UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROGRAMA DE MAESTRÍA Y DOCTORADO EN INGENIERÍA
INGENIERÍA ELÉCTRICA – PROCESAMIENTO DIGITAL DE SEÑALES

GENERACIÓN DE MAPAS TOPOLÓGICOS USANDO INFORMACIÓN TRIDIMENSIONAL APLICADO A LA NAVEGACIÓN ROBÓTICA.

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN INGENIERÍA

PRESENTA:
JESÚS HERNÁNDEZ COYOTZI

TUTOR PRINCIPAL
DR. JESÚS SAVAGE CARMONA
FACULTAD DE INGENIERÍA

MÉXICO, CD. MX. , JUNIO 2019

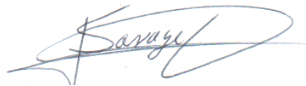
JURADO ASIGNADO:

Presidente:	Dr. Rascón Estebané Caleb
Secretario:	Dr. Pérez Alcázar Pablo Roberto
1 er. Vocal:	Dr. Savage Carmona Jesús
2 do. Vocal:	M.I. escobar Salguero Larry
3 er. Vocal:	Dr. Rivera Rivera Carlos

Lugar o lugares donde se realizó la tesis: Ciudad Universitaria

TUTOR DE TESIS:

DR. JESÚS SAVAGE CARMONA



FIRMA

Resumen

En el campo de la robótica, la mayoría de los algoritmos existentes dependen de contar con un mapa preciso de su ambiente. Es decir, una forma de representar correctamente el medio donde opera el agente. Actualmente, para la mayoría de los robots restringidos a operar en una superficie plana, una representación bidimensional se utiliza para crear mapas. Esto permite que el agente se localice y planee en el espacio del mapa. Sin embargo, los mapas bidimensionales tienen sus desventajas. Primero, algunos objetos comunes como sillas y mesas son omitidos o su representación es mala. Además, esta representación no puede manejar algunos otros algoritmos de alto nivel como la planeación de movimientos para manipuladores robóticos en ambientes desordenados. Finalmente, estas representaciones tienen requisitos altos de memoria.

En este documento presentamos a Sparse-Mapper un sistema de mapeo tridimensional, modular y expandible que permite la creación automática de mapas compactos del ambiente donde opera un robot mediante el uso de técnicas de aprendizaje no supervisado en nubes de puntos no estructuradas. Esta metodología permite generar caminos válidos para que el robot navegue, en tiempos comparables con planeadores de trayectorias bidimensionales, pero manteniendo una representación completamente tridimensional del espacio. Este sistema se evalúa como un planeador de trayectorias pero argumentamos que puede ser fácilmente extendido a planeación de movimientos y reconocimiento de objetos por nombrar algunas posibilidades. El sistema está validado sobre diferentes juegos de datos públicos y se encuentra basado en el popular meta-sistema operativo ROS para robots.

Summary

In the field of robotics most algorithms depend on having an accurate map of it's enviroment to function. That is, a way to accurrately represent the world where the agent operates. Currently, for most robots constrained to operate on a flat surface, a two-dimensional representation is used. This allows the agent to localize itself and to plan on the map space. However a bidimensional map has several shortcomings. First of all, some objects like chairs and tables are usually ommited from such maps or are poorly represented in them. Moreover, this representation can't handle other high level algorithms like motion planning on robotic manipulators on cluttered environments. Finally these representations tend to have high memory requirements. In this document we present Sparse-Mapper a modular and extendable tridimensional mapping pipeline that allows to automatically create a compact map of the robot's enviroment while keeping a small memory footprint by using unsupervised learning techniques on unstructured point clouds. This method allows to create valid paths for our agent to follow in times comparable to bidimentional motion planners, while maintaining a full 3D representation of the world. The systems is evaluated as a path planning framework but we argue it can be easily extended to motion planning, and object recognition tasks to name a few. This approach is validated over several datasets and is based on the popular ROS meta-operative system for robots.

Dedicatoria

A mi madre y a mi padre por su apoyo a lo largo de toda mi vida.

Agradecimientos

Quiero agradecer a la Universidad Nacional Autónoma de México por brindarme una educación superior.
Al laboratorio de Biorobótica por haberme aceptado a su grupo, ayudarme y permitirme colaborar.
A CONACYT por permitirme continuar con mis estudios.
Al Doctor Savage Carmona por ser mi asesor.
Al Doctor Marco Negrete por su ayuda al revisar este manuscrito
Al ingeniero Edgar Silva por enseñarme a utilizar los robots
Al Maestro Carlos Adrián por ayudarme a entender los algoritmos de aprendizaje.
A Toyota Motor Company por el préstamo del robot.
Y a DGAPA por la beca PAPIIT: AG100818

Índice general

1. Introducción **11**
1.1. Motivación 11
1.2. Hipótesis 12
1.3. Objetivos 12
1.4. Organización de la tesis 12

2. Marco Teórico **13**
2.1. Robótica probabilística 13
2.2. Estado 14
2.3. Distribución de la creencia 15
2.4. Cinemática y mediciones 15
2.5. Mapas y mapeo 16
2.5.1. Celdas de ocupación 17
2.6. Mapeo tridimensional 19
2.6.1. Láseres y Cámaras de profundidad 19
2.6.2. Representación tridimensional 22
2.7. Localización y mapeo simultaneo 25
2.7.1. SLAM 3D 27

3. Descripción del sistema **33**
3.1. Visión general del sistema 33
3.1.1. Adquisición de datos 34
3.1.2. Localización 34
3.1.3. Preprocesamiento 35
3.1.4. Cuantización del espacio 36
3.1.5. Mapeo 40
3.1.6. Servidor de Mapas 42
3.1.7. Servidor de Trayectorias 47

3.1.8. Soporte de Voxeles 51

4. Detalles de Implementación 55
4.1. Procesamiento de nubes de puntos 55
4.2. Localización 56
4.3. Implementación en CUDA 56
4.3.1. Servidor de mapas 58

5. Pruebas y Resultados 59
5.1. Celdas de ocupación aumentadas mediante Octomapas 59
5.2. Pruebas de cuantización 60
5.2.1. Resultados 65
5.2.2. Tablas 65
5.2.3. Gráficas 68
5.3. Pruebas de trayectorias 72
5.3.1. Tablas 76
5.3.2. Gráficas 80
5.4. Compresión del espacio 89
5.5. Análisis de los resultados 90
5.5.1. Agrupamiento 90
5.6. Planeación de trayectorias 91
5.7. Compresión 92
5.8. Limitaciones 92

6. Conclusiones y trabajo a futuro 95
6.1. Conclusiones 95
6.2. Trabajo futuro 96

A. Apendices 103
A.1. Archivos 103
A.2. Glosario 105

Capítulo 1

Introducción

1.1. Motivación

En la robótica uno de los problemas más importantes es la representación del ambiente. Es decir, cómo guardar en memoria las características geométricas y semánticas del lugar donde opera nuestro robot o agente. O dicho con otras palabras ¿Cómo generar un mapa que represente adecuadamente el medio?
Este problema se requiere resolver como primer paso para atacar problemas de más alto nivel como la localización y la planeación de trayectorias. Estos mapas son generados a partir de algoritmos de localización y mapeo simultáneos o SLAM, por sus siglas en inglés y han sido una línea de investigación fuerte en la robótica en las últimas décadas. Estos algoritmos generan representaciones bidimensionales y densas que requieren mucha memoria y poder de cómputo para funcionar, así como hardware especializado como sensores láser. Sin embargo, a pesar de su complejidad estos métodos de mapeo tienden a omitir o representar de forma inadecuada ciertos objetos comunes como sillas y mesas ya que no son capaces de captar su estructura completa.
Por otro lado los algoritmos más nuevos son capaces de usar las relativamente nuevas cámaras de profundidad, como el Kinect o sensores Velodyne, para generar una representación más completa del medio, sin embargo son computacionalmente más costosos que los anteriores ya que manejan más información.
Estas desventajas llevan a problemas al utilizar estos mapas en otros algoritmos. Por ejemplo: Un planeador de trayectorias puede generar rutas no válidas que atraviesen obstáculos que no fueron captados adecuadamente por los algoritmos de mapeo. Sería entonces deseable contar con una representación alternativa del espacio de trabajo que fuera compacta y capaz de tomar en cuenta los obstáculos que otros

algoritmos omiten.

1.2. Hipótesis

Considerando lo anterior ¿Es posible desarrollar un mapa tridimensional compacto del ambiente? Es decir, que no requiere usar toda la información que proveen los sensores, pero que todavía sea capaz de generar trayectorias válidas que el robot pueda seguir.

1.3. Objetivos

Con base en lo anterior, se propone implementar y evaluar un algoritmo que genere una representación adecuada del espacio de trabajo de un robot. Entre las características deseadas para este algoritmo se encuentran:

- Ser capaz de captar obstáculos tridimensionales que resultan difíciles de representar por otros algoritmos, como celdas de ocupación.
- Obtener una representación compacta del medio.
- Lograr que esta representación obtenida derive en una planeación de trayectorias rápida y eficiente.

1.4. Organización de la tesis

A continuación se dará una breve descripción de la estructura del presente texto: En el capítulo 1, la introducción, se discuten la estructura objetivos y motivación del presente trabajo. En el capítulo 2, marco teórico se presentan la ideas y conceptos que fundamentan el trabajo desarrollado, así como otras técnicas y algoritmos previamente implementados que ataquen el mismo problema. En el capítulo 3, descripción del sistema se detalla el sistema propuesto, bautizado como Sparse-Mapper. Además en el capítulo 4, se dan detalles de la implementación, librerías y paquetería utilizada. Después, en el capítulo 5, pruebas y resultados, se examinan las pruebas propuestas para validar el sistema y se presentan las tablas y gráficas generadas. Al final de este se discuten los resultados obtenidos. Finalmente, en el capítulo 6, se tienen las conclusiones y el trabajo futuro.

Capítulo 2

Marco Teórico

En este capítulo se describe la teoría, conceptos y definiciones necesarias para sustentar el resto del trabajo. Se analizan algunos elementos de robótica probabilística y de mapeo necesarios para entender el problema central, se describen a su vez algunas representaciones populares de mapas tanto planos como tridimensionales. Se termina el capítulo haciendo una breve descripción del estado del arte para generación automática de mapas.

2.1. Robótica probabilística

Antes de comenzar el tema central que compete a este documento, el mapeo robótico, se definirán algunos conceptos de robótica probabilística que serán utilizados a lo largo de todo el texto.
En el pasado la mayoría de los robots operaban en ambientes controlados y estructurados, como plantas industriales, realizando tareas de ensamble, generalmente repetitivas. Sin embargo, en muchos problemas modernos, como los vehículos autónomos y robots de servicio se requiere que los robots o agentes operen de manera reactiva en ambientes altamente dinámicos y poco predecibles [1].
En estas nuevas tareas y ambientes, el robot carece de información para llevar a cabo las tareas asignada. A esta falta de información se le conoce como *incertidumbre* y para el caso de robots surge de al menos 5 factores [2]:

- El ambiente.
- Los sensores.
- Los actuadores.

- Los modelos.
- El computo.

Todos estos añaden algún grado de incertidumbre a un sistema robótico, el cual no puede ser eliminado del todo. Debido a esto surge el enfoque probabilístico de la robótica alrededor de los noventas. Este enfoque se basa en la idea de Thrun [2]:

"Un robot que tiene la noción de su propia incertidumbre y actúa de acuerdo a esta es superior a uno que no lo hace"

2.2. Estado

En el contexto de la robótica un estado es la colección de los aspectos del ambiente y el robot que son relevantes. Estos pueden ser dinámicos, como la posición de personas en un cuarto, o estáticos como las paredes y muebles del mismo. Se representan generalmente mediante x_t donde t indica el tiempo donde el estado es válido y el contenido específico de x depende del contexto, puede ser velocidad, posición o una combinación de ambas, etc.
Podemos agrupar el contenido del estado en dos categorías: datos de mediciones y datos de control. Los datos de mediciones proveen información acerca del estado instantáneo del ambiente, como sensores de distancia, cámaras etc. Se representa mediante la siguiente notación [2]

$$z_{t_1:t_2} = z_{t_1+1}, z_{t_1+2}, ..., z_{t_2} \tag{2.1}$$

Donde z_{t_i} es la medición, o mediciones, en el tiempo t_i y $z_{t_1:t_2}$ es el conjunto de mediciones de t_1 a t_2 dado que $t_1 \leq t_2$.
Los datos de control indican el cambio en el agente, por ejemplo, la velocidad o la odometría del robot. De forma similar, a las mediciones se les representa de la siguiente manera:

$$u_{t_1:t_2} = u_{t_1+1}, u_{t_1+2}, ..., u_{t_2} \tag{2.2}$$

Con la salvedad que un control u_t representa el cambio del estado entre $(t-1, t]$ y no el valor instantáneo. Los controles y las mediciones son estocásticos, por lo tanto la evolución del estado está determinada por una ley probabilística de la forma:

$$p(x_t|x_{0:t-1}, z_{1:t-1}, u_{1:t}) \tag{2.3}$$

Sin embargo, si el estado es un estadístico suficiente, podemos reducir la expresión anterior conociendo simplemente el estado anterior. A esto también se le conoce como suposición Markoviana.

$$p(x_t|x_{0:t-1}, z_{1:t-1}, u_{1:t}) = p(x_t|x_{t-1}, u_t) \tag{2.4}$$

A esto se le conoce como probabilidad de transición de estado y modela cómo el ambiente evoluciona como función de los controles del robot.
De forma similar, podemos utilizar el estado para predecir la medición actual. Con lo cual tenemos:

$$p(x_t|x_{0:t-1}, z_{1:t-1}, u_{1:t}) = p(z_t|x_t) \tag{2.5}$$

A esto se le conoce como probabilidad de medición.

2.3. Distribución de la creencia

Debido a la naturaleza estocástica del ambiente es imposible medir el estado de forma directa y debe ser inferido de mediciones. Entonces diferenciamos el estado real de la *creencia o belief* interna del robot. Una distribución de creencia asigna una probabilidad a cada hipótesis del valor real del estado. Esta es una distribución *a posteriori* que se denota $bel(x_t)$.

$$bel(x_t) = p(x_t|z_{1:t}, u_{1:t}) \tag{2.6}$$

Es decir, la creencia del estado esta determinada por todas las mediciones y acciones pasadas. En ocasiones es útil calcular la creencia antes de incorporar nueva información de la medición. A esto se le llama predicción:

$$\overline{bel(x_t)} = p(x_t|z_{1:t-1}, u_{1:t}) \tag{2.7}$$

Finalmente, al proceso de calcular la creencia a partir de la predicción se conoce como corrección o actualización de medidas.

2.4. Cinemática y mediciones

La cinemática del robot es la descripción de la forma en que las acciones de control u_t modifican la configuración del robot. En el espacio, el estado cinemático o *pose* se define en seis dimensiones, tres coordenadas cartesianas y tres ángulos de Euler. Sin embargo para muchos robots que operan en ambientes planos la pose esta

completamente definida mediante dos coordenadas lineales y un ángulo llamado a veces dirección [1]. Es decir:

$$x_t = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \tag{2.8}$$

En el caso de utilizar solo la posición del robot, ésta se conoce como la localización del mismo.
Por otro lado, en el caso de las mediciones, muchos sensores ofrecen un arreglo de varias medidas al mismo tiempo, en lugar de un solo valor. Entonces:

$$z_t = z_t^1, ..., z_t^k \tag{2.9}$$

Esto expresa el conjunto de k mediciones en el tiempo t. En el contexto de robótica probabilística nos interesa saber la verosimilitud de un conjunto de medidas [1] dada la pose para modelar el ruido inherente al sensor y compensar. Esto se expresa como:

$$p(z_t|x_t, m) = \prod_{k=1}^{K} p(z_t^K|x_t, m) \tag{2.10}$$

Es decir la probabilidad de tener un conjunto de mediciones z_t dada la pose del robot x_t y un mapa m. De estos hablaremos a continuación.

2.5. Mapas y mapeo

El mapeo consiste en el proceso de lograr que un agente o robot genere una representación adecuada del ambiente donde opera. La mayoría de los robots modernos dependen de mapas para su localización, planeación de trayectorias, planeación de acciones, entre otras tareas.
En esta sección se desarrolla la teoría necesaria para la implementación del sistema de mapeo propuesto, desde las definiciones de mapeo robótico hasta algunos de los algoritmos más usados para solucionar este problema.
Un mapa es una lista de objetos en el ambiente y su localización. Formalmente un mapa es una lista de objetos y sus propiedades, descrito como:

$$m = \{m_1, m_2, m_3, ..., m_n\} \tag{2.11}$$

Donde n es el total de objetos en el ambiente y m_n es una propiedad. Existen dos tipos de mapas, los mapas basados en *características* y los mapas basados en *ubicación* [2]. Los mapas basados en ubicación definen información en cada punto del

espacio y son capaces de modelar el espacio libre o donde no hay nada y el ocupado. Por otro lado los mapas de características solo definen el ambiente en ciertas posiciones. En otras palabras, se pueden considerar los primeros como representaciones densas y los segundos como dispersas.
Uno de los problemas clásicos de la robótica es el de localización, es decir, dado un mapa obtener la pose del robot. Este es un problema sencillo comparado con el problema del mapeo. Este problema es de una muy alta dimensionalidad pues se debe estimar todo el mapa sobre el espacio de todos los mapas posibles. Asimismo para crear un buen mapa se requiere conocer la pose del robot pero para conocer la pose del robot se requiere un mapa. Este último detalle lleva al problema de Localización y Mapeo Simultáneo donde se estima tanto la pose del agente como el mapa al mismo tiempo. Este problema se tratará más adelante en este documento. En esta sección sólo se desarrollará el problema de mapeo con poses conocidas, pues es la base para el trabajo presentado. Formalmente un algoritmo de mapeo debe calcular la probabilidad posterior de los mapas dada la información suministrada:

$$p(m|z_{1:t}, x_{1:t}) \tag{2.12}$$

Donde m es el mapa, $z_{1:t}$ es el conjunto de mediciones desde el inicio de operación hasta un tiempo arbitrario t y $x_{1:t}$ es la secuencia de poses o el camino que ha tomado el agente [2].

2.5.1. Celdas de ocupación

En sentido estricto, el mapa m está definido en un espacio continuo, el cual no se puede representar en una computadora digital de memoria finita, por lo que una opción es usar una discretización del espacio en celdas de tamaño fijo m_i. Entonces, cada celda representa una variable aleatoria binaria que modela la ocupación del espacio. Donde $p(m_i = 1)$ o $p(m_i)$ es la probabilidad de que la celda se encuentre ocupada [3]. En este caso, el mapa se conforma como:

$$m = \sum m_i \tag{2.13}$$

Un ejemplo de este tipo de mapas se aprecia en la figura 2.1. Una de las ventajas de este método es que permite una fácil visualización al interpretar el mapa como una imagen. Las zonas oscuras representan el espacio ocupado, las claras el espacio libre y las zonas grises son lugares donde no se tiene información.
Cabe mencionar que hay algunas suposiciones importantes que se toman en cuenta este método. La primera es que el espacio está completamente libre o completamente

Figura 2.1: Un mapa a partir de celdas de ocupación.

ocupado, la segunda es que el ambiente es estático y por último cada celda se considera independiente de las otras. Además, los mapas generados bajo este método son planos, es decir una rebanada de dos dimensiones del mundo tridimensional.
A pesar de toda esta simplificación, las celdas de ocupación suelen ser muy grandes, de alrededor de 2^{10} celdas por mapa, por lo que sigue siendo intratable intentar calcular la ecuación 2.12. Lo que sigue es, utilizando la aproximación de celdas independientes [2], se descompone el mapa como el producto de las marginales, esto finalmente reduce el problema a algo tratable:

$$p(m|z_{1:t}, x_{1:t}) = \prod_{i=1} p(m_i|z_{1:t}, x_{1:t}) \tag{2.14}$$

El pseudocódigo que implementa este algoritmo, de naturaleza recursiva, se define en el algoritmo 1:

Algoritmo 1 Celdas de ocupación [2]

l_{t-1} mapa anterior
x_t pose
z_t medición
for todas m_i **do**
  if $m_i$es visible enz_t **then**
    $l_{t,i} = l_{t-1,i} + l(m_i|x_{1:t-1}, z_{1:t-1}) - l_0$
  else
    $l_{t,i} = l_{t-1,i}$
return $l_{t,i}$

El termino $l_{t-1,i}$ es el logaritmo de la probabilidad del mapa en el paso anterior e incorpora el modelo inverso de sensor: la probabilidad de que una celda esté ocupada

dada la medición, como es de esperarse depende del sensor utilizado y existen curvas para cada tipo de sensor. Mientras l_0 es la probabilidad *a priori* de que cualquier celda esté ocupada. La probabilidad logarítmica mencionada se define como:

$$l(x) = log \frac{p(x)}{1 - p(x)} \tag{2.15}$$

Antes de finalizar la sección, es importante mencionar cómo se utilizan los mapas generados una vez creados, por ejemplo en el caso de localización. Los mapas son matrices donde cada elemento representa la probabilidad de estar ocupado. Para tomar decisiones a partir de este mapa se considera un umbral. Toda celda donde la probabilidad sea mayor a este valor se considera ocupada y libre en caso contrario. Esto de forma similar a la toma de decisión en filtros gaussianos.

2.6. Mapeo tridimensional

Los algoritmos y métodos descritos en la sección 2.5 producen una representación bidimensional del medio, es decir, una rebanada del mundo. Esto es suficiente para robots que operan solo sobre un plano pero no es una representación completa. Los algoritmos planos no son capaces de representar adecuadamente ciertos obstáculos como sillas y mesas, sólo son capaces de captar las patas. Además, para ciertas acciones, como planeación de movimientos en manipuladores, se requiere una representación completamente tridimensional. En esta sección se analizará brevemente el hardware y el software que nos permiten generar representaciones tridimensionales del mundo.

2.6.1. Láseres y Cámaras de profundidad

Para el caso de la robótica algunos de los sensores más usados y útiles son los sensores de telemetría, también llamados sensores de distancia sin contacto o sensores de profundidad. En décadas pasadas los sensores basados en sonar era los más utilizados por su costo, pero en los últimos años ha avanzado mucho la tecnología basada en láseres. Así, por ejemplo, ambos robots de servicio de la Facultad de Ingeniería de la UNAM utilizan sensores láser para navegación y localización.
Su uso se popularizó alrededor del DARPA Challenge de 2005, un concurso de vehículos autónomos promovido por la agencia de Investigación de Defensa americana, donde el equipo de Stanford utilizó un arreglo de sensores láser de la marca SICK para ganar. Para la siguiente edición, llevada a cabo en 2007, la mayoría de los

(a) LIDAR 1D: Garmin

(b) LIDAR 2D: Hokuyo

(c) LIDAR 3D: HDL-64E

Figura 2.2: Ejemplos de sensores LIDAR

equipos concursaron utilizando sensores Velodyne, específicamente diseñados para esta tarea [4].
Estos dispositivos se conocen formalmente como sensores de Medición y Detección de Luz o LIDAR, por sus siglas en inglés [4]. Su operación es muy similar, al menos conceptualmente, al sonar y al radar. El dispositivo emite una señal de luz, generalmente un láser infrarrojo seguro para los ojos, y mide el tiempo que tarda en rebotar a algún obstáculo, para calcular la distancia. Una forma simple de clasificar este tipo de sensores es mediante la dimensionalidad de la información que retornan. Existen LIDAR de una sola dimensión como el LIDAR Lite v3 de Garmin, que miden la distancia lineal a algún objeto. Los de dos dimensiones generan un conjunto de puntos dentro de un plano, como los láser Hokuyo o SICK y por último los sensores tridimensionales generan una representación espacial dispersa del ambiente. Los más populares son los sensores Velodyne [5] los cuales son muy utilizados en el desarrollo de vehículos autónomos desde Junior [6] en el 2007 hasta los desarrollos actuales de compañías como Uber y Alphabet/Waymo. Podemos ver estos sensores en la figura 2.2.
Cabe mencionar que la mayoría de LIDAR 2D o 3D comerciales están basados en arreglos de sensores lineales móviles. Es decir, se monta uno o más LIDAR de una sola dimensión y se ponen en movimiento. Existen desarrollos de LIDAR de estado sólido [4] que prometen ser mucho más ligeros, compactos y baratos pero por el momento, y a conocimiento de el autor, no han llegado al mercado.
A pesar de proveer una resolución y precisión muy elevadas, en el ámbito de la robótica de servicio el uso de LIDAR tridimensional no se ha generalizado, por cuestiones de costos: Un modelo pequeño de Velodyne puede llegar a costar más de 20,000 dolares, además estos equipos tienden a ser muy pesados y requieren mucha energía

(a) Xtion (b) Kinect 2 (c) Zed

Figura 2.3: Ejemplos de cámaras de profundidad

para operar lo que los vuelve poco atractivos para un sistema móvil y de relativamente baja potencia como un robot de servicio. Sin embargo el uso de sensores de profundidad no ha pasado por alto en el medio. Como se mencionó, muchos robots utilizan LIDAR 2D para navegación y localización pues estos dispositivos son mucho más baratos, compactos y de relativamente bajo consumo comparados con sus homólogos tridimensionales.
Por otro lado, la aparición del Kinect de Microsoft en 2010 fue la primera vez que muchos robots pudieron utilizar un sensor de profundidad. Este equipo tiene un valor de unos cuantos cientos de dolares y nos permite obtener representaciones espaciales del ambiente a color. A partir de entonces surgieron varias alternativas que también eran capaces de cumplir esta tarea. Para clasificar este hardware es más fácil hacerlo en función a su forma de operación.
En este caso tenemos 3 vertientes principales: Las cámaras de luz estructurada, como el Kinect original, las cuales proyectan un patrón infrarrojo conocido sobre una superficie y a partir de la deformación que sufre este patrón, el cual es captado por una cámara infrarroja, el dispositivo triangula la profundidad. Por otro lado tenemos las cámaras de tiempo de vuelo, la segunda versión del Kinect es el ejemplo más conocido, éstas operan de forma similar al LIDAR, emitiendo una señal infrarroja y calculando la profundidad a partir del desfase entre la señal emitida y la recibida [7]. Finalmente tenemos las cámaras estéreo, las cuales son arreglos de 2 cámaras montadas sobre un soporte común. A partir de la disparidad entre ambas cámaras y conociendo las características de las mismas es posible calcular la profundidad de la escena. La cámara ZED de esterolabs es un buen ejemplo de esta tecnología. Ejemplos de estás se observan en la figura 2.3.

