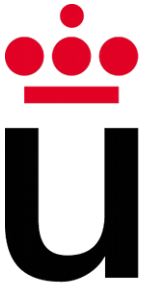


Universidad
Rey Juan Carlos

MÁSTER EN VISIÓN ARTIFICIAL

Curso Académico 2022/2023

Trabajo Fin de Máster

ODOMETRÍA VISUAL TRIDIMENSIONAL EN LA PLATAFORMA EDUCATIVA UNIBOTICS

Autor/a : Pablo Asensio Martínez
Tutor/a : Dr. JoseMaria Cañas Plaza

Trabajo Fin de Máster

Odometría Visual Tridimensional en la Plataforma Educativa Unibotics

Autor/a : Pablo Asensio Martínez
Tutor/a : Dr. JoseMaria Cañas Plaza

La defensa del presente Proyecto Fin de Grado/Máster se realizó el día de de 2023, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Móstoles/Fuenlabrada, a de de 2023

Thor, siempre estarás en mi corazón.

Agradecimientos

Quiero expresar mi más sincero agradecimiento a mi familia, especialmente a mi madre Teresa, por su apoyo y ayuda incondicional durante todo este proyecto. También quiero recordar a mi querido perro y amigo Thor, recientemente fallecido, que siempre estuvo a mi lado y que me enseñó tanto. También quiero agradecer a mi tutor JoseMaria por su guía y por cada uno de sus valiosos consejos y enseñanzas. Sin todo su apoyo y dedicación, este proyecto no habría sido posible. Gracias a todos por hacer que este camino fuera más llevadero y por ayudarme a alcanzar mis metas.

Resumen

El proyecto aborda la creación de un ejercicio de odometría visual tridimensional en la plataforma educativa Unibotics con el objetivo de proporcionar a los estudiantes una herramienta práctica para aprender y poner en práctica este importante concepto en el campo de la robótica, la autolocalización visual.

Para llevar a cabo esta tarea se desarrolla una solución de referencia donde se utilizan *datasets* de odometría visual como KITTI sobre los que se prueba el algoritmo de referencia de odometría visual tridimensional. Para realizar esta *solución de referencia* se realiza una interfaz web con la que visualizar la trayectoria obtenida por el algoritmo de odometría visual frente a la trayectoria verdadera, obteniendo unas métricas de error a la par. Seguidamente esta solución de referencia se adapta para poder integrarla en la plataforma educativa Unibotics, donde se añade un *auto-evaluador* al algoritmo que programe el estudiante.

Summary

The project addresses the creation of a three-dimensional visual odometry exercise in the educational platform Unibotics, aiming to provide students with a practical tool to learn and apply this important concept in the field of robotics, visual self-localization.

To carry out this task, a reference solution is developed, using visual odometry datasets such as KITTI, on which the reference algorithm for three-dimensional visual odometry is tested. To implement this reference solution, a web interface is created to visualize the trajectory obtained by the visual odometry algorithm compared to the ground truth trajectory, while obtaining error metrics. Subsequently, this reference solution is adapted to integrate it into the Unibotics educational platform, where an auto-evaluator is added to the algorithm programmed by the student.

Índice general

Índice de figuras

List of Listings

Capítulo 1

Introducción

En este primer capítulo se propone dar una visión general del contexto en que se encuadra el trabajo fin de máster, que es la visión artificial en el ámbito de las plataformas educativas en línea. Dentro de ésta se abordará el problema al que vamos a hacer frente: creación de un ejercicio de autolocalización visual, o dicho de otro modo, la estimación de la posición y orientación 3D de una cámara haciendo uso de algoritmos de odometría visual. Concretamente se implementará un algoritmo de odometría visual 3D como solución de referencia para el ejercicio dentro de la plataforma educativa Unibotics.

