Trabajo de Fin de Grado

Visión artificial aplicada a la robótica

Artificial vision applied to robotics .

Miguel Ángel Delgado Hernández

La Laguna, 4 de julio de 2016

D. **Jonay Tomás Toledo Carrillo**, con N.I.F. 78.698.554-Y profesor Contratado Doctor de Universidad adscrito al Departamento de Ing. Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

"*Visión artificial aplicada a la robótica.*"

ha sido realizada bajo su dirección por D. **Miguel Ángel Delgado Hernández**, con N.I.F. 54.112.876-V.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 4 de julio de 2016

Agradecimientos

En primer lugar me gustaría agradecer a mi tutor académico, Jonay Tomás Toledo Carrillo, porque sin su inconmensurable ayuda no podría haber desarrollado este proyecto hasta el final, siempre atento y abierto a resolver dudas.

En segundo lugar me gustaría agradecer a mis dos compañeros y amigos, Andrés Cidoncha Carballo y Fabián Díaz Lorenzo, porque ellos más que nadie conocen tanto los buenos como los momentos más difíciles, y juntos nos hemos apoyado para aprender de los conocimientos de los tres.

Finalmente agradecer al resto de familiares y demás amigos que también me han apoyado en todo momento.

Muchas gracias a todos.

Licencia

Resumen

El objetivo de este trabajo ha sido la reconstrucción de mapas tridimensionales del entorno a partir de sistemas estereoscópicos de dos cámaras para la localización de un robot en el mismo.

Las cámaras son uno de los sensores que pueden dar una información más rica del entorno para un robot. Con el uso de dos cámaras se puede conocer no solo la imagen en color sino también la distancia a los objetos, haciendo de la información visual, información espacial. Esta información espacial puede ser integrada para la reconstrucción de mapas en 3D.

Este sistema estereoscópico ha sido probado sobre el proyecto Perenquén, un robot autónomo realizado por el Grupo de Robótica de la Universidad de La Laguna, para la reconstrucción de los mapas en exteriores, donde los sensores actuales del robot se ven muy limitados.

Palabras clave: visión estereoscópica, reconstrucción 3D, localización, odometría, ros

Abstract

The purpose of this project has been 3D mapping of environment using a stereostopic vision system by two cameras for robot localization in this map.

Vision systems are one of the best sensors to get valuable information about environment. The use of a stereo camera is the most widespread usage due to it's low price and the good results that can be achieved. Through stereo camera approach is possible to recognize color image further distance to objects, meaning visual information can turn into spatial information. The spatial information can be integrated to get 3D mapping.

This stereoscopic vision system has been tested on Perenquén project, an autonomous robot made by Robotics Group of the University of La Laguna, for 3D mapping in extenral spaces, where current sensors of teh rebot are very limited.

Keywords: *stereo vision, 3D mapping, localization, odometry, ros*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo se hablará sobre el estado y la situación en la que se encuentra el tema a tratar en este Trabajo de Fin de Grado, y los objetivos marcados.

1.1. Antecedentes

Vivimos en una época emocionante para la ciencia y la ingeniería. Ya desde hace 200 años con el comienzo de la revolución industrial, se empezaron a cimentar las primeras bases de la automatización de todas aquellas pequeñas tareas repetitivas que eran un lastre para los tiempos de producción.

En el siglo XX la auténtica revolución ha llegado a través de la computación e internet. La automatización en la computación ha sido, y es crucial, para el desarrollo de nuevas líneas de trabajo como la inteligencia artificial. Precisamente la inteligencia artificial y su implementación en la robótica ha sido el último gran paso en los avances de la humanidad para optimizar, sustituir o eliminar todo aquel trabajo tedioso, repetitivo o simplemente, peligroso. Pero para ello se requiere del uso de todo tipo de sensores que permitan simular el comportamiento de un ser humano.

La visión artificial es uno de los campos de investigación que más interés han causado en las últimas décadas. Sin embargo, no ha sido hasta hace unos pocos años cuando se ha empezado a conseguir los resultados esperados durante todo este tiempo. El uso de cámaras permite obtener mucha y variada información acerca del entorno de un robot: objetivos, obstáculos e incluso muchos datos que aportan información directa de la situación que se visualiza.

1.2. Estado del arte

En la actualidad la visión artificial está muy presente en sistemas de guiado para vehículos autónomos, es decir, vehículos que son capaces de conducir sin intervención humana. El objetivo de la visión artificial en este campo es ofrecer una alternativa más segura y fiable frente al factor humano.

Figura 1.1: Google Self-Driving Car

El ejemplo más conocido es el Google Self-Driving Car [4], un vehículo totalmente autónomo que tras varios años de investigación, desde 2011 ya es legal su circulación en varios estados de Estados Unidos.

Por otra parte, la Universidad de La Laguna cuenta con el proyecto VERDINO [16], desarrollado por el Grupo de Robótica (GRULL). Se trata de un vehículo similar al de Google que hace uso de sistemas láser, sistemas de visión (tanto en el espectro visible como infrarrojo) y sistemas de odometría óptica, entre otras características. Todo ello para ser capaz de detectar el entorno que le rodea, y poder realizar una toma de decisiones mediante el uso de diferentes algoritmos para llegar hasta un destino, al mismo tiempo que preocupa por evitar los obstáculos que aparecen en su camino.

De momento es difícil predecir cuando se producirá la entrada masiva en el mercado de este tipo de vehículos. Los dos proyectos mencionados anteriormente, aunque fiables a pesar de seguir en desarrollo, hacen uso de tecnología poco accesible para el público general debido a su elevado presupuesto.

Figura 1.2: Proyecto VERDINO

Este problema abre las puertas a nuevas alternativas de investigación, a través de tecnologías más económicas que todavía no se han explotado en este campo, pero que son igual de válidas. Es el caso de los sistemas de visión estereoscópica, que con solamente dos cámaras se puede obtener rica información del entorno.

1.3. Objetivo

El tema central de este proyecto es muy amplio, de este modo, es necesario poder realizar una distinción entre el objetivo principal y los objetivos específicos que lo rodean.

1.3.1. Objetivo general

El objetivo principal de este Trabajo de Fin de Grado es poder integrar el uso de un sistema estereoscópico de dos cámaras en un robot, para la construcción de un mapa en 3D y su posterior localización en el mismo.

1.3.2. Objetivos específicos

Los objetivos específicos que componen el proyecto son:

- Uso de una cámara comercial de entretenimiento en un proyecto de investigación.
- Integración de cámaras estereoscópicas junto a otros sistemas de detección de obstáculos.
- Combinación de odometría mecánica y odometría láser.
- Reconstrucción de un mapa tridimensional a partir de las imágenes recogidas por las cámaras.
- Localización en un mapa tridimensional construido previamente.

Capítulo 2

Conceptos

En este capítulo se abordarán todos aquellos conceptos teóricos que han surgido a lo largo del proyecto y son necesarios para entender correctamente la línea del trabajo.

2.1. Visión artificial

La visión artificial por computador, es la disciplina científica que se basa en la adquisición, procesamiento y análisis de las imágenes que se toman del mundo real, con el objetivo de obtener información relevante acerca de ellas: detección de objetos, seguimiento del movimiento, reconocimiento de eventos, etc [2]. Un ejemplo que podemos ver en nuestro día a día, es la detección de caras en una escena capturada por una cámara digital o smartphone, mediante el uso de técnicas de reconocimiento de patrones.

Al igual que sucede en otras áreas de la inteligencia artificial, la visión artificial tiene como objetivo principal obtener la información explícita y el significado de la realidad de la misma manera que lo haría un ser biológico.

El avance progresivo del hardware con nuevos procesadores digitales de señales (DSP) y unidades de procesamiento gráfico (GPU) [49], junto con nuevas tecnologías y planteamientos de cómputo como la computación paralela, ha permitido que en los últimos años se haya podido implementar nuevos algoritmos más rápidos y eficientes, necesarios para ser utilizados en ámbitos críticos, como sistemas en tiempo real.

2.1.1. Dificultades

La capacidad visual es uno pilares de la inteligencia humana. Su implementación en la robótica supone también un importante avance en la inteligencia artificial. Sin embargo, mientras que la percepción visual es algo innato y cotidiano para nosotros, la visión artificial es muy compleja y conlleva muchas dificultades [42]. Entre las principales dificultades, destacan:

- **Mundo tridimensional:** mientras que las imágenes que se obtienen con una cámara son bidimensionales, el mundo que nos rodea no. Es necesario realizar las transformaciones correspondientes para obtener valores correctos.
- **Zonas de interés:** se necesita extraer elementos de información sutiles en imágenes complejas, por lo que entre tanta información es necesario reconocer formas, colores, etc.
- **Carácter dinámico de las escenas:** el mundo está vivo, por lo que en las imágenes que se toman muchos elementos están en movimiento. Por otro lado, otros factores como luminosidad, contraste, foco... pueden marcar una importante diferencia, y por desgracia, estos factores son variables, no se pueden controlar.

2.1.2. Aplicaciones

La visión artificial resulta de gran utilidad en diferentes áreas de aplicación, tanto en acciones repetitivas como peligrosas:

- **Inspección y ensamblaje industrial:** el proyecto Randon Bin Picking"(RBP) [36] hace uso de visión estéreo para la búsqueda de piezas entre objetos de todo tipo para su rápida recuperación.
- **Apoyo en el diagnóstico médico:** en las últimas décadas la visión artificial se ha hecho un importante en la medicina para detectar, analizar y reconstruir la información obtenida [41].
- **Exploración espacial:** en el proyecto de exploración Mars Rover (Mars Exploration Rover Mission) [47] tiene como objetivo explorar la superficie de Marte en busca de rocas u otros elementos que prueben la existencia de agua.

- **Seguimiento (Tracking):** se hace uso en innumerables situaciones de carácter estadístico como contar el número o de en áreas de vigilancia y seguridad monitorizando trayectorias.

2.2. Visión estéreo

La visión estereoscópica o visión estéreo, es la técnica capaz de extraer información tridimensional (profundidad) a partir de la posición relativa de un objeto en imágenes bidimensionales al ser observado desde distintos ángulo por dos o más cámaras separadas a una cierta distancia.

2.2.1. Adquisición

Usando dos cámaras, el procedimiento a seguir para la adquisición del entorno es capturar dos imágenes de una misma escena, desde dos cámaras separadas ligeramente. De esta forma, las imágenes obtenidas, también tendrán un pequeño desplazamiento entre sí.

De manera más formal, se obtiene que para cada imagen capturada por las cámaras, un objeto está en puntos diferentes del plano. Esta triangulación entre el punto P y Q y el origen de referencia, provocan una sensación de profundidad. Mediante el sistema tradicional de una sola cámara, este punto estaría en las mismas coordenadas [44].

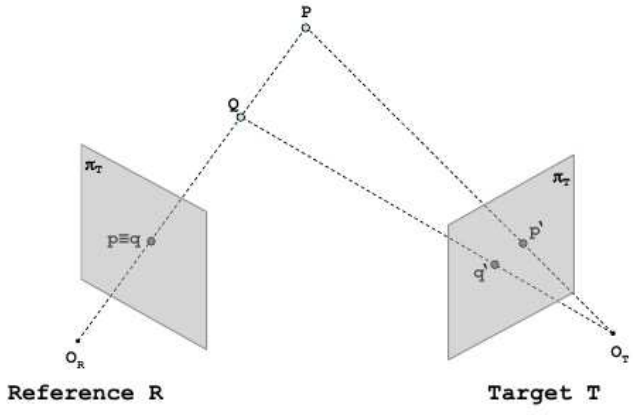

Figura 2.1: Diferencias entre una y dos cámaras

Con estos datos, se pueden poner en correspondencia cada punto de ambas imágenes, para obtener una imagen de disparidad (más información en la sección 2.2.3).

2.2.2. Geometría de las cámaras

En función de la posición relativa de las cámaras entre sí, se pueden apreciar dos métodos principales:

- **Visión paralela:** las cámaras están paralelas entre sí y están separadas por una línea horizontal (línea base). El objetivo que visualiza cada cámara es perpendicular respecto a la línea base, mientras que las líneas de correspondencia que unen los puntos de una imagen respecto a la otra son horizontales.

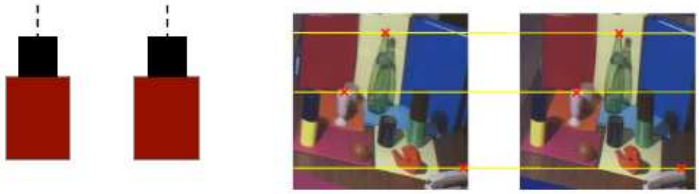

Figura 2.2: Visión paralela

- **Visión cruzada:** las cámaras no están paralelas entre sí, tienen una inclinación de tal forma que el objetivo de cada cámara apunta hacia el lado contrario de una imagen. Por lo que los ejes ópticos se cruzan entre sí. Las líneas de correspondencia, también tienen sufren una inclinación.

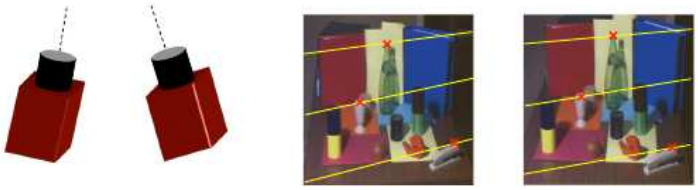

Figura 2.3: Visión cruzada

La visión cruzada tiene la desventaja de distorsionar las imágenes capturadas. En la figura 2.4 se puede observar este efecto al fotografiar un muro de ladrillos. Sin embargo, dependiendo del tipo de escena que se capture, esta distorsión puede suponer un problema o no [50].

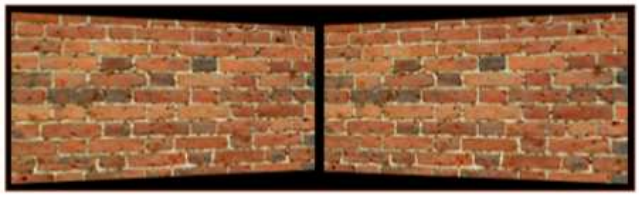

Figura 2.4: Muro distorsionado por visión cruzada

2.2.3. Disparidad

La disparidad de dos imágenes establece la correspondencia entre los píxeles o características que existen entre ambas en el eje x para obtener la profundidad de la escena. Con esto se consigue estimar la profundidad de cada uno de los puntos en la escena.

El objetivo final es poder construir una **imagen o mapa de disparidad**, una imagen que representa la disparidad entre las dos cámaras. Habitualmente el mapa de disparidad se representa como una imagen monocroma donde los objetos con mayor disparidad (más cercanos) son representados con un tono más claro y los objetos con menor disparidad (más lejanos) son representados con un tono más oscuro. Aunque no es extraño encontrar una imagen térmica en vez de monocroma.

Con el único requisito de que las imágenes estén rectificadas, el algoritmo para calcular la disparidad consiste de forma general en buscar cada punto singular de la imagen izquierda en la imagen derecha, para observa en que nuevo píxel se encuentra. A partir de esta información, es posible calcular la profundidad buscando para cada punto de las imágenes la pareja de puntos correspondiente que representan la proyección del mismo punto del espacio [45].

El mapa de disparidad es uno de los elementos básicos para la reconstrucción tridimensional de los objetos de una escenas a partir de varias capturas.

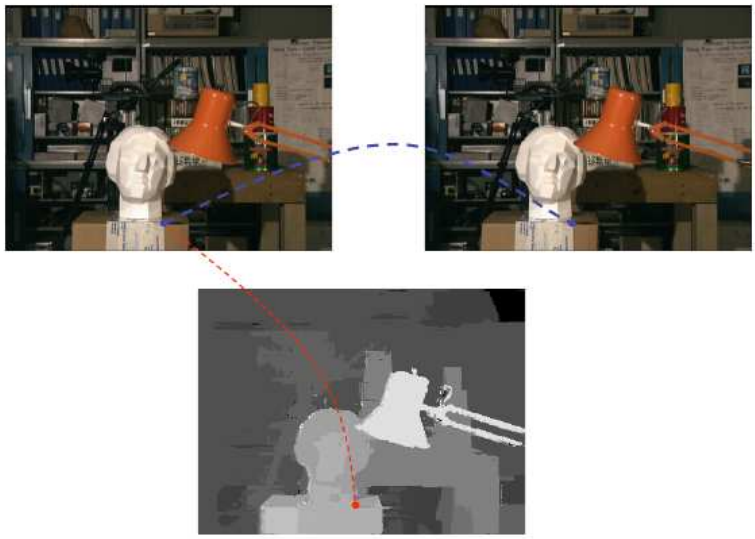

Figura 2.5: Mapa de disparidad a partir de imágenes en estéreo

2.3. Odometría

La odometría es el método que permite estimar la posición de un robot móvil en el entorno en un momento determinado.

Cabe recalcar, que la odometría no permite conocer la posición exacta de un robot, ya que existen muchos factores externos, que provocan errores en la obtención de los datos de entrada. Estos errores son acumulativos, por lo que la veracidad de los resultados es inversamente proporcional a la distancia recorrida de un robot.