A pesar de ser muy útiles para la robótica, estos sensores fueron creados originalmente como periféricos para consolas de videojuegos, y en este aspecto fueron un fracaso comercial. Microsoft descontinuó ambos modelos de Kinect en 2017 y Asus dejo de producir su línea de cámaras de profundidad Xtion después de una

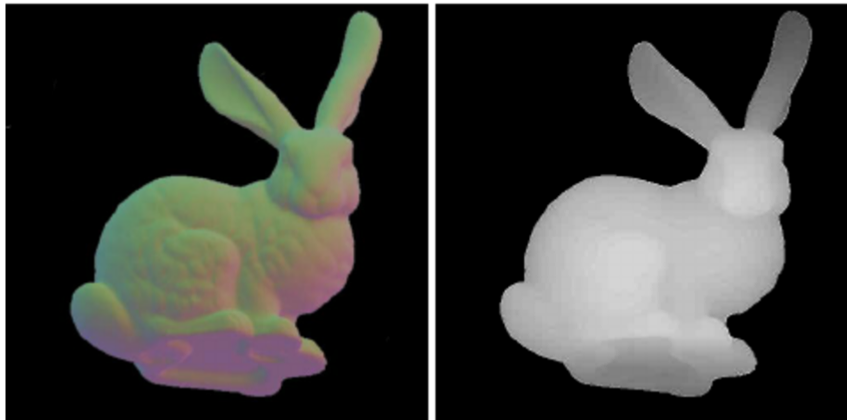


Figura 2.4: A la izquierda: Conejo de Stanford, a la derecha su mapa de profundidad

breve temporada. A saber del autor la única marca que todavía produce cámaras de profundidad a nivel comercial es Intel con su línea RealSense.

2.6.2. Representación tridimensional

En la sección 2.6.1 se discutió sobre el tipo de dispositivos que nos permiten obtener información tridimensional del medio pero se omitió cómo se representa esta dentro de una computadora. Existen varias formas de tomar las medidas de los sensores y almacenarlas de una forma útil. En concreto se discutirán las técnicas más comunes y se examinarán algunas de sus ventajas y desventajas.

Mapas de profundidad

Primero se deben mencionar las imágenes o mapas de profundidad: Como sus contrapartes convencionales son un arreglo bidimensional discreto de valores. En este caso cada pixel contiene la información de la distancia desde ese punto sobre el plano de la imagen a algún objeto en la escena. Es la forma más común en que los sensores entregan información y por lo general deben ser procesadas para obtener alguna representación más útil. Un ejemplo de este tipo de imágenes se observa en la figura 2.4.

Nubes de puntos

Esta es una representación dispersa del ambiente. Consiste en una lista, ordenada o no, de puntos que representan el espacio: Un vector (x, y, z). Generalmente se

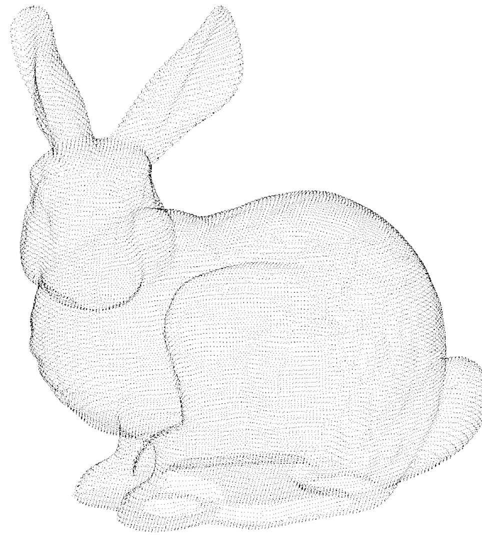


Figura 2.5: Nube de puntos del conejo de Stanford

construyen a partir de un mapa de profundidad y los parámetros intrínsecos de la cámara que tomó la escena.
Esta es una representación muy usada y su popularidad está ligada al auge de las cámaras de profundidad y sensores LIDAR. PCL [8] es una biblioteca en C++ utilizada para procesar este tipo de estructura de datos y está completamente integrada a *ROS: Robot Operating System* siendo el estándar para procesamiento de puntos tridimensionales.
Esta representación tiene la ventaja de no cuantizar las mediciones y solo estamos limitados a la precisión de la máquina por lo general flotantes de 32 bits. Asimismo, el volumen que podemos representar solo esta acotado por la memoria disponible.
A pesar de su popularidad, esta representación tiene desventajas. Es inherentemente dispersa, y solo provee información sobre puntos específicos en el espacio. Y, por ende, es incapaz de diferenciar entre espacio libre y ocupado. Asimismo tiene requerimientos de memoria muy altos: Una nube de puntos generada por un sensor Kinect consiste en 640x480 puntos de cuatro dimensiones: Posición en x, y e z y un color RGB. Considerando que los cuatro valores son de punto flotante de cuatro bytes cada uno tenemos: 4,915,200 bytes por una sola toma. Relacionado a lo anterior tenemos que es un contenedor de datos no limitado por lo que la memoria que ocupa puede crecer arbitrariamente [9]. Una visualización de una nube de puntos se muestra en la figura 2.5

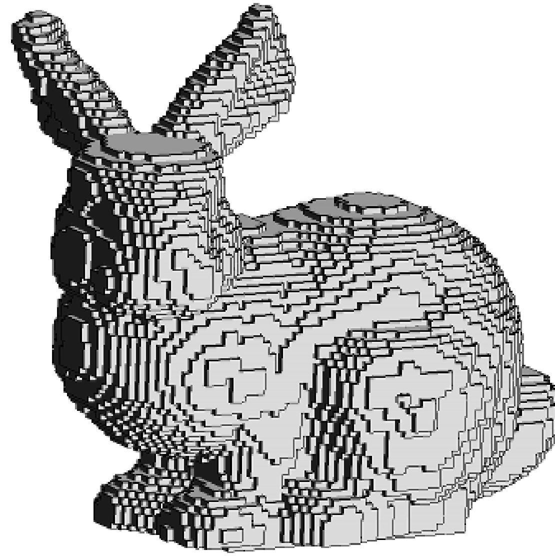


Figura 2.6: Representación voxelizada del conejo de Stanford

Voxeles

Esta representación para espacios en tres dimensiones, es muy similar a la idea de imágenes digitales. En esta, todo el espacio esta almacenado en un arreglo tridimensional de cubos homogéneos llamados voxeles, contracción de *volumetric pixels*, donde cada elemento almacena una característica del espacio, en el caso más simple la existencia o no de algo dentro de cada voxel y en casos complejos se puede almacenar propiedades como color o probabilidad de ocupación.
Este tipo de representación es completamente volumétrica, fácilmente adaptable a los marcos probabilísticos usados en la robótica moderna y es de acceso constante. Sin embargo, utiliza mucha memoria, el tamaño del mapa debe ser conocido de antemano e induce errores de discretización [9]. Un ejemplo se observa en la figura 2.6.

Existen variantes sobre este tipo de algoritmos, como los mapas de elevación, los mapas de elevación extendidos y los mapas de multi-nivel. Sin embargo estas representaciones tienen defectos: Las primeras dos solo pueden representar un nivel y la última es bastante complicada y usarla para localización y navegación es difícil [9].

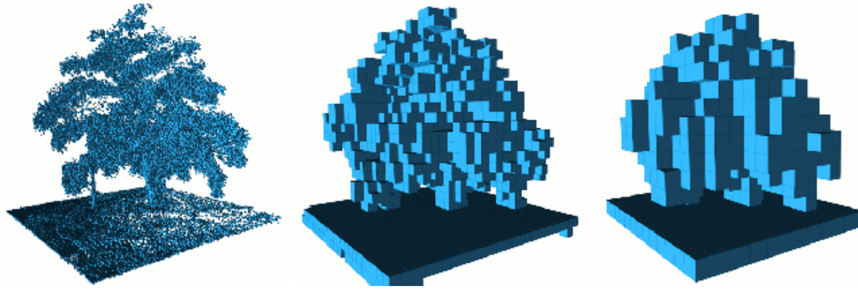


Figura 2.7: Una representación de un árbol a partir de OctoMap [10], debido a que utiliza un grafo internamente se puede consultar el mismo a diferente resolución.

Mapas Octales

Una representación que soluciona casi todos los inconvenientes de las anteriores son los Mapas Octales u OctoMapas [10] presentados por Hornung et. al. en el 2013. Su metodología es una modificación de los mapas voxelizados tradicionales donde utilizan una estructura de arboles octales, *octrees* para almacenar el mapa. En esta, cada nodo representa la probabilidad de tener una celda ocupada. Esto les permite almacenar de manera eficiente un modelo logrando compresiones altas con solo un aumento pequeño en el costo de accesar cada elemento debido a la estructura del árbol. Más aun son capaces de modelar explícitamente tanto el espacio ocupado como el vacío. Lo que permite su uso en sistemas de localización y de planeación de movimientos como la biblioteca de MoveIt para calculo de trayectorias para manipuladores robóticos [11]. En la figura 2.7 se aprecia el resultado de este programa. En este documento se utilizará esta biblioteca para adquirir nubes de puntos así como punto de comparación para el sistema desarrollado.

2.7. Localización y mapeo simultaneo

Como se discutió en la sección 2.5, resolver el problema de mapeo mediante poses conocidas es relativamente sencillo. Sin embargo, tener una buena localización sin un mapa es muy difícil debido a la incertidumbre inherente a los controles y al sensado, por ejemplo la deriva de los instrumentos. El problema general de construir el mapa y localizarse sobre el mismo se conoce como Localización y Mapeo Simultáneos SLAM, por sus siglas en inglés.
Existen dos variantes de algoritmos de SLAM, uno de ellos son los algoritmos en

línea [2] que involucran calcular utilizando sólo la pose actual:

$$p(x_t, m|z_{1:t}, u_{1:t}) \tag{2.16}$$

Donde x_t es la pose en el tiempo t, m es el mapa,y $z_{1:t}, u_{1:t}$ son las mediciones y los controles. Por otro lado tenemos el problema del SLAM completo donde se busca la probabilidad posterior sobre toda la trayectoria del robot desde $1 : t$

$$p(x_{1:t}, m|z_{1:t}, u_{1:t}) \tag{2.17}$$

Y la relación entre ambos problemas se puede expresar mediante la integración de poses pasadas.

$$p(x_t, m|z_{1:t}, u_{1:t}) = \int\int ... \int p(x_{1:t}, m|z_{1:t}, u_{1:t})dx_1 dx_2..dx_{t-1} \tag{2.18}$$

Otra clasificación útil de los métodos de SLAM es en densos y dispersos. En el primer caso se genera una representación donde a cada punto del espacio se le asigna una probabilidad de ocupación, lo que nos lleva a las celdas de ocupación, que se trataron en la sección 2.5.1, directamente de las mediciones de sensores como láseres o cámaras de profundidad. Este tipo de SLAM suelen solucionarse mediante técnicas de optimización de funciones.
En un SLAM disperso se extraen características o *landmarks* discretas de las mediciones del agente y el objetivo es rastrear tanto la posición del agente respecto a estas características como la posición de la mismas en el mapa. Técnicas de filtrado adaptable, como filtros de Kalman y filtros de información, son muy usadas para solucionar este tipo de problema.
En el caso de los SLAM dispersos tenemos a fastSLAM [12] por Montemerlo en el 2002 el cual utiliza información de láser para obtener características del ambiente y funciona a partir de filtros de partículas; es una versión más eficiente del previo trabajo de EKF-SLAM que se basa en usar filtros del Kalman extendidos para rastrear la características tomadas de las mediciones [2]. Por otro lado, en el caso de algoritmos densos, entre los más utilizados en el ámbito de la robótica tenemos a Gmapping [13] que funciona a partir de filtros de partículas y Hector SLAM que funciona a partir de fusión de sensores y optimización de errores entre la medición actual y el mapa [14]. Estos generan una representación de celdas de ocupación por lo que resultan bastante útiles para ambientes pequeños y bien limitados, a su vez tenemos al sistema de Cartographer [15], el cual opera en función de optimizar un mapa local y uno global. Este genera celdas de ocupación de gran calidad, las probabilidades tienden directamente a 1 o 0 y es muy robusto a errores, pero es un

paquete bastante pesado y muy complicado de utilizar lo que limita su uso en la robótica de servicio. Es principalmente utilizado por su desarrollador Google para sus servicios de localización.
A conocimiento del autor, el uso de SLAM disperso ha caído en desuso debido al desempeño de los nuevos métodos de SLAM denso, además, estos permiten representar áreas arbitrarias a resolución finita. Sin embargo, los métodos de SLAM dispersos puede ser útiles al momento de generar representaciones de ambientes muy grandes como en el caso de mapas para vehículos autónomos. El sistema que se propone en este documento tiene la ventaja de obtener una representación tridimensional del medio, lo que le da más generalidad que los aquí expuestos.

2.7.1. SLAM 3D

En la sección 2.6 se discutieron diferentes formas de representar el mundo tridimensional. Sin embargo, hasta ahora se ha tratado solo el problema de mapeo con poses conocidas. OctoMap es inherentemente un algoritmo de este tipo y depende de tener disponible la localización del robot generada por otro medio por ejemplo, a partir de un mapa de celdas de ocupación y localización a partir de filtros de partículas [16].
El problema general de SLAM, como es de esperarse, es más difícil de resolver en tres dimensiones. Los primeros intentos de solucionar este problema fueron basados en imágenes convencionales y el uso de algoritmos de *Structure from Motion* estimando trayectorias en el espacio. Uno de los algoritmos más exitosos en este grupo fue Large Scale Direct SLAM o LSD-SLAM [17] publicado en el 2014 por el grupo de computo visual del Tecnológico de Munich. Su principal contribución es la creación de mapas de profundidad semi-densos a partir de la optimización de poses de la cámara. Este algoritmo aunque ofrecía resultados de alta calidad requiere del uso de hardware muy caro y especializado, como cámaras de instrumentación. Asimismo requiere de mucha memoria y poder de computo. Además, a conocimiento del autor, nunca se ha utilizado en algún sistema robótico.
Actualmente se prefiere combinar información visual, proveniente de cámaras, con información espacial, que viene de láser o LIDAR para generar reconstrucciones tridimensionales del ambienta al mismo tiempo que se construyen celdas de ocupación clásicas.

SLAM por grafos

Debido a que muchos de los algoritmos de SLAM modernos utilizan un enfoque gráfico se discutirá brevemente en que consiste esta metodología.

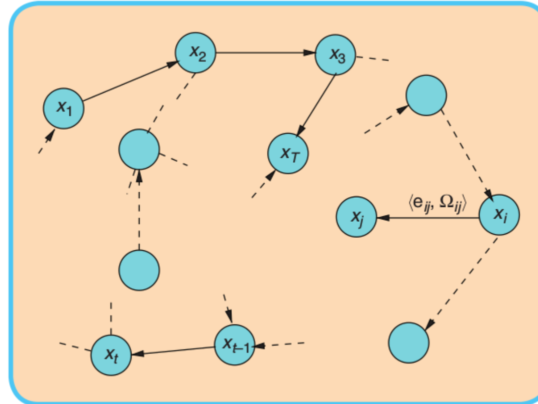


Figura 2.8: La representación gráfica del problema de SLAM

En sí es en una forma de solucionar el problema del SLAM completo, es decir, donde se estima el mapa a partir de todas la trayectoria, no solo la pose inmediata anterior. Este método fue propuesto en los noventas, pero requiere optimizar sobre una Variedad, o *Manifold* en inglés, lo cual es muy complicado, lo que limitó su aplicación práctica. Pero recientes avances sobre la estructura del problema de SLAM y el álgebra dispersa han provocado un resurgimiento de esta técnica lo que ha llevado a que sea parte de muchas soluciones en el *estado del arte* de SLAM [18].
En esta metodología basada en grafos o redes las poses de un robot se modelan mediante nodos en un grafo, en estos nodos se almacenan tanto las poses como las mediciones tomadas en esta posición. Las aristas del grafo representan restricciones espaciales entre poses mediante una distribución de probabilidad entre las transformaciones relativas entre nodos. Estas transformaciones pueden ser controles de la odometría o son determinadas alineando de forma máxima mediciones entre nodos, estas son llamada medidas virtuales.
Una vez que se ha establecido el grafo se busca la configuración de poses que son las más adecuadas para ajustar las mediciones. Esto separa el problema del SLAM en dos tareas: La creación del grafo, también llamado *front-end* la cual es dependiente en el tipo de sensor y la optimización del mismo llamado comúnmente *back-end* que es agnóstica a las mediciones.
El front-end tiene la tarea de calcular cuando se ha visitado un lugar que ya se visitó anteriormente, a esto se le conoce como cerradura de lazo, esto se añade como un restricción al grafo lo que mejora la estimación de la trayectoria del agente. Mientras el back-end se encarga de la optimización del grafo a partir de sus restricciones.

En términos formales se tiene un vector $x = (x_1, ...x_T)^T$ donde x_i representa la

pose en el nodo i. Asimismo contamos con z_{ij} y Ω_{ij} que corresponden a la media y matriz de información de una medida virtual. Así tenemos:

$$\hat{z}_{ij} = f(x_i, x_j) \quad (2.19)$$
$$e_{x_i,x_j} = z_{ij} - \hat{z}_{ij}(x_i, x_j) \quad (2.20)$$

Es decir $\hat{z}_{ij}$ es la predicción de una medida virtual dado dos nodos que han visto la misma escena y e_{x_i,x_j} es el error de predicción. Con esto podemos definir la verosimilitud de la medición y, por último, si minimizamos la verosimilitud negativa logarítmica podemos encontrar las poses que mejor expliquen las mediciones. [18]:

$$l_{ij} \propto e_{ij}^T \Omega_{ij} e_{ij} \quad (2.21)$$
$$F(x) = \sum_{(i,j)\in C} e_{ij}^T \Omega_{ij} e_{ij} \quad (2.22)$$
$$x^* = \operatorname*{argmin}_x F(x) \quad (2.23)$$

Donde C es el conjunto de todos los nodos que comparten alguna medición y x^* las poses encontradas.
La optimización de dicha función se realiza por medio de mínimos cuadrados, pero va más allá del alcance de este texto. Un back-end completo de SLAM por grafos se puede consultar en [18] y una representación visual del grafo de poses se muestra en la figura 2.8.

Técnicas de SLAM 3D

Ahora que se ha tratado la teoría elemental de SLAM por grafos podemos hablar del algunos de los algoritmos que lo utilizan de una forma u otra. Una de las primeras metodologías que utilizan sensores de profundidad para la solución del problema de SLAM es RGBD-SLAM por Endres et.al [19] alrededor del 2014, contemporáneo a los algoritmos basados en imágenes convencionales. Su aproximación se basa en el calculo de un grafo de poses y su optimización en función de encontrar cerraduras de lazo a partir de características 2D de la imagen tipo SIFT y SURF. Tiene soporte hasta la versión "Kinetic Kame"de ROS pero ya no se distribuye en paquetes precompilados.
Otra aproximación de corte similar y de buenos resultados es ORB-SLAM [20] método publicado por la universidad de Zaragoza por primera vez en el 2015 que, como su

Figura 2.9: Reconstrucción semi-densa generada a partir de ORB-SLAM2

nombre indica a partir de las características ORB de una imagen estima la pose de la cámara a lo largo de una trayectoria y a su vez genera una reconstrucción dispersa del ambiente. La segunda versión de ORB SLAM [21] incorpora el uso de imágenes estéreo y de profundidad lo que le permite hacer reconstrucciones semi-densas con la escala correcta y todavía es utilizado en el medio, en especial en plataformas de poco poder de computo como vehículos aéreos no tripulados y robots móviles. Una reconstrucción a partir de este algoritmo se aprecia en la figura 2.9.

Finalmente tenemos a Real Time Appeareance Based Mapping o *RTAB-MAP* [22], [23] por sus siglas en inglés. Este algoritmo fue publicado por Mathieu Labbe y Francois Michaud de la Universidad de Sherbrook por primera vez en el 2014.

Este sigue la idea de RGBD-SLAM donde a partir de descriptores de imágenes 2D busca cerraduras de lazo y optimiza un grafo a partir de estas restricciones. Sin embargo, RTAB-Map propone un método de manejo de memoria eficiente. En un algoritmo de SLAM para detectar la cerradura de lazo se requiere comparar con todas las escenas encontradas anteriormente, sin embargo, tener todas las escenas en memoria no es factible ya que este conjunto crece con cada nueva zona visitada lo que lleva a tener tiempos de ejecución siempre crecientes. Su propuesta mantiene un conjunto limitado de imágenes en una memoria de trabajo, *Working Memory* para la comparación. Asimismo tienen una memoria a largo plazo en el disco *Long Term Memory* donde se almacenan las demás escenas. Si la memoria crece más de lo

Figura 2.10: Reconstrucción del laboratorio de Biorobótica. A partir de RTAB-MAP

especificado algunas escenas se transfieren a la memoria de largo plazo y al detectar una cerradura de lazo se extraen imágenes a la memoria de trabajo. Esto permite un tiempo de ejecución casi constante. Este algoritmo es bastante popular y tiene soporte oficial hasta la versión actual de ROS "Melodic Morenia".
En este trabajo se utilizó RTAB-MAP para la adquisición de nubes de puntos de áreas grandes. Una reconstrucción del laboratorio de Biorobótica de la UNAM generada a partir de este software se muestra en la figura 2.10. Como se aprecia la mayor parte de los sistemas de SLAM tridimensional requieren procesar mucha información en espacios de alta complejidad matemática, además generan mapas muy pesados el método propuesto en este documento es más sencillo y ligero. Pero requiere de conocer la localización del robot.

Capítulo 3

Descripción del sistema

En el capítulo anterior se trataron algunos elementos teóricos y el estado del arte tanto del mapeo como el del SLAM en dos y tres dimensiones. En este capítulo se describe la estructura del trabajo desarrollado.
Se propone aquí el sistema de Sparse-Maper un sistema de mapeo con poses conocidas que genera representaciones dispersas del ambiente en tres dimensiones a partir de la cuantización del espacio. A lo largo del capítulo se detalla su implementación.

3.1. Visión general del sistema

El sistema se encuentra divido en varios módulos que integran su funcionamiento esto con el fin de lograr un programa débilmente acoplado y cuyas partes sean agnósticas al funcionamiento de las demás permitiendo un mayor facilidad en su desarrollo y mantenimiento. A grandes rasgos el sistema está conformado por los siguientes módulos:

- Adquisición de datos.
- Preprocesamiento.
- Segmentación o cuantización del espacio.
- Mapeo.
- Servidor de mapas.
- Planeador de trayectorias.

Un diagrama de bloques ilustrativo de la configuración general del sistema se aprecia en la figura 3.1.

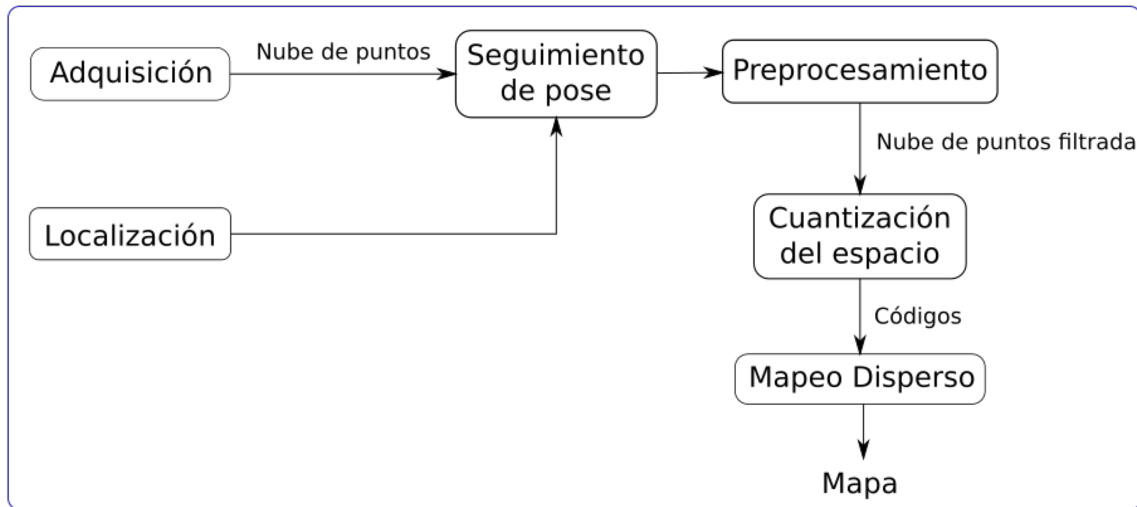


Figura 3.1: Diagrama general del software de mapeo

3.1.1. Adquisición de datos

El sistema completo opera a partir de la nubes de puntos, estructuradas o no. En términos prácticos se pueden utilizar varios tipos de entradas, ya sean los varios tipos de cámaras de profundidad de las que ya se discutió en el capitulo anterior o mediante archivos de objetos tridimensionales. Asimismo el sistema puede recibir la salida de otros algoritmos que generen nubes de puntos. Así, por ejemplo, se pueden utilizar tomas individuales de una cámara de profundidad para generar incrementalmente un mapa o post-procesar la salida de alguna otra biblioteca como RTAB-Map [22] para generar el mapa completo en una sola ejecución.

