Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

ESCUELA SUPERIOR DE INGENIERÍA Y TECNOLOGÍA, TITULACIONES DE INGENIERÍA INDUSTRIAL

TRABAJO DE FIN DE GRADO

SISTEMA DE ODOMETRÍA VISUAL PARA LA LOCALIZACIÓN DE ROBOTS MÓVILES

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Estudiante: Jorge Sanz Marrero

Tutor: Jonay Tomás Toledo Carrillo

Cotutora: Bibiana Fariña Jerónimo

Fecha: 25/05/2023

AGRADECIMIENTOS

Me gustaría agradecer a mis tutores Jonay Toledo y Bibiana Fariña por su implicación en el proyecto, ya que han estado siempre disponibles para ayudarme y resolver los problemas que se iban presentando.

También me gustaría agradecer a mi familia y amigos por todo el apoyo que me han ofrecido.

Jorge Sanz Marrero
S/C de Tenerife, 2023

RESUMEN

El presente Trabajo de Fin de Grado se centra en el estudio y comparación de dos metodologías de odometría: la odometría visual y la mecánica, aplicadas en una silla de ruedas autónoma. No obstante, se hará un estudio más exhaustivo de la odometría visual, comparando el rendimiento de diferentes algoritmos. El objetivo principal es mejorar la calidad de vida de las personas con movilidad reducida, proporcionándoles una solución de desplazamiento autónomo.

La silla de ruedas autónoma busca abordar los desafíos diarios a los que se enfrentan estas personas, permitiéndoles desplazarse de forma independiente. Los sistemas de odometría de la silla de ruedas permiten programar rutas, evitar obstáculos, adaptarse a diferentes terrenos y entornos, brindando mayor independencia y libertad a los usuarios. Esta también puede ser utilizada en entornos médicos, como hospitales y centros de rehabilitación, donde facilita el desplazamiento y la realización de movimientos sin supervisión constante.

La silla de ruedas cuenta con dos sistemas de odometría: sensores encoder en cada rueda para la odometría mecánica, y una cámara estéreo cuyas imágenes son procesadas por un algoritmo en ROS para la odometría visual. Además, se utiliza un sensor láser Velodyne basado en tecnología LIDAR como referencia de la trayectoria real. Asimismo, las pruebas realizadas se llevaron a cabo en la Facultad de Física y Matemáticas, y se realizaron recorridos dentro de sus instalaciones.

Recapitulando, este trabajo tiene como objetivo comparar y evaluar el rendimiento de la odometría visual y mecánica en una silla de ruedas autónoma, con el propósito de mejorar la calidad de vida de las personas con movilidad reducida.

Palabras clave: ROS, Rviz, Odometría Visual, Odometría Mecánica, Viso2, RTAB-Map, ORB-SLAM, SLAM, Extracción de características.

ABSTRACT

This final degree project focuses on the study and comparison of two odometry methodologies: visual odometry and mechanical odometry, applied to an autonomous wheelchair. However, a more thorough study will be conducted on visual odometry, comparing the performance of different algorithms. The main objective is to improve the quality of life for people with reduced mobility by providing them with an autonomous mean of transportation.

The autonomous wheelchair aims to solve the daily challenges faced by these individuals, enabling them to move independently. The wheelchair's odometry systems allow for route planning, obstacle avoidance, and adaptation to various terrains and environments, offering greater independence and freedom to the users. It can also be used in medical settings such as hospitals and rehabilitation centers, where it facilitates movement and performing tasks without constant supervision.

The wheelchair is equipped with two odometry systems: encoder sensors on each wheel for mechanical odometry, and a stereo camera whose images are processed by a ROS algorithm for visual odometry. Additionally, a Velodyne laser sensor based on LIDAR technology is used as a reference for the actual trajectory. The tests were conducted at the Faculty of Physics and Mathematics.

In summary, the aim of this project is to compare and evaluate the performance of visual and mechanical odometry in an autonomous wheelchair, with the goal of improving the quality of life for people with reduced mobility.

Keywords: ROS, Rviz, Visual Odometry, Mechanical Odometry, Viso2, RTAB-Map, ORB-SLAM, SLAM, Features Extraction.

ÍNDICE

Capítulo 1. Antecedentes y situación actual 9
1.1 Introducción 9
1.2 ¿Qué es la odometría? 9
1.3 Aplicaciones de la odometría 10
1.4 Técnicas de odometría 11
1.5 Motivación 12
1.6 Objetivos 14
Capítulo 2. Fundamentos teóricos 16
2.1 Odometría mecánica 16
2.1.1 Definición de la odometría mecánica 16
2.1.2 Cinemática diferencial de la silla de ruedas autónoma 17
2.2 Odometría visual 20
2.2.1 Introducción a la odometría visual 20
2.2.2 Retos de la odometría visual 22
2.2.3 Cámara monocular 23
2.2.4 Cámara estéreo 23
2.2.5 Método basado en características 24
2.2.5.1 Algoritmos de extracción de características 26
2.2.6 Método directo 28
2.2.7 Método híbrido 29
2.2.8 SLAM 29
2.3 Odometría laser 32
Capítulo 3. Material y Herramientas 33
3.1 Hardware 33
3.1.1 Sensores *encoder*. 33
3.1.2 Cámara Estéreo 34
3.1.3 Sensor Velodyne 34
3.2 Software 35
3.2.1 ROS 35
3.2.2 Rviz 35
Capítulo 4. Implementación 37
4.1 Método de ejecución 37

4.2 Implementación de la Odometría Mecánica 39
4.3 Implementación de la Odometría Visual 41
4.3.1 Viso2 41
4.3.2 RTAB-Map 42
4.3.3 ORB-SLAM 46
4.4 Implementación del A-LOAM 49
Capítulo 5. Resultados 51
5.1 Resultados de la Odometría mecánica 52
5.2 Resultados del Viso2 54
5.3 Resultados del RTAB-Map 55
5.4 Resultados del ORB-SLAM 57
5.5 Simulación conjunta 59
5.6 Coste computacional 61
Capítulo 6. Conclusiones y líneas futuras 62
6.1 Conclusiones 62
6.2 Líneas futuras 63
6.3 Conclusions 63
Bibliografía 65
ANEXO I 68
1. Odometría Mecánica 68
2. Odometría Viso2 71
3. Odometría RTAB-Map 72
4. Odometria ORB-SLAM 75
5. Odometría A-LOAM 78
6. Simulación Conjunta 79

ÍNDICE DE FIGURAS

Figura 1.1: Fotografía de la silla de ruedas autónoma……15
Figura 1.2: Fotografía de la silla de ruedas autónoma……15
Figura 2.1: Esquema de funcionamiento de un sensor *encoder*……..*17*
Figura 2.2: Esquema cinemático de la silla de ruedas autónoma……..18
Figura 2.3: Esquema del CIR aplicado a la silla……20
Figura 2.4: Concepto artístico de Opportunity en la superficie de Marte……22
Figura 2.5: Percepción de la profundidad de una cámara estéro……...25
Figura 2.6: Esquema de funcionamiento para extracción de características……26
Figura 2.7: Esquema línea epipolar……..……26
Figura 2.8: Funcionamiento del cierre de bucles……..31
Figura 2.9: Esquema de funcionamiento del SLAM……32
Figura 2.10: Esquema de relación entre VO, SLAM y SfM……..32
Figura 3.1: Sensores encoder instalados en la silla de ruedas……...34
Figura 3.2: Cámara PS4 eye……35
Figura 3.3: Funcionamiento de la cámara PS4 eye……..35
Figura 3.4: Sensor Velodyne……...35
Figura 3.5: Interfaz del Rviz……37
Figura 4.1: Ejemplo de simulación de la odometría mecánica en Rviz……..41
Figura 4.2: Ejemplo de simulación del algoritmo Viso2 en Rviz……..43
Figura 4.3: Ejemplo de simulación del algoritmo RTAB-Map en Rviz……45
Figura 4.4: Mapa 3D creado por el RTAB-Map en Rviz……45
Figura 4.5: Mapa creado por el RTAB-Map en Rviz……45
Figura 4.6: Puntos clave detectados en una imagen por el RTAB-Map……46
Figura 4.7: Puntos clave detectados en una imagen por el RTAB-Map……46
Figura 4.8: Mapa 3D creado por el RTAB-Map en Rtabmaprviz……46
Figura 4.9: Mapa 3D creado por el RTAB-Map en Rtabmaprviz……46
Figura 4.10: Ejemplo de simulación del algoritmo ORB-SLAM en Rviz……49
Figura 4.11: Puntos clave detectados en una imagen por el ORB-SLAM……49
Figura 4.12: Puntos clave detectados en una imagen por el ORB-SLAM……49
Figura 4.13: Mapa 3D creado por el ORB-SLAM en Rviz……49

Figura 4.14: Mapa 3D creado por el ORB-SLAM en Rviz……50
Figura 4.15: Ejemplo de simulación del algoritmo A-LOAM en Rviz……51
Figura 4.16: Mapa 3D creado por el A-LOAM en Rviz……51
Figura 4.17: Mapa 3D creado por el A-LOAM en Rviz……51
Figura 5.1: Esquema del recorrido 1……52
Figura 5.2: Esquema del recorrido 2……53
Figura 5.3: Resultados de la odometría mecánica en el recorrido 1……53
Figura 5.4: Resultados de la odometría mecánica en el recorrido 2……54
Figura 5.5: Resultados del algoritmo Viso2 en el recorrido 1……55
Figura 5.6: Resultados del algoritmo Viso2 en el recorrido 2……56
Figura 5.7: Resultados del algoritmo RTAB-Map en el recorrido 1……57
Figura 5.8: Resultados del algoritmo RTAB-Map en el recorrido 2……57
Figura 5.9: Resultados del algoritmo ORB-SLAM en el recorrido 1……58
Figura 5.10: Resultados del algoritmo ORB-SLAM en el recorrido 2……59
Figura 5.11: Resultados conjuntos del recorrido 1……60
Figura 5.12: Resultados conjuntos del recorrido 2……60
Figura 5.13: Gráfica del error absoluto cometido por cada algoritmo en el eje X.61
Figura 5.14: Gráfica del error absoluto cometido por cada algoritmo en el eje Y.61

ÍNDICE DE TABLAS

Tabla 5.1: Errores absolutos cometidos por cada algoritmo en cada recorrido.61
Tabla 5.2: Coste computacional de cada algoritmo en cada recorrido……62

Capítulo 1.

Antecedentes y situación actual

1.1 Introducción

La robótica es una disciplina de la ciencia que ha evolucionado de forma exponencial en las últimas décadas. Desde los primeros intentos de crear autómatas mecánicos en la antigua Grecia, el ser humano ha intentado optimizar las tareas más arduas y repetitivas de su día a día. Sin embargo, no comenzó a obtener los primeros resultados hasta la Revolución Industrial, donde los primeros robots mecánicos, controlados de forma manual, agilizaron y facilitaron el proceso productivo en algunas fábricas. Con el desarrollo de la electrónica a partir de la década de 1950 y la aparición de las primeras computadoras, se crearon los primeros robots controlados por estas últimas. El avance fue tal, que su uso pasó de simplemente agilizar y mejorar el proceso productivo repetitivo de las fábricas, a usos mucho más especiales como la exploración espacial.

Las posibilidades de aplicación de la robótica comenzaron a despertar un gran interés en la comunidad científica e investigadora, lo que ha propiciado su gran desarrollo en las últimas décadas en infinidad de campos, como la medicina, transporte o agricultura. Nuevas tecnologías como la inteligencia artificial, el aprendizaje automático o la visión por ordenador han ampliado aún más el potencial de los robots. Gracias a estos nuevos avances, es posible la construcción de nuevos robots autónomos provistos de mayor libertad a la hora de desplazarse y de realizar tareas. Para que esto sea posible, surge la necesidad de crear un método capaz de controlar la navegación del robot, así como de controlar sus movimientos de forma precisa, y en tiempo real, en el espacio. Es en este instante cuando la odometría adquiere un papel importantísimo en el desarrollo de robots autónomos.

1.2 ¿Qué es la odometría?

El término odometría proviene de las palabras griegas “odos”, cuyo significado es camino, y “metron”, cuyo significado es medida [4]. Por lo tanto, la odometría consiste en la técnica que estudia la estimación en tiempo real de la posición y orientación relativa de un robot, vehículo autónomo o incluso de un ser humano, en un entorno. Se trata de una técnica inexacta, por lo que no determina dichos parámetros de forma inequívoca, sino que se obtiene una estimación bastante precisa de estos. Esto es debido a que toda la información

usada para calcular la posición de un robot móvil proviene de sensores que no tienen una precisión absoluta y que, por lo tanto, no están exentos de cometer pequeños errores. A partir de todos los datos recabados por estos sensores, se aplica un modelo matemático capaz de interpretar los datos y estimar la trayectoria del robot.

De forma general en el campo de la robótica, la odometría es utilizada para estimar la posición relativa a la localización inicial en la que el robot comenzó el desplazamiento. Esta técnica permite a los dispositivos autónomos conocer su posición y orientación en el espacio, habilitándoles para realizar movimientos más precisos y eficientes. Además, otorga al robot móvil la capacidad de desplazarse de manera efectiva y segura a través de entornos desconocidos o dinámicos.

La odometría es una técnica más económica y eficiente que otras técnicas de localización, como la tecnología GPS, lo que la hace tener una gran utilidad para su aplicación en entornos urbanos con alta densidad de población, donde la señal GPS es más imprecisa. Por otro lado, es capaz de aportar información más detallada sobre el entorno que rodea al robot, así como su desplazamiento y orientación.

La odometría mejora el rendimiento y funcionalidad de los robots, y puede dar cierta autonomía y libertad de movimiento. Para lograr comprender su funcionamiento y aplicaciones, es necesario definir el concepto de un robot autónomo. Un robot autónomo es aquel capaz de realizar tareas y tomar decisiones sobre estas sin intervención humana directa, es decir, son robots que poseen un alto grado de autogestión lo que les permite operar por si solos y de manera independiente a la supervisión humana. Estas cualidades les aportan muchas ventajas respecto a los robots tradicionales, convirtiéndolos en soluciones ideales para realizar tareas en las que se requiere capacidad de adaptación autónoma.

1.3 Aplicaciones de la odometría

La gran versatilidad y eficiencia de la odometría la convierten en una técnica muy popular entre las grandes empresas tecnológicas. Sus aplicaciones abarcan diversos campos [5], los más destacables e importantes son:

1. La fabricación y diseño de vehículos autónomos, ya que la odometría es esencial para la navegación y evasión de obstáculos.

2. La robótica industrial, en la que los robots deben realizar tareas de ensamblaje y fabricación de forma precisa.

3. Robótica móvil, en la que los robots son utilizados para explorar y mapear entornos desconocidos, como en operaciones de búsqueda y rescate o inspecciones de estructuras. De hecho, la empresa Boston Dynamics ya planea implementar su robot cuadrúpedo “spot” para la supervisión de plataformas petrolíferas. Por otro lado, para los drones la odometría es esencial para el control de vuelo y navegación. Además, permite a su vez realizar un reconocimiento y mapeo del terreno que sobrevuelan.

4. Realidad aumentada, la odometría se usa para seguir el movimiento y posición del usuario, permitiendo así una experiencia más inmersiva.

En resumen, tiene una amplia gama de aplicaciones, y dada la tendencia de la industria, sus aplicaciones podrían expandirse a muchos más campos en los próximos años.

1.4 Técnicas de odometría

El inmenso potencial de la odometría ha despertado mucho interés en el sector de la robótica, que ha generado una gran inversión en la investigación de nuevas metodologías que sean más precisas y eficientes. Debido a estos factores, a lo largo de las últimas décadas han aparecido diferentes tipos de odometría, cada una basada en diferentes técnicas para determinar la posición y orientación de un robot en el espacio. Las más utilizadas y conocidas son:

1. Odometría de ruedas, basada en el movimiento de las ruedas mediante el uso de sensores “encoder”. Es la técnica más simple disponible para la estimación de la posición, sin embargo, es muy vulnerable a errores debidos al derrape de las ruedas, errores que, además, se acumulan a lo largo de la trayectoria y provocan que la estimación de la posición y orientación respecto al punto inicial de la trayectoria diverja considerablemente de la realidad.

2. Odometría inercial (INS), utiliza sensores como acelerómetros o giroscopios para medir la aceleración y velocidad angular del robot y estimar así su posición y orientación. Al igual que la odometría de ruedas, es vulnerable a ciertos errores. Por otro lado, los sensores inerciales de alta precisión son muy caros e inviables para el uso comercial.

3. Odometría Visual, es una técnica que obtiene información del entorno a través de una o varias cámaras. Es capaz de estimar el movimiento de la cámara relativo al entorno, mediante la variación de las propiedades cómo el color, la textura, las formas, etc. Su eficacia depende en gran medida de la buena iluminación del entorno y de que este sea estático.

4. Odometría Laser, basada en la tecnología LIDAR, usa un escáner laser para determinar la distancia de los objetos que lo rodean y construir así, un mapa 3D del entorno. Es una tecnología muy eficiente y precisa, sin embargo, puede cometer errores si existen objetos reflectantes en el entorno, como espejos o superficies metálicas. Además, la falta de uniformidad en algunas superficies también puede afectar a su precisión. No obstante, el mayor reto que presenta esta tecnología es la gran cantidad de datos que genera, ya que requiere de un gran coste computacional para que puedan ser interpretados correctamente.

5. Odometría GPS y GNSS, “global position system” y “global navigation satellite system”. Es una tecnología un tanto obsoleta para este tipo de aplicaciones. Esta técnica tan conocida mundialmente no comete errores acumulativos, sin embargo, solo es efectiva en sitios donde el cielo está despejado. Por otro lado, la tecnología GPS de uso comercial comete errores de unidades de metro, lo que la hace inviable para aplicaciones en las que se requiere una precisión de apenas centímetros. Además, el uso de esta tecnología es inviable en espacios confinados.

Como se ha comentado, no existe ninguna metodología de odometría que sea infalible. Debido a este hecho, la solución más común es utilizar varias técnicas de odometría de forma simultánea. Este proyecto se centrará en la comparación y uso de dos técnicas en concreto, la odometría basada en ruedas y la odometría visual, siendo esta última sobre la cual se hará un estudio en mayor profundidad.

1.5 Motivación

Este trabajo de fin de grado se centrará en el estudio y comparación de resultados obtenidos a través de dos metodologías de odometría diferentes. De todas las técnicas de odometría existentes para obtener la trayectoria de un robot

móvil, este trabajo se centrará en el estudio de la odometría visual. Esta será comparada con la odometría mecánica.

Para poder realizar esta comparación, se ha hecho uso del programa ROS, a través del cual se ha obtenido la representación de las trayectorias obtenidas en cada método.

El objeto de estudio es una silla de ruedas autónoma, sobre la que se ha implementado el *hardware* necesario para cada tipo de odometría [1] [2]. La motivación de construir y diseñar una silla de ruedas autónoma surge de la necesidad de mejorar la calidad de vida de personas con movilidad reducida. La independencia de estas personas, asistidas con frecuencia por un cuidador, depende de las sillas de ruedas, las herramientas con las que enfrentan los desafíos en su vida diaria.

La silla de ruedas autónoma tiene como objetivo abordar estos desafíos cotidianos en la vida de estas personas, dándoles la posibilidad de poder desplazarse de manera autónoma sin necesidad de ningún asistente. Proporciona una solución para las personas que no tienen suficiente fuerza física para impulsar manualmente una silla de ruedas o que no disponen de habilidades locomotrices necesarias para controlar una silla de ruedas eléctrica. Los sistemas de odometría de la silla de ruedas le permitirían programar rutas predefinidas, evitar obstáculos, adaptarse a entorno dinámicos y diferentes tipos de terreno. Estás características permiten a los usuarios tener mayor independencia y libertad para realizar actividades y recorrer su entorno de forma eficiente y segura. Además, la silla de ruedas autónoma puede tener también aplicaciones en hospitales, centros de rehabilitación y entornos médicos en general, donde puede ayudar a los pacientes a desplazarse y realizar ejercicios de rehabilitación sin necesidad de supervisión constante.

En resumen, el objetivo principal de la silla de ruedas autónoma es facilitar y mejorar la calidad de vida de las personas con movilidad reducida.

La silla cuenta con dos sistemas de odometría para calcular su posición y orientación:

1. Sensores *encoder* en cada rueda. Estos sensores aportan la información necesaria de la rotación de cada rueda para que sea posible calcular la trayectoria de la silla. Este será el método de implementación de la odometría mecánica.

2. Cámara estéreo, cuyas imágenes se envían a un algoritmo en ROS capaz de calcular una trayectoria a través de la variación de las imágenes obtenidas. Este será el método de implementación de la odometría visual en la silla.

Además, la silla también dispone de un sensor laser Velodyne basado en la tecnología LIDAR. La trayectoria obtenida a través de los datos de este sensor es muy fiable y precisa, por lo tanto, será usada como referencia de la trayectoria real, con el objetivo de comparar la eficacia de las técnicas anteriormente descritas.

En estas dos imágenes, figuras 1.1 y 1.2, se puede apreciar la silla de ruedas que se ha utilizado para realizar el estudio de las ododmetrías. Los recorridos realizados por la silla han sido todos realizados dentro de la Facultad de Física y Matemáticas.

Figura 1. 1

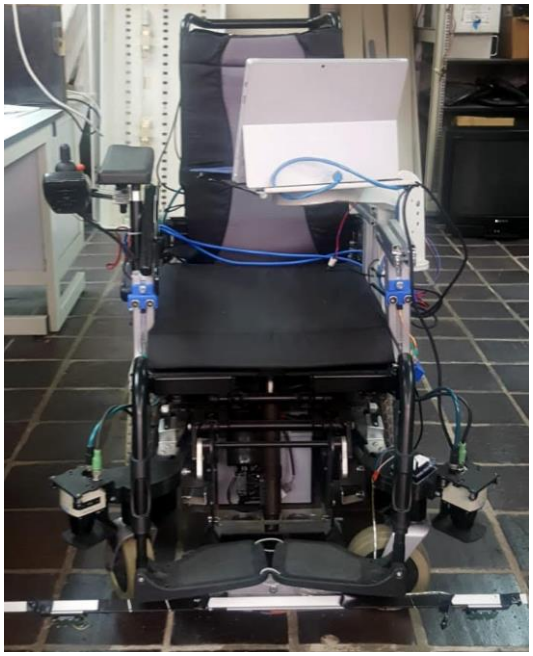

Figura 1. 2

1.6 Objetivos

El objetivo principal del trabajo consiste en hacer un estudio de la eficiencia y eficacia de la odometría visual, mediante la comparación de los resultados de las trayectorias obtenidas a partir de los diferentes tipos de odometría implementados en la silla de ruedas. Para alcanzar dicho propósito, se grabarán varios recorridos de la silla de ruedas dentro de la Facultad de Física y Matemáticas de la ULL para su posterior comparación. Así mismo, los datos

obtenidos por cada sistema de odometría serán simulados y representados a través del programa de software ROS. Las trayectorias se visualizarán mediante una herramienta de visualización llamada RViz.

Para poder alcanzar el objetivo del proyecto de manera satisfactoria, se plantean las siguientes metas y pasos a seguir:

1. Realizar el calibrado de los sensores *encoder*, midiendo de forma precisa el radio de las ruedas, así como de la posición y orientación de la cámara estéreo y del sensor laser respecto al centro de la silla. Estos valores serán de crucial importancia para que los resultados sean lo más exactos posibles y evitar errores.

2. Grabar varios recorridos de la silla de ruedas y guardar los datos obtenidos por cada técnica de odometría para su posterior utilización.

3. Implementar mediante el programa de software ROS los algoritmos de odometría que se deseen comprobar. Usando estos algoritmos en ros y ejecutando los datos obtenidos en los recorridos, se podrán visualizar los resultados a través de RViz.

4. Representar las trayectorias obtenidas a través de cada técnica de odometría en el mismo RViz de manera simultánea.

5. Evaluación de los resultados obtenidos en este último paso, teniendo en cuenta las circunstancias de cada recorrido grabado. Se tomará como referencia la odometría obtenida mediante el sensor laser, ya que es la técnica más fiable y precisa que se dispone.

Finalmente, se procederá a realizar un análisis crítico del trabajo en su totalidad para poder extraer conclusiones objetivas. Estas valoraciones finales recogerán los aspectos positivos y los inconvenientes que se puedan presentar, así como la viabilidad de la propuesta y los logros obtenidos.

Capítulo 2

Fundamentos teóricos

Para que sea posible comprender los experimentos y pruebas que se realizan en este trabajo, es necesario hacer un inciso en los fundamentos teóricos sobre los que se basan las técnicas de odometría aplicadas en la silla de ruedas. Las odometrías mecánica y laser se explicarán brevemente y de forma general, debido a que la edometría visual es el objeto de estudio principal y a la que se le dedicará mayor énfasis y relevancia. Por lo tanto, en este capítulo se explicará de forma detallada los algoritmos utilizados para calcular la trayectoria de la silla de ruedas aplicando las diferentes técnicas de odometría. Así mismo, se explicarán con un mayor detalle los algoritmos de extracción de características de las imágenes, obtenidas por la cámara estéreo, que contienen toda la información necesaria para poder hacer una estimación de la trayectoria de la silla de ruedas.