A pesar de ello, la odometría ofrece una gran precisión a corto plazo y su precio de implementación es realmente bajo, por lo que lo convierte en uno de los pilares básicos para la navegación.

Dependiendo de la tecnología utilizada para calcular la odometría, se puede hablar de diferentes tipos.

2.3.1. Odometría mecánica

La odometría mecánica se base en el estudio del giro de las ruedas de un robot. Conociendo el radio de las ruedas, mediante las revoluciones de las mismas es posible determinar con sencillas ecuaciones el avance del robot [10].

Sin embargo esta sencilla solución tiene varios inconvenientes a tratar:

- **Las ruedas:** las ruedas utilizadas deben contar con las mismas características, pero en muchas ocasiones es el diámetro de las ruedas es diferente, o simplemente el diámetro no se corresponde con las especificaciones dadas por el fabricante. Por otro lado es necesario que las ruedas se encuentren bien alineadas.
- **El entorno:** la superficie por donde se mueve el robot puede no ser la adecuada. Dependiendo de la composición del suelo (gravilla, hierba, baldosas, etc), el robot puede resbalar, derrapar o incluso encontrarse sin ningún punto de contacto con el suelo. Por otra parte, una superficie desnivelada o llena de obstáculos inesperados también supone un problema para obtener la odometría.

2.3.2. Odometría visual

La odometría visual permite estimar el movimiento y la posición de un robot móvil a partir de las imágenes capturadas por una o más cámaras en el propio sistema. De forma sencilla, si se comparan las imágenes obtenidas en un instante t1 respecto a otras imágenes obtenidas en un instante t2, se puede calcular la diferencia existente en un punto determinado de la imagen. Si dicho punto es más cercano, quiere decir que el robot ha avanzado.

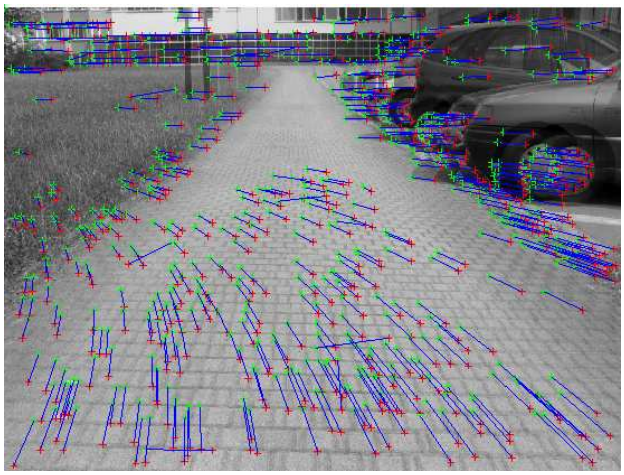

Figura 2.6: Disparidad en odometría visual

Pero a partir de las imágenes ¿cómo se puede determinar la distancia de un objeto respecto al robot? En función del tamaño del objeto en la escena. En un sistema estereoscópico se comparan la diferencia respecto al eje x (disparidad) entre la imagen izquierda y derecha. La distancia de un objeto es inversamente proporcional a la disparidad presente, es decir, si un objeto presenta una gran disparidad en la escena significa que se encuentra cerca. Cuanto más lejano esté el objeto la disparidad será menor [11].

Por otra parte, en los sistemas con una sóla cámara, también es posible hallar la distancia. La cámara debe estar orientada hacia el suelo en un ángulo determinado y a una altura conocida, a partir de la tangente se puede calcular la distancia.

La odometría visual es la alternativa utilizada en robots móviles que no cuentan con ruedas: androides, robots voladores, etc. y en los sistemas que no pueden permitirse el uso de láseres. Aunque es bastante frecuente encontrar la odometría visual junto a cualquiera de las otras odometrías como apoyo.

La odometría visual cuenta con los siguientes inconvenientes a tener en cuenta:

- **Las cámaras:** para funcionar correctamente, es necesario que las cámaras estén calibradas. Por otra parte, si se utiliza un sistema estereoscópico se necesita verificar que la distancia entre las cámaras sea la correcta.
- **El entorno:** las cámaras son muy sensibles a los cambios bruscos de las condiciones del entorno como la luminosidad. También hay que tener en cuenta que las escenas son dinámicas, por lo que muchos objetos aparecen

y desparecen de la escena o se mueven con mucha frecuencia en la escena, y esto puede suponer un importante problema.

Tanto la odometría mecánica, láser y visual presentan varios problemas para estimar la posición y orientación del robot, sin embargo, combinando ambas se pueden conseguir resultados más consistentes. Por ejemplo: en aquellos casos donde la odometría mecánica no coincide con los datos visuales debido a una superficie inconsistente, se confía en la odometría visual. Por otra parte, cuando las condiciones lumínicas no son adecuadas en espacios cerrados, se confía en la odometría láser o mecánica.

2.4. Navegación robótica

La navegación robótica es la habilidad que permite a un robot móvil poder localizarse en el entorno y poder moverse libremente por el mismo, a partir del conocimiento extraído de las imágenes obtenidas del medio. El objetivo principal es conocer por donde debe y no debe navegar con la idea de poder alcanzar un punto de destino marcado previamente.

En la robótica móvil es indispensable saber como moverse por el mundo, además de conocer que situaciones de riesgo se han de evitar: colisiones, superficies irregulares, zonas prohibidas, etc [19].

Es posible subdividir la navegación dependiendo de la zona de estudio: de interiores o de exteriores. En ambos casos las tecnologías pueden ser las mismas, sin embargo mientras que en la localización en exteriores se suele hacer uso de GPS, odometría mecánica, etc., en interiores es posible conseguir mejores resultados con sensores ópticos como cámaras, odometría láser, etc.

La navegación robótica se caracteriza por estos tres tópicos:

- **Localización:** localizarse en el entorno.
- **Búsqueda de caminos:** búsqueda del camino más optimo para llegar a un objetivo.
- **Mapeo robótico:** construcción de un mapa del entorno.

2.4.1. Localización

La localización es el primer tema a abordar en la navegación de un robot. El objetivo es dar respuesta al '¿dónde estoy?', a través de conocer la posición

inicial, conocer la posición del punto de destino y autolocalizarse por el entorno. La localización se puede dividir en dos grupos principales: basada en puntos de referencias y basada en el análisis de las imágenes [51].

La localización basada en puntos de referencias se aprovecha en los puntos de referencias que existen en el entorno y sobresalen de las imágenes de la escena sin verse influenciados por otros factores de las escenas como los cambios en el entorno o de los elementos que lo componen. Este tipo de puntos o marcas pueden ser artificiales (líneas o flechas en un mapa o GPS) o naturales (puertas, esquinas, senderos, etc.).

Figura 2.7: Localización a través de código QR

Cuando este tipo de marcas no se pueden separar de las imágenes, porque bien no se tienen puntos de referencia artificiales, o no se puede reconocer del propio entorno marcas artificiales, se recurre a la localización basada en el análisis de las imágenes capturadas. El primer paso es el 'auto-aprendizaje', habitualmente navegar de forma autónoma y recoger las imágenes del entorno. Estas imágenes se procesan, comparando los elementos en la escena de cada nueva imagen recogida con la anterior y las imágenes que se tiene en el histórico, obteniendo así la localización actual.

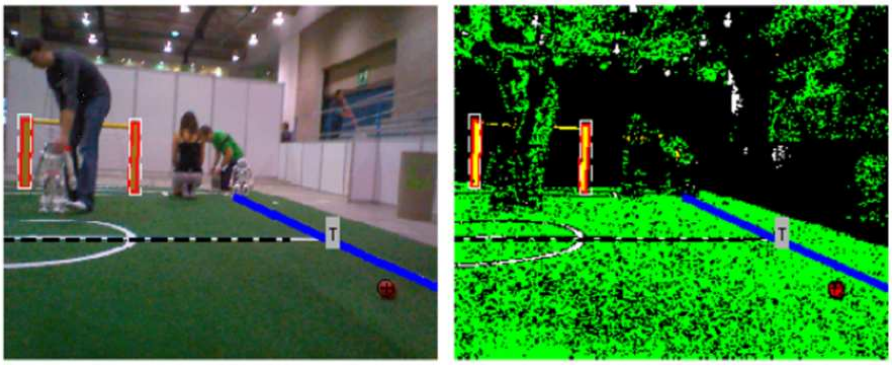

Figura 2.8: Localización mediante los elementos de la escena

Filtro de Kalman Extendido
El filtro de Kalman extendido (EKF) es un algoritmo que permite estimar la posición de un robot a partir de la combinación de las ecuaciones de la odometría y las medidas recogidas por los sensores. Se trata de un algoritmo muy utilizado en sistemas de navegación.

El filtro de Kalman se basa en dos etapas: predicción y corrección.

En la etapa de predicción se tiene solamente en cuenta la odometría, cuyos datos son almacenados en un vector de estado, los cuales se toman como una estimación Este vector contiene las variables de interés, manteniendo el tiempo dos posibles valores: el valor previsto (a priori) y el valor corregido (a posteriori).

En la etapa de corrección, se introducen las medidas de los sensores. Con la odometría y los sensores, se calcula la diferencia entre el valor esperado y el real, se obtiene la matriz de covarianza del sensor y se obtiene la Ganancia de Kalman (determina cuanto se debe corregir la estimación). Finalmente se corrigen los valores del vector de estado después de las nuevas observaciones [48].

Conforme pasa el tiempo crece la imprecisión, por lo que las etapas se repiten sucesivamente con el fin de reducir la imprecisión.

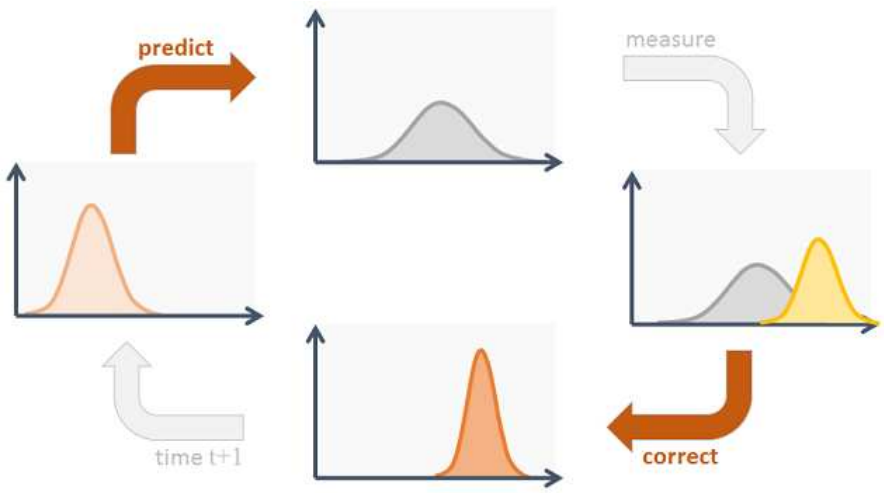

Figura 2.9: Esquema del filtro de Kalman extendido

2.4.2. Búsqueda de caminos

La búsqueda de caminos permite hallar el conjunto de movimientos que permite a un robot encontrar el camino más óptico hasta llegar a su objetivo, para no solamente llegar en el menor tiempo posible, sino también evitar por

el camino los obstáculos que hagan peligrar el estado del robot o simplemente le retrasen [40].

El problema de la búsqueda de caminos ha sido ampliamente estudiado, existiendo múltiples algoritmos para solventar este problema.

Entre los métodos más conocidos están los 'roadmap', los cuales se basan en construir una descripción de todo el espacio libre en el plano mediante un grafo. Posteriormente los nodos del grafo se van conectando a partir de las distancias más cortas entre sí, formando todos los caminos posibles, entre los que se encuentra el más óptimo.

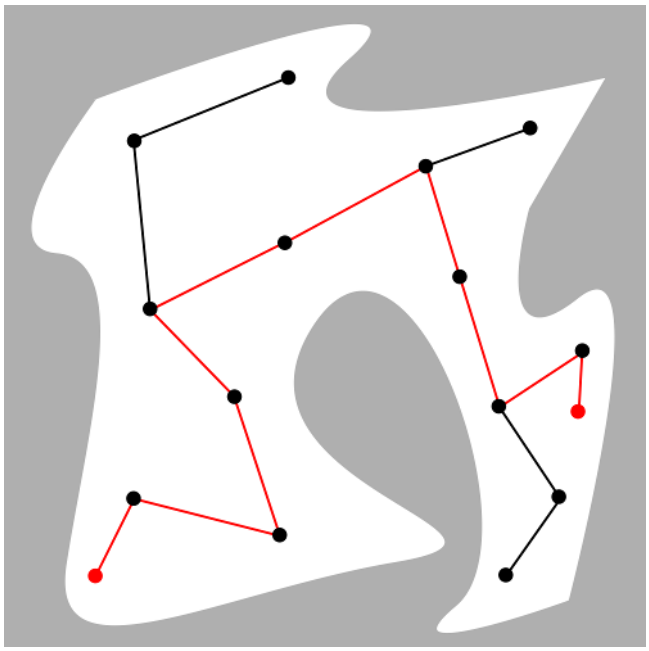

Figura 2.10: Ejemplo de un 'roadmap'

Por otra parte, existen los métodos de descomposición en celdas, que en vez de representar el entorno en un grafo, dividen el espacio libre de colisiones en un conjunto de celdas.

En los últimos años han surgido métodos alternativos que permiten dar solución a la complejidad del entornos y las restricciones cinemáticas que puede presentar un robot en la búsqueda de caminos, son los algoritmos de generación aleatoria, el más famoso es el RRT o 'Rapidly Exploring Random Trees', muy utilizados en la navegación autónoma. Este algoritmo se basa en la construcción al azar de un árbol que va creciendo de manera incremental mediante la captura de nuevas muestras del entorno [46].

2.4.3. Mapeo robótico

La construcción de un mapa constituye el objetivo final de la navegación robótica. Al igual que en la cartografía, un mapa representa toda la información conocida acerca de un entorno: dimensiones del entorno, orografía, localización de los obstáculos, etc. En robótica la utilización de un mapa es de gran utilidad ya que permite tener un registro histórico de aquellas lugares que ya ha visitado en el pasado, evitando así visitar zonas peligrosas o irrelevantes [21].

Los mapas se pueden representar de diferentes formas:

- **Mapa métrico:** representación bidimensional del espacio y todos los objetos que lo rodean. Aunque la posición de los obstáculos es bastante aproximada a la realidad, deja de lado información importante del entorno. Una ampliacion de este tipo de mapas es mediante una representación tridimensional a través de la disposición de una nube de puntos, siendo posible apreciar datos importantes del entorno: orografía del terreno, altura de los obstáculos, imágenes más detalladas del entorno, etc.
- **Mapa topológico:** grafo en el que los nodos corresponden a los lugares y sus arcos a la distancia entre ellos. Permite tener un abstracto claro y sencillo de la relación de los obstáculos entre ellos.

SLAM

A pesar de que la navegación de un robot se ha divido en tres etapas, donde la primera es la localización y la última lo construcción de un mapa, realmente un robot autónomo es capaz de construir un mapa del entorno comenzando desde una posición desconocida en el mismo.

SLAM (Simultaneous Localization And Mapping) es un problema que surge de la necesidad construir un mapa de un entorno desconocido a la vez que es capaz de localizarse en el mismo mediante sus sensores.

Uno de los factores más importantes a tener en cuenta es la incertidumbre: no se conoce ni la posición ni el entorno, y no solo eso, los sensores no son perfectos y pueden arrojar resultados que no son acordes a la realidad, por lo que la clave está en seguir un modelo probabilístico. Esta imprecisión puede corregirse con el paso del tiempo recorriendo continuadamente [43].

SLAM toma como base el teorema de Bayes para dar solución del problema, mediante el uso de algoritmos como el Filtro de Kalman Extendido o los Mapas de ocupación de eldillas. Por otra parte, uno de los retos a los que se enfrenta SLAM es al ‘loop closure‘ (cierre de bucle). Se trata del problema de reconocer

aquel lugar del mapa que ya ha sido visitado previamente. El éxito del SLAM depende poder detectar esta situación [35].

Los principales problemas con los que cuenta SLAM radican en la complejidad que se tiene en los entornos dinámicos donde muchos objetos en el mapa se mueven libremente. Por otra parte, es una técnica muy exigente que no se puede utilizar en grandes entornos con muchos objetos, ya que el coste computacional crece al cuadrado conforme a los objetos contenidos en el mapa.

A pesar de ello, los resultados obtenidos por SLAM son muy atractivos, ya que sin conocer ni la posición ni el mapa, un robot autónomo puede moverse libremente por el, obteniendo al mismo tiempo un mapa del entorno. Por otra parte esta técnica es bastante robusta ante el ruido provocado por lo sensores.

Capítulo 3

Recursos y herramientas

En este capítulo se hablará tanto del hardware como del software utilizado para realizar el proyecto. A grandes rasgos se integrará la PlayStation Camera en el sistema Perenquén, utilizando el framework ROS para el funcionamiento de ambos.