3.1.2. Localización

Como se menciona al principio el sistema requiere de conocer de antemano la pose del robot para funcionar. Para esto se pueden utilizar cualquiera de los algoritmos de SLAM ya tratados o métodos de localización basados en láser y mapas de celdas de ocupación como la localización adaptativa de Montecarlo [16]. Debido a lo anterior a este enfoque también se le denomina como aumentado de mapas ya que se aumenta un mapa plano con información adicional. Cabe resaltar que en caso de contar con otra fuente de localización, como GPS por ejemplo, esta también podría ser utilizada. En el presente documento esta opción no se exploró.

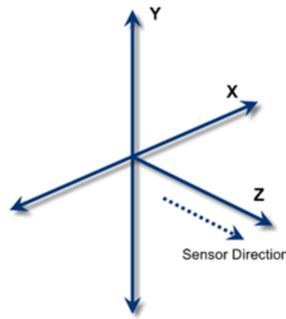


Figura 3.2: Sistema de referencia común para una cámara de profundidad

3.1.3. Preprocesamiento

El primer módulo propiamente parte del sistema, los dos anteriores se implementan de forma externa. Este se encarga de tres tareas de procesamiento de nubes de puntos: Eliminar puntos muy alejados al sensor, submuestrar la nube de puntos y transformar el marco de referencia de la nube de puntos
La mayoría de las cámaras de profundidad tienen rangos de operación de más de siete metros, pero mediciones tomadas a largas distancias tienen un error y ruido significativo en especial comparado con medidas tomadas a una distancia menor, para evitar esto se eliminan todos los puntos que se encuentran a más de una cierta distancia. Dado que esta distancia varia de equipo a equipo se expone como parámetro para el usuario.
Dependiendo del dispositivo utilizado la resolución espacial del mismo varia bastante y a veces reporta mucha más información de la que se requiere para representar adecuadamente la escena. Usar más puntos implica tener un detalle más fino, sin embargo, también implica un mayor tiempo de procesamiento, por lo que se puede usar una técnica sencilla de voxelización para submuestrear la nube de puntos y reducir el número de datos a procesar.
Finalmente el módulo transforma la nube de puntos del marco de referencia del sensor a otro más útil. En la mayoría de los casos el marco de referencia del sensor no es el más conveniente para trabajar el resto del algoritmo. Casi todos los sensores de profundidad cuentan con un marco de referencia similar al mostrado en la figura 3.2. En este el eje z es paralelo al eje focal y los otros dos son coplanares al sensor óptico. Debido a que en muchos robots el sensor de profundidad puede estar actuado y moverse en tres ejes es más fácil trabajar con un marco de referencia fijo dentro del robot. En la mayoría de los casos el marco de referencia de la base es escogido para esto.

3.1.4. Cuantización del espacio

En la literatura [10] se habla de espacio libre y espacio ocupado en el contexto de mapeo. Pero para el propósito de este documento es más útil hablar del espacio navegable aquel que a pesar de poder estar ocupado el robot es capaz de navegar sobre él. En términos prácticos y para propósitos de este trabajo este se refiere al piso.
La mayoría de las metodologías de mapeo y SLAM ya descritas generan representaciones densas o semi-densas del medio donde a cada punto del espacio se le asigna una probabilidad de estar ocupado. Pero son representaciones que utilizan mucha memoria y son computacionalmente caras de tratar. Se considera que para la tarea de navegar sin colisiones no se requiere este nivel de resolución y una representación más dispersa puede ser igual de efectiva pero requerir menos recursos de computo para su ejecución.
Es decir podemos comprimir la información espacial obtenida de los sensores, y a partir de esta representación más compacta determinar el espacio navegable, el ocupado y generar trayectorias válidas para el agente. Para lograr esto se utilizó un método similar al expuesto en [24] donde se usaron supervoxeles para la segmentación semántica de imágenes RGBD. En la metodología aquí propuesta, los supervoxeles o grupos de puntos, son generados a partir de la nube de puntos preprocesada. Esto se realiza mediante técnicas de cuantización vectorial. Este proceso es similar a la cuantización escalar usada en sistemas digitales, donde una señal continua es restringida a tomar valores finitos o cuantizados. Como su nombre indica en una cuantización vectorial la señal de entrada y de salida son de varias dimensiones. Formalmente el proceso de cuantización vectorial toma una señal $x_i \in \mathbb{R}^n$ y encuentra un conjunto finito de tamaño k de vectores $C = \{y_1, y_2, y_3, ..., y_k\}$ llamado alfabeto o *codebook* que represente adecuadamente la señal original dado que $y_k \in \mathbb{R}^n$. Una vez encontrado el conjunto C se definen las operaciones de codificación y decodificación como se muestra en la ecuación 3.1.

$$y_n \in C = \{y_1, y_2, y_3, ..., y_k\} \tag{3.1}$$

$$Q(x_i) = \underset{n}{\operatorname{argmin}}(\|x_i - y_n\|) \tag{3.2}$$

$$D_n = y_n \tag{3.3}$$

Como se aprecia el codificador es un problema del vecino más cercano de un punto de la señal respecto a los códigos y el decodificador es una tabla de búsqueda. Esto dado que ya se conoce el alfabeto C, pero es claro que obtener éste, es el paso más difícil. Esto se aborda a partir de los algoritmos de agrupamiento o *clustering* a veces llamados de aprendizaje no supervisado. De estos existe una gran variedad y

discutirlos todos esta fuera del alcance de este documento. Sin embargo sí se tratan aquellos usados en este sistema: K-medias y LBG ambos algoritmos basados en el algoritmo de Lloyd.
Este tipo de algoritmos tienen la finalidad de generar un alfabeto tal que minimicen la siguiente distorsión.

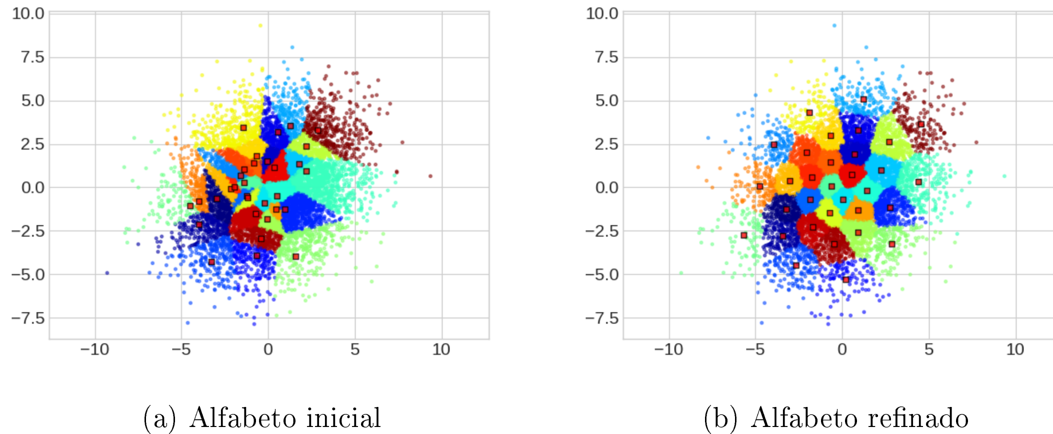
$$\phi = \sum_{x \in X} \min_{y_n \in C_k} ||x_n - y_k|| \tag{3.4}$$

En otras palabras la suma de distancias mínimas de todos los puntos en X al alfabeto C. La métrica que se menciona es una distancia en el sentido estricto pero la definición formal depende del problema. En nuestro caso se utiliza una norma euclidiana pues los datos representan puntos en el espacio tridimensional. Pero en otros dominios, como el reconocimiento de voz, se utilizan distancias espectrales.
Uno de los algoritmos más simples y sencillos que nos permite agrupar datos es conocido como K-medias y nos permite encontrar, como su nombre indica, K grupos en nuestros datos.

Algoritmo 2 K-medias

Input: X - Vector de datos $(x_1, x_2, ..., x_d)$
Input: N - Número de datos
Input: k - Tamaño del alfabeto
Input: i - Numero de iteraciones
  $C_j = c_1, c_2, c_3, ..., c_k$- centroides aleatoriamente escogidos
  for $u = 0$ to i **do**
    S - Partición de los datos
    for all x_n in X **do**
      $S_n = \text{argmin}_m \|x_n - c_m\|$
    $C_j = \frac{1}{|S_j|} \sum_{x \in S_j} x$
  return C_j

En el algoritmo 2 se describe el pseudocódigo de éste. Se inicia con k vectores generados aleatoriamente y luego se repiten dos pasos alternadamente i veces. En el primero se crea una partición de los datos asignando a cada punto un código en función de cual elemento del alfabeto queda más cerca, esto se puede interpretar como un arreglo de etiquetas. Aquí $\|x_n - c_m\|$ es la distancia euclidiana entre un punto y algún vector del alfabeto. Después se actualizan los centroides mediante una media convencional entre todos los puntos que corresponden a un mismo grupo de la partición. En la figura 3.3 se ilustra el proceso de refinar los códigos generados



(a) Alfabeto inicial

(b) Alfabeto refinado

Figura 3.3: Esta figura se ilustra el algoritmo de K-medias, a la izquierda un conjunto de códigos, como cuadros rojos, escogidos aleatoriamente y la partición que definen se visualiza mediante color, a su derecha los códigos optmizados luego de 10 corridas del algoritmo de Lloyd

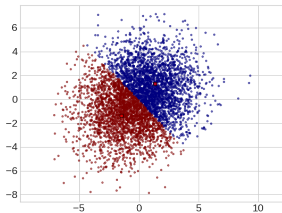
Este es un algoritmo muy rápido pero tiene problemas para segmentar regiones con formas no regulares y sufre de problemas al generar los código iniciales: Al usar elementos aleatorios es susceptible de tener un mala inicialización. A pesar de esto es un algoritmo muy popular y por ende existen muchas modificaciones y variantes del mismo. Una de estas es el algoritmo Linde-Buzo-Gray o LBG por sus siglas [25]. En éste en lugar de inicializar el alfabeto de forma aleatoria este se construye incrementalmente:

Como se describe en el algoritmo 3 al principio se genera un solo centroide del mismo valor a la media de los datos y a partir de un vector e se perturba este para generar 2 centroides nuevos, despues utilizando estos se repite el algoritmo de Lloyd para ajustar los centroides hasta i veces. Se duplican entonces el número de centroides y se repite este proceso hasta conseguir el número deseado de códigos. Es importante destacar que este algoritmo solo puede generar un alfabeto donde el número de elementos es una potencia de dos. La ventaja es que la generación de centroides es determinística y evitamos problemas con la inicialización, en la figura 3.4 se ilustra la forma en que se generan los centroides incrementalmente.

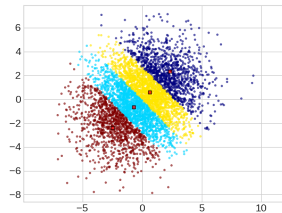
Otra modificación al algoritmo es conocida como K-medias ++ propuesto en el 2007 por Vassilvitskii [26]. A diferencia de otros algoritmos de agrupamiento este no implica grandes modificaciones al algoritmo base, simplemente usa una estrategia inteligente de inicialización. En su publicación se mencionan varias mejoras en tiem-

Algoritmo 3 LBG

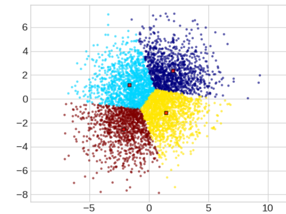
Input: N - Número de datos
Input: X - Vector de datos $(x_1, x_2, ..., x_d)$
Input: k - Tamaño del alfabeto
Input: i - Numero de iteraciones
Input: e - Vector de distorsión $(e_1, e_2, ..., e_d)$
  C_i - Centroides $(c_1, c_2, ..., c_d)$
  $C_0 = \frac{1}{N} \sum_{i=0}^{N} x_n$ - Primer centroide
  $j = 1$
  while $j < k$ **do**
    for all C_j in C_i **do**
      $C_j = C_j + e$
      $C_{j+1}^t = C_j - e$
    $u = 0$ - Contador iteraciones
    while $u < i$ **do**
      S - Partición de los datos
      for all x_n in X **do**
        $S_n = \text{argmin}_m \|x_n - c_m\|$
      $C_j = \frac{1}{|S_j|} \sum_{x \in S_j} x$
      $u++$
    $j = j \times 2$
  return C_j



(a) Alfabeto inicial de 2 centroides



(b) Se generan códigos extras a partir del anterior



(c) El alfabeto extendido se optimiza mediante el algoritmo Lloyd

Figura 3.4: Esta figura ilustra el algoritmo de Linde-Buzo-Gray. Este genera incrementalmente el alfabeto, optimizando un alfabeto parcial para después extenderlo mediante un vector de pertubación.

po y desempeño sobre el algoritmo original. El proceso se ilustra en el algoritmo 4.

Algoritmo 4 k-medias ++

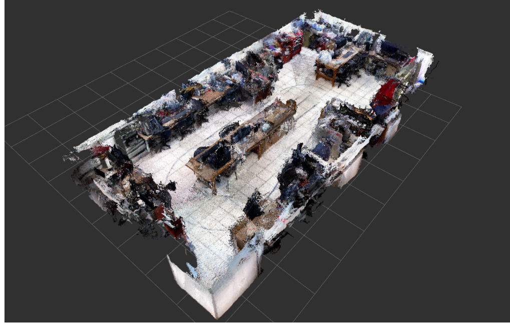
Input: X - Vector de datos $(x_1, x_2, ..., x_d)$
Input: N - Numero de datos
Input: k - Tamaño del alfabeto
  $i \sim U(0, N)$
  $C_0 = x_i$ - Primer centroide
  for $u = 1$ to k **do**
    $D^2(x_i) = \text{mín}_n \, ||x_i - C_n||$
    $j \sim \frac{D^2(x_i)}{\sum D^2(x_i)}$
    $C_u = x_j$
  Continuar con K-medias utilizando C_i como centroides iniciales

---

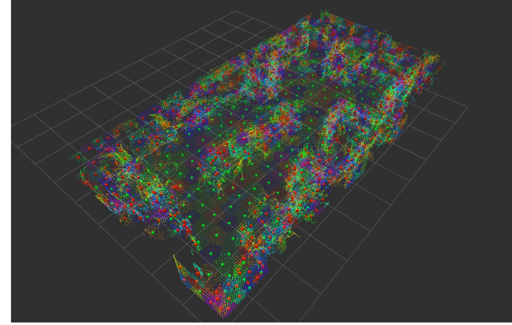
Al inicio se elige el primer centroide de forma aleatoria muestreando uniformemente sobre todos los datos. Despues se construye un arreglo de pesos, que corresponden a la distancia minima de cada punto a los centroides ya escogidos. Entonces se muestrea respecto a la distribución de los pesos y se repite hasta generar k centroides. Finalmente se prosigue con K-medias normal.
La principal diferencia consiste en que los centroides iniciales se eligen tomando un muestreo pesado, es decir es más probable tomar un punto del espacio que se encuentre lejos de todos los centroides ya elegidos. Esto disminuye drásticamente la probabilidad de tener una inicialización mala y genera centroides más distribuidos sobre los datos lo que lleva a un mejor resultado en el agrupamiento y requiere de menos iteraciones para converger.
En la figura 3.5 se observa el resultado de cuantizar una nube de puntos del laboratorio de la Facultad de Ingeniería de la UNAM generada mediante RTAB-Map.

3.1.5. Mapeo

El paquete de cuantización funciona por cada nube de puntos que recibe el sistema. En algunos casos es posible obtener una nube de puntos de todo un ambiente a partir de otras bibliotecas o paqueterías, pero por lo general esto no es factible. Un mapa o ambiente puede estar compuesto de varias nubes de puntos, por lo que se requiere un nodo capaz de acumular diferentes códigos generados en diferentes escenas en un mapa globalmente consistente.



(a) Nube de puntos del laboratorio de Biorobotica de la UNAM.



(b) Espacio cuantizado mediante K-medias++

Figura 3.5: Resultado del sistema de cuantización del espacio, los cuadros verdes representan el espacio navegable y los rojos el ocupado, todos los puntos del mismo color corresponden al mismo centroide, nó tese que la partición genera un patrón hexagonal.

Para esto el nodo de mapeo acumula la lista de códigos en un arreglo en memoria. Asimismo este es responsable de separar el espacio navegable y el espacio ocupado. Una vez generado el alfabeto se puede utilizar alguna característica del mismo para separar el espacio, en nuestro caso se propone utilizar la altura de cada centroide y a partir de esto umbralizar. En este caso todos los códigos arriba de cierta altura, respecto al piso, se consideran ocupados y en caso contrario navegables.

$$T(c) = \begin{cases} 1 \text{ ocupado} & c_z > Thr \\ 0 \text{ libre} & c_z < Thr \end{cases} \tag{3.5}$$

Para esto se asume que todos los puntos que corresponden al grupo de un centroide comparten alguna característica de este, en este caso su altura.
Esta es una aproximación bastante sencilla y para nuestro caso suficiente, sin embargo es generalizable a otras propiedades: Por ejemplo se puede asignar una probabilidad de ocupación a cada código generado basada en el modelo del sensor o utilizar la normal promedio de todos los puntos que pertenecen a la misma partición.
Por último este nodo nos permite guardar el mapa generado al disco en un formato de texto plano. Los códigos se guardan secuencialmente en orden de su distancia al origen del mapa primero los centroides ocupados y al final los libres.

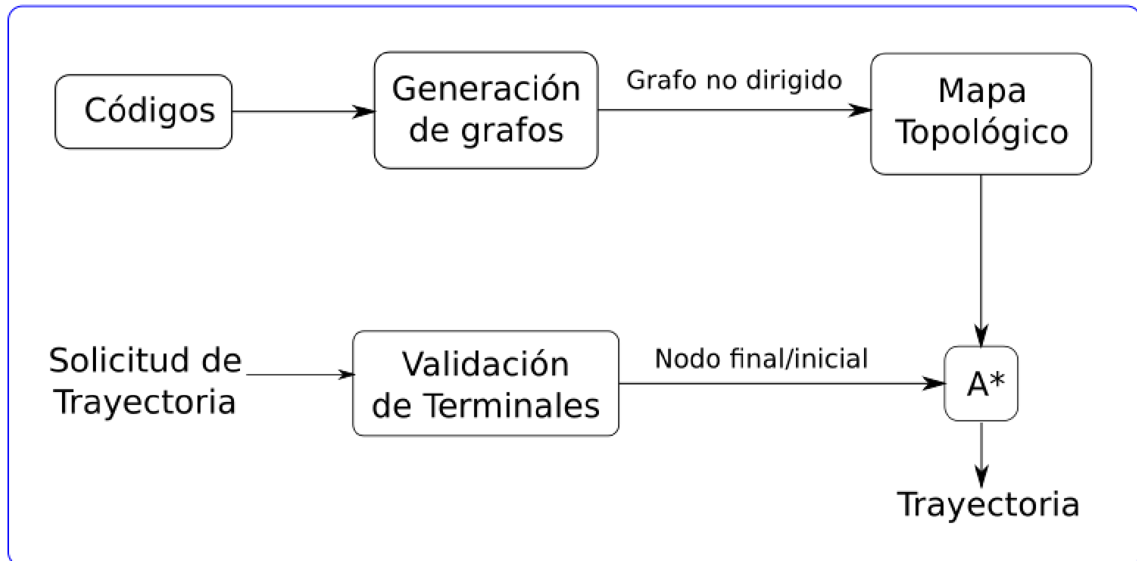


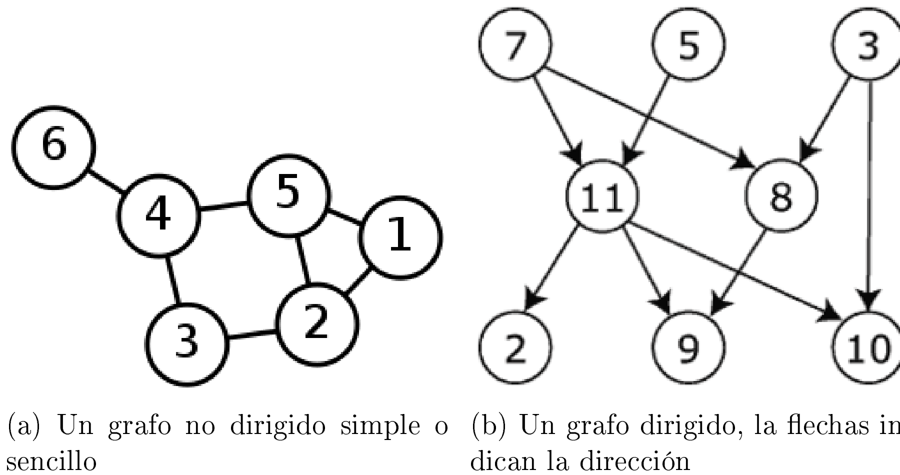
Figura 3.6: Diagrama general del software de mapeo

3.1.6. Servidor de Mapas

Una vez que se ha guardado el mapa en el disco se requiere de un nodo capaz de suministrar los mapas a otros subsistemas y programas que lo utilicen.
Este mapa, a su vez, debe permitir el cálculo de trayectorias válidas para el agente. Por lo que se tiene que generar una relación entre todos los códigos que pertenecen al espacio navegable. Esto se representa mediante un grafo no dirigido que se conoce como mapa topológico.
Este subsistema opera de forma independiente al anterior. Su operación se bosqueja en el diagrama 3.6. Antes de continuar y describir a detalle este módulo se da una breve explicación de que consiste un grafo en la ciencia de computo:

Grafos

Un grafo, en el sentido matemático, es un ente $G = (V, E)$ que consiste en un conjunto de vértices o nodos, V y un conjunto de arcos o aristas, *edges* en inglés E. Un ejemplo elemental es una lista simplemente ligada.
Los vértices o nodos pueden ser cualquier cosa, lugares geográficos, poses, sitios web, etcétera. Mientras tanto las aristas son un conjunto de pares de elementos que



(a) Un grafo no dirigido simple o sencillo

(b) Un grafo dirigido, la flechas indican la dirección

Figura 3.7: Diferentes tipos de grafos.

pertenecen a los vértices. Entonces se dice que dos nodos están conectados, o que son adyacentes, si el par (u, v) pertenece a E [27].

Existen distintos tipos de grafos y la teoría que los rodea es un campo de estudio completo para la ciencia de cómputo y las matemáticas discretas. Un grafo simple es un un conjunto no vacío de vértices que se encuentran conectados por solo una arista. En el caso de que se conecten dos vértices por más de un arco, se le conoce como multigrafo. Por último si se permiten bucles sobre el mismo nodo se conoce como pseudografo. Esta última representación es muy utilizada en cadenas de Markov y en general en modelos gráficos probabilísticos.

También existe otra forma de clasificar los grafos: en función de sus aristas. Si el orden en que se listan no es relevante, se conocen como *no dirigidos*, es decir que la conexión entre ambos nodos es bidireccional. En caso contrario, si las conexiones tienen un orden estricto, o una dirección específica, se conocen como grafos *dirigidos*. Visualmente estos se representan como se muestra en la figura 3.7. Finalmente un tipo de grafo que nos interesa especialmente para este trabajo son los grafos pesados. Es decir aquellos donde existe una función que indica el costo o peso, de ir de un nodo a otro $c(u, v) = f$.

Dentro de un grafo es posible definir un camino, este consiste en una secuencia de vértices conectados por aristas. Un nodo v es accesible desde u si existe un camino entre ellos y un grafo se considera completamente conectado si existe un camino entra cada uno de sus nodos.

En el campo de las matemáticas lo anterior es suficiente para poder trabajar con

grafos. Sin embargo, para su aplicación práctica en sistemas y agentes inteligentes estas estructuras deben ser almacenadas en una computadora digital. Para esto existen dos representaciones populares: Las matrices de adyacencia y las listas de adyacencia.
Las listas consisten en un arreglo de listas ligadas, o en un arreglo de arreglos según la implementación. Cada entrada en el arreglo se refiere a algún vértice y almacena todos los demás nodos que son adyacentes a este.
Encontrar si un nodo es adyacente a otro requiere buscar sobre toda la lista correspondiente lo que puede ser bastante lento, para un acceso rápido se utiliza la representación matricial. En una matriz de adyacencia se organiza una cuadricula de nodos, si los nodos u, v están conectados el valor de la matriz es 1 o 0 en otro caso:

$$M_{u,v} = \begin{cases} 1 & \text{si u esta conectado a v} \\ 0 & \text{caso contrario} \end{cases} \tag{3.6}$$

Como se observa en éste caso pagamos un acceso rápido con un uso de memoria grande. En el caso de que nuestro grafo sea disperso, es decir, cada nodo tiene pocas conexiones, la matriz tendrá predominantemente ceros y se prefiere utilizar una lista de adyacencia para ahorrar memoria.