1.1 Visión Artificial

La visión artificial es un campo de la inteligencia artificial que pretende obtener información del mundo a partir de una o varias imágenes, que normalmente vienen dadas de forma de matriz numérica. La información relevante que se puede obtener a partir de las imágenes puede ser el reconocimiento de objetos, la recreación en 3D de la escena que se observa, el seguimiento de un objeto, etc.

El inicio de la visión artificial se produjo en 1961 por parte de Larry Roberts, quien creó un programa que podía ver una estructura de bloques, analizar su contenido y reproducirla desde otra perspectiva, utilizando para ello una cámara y procesando la imagen desde un ordenador. Sin embargo, para obtener el resultado las condiciones de la prueba estaban muy controladas. Otros muchos científicos también han tratado de solucionar el problema de conectar una cámara a un ordenador y hacer que este describa lo que ve. Finalmente, los científicos de la época se dieron cuenta de que esta tarea no era sencilla de realizar, por lo que se abrió un amplio campo de investigación, que tomó el nombre de visión artificial.

En este campo se persigue, por ejemplo, que el ordenador sea capaz de reconocer en

una imagen distintos objetos al igual que los humanos lo hacemos con nuestra visión. Se ha demostrado que este problema es muy complejo y que algo que para nosotros resulta automático puede que se tarde mucho tiempo en que lo resuelva, con la misma robustez y versatilidad, una máquina. Por otra parte, a pesar del alto precio computacional que se paga por utilizar cámaras como sensor, si se consigue analizar correctamente la imagen, es posible extraer *mucha* información de ella que no podría obtenerse con otro tipo de sensores.

En los años noventa empezaron a aparecer los primeros ordenadores capaces de procesar las imágenes lo suficientemente rápido. Además se comenzó a dividir los posibles problemas de la visión artificial en otros más específicos.

A continuación se presentan algunas aplicaciones y escenarios de aplicación, no solo en el área académica sino también en la vida cotidiana, que hacen uso de los descubrimientos y avances logrados por diversos investigadores en visión artificial.

- Robótica. Unos de los procesos más complejos que tiene que realizar un robot es interpretar el mundo a su alrededor a través de la información que adquiere mediante los sensores, como puede ser una cámara de vídeo. Esta información puede usarse para la localización o reconocimiento de objetos o incluso el seguimiento de los mismos. Los coches autónomos de Tesla o las aspiradoras robóticas de gama alta son ejemplos de robots que usan visión para comprender la escena, detectar objetos de interés o autolocalizarse.

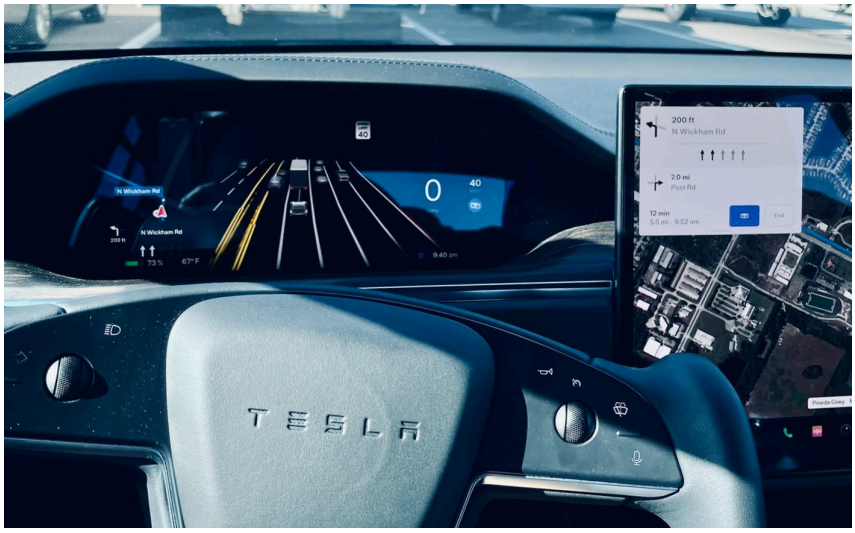


Figura 1.1: Beta de la conducción autónoma total de Tesla

- Videojuegos. Este sector ha contribuido notablemente al desarrollo de la visión artificial. Ejemplos claros de su uso son el dispositivo Kinect desarrollado por Microsoft (cámara RGBD), y Eye Toy, desarrollado por PlayStation para el reconocimiento de los gestos realizados por los jugadores. En la nueva generación de consolas de video-

