CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS
AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL
UNIDAD ZACATENCO

Maestría en Ciencias en Sistemas Autónomos de Navegación Aérea y Submarina

ALGORITMOS DE LOCALIZACIÓN SIN GPS

T E S I S

Que presenta

ING. JOSÉ ANTONIO VILLAVICENCIO CASTILLO

Para obtener el grado de

MAESTRO EN CIENCIAS EN SISTEMAS AUTÓNOMOS DE NAVEGACIÓN AÉREA Y SUBMARINA

Directores de la Tesis :

Dr. Rogelio Lozano Leal

Dr. Iván González Hernández

Ciudad de México Febrero 2020

" *A mis padres, Gina y Tony*
y a mi hermanito Edgar,
por su apoyo y amor incondicional"

Contenido

Lista de Figuras

Resumen

Una de las principales tareas que debe de llevar a cabo un robot, si quiere ser autónomo, es la de navegar libremente por el espacio sin un conocimiento a priori de éste. El problema de SLAM, localización y mapeo simultáneos, consiste en ir localizando al robot a la vez que va construyendo un mapa de su entorno para referencia futura. En principio este problema parece sencillo, sin embargo tiene involucrados muchos retos tecnológicos donde se tienen que resolver varios problemas de manera simultánea. Los 4 bloques básicos para asegurar el éxito en una tarea de navegación son la percepción, localización, cognición y control. Dentro de la percepción, es necesario elegir un sensor adecuado que provea de información suficiente a los algoritmos del robot. Esta tesis utiliza una cámara RGB-D como sensor para realizar las tareas de localización y mapeo simultáneo. Además, explota la información de rango para construir un modelo en 3D del ambiente que sea útil en tareas de navegación autónoma y exploración.

Abstract

One of the main tasks a robot should perform, if it wants to be truly autonomous, is that of navigating through the environment without any previous knowledge. SLAM (Simultaneous Localization and Mapping) consists of localize and, at the same time, build a map of the environment for future reference. At first this seems an easy task, however, it involves many technological challenges. The 4 building blocks for autonomous navigation are perception, localization, cognition and control. Among perception includes the selection of a suitable sensor to gather enough information. This thesis uses an RGB-D camera as a primary sensor to achieve simultaneous localization and mapping. Besides, it uses all this measurements to build a 3D model of the environment in a way that it's useful for autonomous navigation and exploration tasks.

Capítulo 1

Introducción

I Motivación

Desde que fue usada por primera vez hace 100 años en la obra de Karel Čapek, *Rossumovi Univerzální Roboti*, la palabra *robot* ha reflejado el esfuerzo humano por construir máquinas que realicen trabajos que, por alguna razón, las personas no queremos o podemos realizar. Por otro lado, con los avances tecnológicos el uso de estas máquinas ya no se ha limitado a tareas industriales o peligrosas sino que se ha extendido a usos recreativos o educativos, estando al alcance de la población en general.

La robótica ha logrado avances impresionantes en los últimos años. Estando en una posición fija, un brazo robótico puede manipular herramientas con una precisión y velocidad que un ser humano no puede replicar, con la ventaja adicional de que puede realizar dicha tarea de manera repetitiva por largos periodos de tiempo. Tenemos robots capaces de asistir en una cirugía, como el robot DaVinci (Figura 1.1) o brazos robóticos, como los de la marca Fanuc (Figura 1.2) con la posibilidad de intercambiar las herramientas de su efector final.

A pesar de todas sus bondades, este tipo de robots tienen una desventaja fundamental: falta de movilidad. Están restringidos a una posición fija en el espacio con un

Figura 1.1: Robot DaVinci

espacio de trabajo limitado a las características físicas de sus eslabones. En contraste, un robot móvil tiene la posibilidad de moverse libremente en su entorno, aplicando sus talentos donde mejor convenga. Ejemplos de estos últimos son los robots móviles que utiliza Amazon en sus almacenes (Figura 1.3), vehículos aéreos no tripulados (UAV) (Figura) o vehículos submarinos autónomos (AUV) (Figura). El poder trasladarse de un punto a otro abre nuevas posibilidades en el mundo de la robótica.

La *navegación* es una de las competencias más desafiantes requeridas en un robot móvil. Para navegar con éxito de un punto a otro, un robot móvil requiere de estos 4 pilares:

- Percepción
- Localización
- Cognición
- Control de movimiento

Figura 1.2: Robot Fanuc M-2000iA

Figura 1.3: Robots móviles de Amazon

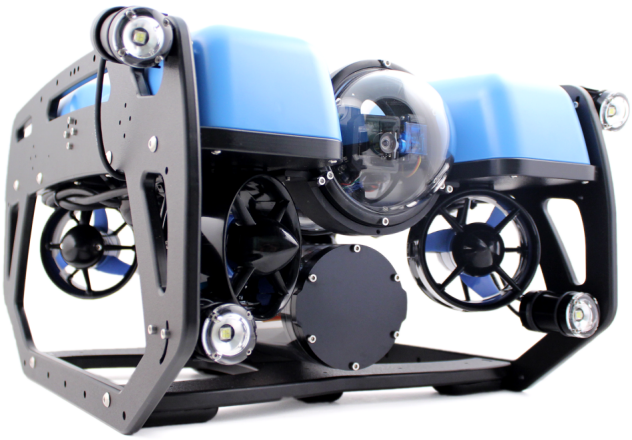


Figura 1.4: BlueROV2

Figura 1.5: Intel Aero

De estos 4 pilares, la localización de un robot ha recibido un interés especial y, como resultado, se han tenido importantes avances. Hay muchas formas de contestar la pregunta "¿dónde estoy?", siendo la más conocida el uso de un GPS (*Global Positioning System*) . Si pudiéramos poner un GPS preciso en cada robot, muchos de los problemas que supone localizarlo serían resueltos. No obstante, no siempre es factible colocar un GPS ya que esta tecnología tiene importantes limitantes como baja resolución y que solo funciona en exteriores. ¿Qué pasa si queremos localizar un robot móvil en un espacio donde no se tiene señal de un GPS? ¿Y si queremos localizar este mismo robot en una posición relativa a un objeto cercano? ¿Si tuviéramos que localizar un robot cuya dimensión es menor que el rango de precisión de los sensores GPS disponibles en el mercado? ¿Si quisiéramos que un robot navegara en un espacio dinámico del cual no se tiene ningún mapa?

La motivación de esta tesis es estudiar algoritmos de localización no convencionales que respondan a todas estas preguntas.

II Objetivos

El objetivo de esta tesis es estudiar y proponer una solución al problema de SLAM (*Simultaneous Localization and Mapping*). El problema de *localización* consiste en es-

timar la posición y trayectoria de un robot, dado un mapa de su entorno. El problema de *mapeo* consiste en la construcción de un mapa del entorno dada la trayectoria *verdadera* del robot. El objetivo del SLAM es recuperar la posición y trayectorias de un robot a la vez que se construye un mapa de su entorno, conociendo únicamente los datos recopilados por los sensores propioceptivos y exterioceptivos del robot. Estos datos corresponden típicamente a mediciones de odometría y características (e.g. esquinas, líneas y planos) obtenidas con imágenes de cámaras, laser, sensores ultrasónicos o encoders [24].

Se plantean los siguientes objetivos:

- Analizar algoritmos de odometría visual que permitan estimar la posición de un robot en el espacio.

- Analizar algoritmos de mapeo que construyan mapas de ocupación.

- Analizar e implementar un algoritmo que resuelva el problema de SLAM a partir de mediciones de odometría visual.

- Implementar una solución que combine la localización en tiempo real de una cámara con la reconstrucción en 3D del entorno en la que se encuentra (mapa de ocupación).

En específico, se busca implementar el algoritmo ORB-SLAM 2 junto con un algoritmo de mapeo basado en *Octrees* (Octomap) sobre una plataforma con ROS (*Robot Operating System*) utilizando una cámara RGB-D, de tal forma que se tenga la trayectoria completa de la cámara y se construya, en *near-realtime*, un mapa de ocupación en 3D que sirva para la navegación autónoma de un vehículo aéreo.

III Planteamiento del problema

El problema del SLAM se describe usando terminología probabilística. Hacemos las siguientes definiciones:

Trayectoria del Robot

$$X_T = \{x_0, x_1, x_2, \ldots, x_T\} \tag{1.1}$$

Acciones de control entre $t-1$ y t

$$U_T = \{u_0, u_1, u_2, \ldots, u_T\} \tag{1.2}$$

Mediciones

$$Z_T = \{z_0, z_1, z_2, \ldots, z_T\} \tag{1.3}$$

Mapa

$$M = \{x_0, x_1, x_2, \ldots, n_1\} \tag{1.4}$$

Con esta terminología, podemos describir el problema del SLAM como la recuperación de un modelo del mapa M y la trayectoria del robot X_T a partir de la odometría U_T y las observaciones Z_T. Se distinguen dos tipos de SLAM:

- *Online* SLAM: Se ocupa de la estimación de las variables que persisten en el tiempo t
- *Full* SLAM: Se ocupa de la estimación de las variables durante toda la trayectoria del robot, es decir, de $1:t$

De manera más formal, el problema del Online SLAM se define como el cálculo de la posteriori de la pose momentánea y el mapa:

$$p(x_t, m \mid z_{1:t}, u_{1:t}) \tag{1.5}$$

mientras que el problema del Full SLAM se define como el cálculo de la posteriori de la trayectoria completa del robot junto con el mapa [29]:

$$p(x_{1:t}, m \mid z_{1:t}, u_{1:t}) \tag{1.6}$$

donde por *posteriori* nos referimos a la distribución de probabilidad a posteriori $p(x|y)$ sobre la variable probabilística X.

El problema de la construcción de un mapa de ocupación se define como el cálculo de la posteriori de un mapa dados la trayectoria y las mediciones:

$$p(m \mid z_{1:t}, x_{1:t}) \tag{1.7}$$

Capítulo 2

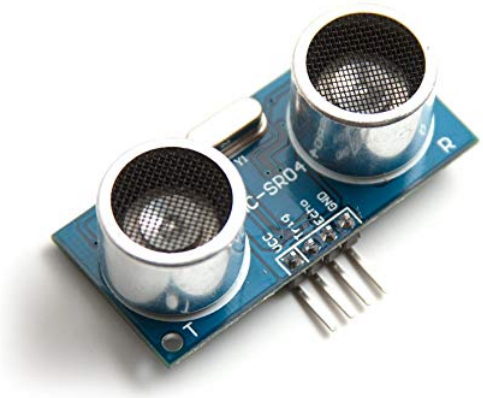
Estado del Arte

SLAM es una abreviación para Localización y Mapeo Simultáneos, por sus siglas en inglés. Es una técnica para estimar el movimiento de un sensor y reconstruir la estructura en un ambiente desconocido. Originalmente, SLAM fue propuesto para lograr un control autónomo en robots. Con el paso del tiempo, las aplicaciones del SLAM se han extendido desde modelado 3D basado en visión artificial, aplicaciones de realidad aumentada (AR) y automóviles autónomos. Dependiendo del tipo de sensor utilizado, se tienen diferentes técnicas para resolver el problema de SLAM. Entre los sensores más comunes tenemos:

- Cámaras
- Radares
- Sonares
- LiDAR
- Encoders
- Sensores inerciales
- GPS

(a) Cámara RGB-D Intel D435

(b) LiDAR Velodyne



(c) Sensor Ultrasonico HC-SR04 de bajo costo

(d) Sensor inercial de Analog Devices

Figura 2.1: Ejemplos de sensores usados en SLAM

El tipo de tecnología utilizado definirá en gran medida el algoritmo a utilizar. En la práctica, también es común ver una combinación de varias técnicas para mejorar los resultados.

I SLAM Visual

El interés en algoritmos de SLAM utilizando únicamente cámaras nace de la simpleza de la configuración de los sensores (únicamente necesitas una cámara que hoy en día está disponible en una amplia variedad de dispositivos portátiles) y de la dificultad técnica que estos algoritmos ofrecen. Los algoritmos de SLAM Visual (vSLAM) han sido propuestos en su mayoría para aplicaciones de AR, visión por computadora y robótica. Específicamente, son útiles en la estimación de la pose de la cámara, abriendo un abanico de posibilidades, desde aplicaciones de entretenimiento hasta el control de vehículos autónomos no tripulados. En [28] hacen una presentación de los principales algoritmos de vSLAM, clasificándolos en tres categorías:

- Algoritmos basados en características
- Algoritmos *directos*
- Basados en cámaras de profundidad RGB-D

Dependiendo del tipo de cámara usada, tenemos algoritmos de SLAM Monoculares (MonoSLAM), SLAM Stereo y RGB-D SLAM. El primer algoritmo de vSLAM fué desarrollado en 2003, ver [3]; fué llamado MonoSLAM y su propuesta fue estimar el movimiento y la estructura 3D de un ambiente desconocido usando un Filtro de Kalman Extendido (EKF). Este enfoque serviría de base para muchos otros algoritmos de vSLAM. En [12] desarrollaron PTAM, un algoritmo de tracking y mapeo para aplicaciones de AR en pequeños ambientes; su principal contribución fue dividir el tracking y el mapeo en diferentes hilos de procesamiento. Como una extensión del PTAM, nace

ORB-SLAM [19] que ya incluye BA [1], detección de lazos cerrados basados en visión artificial y una optimización de la pose. ORB-SLAM evolucionó a ORB-SLAM2 [20], un completo algoritmo de SLAM para sistemas monoculares, stereo y RGB-D que incluye reutilización de mapas, cerradura de lazos y capacidades de relocalización. Este sistema trabaja en tiempo real en una gran variedad de dispositivos, desde celulares hasta sistemas embebidos.

ORB-SLAM ha servido como punto de partida para diferentes algoritmos y aplicaciones. Por ejemplo, en [31] se desarrolla un algoritmo robusto para cámaras RGB-D que opera en tiempo real e interiores. El sistema está compuesto de 3 componentes centrales: tracking, mapeo y cerradura de lazos, basados en ORB-SLAM2. A diferencia de este último, la parte de tracking estima la pose de un marco a través de la optimización del error de reproyección y el error de profundidad inverso de puntos característicos coincidentes. La parte de mapeo optimiza todas las poses de los *keyframes* y las posiciones de todos los puntos del mapa usando BA, además de que construye un mapa de ocupación usando OctoMap [8]. En [16] combinaron ORB-SLAM con un método de estimación de ocupación probabilística para proveer tanto la localización de un robot en tiempo real como un mapa que permita ocupar algoritmos de navegación y evasión de obstáculos (Figura 2.2): el sistema fue implementado usando un sensor Kinect 2.0.

Con el desarrollo de tecnologías como *ToF* (Time of Flight) y *Luz Estructurada* llegan al mercado nuevos dispositivos que combinan la obtención de imágenes con una medición de profundidad de cada pixel. Estos dispositivos son conocidos como *cámaras RGB-D* donde la *D* representa la profundidad o *depth* del pixel. En contraste con los algoritmos de SLAM donde la profundidad se calcula a partir del movimiento de la imagen (cámaras monoculares) o triangulación (cámaras stereo), las cámaras RGB-D proporcionan una estructura 3D del ambiente.