3.1. PlayStation Camera

PlayStation Camera es una cámara utilizada por PlayStation 4 como accesorio. Fue lanzada al mercado en 2013 coincidiendo con el lanzamiento de la consola, aunque vendiéndose como un accesorio a parte a un precio recomendado de 59.99$ [**?**].

Figura 3.1: PlayStation Camera

Esta cámara esta pensada para poder controlar la consola sin necesidad de utilizar el mando tradicional. Los usuarios de PS4 pueden iniciar sesión haciendo uso de las características de reconocimiento facial. También permite utilizar los movimientos del propio cuerpo para moverse por los menús del sistema y utilizar la voz para realizar acciones gracias al micrófono incorporado. PlayStation Camera también es compatible con otros accesorios de PlayStation 4 como PlayStation Move (sistema de control por movimiento) para una navegación más precisa.

Aunque el objetivo principal de PlayStation Camera es su integración en los videojuegos, para conseguir mayor inmersión: control de movimiento, comandos de voz y tecnología de realidad aumentada [**?**].

3.1.1. Historia

Los orígenes de PlayStation Camera vienen desde 1999 cuando Sony empezó a investigar sobre visión artificial y el uso de reconocimiento de gestos mediante una cámara para incorporar esta tecnología en los videojuegos. Finalmente en 2003 se lanzaría EyeToy, una cámara que tuvo un notable éxito para PlayStation 2. En 2007, tras la salid de PlayStation 3, se lanzó al mercado una actualización de esta cámara con el nombre de PlayStation Eye, la cual permitía capturar imágenes a una mayor resolución y una tasa de fotogramas por segundo superior.

Mientras que EyeToy y PlayStation Eye compartían el mismo concepto, no fue hasta el lanzamiento de Kinect por parte de Microsoft en 2010, cuando se pudo ver una evolución real en este tipo de accesorios de entretenimiento. Kinect se basó en una cámara RGB-D que contaba con sensores de profundidad, múltiples micrófonos un procesador independiente de la consola. Kinect permitía captura de movimiento en 3D, reconocimiento facial y reconocimiento de voz [8].

Kinect fue un éxito de ventas y obtuvo muchos elogios por parte de la crítica, pero no en el plano de los videojuegos, si no de la investigación. Microsoft decidió lanzar un SDK oficial para el desarrollo en Windows en base al interés que existía, para proyectos de todo tipo [9].

PlayStation Camera ha tomado nota de Kinect para ofrecer un producto con las mismas ventajas de Kinect, aunque utilizando tecnología estereoscópica y características más modestas para ser una alternativa económica a Kinect.

3.1.2. Especificaciones técnicas

PlayStation Camera cuenta con un chip OV580 encargado de sincronizar las dos cámaras. Cada cámara cuenta con un sensor de imagen OV9713, mientras que el chip que controla el sonido es el AK5703. La configuración inicial de la cámara se almacena en una memoria EEPROM 4g51A [18].

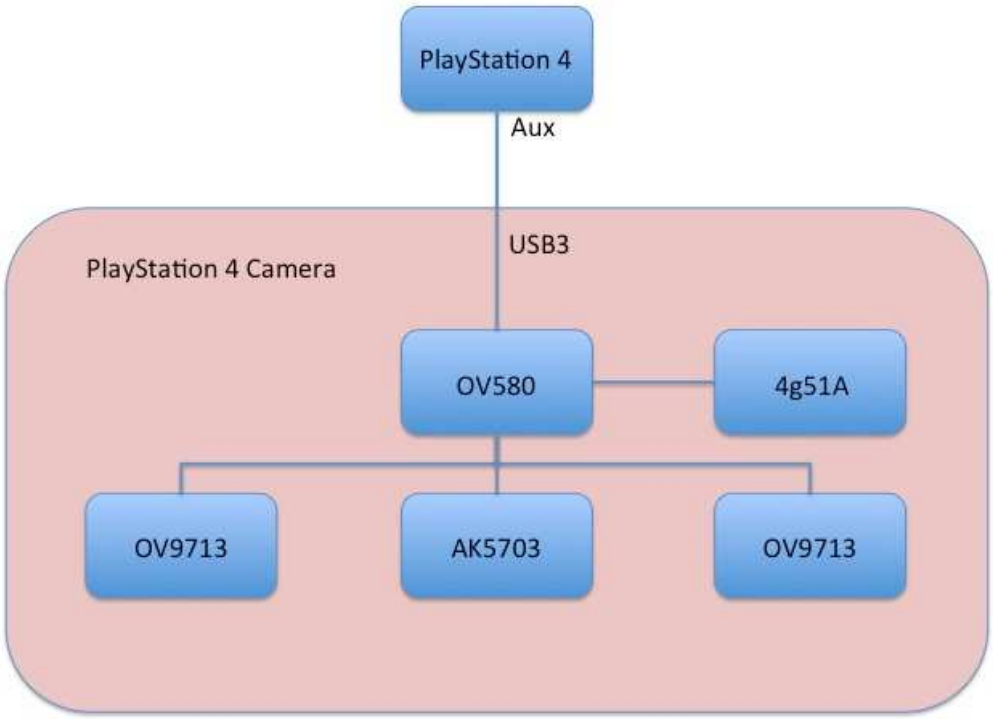

Figura 3.2: Diagrama de PlayStation Camera

Por su parte la cámara cuenta con varios ajustes programables: exposición, balanceo de blancos, gama, saturación, contraste, nitidez y tono.

Nombre	**Descripción**
Dimensiones	186mm x 27mm x 27mm
Peso	183 gramos
Conexión	USB 3.0 propietario
Rango de captura	+30cm
Campo de visión (FOV)	85º
Apertura	f/2.0
Formato de vídeo	RAW16/RAW8, YUV442/YUV8
Profundidad de color	12-bit (4096 colores)
Resolución	1280x800 @60FPS, 640x400 @120FPS, 320x200 @240FPS, 160x100 @240FPS,
Micrófono	Multiarray de 4 canales

Tabla 3.1: Tabla resumen de las características de PlayStation Camera

3.2. Perenquén

Perenquén es el nombre que ha tomado el proyecto de automatización de una silla de ruedas por el Grupo de Robótica de la Universidad de La Laguna, cuya filosofía es llevar los progresos conseguidos con el proyecto Verdino, a una silla de ruedas [15].

Figura 3.3: Perenquén

El objetivo principal de este proyecto es el desplazamiento autónomo de una silla de ruedas, mediante la detección de obstáculos de un entorno, así como su localización en el mismo a través de las imágenes capturadas mediante la versión 2.0 de Kinect A diferencia de Verdino, Perenquén intenta ser una alternativa de bajo coste frente a los sistemas de detección de obstáculo de Verdino.

Actualmente la silla cuenta con el dispositivo Kinect, un sistema de odometría mecánica, y tres detectores láser (lado izquierdo, lado derecho y posterior) que hacen posible una detección más precisa de los obstáculos. Sin embargo, Kinect cuenta con una importante desventaja, y es que no funciona correctamente en exteriores, tiene un alcance acotado de apenas entre 5 y 10 metros [39].

3.3. ROS

ROS (Robot Operating System) es un framework de código abierto diseñado para realizar software robótico. Entre las principales funcionalidades destacan una abstracción del hardware, el control de dispositivos, paso de mensajes entre los procesos y una gran variedad de herramientas y librerías. Por otra parte ROS cuenta con un sistema de paquetes, que permite instalar, compilar e incluso modificar paquetes externos creados por la comunidad [1].

El objetivo principal es permitir la abstracción y la simplificación de muchas de las tareas que conllevan construir un robot de forma independiente, uno de los principales problemas que llevan a condenar al fracaso este tipo de proyectos, mediante el desarrollo de software colaborativo. En definitiva, ROS

promueve la reutilización de código para evitar tener que reinventar la rueda a los desarrolladores y científicos en cada nuevo proyecto [6].

De esta forma, a modo de ejemplo un grupo de desarrolladores puede trabajar e investigar en la construcción de mapas, mientras que en otra parte del mundo, otro grupo de expertos puede estar especializado en la localización con el uso de mapas. Ambos comparten el conocimiento entre ambos y con el resto de la comunidad para conseguir mejores resultados en conjunto.

El software de ROS se distribuye de la siguiente manera:

- Herramientas y lenguajes independientes para construir y distribuir ROS.
- Librerías del cliente de ROS que facilitan el desarrollo.
- Paquetes que contienen la aplicación del usuario.

Tanto las herramientas como las librerías del cliente de ROS están escritas principalmente en C++ y Python. Las librerías principales de ROS siguen los principios Unix-like ya que la mayoría de dependencias son proyectos de código abierto, permitiendo que muchas distribuciones distribuyan las librerías y paquetes de ROS [20].

ROS ha sido diseñado para ser un sistema distribuido lo más modular posible, permitiendo que los usuarios puedan utilizar lo justo y necesario, desde los paquetes del núcleo y la comunidad, hasta utilizar sus propios paquetes personales. Actualmente ROS cuenta con más de 3000 paquetes públicos en su ecosistema, siendo difícil conocer cual es el número real de todos los paquetes. Muchos de estos paquetes ayudan enormemente en la infraestructura de la comunidad, ofreciendo accesos a drivers de dispositivo, capacidades genéricas en robótica, algoritmos, herramientas de desarrollo y librerías.

Desde sus inicios, la comunidad de ROS ha crecido considerablemente en todo el mundo, principalmente en laboratorios de investigación, pero en los últimos tiempos también en otros sectores comerciales e industriales. Más de 1500 personas participan en las listas de correos, y aproximadamente 6000 personas colaboran con la documentación oficial y la comunidad de preguntas y respuestas. Esto implica que más de 30 páginas de la documentación se editen cada día y que hasta la fecha se hayan abierto más de 13000 preguntas con un porcentaje de respuesta alto [7].

3.3.1. Historia

El origen de ROS se remonta a 2007, cuando Willow Garage pensó en la necesidad de de diseñar un sistema más flexible y robusto para el desarrollo de robots después de que en en el laboratorio de inteligencia artificial de la universidad de Standford es estuvieran realizando diferentes proyectos que integraban el uso de robots e inteligencia artificial como STandard AI Robot (STAIR) y Personal Robots (PR).

Varios investigadores contribuyeron y dieron ideas para lo que sería la base fundamental del servicio de paquetes de ROS que existe actualmente. En apenas un año, el proyecto paso a desarrollarse por la institución de investigación robótica Willow Garage, junto con la colaboración de más de veinte instituciones. Mientras tanto, el uso de una licencia permisiva de código abierto permitió que de forma gradual ROS se empezará a implementar en diferentes comunidades de desarrollo robótico. Finalmente, en 2013, ROS se transfirió a la Open Source Robotics Fundation.

Y hasta la fecha se han lanzado las siguientes versiones:

Nombre	**Lanzamiento**	**Soporte**
Kinectic Kame	23/05/2016	Soporte hasta 2021
Jade Turtle	23/05/2015	Soporte hasta 2017
Indigo Igloo	22/07/2014	Soporte hasta 2019
Hydro Medusa	04/09/2013	Sin soporte
Groovy Galapagos	31/12/2012	Sin soporte
Fuerte Turtle	23/04/2012	Sin soporte
Electric Emys	30/08/2011	Sin soporte
Diamondback	02/03/2011	Sin soporte
C Turtle	02/08/2010	Sin soporte
Box Turtle	02/03/2010	Sin soporte

Tabla 3.2: Tabla con versiones de ROS hasta la fecha

3.3.2. Infraestructura de ROS

Uno de los elementos más básicos en la implementación de una aplicación robot es el sistema de comunicación. ROS ofrece un sistema de mensajes entre los nodos del sistema mediante un mecanismo que usa el patrón de publicación/suscripción.

Los nodos son los procesos que se encargan de las tareas concretas, como

por ejemplo controlar un láser, controlar los motores de las ruedas o planificar la trayectoria del robot. Cada nodo tiene un nombre único en el sistema, para evitar ambigüedad a la hora de comunicarse entre sí. Los nodos pueden ser escritos en los diferentes lenguajes de las librerías del cliente de ROS (C++ o Python por ejemplo).

Por otra parte, el patrón publicación/suscripción permite pasar información de una manera sencilla y segura entre los nodos. En este patrón, existen dos actores:

- **Publicador:** publica la información.
- **Suscriptor:** recibe la información.

Una aproximación real de este patrón puede ser el editor de una revista. El editor (publicador) escribe un número nuevo cada mes. Este número se distribuye directamente o a través de un distribuidor. En último lugar, tenemos a una persona suscrita a esa revista (suscriptor), a quien le llega cada mes un nuevo número de la revista [5].

La información que se intercambian los nodos (en el ejemplo anterior los ejemplares de la revista) se conoce como mensaje. Un mensaje es una estructura con una serie de campos primitivos: enteros, flotantes, booleanos, cadenas de texto, etc.

Para el intercambio de información entre los nodos, los mensajes deben transmitirse por un canal, en este caso denominado tópico. Los tópicos utilizan un nombre para describir el contenido de los mensajes, por lo que, un tópico transmite un cierto tipo de datos. Si un nodo desea recibir cierto tipo de datos, se debe suscribir al tópico apropiado del nodo. Por ejemplo una cámara y un visor, ambos comparten el tópico *"imagen"*, cada vez que la cámara captura una imagen el resultado se publica (publicador), si el visor espera recibir una imagen y está suscrito al tópico *"imagen"* de la cámara, esta se transmitirá. Un nodo puede publicar o suscribirse a múltiples tópicos, además, pueden haber muchos publicadores concurrentemente para un sólo tópico [22].

Una de las mayores ventajas del sistema de publicación/suscripción es que es anónimo y asíncrono, por lo que los mensajes que se transmiten no se modifican. Sin embargo, a veces es necesario realizar comunicaciones entre los nodos de manera síncrona o mediante interacciones de petición y respuesta en vez de realizar el transporte de los mensajes en un un único sentido. Para ello, ROS ofrece lo que se llaman servicios. La petición y respuesta se definen mediante una estructura de mensajes, una para la petición y otra para la respuesta.

Es necesario conocer, que los tópicos se usan cuando se desea transmitir mensajes que continuamente envían información (datos de los sensores, estado del robot, imágenes capturadas, etc.), mientras que los servicios están pensados para realizar una petición y recibir una respuesta bajo demanda en ocasiones concretas [3].

Por otra parte, ROS utiliza lo que se conoce como servidor de parámetros, un diccionario compartido de clave-valor y accesible desde la red. Los nodos usan el servidor para almacenar y recibir parámetros en tiempo de ejecución, y de esta forma modificar el funcionamiento del sistema.

ROS cuenta con las herramientas para monitorizar y controlar el estado de los nodos, tópicos, servicios y parámetros mediante *"rosnode"*, *"rostopic"*, *"rosservice"* y *"rosparam"*. Para que el sistema funcione correctamente, es necesario que este activo el ROS Master. Se trata del nodo maestro, sin él los nodos no podrían encontrar al resto de nodos, tampoco se podría intercambiar mensajes o invocar servicios.

En la figura 3.4 se puede observar una implementación de como captura imágenes un robot. Por un lado existen tres nodos: el nodo de la cámara que recibe de la cámara, y los nodos de procesamiento de las imágenes y la visualización de las mismas. Estos dos nodos están suscritos al tópico *"/image_data"* que publica el nodo de la cámara. Cada vez que la cámara registra una nueva imagen se transmite a los otros nodos.