juegos se está avanzando en mejorar la interacción jugador-consola usando este tipo de cámaras.

- Medicina. Uno de los objetivos de la visión artificial, en este contexto, es el tratamiento y análisis de imágenes, detección de patrones y reconstrucción 3D para ayudar al correcto diagnóstico por parte del especialista clínico. Aplicaciones comunes aquí son la detección y caracterización automática de tumores, mejora de la imagen de microscopio análisis hiperespectral para extraer la composición de los tejidos orgánicos contenidos en una imagen.

- Biometría. La biometría estudia los métodos automáticos para el reconocimiento único de humanos basado en uno o más rasgos conductuales o rasgos físicos intrínsecos. Dentro de estos sistemas se encuentran los sistemas de reconocimiento facial. Estos son muy usados en seguridad, como ejemplo de ello se encuentra el control de fronteras en aeropuertos, el sistema de vigilancia de la ciudad de Londres que realiza un reconocimiento facial mediante cámaras distribuidas por la ciudad; o el reconocimiento de caras y posterior emborronado automático de caras de Google Street View.

Figura 1.2: Control de frontera en el aeropuerto

1.2 Autolocalización Visual

Dentro del ámbito de la visión artificial, se presenta un desafío crucial conocido como autolocalización visual. Este problema implica la tarea de determinar en todo momento la

posición 3D de la cámara únicamente a partir de las imágenes que ha capturado. Dado el potencial de aplicaciones que esto puede ofrecer, resulta un reto sumamente relevante en el campo de la robótica.

Por ejemplo, esta técnica se plantea en los sistemas de navegación automáticos náuticos, terrestres y aéreos. Actualmente numerosas empresas están invirtiendo en este tipo de sistemas en el que apuestan por una navegación total o parcialmente autónoma, un ejemplo puede ser Roomba.

La autolocalización visual es una técnica que permite a aplicaciones de realidad aumentada, que es el término que se usa para definir una visión directa o indirecta de un entorno físico del mundo real, combinar el entorno real con elementos virtuales generados por ordenador para la creación de una realidad mixta en tiempo real. En la imagen 1.3 se puede observar un pokemon que parece que se encuentra en la calle, de frente de quien toma la fotografía, pero realmente ha sido el uso de la realidad aumentada lo que ha permitido una integración fidedigna del pokemon con el entorno en el contexto de la imagen. Asimismo, los sistemas de realidad virtual, como las gafas Value u Oculus, han sido una de las tecnologías que han impulsado el desarrollo de la autolocalización visual. Gracias a esta técnica, los usuarios pueden experimentar una inmersión aún más completa en el mundo virtual, ya que el sistema es capaz de determinar la posición de la cámara en todo momento, permitiendo una interacción más realista y precisa con los objetos virtuales en el espacio físico.

Figura 1.3: Realidad aumentada del videojuego para móvil Pokemon Go

Las técnicas de autolocalización han suscitado gran interés por los investigadores en los últimos años. El problema ha sido abordado por dos comunidades distintas. Por un lado la de visión artificial que denominó al problema como *structure from motion* (SfM), donde la información es procesada por lotes, capaz de representar un objeto 2D a 3D con solo unas cuantas imágenes desde diferentes puntos de vista. Y por otro lado la comunidad robótica denominó al problema SLAM (*Simultaneous Localization and Mapping*) que trata de

resolver el problema de una manera más ágil adaptando el funcionamiento de los sistemas en tiempo real.