[1] *BA* o Bundle Adjustment es el problema de refinar una reconstrucción visual para producir una estructura 3D óptima y estimaciones de los parámetros de vista (pose de la cámara y/o calibración)

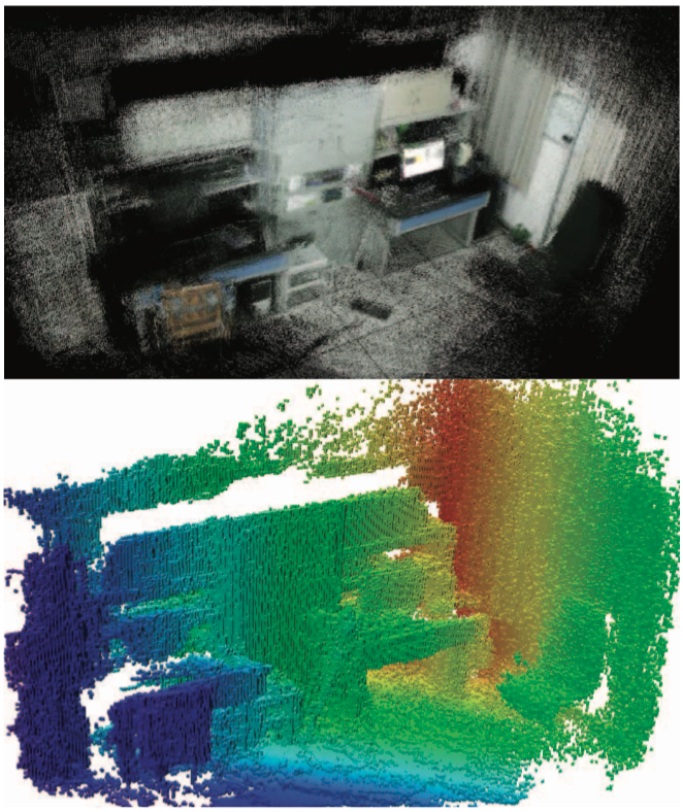


Figura 2.2: Ejemplo de mapa reconstruido con sensor Kinect 2 [16]

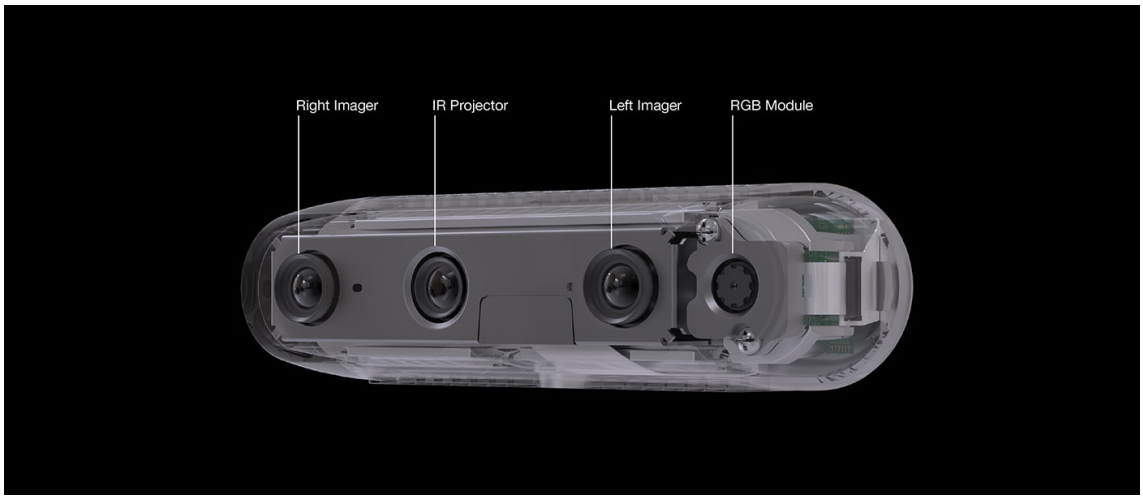


Figura 2.3: Sensores de la cámara RGB-D Intel D435

II Algoritmos basados en filtros

La estimación de estados aborda el problema de inferir cantidades que no son directamente observables a partir de mediciones de sensores. Un caso particular es la estimación de la posición de un robot o vehículo. Éste confía en un conjunto de mediciones ruidosas para determinar en qué posición se encuentra relativo a un marco de referencia. Entre las distintas formas de estimar estados, tenemos a la *estimación probabilística de estados* que no son más que un conjunto de algoritmos que calculan distribuciones de probabilidad sobre los posibles estados. El *filtro de Bayes* es el algoritmo más general para calcular distribuciones de probabilidad; se trata de un algoritmo recursivo que calcula la *creencia* de x_t, $bel(x_t)$, a partir de creencias en $t-1$ y mediciones de sensores. Se tienen dos familias principales de filtros:

- Filtros Gaussianos
- Filtros paramétricos

Entre los filtros más representativos tenemos el Filtro de Kalman, el Filtro de Kalman Extendido (EKF), *Unscented Kalman Filter* (UKF) y el Filtro de Partículas (PF). Una gran cantidad de algoritmos de SLAM están basadas en distintas implementaciones de estos filtros, por ejemplo el algoritmo de *Localización Monte Carlo* y *FastSLAM* [17]. Por su facilidad de implementación, su bajo coste computacional y la variedad de sensores que soportan, existe una amplia literatura con distintas aplicaciones y variaciones a los algoritmos. Por ejemplo, en [2] presentan un nuevo Filtro de Partículas basado en el UKF que presenta mejor desempeño que el PF tradicional. Usan el UKF para refinar la predicción a través de un modelo auxiliar que genera la distribución de importancias en el filtro de partículas, haciéndolo más eficiente. En [15] desarrollan un nuevo algoritmo de FastSLAM basado también en el UKF: su método de construcción incluye un LiDAR para escanear entornos en interiores. En [10] combinan un filtro de partículas de Rao-Blackwell con un mapa de voxels para crear un

FastSLAM en 3D; como sensor utilizan un Kinect y como plataforma un robot Pioneer 2-DX. Cada medición en 3D del sensor provee mediciones de probabilidad de que el espacio esté ocupada y, con los datos recolectados y haciendo uso de OctoMap [8], construyen un modelo en 3D del ambiente. En [13] muestran una comparativa de varios algoritmos de SLAM: se compara el rendimiento del Filtro de Kalman Extendido (EKF), Unscented Kalman Filter (UKF), FastSLAM basado en el EKF (FastSLAM 2.0) y FastSLAM basado en el UKF (uFastSLAM). Para demostrar la fusión de datos, en [14] combinan el uso de un LiDAR con un encoder rotatorio y un sensor Kinect; utilizan un filtro de partículas para calcular la posición del robot y OctoMap [8] para obtener una reconstrucción del ambiente (Figura 2.4).

Los algoritmos de SLAM basados en filtros sirven para la navegación autónoma de distintos vehículos. En [7] implementaron un PSO-uFastSLAM en un vehículo submarino (Figura 2.5). y en [23] implementaron un uFastSLAM en un vehículo aéreo no tripulado (UAV) para dotarlos de autonomía.

Otro ejemplo está en [25] donde combinan un Filtro de Partículas de Rao-Blackwell (RBPF) que usa un sensor activo de rango (láser o transductores sónicos) junto con un sensor de visión estereoscópica para inferir puntos representativos en el espacio para calcular la posición de un robot; el mapa que construyen es un mapa de ocupación de dos dimensiones que permite adjuntar algoritmos de navegación para logar un mapeo autónomo.

Una particularidad del SLAM basado en Filtros de Partículas es que cada partícula tiene su propia versión del mapa del ambiente completo. En [9] proponen un nuevo enfoque: que cada partícula tenga sólo un pequeño mapa de su entorno cercano, llamado *mapa individual* y que el mapa global, llamado *mapa base* sea compartido por todas las partículas. Con esto se logra una mejora en la memoria computacional.

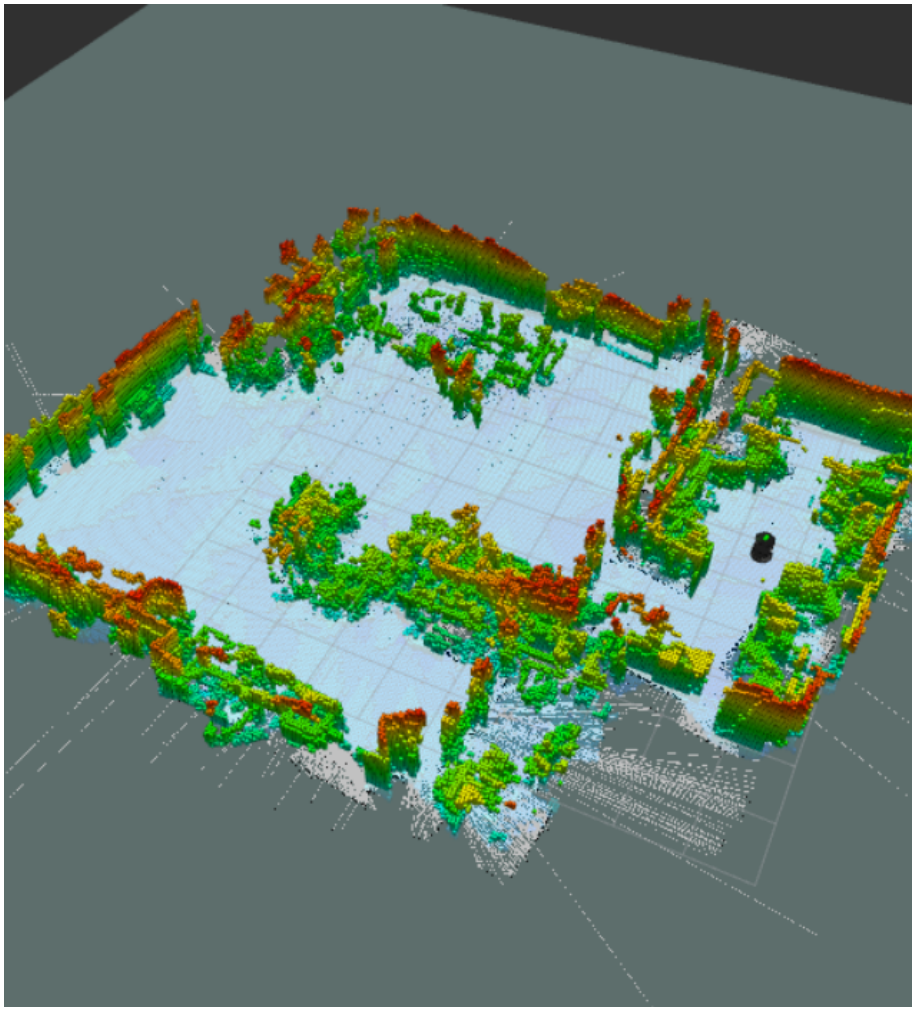


Figura 2.4: Mapa de Cuadrícula con OctoMap [14]

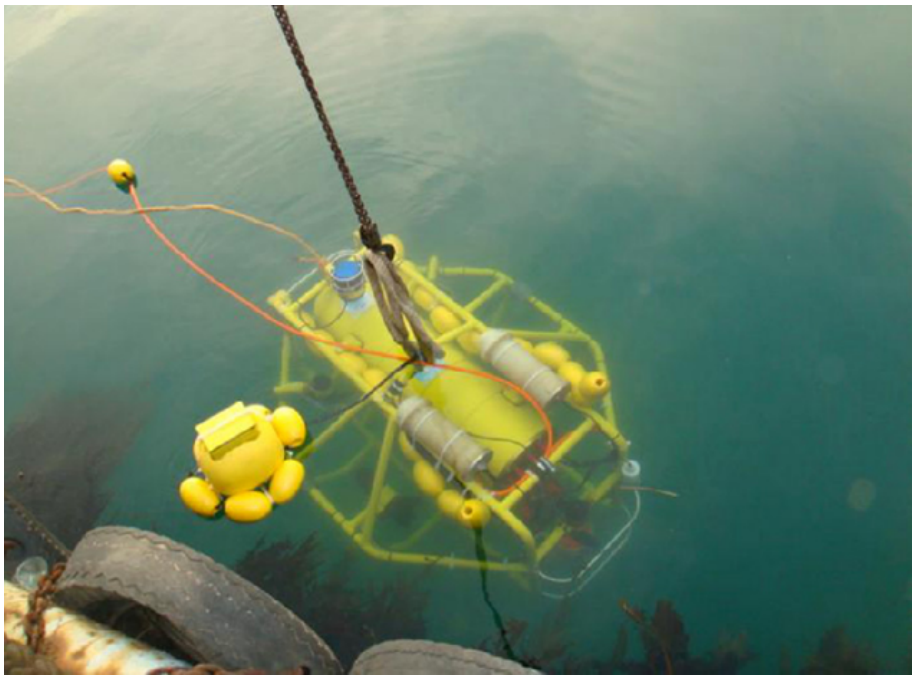


Figura 2.5: Vehículo Submarino Autónomo [7]

III Tecnologías relacionadas

SLAM trata de resolver los problemas de localización de un sensor y reconstrucción del entorno de manera simultánea. No obstante, hay algoritmos que resuelven de forma parcial dichos problemas y que, de alguna forma u otra, sirven de base para el diseño de algoritmos más completos como el caso del SLAM. Además de los algoritmos de SLAM, existen algoritmos de odometría visual y algoritmos de estructura a partir del movimiento diseñados para estimar la pose del sensor y la estructura 3D del ambiente [28].

Odometría se refiere a la estimación de los cambios secuenciales de posición de un sensor en el tiempo. Esta estimación puede lograrse a partir de la lectura de diferentes variables: posición, velocidad y aceleración [2]. En el caso de *odometría visual*, el cálculo

[2]Para calcular la posición x_t de un sensor partimos de las relaciones de velocidad $v = \dot{x}_t$ y aceleración $a = \ddot{x}_t$

de la posición se hace a partir del cálculo de las posiciones de puntos característicos de la imagen, por ejemplo.

Estructura a partir del movimiento es una técnica para estimar el movimiento de una cámara y la estructura en 3D del ambiente.

Capítulo 3

Algoritmos de Localización sin GPS

I Localización y Mapeo Simultáneos

Imagina que un robot necesita realizar alguna tarea en un entorno desconocido. Ejemplo de esto puede ser un dron que necesita monitorear las instalaciones de una presa hidroeléctrica, un vehículo submarino autónomo que va a realizar tareas de inspección o simplemente un robot de limpieza doméstico. Para poder realizar la tarea requerida de manera satisfactoria, una de las primeras cosas que debe de lograr es localizarse en el entorno. El problema de localización no es exclusivo de robots: cuando conducimos nuestro auto, contamos con sistemas de localización externos, como el GPS, que nos ayudan a saber si vamos en la ruta correcta. No obstante, es deseable no depender de sistemas de localización externos sino basar el proceso de localización en algún proceso interno a partir de datos obtenidos por el mismo dispositivo (al fin y al cabo, es la forma en que los seres vivos nos localizamos).

El método más simple para localizar un robot es mediante *odometría*, que es el cálculo del movimiento incremental a partir de datos obtenidos por sensores, como encoders o sensores inerciales. A partir del cálculo de la odometría de un robot podemos determinar su trayectoria. Una de las desventajas de este método es la acumulación

de errores o *deriva*, que tiende a desviar la trayectoria calculada de la real conforme aumenta el movimiento del robot. Una forma de corregir la deriva del movimiento del robot es calcularla y compensarla a partir de la información de un *mapa*. El *mapeo* es la tarea de construir un mapa a partir de la posición de un sensor. Como podemos ver, para poder obtener la localización de un robot de manera precisa necesitamos un mapa y, para poder obtener dicho mapa necesitamos conocer con precisión la localización de un robot. *SLAM* (Localización y Mapeo Simultáneos, por sus siglas en inglés) es una técnica que resuelve este problema.

El problema de SLAM involucra varios retos [18]:

- **Inicialización**: Como pretendemos resolver el problema de localización y mapeo al mismo tiempo, necesitamos localizar al robot en un primer instante, cuando no se tiene ningún mapa

- **Cerradura de lazo**: Cuando un robot está explorando el ambiente, los algoritmos de SLAM tienen un comportamiento similar que las técnicas de odometría, en el sentido que generan cierta deriva conforme aumenta su movimiento. Habíamos dicho que para reducir el efecto de la deriva, necesitábamos de un mapa. La cerradura de lazo se refiere a la identificación de un punto visitado previamente por el robot que le permita corregir la trayectoria calculada.

- **Relocalización**: Es similar a la inicialización. Se refiere a poder localizar un robot en un mapa sin ninguna predicción previa. A este problema también se le conoce como el *problema del robot secuestrado*, donde un robot necesita localizarse después de haber perdido información de sus sensores.

- **Reutilización de mapas**: Si un robot se está moviendo por una zona previamente mapeada, lo deseable es que únicamente se localice sin volver a mapear.

- **Tiempo Real**: Se refiere a la estimación de la localización y la construcción del

mapa del entorno *en línea*, es decir, conforme vaya recibiendo la información de sus sensores y no en un procesamiento posterior.

- **Robustez**: Las técnicas de SLAM asumen mapas estáticos, por lo que los elementos dinámicos de un entorno (personas, objetos que se mueven, cambios de iluminación) pueden traer errores en los cálculos. Entonces, se necesitan mecanismos que disminuyan estos efectos.

ORB-SLAM [19] es un algoritmo que resuelve el problema del SLAM utilizando una cámara monocular como único sensor. Una segunda versión, *ORB-SLAM2* [20] extiende los sensores utilizados a cámaras stereo y RGB-D. Hablando de cámaras, las podemos clasificar como sensores pasivos (monocular y stereo), es decir, que observan el mundo sin alterarlo y sensores activos (RGB-D), que obervan el mundo a partir de la reflexión de señales enviadas por la misma cámara. El elegir una cámara como fuente de información para los algoritmos de SLAM tiene varias ventajas: su precio es accesible, son compactas y existe una alta disponibilidad de modelos en el mercado. Además, una sola imagen contiene una vasta cantidad de información; es por ello que los humanos y distintas especies de animales usamos la visión como sensor principal para movernos en el espacio, reconocer lugares e interactuar con el entorno.

II Fundamentos de Visión por Computadora

El análisis de imágenes y su procesamiento son dos importantes campos de investigación que son conocidos como visión por computadora y procesamiento de imágenes. El instrumento que nos permite obtener datos para estos análisis es la *cámara digital*. Su funcionamiento es el siguiente: después de que la luz se refleja en una o más superficies del ambiente y pasa a través de la óptica de la cámara (lentes) llega al sensor de imagen. Este sensor es el responsable de convertir los fotones recibidos en valores

(R,G,B) que serán usados para analizar la imagen. Los dos tipos de sensores más utilizados son el CCD (charge coupled device) y el CMOS (complementary metal oxide on silicon), cada uno con sus ventajas y desventajas. De ambos, el más común es el CMOS ya que proporciona algunas ventajas sobre la tecnología CCD, como la ausencia de circuitería de tiempo especializada y el poder reutilizar los mismos procesos usados en la industria de microcontroladores para producir sensores CMOS.

II.1 Óptica

Una vez que la luz de la escena llega a la cámara, debe de pasar por una serie de lentes antes de llegar al sensor. En la Figura 3.1 se muestra el modelo más básico de un lente, conocido como *lente delgado*. Este lente está compuesto de una sola pieza de vidrio con muy poca e igual curvatura en ambos lados. De acuerdo con la *Ley de Refracción y Lentes*, la relación entre la distancia a un objeto z y la distancia detrás del lente a la cual el objeto es enfocado se puede expresar como

$$\frac{1}{f} = \frac{1}{z} + \frac{1}{e} \tag{3.1}$$

donde f es la distancia focal. De esta fórmula podemos deducir la distancia estimada a un objeto conociendo la distancia focal del lente y la distancia del plano de imagen al lente. A esta técnica se le conoce como *depth from focus* (profundidad por enfoque). Si el plano de imagen está localizado a una distancia e del lente, entonces para un objeto específico toda la luz estará enfocada en un solo punto en el plano de imagen y la imagen estará *enfocada*. Se tienen dos desventajas fundamentales con esta técnica: necesitamos un algoritmo que determine si una imagen está enfocada o no y se pierde sensibilidad conforme los objetos se alejan del lente. Es importante destacar que este último punto es igual de válido para todas las técnicas que miden rango a partir de visión.