2.1 Odometría mecánica

2.1.1 Definición de la odometría mecánica

La técnica de estimación de posición y orientación de un robot móvil más simple es la odometría mecánica basada en las ruedas. Debido a su sencillez y bajo coste, es la metodología mayormente utilizada en el sector de la robótica móvil en la actualidad. Este sistema obtiene información del movimiento de las ruedas del robot a través de sensores mecánicos, capaces de medir la revolución de cada eje. Los sensores implementados en nuestra silla son sensores *encoder*, estos dispositivos tienen la capacidad de detectar cuantas vueltas o revoluciones se han producido en un eje, en un intervalo de tiempo dado.

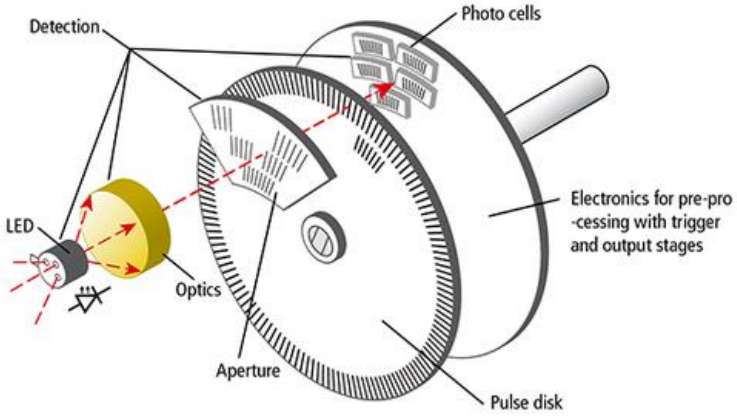

Figura 2.1

2.1.2 Cinemática diferencial de la silla de ruedas autónoma

La cinemática diferencial de un robot móvil es la descripción analítica del movimiento a partir de la disposición de las ruedas y motores de este. Para nuestro caso concreto, se estudiará la configuración de la silla autónoma. Esta, cuenta con dos ruedas motrices unidas por un eje, cada una de ellas lleva instalado un motor eléctrico propio que las hace girar. Este tipo de configuración se conoce como *cinemática diferencial* [3].

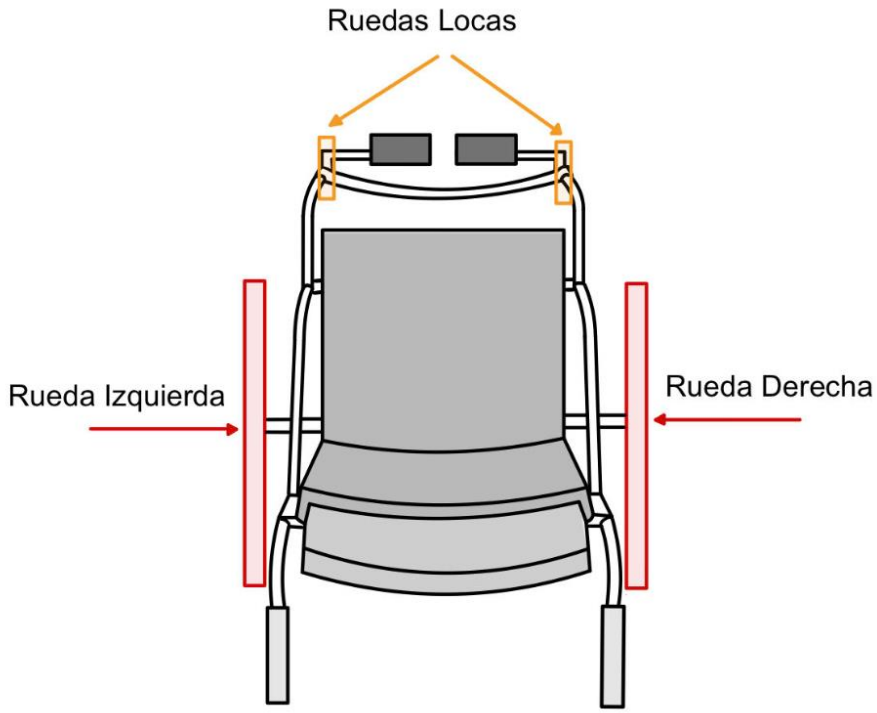

Figura 2.1

En este esquema, figura 2.2, se puede apreciar cómo hay dos ruedas motrices unidas por un eje (ruedas de color rojo) y dos ruedas locas (ruedas de color amarillo). Estas últimas son ruedas pasivas que solo aportan estabilidad a la silla. Las ruedas motrices son las encargadas de controlar el movimiento de la silla, ajustando la velocidad de giro de cada rueda nos permite hacer girar, avanzar o retroceder. Ya que: si las dos ruedas rotan a la misma velocidad en el mismo sentido, la silla avanzará o retrocederá dependiendo del sentido de giro; si la rueda derecha gira más rápido que la izquierda, la silla hará un giro a la izquierda y viceversa. Los sensores *encoder* instalados en cada rueda tienen una resolución concreta que es medida en unidades de pulsos/revolución. Cada pulso se interpreta como la medida discreta de un ángulo, así pues, si nuestro *encoder* tiene una precisión de 360 pulsos/revolución, cada pulso registrado por el *encoder* corresponde a la variación de 1 grado de la rueda.

Una vez explicado el funcionamiento de la silla de ruedas, el objetivo es estimar la posición (x, y, θ) en el plano horizontal, utilizando los datos recogidos por los *encoders*. Lo primero que debemos calcular es cuánto se desplaza una rueda linealmente por pulso registrado en el *encoder* de dicha rueda. A partir de las ecuaciones de la cinemática podemos transformar el desplazamiento angular en desplazamiento lineal. Por lo tanto, se obtiene la siguiente ecuación:

$$C_m = 2\pi R / n C_c$$

Ecuación 2.1

Donde: R = Radio de la rueda
n = Reducción eje/rueda
C_c = Resolución del encoder
C_m = Factor de conversión pulso/desplazamiento

La reducción es una característica muy común en los sistemas rueda/motor y se define como la relación entre las vueltas que da el eje del motor por cada vuelta completa de la rueda.

Conociendo el valor de C_m es posible obtener de manera sencilla el desplazamiento lineal de cada rueda en un tiempo dado. Considerando que los *encoders* leen los datos en intervalos de tiempo regulares, el desplazamiento lineal de cada rueda viene dado por las siguientes ecuaciones:

$$\Delta S_{L.i} = C_m N_{L.i}$$

$$\Delta S_{R.i} = C_m N_{R.i}$$

Ecuación 2.2

Donde: *i* = tiempo de muestreo
$N_{L.i}$ = Número de pulsos leídos por el *encoder* de la rueda izquierda
$N_{R.i}$ = Número de pulsos leídos por el *encoder* de la rueda derecha
$\Delta S_{R.i}$ = Desplazamiento lineal rueda izquierda
$\Delta S_{R.i}$ = Desplazamiento lineal rueda derecha

Sin embargo, no es el desplazamiento de las ruedas lo que interesa, sino el desplazamiento del centro de la silla de ruedas. Por lo tanto, haciendo uso del CIR (centro instantáneo de rotación) y de los valores obtenidos en las ecuaciones anteriores, es posible calcular el desplazamiento lineal del centro de la silla ($\Delta S_{c.i}$)

y la variación de la orientación de este ($\Delta\theta_{c.i}$). Además, es necesario conocer la distancia entre las ruedas (L).

$$\Delta S_{c.i} = (\Delta S_{L.i} + \Delta S_{R.i})/2$$

$$\Delta\theta_{c.i} = (\Delta S_{L.i} - \Delta S_{R.i})/L$$

Ecuación 2.3

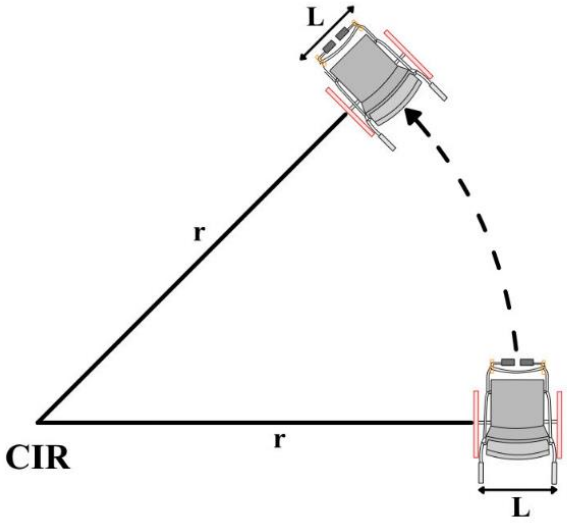

Figura 2.2

Las ecuaciones obtenidas expresan los desplazamientos en cada intervalo de tiempo. Llegados a este punto, solo queda calcular la posición (x, y, θ) del centro de la silla para cada instante *i*. Para lograrlo, es necesario conocer las coordenadas de esta en el instante anterior *i – 1.* Teniendo esta información se obtienen las siguientes ecuaciones que describen la posición y orientación en el plano del centro de la silla:

$$\theta_i = \theta_{i-1} + \Delta\theta_i$$

$$x_i = x_{i-1} + \Delta S_{c,i} cos\theta_i$$

$$y_i = y_{i-1} + \Delta S_{c,i} sen\theta_i$$

Ecuación 2.4

Este proceso se hace de forma iterativa en el tiempo y se considera que las coordenadas de referencia (x_0, y_0, θ_0) es el punto donde comienza el

desplazamiento de la silla de ruedas. Este punto de referencia es esencial debido a que la odometría mecánica solo puede estimar la posición relativa al origen. Esta característica es la mayor desventaja que presenta esta técnica, ya que los errores se acumulan a lo largo de la trayectoria, es decir, cuanto más se desplace el robot móvil, más errores se producirán, por lo que se irán acumulando hasta que dejen de ser despreciables, provocando que la trayectoria estimada deje de ser precisa y fiable. Los cálculos que se han demostrado se basan en supuestos ideales que no se cumplen a la perfección, lo que provoca errores en la estimación de la trayectoria. Por ejemplo, las ruedas se consideran rígidas mientras que una rueda real sufre deformaciones, produciendo variaciones en su radio que producen pequeños errores en los cálculos. La resolución del *encoder* también afecta a la precisión de los cálculos, a mayor resolución, mejores serán los resultados. Estos errores son considerados sistemáticos, no obstante, existen errores no sistemáticos mucho más difíciles de predecir y controlar, como el provocado por el deslizamiento de las ruedas. Este tipo de errores afectan gravemente los resultados obtenidos al registrar movimientos que el robot no ha hecho en la realidad [4] [7].

En resumen, la odometría mecánica basada en las ruedas es una técnica sencilla y fácil de implementar. Sin embargo, aunque su eficacia pueda ser mejorada usando sensores de mayor calidad y coste, es una metodología que acumula errores conforme aumenta el desplazamiento realizado. Debido a estas desventajas, la odometría mecánica solo es fiable para recorridos cortos de menos de 15 metros [8]. Por tanto, para aprovechar la información obtenida por esta metodología, esta suele ir acompañada de otros sistemas de localización, como la odometría visual o laser.

2.2 Odometría visual

La localización de un robot móvil autónomo como nuestra silla de ruedas tiene como objetivo primordial ser capaz de hacer un seguimiento preciso de su trayectoria y de esquivar posibles obstáculos de su entorno [4]. La odometría visual es un método robusto y ampliamente utilizado en la actualidad para este tipo de aplicaciones.

2.2.1 Introducción a la odometría visual

Se define como odometría visual al proceso a través del cual se obtiene la estimación de la traslación y rotación de un robot relativa a un eje de referencia, a partir de la información extraída de la secuencia de imágenes

capturadas de su entorno. Las imágenes contienen una inmensa cantidad de información significativa, de cuyo apropiado procesamiento se puede obtener suficiente información para estimar la trayectoria del robot móvil [5].

La idea de estimar la trayectoria de un robot a través de la información obtenida por una cámara instalada en este surge en la década de 1980 de la mano de Hans Moravec [9], investigador en robótica en la Universidad de Standford. Esta nueva tecnología fue desarrollada por la NASA con el objetivo de implementarla en la misión a Marte de 2004 “Oportunity” [10]. La misión fue un éxito, demostrando la eficacia de la odometría visual en este campo. En la actualidad, la odometría visual ha evolucionado y mejorado aún más gracias al avance en la tecnología de las cámaras y la capacidad de procesamiento de las computadoras. Las IA también han aportado un nuevo enfoque a esta tecnología, ampliando de forma considerable sus aplicaciones. Hoy en día, esta técnica se utiliza en diversos campos aparte de la exploración espacial, como la medicina, la agricultura o la inspección industrial.

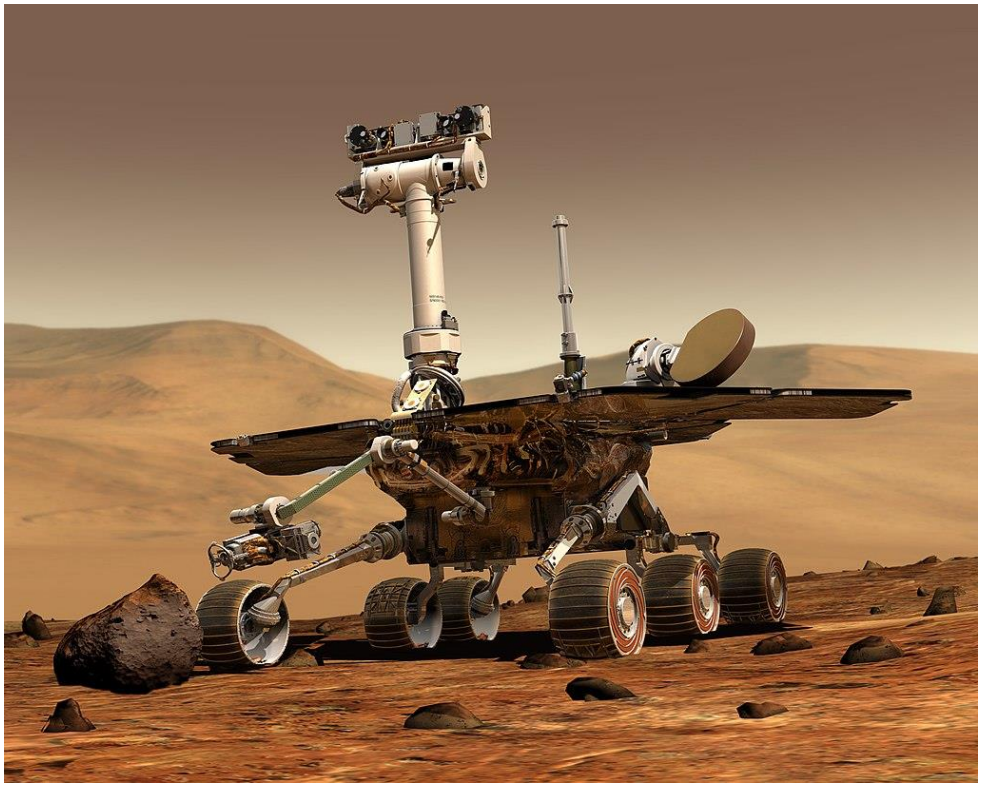

Figura 2.4. Concepto artístico de Opportunity en la superficie de Marte.

2.2.2 Retos de la odometría visual

A diferencia de la odometría mecánica basada en las ruedas, la odometría visual opera estimando de forma incremental la posición y orientación del robot, a partir del examen de los cambios que el movimiento del robot provoca en las imágenes. Asimismo, permite la generación y reconstrucción de mapas 3D de su entorno, denominados SLAM (*Simultanous Localization and Mapping*), que suponen una gran ventaja, ya que mejoran la precisión de la trayectoria estimada mediante el cierre de bucles y aportan más información del entorno al robot.

A pesar de ser una técnica que también acumula errores, esta forma de operar ha demostrado dar resultados más fiables y precisos en la localización de un robot móvil tras largos recorridos, siendo el rango de error relativo de entre 0,1% y 2% [5]. Esto la convierte en una interesante alternativa complementaria a la odometría mecánica, o a otros tipos de sistemas de localización.

No obstante, se requiere buena iluminación del entorno para un funcionamiento eficiente. Además, los fotogramas capturados por la cámara deben asegurar un solapamiento mínimo entre escenas para que la comparación de estas sea efectiva, y poder así calcular el movimiento de manera precisa.

La odometría basada en las cámaras ofrece buenos resultados en interiores y entornos cerrados. No obstante, la localización de un robot se complica en entorno abiertos. Un terreno irregular, sombras, luz solar directa, o los cambios dinámicos del entorno, son los factores más preocupantes y que afectan con mayor gravedad a la eficiencia de esta metodología [11]. Además, el coste computacional también supone un gran reto [7] [12].

Para poder obtener resultados fiables a partir de las imágenes capturadas es necesario procesarlas de manera correcta. Para ello, existen diversos métodos de extracción de características de una imagen, que son vitales para la comparación de fotogramas y el cálculo preciso de la trayectoria descrita por el robot móvil. Por lo tanto, se trata de un proceso crítico en la odometría visual y la elección de un método adecuado según las características del entorno será crucial.

Los métodos utilizados se pueden clasificar tres grandes grupos: Método basado en características (indirect method or feature-based method), método directo (direct method), y método híbrido (semi-direct method) [6] [12] [14]. La mayor parte de los métodos existentes están basados en características, estos tienen la capacidad de detectar puntos característicos y hacer un seguimiento de

ellos fotograma a fotograma. En nuestro caso concreto, se aplicará este método a la silla de ruedas autónoma.

Existen dos líneas de investigación para afrontar estas dificultades, cada una con una configuración de *hardware* diferente. Según el tipo de cámaras implementado, distinguimos la odometría monocular, que usa una única cámara, y la odometría estéreo, que se basa en la información obtenida a través de dos cámaras separadas una distancia determinada la una de la otra. Esta última, es una disposición que permite generar imágenes tridimensionales que permiten calcular la profundidad de los objetos detectados, sobre todo a distancias cortas. Debido a las ventajas de la configuración estéreo respecto a la monocular, esta es la que se ha implementado.

2.2.3 Cámara monocular

Consiste en la utilización de una única cámara montada sobre el robot móvil. A partir de la variación de las características entre fotogramas, como bordes, esquinas o regiones, es posible estimar la trayectoria del robot móvil. Se trata de una metodología más barata y simple que la estéreo [15].

El mayor reto al que se enfrenta esta configuración es ser capaz de generar una estructura del movimiento del robot en tres dimensiones a partir del procesado de información de imágenes en dos dimensiones. Dado que la escala absoluta es desconocida, la distancia entre las dos primeras tomas se considera la unidad. Por otro lado, es necesario conocer el ángulo de la cámara con respecto al plano del suelo y realizar un mapeado del entorno, para obtener resultados fiables [5] [6]. Esta característica genera bastante ruido.

2.2.4 Cámara estéreo

Esta configuración de cámaras está basada en la visión humana y la forma en la que somos capaces de percibir y procesar las características de nuestro entorno. Al igual que los seres humanos tenemos dos ojos, la odometría estéreo usa dos cámaras separadas cierta distancia para capturar el entorno. Esta cámara extra, permite al robot móvil interpretar la profundidad de los objetos detectados aplicando métodos de triangulación [5]. Una vez procesada la información de las cámaras, es posible generar una reconstrucción en tres dimensiones del entorno capturado. Además, las cámaras estereoscópicas deben usar objetivos idénticos y tomar las fotografías de manera simultánea para evitar distorsión indeseada entre ambas imágenes.

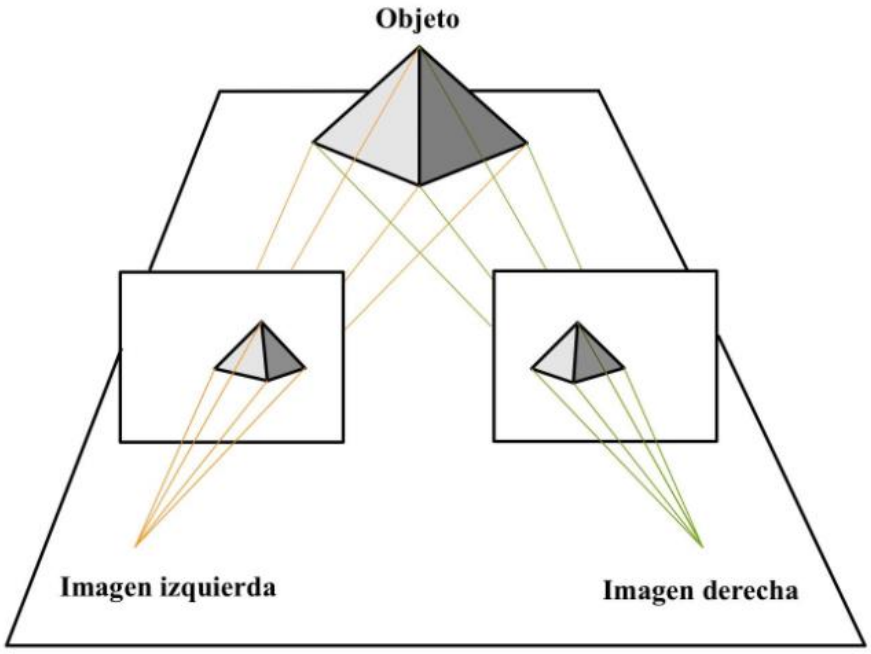

Figura 2.5

Cabe destacar que las ventajas que proporciona la configuración estéreo respecto a la monocular solo son aplicables en distancias cortas. Esto es debido a que, a distancias largas, la separación entre las cámaras estero no es suficientemente grande como para obtener una variación significativa de la perspectiva [5] [6]. Por lo tanto, la odometría estéreo se degrada a odometría monocular en estos casos.

2.2.5 Método basado en características

Como bien describe su propio nombre, este método detecta características, como esquinas, regiones o curvas, en las imágenes obtenidas por la cámara estéreo, estudiando su evolución secuencial en el tiempo. A partir de la correlación y comparación de estas características entre fotogramas, es posible calcular el desplazamiento realizado por el robot entre dichos fotogramas [6] [14] [16]. El proceso se puede resumir en el siguiente esquema, figura 2.6 [17]:

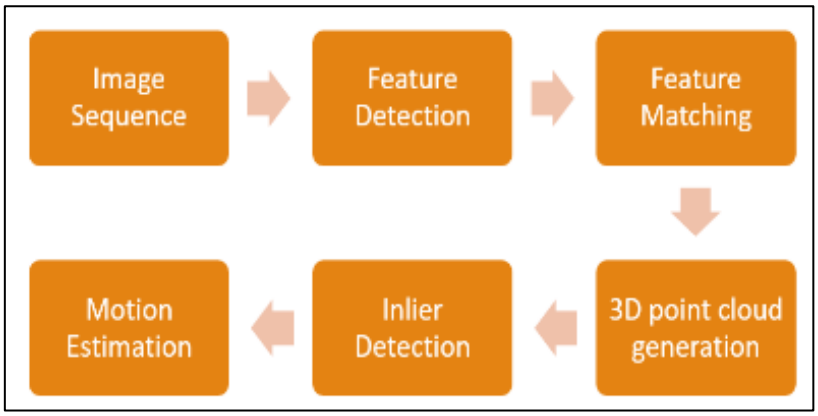

Figura 2.6

En primer lugar, se capturan un par de imágenes estéreo en el instante t y t+1. Estas imágenes son procesadas para compensar la distorsión provocada por las lentes. Además, se realiza una rectificación de manera que las líneas epipolares se vuelvan paralelas a la horizontal [17]. Una línea epipolar es una línea que conecta un punto en el espacio tridimensional de una imagen estéreo con la proyección de ese mismo punto en la otra imagen. Es decir, dado que un punto se proyecta en diferentes posiciones en cada imagen debido a la perspectiva, la línea que conecta ambas proyecciones de denomina epipolar, como se muestra en la figura 2.7. Esta línea se utiliza para reducir el número de posibles características coincidentes, reduciendo el coste computacional y mejorando la eficiencia del algoritmo [5].

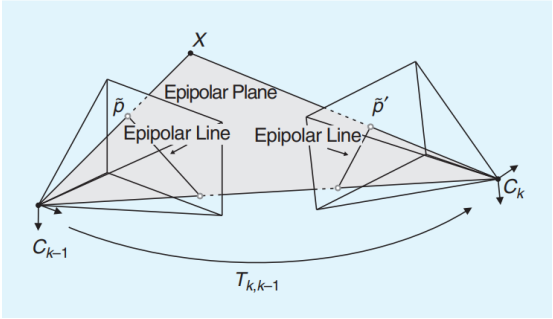

Figura 2.7

En segundo lugar, se detectan las características del par de imágenes obtenidas en el tiempo t y t+1. Existen varios algoritmos de detección de características como SURF, SHIFT, FAST, ORB o BRIEF, que se explicarán con mayor detalle más adelante.

En tercer lugar, a partir de las características obtenidas en el paso anterior, se buscan dichas características en el par de imágenes obtenidas en el instante t+1, que se compararán para obtener una imagen de disparidad en 2D.

En cuarto lugar, se realiza la generación de puntos 3D a partir de la información dada por la imagen de disparidad y las matrices de proyección de la cámara. Por lo tanto, para cada punto característico se obtiene su posición en el espacio tridimensional.

En quinto lugar, se rechazan los valores atípicos que puedan entorpecer y afectar de forma negativa a los cálculos.

Por último, se estima el movimiento de la cámara fotograma a fotograma. Obteniéndose la matriz de rotación y traslación calculada en el punto medio entre las cámaras estéreo. Realizando este cálculo de forma periódica en el tiempo se obtiene la evolución de las posiciones de la cámara a lo largo del tiempo, generándose de esta forma su trayectoria en el espacio.