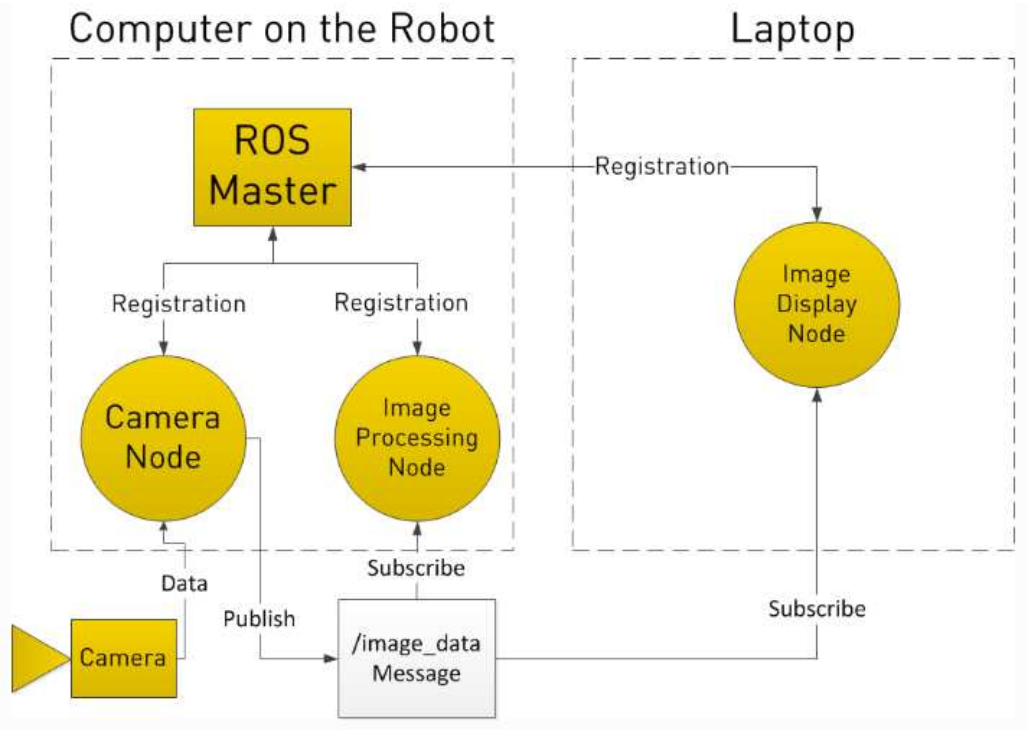

Figura 3.4: Ejemplo de captura de imágenes

3.3.3. Características

ROS proporciona un gran número de librerías para resolver los problemas comunes en el desarrollo robótico con el objetivo de simplificar el la dura carga de tiempo y trabajo que eso supone [23]. Estas son algunas de las principales características:

- **Mensajes estándar:** existe una gran variedad de mensajes básicos ya definidos para la mayoría de tareas del robot como la pose, coordenadas de transformación, sensores, odometría, mapas, etc.
- **Geometría del robot:** la librería *"tf"* se encarga de detectar las diferentes partes de un robot frente al resto de ellas, siendo posible coordinar y transformar más de cien grados de libertad.
- **Lenguaje de descripción:** ROS permite describir las particularidades de un robot de una manera redactable. Para ello se usa URDF (Unified Robot Description Format), un lenguaje de descripción basado en XML que describe todas las características físicas del robot como el tamaño de las ruedas, la distancia entre los diferentes componentes e incluso la apariencia visual de cada elemento.
- **Diagnóstico:** se ha diseñado un sistema capaz de recoger toda la información del robot, para poder registrar, analizar y resolver cualquier problema presente en el estado del robot.
- **Navegación:** la mayoría de problemas en la navegación de un robot como la estimación de la pose, la localización en un mapa y la construcción del misma, son fácilmente resolubles mediante los paquetes que proporciona ROS sin necesidad de buscar soluciones propias o de terceros.

3.3.4. Herramientas

Una de las características más importantes de ROS está en las herramientas de desarrollo disponibles. Existe una gran variedad de software de todo tipo: visualización, depuración, resumen del estado, etc. Gracias al mecanismo de publicación/suscripción es muy sencillo observar y depurar lo que está ocurriendo en todo momento.

Por otra parte, ROS puede ser utilizado sin necesidad de usar una interfaz gráfica (GUI). Toda las características y funcionalidades de ROS pueden realizarse a través de más de 45 comandos en consola: lanzar grupos de nodos,

examinar tópicos y servicios, grabar y reproducir bolsas de datos, etc. En cualquier otro caso existe una importante variedad de herramientas con interfaz gráfica que extienden la funcionalidad de estos comandos. A continuación se pueden ver algunos de ellos.

RTAB-Map ROS

Esta herramienta es una extensión del software original RTAB-Map. RTAB-Map es un aprovechamiento de las técnicas de SLAM capaz de detectar los cierres de bucles en la navegación de un robot a partir de la apariencia de las imágenes, los cuales son registrados en un grafo que representa el mapa. Por otra parte, permite la reconstrucción tridimensional del mapa, el cuál es representado como una nube de puntos [34].

ROS cuenta con RTABMapviz para visualizar de forma gráfica el resultado de la detección de los cierres de bucle y la generación en vivo del mapa tridimensional.

RViz

RViz es una herramienta de propósito general que permite visualizar en un espacio tridimensional mucho de los elementos de un robot como ruedas, brazos o sensores láser. RViz también puede visualizar diferentes mensajes comunes de ROS como barridos láser, nube de puntos, las imágenes de las cámaras o también con la posibilidad de mostrar el mapa del entorno (previamente almacenado).

RViz también puede interactuar con la información de la biblioteca *"tf"* para mostrar toda la información del sistema de coordenadas de los sensores desde diferentes perspectivas. Entre las ventajas, destaca que se trata de una herramienta muy importante ya que permite visualizar que ocurre en el robot en cada momento en busca de errores.

RQt

RQt es un framework escrito en Qt como el nombre indica que permite desarrollar aplicaciones personalizadas con interfaz gráfica para cualquier robot. RQt trae consigo una gran variedad de plugins y/o utilidades para trabajar, de esta forma cualquier usuario puede modificar según sus necesidades la apariencia y distribución de estos. Aunque también es posible escribir nuevos plugins ajustados a la medida de cualquier usuario.

Entre los principales plugins, destacan:

- **rqt_image_view:** se trata de la adaptación del paquete de ROS 'image_view. Permite visualizar todas las imágenes disponibles en los diferentes tópicos activos, tanto imágenes normales, como imágenes en estéreo e imágenes de disparidad.

- **rqt_bag:** permite grabar y gestionar bolsas de datos. Una bolsa en formato que permite almacenar todos tipo de datos previamente grabados como la lectura de sensores, las imágenes capturadas o el sistema de coordenadas utilizado. Son de gran utilidad para depurar y comparar con otras bolsas, ya que se pueden visualizar siempre que se deseen sin necesidad de utilizar el robot que se utilizó.

- **rqt_graph:** muestra el estado actual del sistema en forma de grafo, como los nodos y las conexiones entre ellos, siendo de gran utilidad para la depuración de errores ya que son una forma rápida de ver que todo está el sistema está conectado correctamente.

- **rqt_reconfigure:** adaptación del paquete *"dynamic_reconfigure"* que que permite modificar en tiempo real todos los parámetros de los nodos, para realizar modificaciones rápidas al vuelo. La configuración de los parámetros se puede exportar para utilizar en el futuro.

Capítulo 4

Desarrollo

En este capítulo se explicará como configurar el entorno de trabajo de ROS y se hablará del trabajo realizado en este proyecto.

4.1. Preparación del entorno

Como se pudo observar en el capítulo 3, el desarrollo de este trabajo se realiza en base a ROS, concretamente la versión *"Indigo"*. Es necesario instalar y configurar un directorio de trabajo para ROS donde estará alojado el paquete con el código fuente del proyecto.

El sistema operativo que se ha utilizado ha sido Ubuntu 14.04 LTS (64-bit), una distribución GNU/Linux muy sencilla de utilizar para la que se distribuyen los paquetes que conforman ROS [37]. Cabe mencionar, que todos los pasos que se describen a continuación son perfectamente válidos para versiones más recientes de Ubuntu o distribuciones derivadas, solo es necesario modificar algunos detalles como la versión o el nombre de la distribución.

4.1.1. Instalación y configuración de ROS

Instalar ROS

ROS no se encuentra en los paquetes oficiales de Ubuntu, es necesario añadir un repositorio externo, el oficial de ROS [38]. Para ello se introduce lo siguiente en una terminal:

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

```
    ↪ sources.list.d/ros-latest.list'
```

Se añaden las claves del repositorio externo:

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-
  ↪ keyservers.net --recv-key 0xB01FA116
```

Con esto solamente es necesario actualizar la base de datos de los paquetes disponibles en los diferentes repositorios:

```
$ sudo apt-get update
```

El gestor de paquetes ya puede encontrar los paquetes de ROS entre sus repositorios. Los paquetes se pueden instalar de forma individual, pero es recomendable realizar una instalación completa para no echar en falta ningún paquete en el futuro. Para ello en la terminal instalamos el siguiente paquete:

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

El paquete *"ros-indigo-desktop-full"* viene con muchas de las herramientas que se han usado en el proyecto: Rviz, Rtabmap, Rqt, etc. Aunque no incluye la herramienta *"rosinstall"* la cual permite instalar paquetes de ROS independientemente del sistema:

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

Configuración de ROS

Antes de poder utilizar ROS es necesario configurar *"rosdep"*. Esta herramienta permite instalar fácilmente dependencias del sistema que surgen cuando se requiere compilar el código fuente de un paquete ROS. Es necesario introducir en una terminal lo siguiente:

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

Por último, también es conveniente cargar las variables del entorno de ROS. Si se utiliza Bash como shell basta con introducir lo siguiente:

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

En caso de utilizar otra shell por defecto (por ejemplo Zsh), en *"/opt/ros/indigo"* también se cuenta con el script equivalente, cuyo contenido es necesario volcar en el archivo de configuración de la shell. Con esto último, todas las variables de entorno se cargan automáticamente al abrir una nueva shell.

Crear un workspace

Un workspace es un directorio donde se trabaja con paquetes catkin (los paquetes oficiales utilizados por ROS), pudiendo modificar, compilar o añadir nuevos paquetes.

Al igual que en el entorno real, se puede crear un directorio en el directorio personal del usuario que va a contener el workspace:

```
$ mkdir -p ~/ROS/ws/src
$ cd ~/ROS/ws/src
$ catkin_init_workspace
```

Con el último comando, se inicia el workspce, creando un archivo de instrucciones de Cmake. Por último, es necesario compilar estas instrucciones:

```
$ cd ~/ROS/ws
$ catkin_make
```

Tras finalizar la compilación, el directorio del workspace estará estructurado como un workspace válido. Al igual que es necesario añadir a los archivos de configuración de la shell las variables de ROS, también es recomendable añadir los del workspace:

```
$ echo "source ~/ROS/ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Crear un paquete

Es posible crear un nuevo paquete catkin en un workspace existente, para ello es necesaria escribir en una terminal lo siguiente:

```
$ cd ~/ROS/ws/src
$ catkin_create_pkg myps4eye
```

En este caso se ha creado un paquete llamado *"myps4eye"*. Con este último comando se creará una carpeta nueva con los archivos de configuración necesarios para compilar el paquete. Si se modifica el código fuente, en un paquete del workspace, es necesario compilarlo de la siguiente forma:

```
$ cd ~/ROS/ws
$ catkin_make
```

Sistema de archivos

Una de las mayores ventajas que tiene ROS está en el sistema de archivos. Aunque los proyectos y los paquetes están almacenados en directorios, existen una serie de comandos muy útiles para acceder, o modificar rápidamente paquetes.

Mostrar archivos:

```
$ rosls [ruta[/subdirectorio]]
```

Cambiar de directorio:

```
$ roscd [ruta[/subdirectorio]]
```

Editar archivo de un paquete:

```
$ rosed [paquete] [archivo]
```

Buscar paquetes:

```
$ rospack find [paquete]
```

4.1.2. Instalación y configuración de PlayStation Camera

La PlayStation Camera solamente tiene soporte oficial para la consola PlayStation 4 de Sony. Sin embargo varios desarrolladores de forma independiente han trabajado en hacer funcionar la cámara en PC. Por otra parte también se ha desarrollado un paquete no oficial para ROS que permite la utilización de la cámara como si de una cámara estéreo más se tratase [12].

Requisitos

En primer lugar se necesita el modelo CUH-ZEY1J de PlayStation Camera (hasta el momento el único modelo disponible en el mundo). La principal dificultad con la que cuenta es que es necesario alterar el cable de la cámara para modificar el puerto propietario por un puerto USB 3.0 estándar [14].

En segundo lugar es necesario utilizar una versión de kernel superior a la 3.17.3. Para descargar esta versión concreta podemos realizar las siguientes acciones en una terminal:

```
$ wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v3
  ↪ .17.3-vivid/linux-headers-3.17.3-031703_3
  ↪ .17.3-031703.201411141335_all.deb
$ wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v3
  ↪ .17.3-vivid/linux-headers-3.17.3-031703-generic_3
  ↪ .17.3-031703.201411141335_amd64.deb
$ wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v3
  ↪ .17.3-vivid/linux-image-3.17.3-031703-generic_3
  ↪ .17.3-031703.201411141335_amd64.deb
```

Y para instalar:

```
$ sudo dpkg -i linux-headers-3.17.3*.deb linux-image
  ↪ -3.17.3*.deb
```

También es necesario instalar el paquete Pyusb para que funcione correctamente:

```
$ which pip > /dev/null 2>&1 || sudo apt-get install
  ↪ python-pip
$ sudo pip install --pre pyusb
```

Instalación

Para instalar el paquete de ROS necesitamos clonar el repositorio del proyecto que está alojado en Github:

```
$ cd ~/ROS/ws
$ git clone https://github.com/longjie/ps4eye.git
```

Se necesitan añadir las reglas correspondientes de la cámara en el control de dispositivos de Linux (udev). Para ello ya podemos trabajar con ROS para interactuar con el paquete:

```
$ rosrun ps4eye create_udev_rules
```

Una vez hecho, si se conecta la cámara al ordenador mediante un puerto USB 3.0 se debería poder visualizar la cámara con cualquier aplicación de captura de cámara web. Para comprobar el funcionamiento ejecutamos el siguiente launch del paquete:

```
$ roslaunch ps4eye stereo.launch DEVICE:=/dev/video0
  ↪ viewer:=true
```

En este caso se supone que la cámara está en *"/dev/video0"*, aunque si se utiliza un portátil u otra cámara es necesario que corresponda con otro dispositivo. Se debería visualizar la cámara izquierda y la cámara derecha junto con una imagen de disparidad generada a partir de ambas imágenes.

4.2. Desarrollo con PlayStation Camera

Una vez preparado el entorno de desarrollo: haber instalado y configurado ROS, y posteriormente haber añadido PlayStation Camera solo falta recoger secuencias de imágenes con la cámara. En esta sección se va a explicar como recoger imágenes de la PlayStation Camera, para el análisis posterior de las mismas y la reconstrucción del mapa 3D.

4.2.1. Introducción

Una primera aproximación del problema de la reconstrucción del mapa y la localización en el mismo, es utilizar solamente PlayStation Camera. Haciendo

repaso de lo que se vio en el capítulo 2, la odometría es uno de los pilares de la navegación en robots. Sin embargo, la odometría es muy susceptible a la acumulación de errores, por lo que es necesario contar con todas las medidas necesarias para acotar estos problemas.

Utilizando solamente PlayStation Camera junto con un ordenador para recoger muestras, la única odometría que tiene presencia es la odometría visual. Esto es un punto muy a tener en cuenta en el desarrollo y las medidas que se planean hacer.

Para simular el comportamiento de un robot, la cámara se integrará en un carro industrial y estará conectado al ordenador encargado de recoger los datos mediante ROS. Por otra parte, es necesario contar con un lugar de grabación adecuado, en esta ocasión se hará uso de un hangar de tamaño medio en el que las condiciones lumínicas son idóneas.

4.2.2. Reconstrucción del mapa

La idea general para la reconstrucción del mapa tridimensional es:

- Recoger imágenes de las bolsas de datos
- Rectificación de las imágenes
- Cálculo de la disparidad y odometría
- Construir mapa a partir de la odometría

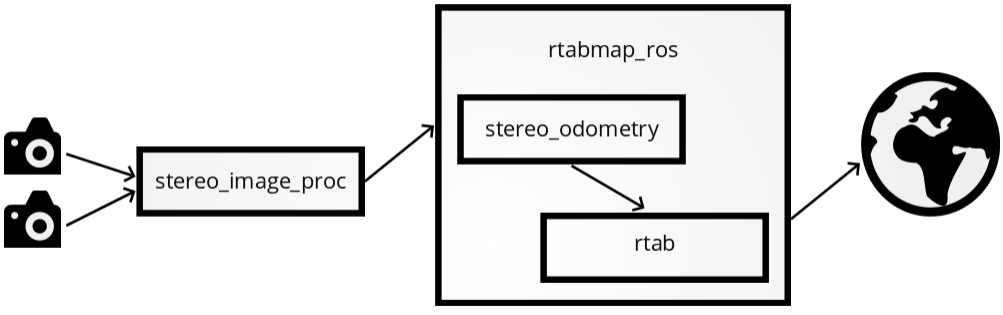

Figura 4.1: Esquema de ROS para construir un mapa 3D

Rectificación

A partir del cálculo del mapa de disparidad a partir de las imágenes rectificadas, es posible obtener una odometría visual. Volviendo a recordar uno de los

objetivos de ROS, este es un ejemplo de aquellas tareas que dificultan el desarrollo de un robot. Para solucionar este problema ROS cuenta con un paquete llamado *"stereo_image_proc"* [32].

Este paquete cuenta con un nodo que recibe el mismo nombre, el cuál tiene como objetivo procesar las imágenes estéreo que recibe. El nodo está suscrito a cuatro tópicos: *"left/image_raw"*, *"'left/camera_info"*, *"right/image_raw"* y *"right/camera_info"*, los cuales corresponden con las imágenes en formato RAW sin comprimir de la cámara y los metadatos asociadas a ellas.