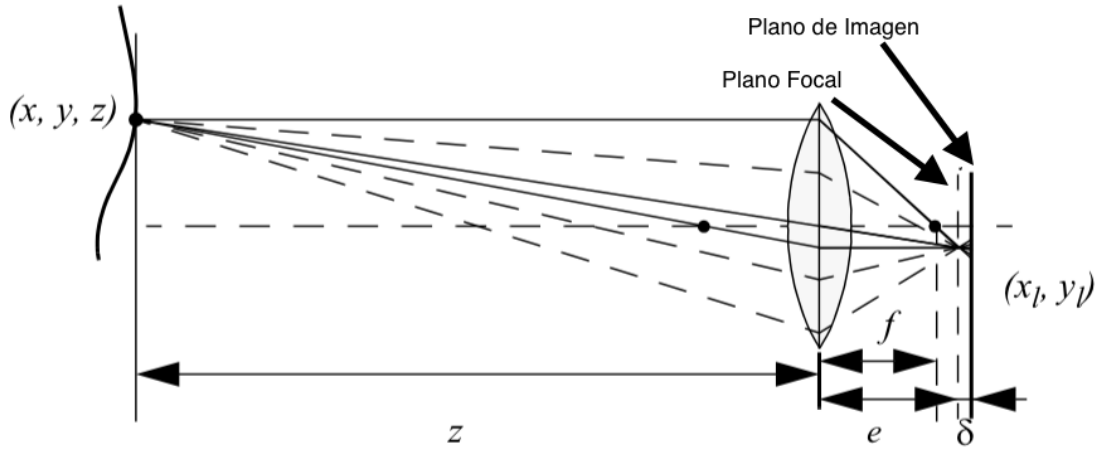


Figura 3.1: Modelo básico de lente

II.2 Modelo de cámara oscura

La *cámara oscura* es el primer ejemplo de cámara de la historia que llevó a la invención de la fotografía. Una cámara oscura no tiene lente, únicamente una pequeña abertura. La luz de la escena pasa a través de esta apertura y projecta una imagen invertida del otro lado de la caja (Figura 3.2). La importancia de este modelo radica en que si $z \to \infty$, esto es, ajustamos el lente de tal forma que el infinito esté enfocado, tenemos que $e = f$, por lo que podemos tomar a un lente con distancia focal f como equivalente a una cámara oscura con distancia f del plano focal (Figura 3.3). Al usar este modelo es importante recordar que el orificio (pinhole) corresponde al centro del lente. A este punto también se le conoce como *centro de proyección* o *centro óptico* (indicado con la letra c en la figura). Al eje perpendicular al plano de imágen Π que pasa por el centro de proyección se le conoce como *eje óptico*. Por conveniencia, el modelo de cámara oscura es representado con el plano de imagne entre el centro de proyección y la escena, para conservar la orientación de la imagen. A la intersección O entre el eje óptico y el plano de imagen se le conoce como *punto principal*. De la Figura 3.3 se ve que el modelo no mide distancias sino ángulos.

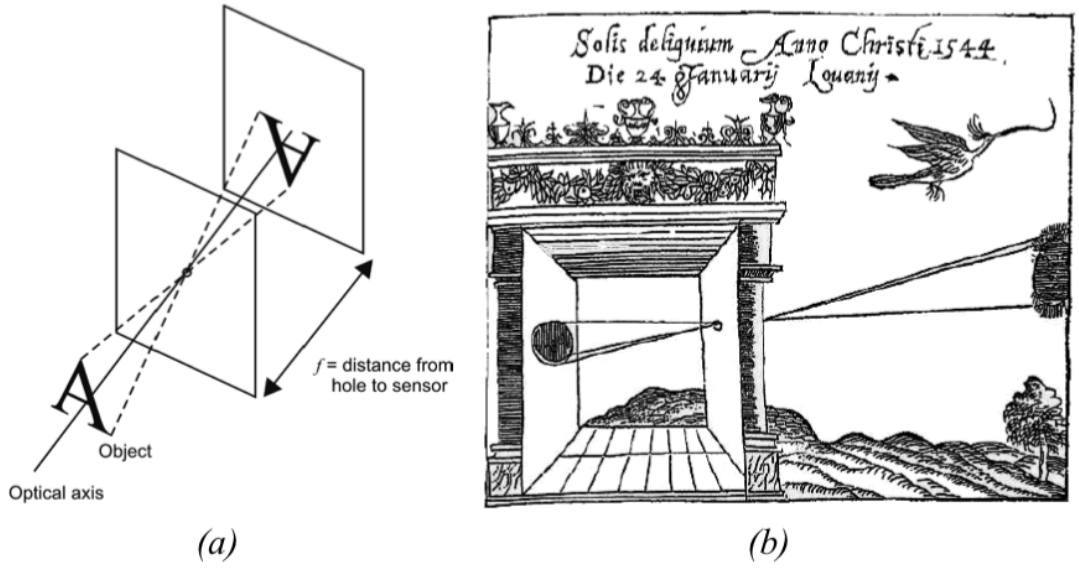


Figura 3.2: Modelo de cámara oscura [24]. (a) Cuando $d \gg f$ la cámara puede ser modelada como una cámara oscura. (b) Dibujo de Reinerus Gemma-Frisius (1508 - 1555) para ilustrar la cámara oscura

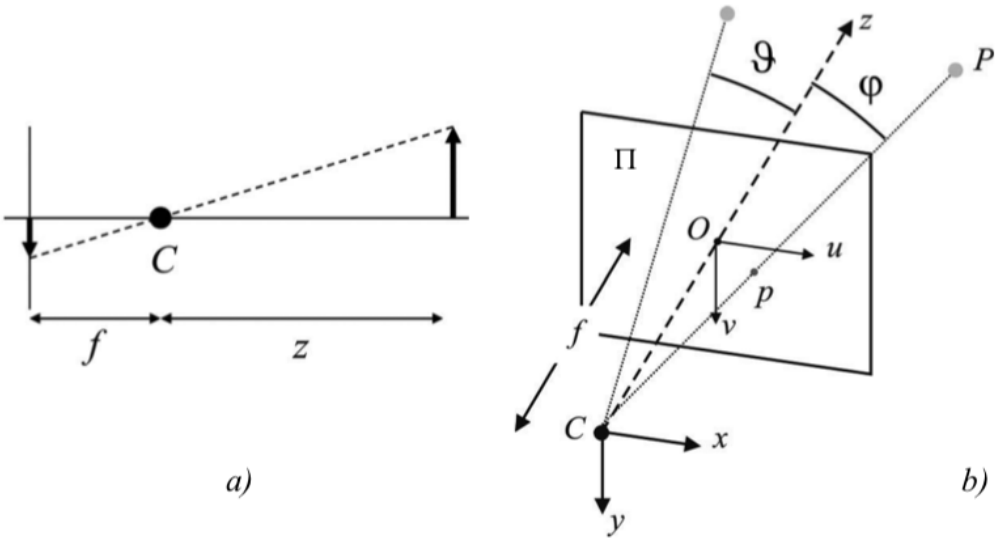


Figura 3.3: (a) Modelo de cámara oscura para representar cámaras de perspectiva estándar (b) El modelo de cámara oscura comúnmente se representa con el plano de imagen entre el centro de proyección y la escena para preservar la orientación de la imagen

Para describir analíticamente la *proyección de perspectiva* operada por una cámara, debemos de introducir marcos de referencia oportunos para expresar las coordenadas 3D de un punto de escena P y las coordenadas p de su proyección en el plano de imagen. Sean (x, y, z) el marco de referencia con origen en C y el eje z coincidente con el eje óptico. Asumamos que el marco de referencia del ambiente coincide con el marco de referencia que acabamos de definir. Introduzcamos ahora un marco de referencia bidimensional (u, v) para el plano de imagen Π con origen en O y los ejes u y v alineados con los ejes x y y respectivamente, como se ve en la Figura 3.3b. Finalmente, definimos las coordenadas del punto de escena y su proyección como $P = (x, y, z)$ y $p = (u, v)$. Por triángulos similares se puede deducir que

$$\frac{f}{z} = \frac{u}{x} = \frac{v}{y} \tag{3.2}$$

por lo que

$$u = \frac{f}{z} \cdot x, \tag{3.3}$$

$$v = \frac{f}{z} \cdot y \tag{3.4}$$

Esta es la *proyección de perspectiva* [24]. Usando *coordenadas homogéneas* podemos obtener un modelo lineal. Definimos

$$\tilde{p} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \text{ y } \tilde{P} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.5}$$

como las coordenadas homogéneas de p y P. La ecuación de proyección de perspectiva

puede ser reescrita como

$$\begin{bmatrix} \lambda u \\ \lambda v \\ \lambda \end{bmatrix} = \begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.6}$$

Notamos que λ es igual a la tercer coordenada de P que, en este marco de referencia, coincide con la distancia del punto al plano xy. Esta ecuación también nos muestra que cualquier punto de la imagen es una proyección de todos los puntos 3D infinitos que se encuentran en la línea que pasa por el punto de la imagen y el centro de proyección. Es por esto que usando una sola cámara oscura no es posible estimar la distancia un punto; necesitamos 2 cámaras (cámaras estéreo).

Las ecuaciones anteriores son muy simplificadas: un modelo más completo requiere tomar en cuenta factores como la pixelización de la imagen, transformaciones entre el marco de referencia de la cámara y del mundo, distorsiones debido a los lentes y parámetros intrínsecos y extrínsecos de la cámara. Un modelo más general de la cámara oscura que toma en cuenta la diferencia entre los marcos de referencia de la cámara y el mundo, los parámetros intrínsecos de la cámara y un centro óptico que no esté exactamente en el centro es [24]

$$\lambda \tilde{p} = A[R|t]\tilde{P}_w \tag{3.7}$$

donde t es una traslación y R una matriz de rotación para transformar entre los marcos de referencia de la cámara y del mundo, $\tilde{P}_w$ son las coordenadas del punto $\tilde{P}$ expresadas en el marco de referencia del mundo y A está dada por

$$A = \begin{bmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.8}$$

donde α_u y α_v son parámetros intrínsecos de la cámara que se obtienen por medio de un proceso llamado *calibración* y u_0 y v_0 son las coordenadas del centro óptico de la cámara.

III Solución del SLAM Visual

El problema se formula de la siguiente forma: *dado un flujo contínuo de imágenes por un sensor de visión (e.g. una cámara), ¿cómo podemos explotar esta información para solucionar el problema de SLAM?*. Existen dos familias de métodos para la resolución de este problema: métodos basado en características y métodos directos.

III.1 Métodos basados en características

Los métodos basados en características procesan las imágenes para extraer puntos de interés distintivos (*keypoints*) que pueden ser detectados de manera confiable y repetida en imágenes de la misma escena desde diferentes puntos de vista y condiciones de iluminación. Un descriptor, típicamente un vector de valores binarios reales de cierta longitud, es calculado para cada keypoint al operarse en un marco de pixeles alrededor del keypoint. Esto permite emparejar keypoints entre imágenes comparando sus descriptores. A la combinación de un keypoint con su descriptor se le conoce como *feature* (característica). Una vez que se extraen todos los features de una imagen, podemos descartar la imagen completa (conjunto de pixeles) ya que los métodos basados en características únicamente operan sobre estos features. La ventaja de este proceso es que los features son entidades geométricas que son fáciles de emparejar y manipular.

La optimización de los métodos basados en características se enfoca en reducir el *error de reproyección*. Dada la correspondencia entre un punto 3D dado en coorde-

nadas globales $\mathbf{X}_w$ y un keypoint 2D x_c, el error de reproyección e_{proj} se calcula con:

$$e_{proj} = x_c - \pi_m(\mathbf{R}_{cw}\mathbf{X}_w + c\vec{p}_w) \tag{3.9}$$

donde $\mathbf{R}_{cw} \in \mathrm{SO}(3)$ y $c\vec{p}_w$ son la rotación y traslación del inverso de la pose de la cámara, que transforman puntos de coordenadas globales a coordenadas de la cámara y π_m es la función de proyección y está dada por:

$$\vec{x} = \pi_m(\mathbf{X}_c) = \begin{bmatrix} f_x \frac{X}{Z} + c_x \\ f_y \frac{Y}{Z} + c_y \end{bmatrix}, \qquad \mathbf{X}_c = [X, Y, Z]^T, \quad x = [u, v]^T \tag{3.10}$$

donde f_x y f_y son las distancias focales horizontal y vertical y c_x y c_y las coordenadas horizontal y vertical del punto principal.

A la optimización de las posiciones de un conjunto de puntos $\boldsymbol{\mathcal{P}}$ y las poses de un conjunto de cámaras $\boldsymbol{\mathcal{C}}$, minimizando el error de reproyección se le conoce como *Bundle Adjustment* (BA) [30] y es la optimización central de los métodos modernos basados en carácterísticas para resolver el problema del SLAM:

$$\left\{\mathbf{X}_w^j, \mathbf{R}_{cw}^i, c\vec{p}_w^i \mid \forall j \in \boldsymbol{\mathcal{P}}, \forall i \in \boldsymbol{\mathcal{C}}\right\} = \underset{\mathbf{X}_w^j, \mathbf{R}_{cw}^i, c\vec{p}_w^i}{\arg\min} \sum_{i,j} \rho\left(\left\| x_i^j - \pi_m\left(\mathbf{R}_{cw}^i \mathbf{X}_w^j + c\vec{p}_w^i\right)\right\|_{\sum_i^j}^2\right) \tag{3.11}$$

donde x_i^j es el keypoint asociado al punto 3D $\mathbf{X}_w^j$, $\sum_i^j$ es la covariancia de la ubicación del punto x_i^j en la imagen de la cámara, $\| \cdot \|$ es la distancia de mahalanobis y ρ es una función de costo robusta para eliminar outliers; en ORBSLAM usan la función de costo de Huber.

La principal limitación de los métodos basados en características es que solo pueden explotar información de imágenes donde se pueden extraer features, típicamente esquinas. Si una imagen carece de textura o es borrosa, estos métodos tienen un mal desempeño. Por otro lado, los mapas generados con estos métodos se constituyen por un conjunto de puntos poco densos que no tienen más uso que la localización del robot.

III.2 Métodos directos

Los métodos directos utilizan de manera directa la información proporcionada por los sensores, en nuestro caso, los valores de intensidad de cada pixel en la imagen. Un método directo puede ser *denso* si utiliza todos los pixeles de la imagen, *semidenso* si solo se consideran pixeles con alto gradiente o *disperso* si solo ocupa una pequeña fracción de los pixeles. Como estos métodos explotan la información de la imagen sin tener que extraer features, son más precisos y robustos en escenas con poca textura o borrosas. La reconstrucción del SLAM con estos métodos consiste en calcular la profundidad asociada a cada pixel y las optimizaciones están basadas en el error fotométrico.

IV Reconocimiento de Lugares con Características ORB

Es fundamental para cualquier sistema de SLAM visual reconocer ambientes previamente mapeados. Esto permite al sistema relocalizar el sensor después de un fallo en el rastreo, ya sea debido a una oclusión de la cámara, un movimiento abrupto o para cerrar lazos en las trayectorias y eliminar el error acumulado durante la exploración.

El reconocimiento de lugares describe la capacidad de nombrar lugares discretos en el mundo. Un requerimiento es que sea posible obtener un particionado disreto del ambiente en lugares y relacionarlos con una *representación* del lugar y que estas representaciones sean almacenadas en una base de datos. El proceso de reconocimiento de lugares funciona calculando una representación de las mediciones de un sensor y buscar en la base de datos la representación más similar almacenada; la representación obtenida nos dirá entonces la localización del robot.

La representación de una imagen por un conjunto de puntos de interés es llamada *bag of features*. Para cada punto de interés, usualmente se calcula un descriptor de manera que sea invariante a rotaciones, escala, intensidad y cambio de punto de vista.

Este conjunto de descriptores se convierte en la nueva representación de la imagen. La similaridad entre dos conjuntos de descriptores puede ser calculada contando el número de descriptores comúnes. Para llevar a cabo esto, necesitamos definir una función de correspondencia que nos permita determinar si dos features son los mismos. Esta función de correspondencia usualmente depende del tipo de descriptor del feature. En general, el descriptor de un feature es un vector de varias dimensiones y la correspondencia entre features puede determinarse mediante el cálculo de la norma L_2. Las *palabras visuales* (visual words) son una representación unidimensional del descriptor de n dimensiones. La conversión a palabras visuales (visual words) crea una *bolsa de palabras visuales* (bag of words) en lugar de una *bolsa de características*. Para llevar a cabo esta conversión el espacio n-dimensional de descriptores es dividido en celdas que no se superponen. A cada una de estas celdas se le asigna un número que será asignado a cualquier descriptor que se encuentre dentro de la celda. A este número se le conoce como *palabra visual*. Por lo tanto, descriptores similares serán ordenados en la misma celda y se les asignará la misma palabra visual. Con esto, conseguimos un método muy eficiente para encontrar correspondencias entre descriptores similares. En la Figura 3.4, se ilustra el proceso de conversión de descriptores a palabras visuales.

IV.1 DBoW2

Las técnicas basadas en *bolsas de palabras* ayudan a resumir el contenido de una imagen por las palabras visuales que contienen, siendo útil para poder detectar si una imagen corresponde a un lugar previamente visitado. Estas palabras visuales corresponden a una discretización del espacio de descriptores, conocido como *vocabulario visual*. DBoW2 crea un vocabulario estructurado en forma de árbol [22] en un paso offline sobre un conjunto grande de descriptores extraidos de un conjunto de imágenes de entrenamiento.