Generación del mapa

El mapa topológico consiste en un grafo, donde los nodos corresponden a centroides del espacio navegable y las aristas codifican la distancia entre cada nodo. Para fines prácticos en este trabajo se considera nodo y centroide o código como la misma cosa. Este grafo se genera en línea, esto nos permite reajustar el grafo en función a las necesidades del agente. El proceso de generación del grafo se describe en el algoritmo 5, recordemos que cada nodo es un punto en el espacio tridimensional $f = (x, y, z)$.

Para empezar se leen los nodos del disco y se establece una distancia mínima entre cada nodo. En caso de que dos nodos se encuentre a menos de esta distancia se consideran redundantes. Esto ocurre cuando se genera el mapa de forma incremental a partir de varias tomas del ambiente, ya que nada impide tomar la misma escena más de una vez.
Eliminar los nodos redundantes provocaría una pérdida fuerte e innecesaria de información, por lo que se propone que todos los nodos dentro de un radio ϵ de otro, sean sustituidos por su centroide, tal como se explica en el algoritmo 6. Este algoritmo se corre solo sobre los nodos libres ya que eliminar nodos ocupados puede llevar a

Algoritmo 5 Topological Map creation

Input: O - Centroides ocupados
Input: F - Centroides navegables
Input: Height - Altura del agente
Input: ϵ - Distancia mínima entre dos nodos distintos
  F = ReducirNodos(F,ϵ)
  for all f in F **do**
    for all o in O **do**
      $f_2 = f + [0, 0, Height]$
      if CylinderCollision(f, f_2, o) **then**
        Drop f
  G - grafo
  for all f in F **do**
    Cálculo de los k vecinos más cercanos de f
    $D = ||F_j - f||$ - Distancia a todos los nodos libres desde f
    sort(F,D)
    for $i = 1, 2, 3, .., k$ **do**
      if not CylinderCollision(f, F_i^t, o) **then**
        G.Join(f, F_i)
  return G

Algoritmo 6 Reducción de nodos

Input: X - Lista de nodos a operar
Input: ϵ - Distancia mínima entre nodos
  Y - Lista reducida de nodos
  while X no esté vacío **do**
    $P = X.pop$ - centroide
    $i = 0$
    for all nodos n en X **do**
      if **then**$||x - n|| < \epsilon$
        Remover n de X
        $P = P + n$
        $i = i + 1$
    $Y.append(P/i)$
  return Y

Algoritmo 7 Colision Cilindro

Input: X_i - Punto de inicio cilindro
Input: X_j - Punto final cilindro
Input: q - Punto a revisar
Input: r - Radio cilindro
  if $(q - X_i) \cdot (X_i - X_j) < 0$ **then**
    return False
  if $(q - X_j) \cdot (X_j - X_i) > 0$ **then**
    return False
  if $\frac{|(q-X_j)\times(X_j-X_i)|}{|X_j-X_i|} < r$ **then**
    return False
  return True

colisiones.
Una vez que se redujeron los nodos redundantes se eliminan todos los nodos navegables que estén demasiado cerca, o debajo, de algún centroide ocupado, para evitar colisiones. Para esto se revisa si alguno de los códigos ocupados se encuentra dentro de un cilindro con eje paralelo al eje z, coincidente sobre el centroide navegable correspondiente, y de altura igual o mayor al agente y de un radio capaz de circunscribir a todo el robot.
Por último se calculan los K vecinos más cercanos de cada nodo navegable y se conectan en el grafo. Se utiliza una lista de adyacencia como estructura de datos subyacente. En caso de que alguno de estos vecinos se encuentra a una distancia grande, mayor a algún valor especificado, estos no se conectan, mismo caso si se presenta algún obstáculo entre los nodos. Para verificar esto se prueba si existe algún centroide ocupado dentro del cilindro definido por la línea que une dos nodos libres y un radio arbitrario.
Este tipo de algoritmos, conocidos como detectores de colisiones, son computacionalmente pesados, pero gracias a que se ha reducido la dimensionalidad del espacio usando la cuantización vectorial, estos procesos pueden correr en un tiempo razonable, finalmente una vez ajustado el mapa el grafo puede almacenarse en disco.
El pseudocódigo para verificar las colisiones, es decir detectar si algún nodo ocupado se encuentra bloqueando algún nodo navegable o la conexión entre dos nodos, se muestra en el algoritmo 7. Para esto se toman dos puntos en el espacio y define un cilindro con eje que atraviesa los dos puntos y un radio r, así como dos planos o "tapas"sobre cada punto y normal paralelas al eje del cilindro. Se revisa entonces si el punto a probar q se encuentra dentro de del cilindro, en caso de estar dentro del

cilindro la función regresa un verdadero booleano y un falso en caso contrario.

3.1.7. Servidor de Trayectorias

La finalidad de contar con un mapa del entorno es poder generar trayectorias válidas para que el robot o agente navegue. Dado que contamos con la representación gráfica del ambiente el problema se reduce a encontrar un trayectoria en el grafo que nos lleve de algún punto origen al destino.
Para encontrar estas trayectorias se utilizó el algoritmo de A* este se explicará a continuación así como una breve introducción a la teoría de la búsqueda en grafos.

Búsqueda en grafos

Como se mencionó en la sección 3.1.6 los grafos nos permiten representar relaciones entre diferentes nodos. Esto por si mismo tiene aplicaciones. Por ejemplo algunos métodos optimizan sobre un grafo de poses para resolver el problema del SLAM. Además muchas de las técnicas de inteligencia artificial clásica, como sistemas expertos, se basan en representar algún problema mediante un grafo y en buscar la respuesta recorriendo este.
Los algoritmos más sencillos de búsqueda sobre grafos simplemente recorren nodo a nodo de forma sistemática hasta encontrar el destino [28]. Pero sobre grafos muy grandes el espacio de búsqueda puede volverse enorme y el tiempo de ejecución inaceptable. Asimismo pueden encontrar trayectorias subóptimas al problema. Por lo que es deseable usar otra estrategia que nos permita visitar solo los nodos que nos acerquen al objetivo. Si conocemos un poco más del grafo, como el costo de moverse entre nodos o la distancia al objetivo, podemos mejorar significativamente el rendimiento de los algoritmos, a este tipo de búsqueda se le conoce como *informada* [28] pues usa información extra a la adyacencia de los nodos del grafo .
El algoritmo de Djikstra es un algoritmo que está garantizado en encontrar el camino óptimo, bajo alguna métrica o costo, de un nodo origen al destino. En el algoritmo 8 se describe este proceso:

Se genera un conjunto *costo* que codifica el costo de moverse del origen a cada nodo, inicialmente se asigna infinito o un valor muy grande. Asimismo se crea otro conjunto *path* que se utiliza para almacenar la trayectoria resultante. A continuación se crea el conjunto Q y se añade el origen con un costo de 0. Se procede entonces a tomar el elemento u con menor costo acumulado dentro de Q. Se examinan todos los vecinos v de u y si se encuentra que el costo de moverse a v desde u es menor a $costo_v$ se actualiza este valor y se escoge este nodo como parte del camino almacenándolo en $path_v = u$. Esto se repite hasta que Q este vacío o se llegue al destino. Finalmente

Algoritmo 8 Djikstra

Input: $Grafo$
Input: $Origen$ - Nodo origen
Input: $Destino$ - Nodo destino
  for all vértice v en Grafo **do**
    $costo_v = infinity$
    $path_v = uknwn$
  Q - nodos a expandir
  Añadir Origen a Q
  $costo_{origen} = 0$
  while Q no vacío **do**
    u = Q con minimo costo
    remover u de Q
    for all vecino v de u **do**
      $A = costo_v + peso(u, v)$
      if $A < costo_v$ **then**
        $costo_v = A$
        $path_v = u$
  return prev[]

el camino puede ser reconstruido a partir de recorrer $path_v$ en el sentido inverso, para lo cual se puede introducir la secuencia en una pila y al leerla tendrá el orden correcto.
Cabe resaltar que *costo* se refiere al peso acumulado de recorrer desde el origen hasta en nodo u por otro lado $peso(u, v)$ es el peso o costo de moverse desde un nodo u a otro v. Para mantener un desempeño aceptable el conjunto Q se implementa como una pila de prioridad esto nos permite tener el nodo con el menor costo de la pila en un tiempo constante.
A pesar de estar garantizado que tendremos una ruta óptima del nodo origen al destino es un algoritmo bastante exhaustivo que busca en todos los vecinos del nodo pero no es dificil imaginar que no todos estos se acercan al destino. Otra opción consiste en utilizar una estimación de que tan cerca se encuentra el nodo actual del destino, en lugar del costo, y seguir el nodo que esté más cerca del nodo final. Este tipo de algoritmo llamado *Greedy First-Best-Search* o busqueda voraz. Este es bastante rápido ya que no busca sobre todos los nodos, solo los más prometedores. Pero tiende a encontrar soluciones subóptimas.
Sin embargo se pueden combinar ambas ideas: Usar el costo de movimiento y la estimación, también llamada heurística, para efectuar la búsqueda. A este algoritmo se le conoce como A*. La diferencia con Djikstra es que al costo acumulado se añade la heurística escogida. Este se describe en el algoritmo 9.

La heurística puede ser cualquier función que relacione dos nodos, su única restricción es que siempre debe subestimar el costo real de moverse del nodo al destino. Esto garantiza que siempre se encuentre un camino óptimo. En el caso de un espacio de cuadrícula, como las celdas de ocupación, se utiliza la distancia de Manhattan, si además el agente se puede mover en diagonal se utiliza la distancia de Chebyshev y finalmente si es una búsqueda en un espacio métrico se puede utilizar la norma L2 o distancia Euclideana.

Generación de Trayectorias

Después de haber ensamblado el grafo con los nodos navegables y dado un origen y destino se puede generar una trayectoria válida utilizando las técnicas mencionadas en la sección anterior. Para nuestro caso se utilizó A* y el peso entre los nodos del grafo es la distancia euclideana entre ellos esto lleva a generar caminos cortos. La función heurística es también la distancia euclidiana entre el nodo actual y el destino.
Al recibir una solicitud para generar una trayectoria se verifica que tanto el origen y el destino no se encuentren en colisión con algún nodo ocupado, utilizando las

Algoritmo 9 A*

```
Input: Grafo
Input: Origen - Nodo origen
Input: Destino - Nodo destino
  for all vertice v en Grafo do
      costo_v = infinity
      prev_v = uknwn
  Q nodos a expandir
  Añadir Origen a Q
  costo_Origen = 0
  while Q no vacío do
      u = Q con mínimo costo
      remover u de Q
      if u == Destino then
          break
      for all vecino v de u do
          alt = costo_v + peso(u, v) + heuristica(v, destino)
          if alt < costo_v then
              costo_v = alt
              prev_v = u
  return prev[]
```

mismas técnicas de detección de colisiones ya descritas. Una vez validadas las terminales de la trayectoria se tiene que buscar su vecino más cercano, ya que al ser un mapa disperso el inicio o final de la trayectoria pueden estar fuera del conjunto de los centroides navegables ya definidos. Se añaden estos dos puntos al grafo sobre su vecino mas cercano y se recorre por medio de A*. Esto nos retorna la trayectoria. Todo esto se resume en el algoritmo 10.

Algoritmo 10 Planeador de Trayectorias

Input: $Grafo$ - Mapa de nodos navegables
Input: Occ - Nodos ocupados
Input: $Free$ - Nodos libres
Input: $Origen$ - Nodo origen
Input: $Destino$ - Nodo destino
Input: $height$ - Altura de seguridad
  for all nodo o en Occ **do**
    $Origen_t = Origen + 0, 0, height$
    if CylinderCollision($Origen, Origen_t, o$) **then**
      return Origen invalido
    else if CylinderCollision($Destino, Destino_t, o$) **then**
      return Destino invalido
  $Inicio = \text{argmin}_j \, ||Origen - Free_j||$
  $Final = \text{argmin}_j \, ||Destino - Free_j||$
  return A*(Grafo,Inicio,Final)

Finalmente una trayectoria generada por este sistema se puede observar en la figura 3.8

3.1.8. Soporte de Voxeles

Realizar la verificación de la posición inicial y final es un proceso muy costoso y lento. Debido a que la lista de nodos no está ordenada de ningún modo se tiene que hacer una busqueda exhaustiva sobre todos ellos. Asimismo la mayoría de los nodos ocupados están demasiado lejos del punto a revisar para que valga la pena hacer la prueba de colisión, esto nos lleva a un proceso muy lento y computacionalmente costoso.
Pensando en esto podemos usar un método similar a la subdivisión espacial. Otro

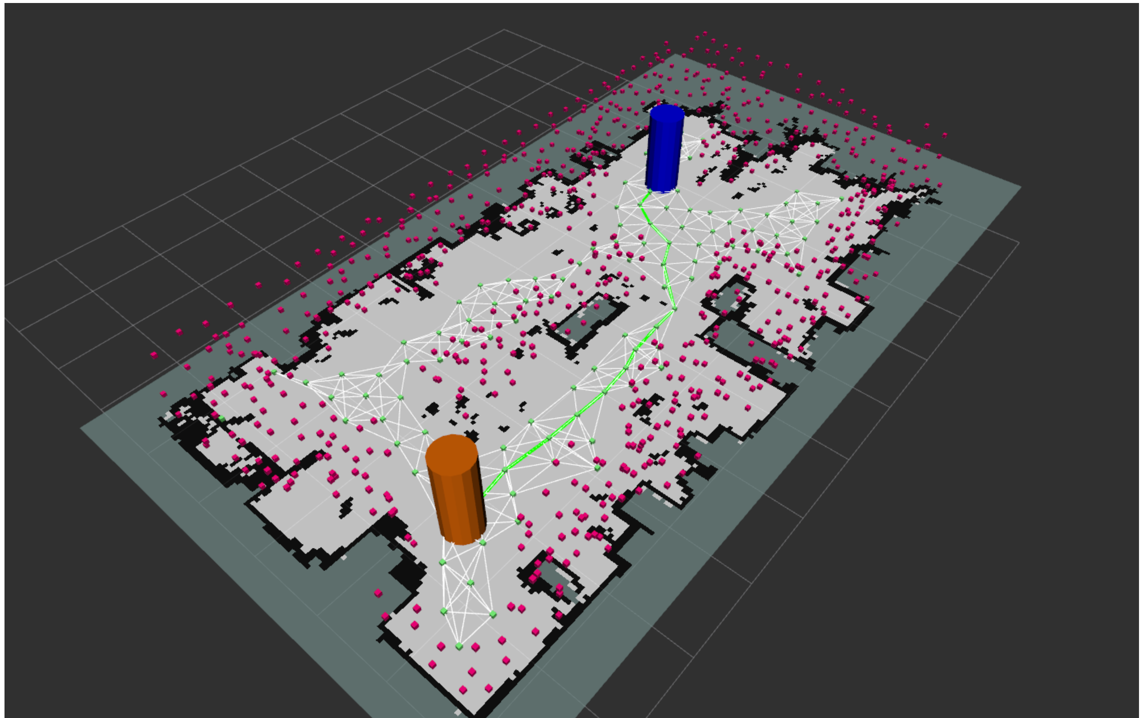


Figura 3.8: Un camino generado mediante el servidor de trayectorias. La posición inicial y final del robot se muestra como cilindros de colores.

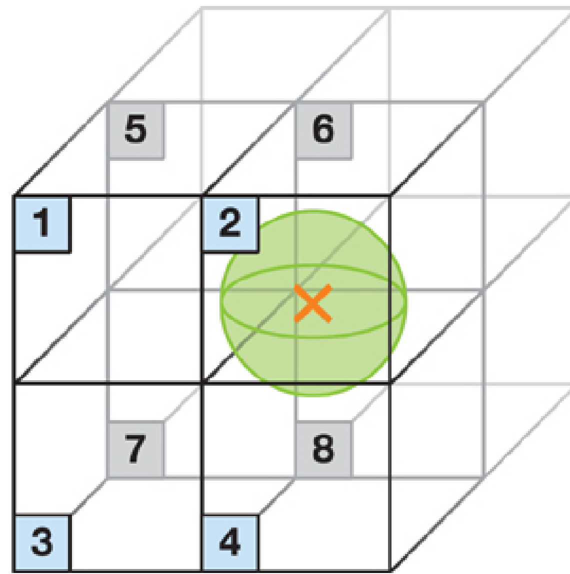


Figura 3.9: La subdivisión espacial permite limitar el número de detecciones de colisiones a realizar, imagen de NVIDIA

algoritmo de detección de colisiones para sistemas de gráficos por computadora, en la figura 3.9 se ilustra como funciona.

En este se crea un arreglo de voxeles orientados con los ejes de tamaño arbitrario sobre el espacio que ocupan los nodos de interés y a cada voxel se asignan todos los centroides que estén dentro del él. Esto nos permite un acceso rápido y geométricamente coherente a los nodos: Dado un punto en el espacio, el número y tamaño de cada voxel es trivial encontrar a cual de todos los voxeles en el arreglo corresponde. Así se realiza la detección de colisiones solo sobre los códigos que pertenezcan al mismo voxel o sobre alguna vecindad, ya sea 4, 6, 8 o 27, el proceso de crear los voxeles se describe en el algoritmo 11 y la forma en que a partir de un punto se obtienen sus vecinos mediante los voxeles en el algoritmo 12. Cabe resaltar que el arreglo de voxeles se implementa como un arreglo lineal por simplicidad es decir: $V[idx] = V[i,j,k]$ donde $idx = i + jC_x + kC_z$.

Algoritmo 11 Voxelize

Input: X - Lista de nodos a voxelizar
Input: u, v, j - Tamaño del voxel en x, y, z

```
x_min = min(X)
x_max = max(X)
C = (x_max - x_min)/(u,v,j)  Número de celdas en cada eje
i = 0, j = 0, k = 0
v_i = x_min
v_x^{i+1} = v_x^i + (i+1)u
v_y^{i+1} = v_y^i + (j+1)v
v_z^{i+1} = v_z^i + (k+1)w
V - Lista de voxeles de tamaño C_x C_y C_z
while X no vacío do
    for all nodo n en X do
        if x pertenece a V[i,j,k] then
            V[i,j,k].append(x)
    i = i + 1
    if i > C_x then
        i = 0
        j = j + 1
    if j > C_j then
        j = 0
        k = k + 1
return V
```

Algoritmo 12 Vecinos en Voxeles

Input: V - arreglo de voxeles
Input: p -punto a evaluar

```
i = p.x - V.x_min / C_x
j = p.y - V.y_min / C_y
k = p.z - V.z_min / C_z
return V[i,j,k]
```

Capítulo 4

Detalles de Implementación

En este capítulo se detallará la implementación del sistema ya descrito. Los módulos previamente mencionados se comunican mediante la librería de ROS [29] un conjunto de paquetes de software orientado a la robótica, el cual está basado en la comunicación entre distintos procesos a partir de mensajes y servicios lo que permite desacoplar las diferentes partes de un sistema robótico de forma sencilla.
La estructura del sistema de mapeo implementado mediante mensajes de ROS se observa en la figura 4.1 Algunos detalles más puntuales y relevantes para la implementación se detallan a continuación.

4.1. Procesamiento de nubes de puntos

Gracias a ROS es posible volver al sistema completo agnóstico a la forma en que recibe esta entrada. Siempre y cuando sea del tipo PointCloud2, un tipo de datos usado para comunicar nubes de puntos de n dimensiones, utilizado por ROS. Esto también nos ofrece la ventaja de utilizar de forma natural la biblioteca de PCL o *Point Cloud Library* [8], esta librería provee de forma directa funciones para procesamiento de nubes de puntos. Se utilizó para implementar el filtro de distancia de la nube, la voxelización inicial y también ofrece interfaces para cargar archivos de mallas y puntos tales como pcd, ply y obj formatos estándar en el ámbito de gráficos 3D.
Por otro lado la nube de puntos es reportada en el marco de referencia del sensor en el cual no es muy práctico trabajar, para transformar la nube de puntos a un marco más conveniente además de PCL se utilizó Tf una biblioteca diseñada para ROS que se encarga de calcular y rastrear las transformaciones existentes entre diferentes marcos de referencia, fijos o móviles, de un robot.

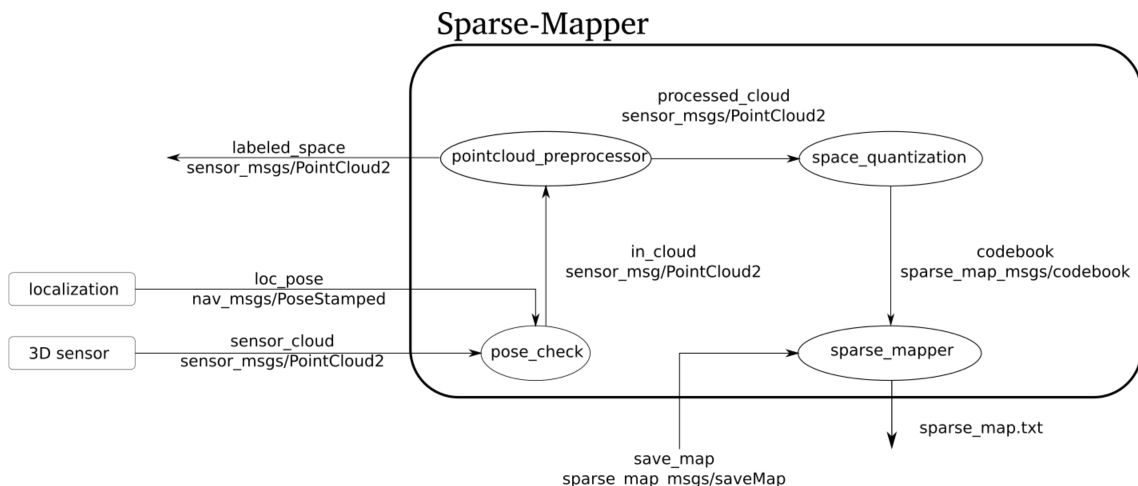


Figura 4.1: Diagrama general del software de mapeo

4.2. Localización

Como se mencionó la forma en que se obtenga una localización es indistinto para el sistema de mapeo. Pero para este documento se usó un método basado en láser y muestreo de Montecarlo, este ya se encuentra implementado como un paquete de ROS llamado Localización Adaptable de Montecarlo o *AMCL* por sus siglas en inglés [16]. Los mapas de celdas de ocupación requeridos fueron generados a partir de Hector-Slam [14] estos mapas también se utilizaron posteriormente para evaluación y comparación de los algoritmos aquí presentados.

4.3. Implementación en CUDA

En los últimos años la tecnología de arquitecturas paralelas ha avanzado mucho gracias a la masificación de tarjetas gráficas o GPU. En un inicio utilizadas principalmente para modelado en 3D y videojuegos, se han utilizado con éxito en el area de cómputo científico y de alto rendimiento donde se requiere operar con una gran cantidad de datos.
En este caso se utilizó un GPU de NVIDIA para acelerar el procesamiento de la nubes de puntos ya que en una sola toma un sensor simple de luz estructurada genera 307,200 puntos en el espacio. Sensores de mayores prestaciones como LIDARs son capaces de generar hasta un millón de puntos por segundo por lo que se vuelve atractivo paralelizar todo el proceso.

Los algoritmos de agrupamiento previamente descritos son paralelizables, principalmente en el paso de cálculo de distancias y de afinación de los centroides.
En un GPU o arquitectura paralela se tiene una estructura de Instrucción Simple y Datos Múltiples o *SIMD* por sus siglas en inglés. Donde a un conjunto de diferentes datos se le aplican las mismas operaciones dentro de hilos separados.
Para el algoritmo de K-medias implementado se sigue la siguiente estructura por cada hilo o dato separada en varios Kernels.

Algoritmo 13 K-medias CUDA

Input: space - N puntos en el espacio (x,y,z)
Input: C_m - Alfabeto inicializado
Input: i - Número de iteraciones
Input: k - Clusters
  for 0 hasta i **do**
    partition = prepararParticion(space,C_m)
    for 0 hasta k **do**
      avgReduce = promParalelo(partition, space)
  return partition , C_m

Algoritmo 14 Preparar partición

Input: points - N puntos en el espacio (x,y,z)
Input: C_m - Alfabeto inicializado
  i indice del hilo
  $Partition$ - Puntos separados por centroide
  p_i punto del espacio a operar en este hilo.
  $partition_i = \text{argmin}_m \|p_i - y_m\| \; y_m \in C_m$
  return $partition$

En el algoritmo 13 se demuestra el proceso general a seguir: Se siguen las mismas dos etapas en la implementación paralela que en la serial, generación de la partición y refinamiento. Primero se calcula la distancia de un punto a todos los códigos esto genera una partición: Un arreglo de etiquetas que relacionan cada punto del espacio a código más cercano. Este proceso se describe en el algoritmo 14.
Posteriormente se refina el centroide recalculando la media de cada grupo en la partición, para esto se separa en un arreglo temporal que solo contiene elementos pertenecientes a un sólo centroide k y se aplica una reducción paralela tradicional [30]. Para calcular la cardinalidad en el algoritmo 15 se utilizan operaciones atómicas.

Algoritmo 15 Promedio en paralelo

Input: points - N puntos en el espacio (x,y,z)
Input: partition - N etiquetas de 0 a k
Input: k - Cluster a operar
  i indice del hilo
  C_k - Centroide a refinar
  for all p in partition **do**
    if $p = k$ **then**
      $temporal_i = space_i$
    else
      $temporal_i = \hat{0}$
  $C_k = \frac{1}{|partition_k|} reduccinParalela(temporal_i)$
  return C_k

Este proceso se repite el número de iteraciones deseadas i para cada elemento del alfabeto. Finalmente el algoritmo regresa tanto los códigos optimizados y la partición del espacio.