Con el nodo ROS Master ejecutándose en segundo plano, para ejecutar el nodo *"stereo_image_proc"* basta con introducir en una terminal:

```
$ rosrun stereo_image_proc stereo_image_proc __ns:=
  ↪ stereo
```

Mientras, el nodo se encarga de publicar varios tópicos como la imágenes en grises, a color y las imágenes correctamente rectificadas, también está preparado para publicar el mapa de disparidad y una nube de puntos. Tanto las tareas de procesamiento de la rectificación, como la disparidad y la nube de puntos son bajo demanda, por lo que si ningún nodo está suscrito a estos tópicos no se utilizarán recursos para ejecutar las tareas.

Para poder visualizar las imágenes ya procesadas, es posible utilizar una de las utilidades ya comentadas de RQt: *"rqt_image_view"* [30]. En una terminal se introduce lo siguiente:

```
$ rosrun image_view disparity_view image:=/stereo/
  ↪ disparity
```

El nodo *"disparity_view"* está preparado para mostrar los tópicos correspondiente a los mensajes de tipo *"stereo_msgs/DisparityImage"* publicados por *"stereo_image_proc"* en el tópico *"/stereo/disparity"*. De esta forma como argumento se selecciona dicho tópico. El resultado es una imagen de disparidad de color, en función del nivel de disparidad presente en un punto concreto de las imágenes, el tono de color sera diferente.

Odometría

Teniendo ya las imágenes izquierda y derecha rectificadas, con el paquete *"rtabmap_ros"* [31]es posible obtener la odometría visual de las imágenes. Este paquete está basado en RTAB-Map, herramienta basada en técnicas de SLAM para la detección de cierres de bucle en tiempo real. Cuenta con muchas carac-

terísticas, por lo que lo mejor es explicar alguno de los principales de los nodos que dispone:

- **rtabmap:** es el nodo principal del paquete, encargado de almacenar el grafo del mapa de forma incremental y optimizarlo cuando se detecta un cierre de bucle.
- **rtabmapviz:** interfaz gráfica para poder visualizar los resultados de rtabmap.
- **Visual Odometry:** nodo general para la obtención de la odometría visual, el cual se puede dividir en dos.
  1. **rgbd_odometry:** odometría basada en imágenes RGBD.
  2. **stereo_odometry:** odometría basada en imágenes en estéreo.

Para obtener la odometría, y viendo que las imágenes que tenemos son estereoscópicas, es necesario utilizar el nodo *"stereo_odometry"*. Este nodo está suscrito a los tópicos de las imágenes rectificadas y a sus metadatos, y el resultado es un tópico de tipo *"nav_msgs/Odometry"*.

Con la odometría lista, faltaría la reconstrucción del mapa en 3D. El nodo principal es el encargado de esta tarea, a partir de la odometría y la transformación de coordenadas correspondiente, teniendo como resultado el tópico *"rtabmap_ros/MapData"* con toda la información acerca del mapa.

Para lanzar el paquete *"rtabmap_ros"* es necesario introducir en una terminal:

```
$ roslaunch myps4eye carrito.launch
```

La utilidad *"roslaunch"* permite cargar un archivo de este tipo. Se trata de un archivo basado en XML en el que se definen los paquetes, los nodos y las opciones que se desean lanzar, en vez de ser ejecutadas directamente desde terminal. Se trata de la mejor opción cuando se desean lanzar muchos nodos a los cuáles hay además que modificar sus parámetros.

Este launch puede encontrarse en el anexo A.1, y en él se puede ver como se lanzan los nodos *"stereo_odometry"* y *"rtabmap"* de *"'rtabmap_ros"*, y de forma opcional también el nodo *"rtabmapviz"* y otros paquetes de visualización como *"rviz"* y el paquete *"tf"*.

El paquete *"tf"* [33]se encarga de determinar las coordenadas de transformación de cada uno de los elementos de un robot. El nodo *"static_transform_publisher"* se encarga de realizar una transformación de coordenadas a partir de los ejes x,y,z.

Reconstrucción y visualización

Con *"rtabmap"* o *"rtabmapviz"* se puede seleccionar dos modos de trabajo: construcción o localización. Para poder localizarse, es necesario tener un mapa construido previamente. Lanzando el launch anterior se puede observar en *"rtabmapviz"* como se va generando el mapa que las imágenes de PlayStation Camera van registrando en base a la odometría. La reconstrucción del mapa se puede realizar en tiempo real o con bolsas de datos con secuencias ya grabadas. En tal caso, solo bastaría con reproducir la bolsa para poder generar el mapa:

```
$ rosbag play --clock carrito.bag
```

El mapa es generado mediante una nube de puntos. Además de la nube de puntos, se puede observar también como *"rtabmap"* tiene presente el cálculo de cierres de bucle. De esta forma, si se utiliza una secuencia donde se produce se comienza desde un punto determinado, y se acaba volviendo al mismo punto, *"rtabmap"* se encargará de detectar este cierre para poder optimizar el mapa.

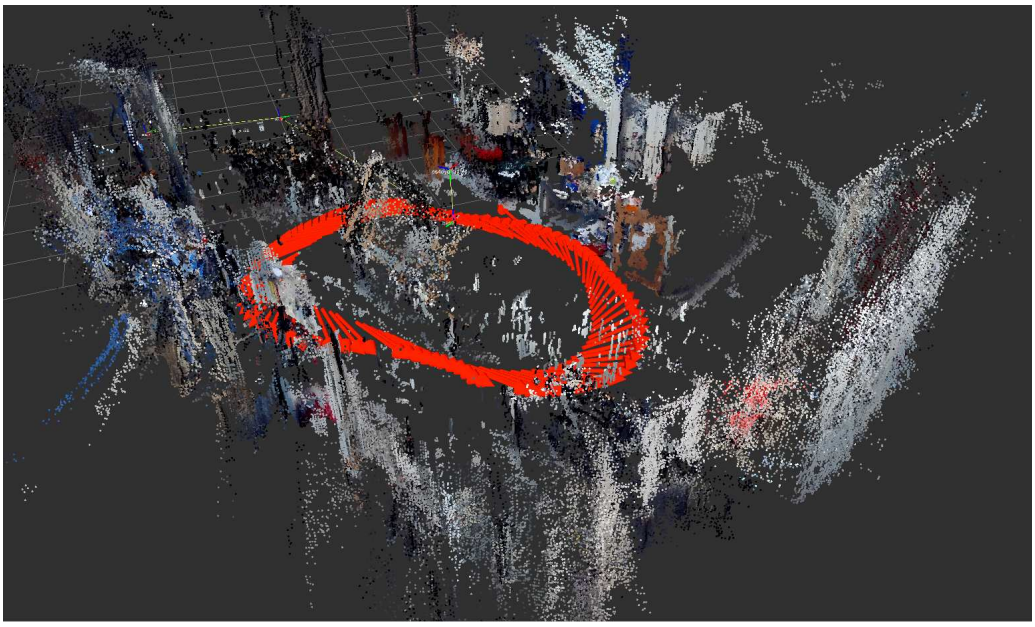

Figura 4.2: Reconstrucción 3D del mapa en Rviz

Tras haber construido el mapa, los resultados pueden visualizarse con *"rviz"*. En la figura 4.2 se puede observar como se ve la nube de puntos que representa al mapa 3D y la odometría del Robot. En este caso se dio dos vueltas alrededor de la misma estructura.

4.2.3. Ajuste de parámetros

El paquete *"stereo_image_proc"* cuenta con una serie de parámetros que pueden modificarse de manera dinámica utilizando la herramienta *"rqt_reconfigure"*. Para lanzarlo se escribe en una terminal:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Gracias a esto podemos obtener mejores resultados. Aquí una pequeña descripción:

- **Prefiltrado:**
  1. **prefilter_size:** tamaño de normalización de la imagen.
  2. **prefilter_cap:** cota normalizada de los píxeles.
- **Correspondencia:**
  1. **correlation_window_size:** diferencia de disparidad.
  2. **min_disparity:** valor mínimo de disparidad.
  3. **disparity_range:** número de disparidades que buscar.
- **Postfiltrado:**
  1. **uniqueness_ratio:** filtro de sigularidades.
  2. **texture_threshold:** disminución del ruido de fondo.
  3. **speckle_size:** tamaño mínimo de las regiones a calcular.
  4. **speckle_range:** máxima diferencia entre disparidades.

4.2.4. Conclusiones

Tras obtener el mapa y la odometría, el resultado no dista demasiado de la realidad. Sin embargo hay que tener en cuenta de que se ha trabajado en un entorno con las condiciones lumínicas necesarias y donde el espacio recorrido ha sido muy pequeño.

La odometría visual es suficiente para conseguir resultados aceptables, sobretodo en ambientes exteriores donde otros sistemas como los RGB-D (Kinect) no pueden trabajar correctamente, pero si se sale de ambientes controlados con picos de luz como en el exterior con mucho sol o en interiores con poca luminosidad, los resultados pueden no tener el efecto esperado.

Es por ello, que habitualmente se utilizan más información para contrastar la odometría visual, como los sistemas de odometría que utiliza la silla del proyecto Perenquén.

4.3. Desarrollo en Perenquén

Como se ha señalado en la sección 4.2, la odometría visual no es suficiente para un sistema de navegación robusto. La silla Perenquén ofrece odometría mecánica y odometría láser, por lo que entre las tres ofrecerán mejores resultados.

4.3.1. Introducción

Figura 4.3: PlayStation Camera en la silla Perenquén

Al igual que sucedió anteriormente, el objetivo sigue está en reconstruir un mapa en tres dimensiones a través de las imágenes capturadas por PlayStation Camera, aunque en esta ocasión, no solamente la cámara estará conectada con el ordenador, también la silla Perenquén estará conectada a él. La silla destaca por contar con un sistema de odometría mecánica y tres sistemas láser, uno a cada lado y otro en la parte trasera.

En esta ocasión no solo se va trabajar con bolsas de datos ya grabadas, también se trabajará en recoger y analizar las imágenes en tiempo real. De esta forma, es necesario modificar las instrucciones descritas en el archivo de configuración utilizado hasta el momento A.1, añadiendo principalmente la adquisición e integración del resto de sistemas de odometría.

4.3.2. Integración de los sistemas de odometría

Ahora la idea general para la reconstrucción del mapa tridimensional pasa por:

- Ajustar cámara y obtener imágenes en estéreo
- Obtener una odometría a partir de la visual, mecánica y láser
- Construir el mapa

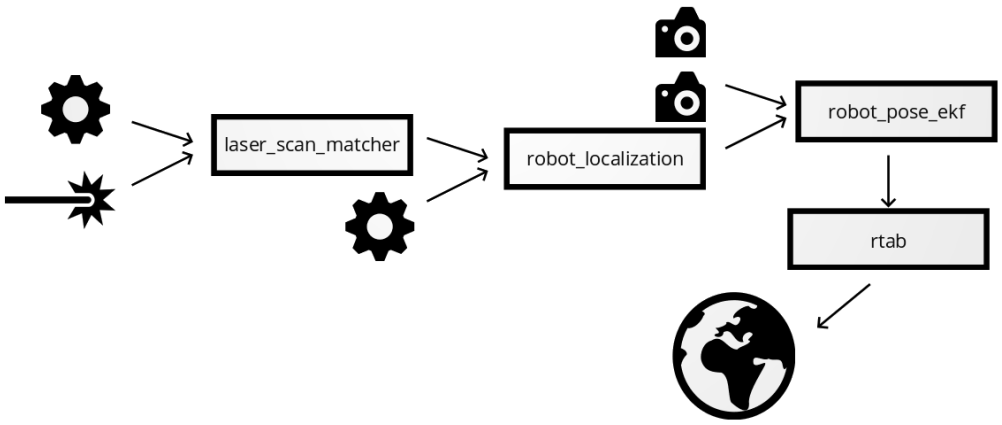

Figura 4.4: Esquema de ROS para construir un mapa 3D integrando las odometrías.

Ajustar cámara

Para poder capturar imágenes directamente desde PlayStation Camera es necesario utilizar el paquete de ROS *"ps4eye"* que se instaló previamente. Este paquete dispone de dos utilidades muy importantes, por un lado un script para la calibración de las cámaras y por otro lado la descomposición de los datos que obtienen ambas cámaras, en dos imágenes diferentes.

Aunque se podría pensar, que cada cámara debería capturar una imagen a su resolución nativa, por ejemplo dos imágenes con una resolución de 640x400 píxeles cada una, la realidad es muy diferente. Si se intenta utilizar PlayStation Camera como una webcam tradicional, el resultado en este caso sería una sola imagen con una resolución de 1748x408 píxeles.

Es necesario utilizar el paquete de ROS *"gscam"* [25]. Se trata de un driver que permite utilizar el framework multimedia GStreamer para conectar cualquier dispositivo de vídeo, como en este caso PlayStation Camera. A este paquete solo es necesario especificarle las características de la cámara: de que dispositivo se trata, el formato de las imágenes, la resolución de captura y la tasa de fotogramas por segundo. De esta forma podemos acceder a la imagen raw que arroja PlayStation Camera.

Posteriormente, se necesita convertir en dos imágenes separadas. Para ello se utiliza el paquete *"image_proc"*. Este paquete es muy similar al de *"stereo_image_proc"*, sin embargo este trabaja solamente con una imagen. Contiene un nodo con el mismo nombre cuyo objetivo principal es el de rectificar la imagen. Aunque, el principal interés radica en la disponibilidad de una utilidad denominada *"image_proc/crop_decimate"* [26] que recibe la imagen de la cámara y permite obtener una imagen nueva a partir de recortar y ajustar la imagen original según las necesidades. En este caso, se desea obtener una nueva imagen izquierda y derecha, cuya resolución será de 640x480.

Si se lanza la herramienta *"image_view"* es posible observar las dos nuevas imágenes correspondiente a lo que ve la cámara izquierda y la derecha a partir de la imagen original. Esto se puede observar en el launch correspondiente descrito en el apéndice A.2.

Calibración

En el caso anterior, para poder obtener las dos nuevas imágenes es necesario que las cámaras estén ya calibradas. La calibración es el proceso en el que se comparan los valores reales obtenidos por las cámaras respecto a las de un patrón de referencia. La formas habitual para realizar esto, es utilizando el método de calibración de Zhang, mediante un plano 2D previamente establecido.

En ROS existe el paquete de *"camera_calibration"* [24], el cuál permite calibrar de una manera muy sencilla cualquier cámara monocular o estéreo mediante el uso de un tablero de de damas o ajedrez como el de la figura 4.5. Este paquete dispone de un nodo llamado *"cameracalibrator.py"*, una aplicación gráfica en la que se pueden visualizar las imágenes que captura PlayStation Camera. Enfocando el tablero hacia la cámara y enseñándole varias perspectivas, se consigue ajustar la calibración de la cámara.

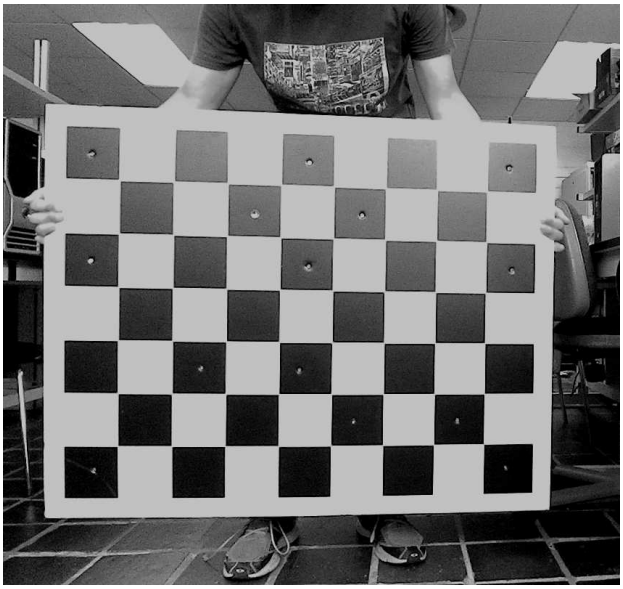

Figura 4.5: Calibración de las cámaras.

El paquete *"ps4eye"* ya cuenta con un script de calibración que cuenta con algunos ajustes básicos como el número de cuadrados y el tamaño de cada uno en el tablero. Basta con ejecutar:

```
$ roslaunch ps4eye calib.launch
```

Que sería equivalente a ejecutar directamente:

```
$ rosrun camera_calibration cameracalibrator.py --size
  ↪ 8x6 --square 0.108 --no-service-check right:=/
  ↪ stereo/right/image_raw left:=/stereo/left/
  ↪ image_raw right_camera:=/stereo/right left_camera
  ↪ :=/stereo/left
```

Tras el proceso de calibración, los resultados se pueden exportar para no tener que calibrar la cámara cada vez que se vaya a usar. Los archivos de calibración se cargados por son cargados en el launch.