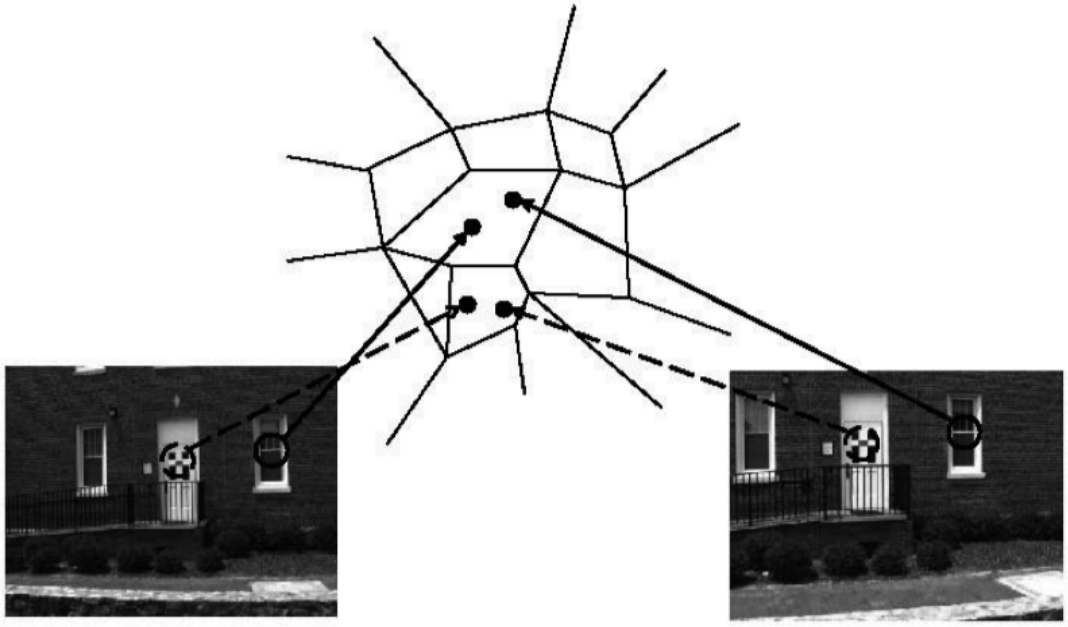


Figura 3.4: Partición del espacio de descriptores en celdas. Cada celda corresponde a una palabra visual. Descriptores similares se ordenan en la misma celda y por lo tanto les corresponde la misma palabra visual [24]

El procesamiento de nuevas imágenes consiste en extraer keypoints y sus descriptores, que son asignados a palabras visuales atravesando el árbol del vocabulario calculado offline. Al ser los descriptores binarios, las distancias son calculadas con la distancia de Hamming. El resultado es un vector de bolsas de palabras que contiene una puntuación *tf-idf* (*term-frequency - inverse document frequency*) que determina la frecuencia de cada palabra presente en la imagen. Esta puntuación es mayor conforme una palabra visual es más frecuente en la imagen y menor en el conjunto de entrenamiento. Este vector es comparado contra los vectores de bolsas de palabras de las imágenes que se encuentran en una base de datos que es construida incrementalmente.

La similaridad entre dos vectores de bolsas de palabras $\vec{v}_1$ y $\vec{V}_2$ es la puntuación L_1:

$$s(\vec{v}_1, \vec{v}_2) = 1 - \frac{1}{2}\left|\frac{\vec{v}_1}{|\vec{v}_1|} - \frac{\vec{v}_2}{|\vec{v}_2|}\right| \tag{3.12}$$

Esta puntuación es normalizada con la puntuación que esperaríamos obtener de una imágen mostrando el mismo lugar. Para detección de lazos en una secuencia de video podemos obtener una puntuación de referencia de la imagen previa:

$$\eta(\vec{v}_1, \vec{v}_2) = \frac{s(\vec{v}_i, \vec{v}_j)}{s(\vec{v}_i, \vec{v}_{i-1})} \tag{3.13}$$

Imágenes en la base de datos cercanas en el tiempo tendrán puntuaciones similares. DBoW2 se aprovecha de esto agrupando imágenes cercanas en el tiempo y calculando una puntuación por el grupo (sumando las puntuaciones individuales). Una vez que se busca en la base de datos, el grupo con la mayor puntuación es seleccionado y la imagen con la mayor puntuación indivual es considerada como candidata para cerrar el lazo.

Para que una imagen candidata para cerradura de lazo no sea rechazada tiene que ser consistente con k consultas previas, esto es, los grupos con mayores puntuaciones para las últimas k imágenes deben de formar una secuencia superpuesta, aumentando la robustez ya que el lazo solo se cierra si hay suficiente evidencia que lo respalde.

Finalmente, una imagen candidata para cerradura de lazo es aceptada si pasa una revisión geométrica, que consiste en calcular una matriz fundamental con RANSAC.

IV.2 Reconocimiento de Lugares con ORB

Una esquina en una imagen puede ser definida como la intersección de dos o más bordes. Las esquinas son características altamente repetibles en una imagen. Uno de los primeros detectores de esquinas en las imágenes fue desarrollado por Moravec. Él definió una esquina como el punto donde hay una alta variación en la intensidad en toda dirección. De manera intuitiva, podemos reconocer esquinas al observar una pequeña ventana centrada en algún pixel arbitrario. Si el pixel está en una región plana (intensidad uniforme), entonces los pixeles adjacentes se veran similar. Si un pixel está a lo largo de un borde, entonces ventanas adjacentes en dirección perpendicular al borde se verán diferentes pero ventanas adjacentes en dirección paralela al borde se verán similares. Finalmente, si un pixel está en una esquina, ninguna de las ventanas adjacentes se verá igual (Figura 3.5).

Existen distintos algoritmos para la detección de esquinas, siendo uno de los más populares FAST (Features from Accelerated Segment Test). Este método está basado en una aproximación morfológica en lugar de calcular gradientes locales en las imágenes, que es computacionalmente caro. FAST funciona de la siguiente forma: para cada pixel en la imagen, considera un circulo de radio fijo centrado en dicho pixel. Después, todos los pixeles dentro de esta circulo son divididos en dos categorias, dependiendo si tienen intensidades "similares" o "diferentes" al pixel central. En regiones planas (de intensidad uniforme) la mayoría de los pixeles en el circulo tendrán un brillo similar al pixel del centro. Cerca de algún borde, la fracción de pixeles con intensidad similar decaerá en un 50% mientras que cerca de las esquinas decrementarán en alrededor de 25%. De esta forma, las esquinas son identificadas como las localidades de la imagen donde el número de pixeles con brillo similar en una vecindad

local alcanza un mínimo local y está debajo de cierto umbral (Figura 3.6).

ORB [22], que significa Oriented FAST and Rotated BRIEF, es un descriptor binario invariante a la rotación y resistente al ruido. Como su nombre lo indica, está basado en el detector de keypoints FAST [21] y en los descriptores BRIEF [1].

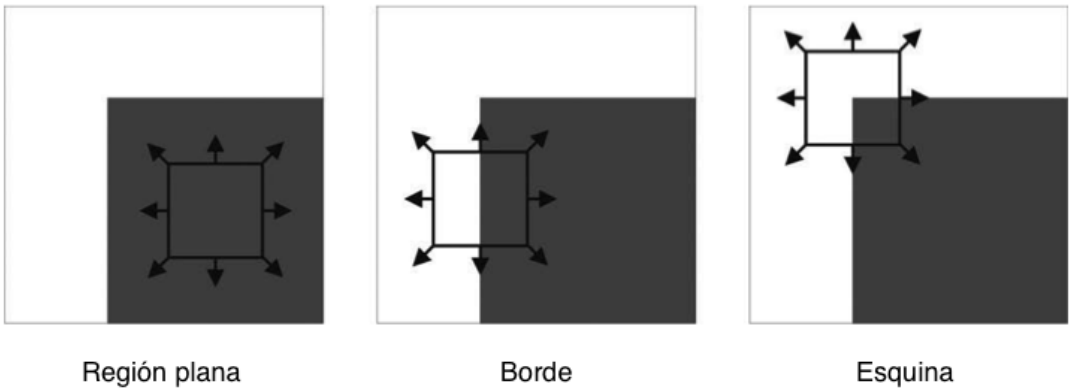


Figura 3.5: Bordes y esquinas en una imagen

ORB-SLAM utiliza un algoritmo de reconocimiento de lugares basado en DBoW2 con características ORB. Usa el extractor ORB de la librería OpenCV y crea un vocabulario visual de manera offline con el dataset Bovisa 2008-09-01, que es una secuencia de imágenes en exteriores e interiores. El resultado final es un vocabulario de 1 millón de palabras que es usado para buscar correspondencias de características ORB [18].

V ORB-SLAM Monocular

Dentro de los métodos de SLAM basados en características tenemos dos tipos: los basados en filtros y los basados en Bundle Adjustment (BA). El primer SLAM visual Monocular fue desarrollado en 2003 [4] [5] y fue llamado MonoSLAM. En MonoSLAM, el movimiento de la cámara y la estructura 3D del entorno son estimados simultaneamente usando un Filtro de Kalman Extendido (EKF). Los 6 grados de libertad (DoF)

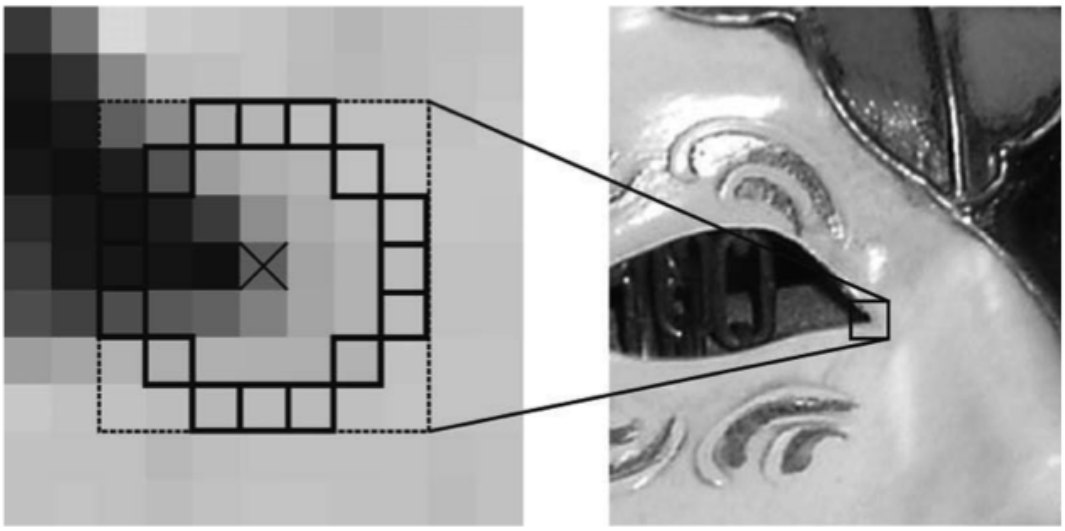


Figura 3.6: Esquinas FAST

de la cámara y las posiciones 3D de los features son representados como un vector de estados en el EKF. En el modelo de predicción, se asume un movimiento uniforme y el resultado del rastreo de los features es usado como observacion para el algoritmo. Dependiendo del movimiento de la cámara, se añaden nuevos features al vector de estado. El problema de este método es el costo computacional que aumenta en proporción al ambiente. En entornos grandes, el tamaño del vector de estados se vuelve enorme porque aumenta el número de features insertados; con este enfoque es dificil conseguir procesamiento en tiempo real.

Para resolver el problema del costo computacional de MonoSLAM, PTAM (Parallel Tracking and Mapping) [11] divide el rastreo y el mapeo en dos hilos de ejecución diferentes en el CPU. Estos dos hilos son ejecutados en paralelo para que el costo computacional del mapeo no afecte al del rastreo. Como resultado de esta separación, es posible usar BA en el mapeo (BA es computacionalmente caro). Esto significa que el rastreo estima el movimiento de la cámara en tiempo real y el mapeo estima la posición 3D de los features con cierto costo computacional. PTAM es el primer algoritmo que

incorpora BA en un algoritmo de SLAM visual de tiempo real.

En el rastreo de PTAM, los puntos mapeados son proyectados en una imagen para hacer correspondencias 2D-3D. De estas correspondencias, podemos calcular la pose de la cámara. En el mapeo, las posiciones 3D de nuevos features son calculadas usando triangulación en ciertos estructuras llamadas *keyframes*. Una contribución importante de PTAM es este mapeo basado en keyframes. Una estructura de entrada es seleccionada como keyframe cuando hay una disparidad grande entre una estructura de entrada y uno de los keyframes que se están midiendo. A diferencia de MonoSLAM, las coordenadas 3D de los features son optimizadas usando BA de manera local con algunos keyframes y de manera global con todos los keyframes del mapa.

ORB-SLAM [18] recoge las ideas principales de PTAM y las combina con un algoritmo de reconocimiento de lugares basado en bolsas de palabras binarias [6], un algoritmo de cierre de lazos robusto ante cambios de escala [27] y el uso de información de covisibiidad para operaciones en gran escala [26].

Una de las principales ideas de diseño de ORB-SLAM es el uso de los mismos features del mapeo y rastreo para el reconocimiento de lugares, relocalización y cierre de lazos. Los features elegidos son los ORB.

En la Figura 3.7 se muestra un diagrama funcional del sistema de ORB-SLAM [18]. El sistema se compone de tres hilos de ejecución que corren en paralelo: rastreo, mapeo local y cerradura de lazo.

El hilo de rastreo está a cargo de localizar a la cámara en cada fotograma y decidir cuándo insertar un nuevo keyframe. Se realiza un emparejamiento de features inicial con el fotograma anterior y se optimiza la pose usando BA únicamente para movimiento. Si se pierde el rastreo, ya sea debido a oclusiones o movimientos abruptos, el módulo de reconocimiento de lugares es utilizado para tratar de relocalizar a la cámara globalmente. Una vez que se tiene una estimación inicial de la pose de la cámara, se recupera un mapa visible local usando la gráfica de covisibilidad de los

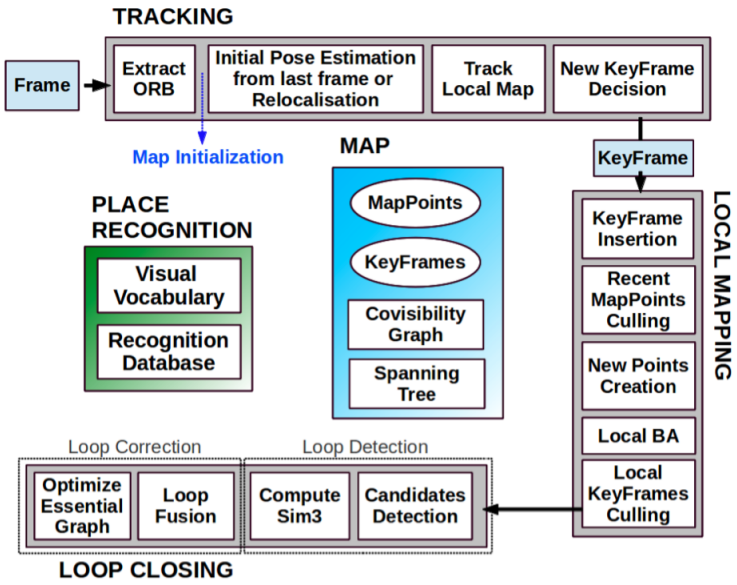


Figura 3.7: Diagrama del sistema de ORB-SLAM [18]

keyframes que mantiene el sistema. Después se buscan coincidencias con los puntos del mapa local usando reproyección y se vuelve a optimizar la pose de la cámara con todas las coincidencias. Finalmente, el hilo de ejecución encargado del rastreo decide si se debe insertar un nuevo keyframe o no.

Para los procesos de mapeo local se procesan nuevos keyframes y se ejecuta un BA local para lograr una reconstrucción óptima en los alrededores de la cámara. Se buscan nuevas correspondencias de features ORB que no han sido emparejados en los keyframes conectados en la gráfica de covisibilidad para triangular nuevos puntos. Basandose en la información recogida durante el rastreo, se aplica una política de eliminación de puntos para conservar únicamente aquellos puntos de alta calidad. El proceso de mapeo local también es responsable de eliminar keyframes redundantes.

El proceso de cerradura de lazo busca lazos con cada nuevo keyframe introducido. Si se detecta un lazo, se calcula una transformación de similitud que informa al sistema sobre la deriva acumulada en la trayectoria. Así, ambos extremos del lazo se alinean y se eliminan puntos duplicados. Finalmente, se realiza una optimización de la pose

para alcanzar consistencia global.

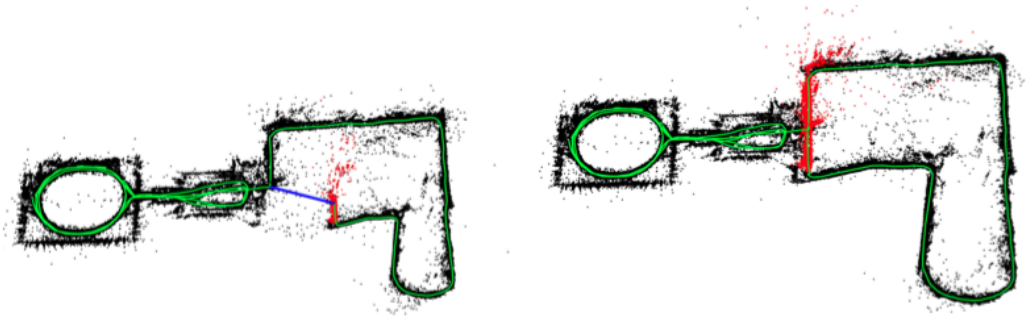


Figura 3.8: Mapas antes y después de cerrar el lazo en la secuencia de datos New-College. El lazo es marcado en azul, la trayectoria en verde y el mapa local en ese momento en rojo [18].