2.2.5.1 Algoritmos de extracción de características

Existen varios tipos de detección de características en el campo de la odometría visual. La detección y selección correcta de puntos característicos de una imagen es crucial para la correcta estimación de la trayectoria de un robot móvil. Los algoritmos deben ser capaces de comparar características de las imágenes de manera consecutiva sin cambios repentinos, ya que debe existir un solapamiento mínimo entre fotogramas para poder realizar la comparación de forma correcta. Las características a detectar de una imagen pueden ser bordes, regiones, esquinas, etc [6].

En este apartado se describen de forma breve los algoritmos más conocidos y utilizados en la actualidad. Los algoritmos que se implementarán en la silla de ruedas son el FAST, BRIEF, ORB, SIFT y SURF.

- HARRIS: Se trata de un algoritmo ampliamente utilizado para la detección de esquinas y bordes en una imagen. Busca puntos donde se produce una variación significativa de la intensidad; si la variación se produce en direcciones normales, se considera un borde, pero si la variación se produce en todas las direcciones se interpreta como esquina. Las variaciones de intensidad se detectan mediante sumas de diferencias al cuadrado (SSD[4]) en zonas colindantes. Es un algoritmo preciso y robusto, no obstante, puede resultar lento y perdiendo eficiencia por el efecto provocado debido al exceso de ruido.

- FAST (Features from Accelerated Segmented Test): Es un algoritmo cuya principal función es localizar puntos clave en imágenes, que tiene una alta

probabilidad de ser encontrados en los siguientes fotogramas. Su funcionamiento se basa en la comparación de valores de intensidad de un píxel central con sus pixeles vecines en un radio de 16 pixeles. Posteriormente, se establece un umbral a partir del cual se decide si la diferencia de intensidad entre los pixeles es suficiente para ser clasificado como borde o esquina [18].

- BRIEF (Binary Robust Independent Elementary Features): Trabaja a partir de puntos claves ya detectados. El BRIEF genera una cadena binaria que describe la relación entre los píxeles vecinos a un punto de interés. Generalmente, se seleccionan 512 pares de pixeles alrededor de dicho punto de manera aleatoria haciendo uso de una función Gaussiana. Posteriormente se compara la intensidad entre cada par; Si la intensidad del primer píxel del par es mayor que la del segundo, se almacena el valor 0 en el descriptor, en caso contrario se almacena el valor 1. Esta comparación se realiza con todos los pares seleccionados para formar un descriptor de características independiente de la rotación y escala [22].

- SIFT (Scale Invariant Feature Transform): Este algoritmo detecta puntos clave a partir de la comparación de pixeles adyacentes invariantes a la orientación y escala. En primer lugar, se estudia mediante la detección de máximos y mínimos si estos difieren en parámetros como la intensidad, el color o la textura, para detectar puntos de interés. La comparación se realiza analizando la diferencia gaussiana de la imagen a diferentes escalas. Luego, a partir de los histogramas de orientación de cada punto clave detectado anteriormente, se seleccionan los puntos con orientación dominante e independiente de la rotación. La imagen resultante es independiente a la escala, orientación e iluminación, lo que la hace ideal para la comparación entre fotogramas [19].

  En resumen, SIFT es un algoritmo bastante robusto, debido su independencia respecto a la escala, rotación e iluminación de la imagen. Esta independencia permite obtener resultados bastante precisos, aunque es computacionalmente costoso.

- SURF (Speeded up Robust Features): Se trata de una versión más rápida que el SIFT que, además, presenta algunas mejoras. En primer lugar, la velocidad computación es notablemente superior, lo que mejora el rendimiento y eficiencia. En segundo lugar, el SURF es más robusto ante posibles transformaciones de la imagen. Para obtener estas mejoras, este algoritmo hace uso de la matriz Hessiana para detectar puntos de interés,

maximizando así la diferencia gaussiana. Posteriormente se calcula el vector característico para cada punto clave, la dimensión de estos vectores es menor que la usada en el algoritmo SIFT. En definitiva, se trata de un algoritmo más rápido, pero menos preciso que el SIFT [20].

- ORB (Oriented FAST and Rotated BRIEF): Este algoritmo de detección de características es bastante rápido y preciso ya que combina el algoritmo FAST y el algoritmo BRIEF. Por lo tanto, el algoritmo ORB detecta los puntos característicos mediante FAST, este selecciona los puntos clave estudiando los pixeles adyacentes en un radio de 16 pixeles. La intensidad de cada píxel dentro de dicho radio es comparada con la del pixel central. En función del porcentaje de pixeles vecinos con valores de intensidad similar se determina si se trata de un borde o no. Por otro lado, utiliza BRIEF para crear el vector descriptor. Éste toma un conjunto de puntos aleatorios alrededor de los puntos clave y mide la diferencia entre ellos, creando una cadena binaria según un umbral, que se combinan formando el descriptor final [21].

2.2.6 Método directo

Esta técnica usa toda la información disponible de la imagen, y no solo los puntos característicos, como en el método basado en características. Al usar más información, se minimiza el error fotométrico, sin embargo, aumenta el coste computacional, lo que lo convierte en un método más lento.

El método directo se centra en la variación de la intensidad de cada pixel. La estimación de la traslación y rotación del robot móvil se puede realizar usando un algoritmo de flujo óptico (OF). Este tipo de algoritmos compara los valores de intensidad de los píxeles vecinos para calcular el desplazamiento de los patrones de brillo entre fotogramas. Los algoritmos basados en el flujo óptico se pueden clasificar en dos tipos: algoritmos de flujo óptico escaso, que estudian el desplazamiento de ciertos píxeles en concreto; y algoritmos de flujo óptico denso, que estudian el desplazamiento de todos los píxeles de la imagen. Los primeros no evitan la extracción de características, pero son menos vulnerables al ruido que los algoritmos de flujo óptico denso [14].

Cabe destacar también otro algoritmo más moderno, el DSO (“direct sparse odometry”), basado también en el flujo óptico. Este algoritmo utiliza la variación de la intensidad de los píxeles de una imagen. Combina un modelo probabilístico directo que minimiza el error fotométrico, y varios parámetros de la imagen como la geometría o la profundidad. Con una optimización conjunta, es

capaz de estimar la posición y orientación de la cámara en tiempo real en plataformas de escasos recursos como robots o drones. Al evitar el suavizado que se aplicaba en otros métodos directos y, en su lugar, realizar un muestreo uniforme de los pixeles en todas las imágenes, el proceso se optimiza y requiere un menor coste computacional. Además, el DSO no depende de detectores de punto característicos, ya que puede muestrear pixeles de cada región de la imagen que tenga un gradiente de intensidad similar. Estas características lo convierten en un algoritmo ideal para su uso en entornos dinámicos y cambiantes [14].

Por último, el tipo de algoritmo más comúnmente utilizado es el método de coincidencia de plantillas “template matching method”. Esté algoritmo se basa en determinar y seleccionar una plantilla o subimagen en el fotograma actual, una vez hecho, intenta hacer coincidir esta plantilla en el siguiente fotograma. A partir de las variaciones de la plantilla entre fotogramas, es posible estimar el desplazamiento del robot móvil.

2.2.7 Método híbrido

El método híbrido combina características del método directo y del método basado en detección de características. El método directo es capaz de reconstruir entorno en 3D detallado, ya que trabaja con una gran cantidad de información, que los hacen, por otra parte, computacionalmente exigentes. Estos mapas densos en 3D suelen reconstruirse al final del proceso. Por otro lado, el método basado en características solo puede basarse en los puntos de interés detectados, por lo que resultan más imprecisos. Para mejorar la velocidad de ejecución, se pueden utilizar puntos característicos para inicializar el mapeo denso, aunque también existen nuevos algoritmos que trabajan a la inversa, usando primero el método directo para estimar el desplazamiento y los puntos característicos a continuación. Por lo tanto, el método híbrido combina continuamente el método directo escaso y el método basado en características, aprovechando la velocidad del seguimiento de características y de la precisión de la variación de gradientes de intensidad [14].

2.2.8 SLAM

La odometría visual permite no solo estimar la posición y orientación de un robot móvil a lo largo del tiempo, sino que también permite escanear su entorno para construir un modelo en 3 dimensiones de este. A esta técnica se la conoce como SLAM (“Simulatneous Localization and Mapping”). El SLAM

permite construir un mapa detallado del entorno a partir de la información de las mismas imágenes que utiliza la odometría visual [5].

La odometría visual recupera y corrige la trayectoria descrita de forma incremental, posición tras posición, optimizando solamente las últimas n posiciones del recorrido. Por lo tanto, la consistencia y fiabilidad de la estimación de la trayectoria a partir de la odometría visual depende del mapa local. En cambio, el SLAM es capaz de corregir la estimación de la trayectoria basándose en un mapa global, identificando de esta manera el momento en el que el robot móvil vuelve a un lugar ya recorrido, es decir, conocido. Esta característica se conoce como cierre de bucle (*loop closure*) [24]. Cuando la cámara detecta una similitud significativa en el entorno con una zona previamente capturada, y la trayectoria registrada concuerda con dicha hipótesis, esta interpretará que se trata de una zona que ya ha recorrido. Cuando esta situación ocurre, la información obtenida se emplea para reducir el error de la trayectoria y optimizar el mapa 3D. En la figura 2.8 se muestra un esquema del funcionamiento del cierre de bucle. En ella se puede apreciar como en el esquema de la izquierda, en el que no está aplicado el cierre de bucle, el punto B lo interpreta como otra zona del mapa, considerando un pasillo infinito. Sin embargo, en el esquema de la izquierda, el punto B es reconocido como un punto por el cual ya ha transitado, por lo que lo integra como tal en la trayectoria y mapeado [23].

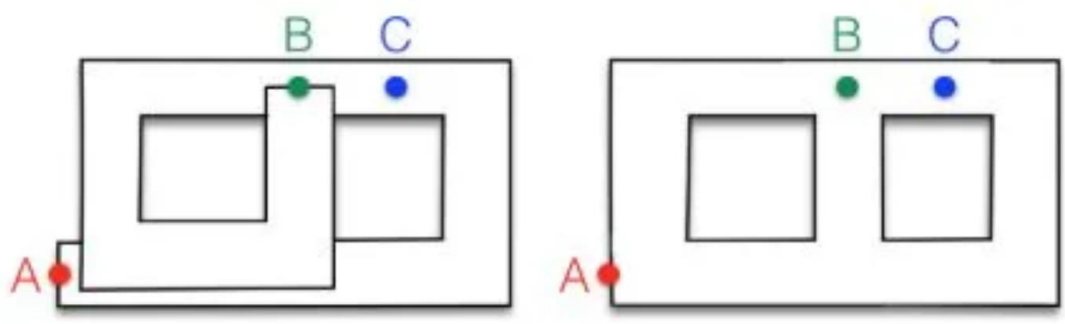

Figura 2.8

La odometría visual es uno de los métodos principales y más usados para construir el *Visual SLAM.* En la figura 2.9 se pueden identificar los tres bloques principales de esta técnica.

1. El *loop closure,* capaz de reconocer entornos previamente capturados, explicado anteriormente [24].

2. La optimización *backend,* encargada de fusionar la información de los sensores y ajustar las estimaciones de la posición, minimizando errores y optimizando el proceso
3. La reconstrucción, utilizado para aplicaciones que requieren de una representación densa y más detallada del entorno. Suele usarse para fines como la evasión de obstáculos, navegación o interacción con el entorno [25].

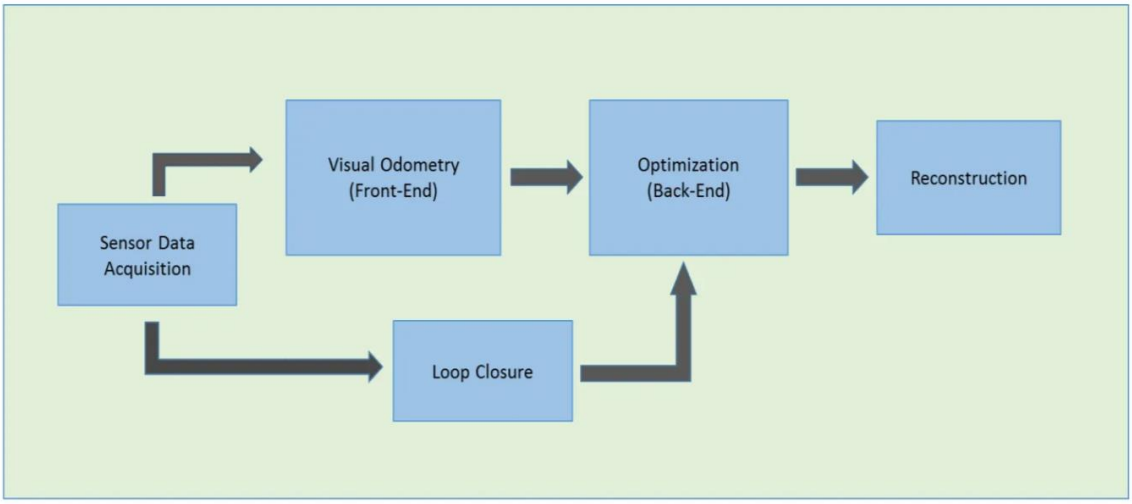

Figura 2.9

En la figura 2.10 es posible visualizar la relación entre la odometría visual, el SLAM visual, y el SfM. El SfM (*Structure from Motion*) es una técnica muy similar al SLAM, pero que utiliza imágenes arbitrarias no secuenciales, tomadas desde diferentes perspectivas, que mejora el detalle y la reconstrucción del entorno en 3D [26].

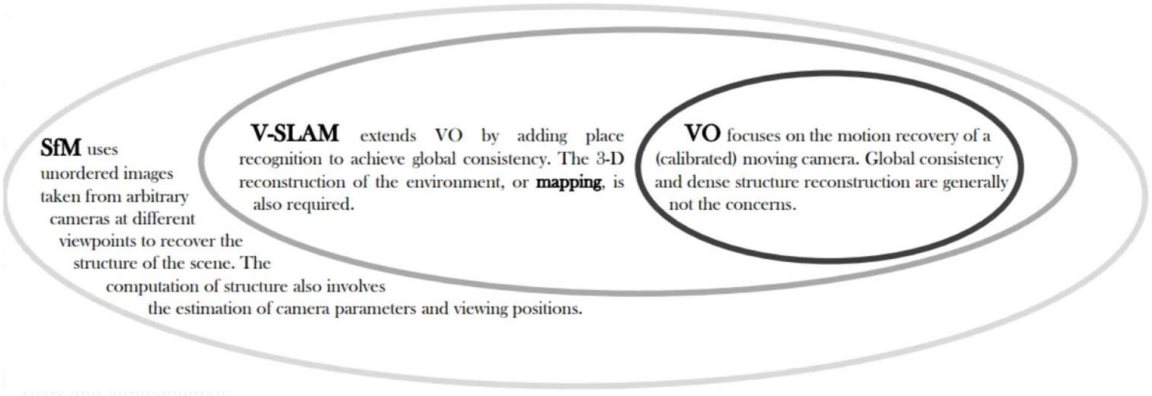

Figura 2.10

2.3 Odometría laser

La odometría láser es una tecnología de localización que consiste en medir la distancia entre el robot móvil y los objetos detectados en su entorno. Según la evolución en el tiempo de los valores de estas distancias es posible calcular la traslación y rotación del robot. El sensor laser emite pulsos de luz en todas las direcciones que rebotan en los objetos, midiendo el tiempo que tarda en volver el haz de luz se calcula la distancia a la que está cada objeto [27]. Debido a su precisión y fiabilidad, se utilizarán las trayectorias obtenidas a través de este método como referencia real o *ground truth,* para ser comparadas posteriormente con el resto de técnicas implementadas en la silla de ruedas.

En resumen, se trata de una técnica que requiere de un gran coste computacional, debido a la enorme cantidad de información del entorno que recopila. Esta desventaja complica su uso en tiempo real. No obstante, los resultados obtenidos a partir de la odometría laser son mucho más precisos y fiables que los obtenidos a partir de otro tipo de odometrias como, la visual o la mecánica. Cabe destacar que el rendimiento de esta tecnología puede verse perjudicado por la presencia de superficies transparentes o reflectantes en el entorno.

Capítulo 3

Material y herramientas

3.1 Hardware

En este apartado se explican de manera más detallada los elementos físicos (*hardware*) de interés implementados en la silla y que son de vital importancia para la obtención de resultados precisos y fiables en los experimentos.

3.1.1 Sensores *encoder*.

Como se explica en el apartado 2.1 del capítulo 2 de este documento, la silla de ruedas cuenta con dos sensores *encoder*, uno en cada rueda motriz, que miden la rotación de cada eje. A partir de esta información, se hace uso de las ecuaciones de la cinemática diferencial para obtener la posición y orientación del centro de la silla a lo largo del tiempo. Los sensores *encoder* montados en la silla tienen una resolución de 32 pulsos, no obstante, al estar instalados antes de la reductora, se obtiene una resolución efectiva de 8.800 pulsos por vuelta de la rueda. Esta gran resolución permite obtener una estimación mucho más precisa al calcular la trayectoria de la silla. La figura 3.1 muestra cómo y dónde están instalados los principales elementos que participan en el cálculo de la odometría mecánica.

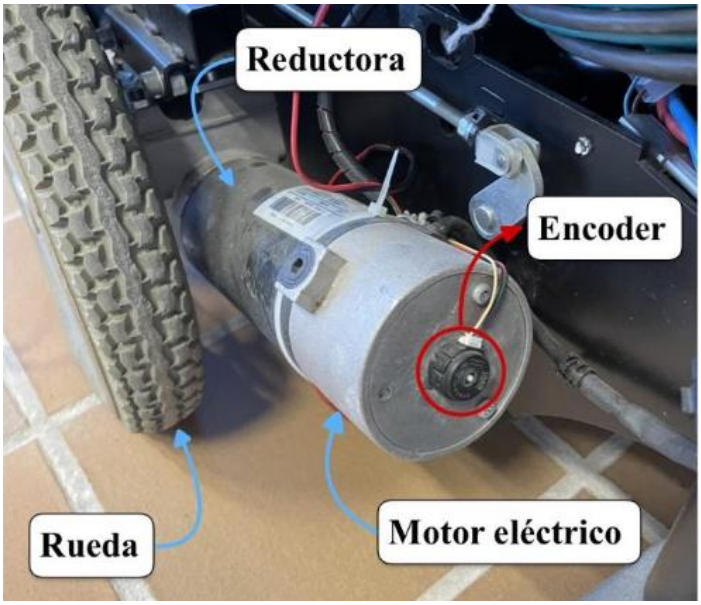

Figura 3.1

3.1.2 Cámara Estéreo

La cámara estéreo instalada en la silla de ruedas es el modelo de sony PS4 eye. Se trata de un modelo que cuenta con dos cámaras de alta resolución y lentes duales, lo que permite capturar imágenes nítidas de alta calidad. En la figura 3.2 se muestra una imagen de la ps4 eye y en la figura 3.3 se puede observar un ejemplo de funcionamiento utilizando el paquete de ROS ps4eye [28].

Figura 3.2

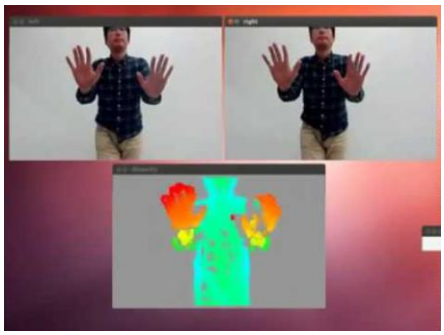

Figura 3.3

3.1.3 Sensor Velodyne

En la silla de ruedas se ha implementado un sensor Velodyne modelo vhdl32 basado en la tecnología LIDAR, la más usada para la detección y evasión de obstáculos, mapeo 3D y estimación de trayectorias. Este sensor es muy preciso y fiable, los resultados de este se usarán como *ground truth* o referencia real de la trayectoria real de la silla [29].

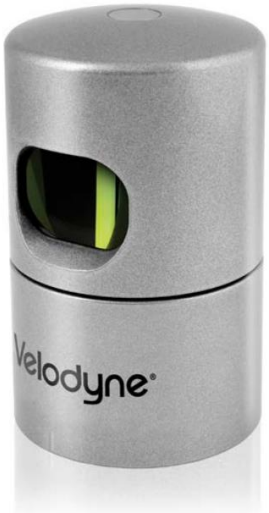

Figura 3.1

3.2 Software

La totalidad de este trabajo se ha realizado sobre el sistema operativo Ubuntu 18.04, instalado a través de una máquina virtual creada con el *software* gratuito Oracle VM VirtualBox. Si bien ROS es un programa compatible con múltiples sistemas operativos, este fue desarrollado inicialmente en Linux, por lo tanto, dado que Ubuntu es un sistema operativo basado en Linux, ROS tiene mejor compatibilidad y soporte en este sistema. Además, la instalación y configuración de ROS en Ubuntu es más sencilla y genera menos problemas.

Las principales herramientas de *software* implementadas son ROS, para la ejecución de las simulaciones y Rviz para la visualización de estas en un entorno 3D.

3.2.1 ROS

ROS (*Robot Operating System*), se define como una colección de *frameworks* de código abierto para el desarrollo de *software* de robots. Un framework es un conjunto de herramientas, librerías y componentes de *software* que proporciona una base y estructura para el desarrollo de aplicaciones. ROS provee de servicios tales como el control de dispositivos a bajo nivel, abstracción de *hardware*, mantenimiento de paquetes y comunicación entre paquetes. Además, proporciona el medio para la comunicación entre diferentes componentes de software, como los nodos y los tópicos. Los nodos son partes ejecutables del código, que se comunican a través de los tópicos que publican [30].

En resumen, ROS es utilizado en una amplia variedad de aplicaciones robóticas, ya que proporciona un entorno robusto y flexible para el desarrollo de *software* en este campo.

3.2.2 Rviz

Rviz es un visualizador 2D/3D de datos inicialmente diseñado para ROS, que permite seleccionar tópicos que están siendo publicados en ROS mostrándolos en un sistema de coordenadas a elección del usuario. Gracias a esta herramienta, es posible visualizar y representar las trayectorias en tiempo real, obtenidas a través de las diferentes técnicas de odometría implementadas en la silla de ruedas [31].

La interfaz del visualizador es bastante sencilla e intuitiva. Ofrece varias opciones interesantes, como elegir el eje de referencia (*frame*) en el que se quiera visualizar la odometría. En nuestro caso, se ha ajustado el código de cada odometría para que publiquen sus tópicos en un *frame* común llamado odom. Para establecer la relación de transformación entre *frames* se escribe el siguiente *script* en el código:

```
<node pkg="tf2_ros" type="static_transform_publisher" name="ps4eye" args="0 0 0 -1.58 0 -1.45 base_link ps4eye_frame" />
```

En este ejemplo, se establece la relación entre el eje base_link y el eje ps4eye_frame, los tres primeros valores indican la traslación y los tres últimos la rotación.

Una vez seleccionado el eje de referencia, Se añaden los tópicos que se desean visualizar mediante el botón “add → add by topic” y se selcciona la opción “Odometry” del tópico que se desea visualizar. Además, es necesario ajustar algunos parámetros como el número de puntos publicados “keep”, que debe ampliarse, o la covarianza, que se debe deseleccionar. Por otro lado, para visualizar las odometrías de los algoritmos RTAB-Map y A-LOAM, se utilizará la opción de “MapGraph” y “Path” respectivamente. En la figura 3.5 se puede ver el entorno Rviz.

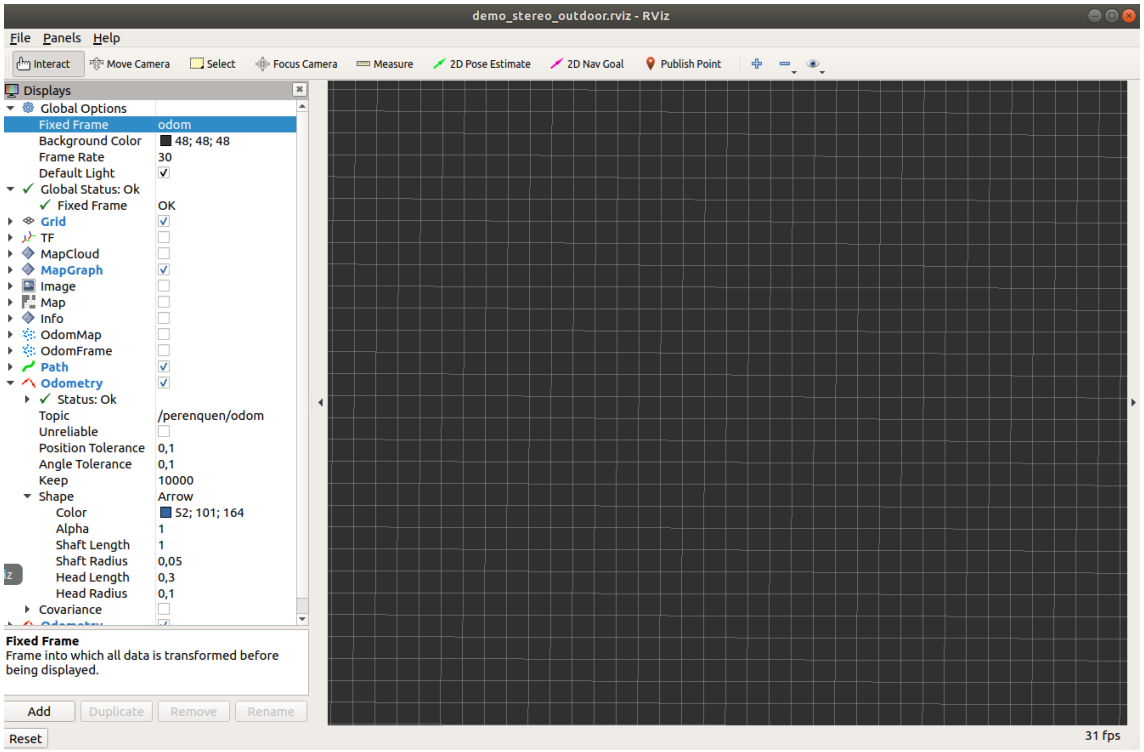

Figura 3.1

Capítulo 4

Implementación

En este capítulo se explica detalladamente el funcionamiento de todos los algoritmos y librerías de ROS implementados, así como los métodos de ejecución de utilizados. Además, a modo de ejemplo, se mostrarán las trayectorias y mapas 3D obtenidas por cada algoritmo, simuladas en Rviz.

