pc_face, status2, statusText2] = receive(go1_pc_face);

14  map3D = occupancyMap3D(1);

19  while 1
    [go1_test_msg, status1, statusText1] = receive(go1_odom);

    if strcmp(go1_test_msg.Transforms.ChildFrameId, 'trunk')
        odomTF = go1_test_msg;
24  end

    tEndOdom = toc(tStartOdom);
    %disp(1/tEndOdom)
    if round(1/tEndOdom) <= fqGet
29      [pc_face, status2, statusText2] = receive(go1_pc_face);
        %showdetails(odomTF);
        %showdetails(pc_face);

34      xPos = odomTF.Transforms.Transform.Translation.X;
        yPos = odomTF.Transforms.Transform.Translation.Y;
        zPos = odomTF.Transforms.Transform.Translation.Z;

        xRot = odomTF.Transforms.Transform.Rotation.X;
39      yRot = odomTF.Transforms.Transform.Rotation.Y;
        zRot = odomTF.Transforms.Transform.Rotation.Z;
        wRot = odomTF.Transforms.Transform.Rotation.W;

```

```
pose = [ xPos yPos zPos wRot xRot yRot zRot];  
44 points = readXYZ(pc_face);  
pc = pointCloud(pc_face.readXYZ);  
  
pc.Color = pc_face.readRGB;  
  
49 insertPointCloud(map3D,pose,pc,maxRange);  
show(map3D);  
  
fprintf('%d HZ\n',round(1/tEndOdom));  
54 disp('-----');  
tStartOdom = tic;  
  
end  
end
```