4.3.1. Servidor de mapas

En el caso de este módulo también fue implementado mediante ROS. Su esquema se aprecia en 4.2. En este caso la implementación utiliza simplemente C++ 11 y ROS.

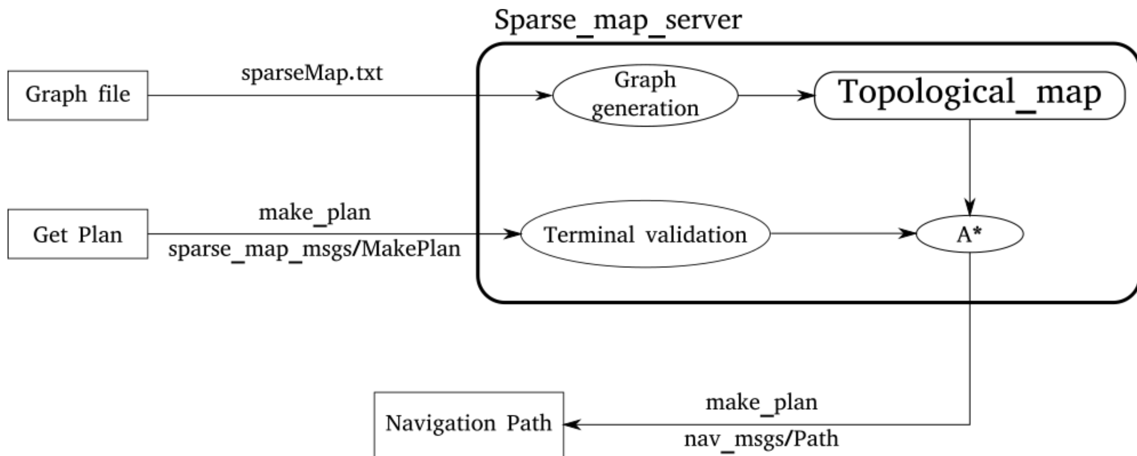


Figura 4.2: Diagrama general del software de mapeo en ROS

Capítulo 5

Pruebas y Resultados

El sistema descrito anteriormente permite la generación de mapas topológicos de forma automática, sin embargo, surge la necesidad de evaluar el desempeño del sistema contra otras metodologías ya existentes.
Se propone así los siguiente parámetros que evaluar en el sistema:

- Calidad de la cuantización del espacio.
- Tiempo de procesamiento para la cuantización.
- Calidad de las trayectorias generadas.
- Tiempo de generación de las trayectorias.
- Tamaño de la representación.

En este capítulo se describirán las pruebas propuestas para evaluar esta metodología y los resultados que estas arrojan. Se consideran dos tipos de pruebas, aquellas centradas en evaluar la cuantización del espacio y aquellas que evalúan las trayectorias generadas. Asimismo se presenta la metodología contra la cual el sistema fue comparado.

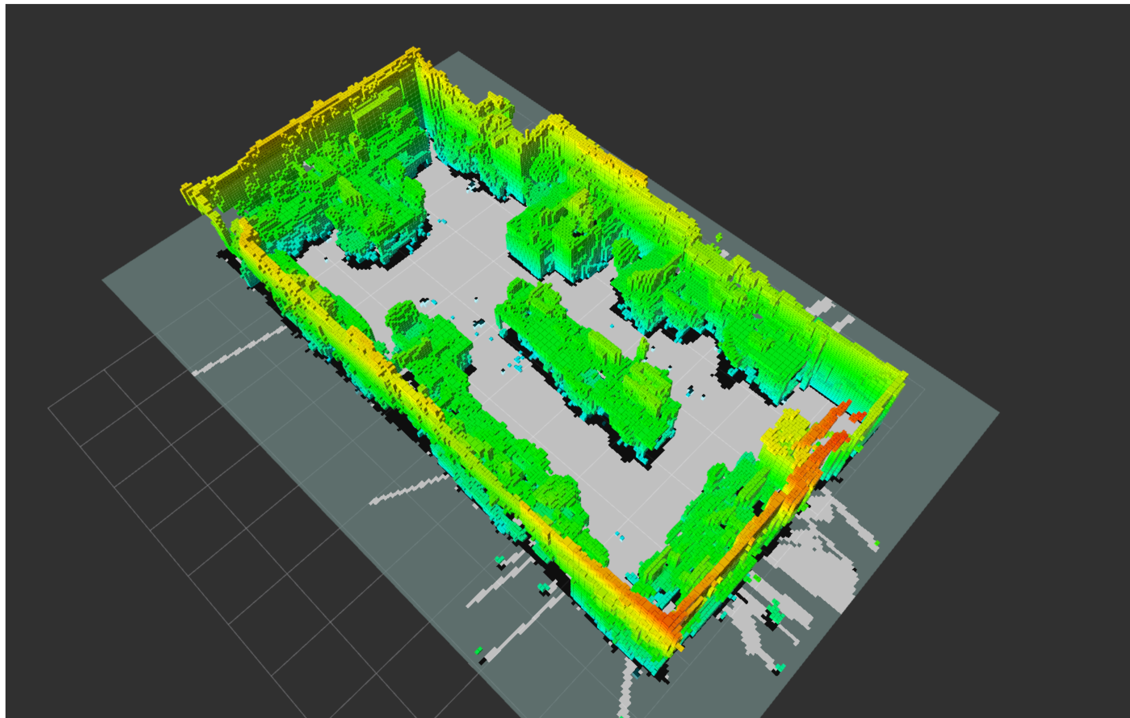
5.1. Celdas de ocupación aumentadas mediante Octomapas

Como ya se discutió en la sección 2.6 otra representación muy utilizada basada en mapeo dado poses conocidas es el de Octomapas, los mismos autores de esta

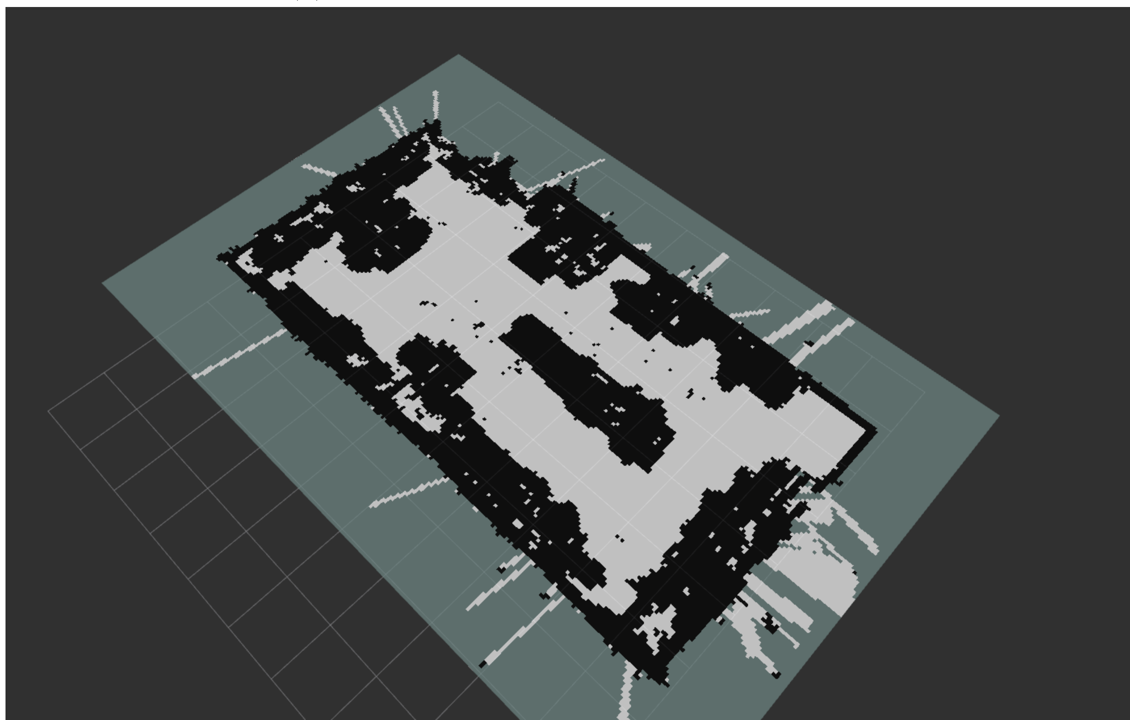
metodología también proponen un paquete de planeación en tres dimensiones [31] que utiliza una representación basada en octomapas para generar capas del ambiente a diferentes alturas. Ellos planean en el espacio de configuración del robot y la información adicional les permite evitar una gran cantidad de costosas detecciones de colisión sobre las trayectorias propuestas.
De acuerdo a sus publicaciones sus resultados fueron satisfactorios, generando trayectorias validas en 3D en espacios muy desordenados, adicionalmente proveen su sistema como software libre, sin embargo se encuentra desactualizado y a conocimiento del autor su uso nunca se popularizó debido a que no funciona en tiempo real.
Sin embargo se propone el uso de octomapas para generar un mapa de celdas de ocupación aumentado con la información tridimensional, esta idea se encuentra inspirado en la metodología del robot Cosero [32]. Este robot de servicio construye un mapa de celdas de ocupación a partir de una representación tridimensional basada en surfeles proyectada en el plano xy.
Para esta metodología se genera una reconstrucción 3D del ambiente mediante Octomapas, al ser también este un sistema de mapeo dado poses conocidas requiere un método de localización externo, aquí también se utiliza el paquete de AMCL [2] para calcular la pose.
Una vez generado el octomapa se elimina el piso y en caso de ser necesario el techo y se proyecta la información al plano XY lo que nos permite obtener un mapa de celdas de ocupación clásico. La ventaja de esto es que este mapa es completamente compatible con el Navigation Stack de ROS, uno de los paquetes más usados para cálculo de trayectorias para robótica móvil, y al mismo tiempo codifica la información de los obstáculos tridimensionales que no son fácilmente captados por el láser plano.
Todo esto se hace con el fin de tener un método a comparar con nuestra propuesta: Sparse-Mapper, siendo a su vez compatible con todo el software ya escrito para navegación en mapas planos. El resultado de este pequeño sistema se ilustra en la figura 5.1.

5.2. Pruebas de cuantización

A diferencia de los sistemas de mapeo densos, donde a cada punto del espacio se le asigna una propiedad, el sistema propuesto genera una representación dispersa del mismo generada a partir de los algoritmos descritos en la sección 3.1.4. Para evaluar este tipo de algoritmos, por lo general, se consideran como un problema de clasificación. Es decir, se tiene un conjunto de datos etiquetado en ciertas clases,



(a) Octomapa del laboratorio de Biorobótica



(b) Mapa de celdas de ocupación aumentado

Figura 5.1: Reconstrucción basada en octomapas y el mapa plano generado

dada esta separación se puede evaluar si un algoritmo dado es capaz de segmentar los datos de forma congruente a las clases propuestas.
Sin embargo, en este caso no se cuenta con ninguna correspondencia *a priori* de la nube de puntos a los centroides. Pues simplemente se buscó una forma de comprimir el espacio. Considerando esto se puede utilizar la distorsión media del resultado de la cuantización como medida de desempeño, lo que equivale a medir el error de cuantización del algoritmo.
La distorsión media se calcula de la siguiente forma:

$$\phi = \frac{1}{|X|} \sum_{x \in X} \min_{c \in C} ||x - c||^2 \quad (5.1)$$

Es decir es una suma de la distancia de cada dato en X a su centroide más cercano en el alfabeto C dividido entre el numero de datos $|X|$. Esta métrica es muy similar a la propuesta en [26] para evaluar el desempeño de su algoritmo de agrupamiento, pero en este trabajo se añade el término de normalización $\frac{1}{|X|}$ para manejar una media muestral.
Evaluar la distorsión media nos ayuda no solo a medir la calidad de la cuantización sino también comparar diferentes algoritmos e hiperparámetros asociados, siendo estos los parámetros que el usuario suministra al sistema. Con el fin de encontrar el mejor algoritmo para este tipo de tarea.
Las pruebas fueron estructuradas de la siguiente manera: Se tomaron los cuatro algoritmos de agrupamiento implementados para este sistema:

- K-medias en CPU.
- K-medias en GPU.
- K-medias ++.
- Linde-Buzo-Gray.

Y se ejecutó cada algoritmo varias veces con el mismo número fijo de centroides en la misma escena, después se aumentó el número de centroides por un valor constante y se repite hasta tener el número deseado de códigos. Las escenas se tomaron tanto de conjuntos de datos de imágenes RGBD públicos así como de reconstrucciones propias hechas mediante el paquete RTAB-Map [22] y Octomap[10].
Los datasets utilizados fueron:

- RGB-D SLAM Dataset and Benchmark [33].

- NYU Depth Dataset V2 [34].
- Stanford 2D-3D-Semantics Dataset (2D-3D-S) [35].
- Indoor Lidar-RGBD Scan Dataset (Redwood) [36].

Asimismo se utilizaron algunas nubes de puntos adquiridas por nuestra cuenta. Las características de las escenas utilizadas como tamaño y formato se mencionan en la tabla: 5.1. Cabe mencionar que dos de los datasets RGBD y NYU se basan en tomas individuales de una cámara de profundidad, mientras que los demás se componen de reconstrucciones completas de pisos y en general ocupan un volumen mayor y tienen, más puntos. Por eso en este documento se agrupan los datasets en dos grupos para mostrar resultados. Por claridad aquí solo se reporta un nombre corto de la escena utilizada, en los anexos se puede consultar exactamente el nombre de cada archivo. Una muestra de la información que suministran los datasets se aprecia en la figura 5.2 y 5.3

Escena	Puntos	Formato	Dataset
Recamara	914,163	PLY	Redwood
Pasillo Stanford	8,349,813	Imagen EXR	2D-3D-Semantics
Área 5a Stanford	122,936	OBJ	2D-3D-Semantics
Biorobótica	2,088,043	PCD	Elaboración propia
WRS2018	1,362,112	PCD	Elaboración propia
NYU-60	307,200	Imagen de profundidad PGM	NYU-v2
NYU-30	307,200	Imagen de profundidad PGM	NYU-v2
NYU-28	307,200	Imagen de profundidad PGM	NYU-v2
NYU-15	307,200	Imagen de profundidad PGM	NYU-v2
NYU-13	307,200	Imagen de profundidad PGM	NYU-v2
RGBD-85	250,339	Imagen de profundidad PGM	RGBD-Pioneer
RGBD-53	262,830	Imagen de profundidad PGM	RGBD-Pioneer
RGBD-44	254,829	Imagen de profundidad PGM	RGBD-Pioneer
RGBD-43	249,184	Imagen de profundidad PGM	RGBD-Pioneer
RGBD-27	147,932	Imagen de profundidad PGM	RGBD-Pioneer

Tabla 5.1: Resumen de escenas RGBD utilizadas

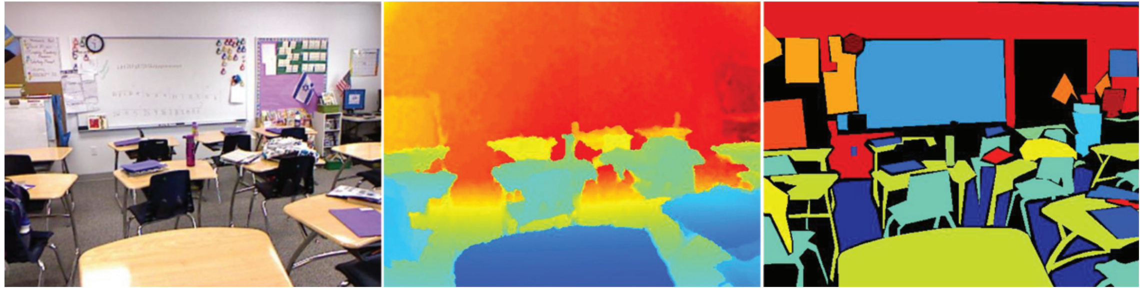


Figura 5.2: información ofrecida por el dataset NYU-v2, color, profundidad y segmentación semántica, en nuestro caso solo se utilizó la información de profundidad para generar una nube de puntos

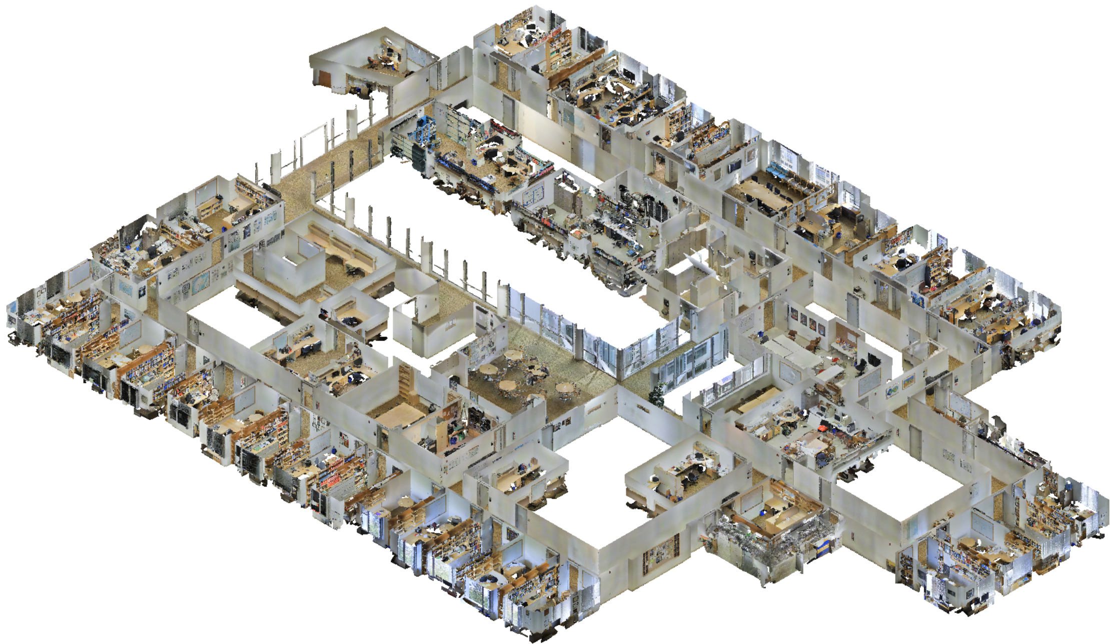


Figura 5.3: En el caso del Dataset Stanford 2d-3d-semantic se ofrece toda la información como una malla, esta contiene la nube de puntos y el color así como la segmentación. De nuevo solo se utilióo la información tridimensional

5.2.1. Resultados

En la tabla 5.2 se indican los hiperparámetros usados para el cuantizador, estos se mantuvieron constantes para casi todas las pruebas. En el caso del algoritmo de LBG no se aumentó de forma uniforme el número de centroides ya que el algoritmo solo funciona con potencias de 2. Así que se probó con 16,32,64,128 y 512 centroides solamente. Asimismo al ser un algoritmo determinístico se corrió una sola vez ya que siempre generaba el mismo resultado.

Tabla 5.2: Hiperparámetros del cuantizador

Parámetro	Valor
Iteraciones de Lloyd	7
Centroides mínimos	16
Centroides máximos	512
Incremento de centroides	16
Experimentos	10

Finalmente todas la pruebas fueron realizadas utilizando ROS Kinetic en Ubuntu 16.04 con PCL 1.7 corriendo en un Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz con 12 núcleos y un GPU Quadro P4000 de NVIDIA.

5.2.2. Tablas

En las tablas 5.3 y 5.4 se muestra el promedio de la distorsión y tiempo de ejecución para 16, 32 , 64, 128, 256 y 512 códigos por cada juego de datos, no se muestran todos los valores usados por claridad, se reporta el promedio de cada juego de datos, la línea de Heavy se refiere a los archivos de Standford, Redwood y Biorobótica estos se agrupan ya que, a diferencia de RGBD y NYU que son tomas de profundidad independientes, todos son reconstrucciones de pisos completos.

Tabla 5.3: Distorsión media

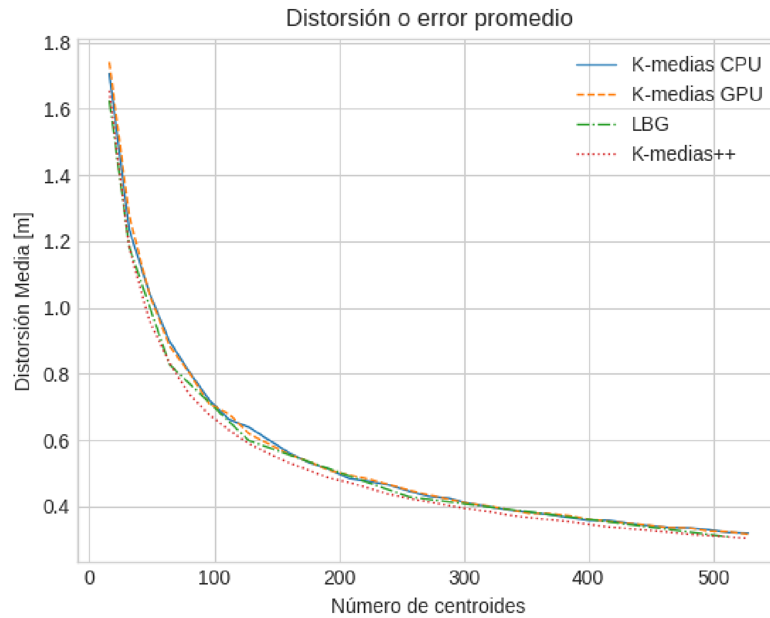
Distorsión media [m/punto]					
Dataset	Centroides	K-medias CPU	K-medias GPU	LBG	K-medias ++
RGBD	16	1.7918	1.8109	1.7655	1.7549
	32	1.2893	1.2918	1.2455	1.2387
	64	0.9106	0.9110	0.8803	0.8668
	128	0.6456	0.6429	0.6242	0.6132
	256	0.4581	0.4556	0.4446	0.4359
	512	0.3260	0.3275	0.3164	0.3117
NYU	16	5.7932	5.8180	5.7823	5.5929
	32	4.0506	3.9854	3.9560	3.8353
	64	2.7731	2.7597	2.7286	2.6398
	128	1.9097	1.9088	1.8920	1.8015
	256	1.3196	1.3236	1.3029	1.2226
	512	0.9048	0.9104	0.9044	0.8366
Heavy	16	1.2718	1.2595	1.2400	1.2520
	32	0.9363	0.9305	0.9079	0.9202
	64	0.6868	0.6882	0.6742	0.6811
	128	0.5054	0.5072	0.4964	0.4982
	256	0.3722	0.3709	0.3621	0.3636
	512	0.2682	0.2700	0.2630	0.2640

Tabla 5.4: Tiempo de Ejecución

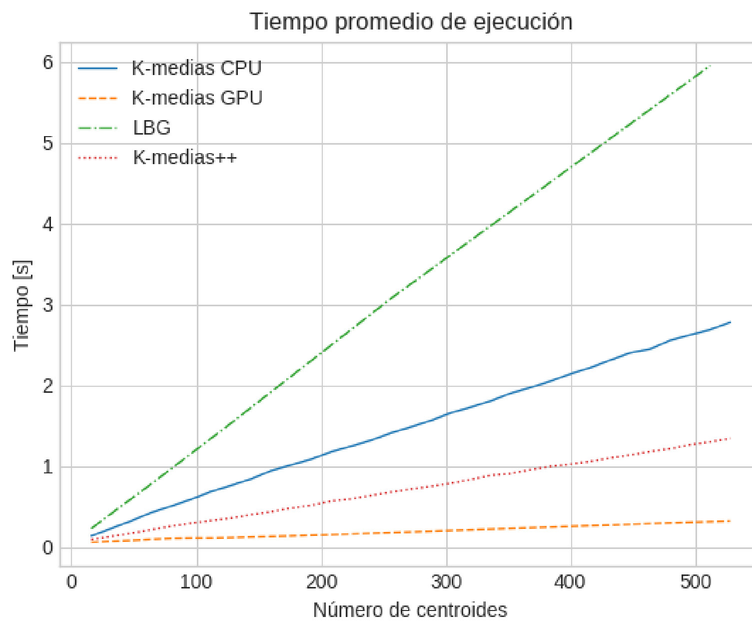
Tiempo de ejecución[s]					
Dataset	Centroides	K-medias CPU	K-medias GPU	LBG	K-medias ++
RGBD	16	0.1344	0.0542	0.2094	0.0878
	32	0.2176	0.0670	0.3770	0.1262
	64	0.3847	0.0894	0.7130	0.2029
	128	0.7009	0.1112	1.4152	0.3363
	256	1.3131	0.1679	2.8808	0.6204
	512	2.5117	0.2936	5.5480	1.2069
NYU	16	0.1741	0.0723	0.2824	0.1124
	32	0.2846	0.0849	0.4990	0.1571
	64	0.5084	0.1140	0.9476	0.2560
	128	0.9378	0.1324	1.8824	0.4314
	256	1.7303	0.2130	3.6854	0.8113
	512	3.3256	0.3719	7.4112	1.5569
Heavy	16	1.4141	0.5040	2.3570	0.8187
	32	2.2989	0.5633	4.2594	1.2007
	64	4.1520	0.7176	8.1108	2.0085
	128	7.6165	1.0087	15.8430	3.5605
	256	15.3984	1.5983	31.4382	6.7505
	512	27.7194	2.8538	61.4796	13.0198

5.2.3. Gráficas

En las figuras 5.4 a 5.6 se muestra la gráfica de, la distorsión media y el tiempo promedio de cuantización para una escena contra el número de centroides, cada curva representa un algoritmo diferente.
Se muestra sólo un conjunto de gráficas representativas para el conjunto de datos de RGBD una para NYU y una la del pasillo de Stanford. En todos los casos la tendencia es la misma.