Odometría

A partir de la silla Perenquén y PlayStation Camera se puede obtener:

- Odometría láser
- Odometría mecánica
- Odometría visual

Recordando lo que se vio en la sección anterior, la reconstrucción del mapa se consigue mediante el nodo *"rtabmap"* del paquete *"rtabmap_ros"*. Este nodo está suscrito a un tópico *"odom"* que contiene la odometría. Por lo que a partir de las tres odometrías anteriores, se necesita obtener una única odometría, combinación de las tres anteriores, una tarea para la que es necesario utilizar los siguientes paquetes:

- **laser_scan_matcher:** a partir de un láser, este paquete permite integrar una odometría [27].
- **robot_localization:** permite estimar el estado de la posición a partir de los sensores aplicando el Filtro de Kalman Extendido [28].
- **robot_pose_ekf:** permite estimar la pose 3D del robot a partir de combinación entre la odometría mecánica y la odometría visual [29].

El resultado final, es una odometría combinación de las anteriores. A partir de aquí el procedimiento sería ponerlo en funcionamiento en *"rtabmapviz"*.

4.3.3. Problemas encontrados en la implementación

A lo largo del proceso de implementación de los tres sistemas de odometría en la silla Perenquén surgieron algunos problemas que son dignos de mención debido a la peculiaridad de los mismos.

Sincronización entre la odometría mecánica y la odometría visual

Tras implementar los paquetes anteriores, la odometría resultante no funciona correctamente. El problema se encuentra en el paquete *"robot_pose_ekf"* encargado de combinar la odometría visual con la odometría mecánica aplicada sobre los diferentes láseres. Por defecto, ambas odometrías no están sincronizadas produciendo unos resultados extraños.

Para solucionar este problema, es necesario modificar parte del código fuente del paquete:

```
OdomEstimationNode::OdomEstimationNode(){
  //  ...
  pose_pub_ =
    nh_private.advertise<geometry_msgs::
      ↪ PoseWithCovarianceStamped>(
      "pose_combined",
      10
    );
  odom_pub_ =
    nh_private.advertise<nav_msgs::Odometry>(
      "odom_combined",
      10
    );
  odom_cam_pub_ =
    nh_private.advertise<nav_msgs::Odometry>(
      "odom_cam_sync",
      10
    );

  filter_stamp_ = Time::now();
  if (camera_sync_used_)
    camera_sync_sub_ =
      nh.subscribe(
        camera_info_sync_,
        10,
        &OdomEstimationNode::camera_sync_Callback,
        this
      );
  //  ...
}

//  ...
```

```
void OdomEstimationNode::camera_sync_Callback(
  const sensor_msgs::CameraInfo& cam_info
) {

  output_odom_.header.stamp.sec =
    cam_info.header.stamp.sec;
  output_odom_.header.stamp.nsec =
    cam_info.header.stamp.nsec;
  odom_cam_pub_.publish(output_odom_);

  return;
}
//  ...
```

Se definen tres publicadores: *"pose_pub_"*, *"odom_pub_"* y *"odom_cam_pub_"*, y un suscriptor: *"camera_sync_sub"*.

Se crea un método nuevo llamado *"camera_sync_Callback"* que es invocado cuando se especifica el parámetro *"sync"* en un launch. El método se encarga de sincronizar ambas odometrías, igualando el registro del tiempo (timestamp) de la odometría mecánica con la odometría visual de la cámara. Posteriormente se publica la odometría sincronizada a un nuevo tópico llamado *"odom_cam_sync"*.

Covarianza de la odometría mecánica y la odometría láser

Una vez funcionando correctamente, surge un nuevo problema. No se puede utilizar *"rtabmapviz"* para la reconstrucción del mapa.

El problema esta vez está relacionado con los valores de covarianza de la odometría. La covarianza determina el margen de error, cuanto más pequeña, más fiable es. Se debe forzar a que la odometría final resultante tenga unos valores de covarianza por defecto, en caso de salirse de los márgenes establecidos previamente.

Para solucionar este problema, es necesario modificar una vez más parte del código fuente del paquete:

```
//  ...
  my_filter_.getEstimate(output_);
  pose_pub_.publish(output_);

  output_odom_.pose = output_.pose;
  output_odom_.header = output_.header;
```

```
  output_odom_.child_frame_id = "base_link";

  for (int i=0; i<=35; i+=7) {
    if (output_odom_.pose.covariance[i] < 1e-9)
      output_odom_.pose.covariance[i] = 1e-9;
    output_odom_.twist.covariance[i] = 1;
  }
// ...
```

4.3.4. Resultados en interiores

Estos son los resultados en el pasillo que comunica la cafetería de la facultad de Física y Matemáticas con la salida al parking.

En la figura 4.8 se puede observar el resultado de la combinación de las odometrías. En color azul se puede apreciar la odometría final (láser, mecánica y visual) la cuál va acorde con el mapa. La línea amarilla representa a la odometría mecánica, y está ligeramente desplazada debido a un error al comienzo de la grabación de las imágenes, probablemente porque la silla giró demasiado rápido provocando una incertidumbre.

Figura 4.6: Vista aérea del mapa en RTAB-Map

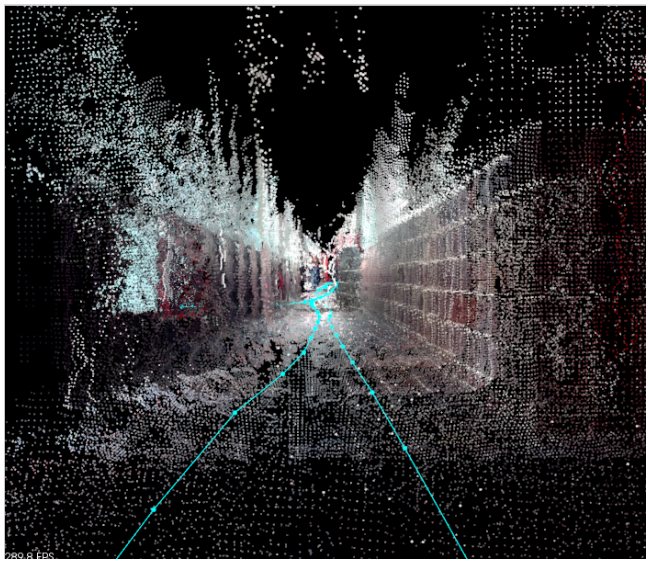

Figura 4.7: Vista dentro del mapa en RTAB-Map

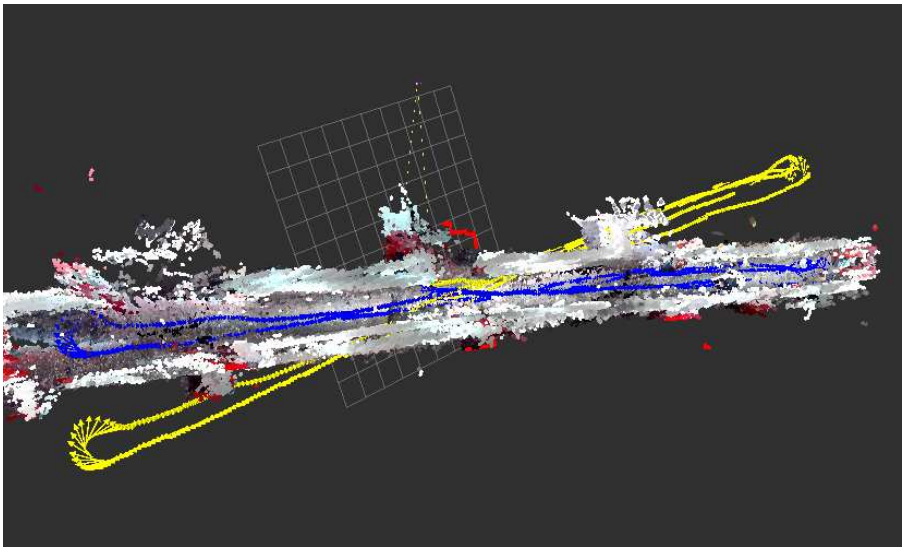

Figura 4.8: Comparación de la odometría en RViz

4.3.5. Resultados en exteriores

Los resultados en los exteriores se tomaron de la Avenida de la Trinidad, en La Laguna.

En líneas generales, los resultados son buenos, aunque como se puede apreciar en la figura 4.9, el mapa sufre de extraños saltos. No es de extrañar que los resultados pueden variar respecto a la realidad, y es que la odometría es muy susceptible a pequeños fallos. En este caso, se debe debido a algunos giros bruscos a la hora de esquivar los objetos y/o peatones que surgían durante la grabación de la escena.

Al igual que en el caso de interiores, se produce una diferencia sustancial entre la odometría combinada respecto a la mecánica como se puede observar

en la figura 4.11.

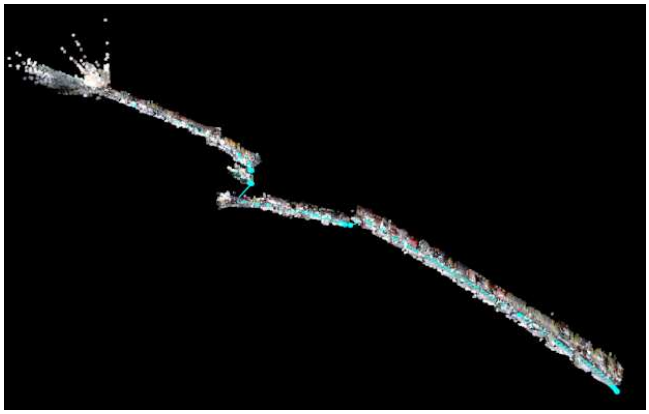

Figura 4.9: Vista aérea del mapa en RTAB-Map

Figura 4.10: Vista dentro del mapa en RTAB-Map

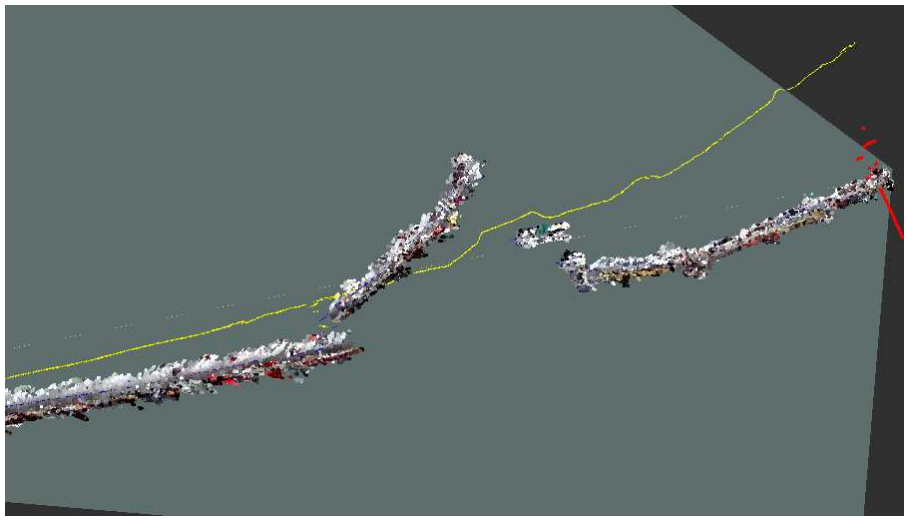

Figura 4.11: Comparación de la odometría en RViz

4.3.6. Mejoras en la adquisición de imágenes

Es posible mejorar la adquisición de las imágenes ajustando los parámetros que dispone la cámara. Para interactuar con la cámara, en GNU/Linux está la posibilidad de utilizar la herramienta *"v4l2-ctl"* que permite interactuar con los dispositivos conectados como webcams. Por defecto, la luminosidad de las imágenes capturadas por la PlayStation Camera es muy limitada, es necesario modificar el parámetro de exposición. En una consola introducimos la siguiente orden:

```
$ v4l2-ctl -d /dev/video0 -c exposure_auto=0
```

De esta forma, obtenemos unas imágenes ricas en luz, siendo de gran ayuda para la odometría visual en espacios cerrados.

Por otra parte, hasta el momento se ha trabajado a una resolución de 1748x408 (640x480 cada cámara). Esta no es la máxima resolución que permite PlayStation Camera. Es posible modificar la configuración utilizada hasta el momento para que se fuerce el uso de una resolución de 3448x808 (1280x800 cada cámara). A mayor resolución de las imágenes, con mayor exactitud se puede obtener información del entorno.

Para utilizar esta resolución, es necesario tener en cuenta que al igual que se hizo anteriormente, es necesario calibrar la cámara para poder obtener las imágenes en estéreo de la imagen original.

El único inconveniente de utilizar esta resolución está en el coste de procesamiento, de ahí la razón por la que solo se ha utilizado de forma experimental. De esta forma con una resolución de 1748x408 la tasa de refresco al que se trasmiten las imágenes ha estado entre los 120 Hz en el mejor caso y los los 30 Hz en el peor caso, aunque en ambos, es una tasa de refresco más que suficiente para trabajar sin ningún inconveniente. En contrapartida, con la resolución de 3448x808 los resultados han sido más pobres, pudiendo operar a unos 40 Hz en el mejor caso, pero a aproximadamente 2 Hz en el peor caso, siendo imposible su implementación para obtener bueno valores de odometría visual.

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

En este trabajo se ha visto que las cámaras estereoscópicas permiten obtener unas imágenes del mundo que le rodea muy próximas a la realidad. Por sí sola, la odometría visual de este tipo de cámaras funciona muy bien en la mayoría de situaciones, tanto en espacios cerrados como en lugares más abiertos, siendo un claro competidor del sistema actual de la silla Perenquén, el uso de una cámara RGB-D. Por otro lado, es importante mencionar que la integración de sensores mecánicos y sensores láser permite corregir algunos de los problemas que son propensos a ocurrir cuando las ocasiones lumínicas no son las idóneas.

Es necesario recordar, que es muy difícil recoger con exactitud la información de la naturaleza, el entorno que nos rodea está vivo, sin embargo, la visión estereoscópica sirve de pilar, junto con otros sensores para obtener la mayor aproximación posible del mundo.

5.2. Líneas futuras

Más allá de la reconstrucción del mapa en 3D y la localización en el mismo, el siguiente punto más atractivo es la detección y posterior esquiva de los obstáculos en el camino del robot. Con este objetivo cumplido, el robot podría navegar de forma autónoma.

Por otro lado, en función de los buenos resultados con la visión estéreo, se debería analizar que otros sistemas alternativos a PlayStation Camera existen.

Capítulo 6

Summary and Conclusions

6.1. Conclusions

In this project we have seen that stereoscopic cameras allow to obtain images of the world very close to reality. Use only visual odometry of these cameras works fine in most situations both indoors and in more open places, being a clear competitor to the current system of the Perenquén project, using a camera RGB-D. Furthermore, it is important to mention that the integration of mechanical sensors and laser sensors allow to correct some of the problems that are likely to happen when the lighting conditions are not ideal.

It is need to remember, it is very difficult to collect accurate information from nature, the environment around us is alive, however, stereo vision is the key, along with other sensors to obtain the best possible approach of the world.

6.2. Future work lines

Beyond 3D mapping and localization in the map, the next most attractive point is the detection and subsequent dodging of obstacles in the path of the robot. With this goal achieved, the robot could navigate autonomously.

Furthermore, according to good results with stereo vision, it should analyze other existing alternative systems PlayStation Camera.