Los recorridos se graban como bolsas de datos con extensión ".bag", que han sido capturados por la cámara estéreo PS4EYE. Además de contener información visual, estas bolsas de datos también contienen la información recogida por los sensores *encoder* y el sensor Velodyne.

4.1 Método de ejecución

A continuación, se describe el procedimiento general llevado a cabo para ejecutar los diferentes algoritmos utilizados:

1. Inicialización de ROS y ejecución de archivos: Primero se usa el comando "roscore" desde el terminal. Posteriormente, se abre otro terminal para lanzar el comando que ejecute nuestro algoritmo;

   - Si se trata de un código de Python ".py", el comando que se debe escribir es el siguiente: "rosrun paquete nombre.py"
     Ejemplo: "rosrun viso2_ros prueba2.py" ejecuta el código Python llamado "prueba2.py" que se encuentra guardado en el paquete de ROS "viso2ros".

   - Si se trata de un código escrito en un archivo ".launch", el comando que se debe escribir es el siguiente: "roslaunch paquete nombre.launch"
     Ejemplo: "roslaunch viso2_ros demo_stereo.launch" ejecuta el código llamado "demo_stereo.launch" que se encuentra guardado en el paquete de ROS "viso2ros".

2. Visualización de los resultados: Mediante el comando "rosrun rviz rviz -d" abrimos un nuevo rviz en el que poder visualizar diferentes tópicos. Pulsamos sobre la opción de añadir y seleccionamos el tópico que se

desea visualizar. Los nombres de los tópicos para la odometría de cada uno de los algoritmos usados son los siguientes:

- Odometría mecánica: /perenquen/odom (Odometry)
- Odometría Viso2: /stereo_odometer/odometry (Odometry)
- Odometría RTAB-Map: /rtabmap/mapGraph (MapGraph)
- Odometría ORB-SLAM: /stereo/odom (Odometry)
- Odometría A-LOAM: /aft_mapped_path (Path)

Además, es posible guardar la configuración del rviz para poder ser ejecutado más adelante usando el siguiente comando: “rosrun rviz rviz -d nombre.rviz”, teniendo en cuenta que es necesario situarse en la carpeta en la que está almacenado.

3. Ejecución de los datos grabados: Por último, se ejecuta la bolsa de datos usando el comando “rosbag play --clock nombre.bag”. Donde se publicarán en el tiempo de simulación los tópicos correspondientes a las cámaras estéreo, el Velodyne y las cuentas de los sensores de la odometría mecánica:

    - /clock
    - /cmd_vel
    - /imu9250
    - /left_laser
    - /left_laser/parameter_descriptions
    - /left_laser/parameter_updates
    - /left_laser_filtered
    - /perenquen/left/feedback
    - /perenquen/right/feedback
    - /perenquen/status
    - /right_laser
    - /right_laser/parameter_descriptions
    - /right_laser/parameter_updates
    - /right_laser_filtered
    - /rosout
    - /rosout_agg
    - /stereo/left/camera_info
    - /stereo/left/image_raw/compressed
    - /stereo/right/camera_info
    - /stereo/right/image_raw/compressed
    - /temperature9250
    - /velodyne_points

Cabe mencionar que, durante la implementación de dichos algoritmos debido al uso de la máquina virtual y la capacidad de ejecución del ordenador, ha sido necesario reducir la velocidad de publicación de imágenes y datos de la

bolsa de datos, ya que el portátil no es capaz de procesar tanta información a la vez, provocando errores en la simulación. Esto ocurrió sobre todo al ejecutar los algoritmos de ORB-SLAM y RTAB-Map. Para solucionar el problema, se añade en el comando: “rosbag play --clock -r0.5 nombre.bag”. En este caso se han obtenidos buenos resultados reduciendo la velocidad de publicación a la mitad.

4.2 Implementación de la Odometría Mecánica

La explicación detallada de los fundamentos teóricos de la cinemática diferencial de la silla de ruedas ha sido explicada en el apartado 2.1 del capítulo 2 de este documento. En base a estos fundamentos, la odometría mecánica se ha implementado a través de un código escrito en lenguaje Python, que hace uso de los datos de los *encoders* provenientes de la bolsa de datos para obtener la trayectoria de la silla. Además, es necesario incluir en este código parámetros como la resolución de los *encoders*, el diámetro de las ruedas o la distancia entre los ejes de estas, como se puede apreciar en este fragmento del código. A continuación, se explican algunos aspectos importantes de este:

```
#Resolución encoder izquierdo y derecho.
   ticks_turn_left = rospy.get_param('ticks_turn_left', 8800)
   ticks_turn_right = rospy.get_param('ticks_turn_right', 8800)
#Diametro en metros rueda izquierda y derecha.
   wheel_diameter_left = rospy.get_param('wheel_diameter_left', 0.3227)
   wheel_diameter_right = rospy.get_param('wheel_diameter_right', 0.3220)
#Distancia en metros entre ejes.
   wheel_track = rospy.get_param(‘wheel_track’, 0.60)
```

Teniendo estos datos, es posible calcular el desplazamiento y orientación de la silla relativo a su posición anterior, basándose en las ecuaciones 2.3 y 2.4. Estas son implementadas en el código como se muestra en el siguiente fragmento del mismo:

```
#Almacena el valor anterior del ángulo de orientación.
  yawAnt = odom_.z
#Se calcula las distancias recorridas por cada rueda, utilizando la información del incremento de los encoders (‘incL’ e ‘incR’) y el diámetro de las ruedas.
   incMetrosLeft = (incL/ticks_turn_left)*math.pi*wheel_diameter_left
   incMetrosRight = (incR/ticks_turn_right)*math.pi*wheel_diameter_right
#A partir de las distancias recorridas por cada rueda, se obtiene el desplazamiento del centro de la silla.
   despCentroEje = (incMetrosRight + incMetrosLeft)/2
```

```
#IncYaw calcula el ángulo de orientación basándose en la diferencia entre las distancias recorridas por cada rueda y la distancia entre los ejes de estas.
    incYaw = ((incMetrosRight - incMetrosLeft) / wheel_track)
#Actualiza el ángulo de orientación sumando el cambio 'incYaw' al valor anterior 'yawAnt', la función constrainAngle asegura que el valor no se salga del rango deseado.
    odom_.z = constrainAngle(incYaw + yawAnt)
#Se almacenan las coordenadas anteriores de la silla X e Y en el plano.
    xAnt = odom_.x
    yAnt = odom_.y
#Se almacena la orientación actualizada
    yaw = odom_.z
#Se calculan los incrementos en las coordenadas X e Y basados en el desplazamiento del dentro de la silla.
    incx = despCentroEje*math.cos(yaw)
    incy = despCentroEje*math.sin(yaw)
#Se actualizan las coordenadas X e Y sumando los incrementos anteriores
    odom_.x = xAnt + incx
    odom_.y = yAnt + incy
……..
#Publica la transformación entre los marcos de referencia 'base_link' (marco de la silla) y 'odom' (marco de referencia global)
odom_broadcaster.sendTransform( (odom_.x, odom_.y, 0.),  odom_quat,  current_time, "base_link", "odom")
```

La totalidad del código se encuentra recogida en el Anexo I de este documento. En la figura 4.1 se pueden apreciar los resultados tras seguir los pasos mencionados anteriormente. En la imagen se ve con claridad la trayectoria seguida por la silla de ruedas según la información obtenida por los *encoders.*

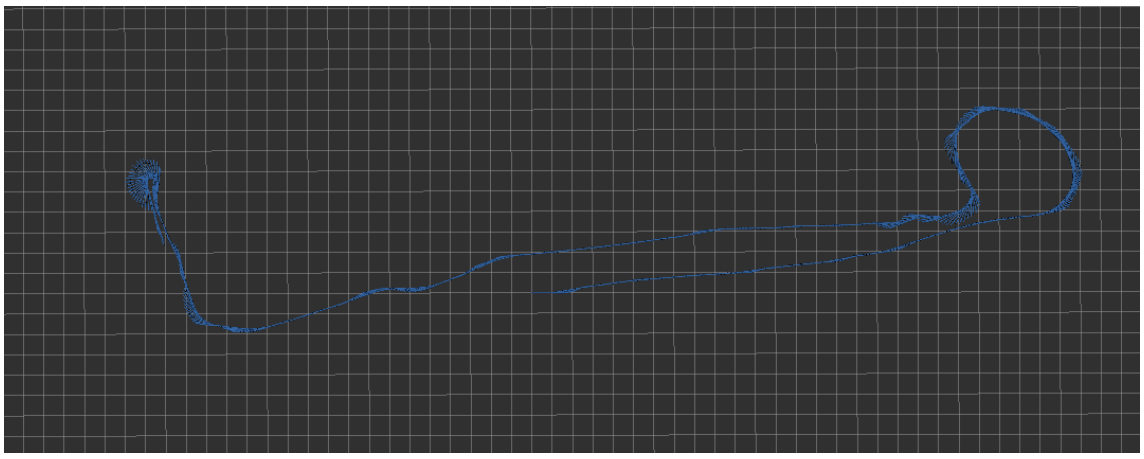

Figura 4.1

4.3 Implementación de la Odometría Visual

En este apartado se explicarán los diferentes algoritmos de odometría visual implementados en la silla de ruedas.

4.3.1 Viso2

Se trata de un algoritmo multiplataforma desarrollado en el lenguaje de programación C++, basado en el método de extracción SIFT. Hace uso de técnicas de procesamiento de imágenes para extraer características de interés que poder rastrear entre fotogramas, estas permiten estimar la trayectoria de un robot relativa a su entorno [32].

La información de entrada que recibe el Viso2 son las características capturadas en 4 imágenes, dos fotogramas consecutivos por cada cámara de la PS4EYE. Este algoritmo hace uso de un filtrado que selecciona los máximos del gradiente de intensidad en regiones de tamaño 5x5, que permite eliminar el ruido y obtener características estables.

La odometría del Viso2 se ejecuta mediante un archivo ".launch" del que se pueden destacar algunos fragmentos:

```
#Estos nodos publican las imágenes estéreo descomprimidas para poder ser usadas posteriormente
  <node name="republish" type="republish" pkg="image_transport" output="screen" args="compressed in:=$(arg camera)/left/image_raw raw out:=$(arg camera)/left/image_raw" />
   <node name="republish1" type="republish" pkg="image_transport" output="screen" args="compressed in:=$(arg camera)/right/image_raw raw out:=$(arg camera)/right/image_raw" />
#Este nodo realiza el procesamiento de las imágenes estéreo, como la rectificación y mapas de disparidad.
<group ns="$(arg camera)" >
  <nodepkg="stereo_image_proc "type="stereo_image_proc" name="stereo_image_proc">
  </node>
</group>
#Este nodo publica una transformación estática entre los marcos de referencia "base_link" y "ps4eye_frame". Esto permite relacionar el marco de referencia de la base del robot con el marco de referencia de la cámara estéreo.
<node pkg="tf2_ros" type="static_transform_publisher" name="ps4eye"
    args="0 0 0 -1.58 0 -1.45 base_link ps4eye_frame" />
```

```
#Este nodo ejecuta el algoritmo de odometría estéreo. Se establecen varios parámetros, como el marco de referencia de la odometría ("odom"), el marco de referencia del sensor de la cámara ("ps4eye_frame") y el marco de referencia de la base del robot ("base_link").
<node pkg="viso2_ros" type="stereo_odometer" name="stereo_odometer" output="screen">
    <remap from="stereo" to="$(arg camera)"/>
    <remap from="image" to="image_rect"/>
    <param name="odom_frame_id" value="odom"/>
    <param name="sensor_frame_id" value="ps4eye_frame"/>
    <param name="base_link_frame_id" value="base_link"/>
</node>
```

La totalidad del código se encuentra recogida en el Anexo I de este documento. En la figura 4.2 se puede apreciar la trayectoria descrita por la silla según el algoritmo Viso2.

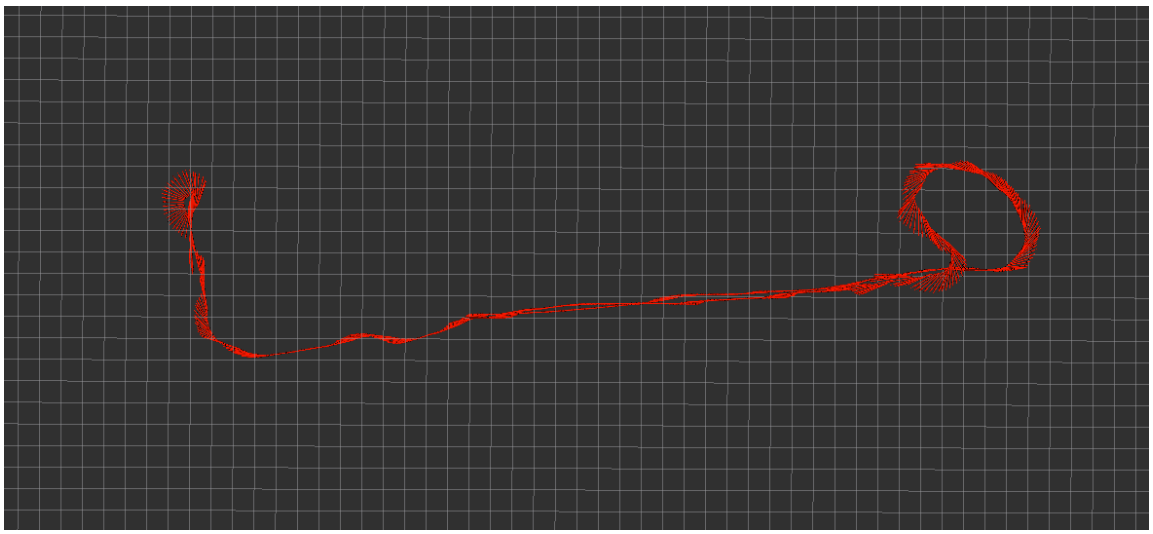

Figura 4.2

4.3.2 RTAB-Map

RTAB-Map (Real-Time Appearance-Based Mapping) es una herramienta de generación de mapas 3D y estimación de la posición dentro de dicho mapa, basada en el algoritmo FAST y GraphSLAM [33]. Su funcionamiento está basado en la extracción de características de imágenes captadas por sensores de cámaras RGB-D. Esta herramienta utiliza información visual y de profundidad, obtenida a partir de cámaras estéreo, para realizar un mapeo del entorno que rodea al robot y la estimación de la posición del robot dentro de este. Además, ofrece ciertas ventajas frente al Viso2 ya que al tratarse de una técnica SLAM, ofrece la posibilidad de implementar cierre de bucles (*loop closure*), que optimiza

el mapa y permite corregir errores, ya que es capaz de reconocer lugares transitados con anterioridad [34].

El código del archivo ".launch" que se ejecuta para poner en funcionamiento este algoritmo es bastante similar al del viso2, aunque consta de algunos nodos destacables como el siguiente:

```
#Este nodo se encarga de realizar el mapeo, 'rtabmap' utiliza la odometría estéreo y las imágenes para realizar el mapeo utilizando el algoritmo RTAB-Map. Además, se establecen varios parámetros que configuran el comportamiento del nodo, como la estrategia de detección de puntos clave o la resolución del mapa creado.
<node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen" args="--delete_db_on_start">
        <param name="frame_id"        type="string" value="base_link"/>
        <param unless="$(arg stereo_sync)" name="subscribe_stereo" type="bool" value="true"/>
        <param name="subscribe_depth"  type="bool" value="false"/>
        <param name="subscribe_rgbd"      type="bool" value="$(arg stereo_sync)"/>
                <remap from="left/image_rect"   to="/stereo/left/image_rect_color"/>
                <remap from="right/image_rect"  to="/stereo/right/image_rect"/>
                <remap from="left/camera_info"  to="/stereo/left/camera_info"/>
                <remap from="right/camera_info" to="/stereo/right/camera_info"/>
                <remap from="rgbd_image"        to="/stereo/rgbd_image"/>
                <remap from="odom" to="/stereo_odometry"/>
                <param name="queue_size" type="int" value="100"/>
                <param name="map_negative_poses_ignored" type="bool" value="true"/>
                <!-- RTAB-Map's parameters -->
                <param name="Rtabmap/TimeThr"                type="string" value="700"/>
                <param name="Grid/DepthDecimation"           type="string" value="4"/>
                <param   name="Grid/FlatObstacleDetected"                  type="string" value="true"/>
                <param name="Kp/MaxDepth"                    type="string" value="0"/>
                <param name="Kp/DetectorStrategy"            type="string" value="6"/>
                <param name="Vis/EstimationType"             type="string" value="1"/>  <!- - 0=3D->3D, 1=3D->2D (PnP) -->
                <param name="Vis/MaxDepth"                   type="string" value="0"/>
                <param  name="RGBD/CreateOccupancyGrid"                  type="string" value="true"/>
            </node>
```

La totalidad del código se encuentra recogida en el Anexo I de este documento.

En la figura 4.3 se puede apreciar la trayectoria descrita por la silla según el algoritmo RTAB-Map. Por otro lado, en las figuras 4.4 y 4.5 se muestra el mapa 3D generado en Rviz.

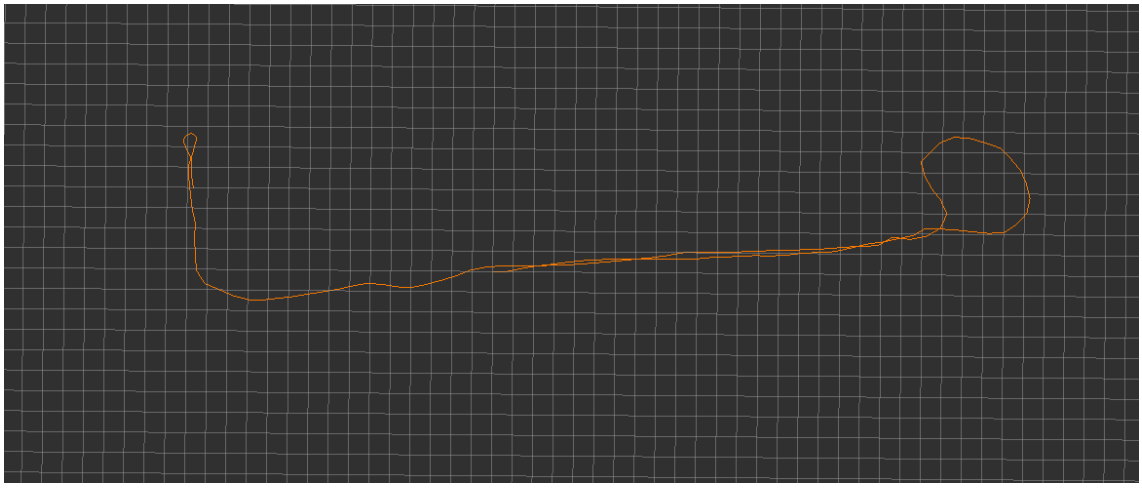

Figura 4.3

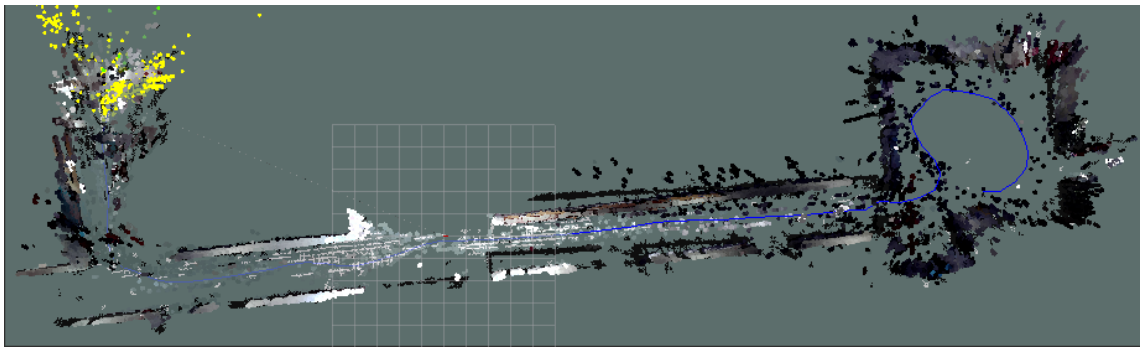

Figura 4.4

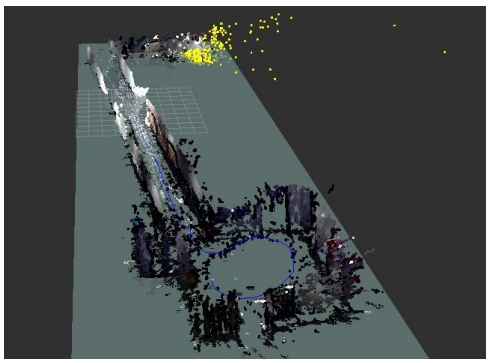

Figura 4.5

Las figuras 4.6 y 4.7 muestran los puntos característicos detectados. Asimismo, en las figuras 4.8 y 4.9 se observa el mapa 3D creado por el algoritmo. Estas imágenes han sido obtenidas a través del propio visor del RTAB-Map llamado “rtabmaprviz”.

Figura 4 .0.1

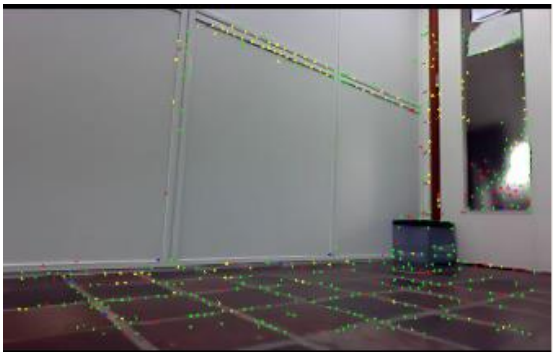

Figura 4.7

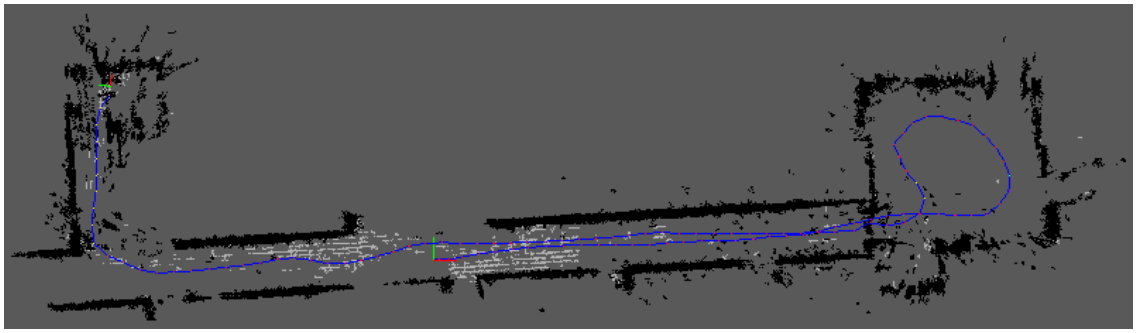

Figura 4.8

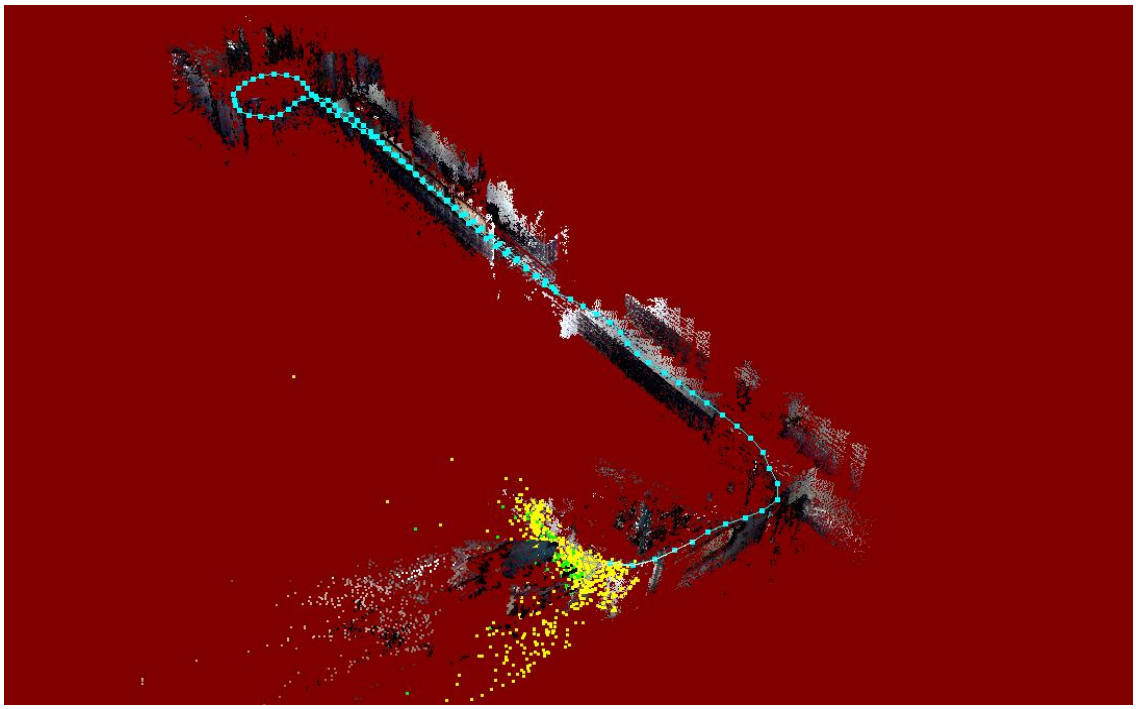

Figura 4.9

4.3.3 ORB-SLAM

El ORB-SLAM es un algoritmo, basado en el método de extracción ORB, diseñado para estimar la posición de un robot y construir un mapa del entorno a medida que este se mueve [35]. La explicación teórica de su funcionamiento se encuentra en el apartado 2.2.8.1 de este mismo documento. Al tratarse de un algoritmo SLAM, es capaz de utilizar técnicas de optimización como el cierre de bucles (*loop closure*).