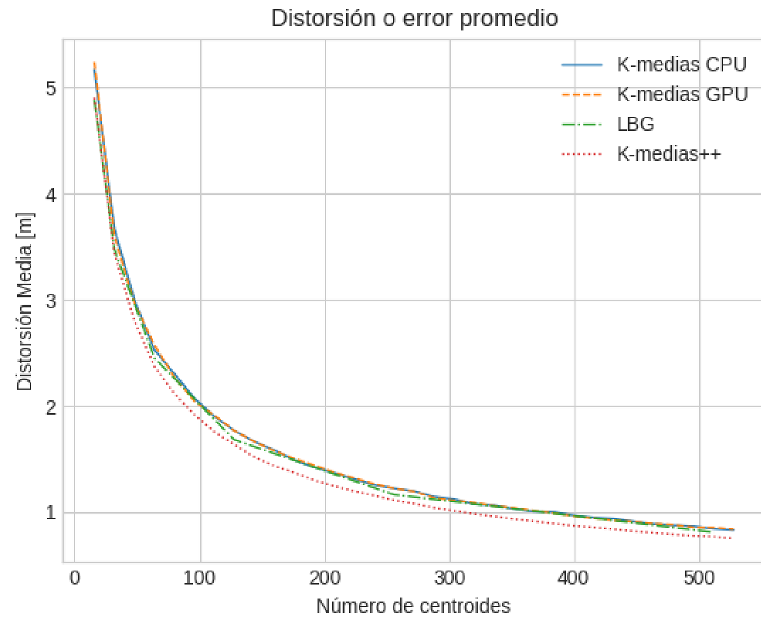


(a)

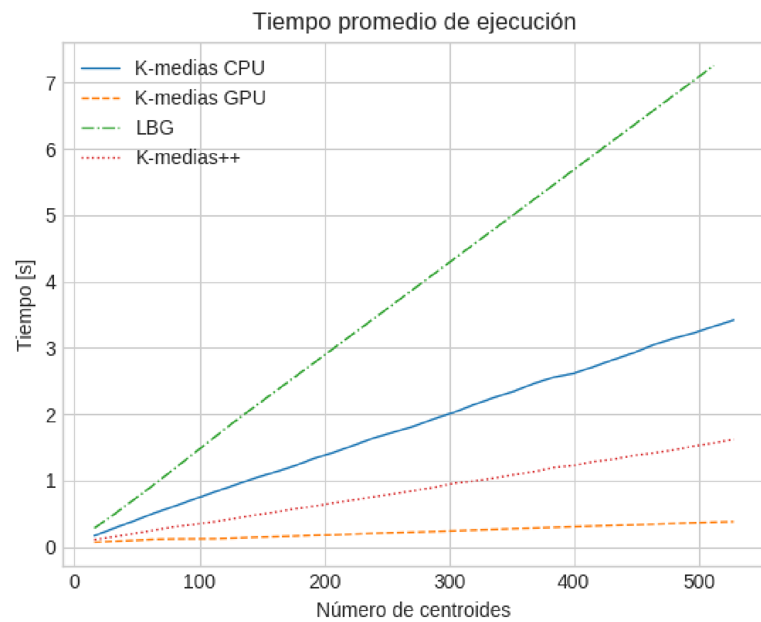


(b)

Figura 5.4: Distorsión media y tiempo de ejecución para RGBD-85

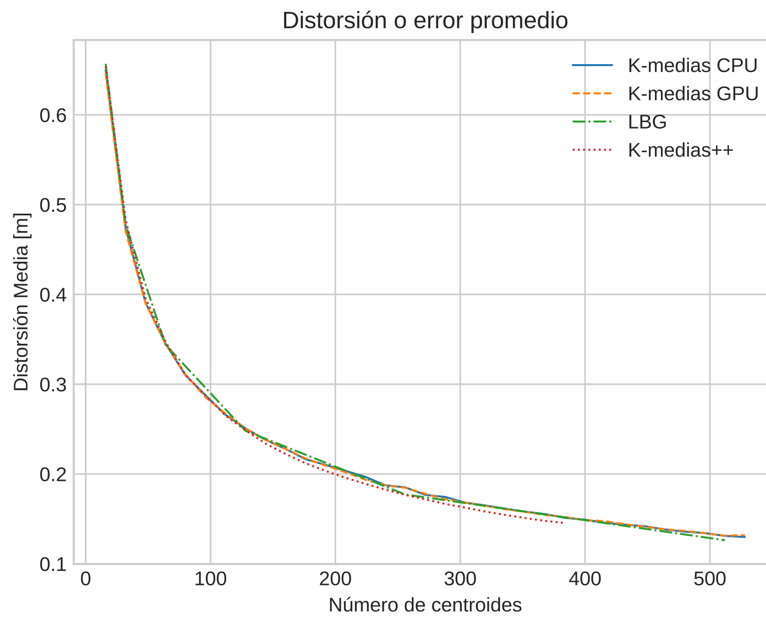


(a)

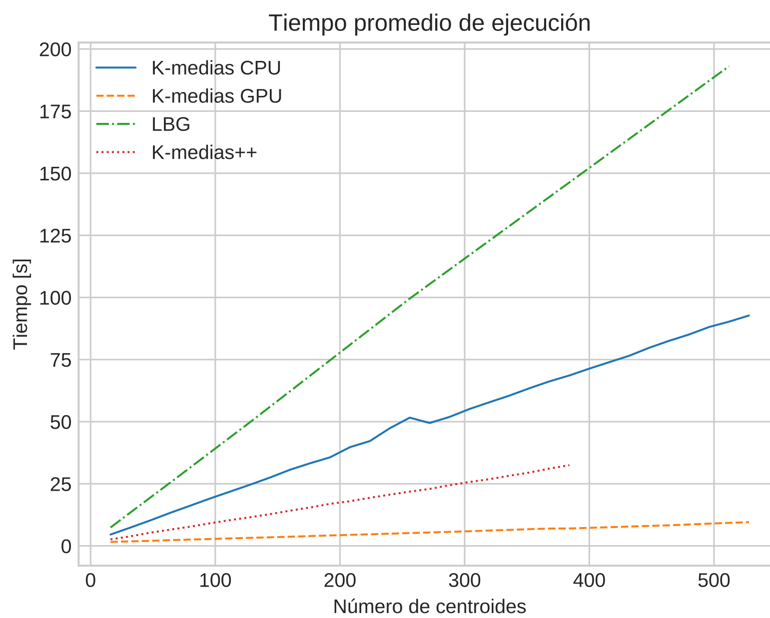


(b)

Figura 5.5: Distorsión media y tiempo de ejecución para NYU-60



(a)



(b)

Figura 5.6: Distorsión media y tiempo de ejecución para pasillo de Standford

5.3. Pruebas de trayectorias

El segundo módulo de nuestro interés es el de generación de trayectorias en el grafo. Para evaluar este se propuso medir las siguientes características especificas de un gran número de trayectorias generadas sobre diferentes mapas:

- Longitud de la trayectoria.
- Número de nodos visitados.
- Tiempo de ejecución.
- Suavidad de la trayectoria.

Para evaluar esto se necesita una base de comparación: Un mapa que indique de antemano el espacio navegable y ocupado. Para esto se crearon algunos mapas de algunos ambientes reales y simulados así como de archivos de nubes de puntos de los conjuntos de datos ya mencionados. Ya sean mapas planos por medio de técnicas de SLAM 2D tradicionales como Hector SLAM, o en algunos casos, se utilizó RTAB-Map u Octomap para generar una reconstrucción tridimensional completa del ambiente. Esta reconstrucción se proyectó al plano xy para formar un mapa de celdas de ocupación tradicional. Este mapa se post-procesó y se ajustó de forma manual. A partir de éste se pueden proponer trayectorias que se sabe de antemano si son o no viables, es decir si ambos puntos, el inicio y el final, están dentro de celdas navegables. Cabe mencionar que en este caso no se evaluó el caso de mapas disjuntos, es decir, que a pesar de que tanto el origen como el destino sean posiciones libres no existe camino que una ambas poses.
Una vez creadas las referencias se construyeron mapas topológicos de los mismos ambientes, tanto de forma incremental usando varias escenas conociendo su pose, como a partir de la reconstrucción completa de la planta, esto para evaluar el desempeño en ambos casos. Es de esperar que una reconstrucción completa del ambiente se encuentre más cerca de la distribución real de puntos y que entregue resultados mejores comparado a unir varias escenas. Para los casos en que se generaron mapas de ambientes reales se utilizó el robot Human Support Robot o HSR de Toyota [37], este cuenta con una cámara de profundidad Xtion que se utiliza para adquirir la nube de puntos y un láser Hokuyo lo que nos permite utilizar algoritmos de localización basados en láser, así como un par de cámaras estéreo que también se pueden utilizar para generar nubes de puntos, sin embargo, en este trabajo solo se utilizó la cámara de profundidad. En el caso de ambientes simulados también se utilizó una versión virtual del mismo robot que opera dentro del simulador Gazebo [38] versión 7, una fotografía del robot se aprecia en la figura 5.7.

Figura 5.7: El robot HSR de Toyota usado en las pruebas.

Con lo anterior se pueden evaluar las métricas propuestas y el desempeño del sistema, como punto de comparación se utiliza el sistema descrito en la sección 5.1 utilizando el planeador de trayectorias del Navigation Stack (Globabl Planner) de ROS el cual es el estándar *de facto* para la planeación en robots móviles en ambientes interiores. Ambos mapas, el topológico y la proyección del Octomapa son refinados a mano para eliminar errores de mapeo, estos surgen debido a fallas de localización y a ruido del sensor.
En la tabla 5.5 se muestran los parámetros usados en la creación de los grafos para los mapas a partir de varias escenas. El radio y altura de colisión se refieren a la altura y radio del cilindro definidos sobre el punto final o inicial que no debe intersectar ningún código ocupado para que sean considerados válidos. El radio de conexión es el radio del cilindro que definen dos nodos navegables conectados y que no debe de intersectar con algún nodo ocupado. Vecinos por nodo se refiere a cuantas conexiones debe tener cada nodo. En todos los casos se utilizaron 32 centroides por escena y solo se incluía información al mapa global si se detectaba un avance de al menos 50 centímetros o una rotación de 45 grados, el algoritmo utilizado para cuantizar fue K-medias++ por su bajo error y relativamente rápida ejecución. Además para cada escena la nube de puntos fue preprocesada eliminando todos los puntos a más de cuatro metros del sensor de profundidad, ya que el ruido en la señal es muy fuerte a partir de esa longitud, en el caso de ambientes simulados no se tiene ruido y se

Tabla 5.5: Parámetros para mapas dispersos, mapas incrementales

Ambiente	Nodos totales	Nodos libres	Nodos ocupados	Radio de colisión	Altura de colisión	Radio de conexión	Vecinos por nodo
Lab. de Biorobótica	3904	1445	2459	0.3	1.0	0.3	8
Lab. Proc. de Señales	3747	1724	2023	0.3	1.2	0.2	6
Megaweb HSR	1231	745	486	0.3	1.0	0.3	6
Apartamento HSR	5276	2850	2426	0.3	1.5	0.3	6

Tabla 5.6: Parámetros para mapas dispersos, reconstrucción completa

Ambiente	Nodos totales	Nodos libres	Nodos ocupados	Radio de colisión	Altura de colisión	Radio de conexión	Vecinos por nodo
Lab. de Biorobótica	1024	361	663	0.3	1.0	0.3	8
Lab. Proc. de Señales	2048	529	1519	0.3	1.2	0.2	6
Megaweb	1024	462	562	0.3	1.0	0.3	6
Apartamento HSR	2048	937	1111	0.3	1.5	0.3	6

utilizó el sensor virtual en el rango completo de distancia.
En el caso de los mapas creados a partir de reconstrucciones completas del ambiente la tabla los parámetros utilizados se muestran en la tabla 5.6. En este caso no se preprocesó la nube de puntos.
Para el caso del Navigation Stack de ROS se utilizaron los parámetros por defecto excepto en el planeador global, para este se utilizo el Global Planner con el algoritmo de A* y un mapa de costo inflado hasta 30 centímetros.

Las métricas propuestas se definen formalmente a continuación: El número de nodos se refiere a cuantos puntos tiene cada trayectoria y el tiempo es cuantos segundos tarda cada algoritmo en responder con un plan. La longitud de la trayectoria es la distancia que se recorre siguiendo el plan generado. En ambos casos, Sparsemap y Navigation Stack, regresan la trayectoria como un conjunto de puntos en el

espacio esta se calcula de la siguiente manera:

$$L = \sum_{i=1}^{N} ||P_i - P_{i-1}|| \tag{5.2}$$

$$P_i \in R^3 \tag{5.3}$$

Es decir la suma de la distancia entre cada punto de la trayectoria. Donde N es el número de puntos en la trayectoria.
Se proponen dos medidas de suavidad, una basada en la rotación que impone la trayectoria y otra basada en la traslación: Se define como una medida de tortuosidad la suma del valor absoluto del angulo entre cada dos rectas definidas por puntos contiguos en el camino:

$$T = \sum_{i=1}^{N-1} |arcos \frac{u \cdot v}{||u||||v||}| \tag{5.4}$$

$$u = p_{i+1} - p_i \tag{5.5}$$

$$v = p_i - p_{i-1} \tag{5.6}$$

$$\tag{5.7}$$

Este valor evalúa la cantidad de rotación que impone esta trayectoria. Entre mayor sea este valor el agente rotará más al seguir la trayectoria. Por otro lado se propone como medida de dispersión la desviación estándar de la trayectoria generada.

$$D = \sqrt{\frac{\sum_{i=0}^{N} (p_i - \hat{p})^2}{N-1}} \tag{5.8}$$

$$\hat{p} = \frac{1}{N} \sum_{i=0}^{N} p_i \tag{5.9}$$

Entre más pequeño sea el valor las trayectorias son más compactas y provocan menos traslación en el agente. Cabe resaltar que estas dos últimas métricas son relativas, no se conocen los valores óptimos pero nos permiten comparar entre los diferentes planeadores. Para las pruebas usando el mapa de referencia se muestrearon aleatoria mente 1000 puntos de inicio y destino, entonces se solicitaba a ambos planeadores generar una trayectoria para el mismo par origen-final y se evaluaban las métricas ya descritas cuando ambos planeadores retornaban una trayectoria. Las pruebas se ejecutaron el la misma computadora donde fueron realizadas las pruebas de agrupamiento.

Tabla 5.7: Estadísticos longitud de trayectoria

Longitud [m]				
Mapas generados incrementalmente				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	4.624	3.178	3.728	2.177
Lab. de Procesamiento de Señales	9.539	5.811	9.51	5.59
Megaweb	5.311	2.658	5.364	2.550
Apartamento HSR	7.218	3.524	7.241	3.305
Mapas generados a partir de reconstrucciones completas				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	4.220	3.081	3.559	2.195
Lab. de Procesamiento de Señales	9.445	5.843	9.322	5.535
Megaweb	5.369	2.751	5.346	2.588
Apartamento HSR	7.389	3.626	7.226	3.353

5.3.1. Tablas

Las tablas 5.7 a 5.11 resumen los estadísticos, de las pruebas. A pesar de que se anotó manualmente el espacio libre y el navegable en el mapa usado para generar los puntos de inicio y final en algunos casos los planeadores no eran capaces de calcular un camino entre ambos puntos. En la tabla 5.12 se anexa el número de veces que cada planeador no fue capaz de calcular una trayectoria válida entre ambos puntos. Además ya que todas la métricas son comparativas se indica en la misma tabla cuantas veces ambos planeadores calcularon una ruta.

Tabla 5.8: Estadísticos nodos en trayectoria, mapas generados incrementalmente

Numero de nodos				
Mapas generados incrementalmente				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	80.974	55.034	11.039	4.9899
Lab. de Procesamiento de Señales	176.877	108.381	26.485	14.37
Megaweb	94.343	46.377	14.873	5.905
Apartamento HSR	126.605	62.194	15.933	6.092
Mapas generados a partir de reconstrucciones completas				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	74.050	53.592	8.677	3.558
Lab. de Procesamiento de Señales	175.505	108.580	23.382	12.332
Megaweb	95.125	48.491	13.158	5.188
Apartamento HSR	129.72	63.740	17.141	6.796

Tabla 5.9: Estadísticos tiempo de ejecución del planeador

Tiempo de planeación [s]				
Mapas generados incrementalmente				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	0.00236	0.000827	0.00215	0.000291
Lab. de Procesamiento de Señales	0.00806	0.00711	0.00715	0.00238
Megaweb	0.00882	0.0196	0.00447	0.000921
Apartamento HSR	0.00634	0.00455	0.00549	0.00121
Mapas generados a partir de reconstrucciones completas				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	0.00481	0.00242	0.00362	0.000406
Lab. de Procesamiento de Señales	0.0087	0.00868	0.00640	0.00202
Megaweb	0.0109	0.0253	0.00416	0.000733
Apartamento HSR	0.00750	0.00656	0.00556	0.00116

Tabla 5.10: Dispersión de Trayectorias

Dispersión media [m]				
Mapas generados incrementalmente				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	0.631	0.383	0.682	0.382
Lab. de Procesamiento de Señales	1.437	0.8346	1.524	0.811
Megaweb	0.787	0.364	0.882	0.376
Apartamento HSR	1.053	0.480	1.184	0.500
Mapas generados a partir de reconstrucciones completas				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	0.593	0.382	0.683	0.414
Lab. de Procesamiento de Señales	1.426	0.846	1.546	0.847
Megaweb	0.799	0.381	0.900	0.388
Apartamento HSR	1.081	0.497	1.199	0.51

Tabla 5.11: Tortuosidad de Trayectorias

Tortuosidad [rad]				
Mapas generados incrementalmente				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	23.112	17.319	5.258	1.845
Lab. de Procesamiento de Señales	21.675	18.189	11.177	5.225
Megaweb	11.765	7.693	7.506	2.734
Apartamento HSR	15.874	9.123	8.243	3.076
Mapas generados a partir de reconstrucciones completas				
Ambiente	Navigation Stack		Sparse Map	
	μ	σ	μ	σ
Lab. de Biorobótica	17.772	13.63	4.52	1.511
Lab. de Procesamiento de Señales	21.435	17.561	10.101	4.528
Megaweb	11.697	7.9511	7.061	2.672
Apartamento HSR	16.335	9.277	8.382	3.058

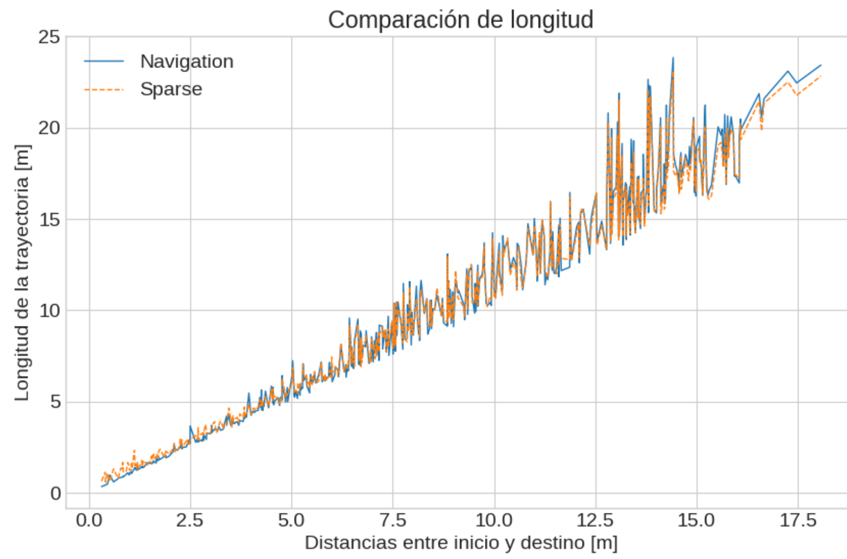
Tabla 5.12: Tabla de trayectorias generadas

Mapas generados incrementalmente				
Ambiente	Caminos solicitados	Caminos no calculados		Caminos comparados
		Navigation Stack	Sparse-Mapper	
Lab. de Biorobótica	1000	420	324	465
Lab. de Proc. de Señales	1000	401	286	536
Megaweb	1000	181	142	750
Apartamento HSR	1000	101	307	679
Mapas generados a partir de reconstrucciones completas				
Ambiente	Caminos solicitados	Caminos no calculados		Caminos comparados
		Navigation Stack	Sparse-Mapper	
Lab. de Biorobótica	1000	534	174	440
Lab. de Proc. de Señales	1000	387	228	568
Megaweb	1000	170	41	816
Apartamento HSR	1000	115	89	853

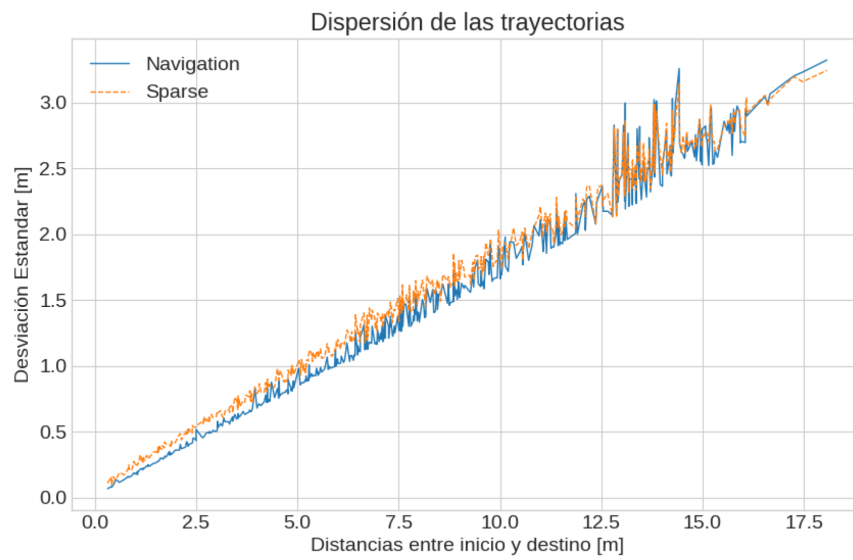
5.3.2. Gráficas

Las gráficas de 5.8 a 5.9 corresponden a mapas generados a partir de varias escenas y 5.10 a 5.11 se refiere a los mapas topológicos generados a partir de reconstrucciones de plantas completas. Solo se muestran gráficas para 2 ambientes, ya que la tendencia en todos los casos fue la misma y las gráficas mostradas son las más representativas de los experimentos.

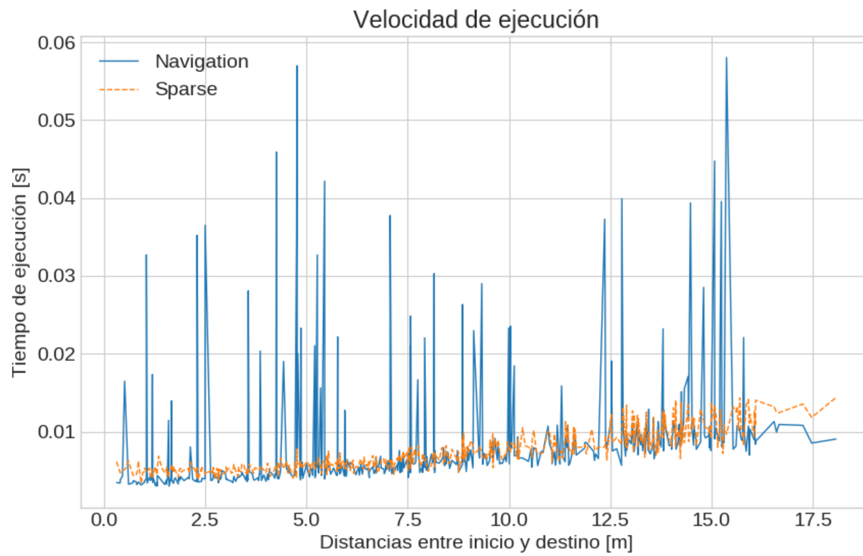
En la gráficas se muestra la longitud de la trayectoria, el tiempo de ejecución, el número de nodos y la tortuosidad del camino propuesto, en el eje de las ordenadas y en el eje de las abscisas la norma euclidiana entre el origen y el destino, la distancia en linea recta entre ambos, esto para observar la tendencia del planeador en puntos cercanos y alejados, asimismo en el caso de la longitud es imposible encontrar una trayectoria más corta que la línea que une ambos puntos, esto nos permite ilustrar la cota inferior.