Capítulo 7

Presupuesto

7.1. Presupuesto total

Elemento	**Precio**
PlayStation Camera	€59.95

Tabla 7.1: Material

Tarea	**Horas**	**Precio**	**Total**
Investigación	80h	€15	€1200
Implementación	120h	€20	€2400
Escritura de la Memoria	40h	€5	€200
			€3800

Tabla 7.2: Tareas

Elemento	**Precio**
Material	€59.95
Tareas	€3800
	€3859.95

Tabla 7.3: Presupuesto total

Apéndice A

Launchs

A.1. Launch: PlayStation Camera en Carro

```
<launch>
  <param name="use_sim_time" type="bool" value="True"/>

  <arg name="rtabmapviz" default="true" />
  <arg name="rviz" default="true" />

  <arg name="rtabmapviz_cfg" default="-d $(find script_ps4eye)/launch/config/rgbd_gui.ini" />
  <arg name="rviz_cfg" default="-d $(find script_ps4eye)/launch/config/ps4eye_verdino.rviz" />

  <arg name="frame_id" default="/base_link"/>
  <arg name="time_threshold" default="0"/>
  <arg name="optimize_from_last_node" default="false"/>
  <arg name="database_path" default="$(find script_ps4eye)/map/rtabmap_carrito2.db"/>
  <arg name="rtabmap_args" default=""/>

  <arg name="stereo_namespace" default="/stereo"/>
  <arg name="left_image_topic" default="$(arg stereo_namespace)/left/image_rect_color" />
  <arg name="right_image_topic" default="$(arg stereo_namespace)/right/image_rect_color" />

  <arg name="left_camera_info_topic"  default="$(arg stereo_namespace)/left/camera_info" />
  <arg name="right_camera_info_topic" default="$(arg stereo_namespace)/right/camera_info" />
  <arg name="approximate_sync" default="false"/>
  <arg name="compressed" default="false"/>

  <arg name="subscribe_scan" default="false"/>
  <arg name="scan_topic" default="/scan"/>

  <arg name="visual_odometry"         default="true"/>
  <arg name="odom_topic"              default="/odom"/>

  <arg name="namespace"               default="rtabmap"/>
  <arg name="wait_for_transform"      default="0.1"/>

  <!-- Odometry parameters: -->
  <arg name="strategy"            default="0" />
  <arg name="feature"             default="6" />
```

```
<arg name="estimation"          default="1" />
<arg name="nn"                  default="3" />
<arg name="max_depth"           default="0" />
<arg name="min_inliers"         default="20" />
<arg name="inlier_distance"     default="0.1" />
<arg name="local_map"           default="1000" />
<arg name="odom_info_data"      default="true" />
<arg name="variance_inliers"    default="true"/>

<group ns="$(arg namespace)">
  <!-- Odometry -->
  <node if="$(arg visual_odometry)" pkg="rtabmap_ros" type="stereo_odometry" name="stereo_odometry" o
    <remap from="left/image_rect"        to="$(arg left_image_topic)"/>
    <remap from="right/image_rect"       to="$(arg right_image_topic)"/>
    <remap from="left/camera_info"       to="$(arg left_camera_info_topic)"/>
    <remap from="right/camera_info"      to="$(arg right_camera_info_topic)"/>

    <param name="frame_id"                 type="string" value="$(arg frame_id)"/>
    <param name="wait_for_transform_duration"       type="double"   value="$(arg wait_for_transform)"
    <param name="approx_sync"              type="bool"   value="$(arg approximate_sync)"/>

    <param name="Odom/Strategy"            type="string" value="$(arg strategy)"/>
    <param name="Odom/FeatureType"         type="string" value="$(arg feature)"/>
    <param name="OdomBow/NNType"           type="string" value="$(arg nn)"/>
    <param name="Odom/EstimationType"      type="string" value="$(arg estimation)"/>
    <param name="Odom/MaxDepth"            type="string" value="$(arg max_depth)"/>
    <param name="Odom/MinInliers"          type="string" value="$(arg min_inliers)"/>
    <param name="Odom/InlierDistance"      type="string" value="$(arg inlier_distance)"/>
    <param name="OdomBow/LocalHistorySize" type="string" value="$(arg local_map)"/>
    <param name="Odom/FillInfoData"        type="string" value="true"/>
    <param name="Odom/VarianceFromInliersCount" type="string" value="$(arg variance_inliers)"/>
  </node>

  <!-- Visual SLAM (robot side) -->
  <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen" args="$(arg rtabmap_args)">
    <param name="subscribe_depth"     type="bool"   value="false"/>
    <param name="subscribe_stereo"    type="bool"   value="true"/>
    <param name="subscribe_laserScan"      type="bool"   value="$(arg subscribe_scan)"/>
    <param name="frame_id"            type="string" value="$(arg frame_id)"/>
    <param name="wait_for_transform_duration"  type="double"   value="$(arg wait_for_transform)"/>
    <param name="database_path"       type="string" value="$(arg database_path)"/>
    <param name="stereo_approx_sync"  type="bool"   value="$(arg approximate_sync)"/>

    <remap from="left/image_rect"        to="$(arg left_image_topic)"/>
    <remap from="right/image_rect"       to="$(arg right_image_topic)"/>
    <remap from="left/camera_info"       to="$(arg left_camera_info_topic)"/>
    <remap from="right/camera_info"      to="$(arg right_camera_info_topic)"/>
    <remap from="scan"                   to="$(arg scan_topic)"/>
    <remap unless="$(arg visual_odometry)" from="odom"  to="$(arg odom_topic)"/>

    <param name="Rtabmap/TimeThr"           type="string" value="$(arg time_threshold)"/>
    <param name="RGBD/OptimizeFromGraphEnd" type="string" value="$(arg optimize_from_last_node)"/>
    <param name="LccBow/MinInliers"         type="string" value="10"/>
    <param name="LccBow/InlierDistance"     type="string" value="$(arg inlier_distance)"/>
    <param name="LccBow/EstimationType"     type="string" value="$(arg estimation)"/>
    <param name="LccBow/VarianceFromInliersCount" type="string" value="$(arg variance_inliers)"/>

    <param name="Mem/IncrementalMemory" type="string" value="false"/>

    <param if="$(arg subscribe_scan)" name="RGBD/OptimizeSlam2D"         type="string" value="true"/
```

```
      <param if="$(arg subscribe_scan)" name="RGBD/LocalLoopDetectionSpace" type="string" value="true"/
      <param if="$(arg subscribe_scan)" name="LccIcp/Type"                  type="string" value="2"/>
  <param if="$(arg subscribe_scan)" name="LccIcp2/CorrespondenceRatio"  type="string" value="0.25"/>
    </node>

    <node if="$(arg rtabmapviz)" pkg="rtabmap_ros" type="rtabmapviz" name="rtabmapviz" args="$(arg rtab
      <param name="subscribe_depth" type="bool"   value="false"/>
      <param name="subscribe_stereo" type="bool"   value="true"/>
      <param name="subscribe_laserScan" type="bool"   value="$(arg subscribe_scan)"/>
      <param name="subscribe_odom_info"  type="bool"   value="$(arg visual_odometry)"/>
      <param name="frame_id" type="string" value="$(arg frame_id)"/>
      <param name="wait_for_transform_duration"   type="double"   value="$(arg wait_for_transform)"/>

      <remap from="left/image_rect"        to="$(arg left_image_topic)"/>
      <remap from="right/image_rect"       to="$(arg right_image_topic)"/>
      <remap from="left/camera_info"       to="$(arg left_camera_info_topic)"/>
      <remap from="right/camera_info"      to="$(arg right_camera_info_topic)"/>
      <remap from="scan"                   to="$(arg scan_topic)"/>
      <remap unless="$(arg visual_odometry)" from="odom"  to="$(arg odom_topic)"/>
    </node>

  </group>

  <!-- Visualization RVIZ -->
  <node if="$(arg rviz)" pkg="rviz" type="rviz" name="rviz" args="$(arg rviz_cfg)"/>
  <node if="$(arg rviz)" pkg="nodelet" type="nodelet" name="points_xyzrgb" args="standalone rtabmap_ros
    <remap from="left/image"        to="$(arg left_image_topic)"/>
    <remap from="right/image"       to="$(arg right_image_topic)"/>
    <remap from="left/camera_info"  to="$(arg left_camera_info_topic)"/>
    <remap from="right/camera_info" to="$(arg right_camera_info_topic)"/>
    <remap from="cloud"             to="voxel_cloud" />

    <param name="decimation"  type="double" value="2"/>
    <param name="voxel_size"  type="double" value="0.02"/>
    <param name="approx_sync" type="bool"   value="$(arg approximate_sync)"/>
  </node>

</launch>
```

A.2. Launch: Visualizar cámaras en Perenquén

```
  <launch>
    <arg name="viewer" default="true" />
    <arg name="manager" default="manager" />
    <arg name="respawn" default="false" />

    <arg     if="$(arg respawn)" name="bond" value="" />
    <arg unless="$(arg respawn)" name="bond" value="--no-bond" />

    <node pkg="nodelet" type="nodelet" name="$(arg manager)" args="manager" output="screen" />

    <arg name="camera_info_file_right"
         default="$(find ps4eye)/camera_info/right.yaml" />
    <arg name="camera_info_file_left"
         default="$(find ps4eye)/camera_info/left.yaml" />
```

```
<arg name="camera_info_url_default" default="file://$(find ps4eye)/camera_info/default.yaml" />
<arg name="camera_info_url_right" default="file://$(arg camera_info_file_right)" />
<arg name="camera_info_url_left" default="file://$(arg camera_info_file_left)" />

<arg name="parent_frame" default="/base_link" />
<arg name="parent_transform" default="0 0 1 -1.57 0 -1.57" />

<arg name="load_driver" default="true" />
<arg name="DEVICE" default="/dev/video0"/>
<arg name="FPS" default="60/1"/>
<arg name="width" default="1748"/>
<arg name="height" default="408"/>
<arg name="PUBLISH_FRAME" default="false"/>
<node name="gscam_driver" pkg="nodelet" type="nodelet"
      args="load gscam/GSCamNodelet $(arg manager)"
      output="screen"
      if="$(arg load_driver)">
  <param name="camera_name" value="default"/>
  <param name="gscam_config" value="v4l2src device=$(arg DEVICE) ! video/x-raw-yuv,framerate=$(arg
  <param name="frame_id" value="/ps4eye_frame"/>
  <param name="sync_sink" value="true"/>
  <param name="camera_info_url" value="$(arg camera_info_url_default)" />
</node>
<!-- Split image to left and right image -->
<node pkg="nodelet" type="nodelet" name="split_right" args="load image_proc/crop_decimate $(arg man
  <param name="camera_info_url" value="$(arg camera_info_url_right)" />
  <param name="queue_size" type="int" value="10" />
  <param name="x_offset" type="int" value="48" />
  <param name="y_offset" type="int" value="0" />
  <param name="width" type="int" value="640" />
  <param name="height" type="int" value="400" />
  <param name="camera_name" value="right"/>
  <remap from="camera_out/image_raw" to="/stereo/right/image_raw" />
  <remap from="/stereo/right/camera_info" to="/null/right/camera_info" />
</node>
<node pkg="nodelet" type="nodelet" name="split_left" args="load image_proc/crop_decimate $(arg mana
  <param name="camera_info_url" value="$(arg camera_info_url_left)" />
  <param name="queue_size" type="int" value="10" />
  <param name="x_offset" type="int" value="688" />
  <param name="y_offset" type="int" value="0" />
  <param name="width" type="int" value="640" />
  <param name="height" type="int" value="400" />
  <param name="camera_name" value="left"/>
  <remap from="camera_out/image_raw" to="/stereo/left/image_raw" />
  <remap from="/stereo/left/camera_info" to="/null/left/camera_info" />
</node>

<node pkg="tf" type="static_transform_publisher" name="camera_transform"
  args="0.35  -0.35 0.54 -1.57 0.0 -1.57 base_link /ps4eye_frame 100" />

<node pkg="ps4eye" type="camera_info_publisher.py" name="camera_info_publisher" >
  <param name="left_file_name"  value="$(arg camera_info_file_left)"  />
  <param name="right_file_name" value="$(arg camera_info_file_right)" />
</node>

<group ns="stereo">
  <include file="$(find stereo_image_proc)/launch/stereo_image_proc.launch">
    <arg name="manager" value="/$(arg manager)" />
  </include>
</group>
```

```
  <node name="stereo_view" pkg="image_view" type="stereo_view" args="stereo:=/stereo image:=image_rec

</launch>
```

A.3. Launch: Combinación de odometría láser y mecánica en Perenquén

```
<launch>
   <include file="$(find roboteq_driver)/launch/bring_up.launch"/>

   <include file="$(find grull_sick_lms100)/launch/launch.launch">
      <arg name="HOST" value="192.168.0.1" />
      <arg name="LASER_FRAME_ID" value="left_laser" />
      <arg name="NODE_NAME" value="left_laser" />
      <arg name="SCAN" value="left_laser" />
   </include>
   <include file="$(find grull_sick_lms100)/launch/launch.launch">
      <arg name="HOST" value="192.168.0.3" />
      <arg name="LASER_FRAME_ID" value="right_laser" />
      <arg name="NODE_NAME" value="right_laser" />
      <arg name="SCAN" value="right_laser" />
    </include>

   <include file="$(find grull_sick_lms100)/launch/launch.launch">
      <arg name="HOST" value="192.168.0.15" />
      <arg name="LASER_FRAME_ID" value="rear_laser" />
      <arg name="NODE_NAME" value="rear_laser" />
      <arg name="SCAN" value="rear_laser" />
    </include>

    <node pkg="laser_filters" type="scan_to_scan_filter_chain" output="screen" name="left_angular_bou
      <remap from="scan" to="left_laser" />
      <remap from="scan_filtered" to="left_laser_filtered"/>
      <param name="target_frame" value="base_link"/>
      <rosparam command="load" file="$(find silla)/filters/leftLaserFilter.yaml"/>
    </node>
     <node pkg="laser_filters" type="scan_to_scan_filter_chain" output="screen" name="right_angular_b
      <remap from="scan" to="right_laser" />
      <remap from="scan_filtered" to="right_laser_filtered"/>
      <param name="target_frame" value="base_link"/>
      <rosparam command="load" file="$(find silla)/filters/rightLaserFilter.yaml"/>
    </node>

     <node pkg="laser_filters" type="scan_to_scan_filter_chain" output="screen" name="rear_angular_bo
      <remap from="scan" to="rear_laser" />
      <remap from="scan_filtered" to="rear_laser_filtered"/>
      <param name="target_frame" value="base_link"/>
      <rosparam command="load" file="$(find silla)/filters/rearLaserFilter.yaml"/>
    </node>


  <node pkg="laser_scan_matcher" type="laser_scan_matcher_node"
    name="laser_scan_matcher_node_left" output="screen">
```

```
    <remap from="odom" to="/perenquen/odom"/>
    <remap from="scan" to="/left_laser_filtered"/>
    <remap from="/pose_with_covariance_stamped" to="/pose_with_covariance_stamped_left" />
    <remap from="/laser_odometry" to="/laser_odometry_left" />

    <param name="fixed_frame" value="odom"/>
    <param name="base_frame" value="base_link"/>

    <param name="max_linear_correction" value="0.15"/>
    <param name="max_angular_correction_deg" value="6"/>

    <param name="do_compute_covariance" value="1"/>
    <param name="publish_pose_with_covariance_stamped" value="true"/>

    <param name="use_imu" value="false"/>
    <param name="publish_tf" value="false"/>
    <param name="max_iterations" value="10"/>
  </node>

<node pkg="laser_scan_matcher" type="laser_scan_matcher_node"
    name="laser_scan_matcher_node_right" output="screen">



    <remap from="odom" to="/perenquen/odom"/>
    <remap from="scan" to="/right_laser_filtered"/>
    <remap from="/pose_with_covariance_stamped" to="/pose_with_covariance_stamped_right" />
    <remap from="/laser_odometry" to="/laser_odometry_right" />

    <param name="fixed_frame" value="odom"/>
    <param name="base_frame" value="base_link"/>

    <param name="max_linear_correction" value="0.15"/>
    <param name="max_angular_correction_deg" value="6"/>

    <param name="do_compute_covariance" value="1"/>
    <param name="publish_pose_with_covariance_stamped" value="true"/>

    <param name="publish_tf" value="false"/>
    <param name="use_imu" value="false"/>
    <param name="max_iterations" value="10"/>
  </node>


 <node pkg="robot_pose_ekf" type="robot_pose_ekf" name="robot_pose_ekf">
    <param name="output_frame" value="odom"/>
    <param name="base_footprint_frame" value="base_link"/>
    <param name="freq" value="30.0"/>
    <param name="sensor_timeout" value="1.0"/>
    <param name="odom_used" value="true"/>
    <param name="imu_used" value="false"/>
    <param name="vo_used" value="false"/>

  <param name="camera_sync_used" value="true"/>
  <param name="camera_info_sync" value="/stereo/left/camera_info"/>
```

```
    <remap from="odom" to="/odometry/filtered" />
    <remap from="vo" to="/laser_odom" />
    <remap from="/robot_pose_ekf/odom_combined" to="/odom_combined" />

  </node>

 <node pkg="tf" type="static_transform_publisher" name="kinect2_to_base_link_static_transform"
      args="0.5 0 0.17 -1.57 0.0 -1.45 base_link kinect2_ir_optical_frame 100" />
    <node pkg="tf" type="static_transform_publisher" name="left_laser_to_base_link_static_transform"
      args="0.35 0.35 0.58 1.57 0 0 base_link left_laser 100" />
    <node pkg="tf" type="static_transform_publisher" name="right_laser_to_base_link_static_transform"
      args="0.35 -0.35 0.58 -1.61 0 0 base_link right_laser 100" />
<node pkg="tf" type="static_transform_publisher" name="rear_laser_to_base_link_static_transform"
      args="-0.30 0 0.5 3.14 0 3.14 base_link rear_laser 100" />

    <node pkg="robot_localization" type="ekf_localization_node" name="ekf_localization" clear_params=

      <param name="frequency" value="30"/>
      <param name="sensor_timeout" value="0.5"/>
      <param name="two_d_mode" value="true"/>
      <param name="map_frame" value="map"/>
      <param name="odom_frame" value="odom"/>
      <param name="base_link_frame" value="base_link"/>
      <param name="world_frame" value="odom"/>

      <param name="transform_time_offset" value="0.0"/>

      <param name="odom0" value="/perenquen/odom"/>
      <param name="pose0" value="/pose_with_covariance_stamped_left"/>
      <param name="pose1" value="/pose_with_covariance_stamped_right"/>

      <rosparam param="odom0_config">[true, true, false,
                                      false, false, true,
                                      true,  false, false,
                                      false, false, false,
                                      false, false, false]</rosparam>

      <rosparam param="pose0_config">[true,  true,  false,
                                      false, false, true,
                                      false, false, false,
                                      false, false, false,
                                      false, false, false]</rosparam>

      <rosparam param="pose1_config">[true,  true,  false,
                                      false, false, true,
                                      false, false, false,
                                      false, false, false,
                                      false, false, false]</rosparam>

      <param name="odom0_differential" value="true"/>
      <param name="pose0_differential" value="false"/>
      <param name="pose1_differential" value="false"/>

      <param name="odom0_relative" value="true"/>
      <param name="pose0_relative" value="true"/>
      <param name="pose1_relative" value="true"/>

      <param name="imu0_remove_gravitational_acceleration" value="true"/>

      <param name="print_diagnostics" value="true"/>
```

```
    <param name="odom0_queue_size" value="10"/>
    <param name="pose0_queue_size" value="10"/>
    <param name="pose1_queue_size" value="10"/>

    <param name="odom0_pose_rejection_threshold" value="1"/>
    <param name="pose0_rejection_threshold" value="1"/>
    <param name="pose1_rejection_threshold" value="1"/>

    <param name="debug"           value="false"/>
    <param name="debug_out_file"  value="debug_ekf_localization.txt"/>


    <rosparam param="process_noise_covariance">[
      0.25, 0,    0,    0,    0,    0,    0,     0,     0,    0,    0,    0,    0,    0,    0,
      0,    0.25, 0,    0,    0,    0,    0,     0,     0,    0,    0,    0,    0,    0,    0,
      0,    0,    0.25, 0,    0,    0,    0,     0,     0,    0,    0,    0,    0,    0,    0,
      0,    0,    0,    0.25, 0,    0,    0,     0,     0,    0,    0,    0,    0,    0,    0,
      0,    0,    0,    0,    0.25, 0,    0,     0,     0,    0,    0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,    0.25, 0,     0,     0,    0,    0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,    0,    0.25,  0,     0,    0,    0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,     0.25,  0,    0,    0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,     0,     0.25, 0,    0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,     0,     0,    0.25, 0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,     0,     0,    0,    0.25, 0,    0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,     0,     0,    0,    0,    0.25, 0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,     0,     0,    0,    0,    0,    0.25, 0,    0,
      0,    0,    0,    0,    0,    0,    0,     0,     0,    0,    0,    0,    0,    0.25, 0,
      0,    0,    0,    0,    0,    0,    0,     0,     0,    0,    0,    0,    0,    0,    0.25]</
    <rosparam param="initial_estimate_covariance">[
      1e-9, 0,    0,    0,    0,    0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
      0,    1e-9, 0,    0,    0,    0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
      0,    0,    1e-9, 0,    0,    0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
      0,    0,    0,    1e-9, 0,    0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
      0,    0,    0,    0,    1e-9, 0,    0,    0,    0,    0,     0,     0,     0,    0,    0,
      0,    0,    0,    0,    0,    1e-9, 0,    0,    0,    0,     0,     0,     0,    0,    0,
      0,    0,    0,    0,    0,    0,    1e-9, 0,    0,    0,     0,     0,     0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,    1e-9, 0,    0,     0,     0,     0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,    0,    1e-9, 0,     0,     0,     0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,    0,    0,    1e-9,  0,     0,     0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,    0,    0,    0,     1e-9,  0,     0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,    0,    0,    0,     0,     1e-9,  0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,    0,    0,    0,     0,     0,     1e-9, 0,    0,
      0,    0,    0,    0,    0,    0,    0,    0,    0,    0,     0,     0,     0,    1e-9, 0,
      0,    0,    0,    0,    0,    0,    0,    0,    0,    0,     0,     0,     0,    0,    1e-9]<


  </node>


</launch>
```