En general, cada punto p del mapa en ORB-SLAM almacena

- Su posición 3D $\mathbf{X}_w$ en coordenadas globales del sistema w
- La dirección de observación $\vec{n}$, que es el vector unitario de la media de todas las direcciones de observación (los vectores que unen el punto con el centro óptico de los keyframes que lo observan)
- Un descriptor ORB representativo $\vec{D}$, que es el descriptor ORB asociado cuya distancia de Hamming es mínima con respecto a todos los otros descriptores asociados en los keyframes en los cuales el punto es observado.
- Las distancias máxima y mínima d_{max} y d_{min} en las cuales el punto puede ser observado, de acuerdo a los límites de invariancia de escala de los features ORB.

A su vez, cada keyframe almacena:

- La pose de la cámara $\mathbf{T}_{cw} \in \mathrm{SO}(3)$, que es una transformación de cuerpo rígido que transforma puntos del ambiente al sistema coordenado de la cámara

- Los parámetros intrínsecos de la cámara, incluidos la distancia focal y el punto principal.

- Todos los features ORB extraidos en el fotograma, asociados o no a un punto en el mapa, cuyas coordenadas no están distorsionadas si es que se provee un modelo de distorción.

Como se ha mencionado anteriormente, para el reconocimiento de lugares se utilizan *bolsas de palabras (bag of words)*. El sistema de ORB-SLAM tiene embebido un módulo de reconocimiento de lugares basado DBoW2 para realizar detección de lazos y relocalización. Las palabras visuales son simplemente una discretización del espacio de descriptores de features, tambien conocido como *vocabulario visual*. En ORB-SLAM, este vocabulario es creado de manera offline con descriptores ORB extraidos de un conjunto de imágenes de entrenamiento. Si las imágenes son lo suficientemente generales, se puede usar el mismo vocabulario para diferentes ambientes y aún así tener un buen desempeño. El sistema construye una base de datos de manera incremental que contiene un índice inverso, el cual almacena, para cada palabra visual en el vocabulario, en qué keyframes fue observada, logrando así consultas eficientes en la base de datos. Cuando se quiere calcular la correspondencia entre dos conjuntos de features ORB, podemos limitar la *fuerza bruta* a encontrar equivalencias únicamente entre aquellos features que pertenecen al mismo nodo en el árbol del vocabulario, acelerando la búsqueda.

V.1 Mapeo Semidenso Probabilístico

Usualmente se cree que los métodos de SLAM directos son mejores y más robustos porque no dependen de la identificación de características en las imagenes, y a su vez que son más precisos pues ocupan más información en su procesamiento. Sin embargo, hay indicios de que esto no siempre es cierto [18]. En su proceso de local-

ización, ORB-SLAM produce mapas poco densos que son muy útiles en la localización de cámaras pero de poco uso para describir semánticamente el ambiente. Con esto en mente, adicional a los tres hilos de ejecución mencionados anteriormente usados para localización y rastreo, se propone un cuarto hilo de ejecución cuya función es la de construir mapas semi-densos que puedan ser utilizados por otros algoritmos, como Octomap, para aumentar sus capacidades de navegación y exploración.

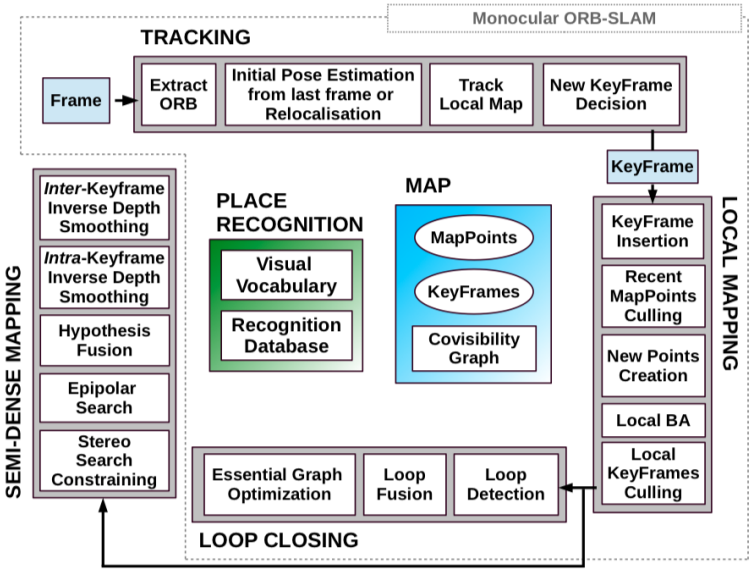


Figura 3.9: Diagrama del sistema de ORB-SLAM con el módulo de mapeo semi-denso [18]

En la Figura 3.9 se muestra el diagrama del sistema de ORB-SLAM con el módulo de mapeo semi-denso. Su funcionamiento se resume de la siguiente forma:

1. Cada keyframe K_i se procesa desde cero. Cada pixel en un área con un alto gradiente es buscado a lo largo de la línea epipolar de los N keyframes vecinos, llevando a N hipótesis de profundidad inversa.

2. Cada hipótesis de profundidad inversa es representada por una distribución gaussiana que toma en cuenta el ruido de la imagen, el paralaje y la ambiguedad

en el emparejamiento. Se considera que las poses del keyframe están bien localizadas, es decir, no se toma en cuenta su incertidumbre.

3. Como la línea base entre keyframes es extensa, el rango de búsqueda a lo largo de la línea epipolar es amplio. Para lidiar con *outliers* en las mediciones, se fusiona el máximo subconjunto de las N hipótesis que son mutuamente compatibles. Cada pixel p del mapa inverso de profundidad es entonces caracterizado por una distribución gaussiana.

4. Se incluye un paso de *suavizado* en el mapa inverso de profundidad de tal forma que cada pixel es promediado con sus vecinos. Si un pixel no es compatible con sus vecinos, es descartado.

5. Después de que se han procesado los respectivos mapas inversos de profundidad de los keyframes vecinos, se verifica la consistencia de las profundidades de cada pixel a lo largo de keyframes vecinos para descartar outliers.

Figura 3.10: Ejemplo de reconstrucción semi-densa. Se puede apreciar la secuencia de keyframes indicando la trayectoria de la cámara [**?**].

Capítulo 4

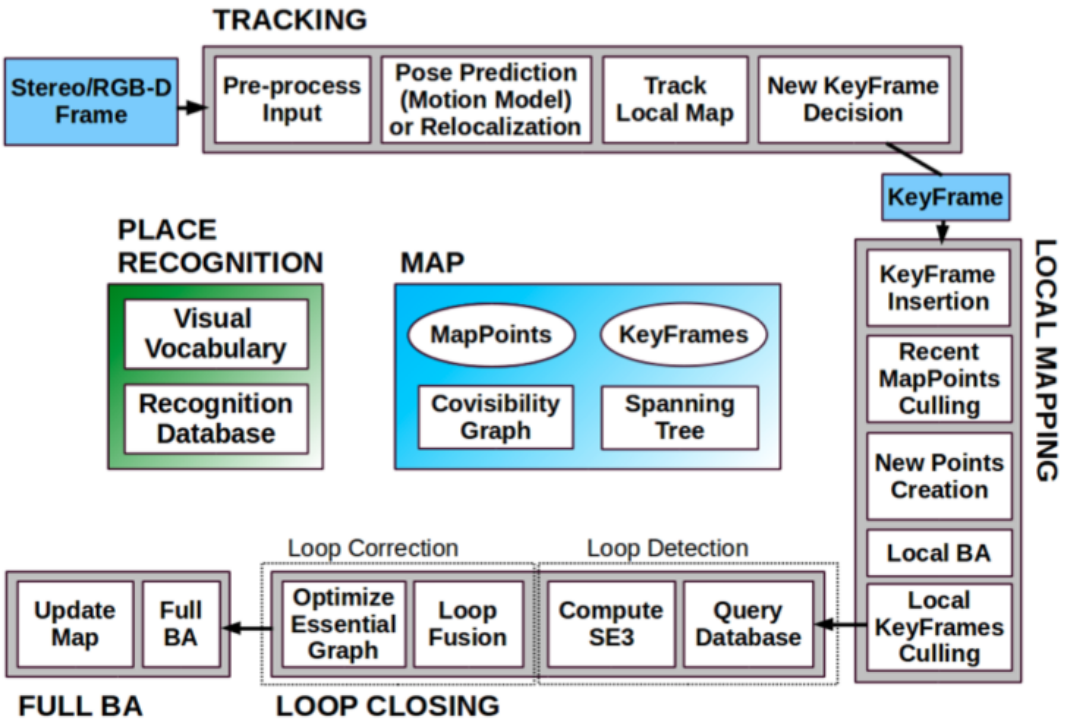
Resultados Experimentales

I RGB-D SLAM

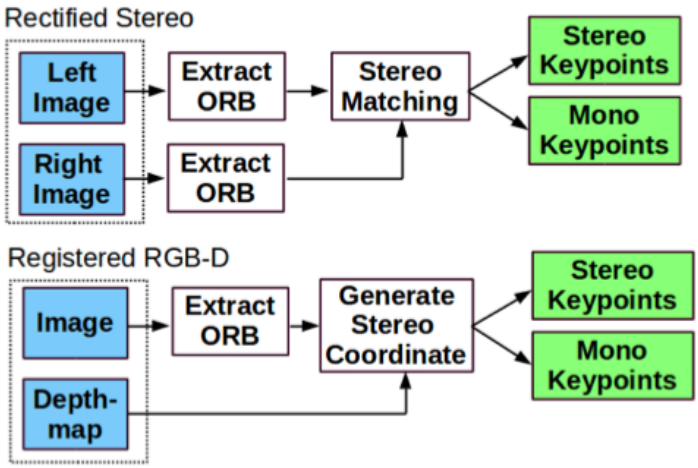
Puesto que no se puede obtener la profundidad de un punto utilizando únicamente una cámara monocular, la escala de un mapa y la trayectoria estimada son desconocidas. Una manera fácil de solventar esto es utilizando cámaras estéreo o, mejor aún, cámaras RGB-D que nos proporcionan una medición de profundidad para cada pixel de la imagen. ORB-SLAM2 tiene la posibilidad de funcionar utilizando cámaras estéreo y RGB-D. En la Figura 4.1 se muestra un overview del sistema.

El sistema tiene tres hilos de ejecución que corren en paralelo: 1) el hilo de rastreo que localiza la cámara en cada fotograma al encontrar coincidencias entre features en el mapa local y minimizando el error de reproyección aplicando BA únicamente para el movimiento, 2) el hilo de mapeo local para administrar y optimizar el mapa local, calculando BA localmente y 3) el hilo de cerradura de lazo detectando lazos y corrigiendo errores de deriva.

El sistema tiene embebido un módulo de reconocimiento de lugares basado en DBoW2 para relocalizar la cámara, en caso de falla, de oclusión y para cerradura de lazos. El sistema mantiene una gráfica de covisibilidad que liga dos fotogramas con



(a) System Threads and Modules.



(b) Input pre-processing

Figura 4.1: Overview del sistema ORB-SLAM2 para cámaras RGB-D y Estéreo [18]

puntos en común. Al igual que el ORB-SLAM2 monocular, el sistema usa features ORB para tareas de rastreo, mapeo y reconocimiento de lugares.

II OctoMap

Varias aplicaciones en robótica, como vehículos aéreos, submarinos, terrestres en interiores y exteriores, requieren un modelo 3D del ambiente. Estos modelos 3D en muchas ocasiones deben de tener una naturaleza probabilística que representen el espacio como vacío, ocupado o sin mapear. Para crear un mapa en 3D del ambiente, el robot sensa su entorno tomando muchas mediciones de rango. Como bien sabemos, cada medición tiene una incertidumbre inherente (típicamente en el orden de centímetros), sin mencionar que la medición puede ser errónea debido a reflexiones por diversos materiales u obstáculos dinámicos. Si queremos crear un mapa confiable, debemos de tomar en cuenta estas incertidumbres. Una forma de hacer esto es fusionando varias mediciones para obtener un estimado robusto de la probabilidad de que cierto volúmen de espacio en el ambiente esté ocupada o no. Otro aspecto importante a considerar es el modelado de zonas que no han sido mapeadas: en navegación autónoma, un robot puede calcular una ruta de navegación en espacios libres únicamente en zonas que ya han sido mapeadas, es decir, sería incorrecto asumir que ciertas zonas son libres sin ninguna medición que lo respalde. Por otro lado, el conocimiento de zonas inexploradas es importante en tareas de exploración autónoma.

OctoMap [8] es un framework open-source para generar modelos volumétricos del ambiente. Este método está basado en *octrees* y usa una estimación de ocupación probabilística. Explícitamente representa no solo el espacio ocupado, sino el espacio libre y espacio desconocido. No es el único método para crear modelos 3D del ambiente. Uno muy conocido es el de Nubes de Puntos, en donde cada punto representa una medición de rango. La desventaja de esta representación es que no es eficiente y

además no permite diferenciar entre zonas libres de obstáculos y áreas no mapeadas, además de que no se pueden fusionar muchas mediciones probabilísticamente.

Un *octree* es una estructura de datos jerárquica para subdivisiones espaciales en 3D. Cada nodo en un octree representa el espacio contenido en un volúmen cúbico, llamado *voxel*. Este volúmen es dividido recursivamente en ocho sub-volúmenes hasta que se llega al tamaño mínimo del voxel (Figura 4.2). El tamaño mínimo del voxel representa la resolución del octree.

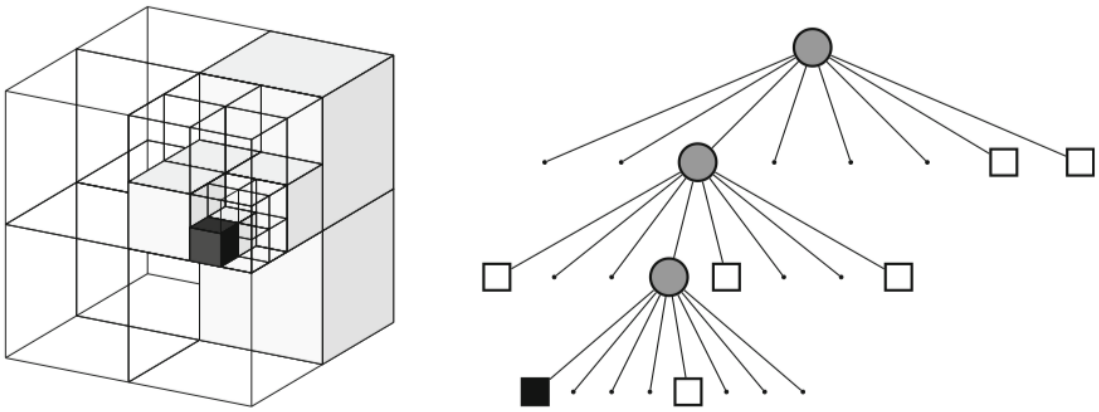


Figura 4.2: Ejemplo de un octree (representación gráfica y representación de árbol) [8]

En su forma mas básica, los octrees pueden usarse para modelar una propiedad booleana. En el contexto que nos corresponde, esta propiedad es la ocupación de un volumen: si cierto volumen es medido como ocupado, se inicializa el nodo correspondiente en el octree. De acuerdo a esta configuración, un nodo no inicializado puede referirse a un espacio libre o no mapeado. Para resolver esta ambiguedad, en OctoMap se representa explícitamente el espacio libre. Se considera espacio libre a toda el área entre el sensor y el punto medido como ocupado, e.g. a lo largo de un rayo determinado con raycasting. El cálculo de la probabilidad $P(n|z_{1:t})$ de que un nodo n este ocupado dadas las mediciones de un sensor $z_{1:t}$ está dada por:

$$P(n|z_{1:t}) = \left[1 + \frac{1-P(n|z_t)}{P(n|z_t)} \frac{1-P(n|z_{1:t-1})}{P(n|z_{1:t.1})} \frac{P(n)}{1-P(n)}\right]^{-1} \tag{4.1}$$

que, expresado en notación log-odds es

$$L(n|z_{1:t}) = L(n|z_{1:t-1} + L(n|z_t)) \tag{4.2}$$

donde

$$L(n) = \log\left[\frac{P(n)}{1-P(n)}\right] \tag{4.3}$$

En estas ecuaciones, el término $P(n|z_t)$ denota la probabilidad de que un voxel n esté ocupado dada una medición z_t.