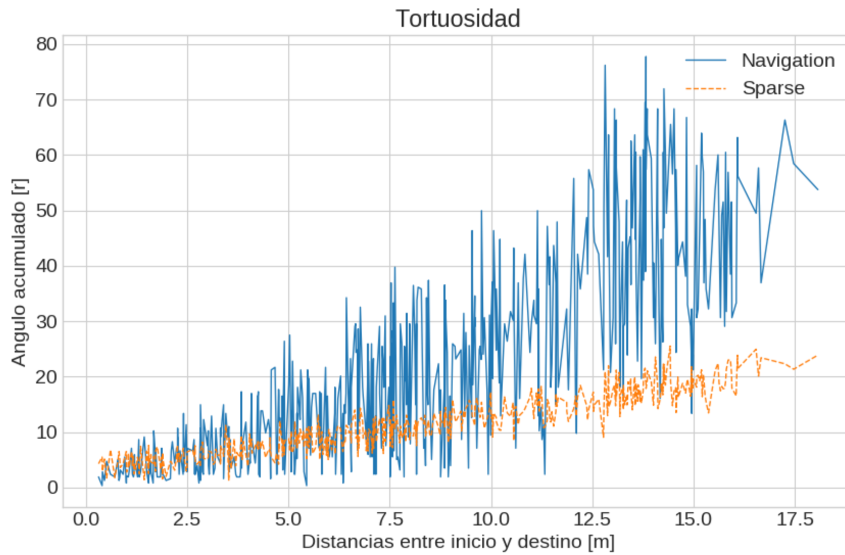
(a) Longitud de Trayectoria



(b) Dispersión de la Trayectoria

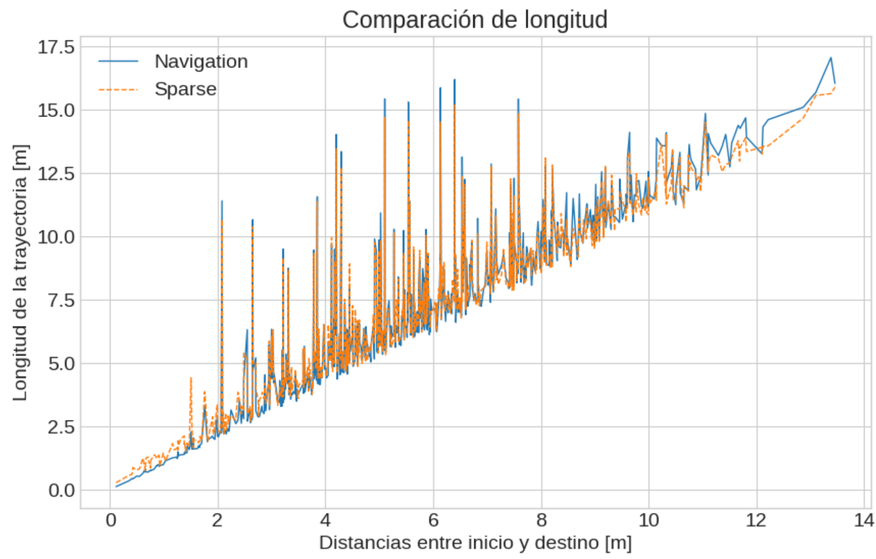


(c) Tiempo de ejecución

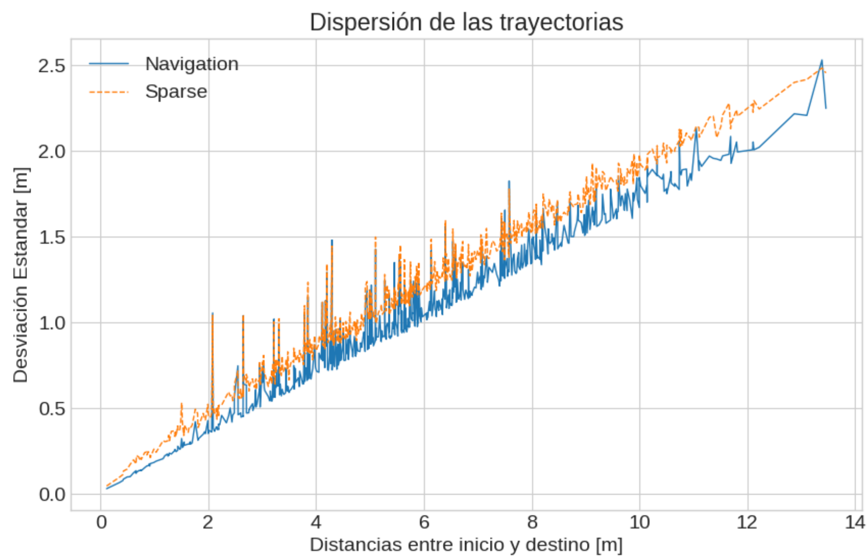


(d) Tortuosidad

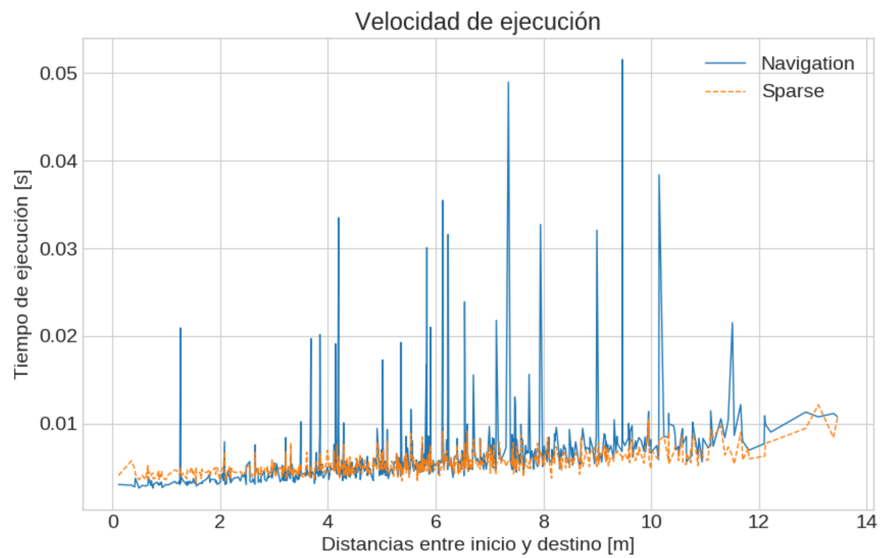
Figura 5.8: Resultados para mapeo por escenas laboratorio Procesamiento de señales



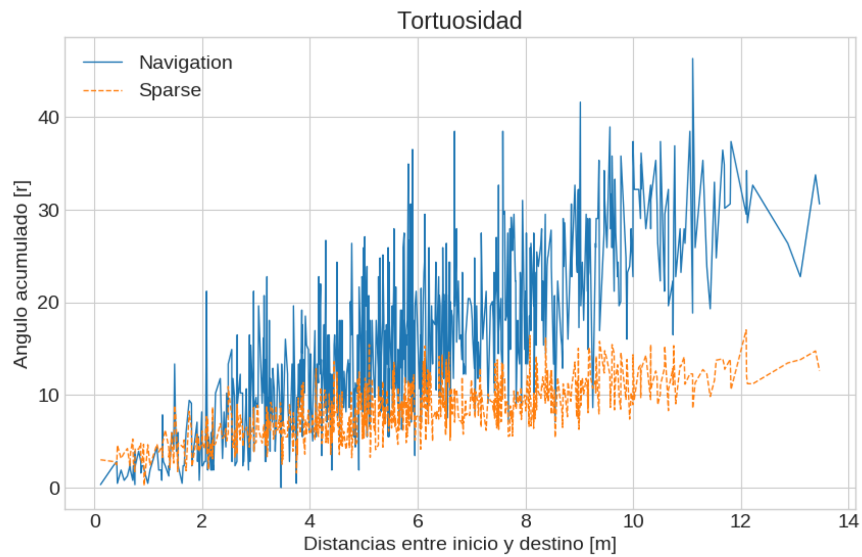
(a) Longitud de Trayectoria



(b) Dispersión de la Trayectoria

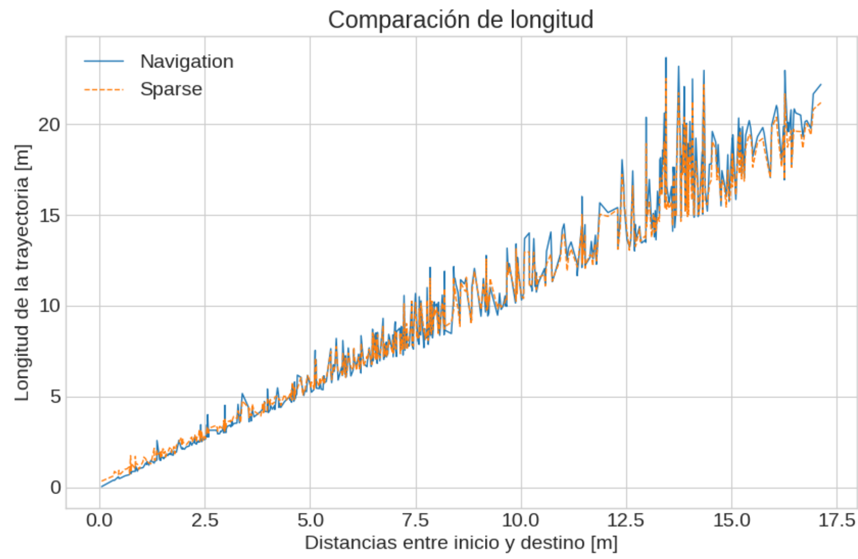


(c) Tiempo de ejecución

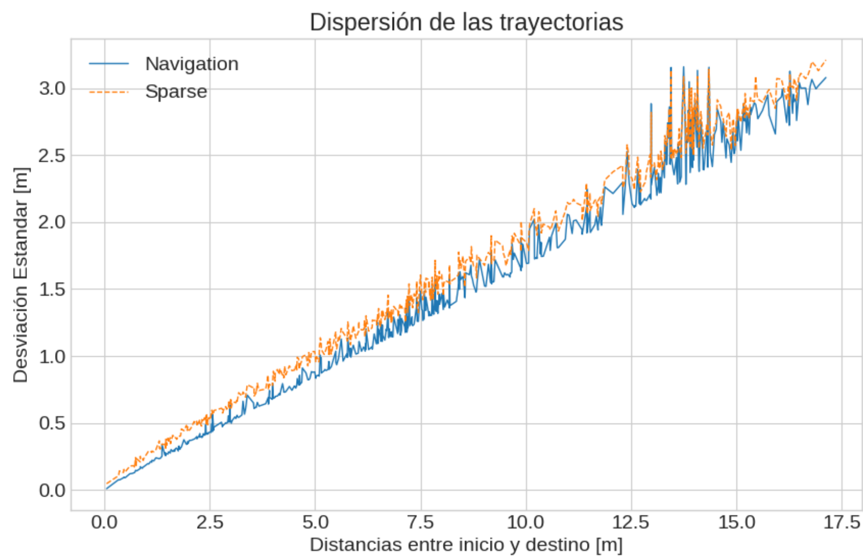


(d) Tortuosidad

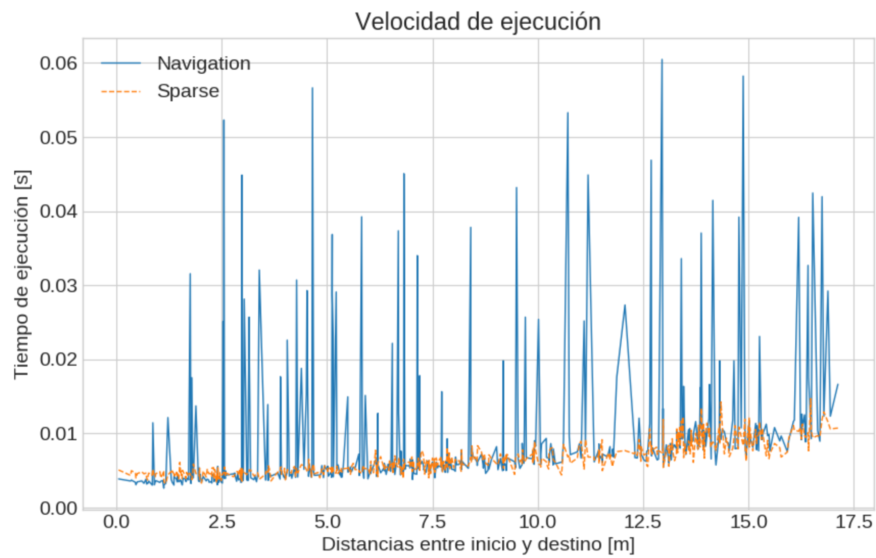
Figura 5.9: Resultados para mapeo por escenas apartamento HSR



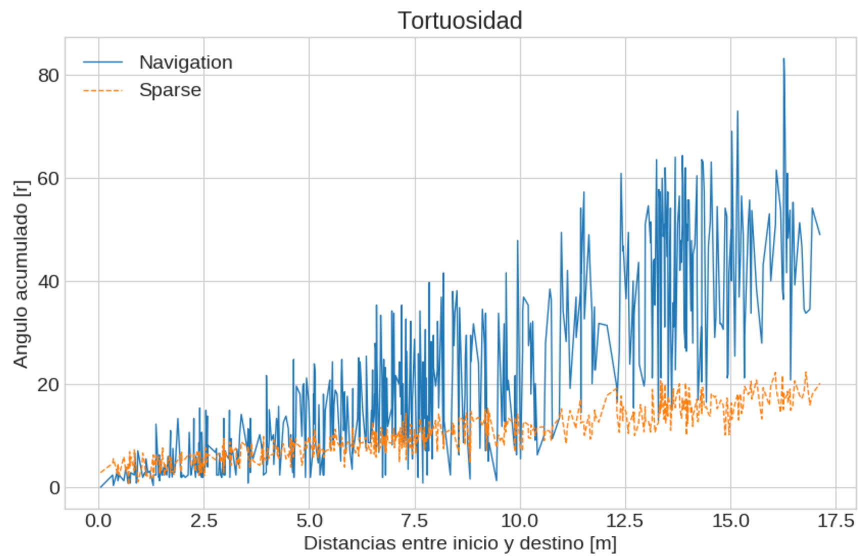
(a) Longitud de Trayectoria



(b) Dispersión de la Trayectoria

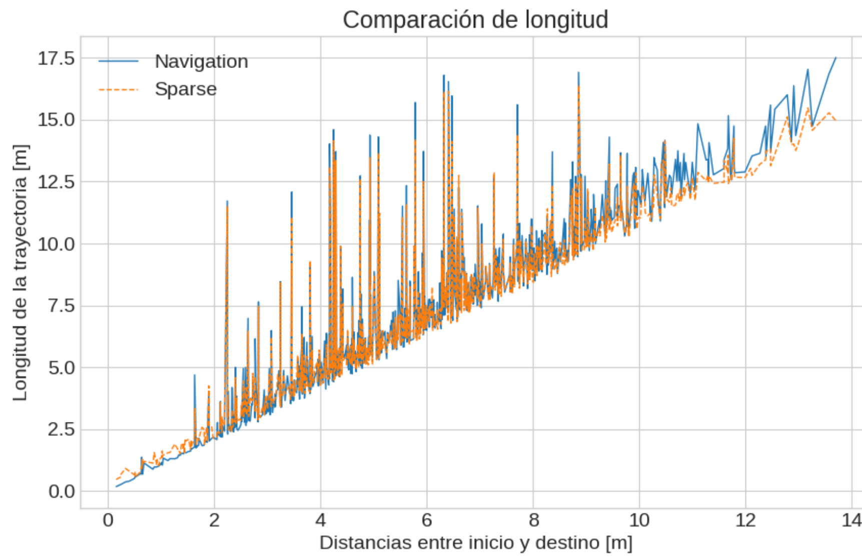


(c) Tiempo de ejecución

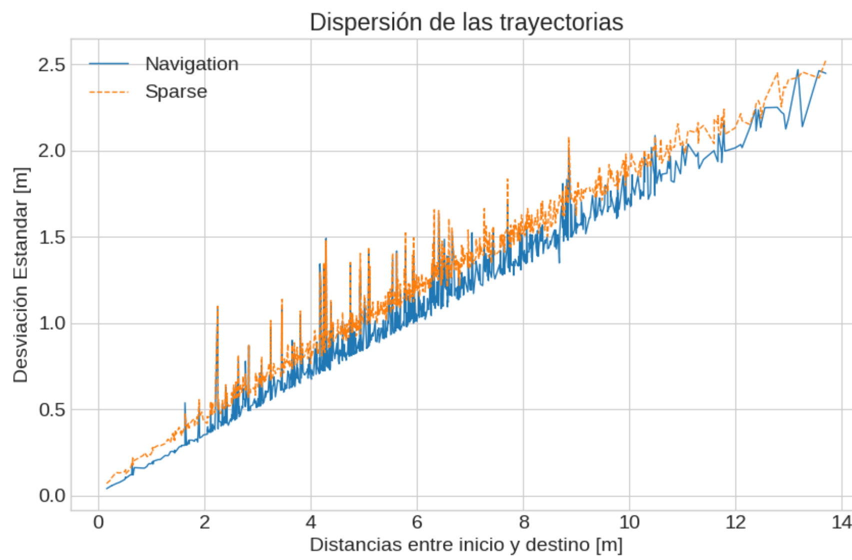


(d) Tortuosidad

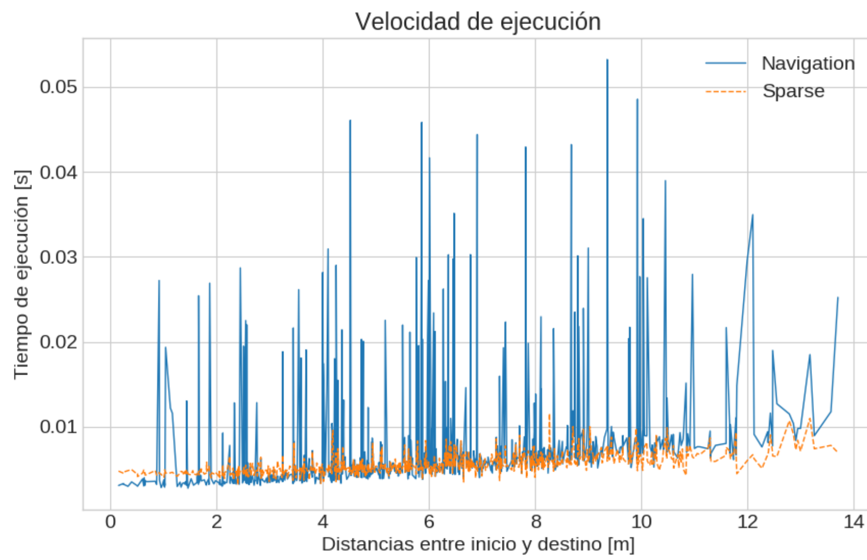
Figura 5.10: Resultados para mapeo por reconstrucción completa Laboratorio Procesamiento de señales



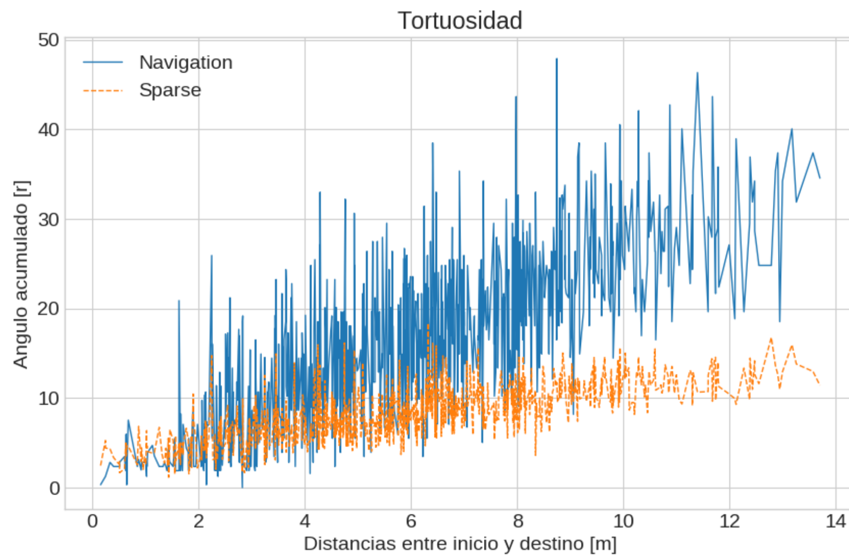
(a) Longitud de Trayectoria



(b) Dispersión de la Trayectoria



(c) Tiempo de ejecución



(d) Tortuosidad

Figura 5.11: Resultados para mapeo por recontrucción completa apartamento HSR

5.4. Compresión del espacio

Finalmente se puede comparar el espacio, en memoria, ocupado por el mapa topológico contra otro tipo de mapa, en este caso un Octomapa, donde ambos son representaciones tridimensionales. En la tabla 5.13 se muestra la memoria, en disco, utilizada por un Octomapa y el la del mapa topológico tanto en texto plano como en formato binario. Los octomapas son almacenados en dos tipos: Mapa binario que solo indica si una rama de árbol octal esta ocupada o no y arreglo de ocupación u octomapa completo donde se almacena la probabilidad de que sea una celda ocupada. En el caso de los mapas topológicos estos se almacenan en texto plano para facilitar su edición y lectura por parte del usuario pero una vez cargado en memoria cada código se puede representar con solo 16 bytes: tres flotantes de precisión simple para posición y un entero que indica el indice en el grafo.
Cabe resaltar que, como se observa en las tablas 5.5 y 5.6 varios ambientes son representados con el mismo número de centroides. Por lo que utilizan el mismo espacio en memoria a pesar de codificar un ambiente diferente. Una ventaja de contar con una reconstrucción completa de la planta es que se puede especificar de antemano el número deseado de centroides para efectuar la cuantización, algo que no es posible si el mapa se ensambla a partir de diferentes escenas.

Tabla 5.13: Tamaño de mapas: Octomap contra SparseMap

Uso de memoria mapas generados por escenas [kB]				
Ambiente	Octomap Binario	Octomap Completo	SparseMap: texto	SparseMap: Binario
Lab. de Biorobótica	77.4	1413.1	120.3	62.46
Lab. de Proc. de Señales	168.2	3871.7	117.5	59.9
Megaweb	248.3	5202.7	38.2	19.69
Apartamento HSR	121.7	2771.8	162.6	84.4
Uso de memoria mapas reconstrucción completa [kB]				
Ambiente	Octomap Binario	Octomap Completo	SparseMap: texto	SparseMap: Binario
Lab. de Biorobótica	77.4	1413.1	30.9	16.3
Lab. de Proc. de Señales	168.2	3871.7	61.5	32.7
Megaweb	248.3	5202.7	29.9	16.3
Apartamento HSR	121.7	2771.8	59.7	32.7

5.5. Análisis de los resultados

En esta sección se analizan los datos de la pruebas reportados en la sección anterior, además, se discuten las limitaciones del sistema propuesto.

5.5.1. Agrupamiento

Se puede observar de las tablas de la sección 5.2 y que en todos los casos los algoritmos que tuvieron la menor distorsión fueron Kmedias++ y LBG. El primero tuvo un desempeño mejor en casi todos los casos pero se aprecia fácilmente en las tabla 5.3 que la diferencia entre los resultados generados por cada algoritmo son muy pequeñas. De la misma tabla se observa que LBG tuvo menor distorsión en las reconstrucciones de pisos completos, pero la diferencia todavía es muy pequeña. Al final se concluye que de los cuatro algoritmos cualquiera genera resultados similares en cuanto a la distorsión media, solo cabe mencionar que para escenas pequeñas, como las de NYU y RGBD, K-medias++ es un poco mejor que LBG, sin embargo, para escenas grandes la relación se invierte.
En donde se aprecia un cambio fuerte es en el tiempo de ejecución, debido a su complejidad el algoritmo de Linde-Buzo-Gray siempre fue el más lento mientras que K-medias auxiliado por el GPU fue el más rápido. Cabe resaltar que LBG no se implementó en GPU debido a que es un algoritmo inherentemente serial y paralelizarlo resulta complicado y no se espera obtener mucha ganancia.
Algo a notar es que la implementación de K-medias++ fue un poco más lenta que la K-medias simple, el autor del algoritmo [26] menciona una fuerte reducción en tiempo de ejecución pero la implementación mostrada aquí ejecuta el algoritmo de Lloyd i veces indistintamente de la inicialización, ya que tanto K-medias y K-medias++ se ejecutaron con las mismas iteraciones es razonable esperar que el calculo extra de la inicialización retrasara al algoritmo pero aun así resultó en una distorsión menor, además en un ámbito real si se usará K-medias++ se correría con menos iteraciones en el algoritmo de Lloyd. Es importante resaltar que la implementación de K-medias++ es híbrida, la inicialización es secuencial por lo que se ejecuta en el CPU y se utilizan los centroides generados para inicializar el algoritmo de K-medias en GPU.
En términos prácticos se puede lograr un equilibrio entre velocidad y calidad escogiendo K-medias++ esto también disminuye drásticamente la posibilidad de tener una mala inicialización de nuestros centroides, por lo que se recomienda su uso si se cuenta con GPU y se planea ensamblar los mapas dispersos a partir de diferentes tomas en línea. En caso de no contar con GPU utilizar la implementación serial de

K-medias resulta suficiente. Debido a todo lo anterior en pruebas descritas en este documento se uso siempre K-medias++.
Por último es importante mencionar que no se logra llegar a tener una implementación en tiempo real, la mayoría del hardware comercial entrega información a 30 hertz de 640x480 pixel es, lo que nos otorga un tiempo de respuesta de 33 milisegundos, incluso utilizando nubes de puntos pequeñas con tan solo 16 centroides llegamos solo a un tiempo de procesamiento de alrededor de 50 milisegundos como se aprecia en la tabla 5.4. En este caso el sistema de mapeo aquí propuesto no requiere cuantizar en tiempo real pero en el caso de necesitar procesamiento a mayor velocidad la nube de puntos de entrada puede ser submuestreada esto reducirá bastante el tiempo de ejecución.