Algunos de los conceptos más interesantes de conocer en la autolocalización visual son:

- Localización absoluta. Esta técnica pretende situar la cámara en una posición cuyo sistema de referencia sea común para todos, sin partir del conocimiento de posiciones previas. Podría usarse el sistema de referencia GPS, o el inicio de la escena, por ejemplo.
- Localización incremental. A diferencia de la localización absoluta, esta técnica hace uso de un sistema de referencia horizonte local, siendo origen de referencia la del fotograma inmediatamente anterior. Haciendo la suma de estos incrementos se podría calcular la posición absoluta si se conoce la posición inicial.
- Error acumulativo. Es un tipo de error que está en la naturaleza de este problema, ya que al haber pequeños errores de cálculo numérico al computar las matrices de rotación en cada iteración, a la hora de actualizar la pose absoluta, se va acumulando este pequeño error.
- Localización desde un mapa conocido. Una técnica que permite, conociendo el escenario de trabajo, estimar de una mejor forma, la posición de la cámara.
- Localización sin mapa. Aquella técnica de localización que no hace uso de información conocida previa del mapa a priori.
- Cierre de bucle. Es un mecanismo que se suele utilizar para reajustar los cálculos y reducir en mayor medida el error acumulativo. Cuando se pasa por una zona ya conocida, se compara la estimación 3D las dos veces, sabiendo que es la misma zona, y por ello deberían ser las mismas coordenadas.

1.2.1 Structure from Motion (SfM)

Dentro de la visión artificial el *Structure from Motion* (SfM) es la línea de investigación que toma como entrada únicamente un conjunto de imágenes, y pretende conocer de manera totalmente automática la estructura 3D de la escena vista y las ubicaciones de las cámaras desde donde esas imágenes fueron captadas. El SfM ha llegado a un estado de madurez donde los algoritmos tienen aplicación comercial.

El SfM surge en la segunda mitad del siglo XIX, denominado fotogrametría, que tuvo como objetivo extraer información geométrica de las imágenes a partir de un conjunto de características manualmente identificadas por el usuario. La fotogrametría hace uso de técnicas de optimización no lineales, como el ajuste de haces, para reducir al mínimo error de