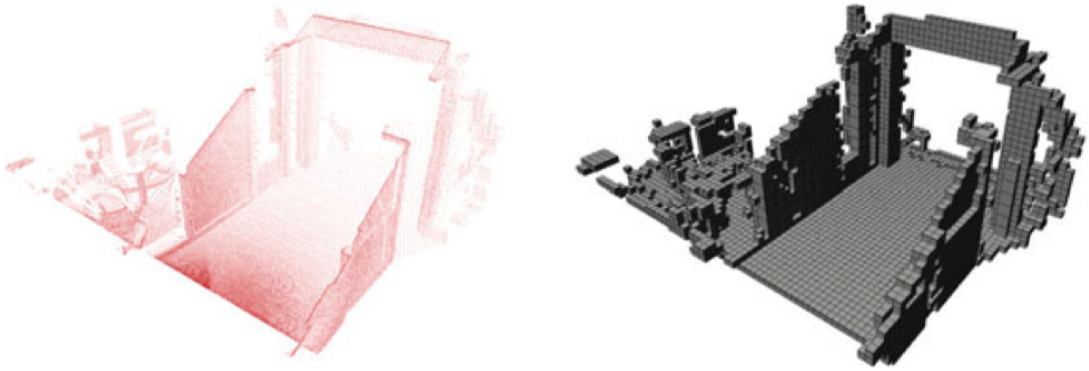


Figura 4.3: Ejemplo de una representación en octrees de una nube de puntos [8]

III ROS (Robot Operating System)

ROS (Robot Operating System) es un framework para escribir software para robots. Es una colección de herramientas, librerías y convenciones cuyo propósito es simplificar la tarea de crear un comportamiento robusto y complejo en una variedad de plataformas robóticas. ROS a veces es considerado un meta sistema operativo porque lleva

a cabo muchas tareas de un sistema operativo, sin embargo requiere de uno (como Linux). Uno de sus principales propósitos es proveer de comunicación entre el usuario, la computadora y equipo externo, como sensores, cámaras y el mismo robot. Como con cualquier SO, el beneficio de ROS es la abstracción del hardware y su habilidad de controlar un robot sin que el usuario conozca los detalles de éste. ROS es open-source y en la actualidad es ocupado por una gran variedad de académicos y en la industria privada. La versión de ROS que se utilizó en esta tesis es ROS Kinetic sobre Linux 16.04.

Para instalar ROS se utilizan los siguientes comandos desde la terminal:

```
$ sudo apt-get update
$ sudo apt-get install ros-kinetic-desktop-full
```

ROS puede depender de paquetes que no están cargados inicialmente. Estos paquetes son externos a ROS y los provee el sistema operativo. El comando `rosdep` se utiliza para descargar e instalar todas estas dependencias:

```
$ sudo rosdep init
$ rosdep update
```

Antes de correr cualquier comando de ROS, nuestra sesión en la terminal debe de estar consciente de todos los archivos de ROS y sus programas. En la instalación de ROS se incluye un script que facilita esta tarea, definiendo las variables de entorno que ROS utiliza:

```
$ source /opt/ros/kinetic/setup.bash
```

Lo siguiente que debemos hacer es crear un espacio de trabajo de catkin. Un espacio de trabajo de catkin (catkin workspace) es un directorio en donde podemos crear o modificar paquetes de catkin. La estructura de catkin simplifica el proceso de compilar e instalar paquetes de ROS. Un espacio de trabajo de catkin puede tener tres o más subdirectorios: `/build`, `/devel` y `/src`. Para crear un espacio de trabajo de catkin usamos los siguientes comandos:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

Para compilar el espacio de trabajo, se usa el siguiente comando:

```
$ cd ~/catkin_ws/
$ catkin_make
```

El comando `catkin_make` crea el espacio de trabajo de catkin. Dentro de la carpeta `/devel` se crean varios archivos del tipo `setup.*sh`. Necesitamos hacer un *source* al archivo `setup.bash` para *cubrir* el workspace default de ROS con este:

```
$ source ~/catkin_ws/devel/setup.bash
```

IV Simulación de OctoMap con Turtlebot

OctoMap está disponible como paquete para ser explotado en ROS. Para instalarlo como librería standalone (sin dependencias de ROS), usa el siguiente comando:

```
$ sudo apt-get install ros-kinetic-octomap
```

Esto significa que puedes usar OctoMap como librería independiente sin necesidad de usar herramientas específicas de ROS o macros catkin. Por conveniencia, la instalación de OctoMap en el sistema incluye archivos de configuracion CMake para poder incluirlos en nuestros `CMakeLists.txt` usando el macro `find_package()`:

```
find_package(octomap REQUIRED)
include_directories(${OCTOMAP_INCLUDE_DIRS})
target_link_libraries(${OCTOMAP_LIBRARIES})
```

Ya instalado OctoMap en nuestro sistema, podemos añadir la dependencia en ROS en nuestro `package.xml` de la siguiente forma:

```
<build_depend>octomap</build_depend>
<run_depend>octomap</run_depend>
```

Turtlebot es una plataforma robótica de bajo coste usado ampliamente por la academia para probar diversos algoritmos. Entre sus muchas características contiene una cámara Kinect (RGB-D) y una computadora con Linux a bordo. Por su popularidad, este robot ha sido modelado y simulado en ROS y Gazebo.

Figura 4.4: Turtlebot 2

Para probar el algoritmo de OctoMap y su construcción de octrees a partir de nubes de puntos generadas por una cámara RGB-D, vamos a configurar un archivo *launch* de ROS que mapee la nube de puntos publicada por la cámara Kinect simulada por Turtlebot a un topic que entienda OctoMap y configurar el servidor OctoMap correspondiente. Se crea un archivo `octomap_turtlebot.launch` que contendrá lo siguiente:

```
<launch>
 <node pkg="octomap_server" type="octomap_server_node"
```

```
                    name="octomap_server">

  <param name="resolution" value="0.05" />
  <param name="frame_id" type="string" value="odom" />
  <param name="sensor_model/max_range" value="5.0" />
  <remap from="cloud_in" to="/camera/depth/points" />

 </node>
</launch>
```

Para ejecutar el archivo que acabamos de llamar, es con el comando

```
$ roslaunch octomap_turtlebot.launch
```

El resultado de la simulación se puede ver en la Figura 4.5. La resolución de cada voxel es de 5cm y para poder construir un marco inercial y mantener la posición del modelo 3D se ocupo la odometría del robot. Al ser un ambiente simulado, la odometría nos proporciona la trayectoria real del robot, no hay errores de deriva. Posteriormente, este marco va a ser calculado con ORB-SLAM2.

V Instalación de cámara RealSense en ROS

La cámara RealSense D435 es una cámara estéreo de profundidad para soluciones de rastreo y mapeo. Cuenta con un proyector infrarrojo y un módulo RGB-D. Es la cámara elegida en esta tesis para la solución de localización y mapeo utilizando ORB-SLAM2 y OctoMap.

Lo primero que necesitamos hacer es instalar el SDK de Intel RealSense. Para esto, seguimos los siguientes pasos: Primero, descargamos los archivos fuentes de las librerias de RealSense

```
$ git clone https://github.com/IntelRealSense/librealsense.git
```

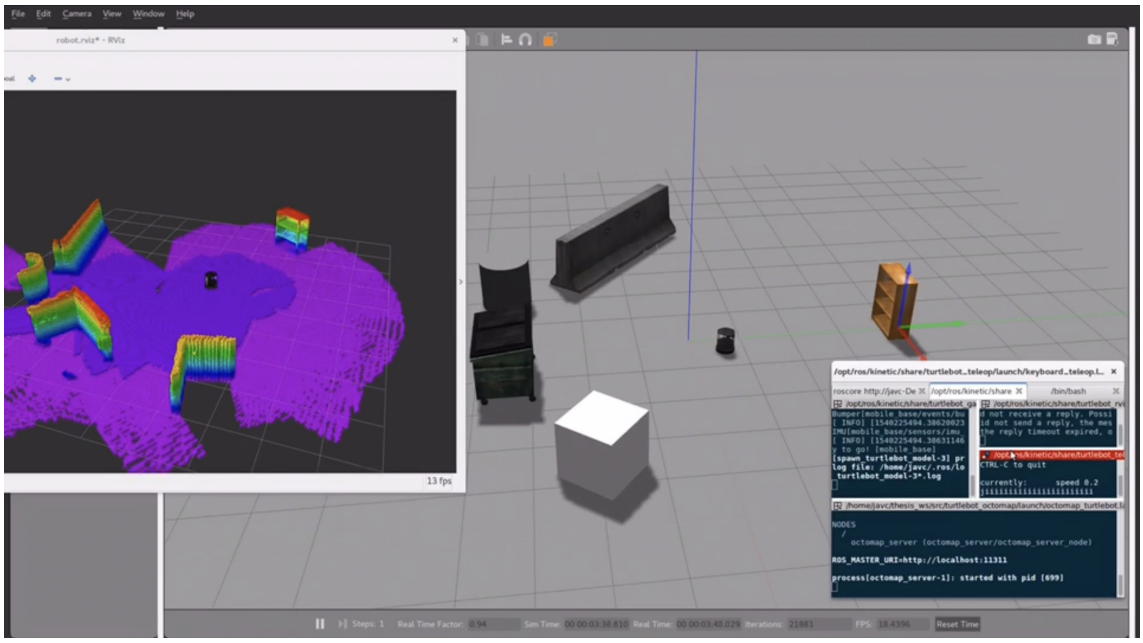


Figura 4.5: Simulación de OctoMap con Turtlebot

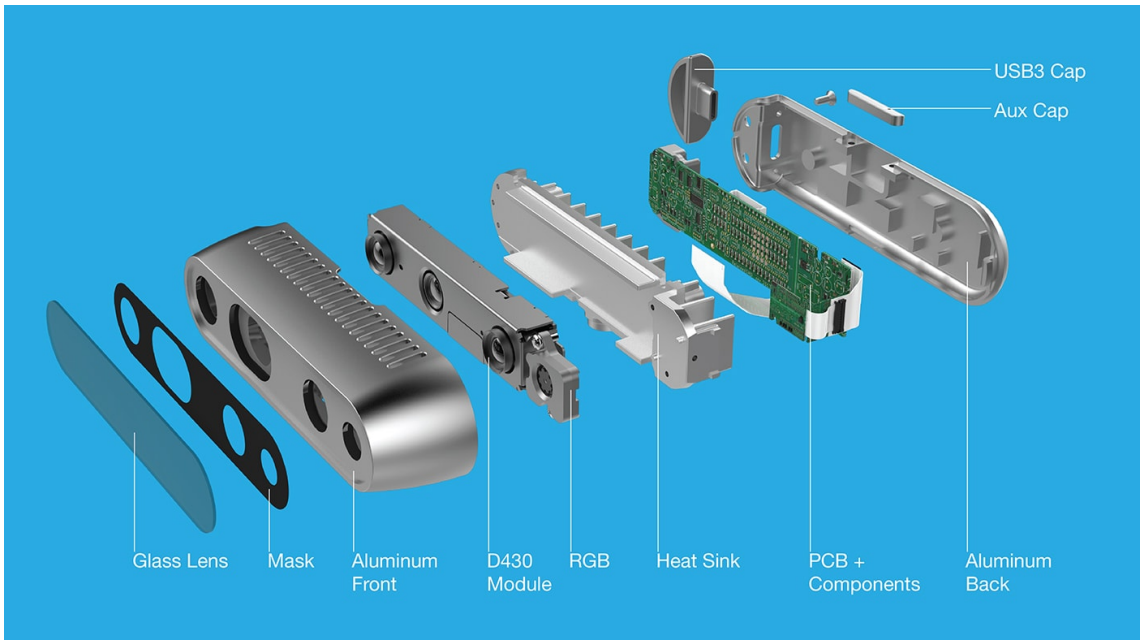


Figura 4.6: Cámara Intel RealSense D435

Después, navegamos a la carpeta raíz para correr los scripts de instalación. Instalamos las dependencias para compilar *librealsense*:

```
sudo apt-get install git
libssl-dev libusb-1.0-0-dev
pkg-config libgtk-3-dev libglfw3-dev
```

Después, corremos un script para configurar permisos

```
$ ./scripts/setup_udev_rules.sh
```

Luego, corremos el siguiente script:

```
$ ./scripts/patch-realsense-ubuntu-lts.sh
```

Finalmente, compilamos el SDK

```
$ mkdir build && cd build
$ cmake ../ -DBUILD_EXAMPLES=true
```

Una vez que hemos instalado el SDK para RealSense, es hora de instalar los paquetes de Intel RealSense en ROS. Dentro de la carpeta `/src` de nuestro espacio de trabajo de catkin, clonamos el último repositorio de Intel RealSense para ROS:

```
$ git clone https://github.com/IntelRealSense/realsense-ros.git
$ cd realsense-ros/
$ git checkout `git tag | sort -V | grep -P "^\d+\.\d+\.\d+" | tail -1`
$ cd ..
```

Después, compilamos nuestro espacio de trabajo:

```
$ catkin_make clean
$ catkin_make -DCATKIN_ENABLE_TESTING=False -DCMAKE_BUILD_TYPE=Release
$ catkin_make install
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Para comenzar a utilizar la cámara, RealSense nos provee de un archivo *launch* que publica en un los topics correspondientes todos los sensores de la cámara. En específico, los topics que publica son:

- /camera/color/camera_info
- /camera/color/image_raw
- /camera/depth/camera_info
- /camera/depth/image_rect_raw
- /camera/extrinsics/depth_to_color
- /camera/extrinsics/depth_to_infra1
- /camera/extrinsics/depth_to_infra2
- /camera/infra1/camera_info
- /camera/infra1/image_rect_raw
- /camera/infra2/camera_info
- /camera/infra2/image_rect_raw
- /camera/gyro/imu_info
- /camera/gyro/sample
- /camera/accel/imu_info
- /camera/accel/sample
- /diagnostics

Para verificar que la instalación fue correcta, podemos utilizar uno de los archivos *launch* que vienen por default y visualizar la nube de puntos publicada por Intel RealSense con rviz. Corre el siguiente comando

```
$ roslaunch realsense2_camera rs_camera.launch filters:=pointcloud
```

Desde rviz puedes ver la nube de puntos publicada.

VI Instalación de ORB-SLAM2 en ROS

Para instalar ORB-SLAM2 en nuestro sistema, debemos de clonar el repositorio en la carpeta `/src` de nuestro espacio de trabajo de catkin:

```
$ git clone https://github.com/raulmur/ORB_SLAM2.git ORB_SLAM2
```

ORB-SLAM2 provee de un script para compilar todas las librerias y dependencias necesarias:

```
$ cd ORB_SLAM2
$ chmod +x build.sh
$ ./build.sh
```

Este último comando creará la librería de ORB-SLAM2 `libORB_SLAM2.so`. Para incluir ORB-SLAM2 a ROS, necesitamos incluir en el *path* de `ROS_PACKAGE_PATH` la variable de ambiente de ORB-SLAM2:

```
$ export ROS_PACKAGE_PATH=${ROS_PACKAGE_PATH}:PATH/ORB_SLAM2/Examples/ROS
```

Luego, compilamos los paquetes de ROS:

```
$ chmod +x build_ros.sh
$ ./build_ros.sh
```

Finalmente, podemos correr el nodo de RGBD de ORB-SLAM2 en ROS con el comando `rosrun` con la siguiente estructura:

```
$ rosrun ORB_SLAM2 RGBD PATH_TO_VOCABULARY PATH_TO_SETTINGS_FILE
```

VII Integración de ORB-SLAM2 con OctoMap e Intel RealSense

La integración de estos 3 actores conlleva varios pasos: primero, debemos de lograr que ORB-SLAM2 trabaje con la información publicada por la cámara de Intel RealSense. Para esto, tenemos que configurar varios parámetros de la cámara y programar las transformadas homogéneas que correspondan para que los marcos coordenados entre ORB-SLAM2 y el de la cámara coincidan. A su vez, debemos de aprovechar la información publicada por la cámara e ir construyendo octrees para el algoritmo de OctoMap. OctoMap necesita un marco de referencia global para poder mantener la consistencia de sus voxels; este marco de referencia será proporcionado por ORB-SLAM2. Al final, tendremos que la cámara RealSense publica en un topic imágenes RGB-D que serán utilizadas por ORB-SLAM2 para las tareas de localización, rastreo, mapeo y cierre de lazo y una nube de puntos que serán utilizados por OctoMap para la construcción de un modelo 3D del ambiente, siendo responsabilidad de ORB-SLAM2 la consistencia de la información entre todos los actores.

Se modificó el archivo fuente de ORB-SLAM2 para RGB-D SLAM de tal forma que considerara la información publicada por la cámara RealSense y actualizara el mapa de octrees, junto con las transformaciones entre marcos coordenados necesarios. El algoritmo es el siguiente:

```
#include <iostream>
#include <algorithm>
#include <fstream>
#include <chrono>

```

```
#include <ros/ros.h>
#include <cv_bridge/cv_bridge.h>
#include <message_filters/subscriber.h>
#include <message_filters/time_synchronizer.h>
#include <message_filters/sync_policies/approximate_time.h>

#include <opencv2/core/core.hpp>

#include "../../../include/System.h"

// Includes for pose publication
#include <ros/console.h>
#include <geometry_msgs/PoseStamped.h>
#include <sensor_msgs/PointCloud2.h>
#include <sensor_msgs/PointCloud.h>
#include <sensor_msgs/point_cloud_conversion.h>

#include <geometry_msgs/Point32.h>
#include <tf/tf.h>
#include <tf/transform_datatypes.h>
#include "../../../include/Converter.h"
#include <tf/transform_broadcaster.h>

#include <octomap/octomap.h>
#include <octomap_ros/conversions.h>
#include <octomap_msgs/Octomap.h>
#include <octomap_msgs/conversions.h>

using namespace std;
using namespace octomap;

class ImageGrabber
{
```

```
39  public:
40  ImageGrabber(ORB_SLAM2::System *pSLAM) : mpSLAM(pSLAM) {}
41
42  void GrabRGBD(const sensor_msgs::ImageConstPtr &msgRGB,
43          const sensor_msgs::ImageConstPtr &msgD);
44
45  ORB_SLAM2::System *mpSLAM;
46 };
47
48 // Declaracion del publisher
49 ros::Publisher pose_pub;
50 ros::Publisher cloud_pub;
51
52 geometry_msgs::PoseStamped pose;
53 sensor_msgs::PointCloud2 cloud;
54 OcTree *tree;
55 string filesPath = "/home/javc/catkin_ws/src/orbslam_octomap/";
56
57 // Operador !
58 bool operator!(const cv::Mat &m) { return m.empty(); }
59
60 void pc_callback(const sensor_msgs::PointCloud2ConstPtr& msg)
61 {
62  cloud = *msg;
63  //cloud.header.frame_id = "camera_color_optical_frame"; // For realsense
64  cloud.header.frame_id = "camera_rgb_optical_frame"; // For simulation
65 }
66
67 /*****************************************************
68 * Update the octomap tree
69 * ****************************************************/
70 void octomapCallback(const octomap_msgs::OctomapConstPtr &msg) {
71
```