Para ser ejecutado en ROS, es necesario ejecutar dos archivos, uno ".launch" y otro de Python ".py", este último se encarga de pasar la posición obtenida por el ORB-SLAM a odometría. Con el objetivo de evitar abrir más terminales, es posible ejecutar un código ".py" desde el archivo ".launch" a través del siguiente *script:*

```
<node pkg="orb_slam2_ros" type="Pose_to_Odometry2.py" name="odometry" output="screen"/>
```

Aunque el funcionamiento es relativamente parecido al de los algoritmos Viso2 y RTABMap, es necesario explicar algunos fragmentos de interés del código.

```
#Este nodo ejecuta el nodo orb_slam2_ros_stereo del paquete orb_slam2_ros. Es el responsable de realizar el SLAM, estimando la posición de la silla y construyendo un mapa del entorno en 3D utilizando las imágenes estéreo. Además, se definen varios parámetros y configuraciones para este nodo, como los tópicos de la cámara, los marcos de referencia, los parámetros de la cámara y varios ajustes de ORB-SLAM.
<node name="slam" pkg="orb_slam2_ros"
    type="orb_slam2_ros_stereo" output="screen">
    <!-- Camera topics to listen on -->
    <remap from="image_left/image_color_rect" to="/stereo/left/image_rect"/>
    <remap from="image_right/image_color_rect" to="/stereo/right/image_rect"/>
    <remap from="image_left/camera_info" to="/stereo/left/camera_info" />
    ……..
    ……..
    ……..
#Marco de referencia utilizado por la nube de puntos.
    <param name="pointcloud_frame_id" type="string" value="map" />
#Marco de referencia utilizado por la cámara.
    <param name="camera_frame_id" type="string" value="ps4eye_frame" />
#Marco de referencia al que se transforman los mensajes de orientación y posición.
    <param name="target_frame_id" type="string" value="ps4eye_frame" />
#Número de keyframes (cuadros clave) necesarios antes de realizar optimización.
```

```
    <param name="min_num_kf_in_map" type="int" value="8" />
#Parametros ORB
#Número máximo de características extraídas por keyframe.
    <param name="/ORBextractor/nFeatures" type="int" value="2000" />
#Factor de escala utilizado para construir una pirámide de imágenes en la extracción de características.
    <param name="/ORBextractor/scaleFactor" type="double" value="1.2" />
#Número de niveles de la pirámide.
    <param name="/ORBextractor/nLevels" type="int" value="8" />
#Umbral del detector FAST.
    <param name="/ORBextractor/iniThFAST" type="int" value="12" />
#Umbral mínimo del detector FAST.
    <param name="/ORBextractor/minThFAST" type="int" value="1" />

#Parámetros de la cámara.
#Tasa de fotogramas por segundo.
    <param name="camera_fps" type="int" value="70" />
#Especifica el orden de color de las imágenes (0:BGR, 1:RGB).
    <param name="camera_rgb_encoding" type="bool" value="false" />
#Umbral de distancia utilizado para filtrar valores de profundidad cercanos y lejanos.
    <param name="ThDepth" type="double" value="50.0" />
#Relaciona la profundidad con el valor del píxel. Un píxel con valor de 1 representa 1 m.
    <!--param name="depth_map_factor" type="double" value="1" -->

#Parámetros de calibración de la cámara
    <!-- If the node should wait for a camera_info topic to take the camera calibration data -->
    <param name="load_calibration_from_cam" type="bool" value="false" />
#Son parametros intrínsecos de la cámara (focal, centro óptico).
    <param name="camera_fx" type="double" value="430.52" />
    <param name="camera_fy" type="double" value = "429.515" />
    <param name="camera_cx" type="double" value="311.699" />
    <param name="camera_cy" type="double" value="194.045" />
#Son los coeficientes de distorsión radial y tangencial de la cámara.
    <param name="camera_k1" type="double" value="-0.013794" />
    <param name="camera_k2" type="double" value="-0.013489" />
    <param name="camera_p1" type="double" value="-0.001369" />
    <param name="camera_p2" type="double" value="-0.001168" />
    <param name="camera_k3" type="double" value="0.0" />
    <!-- IR projector baseline times fx (aprox.) -->
    <param name="camera_baseline" type="double" value="34.57" />
 </node>
```

La totalidad del código se encuentra recogida en el Anexo I de este documento.

En la figura 4.10 se puede apreciar la trayectoria descrita por la silla según el algoritmo ORB-SLAM.

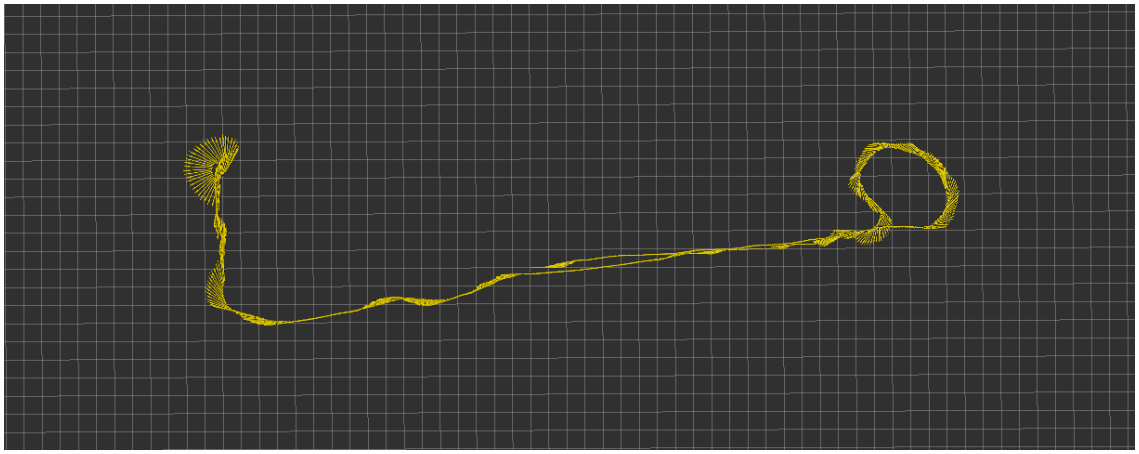

Figura 4.10

En las figuras 4.11 y 4.12 es posible observar los puntos característicos detectados. Asimismo, en las figuras 4.13 y 4.14 se muestra el mapa 3D creado por el algoritmo.

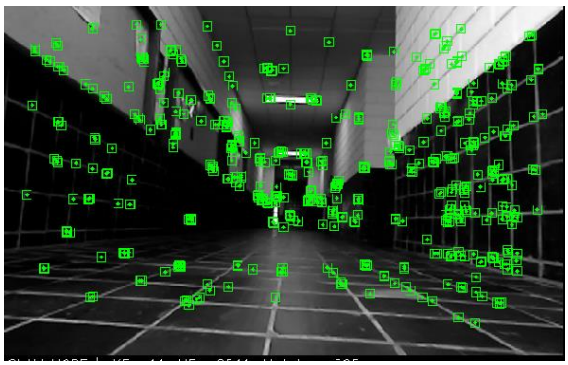

Figura 4.11

Figura 4.12

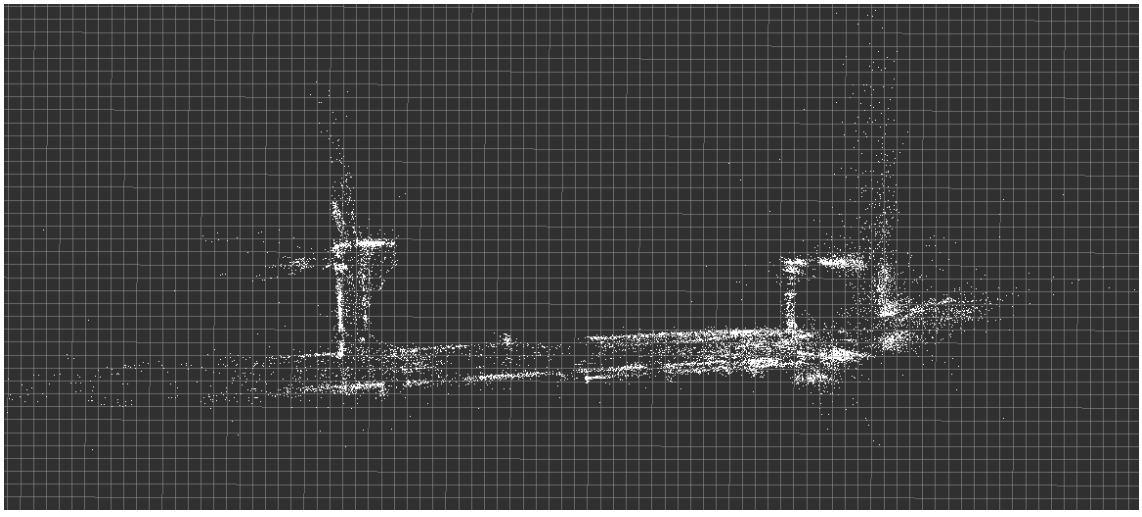

Figura 4.13

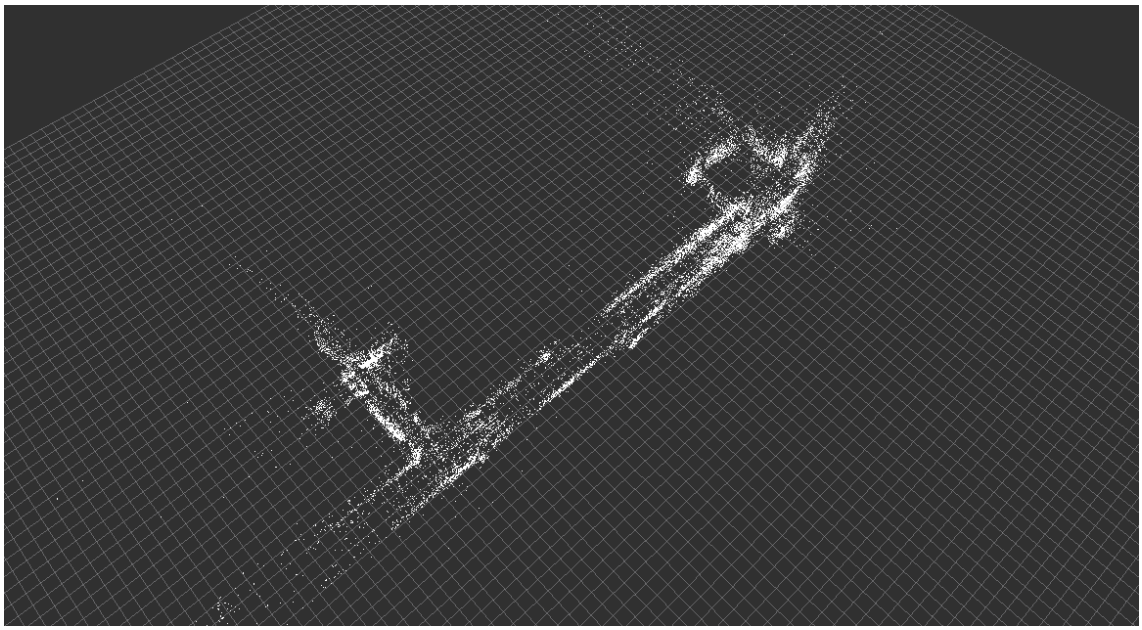

Figura 4.14

4.4 Implementación del A-LOAM

El A-LOAM (*Adaptative Lidar Odometry and Mapping*) es un algoritmo de mapeo y localización simultanea basado en la tecnología láser Lidar (*Light Detection and Ranging*) [36]. A partir de la nube de puntos generada por el sensor Velodyne, capaz de realizar un reconocimiento de 360º del entorno, se extraen elementos característicos de este, como bordes, esquinas o superficies planas. Posteriormente, se buscan las correspondencias más probables entre los puntos característicos en diferentes escaneos a lo largo del tiempo, permitiendo establecer la relación de movimiento entre dichos puntos, para estimar el movimiento del robot. Además, el algoritmo es capaz de generar un mapa 3D bastante detallado y preciso del entorno transitado. Dado que se trata de una técnica muy precisa y fiable, la trayectoria obtenida por este algoritmo se utilizará como referencia de la trayectoria real de la silla de ruedas, lo que permitirá estudiar la eficacia y rendimiento del resto de algoritmos implementados.

En la figura 4.15 se puede apreciar la trayectoria descrita por la silla según el algoritmo A-LOAM. Por otro lado, en las figuras 4.16 y 4.17 es posible observar el mapa 3D generado a partir de los datos recopilados por el sensor Velodyne.

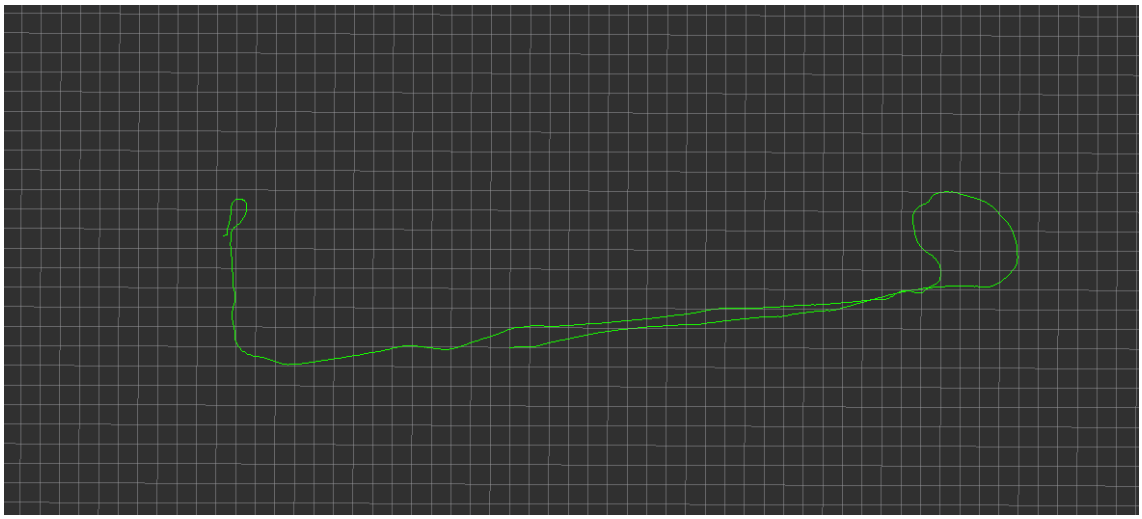

Figura 4.15

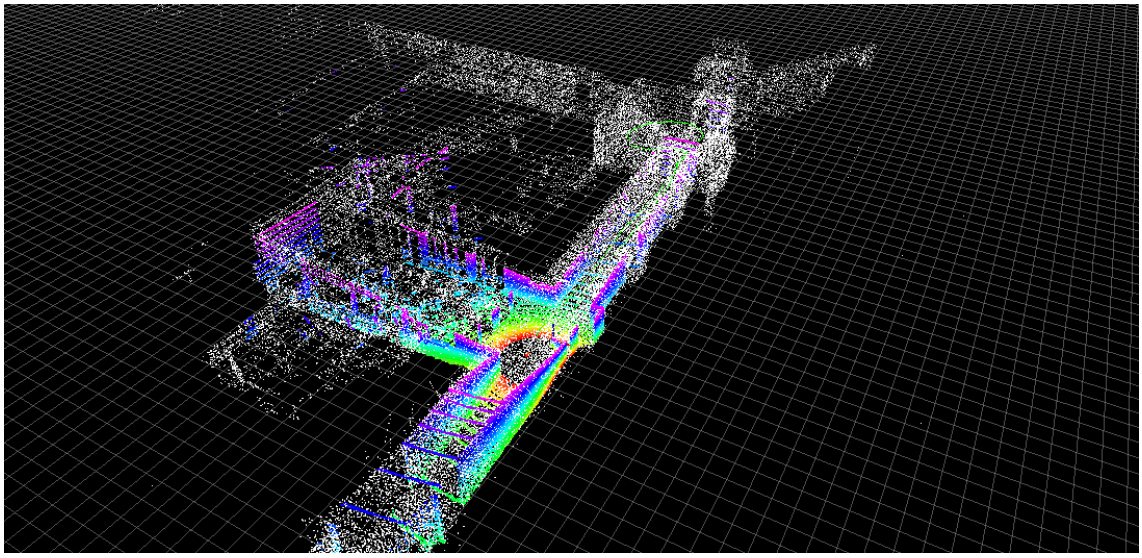

Figura 4.16

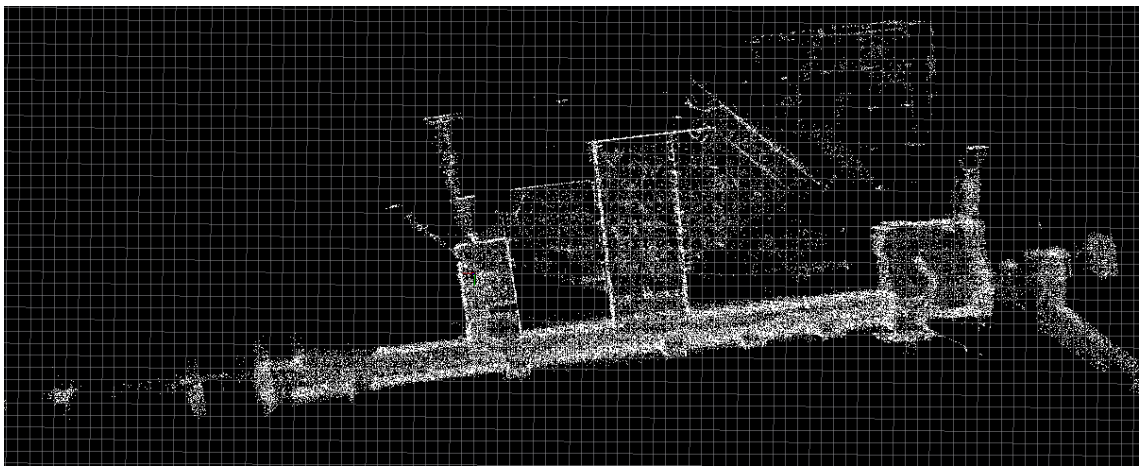

Figura 4.17

Capítulo 5

Resultados

En este capítulo se relata minuciosamente todas las pruebas y ensayos realizados, con el objetivo de realizar una comparación y análisis de los diferentes algoritmos de odometría visual implementados. Para obtener más información con la que comparar el desempeño de cada uno, se hará uso de dos recorridos diferentes que han sido grabados en el pasillo de la Facultad de Física y Matemáticas de la ULL.

En primer lugar, se mostrarán los resultados de los diferentes algoritmos por separado, realizando únicamente la comparación con el *ground truth* del A-LOAM. Posteriormente, se presentará una simulación conjunta de todos los algoritmos, para poder apreciar de manera visual y sencilla las diferencias entre los resultados de cada uno. Por último, se calculará el error relativo de la posición final de cada trayectoria, para medir la precisión de cada algoritmo implementado, tomando como referencia la odometría del sensor Velodyne. Además, se hará un estudio del coste computacional de cada algoritmo.

Los dos recorridos grabados en el pasillo de la Facultad de Física y Matemáticas de la ULL son muy similares, la principal diferencia entre el recorrido 1 y el recorrido 2 es que, en el segundo, el punto de inicio es el mismo que el punto final. A continuación, en las figuras 5.1 y 5.2 se muestra de forma esquemática los recorridos:

Recorrido 1: (punto rojo = posición inicial, punto amarillo = posición final)

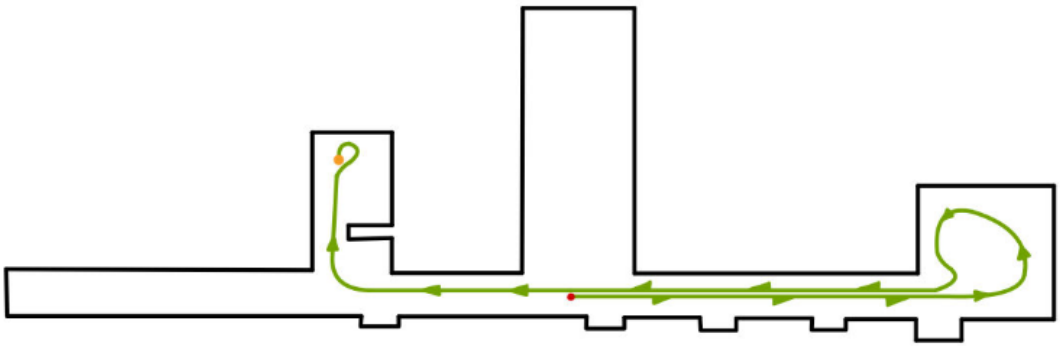

Figura 5.1

Recorrido 2: (punto rojo = posición inicial y final)

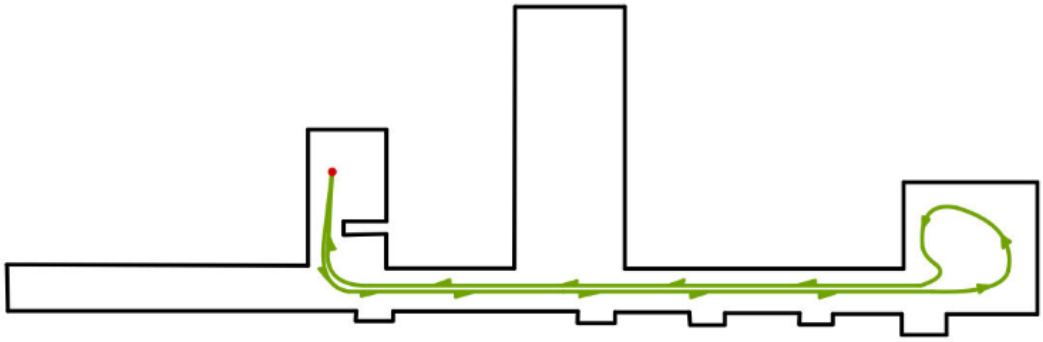

Figura 5.2

5.1 Resultados de la Odometría mecánica

Los resultados de la simulación del recorrido 1 se pueden observar en figura 5.3, siendo la trayectoria verde la dada por el velodyne y la azul la odometría mecánica.

A partir de la información dada por rviz, es posible saber las posiciones iniciales y finales de cada trayectoria. Conociendo estas posiciones, se calcula el error tomando como punto de estudio la posición final de la trayectoria.

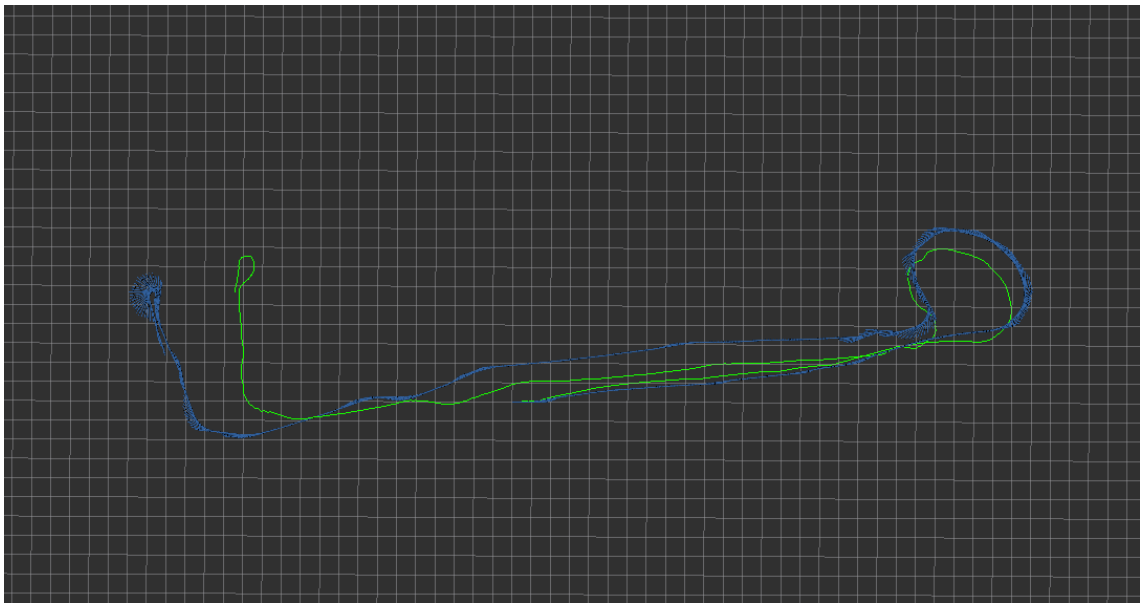

Figura 5.3

La posición inicial de ambas odometrías es la (0, 0, 0)m. Las posiciones finales son relativas a esta posición inicial.