A.4. Launch: Generación del mapa en Perenquén

```
<launch>
  <arg name="rtabmapviz"              default="true" />
  <arg name="rviz"                    default="false" />

  <arg name="localization"            default="false"/>

  <arg name="rtabmapviz_cfg"          default="-d $(find rtabmap_ros)/launch/config/rgbd_gui.ini" />
  <arg name="rviz_cfg"                default="-d $(find rtabmap_ros)/launch/config/rgbd.rviz" />

  <arg name="frame_id"                default="base_link"/>
  <arg name="time_threshold"          default="0"/>
  <arg name="optimize_from_last_node" default="false"/>
  <arg name="database_path"           default="~/.ros/rtabmap.db"/>
  <arg name="rtabmap_args"            default=""/>
  <arg name="launch_prefix"           default=""/>

  <arg name="stereo_namespace"        default="/stereo"/>
  <arg name="left_image_topic"        default="$(arg stereo_namespace)/left/image_rect_color" />
  <arg name="right_image_topic"       default="$(arg stereo_namespace)/right/image_rect" />
  <arg name="left_camera_info_topic"  default="$(arg stereo_namespace)/left/camera_info" />
  <arg name="right_camera_info_topic" default="$(arg stereo_namespace)/right/camera_info" />
  <arg name="approximate_sync"        default="false"/>
  <arg name="compressed"              default="false"/>
  <arg name="convert_depth_to_mm"     default="true"/>

  <arg name="subscribe_scan"          default="true"/>
  <arg name="scan_topic"              default="/left_laser_filtered"/>

  <arg name="subscribe_scan_cloud"    default="false"/>
  <arg name="scan_cloud_topic"        default="/scan_cloud"/>

  <arg name="visual_odometry"         default="false"/>
  <arg name="odom_topic"              default="/robot_pose_ekf/odom_cam_sync"/>

  <arg name="namespace"               default="rtabmap"/>
  <arg name="wait_for_transform"      default="0.2"/>

  <!-- Nodes -->
  <group ns="$(arg namespace)">

    <node if="$(arg compressed)" name="republish_left"  type="republish" pkg="image_transport" args="
    <node if="$(arg compressed)" name="republish_right" type="republish" pkg="image_transport" args="

    <!-- Odometry -->
    <node if="$(arg visual_odometry)" pkg="rtabmap_ros" type="stereo_odometry" name="stereo_odometry"
      <remap from="left/image_rect"        to="$(arg left_image_topic)"/>
      <remap from="right/image_rect"       to="$(arg right_image_topic)"/>
      <remap from="left/camera_info"       to="$(arg left_camera_info_topic)"/>
      <remap from="right/camera_info"      to="$(arg right_camera_info_topic)"/>

      <param name="frame_id"                 type="string" value="$(arg frame_id)"/>
      <param name="wait_for_transform_duration"       type="double"   value="$(arg wait_for_transform
      <param name="approx_sync"              type="bool"   value="$(arg approximate_sync)"/>
```

```
  <param name="Odom/FillInfoData"        type="string" value="true"/>
</node>

<!-- Visual SLAM (robot side) -->
<node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen" args="$(arg rtabmap_args)"
  <param name="subscribe_depth"              type="bool"   value="false"/>
  <param name="subscribe_stereo"             type="bool"   value="true"/>
  <param name="subscribe_scan"               type="bool"   value="$(arg subscribe_scan)"/>
  <param name="subscribe_scan_cloud"         type="bool"   value="$(arg subscribe_scan_cloud)"/>
  <param name="frame_id"                     type="string" value="$(arg frame_id)"/>
  <param name="wait_for_transform_duration"  type="double"   value="$(arg wait_for_transform)"/>
  <param name="database_path"                type="string" value="$(arg database_path)"/>
  <param name="stereo_approx_sync"           type="bool"   value="$(arg approximate_sync)"/>

  <remap from="left/image_rect"        to="$(arg left_image_topic)"/>
  <remap from="right/image_rect"       to="$(arg right_image_topic)"/>
  <remap from="left/camera_info"       to="$(arg left_camera_info_topic)"/>
  <remap from="right/camera_info"      to="$(arg right_camera_info_topic)"/>
  <remap from="scan"                   to="$(arg scan_topic)"/>
  <remap from="scan_cloud"      to="$(arg scan_cloud_topic)"/>
  <remap unless="$(arg visual_odometry)" from="odom"  to="$(arg odom_topic)"/>

  <param name="Rtabmap/TimeThr"              type="string" value="$(arg time_threshold)"/>
  <param name="RGBD/OptimizeFromGraphEnd"    type="string" value="$(arg optimize_from_last_node)"
  <param name="Mem/SaveDepth16Format"        type="string" value="$(arg convert_depth_to_mm)"/>

  <!-- localization mode -->
<param     if="$(arg localization)" name="Mem/IncrementalMemory" type="string" value="false"/>
<param unless="$(arg localization)" name="Mem/IncrementalMemory" type="string" value="true"/>
<param name="Mem/InitWMWithAllNodes" type="string" value="$(arg localization)"/>

  <!-- when 2D scan is set -->
  <param if="$(arg subscribe_scan)" name="Optimizer/Slam2D"        type="string" value="true"/>
<param if="$(arg subscribe_scan)" name="Icp/CorrespondenceRatio" type="string" value="0.25"/>
<param if="$(arg subscribe_scan)" name="Reg/Strategy"            type="string" value="1"/>
<param if="$(arg subscribe_scan)" name="Reg/Force3DoF"           type="string" value="true"/>

<!-- when 3D scan is set -->
<param if="$(arg subscribe_scan_cloud)" name="Reg/Strategy"      type="string" value="1"/>
</node>

<!-- Visualisation RTAB-Map -->
<node if="$(arg rtabmapviz)" pkg="rtabmap_ros" type="rtabmapviz" name="rtabmapviz" args="$(arg rt
  <param name="subscribe_depth"           type="bool"   value="false"/>
  <param name="subscribe_stereo"          type="bool"   value="true"/>
  <param name="subscribe_scan"            type="bool"   value="$(arg subscribe_scan)"/>
  <param name="subscribe_scan_cloud"      type="bool"   value="$(arg subscribe_scan_cloud)"/>
  <param name="subscribe_odom_info"       type="bool"   value="$(arg visual_odometry)"/>
  <param name="frame_id"                  type="string" value="$(arg frame_id)"/>
  <param name="wait_for_transform_duration"   type="double"   value="$(arg wait_for_transform)"/>

  <remap from="left/image_rect"        to="$(arg left_image_topic)"/>
  <remap from="right/image_rect"       to="$(arg right_image_topic)"/>
  <remap from="left/camera_info"       to="$(arg left_camera_info_topic)"/>
  <remap from="right/camera_info"      to="$(arg right_camera_info_topic)"/>
  <remap from="scan"                   to="$(arg scan_topic)"/>
  <remap from="scan_cloud"             to="$(arg scan_cloud_topic)"/>
  <remap unless="$(arg visual_odometry)" from="odom"  to="$(arg odom_topic)"/>
</node>
```

```
  </group>


  <!-- Visualization RVIZ -->
  <node if="$(arg rviz)" pkg="rviz" type="rviz" name="rviz" args="$(arg rviz_cfg)"/>
  <node if="$(arg rviz)" pkg="nodelet" type="nodelet" name="points_xyzrgb" args="standalone rtabmap_r
    <remap from="left/image"        to="$(arg left_image_topic)"/>
    <remap from="right/image"       to="$(arg right_image_topic)"/>
    <remap from="left/camera_info"  to="$(arg left_camera_info_topic)"/>
    <remap from="right/camera_info" to="$(arg right_camera_info_topic)"/>
    <remap from="cloud"             to="voxel_cloud" />

    <param name="decimation"  type="double" value="2"/>
    <param name="voxel_size"  type="double" value="0.02"/>
    <param name="approx_sync" type="bool"   value="$(arg approximate_sync)"/>
  </node>

</launch>
```

Bibliografía

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

[2] Computer Vision. https://en.wikipedia.org/wiki/Computer_vision.

[3] Comunication via Topics vs Services. http://wiki.ros.org/ROS/Patterns/Commun

[4] Google Self-Driving Car. https://www.google.com/selfdrivingcar/.

[5] Información general del modelo de publicación de replicación. https://msdn.microsoft.com/es-es/library/ms152567(v=sql.120).aspx.

[6] Introducción a ROS. http://erlerobotics.com/blog/ros-introduction-es/.

[7] Is ROS for me? http://www.ros.org/is-ros-for-me/.

[8] Kinect. http://www.xbox.com/es-ES/xbox-one/accessories/kinect-for-xbox

[9] Kinect for Windows. https://developer.microsoft.com/en-us/windows/kinect

[10] Odometría mecánica. https://es.wikipedia.org/wiki/Odometr%C3%ADa.

[11] Odometría visual. https://avisingh599.github.io/vision/visual-odometry-f

[12] PlayStation 4 stereo camera package for ROS. https://github.com/longjie/ps4eye.

[13] PlayStation Camera. https://en.wikipedia.org/wiki/PlayStation_Camera.

[14] PlayStation Camera hacking. http://ps4eye.tumblr.com/.

[15] Proyecto Perenquén. http://verdino.webs.ull.es/project-perenquen.html.

[16] Proyecto Verdino. http://verdino.webs.ull.es/.

[17] PS Camera Preguntas Frecuentes. https://blog.es.playstation.com/2013/10/3

[18] PS Camera Specifications. http://www.psdevwiki.com/ps4/PlayStation_4_Came

[19] Robot Navigation. https://en.wikipedia.org/wiki/Mobile_robot_navigation.

[20] Robot Operating System. https://en.wikipedia.org/wiki/Robot_Operating_Sy

[21] Robotic Mapping. https://en.wikipedia.org/wiki/Robotic_mapping.

[22] ROS: Conceptos. http://erlerobotics.com/blog/ros-introduction-es/#conce

[23] ROS Core Components. http://www.ros.org/core-components/.

[24] ROS Packages: CameraCalibration. http://wiki.ros.org/camera_calibration.

[25] ROS Packages: Gscam. http://wiki.ros.org/gscam.

[26] ROS Packages: ImageProcCropDecimate. http://wiki.ros.org/image_proc#image_proc.2BAC8-cturtle.image_proc.

[27] ROS Packages: LaserScanMatcher. http://wiki.ros.org/laser_scan_matcher.

[28] ROS Packages: RobotLocalization. http://wiki.ros.org/robot_localization#e

[29] ROS Packages: RobotPoseEkf. http://wiki.ros.org/robot_pose_ekf.

[30] ROS Packages: RqtImageView. http://wiki.ros.org/rqt_image_view.

[31] ROS Packages: RtabmapRos. http://wiki.ros.org/rtabmap_ros.

[32] ROS Packages: StereoImageProc. http://wiki.ros.org/stereo_image_proc.

[33] ROS Packages: Tf. http://wiki.ros.org/tf.

[34] RTAB-Map. http://introlab.github.io/rtabmap/.

[35] SLAM (robótica). https://es.wikipedia.org/wiki/SLAM_(rob%C3%B3tica).

[36] Stereo Vision Introduction and Applications. https://www.ptgrey.com/tan/10570.

[37] Ubuntu. http://www.ubuntu.com/download.

[38] Ubuntu install of ROS Indigo. http://wiki.ros.org/indigo/Installation/Ubun

[39] Ingenieros de la ull crean un sistema que da autonomía a las sillas de ruedas. *La Opinión de Tenerife*, Mayo 2016. http://www.laopinion.es/sociedad/2016/05/04/ingenieros-ull-crean-sist

[40] Wolfram Burgard, Cyrill Stachniss, Maren Bennewitz, and Kai Arras. Robot motion planning, Julio 2011. http://ais.informatik.uni-freiburg.de/teaching/ss11/robotics/slides/1

[41] Chen Chi Hau. *Computer Vision in Medical Imaging*, volume 2. World Scientific, January 2014. http://www.worldscientific.com/doi/abs/10.1142/8766.

[42] J. González González. *Visión Por Computador*. Editorial Paraninfo, 2000.

[43] Ramón González Sánchez. Simultaneous And Localization Mapping. http://www.ual.es/personal/rgonzalez/documents/slam_ramon.pdf.

[44] José Miguel Guerrero Hernández, Gonzalo Pajares Martinsanz, and María Guijarro Mata-García. Técnicas de procesamiento de imágenes estereoscópicas. (13):9, 2011.

[45] Federico Lecumberry. Cálculo de disparidad en imágenes estéreo, una comparación. 2005.

[46] Diego Antonio López García et al. Nuevas aportaciones en algoritmos de planificación para la ejecución de maniobras en robots autónomos no holónomos. 2011.

[47] Larry Matthies, Mark Maimone, Andrew Johnson, Yang Cheng, Reg Willson, Carlos Villalpando, Steve Goldberg, Andres Huertas, Andrew Stein, and Anelia Angelova. Computer vision on mars. *International Journal of Computer Vision*, 75(1):67–92, 2007.

[48] Sergio Pereira Ruiz. Pfc: Localización de robots mediante filtro de kalman, Septiembre 2003. http://bibing.us.es/proyectos/abreproy/11879/fichero/PFC+Sergio+Perei

[49] Matt Pharr and Fernando Randima. *GPU Gems 2*. 2 edition, Abril 2005.

[50] Steve Wright. Parallel vs converged. 2011.

[51] Chao Zhou, Yucheng Wei, and Tieniu Tan. Mobile robot self-localization based on global visual appearance features. 1:1271–1276 vol.1, Septiembre 2003.