```
72   AbstractOcTree *t = octomap_msgs::fullMsgToMap(*msg);
73   if (t)
74   {
75    printf("Casting Octomap...\n");
76    tree = dynamic_cast<OcTree *>(t);
77    tree->setResolution(msg->resolution);
78   }
79
80  }
81
82  int main(int argc, char **argv)
83  {
84   ros::init(argc, argv, "RGBD");
85   ros::start();
86
87   if (argc != 3)
88   {
89    cerr << endl
90    << "Usage: rosrun ORB_SLAM2 RGBD path_to_vocabulary path_to_settings"
91    << endl;
92    ros::shutdown();
93    return 1;
94   }
95
96   // Create SLAM system. It initializes all system threads and gets
97   // ready to process frames.
98   ORB_SLAM2::System SLAM(argv[1], argv[2], ORB_SLAM2::System::RGBD, true);
99
100  ImageGrabber igb(&SLAM);
101
102  ros::NodeHandle nh;
103
104  message_filters::Subscriber<sensor_msgs::Image>
```

```
105      rgb_sub(nh, "/camera/color/image_raw", 1);
106  message_filters::Subscriber<sensor_msgs::Image>
107      depth_sub(nh, "/camera/aligned_depth_to_color/image_raw", 1);
108 ypedef message_filters::sync_policies::ApproximateTime<sensor_msgs::Image,
109                      sensor_msgs::Image> sync_pol;
110 essage_filters::Synchronizer<sync_pol> sync(sync_pol(10),
111                   rgb_sub, depth_sub);
112 ync.registerCallback(boost::bind(&ImageGrabber::GrabRGBD, &igb, _1, _2));
113
114  // Publish topics
115  ROS_INFO("Creating pose publisher...");
116  pose_pub = nh.advertise<geometry_msgs::PoseStamped>("orb_pose", 5);
117  cloud_pub = nh.advertise<sensor_msgs::PointCloud2>("pointcloud", 1);
118
119  // Suscriber to pointcloud
120  ros::Subscriber sub = nh.subscribe("/camera/rgb/points",
121                1000, pc_callback);
122  // Subscriber to Octomap to save file
123  ros::Subscriber octomap_sub = nh.subscribe("/octomap_full", 1,
124                     octomapCallback);
125
126  ros::spin();
127
128  // Stop all threads
129  SLAM.Shutdown();
130  if (tree)
131  {
132   printf("Writing map file...\n");
133   tree->writeBinary(filesPath + "map.bt");
134   printf("Map Saved! \n");
135  }
136
137  printf("Saving KeyFrameTrajectory and ORB Trajectory\n");
```

```
138  // Save camera trajectory
139  SLAM.SaveKeyFrameTrajectoryTUM("KeyFrameTrajectory.txt");
140  SLAM.SaveTrajectoryTUM("ORB_Trajectory.txt");
141
142  ros::shutdown();
143
144  return 0;
145 }
146
147 void ImageGrabber::GrabRGBD(const sensor_msgs::ImageConstPtr &msgRGB,
148                 const sensor_msgs::ImageConstPtr &msgD)
149 {
150  // Copy the ros image message to cv::Mat.
151  cv_bridge::CvImageConstPtr cv_ptrRGB;
152  try
153  {
154   cv_ptrRGB = cv_bridge::toCvShare(msgRGB);
155  }
156  catch (cv_bridge::Exception &e)
157  {
158   ROS_ERROR("cv_bridge exception: %s", e.what());
159   return;
160  }
161
162  bool test = 0;
163
164  cv_bridge::CvImageConstPtr cv_ptrD;
165  try
166  {
167   cv_ptrD = cv_bridge::toCvShare(msgD);
168  }
169   catch (cv_bridge::Exception &e)
170  {
```

```
171   ROS_ERROR("cv_bridge exception: %s", e.what());
172   return;
173  }
174
175  cv::Mat Tcw = mpSLAM->TrackRGBD(cv_ptrRGB
176              ->image, cv_ptrD->image, cv_ptrRGB
177              ->header.stamp.toSec());
178
179  pose.header.stamp = ros::Time::now();
180  pose.header.frame_id = "odom";
181
182  test = !Tcw;
183
184  if (test == 1)
185  {
186   ROS_INFO("SLAM lost! \n");
187   ros::param::set("/octomap_server/resolution", 0);
188   pose_pub.publish(pose);
189   sensor_msgs::PointCloud2 empty_pcl;
190   cloud_pub.publish(empty_pcl);
191  }
192  else
193  {
194   ros::param::set("/octomap_server/resolution", 0.05);
195   // Pose information
196   // Rotation information
197   cv::Mat Rwc = Tcw.rowRange(0, 3).colRange(0, 3).t();
198   // translation information
199   cv::Mat twc = -Rwc * Tcw.rowRange(0, 3).col(3);
200
201   vector<float> q = ORB_SLAM2::Converter::toQuaternion(Rwc);
202
203   // Transform from camera to robot
```

```
204   tf::Transform new_transform;
205   tf::Quaternion quaternion(q[0], q[1], q[2], q[3]);
206
207   // Quaternion for rotation
208   tf::Quaternion q_new, qq;
209   // Rotations (-90deg z, then -90deg in x) World--->Camera_pose
210   tf::Quaternion q1(0, 0, -0.7071, 0.7071);
211   tf::Quaternion q2(-0.7071, 0, 0, 0.7071);
212   qq = q1*q2;
213   q_new = qq*quaternion;
214   //Normalize the new quaternion
215   q_new.normalize();
216   new_transform.setRotation(q_new);
217   new_transform.setOrigin(tf::Vector3(twc.at<float>(0, 2),
218                 -twc.at<float>(0, 0),
219                 -twc.at<float>(0, 1)));
220
221   static tf::TransformBroadcaster br;
222   // For simulation
223   br.sendTransform(tf::StampedTransform(new_transform, ros::Time::now(),
224                   "odom", "camera_rgb_optical_frame"));
225
226   tf::poseTFToMsg(new_transform, pose.pose);
227   pose_pub.publish(pose);
228   cloud_pub.publish(cloud);
229
230  }
231 }
```

Para correr el nodo de RGBD de ORB-SLAM2 necesitamos dos cosas: el vocabulario (bag of words) y un archivo de configuración de parámetros de la cámara. En cuanto al vocabulario, ORB-SLAM incluye uno por defecto con suficientes features para poder realizar la localización y el reconocimiento de lugares de manera efectiva. En el caso

de la cámara RealSense D435, el archivo de configuración utilizado en esta tesis es el siguiente:

```
%YAML:1.0

#--------------------------------------------------------------------------------------------
# Camera Parameters. Adjust them!
#--------------------------------------------------------------------------------------------

# Camera calibration and distortion parameters (OpenCV)
Camera.fx: 613.8817749023438 #385.1329345703125
Camera.fy: 614.3071899414062 #385.1329345703125 #
Camera.cx: 324.786376953125 #320.6764221191406 #
Camera.cy: 255.0802764892578 #241.52432250976562 #255.0802764892578

Camera.k1: 0.0
Camera.k2: 0.0
Camera.p1: 0.0
Camera.p2: 0.0
Camera.k3: 0.0

Camera.width: 640
Camera.height: 480

# Camera frames per second
Camera.fps: 30.0

# IR projector baseline times fx (aprox.)
#Camera.bf: 24
Camera.bf: 30.69 #30.69 21.18 17.2

# Color order of the images (0: BGR, 1: RGB. It is ignored if images
# are grayscale)
Camera.RGB: 1
```

```
32
33 # Close/Far threshold. Baseline times.
34 ThDepth: 55.0
35
36 # Deptmap values factor
37 DepthMapFactor: 1000.0
38
39 #--------------------------------------------------------------------------
40 # ORB Parameters
41 #--------------------------------------------------------------------------
42
43 # ORB Extractor: Number of features per image
44 ORBextractor.nFeatures: 1000
45
46 # ORB Extractor: Scale factor between levels in the scale pyramid
47 ORBextractor.scaleFactor: 1.2
48
49 # ORB Extractor: Number of levels in the scale pyramid
50 ORBextractor.nLevels: 8
51
52 # ORB Extractor: Fast threshold
53 # Image is divided in a grid. At each cell FAST are extracted imposing a
54 # minimum response.
55 # Firstly we impose iniThFAST. If no corners are detected we impose a
56 # lower value minThFAST
57 # You can lower these values if your images have low contrast
58 ORBextractor.iniThFAST: 20
59 ORBextractor.minThFAST: 7
60
61 #--------------------------------------------------------------------------
62 # Viewer Parameters
63 #--------------------------------------------------------------------------
64 Viewer.KeyFrameSize: 0.05
```

```
Viewer.KeyFrameLineWidth: 1
Viewer.GraphLineWidth: 0.9
Viewer.PointSize:2
Viewer.CameraSize: 0.08
Viewer.CameraLineWidth: 3
Viewer.ViewpointX: 0
Viewer.ViewpointY: -0.7
Viewer.ViewpointZ: -1.8
Viewer.ViewpointF: 500
```

Para compilar el nuevo nodo de ORB-SLAM2 para RGBD que hemos modificado, hacemos uso de un script incluido en el mismo paquete:

```
$ ./src/ORB_SLAM2/build_ros_nodes.sh
```

Ya que está compilado, podemos correr el nodo de ORB-SLAM2 para RGBD utilizando el comando `rosrun` de ROS, apuntando al archivo de configuración de la cámara [1]:

```
$ rosrun ORB_SLAM2 RGBD
  /home/javc/thesis_ws/src/ORB_SLAM2/Vocabulary/ORBVoc.txt
  /home/javc/thesis_ws/src/ORB_SLAM2/Examples/RGB-D/rs_d435.yaml
```

Una vez que el nodo RGBD de ORB-SLAM2 está corriendo, necesitamos que la cámara RealSense comience a publicar las imágenes junto con la información de profundidad. El paquete de RealSense para ROS incluye algunos archivos *launch* con configuración por defecto, sin embargo, necesitamos hacer un par de modificaciones para la publicación de nubes de puntos, confiugración de parámetros de la cámara, rectificación de imágenes estéreo y los topics a los que publica. El archivo *launch* `rs_rgb2.launch` para la cámara RealSense es el siguiente:

```
<launch>

```

[1] El directorio al que apunta dependerá del espacio de trabajo que hayas definido para catkin

```
  <arg name="camera"              default="camera"/>
  <arg name="tf_prefix"           default="$(arg camera)"/>
  <arg name="manager"             default="realsense2_camera_manager"/>

  <!-- Camera device specific arguments -->

  <arg name="serial_no"           default=""/>
  <arg name="json_file_path"      default=""/>

  <arg name="fisheye_width"       default="640"/>
  <arg name="fisheye_height"      default="480"/>
  <arg name="enable_fisheye"      default="true"/>

  <arg name="depth_width"         default="640"/>
  <arg name="depth_height"        default="480"/>
  <arg name="enable_depth"        default="true"/>

  <arg name="infra1_width"        default="640"/>
  <arg name="infra1_height"       default="480"/>
  <arg name="enable_infra1"       default="true"/>

  <arg name="infra2_width"        default="640"/>
  <arg name="infra2_height"       default="480"/>
  <arg name="enable_infra2"       default="true"/>

  <arg name="color_width"         default="640"/>
  <arg name="color_height"        default="480"/>
  <arg name="enable_color"        default="true"/>

  <arg name="fisheye_fps"         default="30"/>
  <arg name="depth_fps"           default="30"/>
  <arg name="infra1_fps"          default="30"/>
  <arg name="infra2_fps"          default="30"/>
```

```
36  <arg name="color_fps"           default="30"/>
37  <arg name="gyro_fps"            default="1000"/>
38  <arg name="accel_fps"           default="1000"/>
39  <arg name="enable_imu"          default="true"/>
40
41  <arg name="enable_pointcloud"   default="false"/>
42  <arg name="enable_sync"         default="true"/>
43  <arg name="align_depth"         default="true"/>
44
45  <!-- rgbd_launch specific arguments -->
46
47  <!-- Arguments for remapping all device namespaces -->
48  <arg name="rgb"                             default="color" />
49  <arg name="ir"                              default="infra1" />
50  <arg name="depth"                           default="depth" />
51  <arg name="depth_registered_pub"            default="rgb" />
52  <arg name="depth_registered"            default="depth_registered"
53                  unless="$(arg align_depth)"/>
54  <arg name="depth_registered"        default="aligned_depth_to_color"
55                  if="$(arg align_depth)" />
56  <arg name="depth_registered_filtered"
57     default="$(arg depth_registered)"/>
58  <arg name="projector"                       default="projector" />
59
60  <!-- Disable bond topics by default -->
61  <arg name="bond"                            default="false" />
62  <arg name="respawn"                         default="$(arg bond)" />
63
64  <!-- Processing Modules -->
65  <arg name="rgb_processing"                  default="true"/>
66  <arg name="debayer_processing"              default="false" />
67  <arg name="ir_processing"                   default="false"/>
68  <arg name="depth_processing"                default="false"/>
```

```
69    <arg name="depth_registered_processing"      default="true"/>
70    <arg name="disparity_processing"             default="false"/>
71    <arg name="disparity_registered_processing" default="false"/>
72    <arg name="hw_registered_processing"     default="$(arg align_depth)" />
73    <arg name="sw_registered_processing"     default="true"
74                      unless="$(arg align_depth)" />
75    <arg name="sw_registered_processing"         default="false"
76                       if="$(arg align_depth)" />
77
78    <group ns="$(arg camera)">
79
80     <!-- Launch the camera device nodelet -->
81  <include
82     file="$(find realsense2_camera)/launch/includes/nodelet.launch.xml">
83    <arg name="manager"                  value="$(arg manager)"/>
84    <arg name="tf_prefix"                value="$(arg tf_prefix)"/>
85    <arg name="serial_no"                value="$(arg serial_no)"/>
86    <arg name="json_file_path"           value="$(arg json_file_path)"/>
87
88    <arg name="enable_pointcloud"    value="$(arg enable_pointcloud)"/>
89    <arg name="enable_sync"              value="$(arg enable_sync)"/>
90    <arg name="align_depth"              value="$(arg align_depth)"/>
91
92    <arg name="fisheye_width"            value="$(arg fisheye_width)"/>
93    <arg name="fisheye_height"           value="$(arg fisheye_height)"/>
94    <arg name="enable_fisheye"           value="$(arg enable_fisheye)"/>
95
96    <arg name="depth_width"              value="$(arg depth_width)"/>
97    <arg name="depth_height"             value="$(arg depth_height)"/>
98    <arg name="enable_depth"             value="$(arg enable_depth)"/>
99
100   <arg name="color_width"              value="$(arg color_width)"/>
101   <arg name="color_height"             value="$(arg color_height)"/>
```

```
102   <arg name="enable_color"              value="$(arg enable_color)"/>
103
104   <arg name="infra1_width"              value="$(arg infra1_width)"/>
105   <arg name="infra1_height"             value="$(arg infra1_height)"/>
106   <arg name="enable_infra1"             value="$(arg enable_infra1)"/>
107
108   <arg name="infra2_width"              value="$(arg infra2_width)"/>
109   <arg name="infra2_height"             value="$(arg infra2_height)"/>
110   <arg name="enable_infra2"             value="$(arg enable_infra2)"/>
111
112   <arg name="fisheye_fps"               value="$(arg fisheye_fps)"/>
113   <arg name="depth_fps"                 value="$(arg depth_fps)"/>
114   <arg name="infra1_fps"                value="$(arg infra1_fps)"/>
115   <arg name="infra2_fps"                value="$(arg infra2_fps)"/>
116   <arg name="color_fps"                 value="$(arg color_fps)"/>
117   <arg name="gyro_fps"                  value="$(arg gyro_fps)"/>
118   <arg name="accel_fps"                 value="$(arg accel_fps)"/>
119   <arg name="enable_imu"                value="$(arg enable_imu)"/>
120 </include>
121
122   <!-- RGB processing -->
123   <include if="$(arg rgb_processing)"
124     file="$(find rgbd_launch)/launch/includes/rgb.launch.xml">
125     <arg name="manager"                       value="$(arg manager)" />
126     <arg name="respawn"                       value="$(arg respawn)" />
127     <arg name="rgb"                           value="$(arg rgb)" />
128     <arg name="debayer_processing"    value="$(arg debayer_processing)"/>
129   </include>
130
131   <group
132     if="$(eval depth_registered_processing
133       and
134       sw_registered_processing)">
```

```
135   <node pkg="nodelet" type="nodelet" name="register_depth"
136    args="load depth_image_proc/register $(arg manager) $(arg bond)"
137      respawn="$(arg respawn)">
138    <remap from="rgb/camera_info"      to="$(arg rgb)/camera_info"/>
139    <remap from="depth/camera_info"    to="$(arg depth)/camera_info"/>
140    <remap from="depth/image_rect"     to="$(arg depth)/image_rect_raw"/>
141    <remap from="depth_registered/image_rect"
142       to="$(arg depth_registered)/sw_registered/image_rect_raw" />
143    </node>
144
145    <!-- Publish registered XYZRGB point cloud
146    with software registered input -->
147    <node pkg="nodelet" type="nodelet" name="points_xyzrgb_sw_registered"
148   args="load depth_image_proc/point_cloud_xyzrgb
149     $(arg manager) $(arg bond)"
150      respawn="$(arg respawn)">
151    <remap from="rgb/image_rect_color"  to="$(arg rgb)/image_rect_color"/>
152    <remap from="rgb/camera_info"       to="$(arg rgb)/camera_info" />
153    <remap from="depth_registered/image_rect"
154    to="$(arg depth_registered_filtered)/sw_registered/image_rect_raw"/>
155    <remap from="depth_registered/points"
156      to="$(arg depth_registered)/points"/>
157    </node>
158   </group>
159
160   <group
161    if="$(eval depth_registered_processing and hw_registered_processing)">
162    <!-- Publish registered XYZRGB point cloud with hardware
163     registered input (ROS Realsense depth alignment) -->
164    <node pkg="nodelet" type="nodelet" name="points_xyzrgb_hw_registered"
165    args="load depth_image_proc/point_cloud_xyzrgb
166      $(arg manager) $(arg bond)"
167           respawn="$(arg respawn)">
```

```
    <remap from="rgb/image_rect_color"
        to="$(arg rgb)/image_rect_color" />
    <remap from="rgb/camera_info"
      to="$(arg rgb)/camera_info" />
    <remap from="depth_registered/image_rect"
      to="$(arg depth_registered)/image_raw" />
    <remap from="depth_registered/points"
      to="$(arg depth_registered_pub)/points" />
    </node>
   </group>

   </group>

 </launch>
```

Para ejecutar este archivo, usamos el comando `roslaunch` de ROS:

```
$ roslaunch realsense2_camera rs_rgb2.launch
```

El último paso es correr el servidor y nodo de OctoMap. Al igual que con la cámara RealSense, OctoMap nos proporciona de un archivo *launch* por defecto con configuraciones básicas. Para facilitar el lanzamiento de los 3 sistemas (ORB-SLAM2, RealSense y OctoMap) se creó un archivo *launch*, `orbslam_octomap.launch` que junta los archivos mencionados anteriormente y ejecuta todo de un solo comando. Este archivo también incluye la configuración de OctoMap, como el mapeo de los topics de nubes de puntos, la resolución del voxel y el marco coordenado que tomará como referencia (el proporcionado por ORB-SLAM2).