La posición final del *ground truth* es (-14.1, 6.47, 0)m.
La posición final de la odometría mecánica es (-18.2, 3.01, 0)m.

Error absoluto en el eje X: $\varepsilon = |-14.1 - (-18.2)| = 4.1\ m$

Error absoluto en el eje Y: $\varepsilon = |6.47 - 3.01| = 3.46\ m$

Los resultados de la simulación del recorrido 2 se pueden observar en figura 5.4,

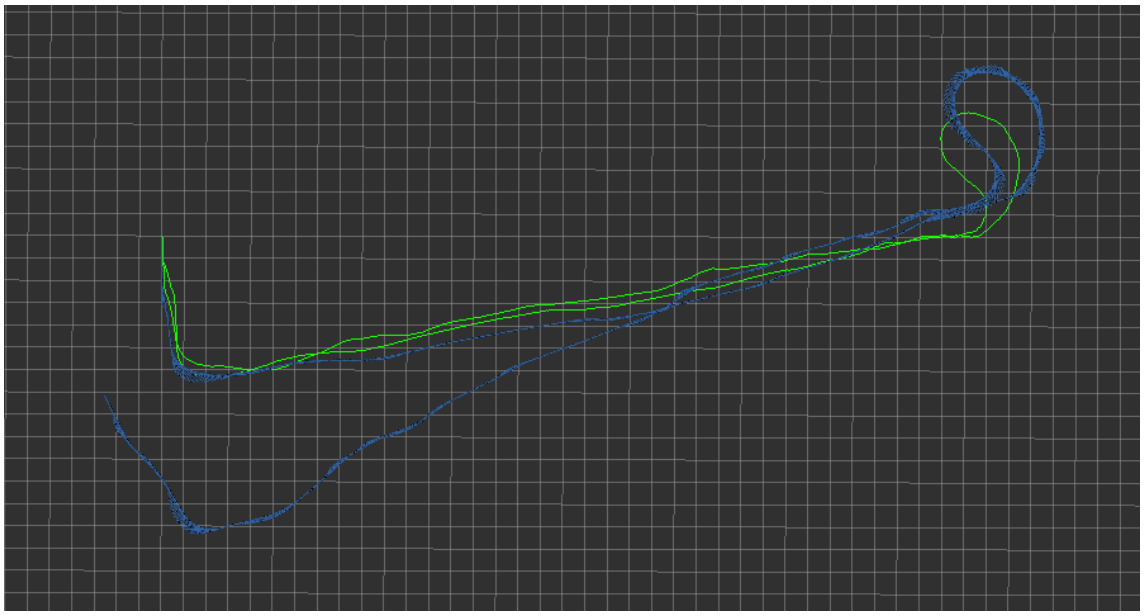

Figura 5.4

La posición inicial de ambas odometrías es la (0, 0, 0)m. Las posiciones finales son relativas a esta posición inicial.

La posición final del *ground truth* es (0, 0, 0)m.
La posición final de la odometría mecánica es (-2.57, -7.1, 0)m.

Error absoluto en el eje X: $\varepsilon = |0 - (-2.57)| = 2.57\ m$

Error absoluto en el eje Y: $\varepsilon = |0 - (-7.1)| = 7.1m$

Aunque se puede apreciar claramente en la simulación que la trayectoria de la odometría mecánica difiere bastante de la real en los dos recorridos, el error absoluto reafirma que se produce una variación importante de la posición final en ambos ejes, especialmente en el eje Y.

En conclusión, en ambas simulaciones es posible observar como la odometría mecánica pierde precisión a medida que avanza la silla, es decir, la trayectoria dada por la odometría mecánica es más fiel a la real en los primeros

instantes. Al tratarse de una técnica que acumula errores sistemáticos y no sistemáticos, como el derrape de las ruedas, la trayectoria pierde precisión a medida que se acumulan estos errores. En consecuencia, se obtienen posiciones finales muy dispares respecto al *ground truht,* de hecho, el error se acentúa en el segundo recorrido, ya que este es más largo.

5.2 Resultados del Viso2

Los resultados de la simulación del recorrido 1 se pueden observar en figura 5.5, siendo la trayectoria verde la dada por el Velodyne y la roja la odometría estéreo del Viso2.

A partir de la información dada por rviz, es posible saber las posiciones iniciales y finales de cada trayectoria. Conociendo estas posiciones, se calcula el error tomando como punto de estudio la posición final de la trayectoria.

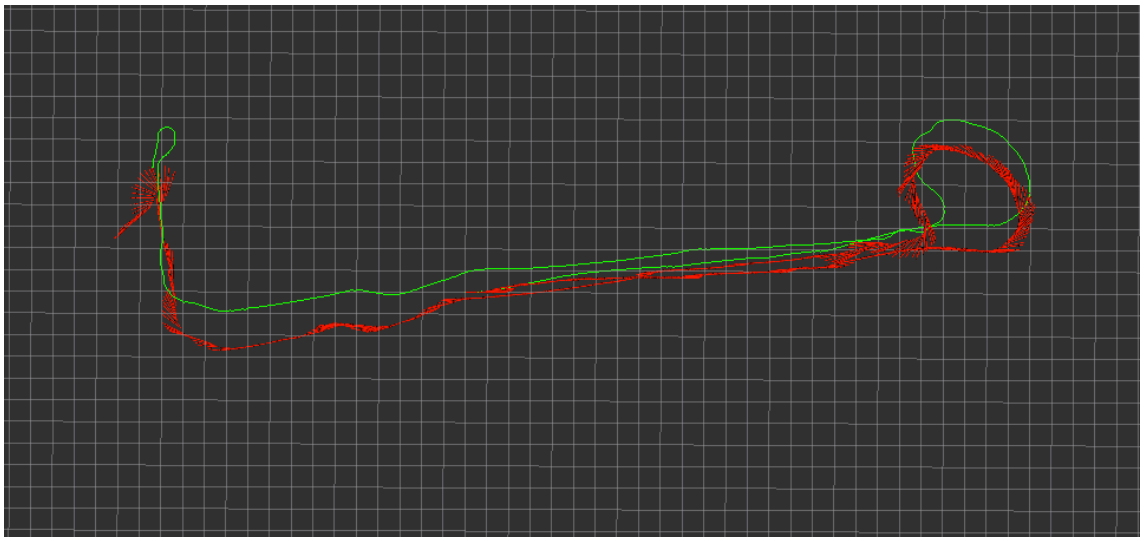

Figura 5.5

La posición inicial de ambas odometrías es la (0, 0, 0)m. Las posiciones finales son relativas a esta posición inicial.

La posición final del *ground truth* es (-14.1, 6.47, 0)m.
La posición final de la odometría Viso2 es (-16.1, 4.23, 0)m.

Error absoluto en el eje X: $\varepsilon = |-14.1 - (-16.1)| = 2\,m$

Error absoluto en el eje Y: $\varepsilon = |6.47 - 4.23)| = 2.24\,m$

Los resultados de la simulación del recorrido 2 se pueden observar en figura 5.6.

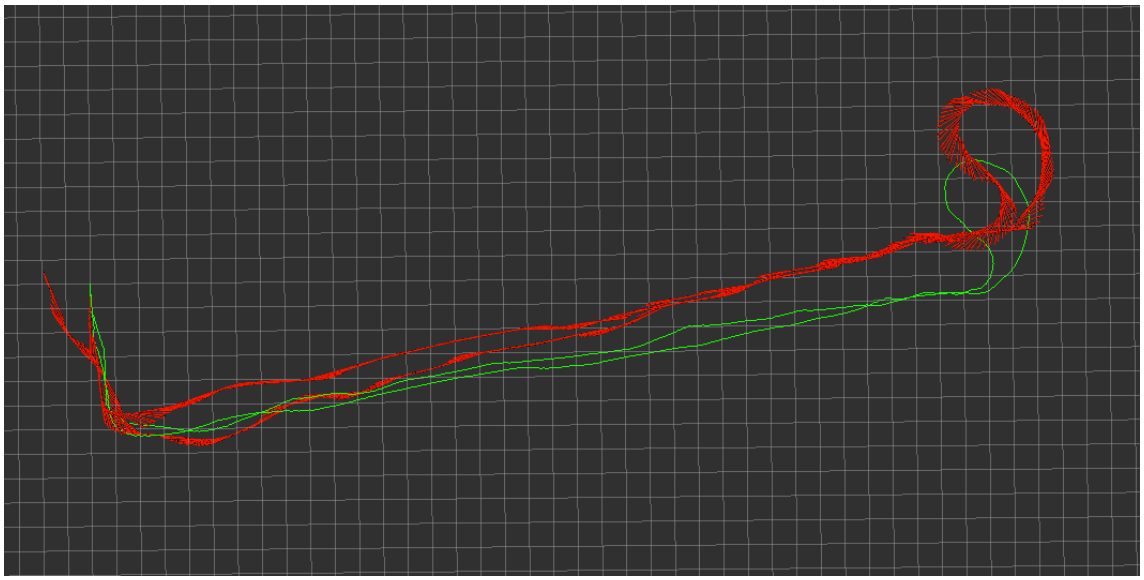

Figura 5.6

La posición inicial de ambas odometrías es la (0, 0, 0)m. Las posiciones finales son relativas a esta posición inicial.

La posición final del *ground truth* es (0, 0, 0)m.
La posición final de la odometría Viso2 es (-1.9, 0.5, 0)m.

Error absoluto en el eje X: $\varepsilon = |0 - (-1.4)| = 1.9\, m$

Error absoluto en el eje Y: $\varepsilon = |0 - 5.24| = 0.5\, m$

En conclusión, en ambos recorridos se puede observar cierta disparidad entre la trayectoria del Viso2 y la trayectoria real. El error de la posición final es bastante menor que en la odometría mecánica, demostrando que se trata de una técnica más precisa y eficiente. Además, se puede observar como la zona más conflictiva es la curva, ya que en este tramo las características de las imágenes varían mucho más rápido que en tramos rectos.

5.3 Resultados del RTAB-Map

Los resultados de la simulación del recorrido 1 se pueden observar en figura 5.7, siendo la trayectoria verde la dada por el Velodyne y la naranja la odometría estéreo del RTAB-Map.

A partir de la información dada por rviz, es posible saber las posiciones iniciales y finales de cada trayectoria. Conociendo estas posiciones, se calcula el error tomando como punto de estudio la posición final de la trayectoria.

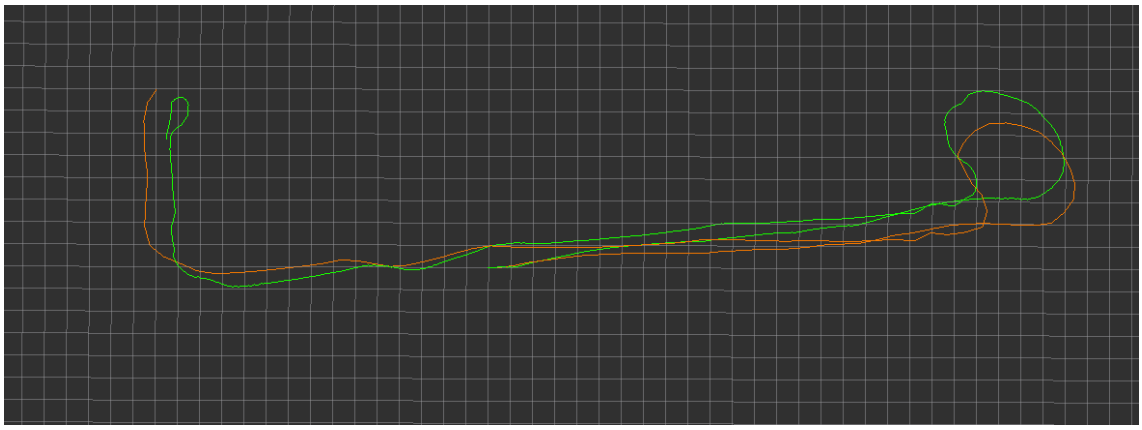

Figura 5.7

La posición inicial de ambas odometrías es la (0, 0, 0)m. Las posiciones finales son relativas a esta posición inicial.

La posición final del *ground truth* es (-14.1, 6.47, 0)m.
La posición final de la odometría Viso2 es (-15, 8, 0)m.

Error absoluto en el eje X: $\varepsilon = |-14.1 - (-15)| = 0{,}9\ m$

Error absoluto en el eje Y: $\varepsilon = |6.47 - 8)| = 1.53\ m$

Los resultados de la simulación del recorrido 2 se pueden observar en figura 5.8.

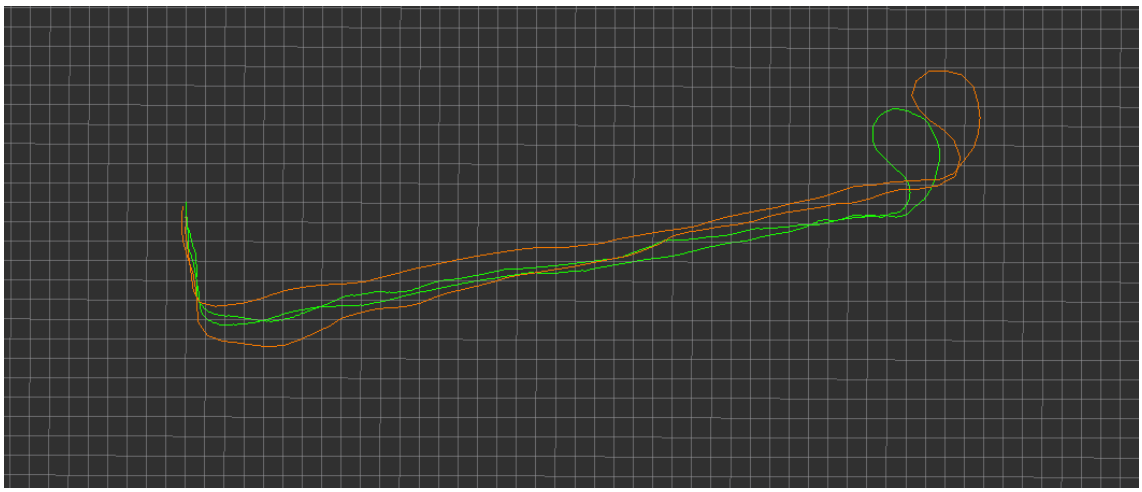

Figura 5.8

La posición inicial de ambas odometrías es la (0, 0, 0)m. Las posiciones finales son relativas a esta posición inicial.

La posición final del *ground truth* es (0, 0, 0)m.

La posición final de la odometría Viso2 es (-0.3, -0.1, 0)m.

Error absoluto en el eje X: $\varepsilon = |0 - (-0.3)| = 0.3\ m$

Error absoluto en el eje Y: $\varepsilon = |0 - 0.1| = 0.1\ m$

En conclusión, el algoritmo RTAB-Map es bastante preciso y fiable, como demuestra el error absoluto calculado sobre la posición final, ya que no supera las unidades de metro. Sin embargo, a pesar de que la posición final estimada difiere muy poco de la real, a lo largo de la trayectoria es posible observar una mayor disparidad, sobre todo en el tramo de la curva.

5.4 Resultados del ORB-SLAM

Los resultados de la simulación del recorrido 1 se pueden observar en figura 5.9, siendo la trayectoria verde la dada por el Velodyne y la amarilla la odometría estéreo del ORB-SLAM.

A partir de la información dada por rviz, es posible saber las posiciones iniciales y finales de cada trayectoria. Conociendo estas posiciones, se calcula el error tomando como punto de estudio la posición final de la trayectoria.

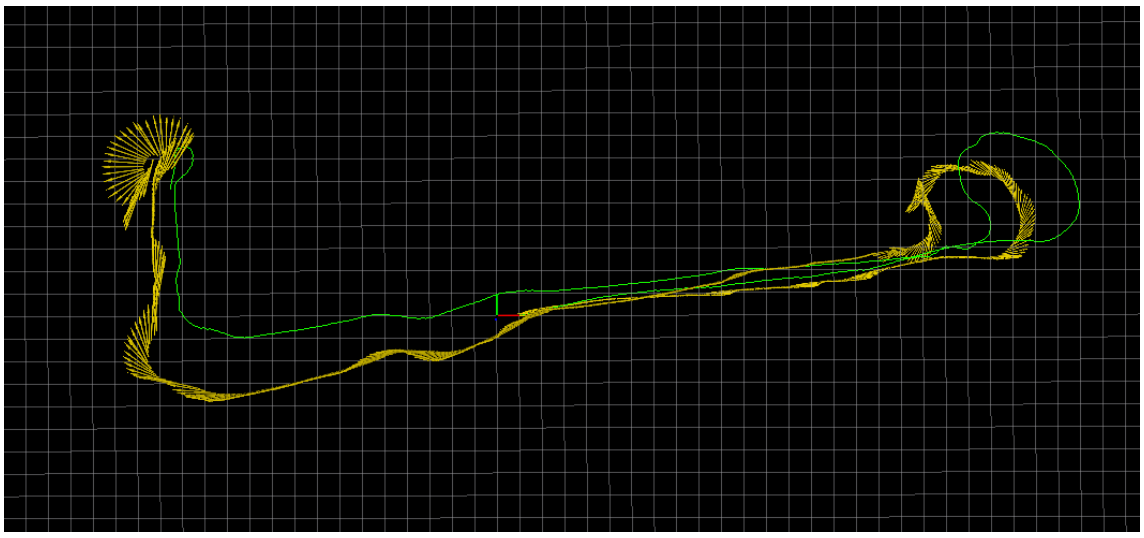

Figura 5.9

La posición inicial de ambas odometrías es la (0, 0, 0)m. Las posiciones finales son relativas a esta posición inicial.

La posición final del *ground truth* es (-14.1, 6.47, 0)m.
La posición final de la odometría Viso2 es (-15, 7.1, 0)m.

Error absoluto en el eje X: $\varepsilon = |-14.1 - (-15)| = 1.1\, m$

Error absoluto en el eje Y: $\varepsilon = |6.47 - 7.1)| = 0.57\, m$

Los resultados de la simulación del recorrido 2 se pueden observar en figura 5.10.

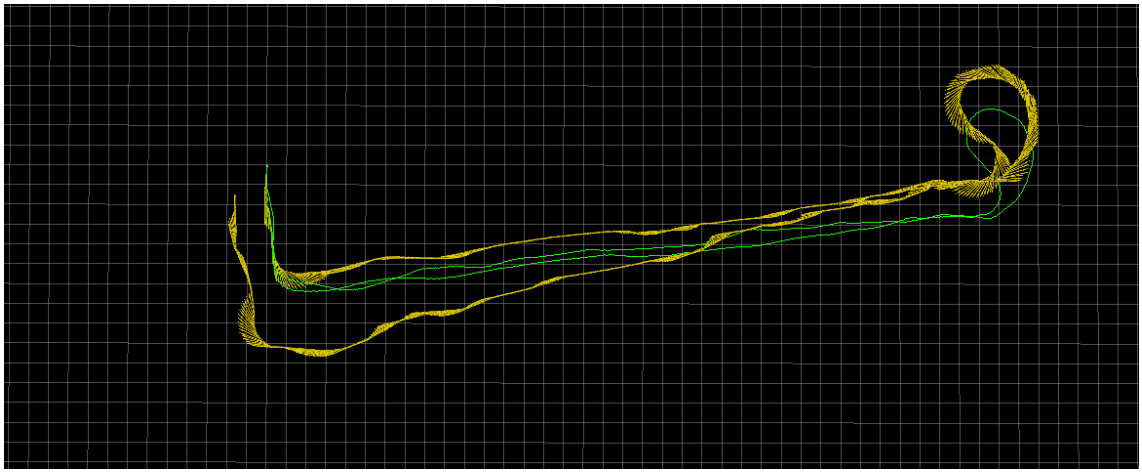

Figura 5.10

La posición inicial de ambas odometrías es la (0, 0, 0)m. Las posiciones finales son relativas a esta posición inicial.

La posición final del *ground truth* es (0, 0, 0)m.
La posición final de la odometría Viso2 es (-1.59, -1.3, 0)m.

Error absoluto en el eje X: $\varepsilon = |0 - (-1.59)| = 1.59\, m$

Error absoluto en el eje Y: $\varepsilon = |0 - (-1.3)| = 1.3\, m$

En conclusión, el algoritmo ORB-SLAM es bastante preciso y fiable, como demuestra el error absoluto calculado sobre la posición final. Sin embargo, a pesar de que la posición final estimada difiere muy poco de la real, a lo largo de la trayectoria es posible observar una mayor disparidad, sobre todo en el tramo de la curva, siendo este tramo el más crítico de la trayectoria.

5.5 Simulación conjunta

Con el objetivo de comparar visualmente todas las técnicas de odometría, es necesario simularlas en el mismo Rviz. A continuación, en la figura 5.11, se muestran todas las trayectorias obtenidas del recorrido 1 simuladas al mismo tiempo.

- Odometría mecánica: azul
- Odometría VISO2: rojo
- A-LOAM: verde
- RTAB-Map:amarillo
- ORB-SLAM: violeta

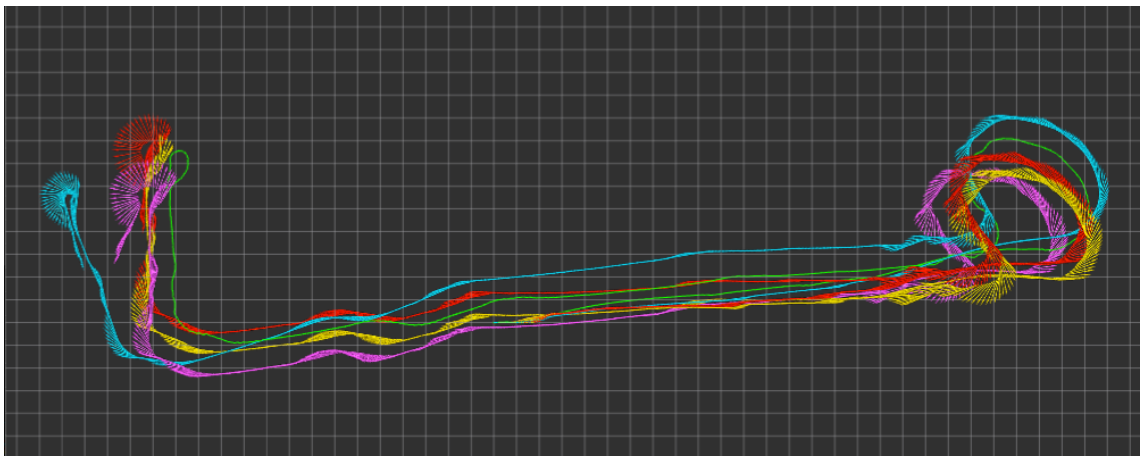

Figura 5.11

En la figura 5.12, se muestran todas las trayectorias obtenidas del recorrido 2 simuladas al mismo tiempo.

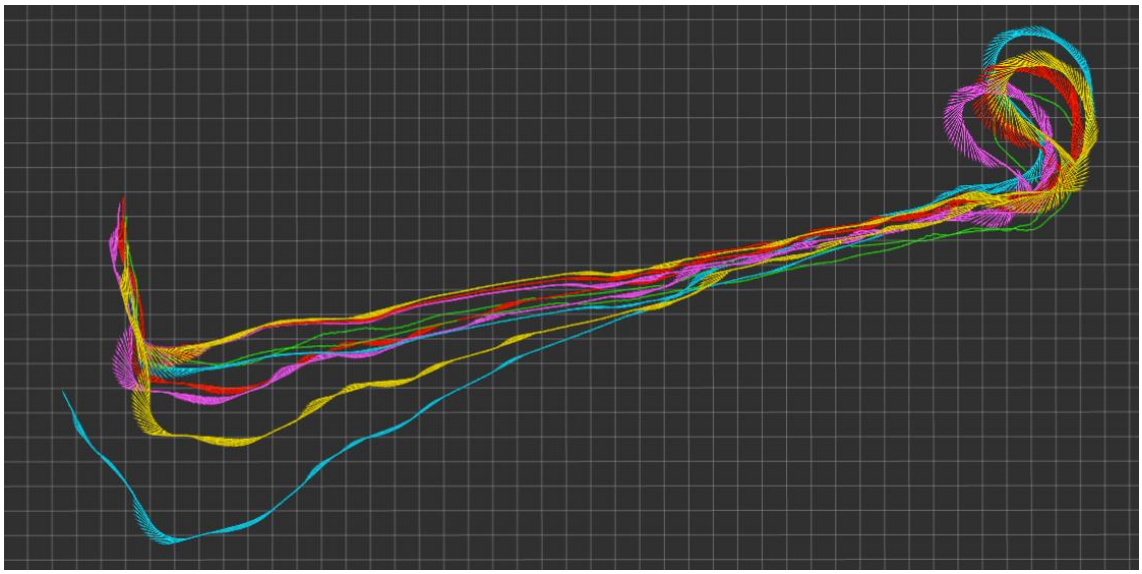

Figura 5.12

En la tabla 5.1 y las figuras 5.13 y 5.14, se recogen los errores absolutos de la posición final de cada trayectoria respecto a la trayectoria real. De la cual se puede deducir que la técnica de odometría más fiable y precisa es el RTAB-Map, seguida de cerca por el ORB-SLAM. El algoritmo Viso2 comete un error más notorio, ya que no dispone de las técnicas de optimización del SLAM. Por último, la odometría mecánica es la que más difiere de la realidad, debido a la acumulación de errores sistemáticos y no sistemáticos.