5.6. Planeación de trayectorias

En el caso de la planeación se observa que en cuanto a longitud Sparse-Mapper genera trayectorias de tamaño similar al Navigation Stack. A su vez se aprecia que en el caso de que el inicio y el destino se encuentre muy cercanos la mayor resolución de las celdas de ocupación permiten generar trayectorias más cortas, asimismo cuando las terminales del plan se encontraban muy alejadas, a varios metros, la compresión otorgada por la cuantización redundaba en trayectorias más cortas.
Como era de esperar al ser un grafo mucho más pequeño los planes obtenidos a partir de Sparse-Mapper tiene muy pocos nodos, esto tiene la ventaja de una planeación más rápida, en teoría y la creación de trayectorias más simples, esto permite, en caso de ser necesario, post-procesar las trayectorias obtenidas para suavizarlas o simplificarlas aun más.
Donde se tiene una mejora importanto es en el tiempo de ejecución. Debido a la naturaleza más pequeña de un mapa disperso el plan se crea más rápido ya que los nodos a expandir son pocos, asimismo la cuantización y la malla de voxeles propuesta reduce fuertemente la cantidad de colisiones en 3D que deben ser calculadas para validar trayectorias, estas son operaciones caras y cabe resaltar que el Navigation Stack no las realiza por ser un mapa plano.
Inicialmente se esperaba que debido a la naturaleza dispersa del grafo la calidad de las trayectorias generadas fuera menor, es decir que forzaran al agente que las siguiera a zigzaguear demasiado, pero a pesar de la fuerte compresión del ambiente. Las rutas generadas tienen menos rotación que aquellas propuestas por el Navigation Stack. Esto se debe muy probablemente a que el mapa topológico es métrico mientras que las celdas de ocupación son una malla discretizada.
Por otro lado a pesar de generar trayectorias con poca rotación estas tienden a ser

más dispersas que su contraparte en celdas de ocupación, Lo que lleva a un mayor movimiento traslacional por parte del agente, sin embargo, como se aprecia en las gráficas de dispersión la diferencia no es grande. En estos últimos puntos es importante mencionar que el controlador de movimiento, elemento que no se desarrolla en este documento, es determinante cuando se ejecutan las trayectorias, es decir, incluso si se tiene una trayectoria muy buena y suave si el controlador es malo el desplazamiento del robot será pobre.
Por último no se observa mucha diferencia entre usar una reconstrucción completa del medio a usar varias escenas ensambladas, mientras se tenga un número similar de códigos. Incluso en algunos casos se tuvo peor desempeño debido a que cuando se contó con la reconstrucción completa se terminó con un alfabeto un poco más pequeño lo que lleva a no tener suficientes elementos para generar trayectorias de calidad. Cabe aclarar que al usar un ensamble de diferentes escenas no se tiene control del número final del centroides en el mapa, caso contrario a usar toda la reconstrucción.
En términos cualitativos las trayectorias generadas por Sparse-Map tienden a navegar más cerca de los obstáculos pero sin colisionar. Esto debido al costo y heuristica propuesta favorecen trayectorias cortas, el Navigation Stack, por su parte penaliza acercarse a los obstáculos al usar mapas de costo. Esto lleva a que tome trayectorias lo más alejado posible de obstáculos.

5.7. Compresión

Respecto a la reducción de memoria se logró una reducción significativa de recursos de almacenamiento comparado con el octomapa correspondiente. Tanto si se compara con los octomapas binarios o la representación completa probabilista. Asimismo se evita la discretización *a priori* del espacio manteniendo una alta resolución en la posición de los centroides.

5.8. Limitaciones

Como era de esperase comprimir un mapa lleva a situaciones donde se pierde resolución: En segmentos como puertas o pasillos estrechos la metodología aquí descrita tiene problemas para generar los suficientes centroides navegables para generar un mapa adecuado. Esto lleva a declarar áreas navegables como ocupadas y a ejecutar trayectorias peligrosas cerca de puertas muy estrechas, esto se observa en la figura 5.12. Usando mapeo incremental o aumentando el número de códigos en el mapa esto se puede remediar.

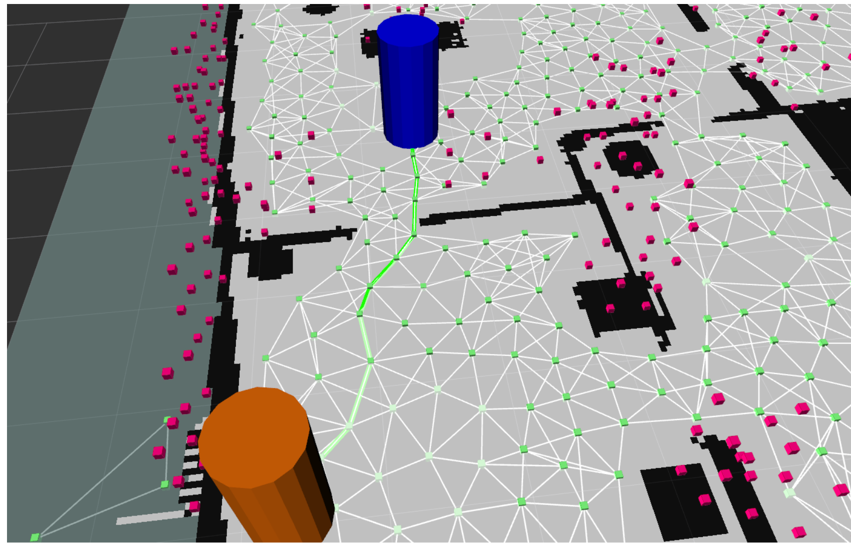


Figura 5.12: En la figura una trayectoria sobrepuesta a las celdas de ocupación, debido a la posición de los centroides una colisión es probable

Otra limitación son los obstáculos dinámicos, en este documento se implementó solo un planeador global, los obstáculos dinámicos y no mapeados son generalmente resueltos por un planeador local que utiliza solo información de la toma actual de los sensores. En el caso de celdas de ocupación en caso de detectar un obstáculo no mapeado se puede actualizar temporalmente el mapa y planear de nuevo, lo cual es sencillo debido a la naturaleza matricial de esta representación. En el caso de los mapas topológicos esto requiere entrar al grafo y desconectar, temporalmente, todos los nodos libres que se encuentren en colisión con algún obstáculo, volver a planear y finalmente restaurar el grafo. Esto requiere varias corridas a algoritmos de vecinos más cercanos y a detectores de colisiones 3D lo que es complicado y computacionalmente caro. Esto se puede subsanar mediante el uso de búsquedas rápidas como Arboles K-d y auxiliarse de una lista de voxeles como la propuesta en el algoritmo 12.

Finalmente una ventaja de las celdas de ocupación es que es fácil de editar manualmente para eliminar errores mediante cualquier programa de edición de imágenes. Sin embargo los mapas topológicos son más difíciles de editar debido a su representación no estructurada y a que no están en algún formato de fácil edición. Con ayuda de RVIZ un visualizador desarrollado para ROS es posible hacer ediciones básicas, pero es mucho más complicado que operaciones similares en celdas de ocupación, parte de esto es debido a que es una representación tridimensional.

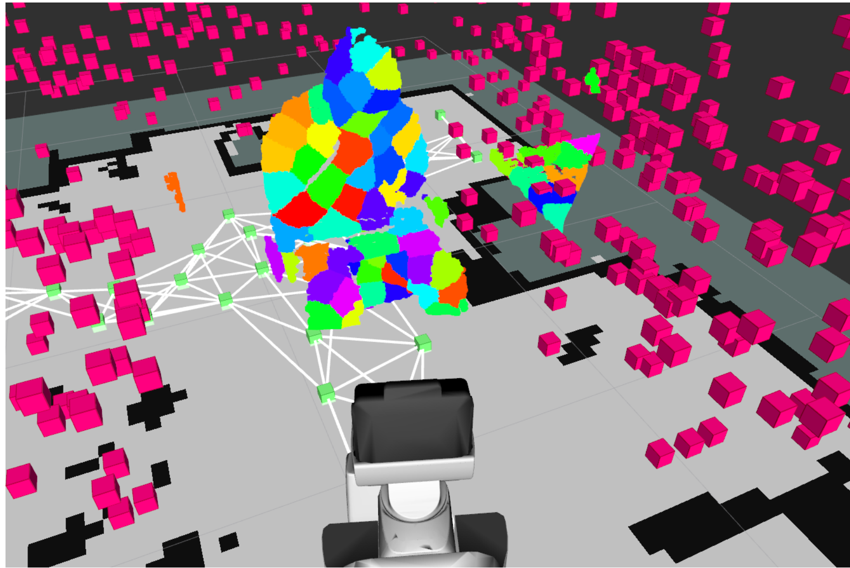


Figura 5.13: El sistema propuesto no es capaz de planear alrededor de obstáculos dinámicos, pero un sistema de cuantización similar puede ser propuesto para este fin. En la figura un obstáculo cuantizado no mapeado enfrente al robot.

Capítulo 6

Conclusiones y trabajo a futuro

6.1. Conclusiones

En este trabajo se demostró que es posible generar trayectorias validas para navegación robótica mediante mapas generados a partir de la cuantización de nubes de puntos a partir de algoritmos de agrupamiento o clustering. Se analizaron varias alternativas de algoritmos y su desempeño en nubes de puntos tanto públicas como generadas por cuenta propia con el fin de determinar cual es la mejor opción para cuantizar nubes de puntos tridimensionales. De acuerdo a los resultados de la sección 5.2 podemos afirmar que incluso los algoritmos más sencillos son capaces de generar una partición adecuada para un mapa siendo el factor que distingue a los algoritmos el tiempo de ejecución.
Asimismo se evaluó el desempeño del planeador en contra del Navigation Stack de ROS y se encuentra que es similar o mejor en algunos aspectos, por ejemplo tenemos trayectorias con menor tortuosidad como se aprecia en la tabla 5.11 lo que lleva a generar caminos más suaves para el agente, pero se impone mayor movimiento traslacional. Esto debido a la naturaleza dispersa y métrica del mapa topológico. Además, como se observa en la tabla 5.9 esto se logra en casi el mismo o menor tiempo que el Navigation Stack. Finalmente la longitud de estas trayectorias son muy similares a las generadas por el Stack de ROS. En el caso de que el inicio y destino estén muy separados la trayectorias generadas por Sparse-Mapper son más cortas, pero en el caso contrario, que las terminales esten muy cerca el mapa denso tiene mejor desempeño como se observa en la gráfica 5.8. En promedio nuestro sistema genero trayectorias 2 milisegundos más rápido que el Navigation Stack, estas trayectorias fueron 22 centímetros más cortas y con 9.67 radianes menos de tortuosidad, pero en promedio son 9.9 centímetros más dispersas. Esto nos lleva a que incluso a pesar

de la fuerte cuantización y naturaleza dispersa del mapa éste es todavía capaz de generar trayectorias comparables a sistemas densos.
Se prueba además que no hay gran diferencia entre utilizar una reconstrucción completa del ambiente o ensamblar un mapa topológico a partir de multiples escenas. En promedio las trayectorias generadas a partir de mapas completos fueron, en promedio, 9 centimetros más cortas, corrieron 0.12 milisegundos más rápido mientras que su tortuosidad fue de 1.4 centimetros menor y con una dispersión de 0.53 radianes menos. Ligeramente mejores pero no por mucho. Se considera que este sistema es capaz de representar mapas muy grandes que resultan prohibitivos para operar o almacenar en un formato denso o semidenso, como octomapas, celdas de ocupación o nubes de puntos. Como se aprecia en la tabla 5.13 se tiene un fuerte ahorro en memoria comparado con Octomapas, no se introducen artefactos por la voxelización y solo estamos limitados a la precisión de la máquina.

6.2. Trabajo futuro

El sistema de cuantización puede ser utilizado para otros fines, el problema de registrar nubes de puntos diferentes es muy complicado, y se resuelve generalmente mediante el algoritmo del punto más cercano iterativo: ICP [39] por sus siglas en inglés donde se busca una transformación rígida entre 2 nubes de puntos asumiendo que el punto más cercano en ambas nubes corresponden. Encontrar la transformación dadas las correspondencias es un problema relativamente sencillo pero encontrar el vecino más cercano para todos los puntos es un problema extenso y computacionalmente caro. Si ambas nubes se cuantizan de antemano se puede hacer la búsqueda en el espacio de códigos y no de puntos disminuyendo fuertemente el tamaño de la búsqueda manteniendo una coherencia espacial. Esto puede generalizarse a un sistema de odometría por ICP.
Por otro lado la representación gráfica se puede utilizar para reconocer objetos. Para esto cuantiza la nube de puntos de un objeto y se arma su grafo, este puede ser comparado contra una base de datos y devolver el grafo más cercano. Para mejorar el desempeño se pueden usar nubes de puntos con color y cuantizar de forma similar a [24] esto generaría centroides que describen tanto información espacial como de color. Bajo la misma línea se puede considerar el uso de este algoritmo para comprimir representaciones tridimensionales de objetos cuando se requiera manejar un objeto tridimensional que ocupe poca memoria, por ejemplo transmisión de nubes de puntos a través de redes.
En cuanto al mapeo, en este documento se separó el espacio solo en función de la altura del centroide pero la metodología se puede generalizar y añadir otra propie-

dad para separar el espacio, por ejemplo a partir de todos los puntos que pertenecen al mismo centroide se puede calcular la normal y añadir esta información al código, así se puede añadir la restricción que los nodos navegables sean solo aquellos cuya normal sea paralela al eje Z del mapa.
Por otro lado tenemos la opción de añadir más información al grafo para cambiar el comportamiento del agente. Por ejemplo se puede añadir un costo a cada nodo que penalice rutas muy cercanas a obstáculos. A su vez y como se propone en la literatura [32] se puede calcular la transversabilidad del terreno a partir del gradiente de alturas en centroides vecinos, esto nos llevaría a evitar terreno muy irregular, algo útil fuera de ambientes interiores.
En términos de desempeño las rutinas de agrupamiento todavía pueden ser mejoradas utilizando algoritmos de reducción paralela más completos, asimismo se pueden añadir más algoritmos de cuantización y evaluar su desempeño. También es posible reducir el tiempo de serialización propio de usar mensajes en ROS mediante el uso de Nodelets, estos no serializan el mensaje y comparten la información mediante hilos internos al proceso.
Finalmente se puede generalizar esto a un sistema de SLAM por grafos completo, utilizando el cuantizador como observación se podría disminuir fuertemente el tiempo y espacio de búsqueda para encontrar cerraduras de lazo. Sistemas actuales usan descriptores visuales [22] de muy alta dimensionalidad de alrededor de 64 dimensiones pero un código en el espacio tridimensional tiene 3, 4 en caso de utilizar intensidad de escala de gris y 6 para si se añade un espacio de color.

Bibliografía

[1] B. Siciliano y O. Khatib, *Springer handbook of robotics*. Springer, 2016.

[2] Thrun Sebastian, Burgard Wolfram, Fox Dieter, *Probabilistic Robotics*. MIT Press, 2005.

[3] C. Stachniss, *Lectures on mobile robots: Grid maps*, 2012. dirección: http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/.

[4] O. Cameron. (2017). An introduction to lidar: The key self-driving car sensor, dirección: https://news.voyage.auto/an-introduction-to-lidar-the-key-self-driving-car-sensor-a7e405590cff?gi=584783c4aa4f (visitado 03-07-2018).

[5] V. LiDAR. (2018). Lidar 101, dirección: http://velodynelidar.com/lidar-101.html (visitado 11-05-2018).

[6] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling y S. Thrun, «Towards fully autonomous driving: Systems and algorithms», en *2011 IEEE Intelligent Vehicles Symposium (IV)*, jun. de 2011, págs. 163-168. DOI: 10.1109/IVS.2011.5940562.

[7] L. Larry. (mayo de 2014). Time-of-flight camera– an introduction, dirección: http://www.ti.com/lit/wp/sloa190b/sloa190b.pdf (visitado 11-05-2018).

[8] R. B. Rusu y S. Cousins, «3d is here: Point cloud library (pcl)», en *2011 IEEE International Conference on Robotics and Automation*, mayo de 2011, págs. 1-4. DOI: 10.1109/ICRA.2011.5980567.

[9] W. Burgard. (2017). Techniques for 3d mapping, dirección: http://ais.informatik.uni-freiburg.de/teaching/ss17/robotics/ (visitado 04-07-2018).

[10] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss y W. Burgard, «Octomap: An efficient probabilistic 3D mapping framework based on octrees», *Autonomous Robots*, 2013, Software available at `http://octomap.github.com`. DOI: `10.1007/s10514-012-9321-0`. dirección: `http://octomap.github.com`.

[11] S. Chitta, I. Sucan y S. Cousins, «Moveit!», vol. 19, págs. 18-19, mar. de 2012.

[12] M. Montemerlo, S. Thrun, D. Koller y B. Wegbreit, «Fastslam: A factored solution to the simultaneous localization and mapping problem», en *In Proceedings of the AAAI National Conference on Artificial Intelligence*, AAAI, 2002, págs. 593-598.

[13] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard, «Improved techniques for grid mapping with rao-blackwellized particle filters», 2007.

[14] S. Kohlbrecher, J. Meyer, O. von Stryk y U. Klingauf, «A flexible and scalable slam system with full 3d motion estimation», en *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, IEEE, nov. de 2011.

[15] W. Hess, D. Kohler, H. Rapp y D. Andor, «Real-time loop closure in 2d lidar slam», en *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, págs. 1271-1278.

[16] D. Fox, «Kld-sampling: Adaptive particle filters», en *Advances in neural information processing systems*, 2002, págs. 713-720.

[17] J. Engel, T. Schöps y D. Cremers, «Lsd-slam: Large-scale direct monocular SLAM», en *European Conference on Computer Vision (ECCV)*, sep. de 2014.

[18] G. Grisetti, R. Kummerle, C. Stachniss y W. Burgard, «A tutorial on graph-based slam», *IEEE Intelligent Transportation Systems Magazine*, vol. 2, n.º 4, págs. 31-43, 2010, ISSN: 1939-1390. DOI: `10.1109/MITS.2010.939925`.

[19] F. Endres, J. Hess, J. Sturm, D. Cremers y W. Burgard, «3-d mapping with an rgb-d camera», *IEEE Transactions on Robotics*, vol. 30, n.º 1, págs. 177-187, feb. de 2014, ISSN: 1552-3098. DOI: `10.1109/TRO.2013.2279412`.

[20] Mur-Artal, Raúl, Montiel, J. M. M. and Tardós, Juan D., «Orb-slam: A versatile and accurate monocular SLAM system», *IEEE Transactions on Robotics*, vol. 31, n.º 5, págs. 1147-1163, 2015. DOI: `10.1109/TRO.2015.2463671`.

[21] R. Mur-Artal y J. D. Tardós, «Orb-slam2: An open-source SLAM system for monocular, stereo and RGB-D cameras», *IEEE Transactions on Robotics*, vol. 33, n.º 5, págs. 1255-1262, 2017. DOI: `10.1109/TRO.2017.2705103`.

[22] M. Labbé y F. Michaud, «Appearance-based loop closure detection for online large-scale and long-term operation», *IEEE Transactions on Robotics*, vol. 29, n.º 3, págs. 734-745, jun. de 2013, ISSN: 1552-3098. DOI: 10.1109/TRO.2013.2242375.

[23] ——, «Online global loop closure detection for large-scale multi-session graph-based slam», en *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, sep. de 2014, págs. 2661-2666. DOI: 10.1109/IROS.2014.6942926.

[24] J. Papon, A. Abramov, M. Schoeler y F. Worgotter, «Voxel cloud connectivity segmentation-supervoxels for point clouds», en *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013, págs. 2027-2034.

[25] Y. Linde, A. Buzo y R. Gray, «An algorithm for vector quantizer design», *IEEE Transactions on Communications*, vol. 28, n.º 1, págs. 84-95, ene. de 1980, ISSN: 0090-6778. DOI: 10.1109/TCOM.1980.1094577.

[26] D. Arthur y S. Vassilvitskii, «K-means++: The advantages of careful seeding», en *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial y Applied Mathematics, 2007, págs. 1027-1035.

[27] V. Adamchik, *Graph theory*, 2005.

[28] S. Russell y P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[29] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler y A. Y. Ng, «Ros: An open-source robot operating system», en *ICRA Workshop on Open Source Software*, 2009.

[30] S. Cook, *CUDA programming: A developer's guide to parallel computing with GPUs*. Newnes, 2012.

[31] A. Hornung, M. Phillips, E. G. Jones, M. Bennewitz, M. Likhachev y S. Chitta, «Navigation in three-dimensional cluttered environments for mobile manipulation», en *2012 IEEE International Conference on Robotics and Automation*, mayo de 2012, págs. 423-429. DOI: 10.1109/ICRA.2012.6225029.

[32] J. Stückler, M. Schwarz y S. Behnke, «Mobile manipulation, tool use, and intuitive interaction for cognitive service robot cosero», *Frontiers in Robotics and AI*, vol. 3, pág. 58, 2016, ISSN: 2296-9144. DOI: 10.3389/frobt.2016.00058. dirección: https://www.frontiersin.org/article/10.3389/frobt.2016.00058.

[33] J. Sturm, N. Engelhard, F. Endres, W. Burgard y D. Cremers, «A benchmark for the evaluation of rgb-d slam systems», en *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, oct. de 2012.

[34] P. K. Nathan Silberman Derek Hoiem y R. Fergus, «Indoor segmentation and support inference from rgbd images», en *ECCV*, 2012.

[35] I. Armeni, A. Sax, A. R. Zamir y S. Savarese, «Joint 2d-3d-semantic data for indoor scene understanding», *ArXiv e-prints*, feb. de 2017. arXiv: `1702.01105 [cs.CV]`.

[36] J. Park, Q.-Y. Zhou y V. Koltun, «Colored point cloud registration revisited», en *ICCV*, 2017.

[37] T. Yamamoto, K. Terada, A. Ochiai, F. Saito, Y. Asahara y K. Murase, «Development of human support robot as the research platform of a domestic mobile manipulator», *ROBOMECH Journal*, vol. 6, n.º 1, pág. 4, abr. de 2019, ISSN: 2197-4225. DOI: `10.1186/s40648-019-0132-3`. dirección: `https://doi.org/10.1186/s40648-019-0132-3`.

[38] N. P. Koenig y A. Howard, «Design and use paradigms for gazebo, an open-source multi-robot simulator.», en *IROS*, Citeseer, vol. 4, 2004, págs. 2149-2154.

[39] W. Burgard. (2017). Iterative closest point algorithm, dirección: `http://ais.informatik.uni-freiburg.de/teaching/ss11/robotics/` (visitado 04-07-2018).

Apéndice A

Apendices

A.1. Archivos

Como se mencionó en la sección 5.2. Se utilizaron nubes de puntos de varios juegos de datos. Pero se utilizó un nombre corto para hacer referencia a cada archivo individual por brevedad. Aquí se provee la relación entre el nombre utilzado en el documento y el nombre utilizado internamente en cada juego de datos.

Tabla A.1: Nombre de archivo para cada escena o archivo

Escena	Dataset	Nombre del archivo
Recamara	Redwood	bedrom.ply
Pasillo Stanford	Stanford 2D-3D-S	camera _c520a9e9cb4947b7b728a0fbeab19877 _office_39_frame_equirectangular _domain_global_xyz.exr
Área 5a Stanford	Stanford 2D-3D-S	semantic.obj
Biorobótica	Propio	BioroboticaIII.pcd
WRS2018	Propio	wrs2018.pcd
NYU-60	NYU-v2	d-1295527096.208352-1895648860.pgm
NYU-30	NYU-v2	d-1295527104.022905-2364153130.pgm
NYU-28	NYU-v2	d-1295527253.447308-2739868628.pgm
NYU-15	NYU-v2	d-1295527107.579828-2578383715.pgm
NYU-13	NYU-v2	d-1295527265.699710-3474659513.pgm
RGBD-85	RGBD-Pioneer	1311878241.979085.pgm
RGBD-53	RGBD-Pioneer	1311878277.418453.pgm
RGBD-44	RGBD-Pioneer	1311878323.865744.pgm
RGBD-43	RGBD-Pioneer	1311878212.349243.pgm
RGBD-27	RGBD-Pioneer	1311878203.301427.pgm

A.2. Glosario

- **SLAM**: *Simultaneous Localization and Mapping* o Localización y Mapeo Simultaneo, problema más general en el mapeo robótico, que consiste en estimar un mapa a la vez que el agente se localiza sobre este.
- **Localización**: En la robótica, problema donde el objetivo consiste en localizar a un agente sobre un mapa dado.
- **Mapeo dado poses conocidas**: En la robótica, problema en el cual dada una serie de poses y mediciones se debe estimar un mapa.
- **Grafo**: Ente matemático $G = (V, E)$ que consiste en un conjunto de vértices o nodos, V y un conjunto de arcos o aristas E.
- **CUDA**: Lenguaje para programación de GPU de la marca NVIDIA.
- **GPU**: Siglas en inglés de *Graphical Processing Unit*. Procesador de arquitectura de instrucción sencilla y múltiples datos, capaz de operar sobre datos en paralelo. Usado principalmente para procesamiento de gráficos de computadora.
- **Cuantización Vectorial** Mapeo de $x = \{x_1, x_2, ..., x_N\}$ a $C = \{y_1, y_2, ..., y_k\}$ donde $n > k$ y $x_i, y_j \in \mathbb{R}^3$. Permite agrupar y reducir datos vectoriales.
- **Nube de puntos** Lista ordenada o no de puntos o vectores que representan un objeto en el espacio.
- **PCL** Point Cloud Library: Biblioteca de C++ enfocada en el procesamiento de nubes de puntos.
- **Octomapa** Representación tridimensional densa. Una extensión de celdas de ocupación a $\mathbb{R}^3$ pero basada en árboles octales lo que permite una representación más compacta.