```
<launch>
 <!-- Launch ORB_SLAM2 -->
 <node name="ORB_SLAM2_RGBD"
     pkg="ORB_SLAM2" type="RGBD"
     args="$(find orbslam_octomap)/Vocabulary/ORBvoc.txt
       $(find orbslam_octomap)/Configurations/rs_d435.yaml" />
```

```
<!-- Launch Realsense Camera -->
<include file="$(find realsense2_camera)/launch/rs_rgbd2.launch"/>
<!-- Launch Octomap -->
<node pkg="octomap_server"
 type="octomap_server_node"
 name="octomap_server">
 <!-- <remap from="cloud_in" to="/camera/rgb/points" /> -->
 <remap from="cloud_in" to="/pointcloud" />
 <param name="resolution" value="0.1" />
 <param name="frame_id" type="string" value="world" />
 <param name="sensor_model/max_range" value="4.0" />
 <param name="filter_ground" value="false" />
</node>
</launch>
```

En la Figura 4.7 se puede ver el resultado de la integración de los 3 sistemas en un ambiente interior. Del lado inferior izquierdo, se muestran los keyframes de ORB-SLAM2 para el rastreo. En el lado superior izquierdo se resaltan los features detectados por ORB-SLAM2 y que servirán para el rastreo, mapeo y localización de la cámara. Del lado derecho vemos la reconstrucción del ambiente con OctoMap, donde cada cubo representa un voxel. En la representación volumétrica, únicamente se resalta el espacio ocupado. Cabe destaar que la reconstrucción se hizo en tiempo real. Al tratarse de un algoritmo probabilístico, la reconstrucción es cada vez mejor conforme se tengan más muestras. Con cada medición de rango aumenta la certidumbre de que cierto voxel esté ocupado o libre.

El comportamiento en exteriores se muestra en la Figura 4.8. La resolución del voxel y la calidad del rastreo depende de varios factores: por un lado, se trata de una cámara RGB-D que mide la profundidad a partir de la proyección de una matriz de puntos infrarrojos, por lo que puede fallar en superficies reflejantes o en zonas con alto brillo. En cuanto a ORB-SLAM, al depender de la identificación de features, su desem-

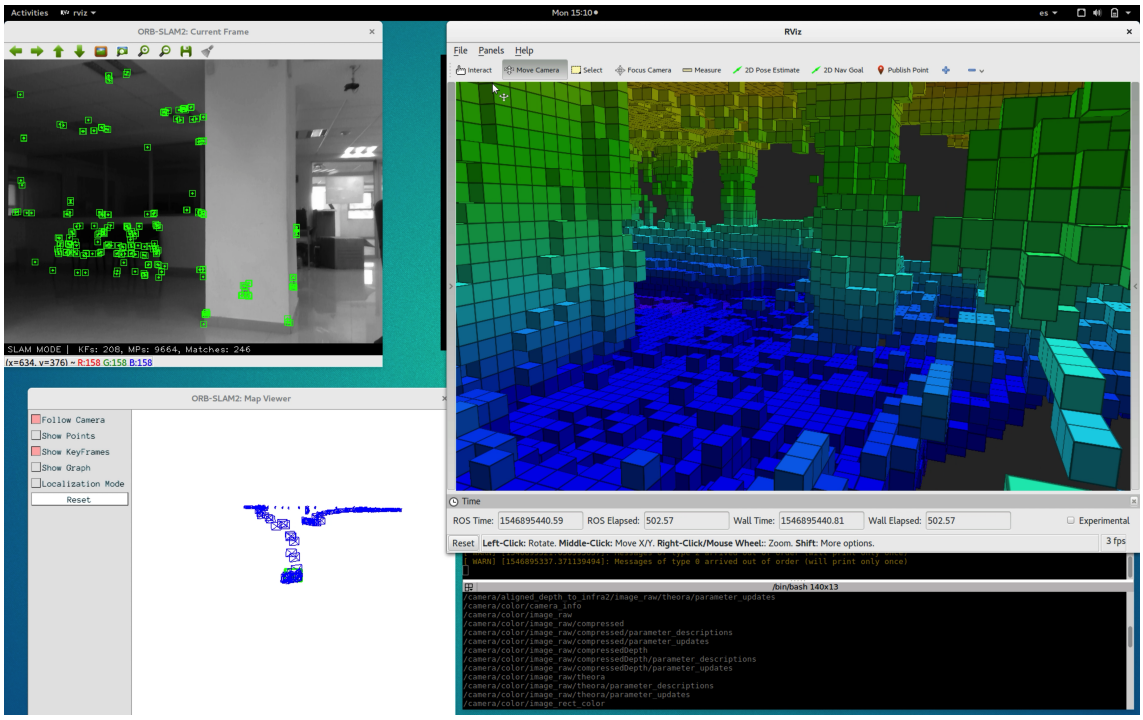


Figura 4.7: Integración de ORB-SLAM2 con cámara RGB-D y OctoMap en interiores

peño está directamente ligado a la identificación de características como esquinas o bordes; en imágenes muy homogéneas con poca textura, el rastreo puede no ser el más óptimo. Sin embargo, con todas estas consideraciones, el resultado es aceptable para tareas de navegación y exploración ya que, como se puede ver en la imagen, el espacio ocupado, libre y no explorado quedan muy bien representados.

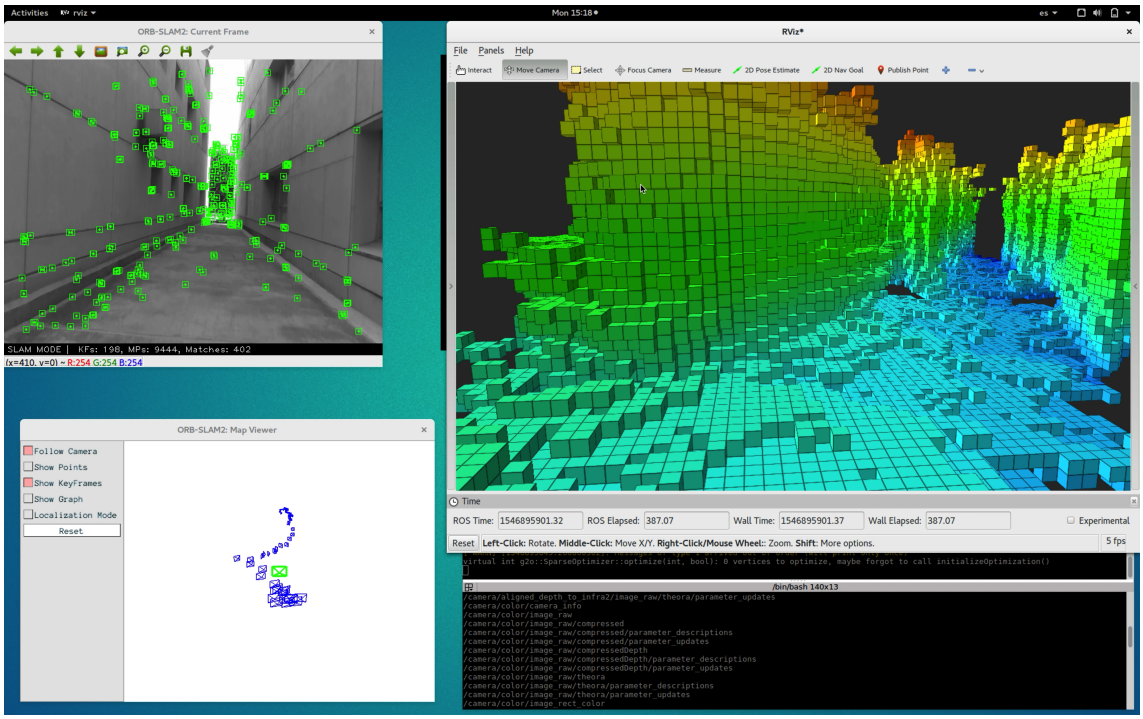


Figura 4.8: Integración de ORB-SLAM2 con cámara RGB-D y OctoMap en exteriores

Capítulo 5

Conclusiones

En esta tesis se abordó el problema de SLAM, localización y mapeo simultáneos, uno de los problemas más populares y relevantes de la robótica moderna. Se habló de las diferentes formas de resolver este problema, los distintos algoritmos que existen así como sus ventajas y desventajas. Nos enfocamos en el SLAM Visual, una familia de algoritmos que busca resolver el problema del SLAM usando como sensor primario una cámara (monocular, estéreo o RGBD). Se explicaron los fundamentos de la visión por computadora y cómo se utilizan para resolver tareas de localización de una cámara. También se habló de algoritmos modernos para la localización y reconocimiento de lugares. Finalmente, se estudió uno de los algoritmos de SLAM visual más populares, ORB-SLAM2 y se vio cómo integrarlo con un algoritmo de modelado en 3D basado en octrees, OctoMap, utilizando el framework ROS. El resultado final es un sistema de Localización y Mapeo Simultáneos con capacidad de generar una representación volumétrica del ambiente, indicando explícitamente el espacio libre, ocupado e inexplorado, todo esto en tiempo real y utilizando tecnolgías open-source. El sensor utilizado fue una cámara RGB-D de Intel de bajo costo y alto rendimiento, que se mostró muy capaz tanto en interiores como en exteriores.

El trabajo desarrollado servirá para desarrollar plataformas autónomas e imple-

mentar algoritmos de exploración y navegación, todo de manera visual. Como trabajo futuro, se pueden fusionar distintas fuentes de datos, e.g. cámara RGB-D con un LiDAR y así aumentar la precisión de la localización y el mapeo. El hecho de poder localizar una cámara y construir una representación volumétrica del ambiente sin utilizar mecanismos de localización convencionales como el GPS o brújulas supone una gran ventaja y extiende las posibilidades de la robótica a entornos nunca antes vistos, desde cuevas y túneles hasta otros planetas.

Bibliografía

[1] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. BRIEF: Binary Robust Independent Elementary Features. Technical report.

[2] Qi Cheng and Pascal Bondon. A new unscented particle filter. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pages 3417–3420, 2008.

[3] Andrew J. Davison. Real-time simultaneous localisation and mapping with a single camera. *Proceedings of the IEEE International Conference on Computer Vision*, 2:1403–1410, 2003.

[4] Andrew J. Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proceedings of the IEEE International Conference on Computer Vision*, volume 2, pages 1403–1410, 2003.

[5] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. MonoSLAM: Real-time single camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, jun 2007.

[6] Dorian Gálvez-López and Juan D. Tardós. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012.

[7] Bo He, Lulu Ying, Shujing Zhang, Xiao Feng, Tianhong Yan, Rui Nian, and Yue Shen. Autonomous navigation based on unscented-FastSLAM using particle swarm optimization for autonomous underwater vehicles. *Measurement: Journal of the International Measurement Confederation*, 71:89–101, 2015.

[8] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[9] Hyunggi Jo, Hae Min Cho, Sungjin Jo, and Euntai Kim. Efficient Grid-Based Rao-Blackwellized Particle Filter SLAM with Interparticle Map Sharing. *IEEE/ASME Transactions on Mechatronics*, 23(2):714–724, 2018.

[10] Hyunggi Jo, Sungjin Jo, Euntai Kim, Changyong Yoon, and Sewoong Jun. 3D FastSLAM algorithm with Kinect sensor. *2014 Joint 7th International Conference on Soft Computing and Intelligent Systems, SCIS 2014 and 15th International Symposium on Advanced Intelligent Systems, ISIS 2014*, pages 214–217, 2014.

[11] Georg Klein and David Murray. Parallel tracking and mapping for small AR workspaces. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality, ISMAR*, 2007.

[12] George Klein and David Murray. Parallel Tracking and Mapping for Small AR Workspaces. *Proceedings of the 20th International Meshing Roundtable, IMR 2011*, pages 125–141, 2011.

[13] Zeyneb Kurt-Yavuz and Sirma Yavuz. A comparison of EKF, UKF, FastSLAM2.0, and UKF-based FastSLAM algorithms. *INES 2012 - IEEE 16th International Conference on Intelligent Engineering Systems, Proceedings*, pages 37–43, 2012.

[14] Yao Li, Fan Zhun, Zhu Guijie, Li Wenji, Li Chong, Wang Yupeng, and Xie Honghui. A SLAM with simultaneous construction of 2D and 3D maps based on Rao-

Blackwellized particle filters. *Proceedings - 2018 10th International Conference on Advanced Computational Intelligence, ICACI 2018*, pages 374–378, 2018.

[15] Jie Luo, Lei Sun, and Yunhui Jia. A new FastSLAM algorithm based on the unscented particle filter. *Proceedings of the 30th Chinese Control and Decision Conference, CCDC 2018*, pages 1259–1263, 2018.

[16] Qiang Lv, Huican Lin, Guosheng Wang, Heng Wei, and Yang Wang. ORB-SLAM-based tracing and 3D reconstruction for robot using Kinect 2.0. *Proceedings of the 29th Chinese Control and Decision Conference, CCDC 2017*, pages 3319–3324, 2017.

[17] Michael Montemerlo, Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. *IN PROCEEDINGS OF THE AAAI NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 593—-598, 2002.

[18] Raúl Mur-Artal. *Real-Time Accurate Visual SLAM with Place Recognition*. PhD thesis, Universidad de Zaragoza, 2017.

[19] Raul Mur-Artal, J. M.M. Montiel, and Juan D. Tardos. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[20] J.D. Mur-Artal, R. and Tardos. ORB-SLAM2 An Open-Source SLAM System for Monocular Stereo.pdf. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[21] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3951 LNCS, pages 430–443. Springer Verlag, 2006.

[22] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. ORB: An efficient alternative to SIFT or SURF. *Proceedings of the IEEE International Conference on Computer Vision*, pages 2564–2571, 2011.

[23] Jianli Shi, Yuqiang Wu, Shuang Pan, and Xibin Wang. Unscented FastSLAM for UAV. *Proceedings of 2011 International Conference on Computer Science and Network Technology, ICCSNT 2011*, 4:2529–2532, 2011.

[24] Roland Siegwart, Illah Reza Nourbakhsh, Davide Scaramuzza, and Roland. Siegwart. *Introduction to autonomous mobile robots*, volume 49. MIT Press, 2nd edition, 2011.

[25] Robert Sim and James J. Little. Autonomous vision-based exploration and mapping using hybrid maps and Rao-Blackwellised particle filters. *IEEE International Conference on Intelligent Robots and Systems*, pages 2082–2089, 2006.

[26] Hauke Strasdat, Andrew J. Davison, J. M.M. Montiel, and Kurt Konolige. Double window optimisation for constant time visual SLAM. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2352–2359, 2011.

[27] Hauke Strasdat, J M M Montiel, and Andrew J Davison. Scale Drift-Aware Large Scale Monocular SLAM. Technical report.

[28] Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. Visual SLAM algorithms: a survey from 2010 to 2016. *IPSJ Transactions on Computer Vision and Applications*, 9(1), 2017.

[29] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*, volume 45. MIT Press, 2005.

[30] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment – a modern synthesis. In *Lecture Notes in Computer Science*

(including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 1883, pages 298–372. Springer Verlag, 2000.

[31] Huayou Wang, Yanmig Hu, Liying Yang, and Yuqing He. A robust and accurate simultaneous localization and mapping system for RGB-D cameras. *8th International Conference on Information Science and Technology, ICIST 2018*, pages 458–465, 2018.