Metodología	Recorrido 1		Recorrido 2	
	Error eje X [m]	Error eje Y [m]	Error eje X [m]	Error eje Y [m]
Odometría mecánica	4,1	2,57	3,46	7,1
VIso2	2	2,24	1,9	0,5
RTAB-Map	0,9	1,53	0,3	0,1
ORB-SLAM	1,1	0,57	1,59	1,3

Tabla 5.1

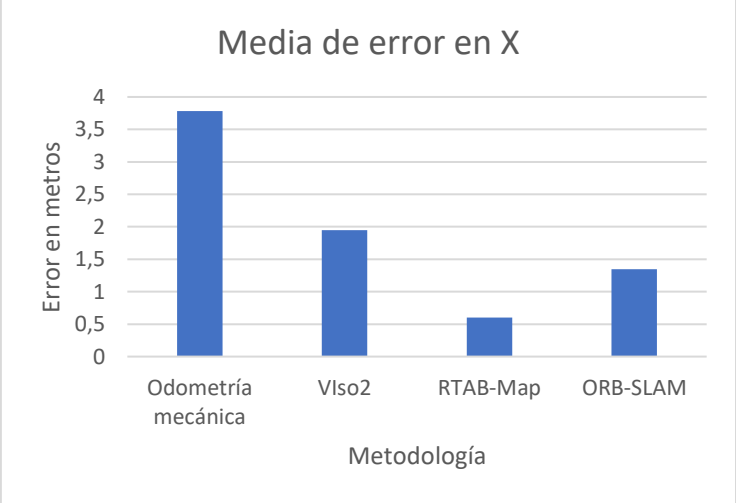

Figura 5.13

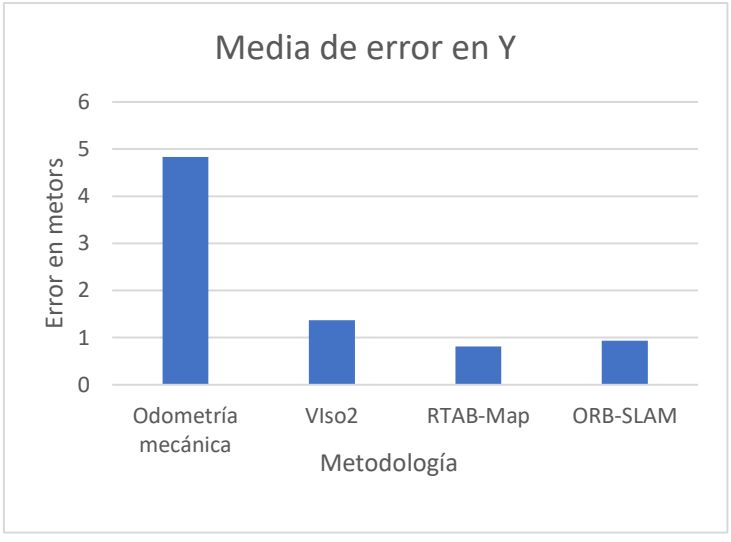

Figura 5.14

5.6 Coste computacional

El coste computacional que requiere cada algoritmo es un factor muy importante que tener en cuenta. La eficiencia en tiempo real es crucial para que los algoritmos sean capaces de procesar la información y generar la estimación de posición lo más rápido posible. Si un algoritmo es demasiado costoso, puede generar retrasos que afectan negativamente a la capacidad de respuesta y rendimiento del sistema.

A través de comando “top” ejecutado en el terminal, es posible obtener el tiempo de ejecución de cada algoritmo, como se muestra en la tabla 5.2, en la que se puede observar como el ORB-SLAM es el método más costoso, seguido por el Viso2. Sorprendentemente, a pesar de tratarse de un método SLAM, el RTAB-Map es el algoritmo visual más rápido. Por último, la odometría mecánica es la técnica más sencilla, por lo que es natural que su coste computacional sea tan bajo.

Metodología	Recorrido 1	Recorrido 2
	tiempo [min]	tiempo [min]
Odometría mecánica	0:11.18	0:08.33
VIso2	2:26.66	2:34.21
RTAB-Map	1:00.84	0:41.70
ORB-SLAM	4:33.36	5:44.28

Tabla 5.2

Capítulo 6

Conclusiones y líneas futuras

6.1 Conclusiones

En este trabajo se ha realizado un estudio sobre el rendimiento y eficiencia de diferentes algoritmos de odometría visual, implementados en una silla de ruedas autónoma. El objetivo de este proyecto es determinar que método o métodos son los más eficientes para esta aplicación en concreto. Además, también se ha realizado la comparación con la odometría mecánica, demostrando las desventajas que esta presenta frente a los algoritmos visuales.

El entorno donde se han realizado las pruebas es el pasillo de la Facultad de Física y Matemáticas de la ULL, debido a que se trata de un espacio cerrado en el que la odometría visual es más eficiente. Se han realizado dos pruebas en este espacio, un recorrido circular y otro lineal, para poner a prueba los algoritmos ante diferentes situaciones y comprobar la eficiencia del cierre de bucles.

Como se ha podido comprobar en las pruebas realizadas y simuladas en ROS y Rviz, los algoritmos de odometría visual son bastante fiables y efectivos a pesar de los errores cometidos. Los métodos que han obtenido mejores resultados son el ORB-SLAM y el RTAB-Map, especialmente este último, que además tiene un coste computacional menor. Estos se benefician de técnicas de optimización y corrección de errores como el *loop closure,* solo disponibles en algoritmos SLAM. Por otro lado, a pesar de que el Viso2 no cuenta con estas ventajas, las trayectorias obtenidas no difieren demasiado de la real.

En resumen, los algoritmos de odometría visual capaces de realizar mapeos 3D del entorno utilizando el SLAM obtienen resultados más precisos y fiables. El Viso2 ha obtenidos resultados bastante similares por lo que no es una mala opción, aunque no es tan efectivo y no permite construir mapas 3. Por último, la odometría mecánica es la técnica que más se distancia realidad en la estimación de la posición final. Esto se debe a la acumulación errores sistemáticos y no sistemáticos. No obstante, ofrece resultados bastante precisos en los primeros tramos de las trayectorias.

6.2 Líneas futuras

El presente trabajo ha abordado diferentes algoritmos de odometría visual para ser implementados en una silla de ruedas autónoma. Este propósito requiere de mayor investigación para ser viable, ya que, como se ha demostrado en las pruebas realizadas, los algoritmos cometen errores notorios. Por lo tanto, se proponen algunas líneas de investigación futuras.

1. **Fusión sensorial**

Se propone implementar algoritmos de fusión de datos, que sean capaces de combinar la información de diferentes odometrías, con el objetivo de mejorar los resultados obtenidos y evitar errores en la estimación de la trayectoria. Así pues, si se cominan los resultados del RTAB-Map junto con la odometría mecánica, se podría evitar la acumulación de errores de esta mediante correcciones.

2. **Mejora del mapa y reconocimiento de objetos**

Esta idea propone mejorar la capacidad de mapeado 3D del entorno para obtener una odometría más precisa dentro de este. Cuanto mejor sea la reconstrucción 3D, mejor será la estimación de la trayectoria de la silla. Asimismo, incrementar el nivel de detalle mejoraría la capacidad de reconocer objetos y obstáculos, que a su vez, permitiría a la silla evitarlos e interactuar con su entorno.

6.3 Conclusions

This work has conducted a study on the performance and efficiency of different visual odometry algorithms implemented in an autonomous wheelchair. The objective of this project is to determine which methods are the most efficient for this specific application. Additionally, a comparison has been made with mechanical odometry, demonstrating its disadvantages compared to visual algorithms.

The testing environment for the experiments was the hallway of the Faculty of Physics and Mathematics at ULL, as it is a closed space where visual odometry is more efficient. Two tests were conducted in this space: a circular path and a linear path, to evaluate the algorithms under different scenarios and assess loop closure efficiency.

As observed in the conducted and simulated tests using ROS and Rviz, visual odometry algorithms are quite reliable and effective despite some errors. The methods that have achieved better results are ORB-SLAM and RTAB-Map, especially the latter, which also has lower computational cost. These algorithms benefit from optimization techniques and error correction, such as loop closure, which are only available in SLAM algorithms. On the other hand, although Viso2 lacks these advantages, the obtained trajectories do not differ significantly from the actual trajectory.

In summary, visual odometry algorithms capable of performing 3D mapping of the environment using SLAM techniques achieve more precise and reliable results. Viso2 has obtained fairly similar results, making it a viable option, although not as effective and unable to construct 3D maps. Finally, mechanical odometry is the technique that deviates the most from reality in estimating the final position. This is due to the accumulation of systematic and non-systematic errors. However, it provides quite accurate results in the initial segments of the trajectories.

Bibliografía

[1] Fariña, B., Toledo, J., Estevez, J. I., & Acosta, L. (2020). Improving robot localization using doppler-based variable sensor covariance calculation. Sensors (Switzerland), 20(8). https://doi.org/10.3390/S20082287

[2] Arnay, R., Hernandez-Aceituno, J., Toledo, J., & Acosta, L. (2018). Laser and Optical Flow Fusion for a Non-Intrusive Obstacle Detection System on an Intelligent Wheelchair. IEEE Sensors Journal, 18(9), 3799-3805. https://doi.org/10.1109/JSEN.2018.2815566

[3] Cinemática diferencial. https://cuentoscuanticos. com/2011/12/15/robótica-estimacion-de-posicion-por-odometria/

[4] Fernandez D, Price A (2004) Visual odometry for an outdoor mobile robot. IEEE Conference on robotics, automation and mechatronics. IEEE, Piscataway, p 816–821, 2004

[5] D. Scaramuzza and F. Fraundorfer. “Visual Odometry [Tutorial],” IEEE Robot. Autom. Mag., vol. 18, no.4 pp. 80-92, Dec. 2011.

[6] D. Kai Li Lim and D. Thomas Bräun. “A review of visual odometry methods and its applications for autonomous driving”, Sep. 2022.

[7] Nourani-Vatani N, Roberts J, Srinivasan MV “Practical visual odometry for car-like vehicles”. IEEE, Piscataway, p 3551–3557, 2009.

[8] Borenstein J, Everett HR, Feng L “Mobile robot positioning-sensors and techniques”. Naval Command, Control and Ocean Surveillance Center RDT and E Division, San Diego, 1997.

[9] H. P. Moravec, “Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover,” 1980.

[10] Helmick DM, Cheng Y, Clouse DS, “Path following using visual odometry for a mars rover in high-slip environments”. Proceedings anonymous aerospace conference on IEEE, vol 2, IEEE, Piscataway, p 772–789, 2004

[11] Takahashi T 2D “Localization of outdoor mobile robots using 3D laser range data. Doctoral dissertation”, Carnegie Mellon University, 2007

[12] Gonzalez R, Rodriguez F, Guzman JL “Control of off-road mobile robots using visual odometry and slip compensation”. Adv Robot 27(11). P 893–906, 2013.

[13] Khalid Yousif, Alireza Bab-Hadiashar, and Reza Hoseinnezhad. “An overview to visual odometry and visual slam”: Applications to mobile robotics. Intelligent Industrial Systems, p 289–311, 2015.

[14] Nicola Krombach, David Droeschel, and Sven Behnke. “Combining Feature-based and Direct Methodsfor Semi-dense Real-time StereoVisual Odometry”.
https://www.researchgate.net/publication/303895464_Combining_Feature-Based_and_Direct_Methods_for_Semi-dense_Real-Time_Stereo_Visual_Odometry

[15] David Nistér, Oleg Naroditsky, and James Bergen. Visual odometry. In Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on, volume 1, pages I–652. IEEE, 2004.

[16] Stewenius H, Engels C, Nistér D. “Recent developments on direct relative orientation”. ISPRS J Photogramm Remote Sens p 284–294, 2006.

[17] A. Howard. Real-time stereo visual odometry for autonomous ground vehicles. In IEEE Int. Conf. on Intelligent Robots and Systems , 2008

[18] E. Rosten and T. Drummond, “Machine Learning for High-Speed Corner Detection,” Springer, Berlin, Heidelberg, p. 430-443, 2006.

[19] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, Introduction to autonomous mobile robots. MIT Press, 2011.

[20] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," Springer, Berlin, Heidelberg, pp. 404-417, 2006.

[21] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski. Orb: An efficient alternative to sift or surf. In ICCV, pages 2564–2571, 2011.

[22] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In ECCV (4), pages 778–792, 2010.

[23] C. Cadena and L. Carlone and H. Carrillo and Y. Latif and D. Scaramuzza and J. Neira and I. Reid and J.J. Leonard, “Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age”, in IEEE Transactions on Robotics 32 (6) pp 1309–1332, 2016

[24] B. Williams, M. Cummins, J. Neira, P. Newman, I. Reid, and J. Tardós, "A comparison of loop closing techniques in monocular SLAM," Rob. Auton. Syst., vol. 57, pp. 1188-1197, 2009.

[25] Xiang Gao, Tao Zhang, Yi Liu, and Qinrui Yan.14 Lectureson Visual SLAM: From Theory to Practice. Publishing Houseof Electronics Industry, 2017.

[26] Visual Odometry vs. Visual SLAM vs. Structure-from-Motion. https://guvencetinkaya.medium.com/visual-odometry-vs-visual-slam-cdda75df592

[27] Balazadegan Sarvrood, Y. Hosseinyalamdary, S., & Gao, Y. (2016). Visual-LiDAR Odometry Aided by Reduced IMU. ISPRS International Journal of Geo-Information, 5(1), 3, 2016.

[28] Cámara estéreo: http://wiki.ros.org/ps4eye

[29] Sensor Velodyne: https://velodynelidar.com/wp-content/uploads/2019/12/97-0038-Rev-N-97-0038-DATASHEETWEBHDL32E_Web.pdf

[30] ROS: http://www.ros.org/

[31] Rviz: http://wiki.ros.org/rviz

[32] Odometría VISO2. http://wiki.ros.org/viso2_ros.

[33] Mathieu Labbé and François Michaud. Online global loop closure detection for large-scale multisession graph-based slam. Intelligent Robots and Systems (IROS 2014), 2014.

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

[35] ORB-SLAM. http://wiki.ros.org/orb_slam2_roste

[36] A-LOAM. https://github.com/HKUST-Aerial-Robotics/A-LOAM

ANEXO I

Códigos

1. Odometría Mecánica.

```
#!/usr/bin/env python

import message_filters
import rospy
from std_msgs.msg import String
from nav_msgs.msg import Odometry
from roboteq_msgs.msg import Feedback
from geometry_msgs.msg import Point, Pose, Quaternion, Twist, Vector3
import math
import tf

recL=-1
recR=-1
odom_ = Point()
odom2_ = Point()
last_time = -1

def constrainAngle(x):
    x = math.fmod(x + math.pi,2*math.pi)
    if (x < 0):
        x += 2*math.pi
    return x - math.pi

def normalize_angle(angle):
     a = math.fmod(math.fmod(angle, 2.0*math.pi) + 2.0*math.pi, 2.0*math.pi)
     if (a > math.pi):
        a = a - 2.0 *math.pi;
     return a

def callback(left,right):
    global recL,recR, ticks_turn_left, ticks_turn_right,wheel_diameter_left,wheel_diameter_right,wheel_track, odom_, odom_pub, odom_broadcaster, odom_pub2, odom2_, last_time

    recLant = recL
    recRant = recR

    if (recL == -1):
        recL = left.measured_position
        recR = right.measured_position
        last_time = rospy.Time(left.header.stamp.secs,left.header.stamp.nsecs).to_nsec()
        return
    else:
        recL = left.measured_position
        recR = right.measured_position
```

```
    time =  rospy.Time(left.header.stamp.secs,left.header.stamp.nsecs).to_nsec()
    incTime = (time  - last_time)/10e8
    last_time = time

    incL=recL - recLant;
    if( incL < (-0x10000/2) ):
        incL += 0x10000;
    elif ( incL > (0x10000/2) ):
        incL -=0x10000;

    incR=recR - recRant;
    if( incR < (-0x10000/2) ):
        incR += 0x10000;
    elif ( incR > (0x10000/2) ):
        incR -=0x10000;

    yawAnt = odom_.z
    incMetrosLeft = (incL/ticks_turn_left)*math.pi*wheel_diameter_left
    incMetrosRight = (incR/ticks_turn_right)*math.pi*wheel_diameter_right
    despCentroEje = (incMetrosRight + incMetrosLeft)/2

    incYaw = ((incMetrosRight - incMetrosLeft) / wheel_track)

    odom_.z = constrainAngle(incYaw + yawAnt)
    xAnt = odom_.x
    yAnt = odom_.y
    yaw = odom_.z
    incx = despCentroEje*math.cos(yaw)
    incy = despCentroEje*math.sin(yaw)

    odom_.x = xAnt + incx
    odom_.y = yAnt + incy

    odom_quat = tf.transformations.quaternion_from_euler(0, 0, yaw)

    current_time = rospy.Time.now()

    #first, we'll publish the transform over tf
#         odom_broadcaster.sendTransform(  (odom_.x,  odom_.y,  0.),      odom_quat,
current_time, "base_link", "odom")

    odom = Odometry()
    odom.header.stamp = current_time
    odom.header.frame_id = "odom"
    odom.pose.pose = Pose(Point(odom_.x, odom_.y, 0.), Quaternion(*odom_quat))

    odom.child_frame_id = "base_link"
    odom.twist.twist = Twist(Vector3(despCentroEje/incTime, despCentroEje/incTime, 0),
Vector3(0, 0, incYaw/(incTime)))
    odom.pose.covariance[0] = 1e-5
    odom.pose.covariance[7] = 1e-5
    odom.pose.covariance[14] = 999999
```

```
    odom.pose.covariance[21] = 999999
    odom.pose.covariance[28] = 999999
    odom.pose.covariance[35] = 1e-4

    odom.twist.covariance[0] = 1e-6
    odom.twist.covariance[7] = 1e-6
    odom.twist.covariance[14] = 999999
    odom.twist.covariance[21] = 999999
    odom.twist.covariance[28] = 999999
    odom.twist.covariance[35] = 1e-4

    odom_pub.publish(odom)

def listener():
    global ticks_turn_left, ticks_turn_right,wheel_diameter_left,wheel_diameter_right,wheel_track, odom_pub, odom_broadcaster, odom_pub2

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.

    odom_.x = 0
    odom_.y = 0
    odom_.z = 0
    odom = Odometry()

    rospy.init_node('odom_reconstruct', anonymous=True)

    ticks_turn_left = rospy.get_param('ticks_turn_left', 8800)
    ticks_turn_right = rospy.get_param('ticks_turn_right', 8800)
    wheel_diameter_left = rospy.get_param('wheel_diameter_left', 0.3227)
    wheel_diameter_right = rospy.get_param('wheel_diameter_right', 0.3220)
    wheel_track = rospy.get_param('wheel_track', 0.60)

    odom_left_sub = message_filters.Subscriber('/perenquen/left/feedback', Feedback)
    odom_right_sub = message_filters.Subscriber('/perenquen/right/feedback', Feedback)

    ts = message_filters.ApproximateTimeSynchronizer([odom_left_sub, odom_right_sub], 10, 0.1)

    ts.registerCallback(callback)
    odom_pub = rospy.Publisher('/perenquen/odom', Odometry, queue_size=10)
#    odom_broadcaster = tf.TransformBroadcaster()

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()
if __name__ == '__main__':
    listener()
```

2. Odometría Viso2

```
<launch>
<param name="use_sim_time" type="bool" value="true"/>
<!-- **********Run the TEST BAG********** -->
<node pkg="rosbag" type="play" name="player" output="screen" args="
/home/jorgesanz/catkin_ws/recorrido_49.bag -r1"/>

<arg name="camera" default="/stereo"/> <!-- The namespace where images are
published -->
<param name="/use_sim_time" value="false"/>

  <!-- Republishes the compressed topic so it can be used to image_proc node -->
  <node name="republish" type="republish" pkg="image_transport" output="screen"
args="compressed in:=$(arg camera)/left/image_raw raw out:=$(arg
camera)/left/image_raw" />
  <node name="republish1" type="republish" pkg="image_transport" output="screen"
args="compressed in:=$(arg camera)/right/image_raw raw out:=$(arg
camera)/right/image_raw" />

<!-- **********Run the ROS package stereo_image_proc********** -->
<group ns="$(arg camera)" >
   <node pkg="stereo_image_proc" type="stereo_image_proc"
name="stereo_image_proc">
   </node>
</group>

<!-- **********Cambia de plano la odometria stereo********** -->
<node pkg="tf2_ros" type="static_transform_publisher" name="ps4eye"
     args="0 0 0 -1.58 0 -1.45 base_link ps4eye_frame" />

<!-- **********Ejecuta el codigo python de la odom mecanica********** -->
<node pkg="viso2_ros" type="prueba2.py" name="prueba2" output="screen" />

<!-- **********Run the viso2_ros package********** -->
<node pkg="viso2_ros" type="stereo_odometer" name="stereo_odometer"
output="screen">
   <remap from="stereo" to="$(arg camera)"/>
   <remap from="image" to="image_rect"/>
   <param name="odom_frame_id" value="odom"/>
   <param name="sensor_frame_id" value="ps4eye_frame"/>
   <param name="base_link_frame_id" value="base_link"/>

</node>
<!-- **********Run the RVIZ********** -->
<arg name="rviz" default="true" />
   <group if="$(arg rviz)">
      <node launch-prefix="nice" pkg="rviz" type="rviz" name="rviz" args="-d $(find
viso2_ros)/rviz_cnf/stereo_mec.rviz" />
   </group>

</launch>
```

3. Odometría RTAB-Map

```
<?xml version="1.0"?>
<launch>

  <!--
    Demo of outdoor stereo mapping.
    From bag:
    $ rosbag record
         /stereo_camera/left/image_raw_throttle/compressed
         /stereo_camera/right/image_raw_throttle/compressed
         /stereo_camera/left/camera_info_throttle
         /stereo_camera/right/camera_info_throttle
         /tf

    $ roslaunch rtabmap demo_stereo_outdoor.launch
    $ rosbag play -.-clock stereo_oudoorA.bag
  -->

  <!-- Choose visualization -->
  <arg name="rviz" default="true" />
  <arg name="local_bundle" default="true" />
  <arg name="stereo_sync" default="false" />
  <arg name="rtabmapviz" default="false" />

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

 <node pkg="tf2_ros" type="static_transform_publisher" name="ps4eye"
    args="0 0 0 -1.58 0 -1.45 base_link ps4eye_frame" />

 <node pkg="tf2_ros" type="static_transform_publisher" name="k"
    args="0 0 0 0 0 0 map odom" />

  <!-- Just to uncompress images for stereo_image_rect -->
  <node name="republish_left"  type="republish" pkg="image_transport"
args="compressed in:=/stereo/left/image_raw raw out:=/stereo/left/image_raw" />
  <node name="republish_right" type="republish" pkg="image_transport"
args="compressed in:=/stereo/right/image_raw raw out:=/stereo/right/image_raw" />

  <!-- Run the ROS package stereo_image_proc for image rectification -->
  <group ns="/stereo" >
    <node pkg="stereo_image_proc" type="stereo_image_proc"
name="stereo_image_proc">
       <remap from="left/image_raw"    to="left/image_raw"/>
       <remap from="left/camera_info"  to="left/camera_info"/>
       <remap from="right/image_raw"   to="right/image_raw"/>
       <remap from="right/camera_info" to="right/camera_info"/>
       <!--param name="disparity_range" value="128"/-->
    </node>

    <node if="$(arg stereo_sync)" pkg="nodelet" type="nodelet" name="stereo_sync"
args="standalone rtabmap_ros/stereo_sync">
```

```
    <remap from="left/image_rect"   to="left/image_rect_color"/>
    <remap from="right/image_rect"   to="right/image_rect"/>
    <remap from="left/camera_info"   to="left/camera_info"/>
    <remap from="right/camera_info"   to="right/camera_info"/>
   </node>
  </group>

  <group ns="rtabmap">

    <!-- Stereo Odometry -->
    <node pkg="rtabmap_ros" type="stereo_odometry" name="stereo_odometry"
output="screen">
       <remap from="left/image_rect"       to="/stereo/left/image_rect"/>
       <remap from="right/image_rect"      to="/stereo/right/image_rect"/>
       <remap from="left/camera_info"      to="/stereo/left/camera_info"/>
       <remap from="right/camera_info"     to="/stereo/right/camera_info"/>
       <remap from="rgbd_image"            to="/stereo/rgbd_image"/>
       <remap from="odom"                  to="/stereo_odometry"/>

       <param name="subscribe_rgbd"  type="bool" value="$(arg stereo_sync)"/>
       <param name="frame_id"        type="string" value="base_link"/>
       <param name="odom_frame_id"   type="string" value="odom"/>

       <param name="Odom/Strategy"      type="string" value="0"/> <!-- 0=Frame-to-
Map, 1=Frame=to=Frame -->
       <param name="Vis/EstimationType" type="string" value="1"/> <!-- 0=3D->3D
1=3D->2D (PnP) -->
       <param name="Vis/MaxDepth"       type="string" value="0"/>
       <param name="Odom/GuessMotion" type="string" value="true"/>
       <param name="Vis/MinInliers"     type="string" value="10"/>
       <param unless="$(arg local_bundle)" name="OdomF2M/BundleAdjustment"
type="string" value="0"/>
       <param name="OdomF2M/MaxSize"    type="string" value="1000"/>
       <param name="GFTT/MinDistance"   type="string" value="10"/>
       <param name="GFTT/QualityLevel"  type="string" value="0.00001"/>
       <param name="GFTT/QualityLevel"  type="string" value="0.00001"/>
     </node>

    <!-- Visual SLAM: args: "delete_db_on_start" and "udebug" -->
    <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen"
args="--delete_db_on_start">
       <param name="frame_id"         type="string" value="base_link"/>
       <param unless="$(arg stereo_sync)" name="subscribe_stereo" type="bool"
value="true"/>
       <param name="subscribe_depth"  type="bool" value="false"/>
       <param name="subscribe_rgbd"      type="bool" value="$(arg stereo_sync)"/>

       <remap from="left/image_rect"   to="/stereo/left/image_rect_color"/>
       <remap from="right/image_rect"  to="/stereo/right/image_rect"/>
       <remap from="left/camera_info"  to="/stereo/left/camera_info"/>
       <remap from="right/camera_info" to="/stereo/right/camera_info"/>
       <remap from="rgbd_image"        to="/stereo/rgbd_image"/>
```

```
        <remap from="odom" to="/stereo_odometry"/>

        <param name="queue_size" type="int" value="100"/>
        <param name="map_negative_poses_ignored" type="bool" value="true"/>

        <!-- RTAB-Map's parameters -->
        <param name="Rtabmap/TimeThr"                type="string" value="700"/>
        <param name="Grid/DepthDecimation"           type="string" value="4"/>
        <param name="Grid/FlatObstacleDetected"      type="string" value="true"/>
        <param name="Kp/MaxDepth"                    type="string" value="0"/>
        <param name="Kp/DetectorStrategy"            type="string" value="6"/>
        <param name="Vis/EstimationType"             type="string" value="1"/>   <!--
0=3D->3D, 1=3D->2D (PnP) -->
        <param name="Vis/MaxDepth"                   type="string" value="0"/>
        <param name="RGBD/CreateOccupancyGrid"       type="string" value="true"/>
      </node>

      <!-- Visualisation RTAB-Map -->
      <node if="$(arg rtabmapviz)" pkg="rtabmap_ros" type="rtabmapviz"
name="rtabmapviz" args="-d $(find rtabmap_ros)/launch/config/rgbd_gui.ini"
output="screen">
        <param unless="$(arg stereo_sync)" name="subscribe_stereo"    type="bool"
value="true"/>
        <param name="subscribe_odom_info" type="bool" value="true"/>
        <param name="subscribe_rgbd"      type="bool" value="$(arg stereo_sync)"/>
        <param name="queue_size"          type="int" value="100"/>
        <param name="frame_id"            type="string" value="base_link"/>

        <remap from="left/image_rect"  to="/stereo/left/image_rect_color"/>
        <remap from="right/image_rect"  to="/stereo/right/image_rect"/>
        <remap from="left/camera_info"  to="/stereo/left/camera_info"/>
        <remap from="right/camera_info" to="/stereo/right/camera_info"/>
        <remap from="rgbd_image"          to="/stereo/rgbd_image"/>
        <remap from="odom_info"         to="odom_info"/>
        <remap from="odom"              to="/stereo_odometry"/>
        <remap from="mapData"           to="mapData"/>
      </node>
    </group>

    <!-- Visualisation RVIZ -->
    <node if="$(arg rviz)" pkg="rviz" type="rviz" name="rviz" args="-d $(find
rtabmap_ros)/launch/config/demo_stereo_outdoor.rviz"/>

</launch>
```

4. Odometria ORB-SLAM

".launch"

```
<launch>

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

 <!-- Republishes the compressed topic so it can be used to image_proc node -->
  <node name="republish" type="republish" pkg="image_transport" output="screen"
args="compressed in:=/stereo/left/image_raw raw out:=/stereo/left/image_raw" />
  <node name="republish1" type="republish" pkg="image_transport" output="screen"
args="compressed in:=/stereo/right/image_raw raw out:=/stereo/right/image_raw" />
<!-- **********Run the ROS package stereo_image_proc********** -->

<node pkg="orb_slam2_ros" type="Pose_to_Odometry2.py" name="odometry"
output="screen"/>

 <node pkg="tf2_ros" type="static_transform_publisher" name="ps4eye"
     args="0 0 0 -1.58 0 -1.45 base_link ps4eye_frame" />

 <!--node pkg="tf2_ros" type="static_transform_publisher" name="ps4eye1"
     args="0 0 0 -1.58 0 -1.45 odom map" /-->

 <node pkg="tf" type="static_transform_publisher" name="aft_odom"
     args="0 0 0 -1.515758 0 0 odom camera_init 100" />

 <node name="stereo_image_proc" pkg="stereo_image_proc"
type="stereo_image_proc" ns="stereo"/>

   <node name="slam" pkg="orb_slam2_ros"
     type="orb_slam2_ros_stereo" output="screen">
     <!-- Camera topics to listen on -->
     <remap from="image_left/image_color_rect" to="/stereo/left/image_rect"/>
     <remap from="image_right/image_color_rect" to="/stereo/right/image_rect"/>
     <remap from="image_left/camera_info" to="/stereo/left/camera_info" />
     <remap from="image_right/camera_info" to="/stereo/right/camera_info" />
     <!-- Settings -->
     <param name="publish_pointcloud" type="bool" value="true" />
     <param name="publish_pose" type="bool" value="true" />
     <param name="localize_only" type="bool" value="false" />
     <param name="reset_map" type="bool" value="false" />

     <!-- Static parameters -->
     <param name="load_map" type="bool" value="false" />
     <param name="map_file" type="string" value="map.bin" />
     <param name="voc_file" type="string" value="$(find
orb_slam2_ros)/orb_slam2/Vocabulary/ORBvoc.txt" />

     <!-- Frames -->
     <param name="pointcloud_frame_id" type="string" value="map" />
     <param name="camera_frame_id" type="string" value="ps4eye_frame" />
```

```
    <!-- Position transform and pose topic msg will be transformed to this frame -->
    <param name="target_frame_id" type="string" value="ps4eye_frame" />

    <!-- Settings -->
    <param name="min_num_kf_in_map" type="int" value="8" />

    <!-- ORB parameters -->
    <param name="/ORBextractor/nFeatures" type="int" value="2000" />
    <param name="/ORBextractor/scaleFactor" type="double" value="1.2" />
    <param name="/ORBextractor/nLevels" type="int" value="8" />
    <param name="/ORBextractor/iniThFAST" type="int" value="12" />
    <param name="/ORBextractor/minThFAST" type="int" value="1" />

    <!-- Camera parameters -->
    <!-- Camera frames per second -->
    <param name="camera_fps" type="int" value="70" />
    <!-- Color order of the images (0: BGR, 1: RGB. It is ignored if images are
grayscale) -->
    <param name="camera_rgb_encoding" type="bool" value="false" />
    <!-- Close/Far threshold. Baseline times. -->
    <param name="ThDepth" type="double" value="50.0" />
    <!-- Depthmap values factor (what pixel value in the depth image corresponds to
1m) -->
    <!--param name="depth_map_factor" type="double" value="1" -->

    <!-- Camera calibration parameters -->
    <!-- If the node should wait for a camera_info topic to take the camera calibration
data -->
    <param name="load_calibration_from_cam" type="bool" value="false" />
     <!-- Camera calibration and distortion parameters (OpenCV) -->
    <param name="camera_fx" type="double" value="430.52" />
    <param name="camera_fy" type="double" value = "429.515" />
    <param name="camera_cx" type="double" value="311.699" />
    <param name="camera_cy" type="double" value="194.045" />
     <!-- Camera calibration and distortion parameters (OpenCV) -->
    <param name="camera_k1" type="double" value="-0.013794" />
    <param name="camera_k2" type="double" value="-0.013489" />
    <param name="camera_p1" type="double" value="-0.001369" />
    <param name="camera_p2" type="double" value="-0.001168" />
    <param name="camera_k3" type="double" value="0.0" />
    <!-- IR projector baseline times fx (aprox.) -->
    <param name="camera_baseline" type="double" value="34.57" />

  </node>

<node type="rviz" name="rviz" pkg="rviz" args="-d $(find
orb_slam2_ros)/ros/prueba.rviz" />
</launch>
```

“.py Pose_to_Odometry”

```
#!/usr/bin/env python

import message_filters
import rospy
from std_msgs.msg import String
from nav_msgs.msg import Odometry
from roboteq_msgs.msg import Feedback
from geometry_msgs.msg import Point, Pose, Quaternion, Twist, Vector3,
PoseStamped
import math
import tf

recL=-1
recR=-1
odom_ = Point()
odom2_ = Point()
last_time = -1

def constrainAngle(x):
    x = math.fmod(x + math.pi,2*math.pi)
    if (x < 0):
        x += 2*math.pi
    return x - math.pi

def normalize_angle(angle):
     a = math.fmod(math.fmod(angle, 2.0*math.pi) + 2.0*math.pi, 2.0*math.pi)
     if (a > math.pi):
                    a = a - 2.0 *math.pi;
     return a

def callback(pose):

    odom = Odometry()
    odom.header.stamp = rospy.Time.now()
    odom.header.frame_id = "odom"
    odom.pose.pose = Pose(Point(pose.pose.position.x, pose.pose.position.y,
pose.pose.position.z), pose.pose.orientation)
    odom.child_frame_id = "base_link"

    odom_pub.publish(odom)

def listener():

    global odom_pub
    rospy.init_node('odom_reconstruct', anonymous=True)

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
```

```
    odom_laser_sub = rospy.Subscriber('/slam/pose',PoseStamped,callback)
    odom_pub = rospy.Publisher('/stereo/odom', Odometry, queue_size=10)

    #odom_broadcaster = tf.TransformBroadcaster()

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

5. Odometría A-LOAM

```
<launch>
    <!--node pkg="loam_velodyne" type="transformMaintenance"
name="transformMaintenance" output="screen">
    </node-->

 param name="scan_line" type="int" value="32" /

    <!-- if 1, do mapping 10 Hz, if 2, do mapping 5 Hz. Suggest to use 1, it will adjust
frequence automaticlly -->
    <param name="mapping_skip_frame" type="int" value="1" />

    <!-- remove too closed points -->
    <param name="minimum_range" type="double" value="0.3"/>

    <param name="mapping_line_resolution" type="double" value="0.2"/>
    <param name="mapping_plane_resolution" type="double" value="0.4"/>

    <node pkg="aloam_velodyne" type="ascanRegistration" name="ascanRegistration"
output="screen" />

    <node pkg="aloam_velodyne" type="alaserOdometry" name="alaserOdometry"
output="screen" />

    <node pkg="aloam_velodyne" type="alaserMapping" name="alaserMapping"
output="screen" />

    <arg name="rviz" default="false" />
    <group if="$(arg rviz)">
        <node launch-prefix="nice" pkg="rviz" type="rviz" name="rviz" args="-d $(find
aloam_velodyne)/rviz_cfg/aloam_velodyne.rviz" />
    </group>

<node pkg="tf" type="static_transform_publisher" name="aft_map"
      args="0 0 0 -1.515758 0 0 odom camera_init 100" />

</launch>
```

6. Simulación Conjunta

```
<?xml version="1.0"?>
<launch>

  <!--
    Demo of outdoor stereo mapping.
    From bag:
    $ rosbag record
         /stereo_camera/left/image_raw_throttle/compressed
         /stereo_camera/right/image_raw_throttle/compressed
         /stereo_camera/left/camera_info_throttle
         /stereo_camera/right/camera_info_throttle
         /tf

    $ roslaunch rtabmap demo_stereo_outdoor.launch
    $ rosbag play -.-clock stereo_oudoorA.bag
  -->

  <!-- Choose visualization -->
  <arg name="rviz" default="true" />
  <arg name="local_bundle" default="true" />
  <arg name="stereo_sync" default="false" />
  <arg name="rtabmapviz" default="false" />

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

 <node pkg="tf2_ros" type="static_transform_publisher" name="ps4eye"
    args="0 0 0 -1.58 0 -1.45 base_link ps4eye_frame" />

 <node pkg="tf2_ros" type="static_transform_publisher" name="k"
    args="0 0 0 0 0 0 map odom" />

  <!-- Just to uncompress images for stereo_image_rect -->
  <node name="republish_left"  type="republish" pkg="image_transport"
args="compressed in:=/stereo/left/image_raw raw out:=/stereo/left/image_raw" />
  <node name="republish_right" type="republish" pkg="image_transport"
args="compressed in:=/stereo/right/image_raw raw out:=/stereo/right/image_raw" />

  <!-- Run the ROS package stereo_image_proc for image rectification -->
  <group ns="/stereo" >
    <node pkg="stereo_image_proc" type="stereo_image_proc"
name="stereo_image_proc">
       <remap from="left/image_raw"    to="left/image_raw"/>
       <remap from="left/camera_info"  to="left/camera_info"/>
       <remap from="right/image_raw"   to="right/image_raw"/>
       <remap from="right/camera_info" to="right/camera_info"/>
       <!--param name="disparity_range" value="128"/-->
    </node>

    <node if="$(arg stereo_sync)" pkg="nodelet" type="nodelet" name="stereo_sync"
args="standalone rtabmap_ros/stereo_sync">
```

```
    <remap from="left/image_rect"   to="left/image_rect_color"/>
    <remap from="right/image_rect"   to="right/image_rect"/>
    <remap from="left/camera_info"   to="left/camera_info"/>
    <remap from="right/camera_info"   to="right/camera_info"/>
   </node>
  </group>

  <group ns="rtabmap">

    <!-- Stereo Odometry -->
    <node pkg="rtabmap_ros" type="stereo_odometry" name="stereo_odometry"
output="screen">
       <remap from="left/image_rect"       to="/stereo/left/image_rect"/>
       <remap from="right/image_rect"      to="/stereo/right/image_rect"/>
       <remap from="left/camera_info"      to="/stereo/left/camera_info"/>
       <remap from="right/camera_info"     to="/stereo/right/camera_info"/>
       <remap from="rgbd_image"            to="/stereo/rgbd_image"/>
       <remap from="odom"                  to="/stereo_odometry"/>

       <param name="subscribe_rgbd"  type="bool" value="$(arg stereo_sync)"/>
       <param name="frame_id"        type="string" value="base_link"/>
       <param name="odom_frame_id"   type="string" value="odom"/>

       <param name="Odom/Strategy"      type="string" value="0"/> <!-- 0=Frame-to-
Map, 1=Frame=to=Frame -->
       <param name="Vis/EstimationType" type="string" value="1"/> <!-- 0=3D->3D
1=3D->2D (PnP) -->
       <param name="Vis/MaxDepth"       type="string" value="0"/>
       <param name="Odom/GuessMotion" type="string" value="true"/>
       <param name="Vis/MinInliers"     type="string" value="10"/>
       <param unless="$(arg local_bundle)" name="OdomF2M/BundleAdjustment"
type="string" value="0"/>
       <param name="OdomF2M/MaxSize"    type="string" value="1000"/>
       <param name="GFTT/MinDistance"   type="string" value="10"/>
       <param name="GFTT/QualityLevel"  type="string" value="0.00001"/>
       <param name="GFTT/QualityLevel"  type="string" value="0.00001"/>
     </node>

    <!-- Visual SLAM: args: "delete_db_on_start" and "udebug" -->
    <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen"
args="--delete_db_on_start">
       <param name="frame_id"        type="string" value="base_link"/>
       <param unless="$(arg stereo_sync)" name="subscribe_stereo" type="bool"
value="true"/>
       <param name="subscribe_depth"  type="bool" value="false"/>
       <param name="subscribe_rgbd"      type="bool" value="$(arg stereo_sync)"/>

       <remap from="left/image_rect"   to="/stereo/left/image_rect_color"/>
       <remap from="right/image_rect"  to="/stereo/right/image_rect"/>
       <remap from="left/camera_info"  to="/stereo/left/camera_info"/>
       <remap from="right/camera_info" to="/stereo/right/camera_info"/>
       <remap from="rgbd_image"        to="/stereo/rgbd_image"/>
```

```
        <remap from="odom" to="/stereo_odometry"/>

        <param name="queue_size" type="int" value="100"/>
        <param name="map_negative_poses_ignored" type="bool" value="true"/>

        <!-- RTAB-Map's parameters -->
        <param name="Rtabmap/TimeThr"                type="string" value="700"/>
        <param name="Grid/DepthDecimation"           type="string" value="4"/>
        <param name="Grid/FlatObstacleDetected"      type="string" value="true"/>
        <param name="Kp/MaxDepth"                    type="string" value="0"/>
        <param name="Kp/DetectorStrategy"            type="string" value="6"/>
        <param name="Vis/EstimationType"             type="string" value="1"/>   <!--
0=3D->3D, 1=3D->2D (PnP) -->
        <param name="Vis/MaxDepth"                   type="string" value="0"/>
        <param name="RGBD/CreateOccupancyGrid"       type="string" value="true"/>
      </node>

      <!-- Visualisation RTAB-Map -->
      <node if="$(arg rtabmapviz)" pkg="rtabmap_ros" type="rtabmapviz"
name="rtabmapviz" args="-d $(find rtabmap_ros)/launch/config/rgbd_gui.ini"
output="screen">
        <param unless="$(arg stereo_sync)" name="subscribe_stereo"    type="bool"
value="true"/>
        <param name="subscribe_odom_info" type="bool" value="true"/>
        <param name="subscribe_rgbd"      type="bool" value="$(arg stereo_sync)"/>
        <param name="queue_size"          type="int" value="100"/>
        <param name="frame_id"            type="string" value="base_link"/>

        <remap from="left/image_rect"  to="/stereo/left/image_rect_color"/>
        <remap from="right/image_rect"  to="/stereo/right/image_rect"/>
        <remap from="left/camera_info"  to="/stereo/left/camera_info"/>
        <remap from="right/camera_info" to="/stereo/right/camera_info"/>
        <remap from="rgbd_image"          to="/stereo/rgbd_image"/>
        <remap from="odom_info"         to="odom_info"/>
        <remap from="odom"              to="/stereo_odometry"/>
        <remap from="mapData"           to="mapData"/>
      </node>
    </group>

<!-- **********Ejecuta el codigo python de la odom mecanica********** -->
<node pkg="viso2_ros" type="prueba2.py" name="prueba2" output="screen" />


<!-- **********Run the VELODYNE********** -->
<!--node pkg="loam_velodyne" type="transformMaintenance"
name="transformMaintenance" output="screen">
    </node-->

    <param name="scan_line" type="int" value="32" />
```

```
   <!-- if 1, do mapping 10 Hz, if 2, do mapping 5 Hz. Suggest to use 1, it will adjust
frequence automaticlly -->
   <param name="mapping_skip_frame" type="int" value="1" />

   <!-- remove too closed points -->
   <param name="minimum_range" type="double" value="0.3"/>


   <param name="mapping_line_resolution" type="double" value="0.2"/>
   <param name="mapping_plane_resolution" type="double" value="0.4"/>

   <node pkg="aloam_velodyne" type="ascanRegistration" name="ascanRegistration"
output="screen" />

   <node pkg="aloam_velodyne" type="alaserOdometry" name="alaserOdometry"
output="screen" />

   <node pkg="aloam_velodyne" type="alaserMapping" name="alaserMapping"
output="screen" />

<!-- **********Publica odom velodyne en odom********** -->
<node pkg="tf2_ros" type="static_transform_publisher" name="aft_map"
     args="0 0 0 -1.515758 0 0 odom camera_init" />

<!--
**********************************************************estereo**********************************
**********-->
<!--param name="use_sim_time" type="bool" value="true"/-->
<!--arg name="camera" default="/stereo"/--> <!-- The namespace where images are
published -->

<!--param name="/use_sim_time" value="false"/-->

  <!-- Republishes the compressed topic so it can be used to image_proc node -->
  <!--node name="republish" type="republish" pkg="image_transport" output="screen"
args="compressed in:=$(arg camera)/left/image_raw raw out:=$(arg
camera)/left/image_raw" /-->
  <!--node name="republish1" type="republish" pkg="image_transport" output="screen"
args="compressed in:=$(arg camera)/right/image_raw raw out:=$(arg
camera)/right/image_raw" /-->


<!-- **********Run the ROS package stereo_image_proc********** -->
<!--group ns="$(arg camera)" >
   <node pkg="stereo_image_proc" type="stereo_image_proc"
name="stereo_image_proc1">
   </node>
</group-->

<!-- **********Run the viso2_ros package********** -->
<node pkg="viso2_ros" type="stereo_odometer" name="stereo_odometer"
output="screen">
```

```
    <!--remap from="stereo" to="$(arg camera)"/-->
    <remap from="image" to="image_rect"/>
    <param name="odom_frame_id" value="odom"/>
    <param name="sensor_frame_id" value="ps4eye_frame"/>
    <param name="base_link_frame_id" value="base_link"/>
    <param name="publish_tf" value = "false"/>

</node>


<!--**********************************OBR SLAM ****************************************-->

<launch>

<node pkg="orb_slam2_ros" type="Pose_to_Odometry2.py" name="odometry"
output="screen"/>

   <node name="slam" pkg="orb_slam2_ros"
      type="orb_slam2_ros_stereo" output="screen">
      <!-- Camera topics to listen on -->
      <remap from="image_left/image_color_rect" to="/stereo/left/image_rect"/>
      <remap from="image_right/image_color_rect" to="/stereo/right/image_rect"/>
      <remap from="image_left/camera_info" to="/stereo/left/camera_info" />
      <remap from="image_right/camera_info" to="/stereo/right/camera_info" />
      <!-- Settings -->
      <param name="publish_pointcloud" type="bool" value="true" />
      <param name="publish_pose" type="bool" value="true" />
      <param name="localize_only" type="bool" value="false" />
      <param name="reset_map" type="bool" value="false" />

      <!-- Static parameters -->
      <param name="load_map" type="bool" value="false" />
      <param name="map_file" type="string" value="map.bin" />
      <param name="voc_file" type="string" value="$(find
orb_slam2_ros)/orb_slam2/Vocabulary/ORBvoc.txt" />

      <!-- Frames -->
      <param name="pointcloud_frame_id" type="string" value="map" />
      <param name="camera_frame_id" type="string" value="ps4eye_frame" />

      <!-- Position transform and pose topic msg will be transformed to this frame -->
      <param name="target_frame_id" type="string" value="ps4eye_frame" />

      <!-- Settings -->
      <param name="min_num_kf_in_map" type="int" value="8" />

      <!-- ORB parameters -->
      <param name="/ORBextractor/nFeatures" type="int" value="2000" />
      <param name="/ORBextractor/scaleFactor" type="double" value="1.2" />
      <param name="/ORBextractor/nLevels" type="int" value="8" />
      <param name="/ORBextractor/iniThFAST" type="int" value="12" />
      <param name="/ORBextractor/minThFAST" type="int" value="1" />
```

```
    <!-- Camera parameters -->
    <!-- Camera frames per second -->
    <param name="camera_fps" type="int" value="70" />
    <!-- Color order of the images (0: BGR, 1: RGB. It is ignored if images are
grayscale) -->
    <param name="camera_rgb_encoding" type="bool" value="false" />
    <!-- Close/Far threshold. Baseline times. -->
    <param name="ThDepth" type="double" value="50.0" />
    <!-- Depthmap values factor (what pixel value in the depth image corresponds to
1m) -->
    <!--param name="depth_map_factor" type="double" value="1" -->

    <!-- Camera calibration parameters -->
    <!-- If the node should wait for a camera_info topic to take the camera calibration
data -->
    <param name="load_calibration_from_cam" type="bool" value="false" />
     <!-- Camera calibration and distortion parameters (OpenCV) -->
    <param name="camera_fx" type="double" value="430.52" />
    <param name="camera_fy" type="double" value = "429.515" />
    <param name="camera_cx" type="double" value="311.699" />
    <param name="camera_cy" type="double" value="194.045" />
     <!-- Camera calibration and distortion parameters (OpenCV) -->
    <param name="camera_k1" type="double" value="-0.013794" />
    <param name="camera_k2" type="double" value="-0.013489" />
    <param name="camera_p1" type="double" value="-0.001369" />
    <param name="camera_p2" type="double" value="-0.001168" />
    <param name="camera_k3" type="double" value="0.0" />
    <!-- IR projector baseline times fx (aprox.) -->
    <param name="camera_baseline" type="double" value="34.57" />

  </node>

  <!-- Visualisation RVIZ -->
  <node if="$(arg rviz)" pkg="rviz" type="rviz" name="rviz" args="-d $(find
rtabmap_ros)/launch/config/demo_stereo_outdoor.rviz"/>

</launch>
```