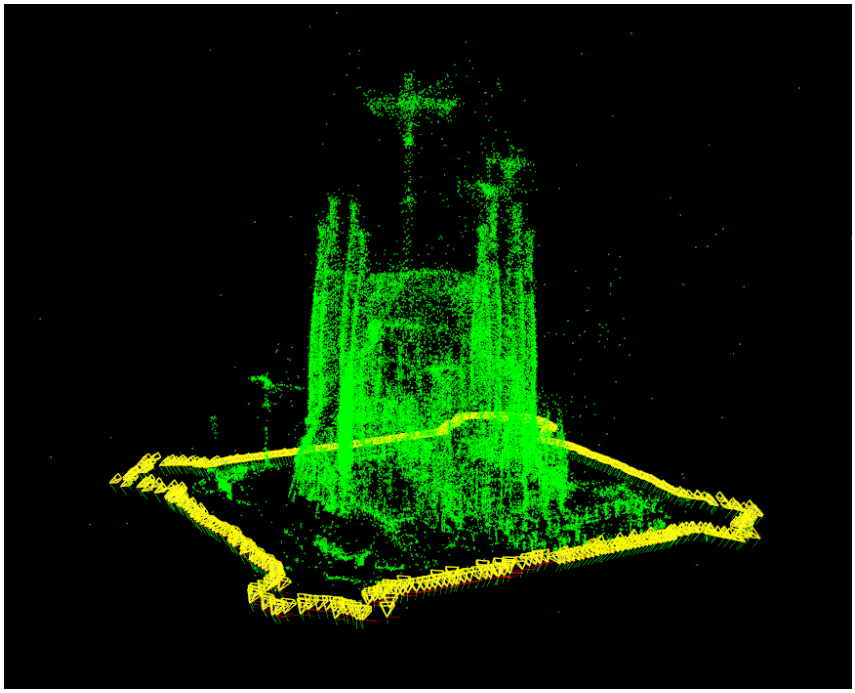


Figura 1.4: Structure from Motion: La sagrada familia

retroproyección. El problema abordado por la comunidad de visión artificial ha sido en su mayoría lograr la completa automatización del proceso. Esto provocó avances en tres aspectos: en primer lugar, restricciones impuestas al movimiento bajo el supuesto de rigidez de la escena; en segundo lugar, la detección de características y descriptores [Canny, 1986], [Harris, Stephens y col., 1988]; y por último, la eliminación de espúreos.

Dadas múltiples vistas, un punto tridimensional puede ser reconstruido mediante triangulación. Un prerrequisito importante es determinar la calibración de la cámara. Se puede representar por una matriz de proyección. La teoría geométrica de SfM permite calcular las matrices de proyección y los puntos 3D utilizando solo correspondencia entre los puntos de cada imagen. Para mejorar el rendimiento del SfM se puede aprovechar el conocimiento sobre la escena con el fin de reducir el número de grados de libertad. Por ejemplo, las restricciones que imponen el paralelismo y la coplanaridad, pueden utilizarse para reconstruir formas geométricas simples como líneas y polígonos planos, a partir de sus posiciones proyectas en vistas simples.

1.2.2 Visual SLAM

La cuestión conocida como *Simultaneous Localization and Mapping* (SLAM) busca resolver los problemas que plantea colocar un robot móvil en un entorno y una posición desconocidas, y que él mismo se encuentre, sea capaz de construir incrementalmente un mapa de su entorno consistente y a la vez utilizar dicho mapa para determinar su propia loca-

lización. Cuando el sensor principal es una cámara, por ejemplo en drones, se denomina Visual SLAM.

La solución a este problema junto con un mecanismo de navegación haría que el sistema se encuentre con la capacidad para saber a dónde desplazarse, ser capaz de encontrar obstáculos y reaccionar ante ellos de manera inteligente. Esto consigue hacer sistemas de robots completamente autónomos.

La resolución al problema SLAM visual ha suscitado un gran interés en el campo de la robótica y se han propuesto muchas técnicas y algoritmos para darle solución, como es el caso del artículo de Durrant-Whyte y Bailey [7]. Y aunque algunas de ellas han obtenido buenos resultados, en la práctica siguen surgiendo problemas a la hora de buscar el método más rápido o el que genere un mejor resultado con menos índice de fallo. La búsqueda de algoritmos y métodos que resuelvan completamente estos problemas sigue siendo una tarea abierta.

Uno de los trabajos fundacionales más importantes en el ámbito es el de monoSLAM de Davison [1] (Andrew J. Davison y Stasse, 2007)[6] que propone resolver este problema con una única cámara RGB como sensor y realizar el mapeado y la localización simultáneamente. El algoritmo propuesto por Davison utiliza un filtro extendido de Kalman para estimar la posición y la orientación de la cámara, así como la posición de una serie de puntos en el espacio 3D. Para determinar la posición inicial de la cámara es necesario a priori dotar de información sobre la posición 3D de por lo menos 3 puntos. Después el algoritmo es capaz de situar la cámara continuamente en el espacio tridimensional y de generar nuevos puntos para ampliar el mapa y servir como apoyo a la propia localización de la cámara. En la Figura 1.8 se pueden ver unas capturas de pantalla sobre uno de los experimentos realizados.

Es importante destacar también la trascendencia que ha tenido el trabajo PTAM (Klein y Murray, 2007) [13] que viene a solucionar uno de los principales problemas que tienen los algoritmos monoSLAM; el tiempo de cómputo, ya que aumenta exponencialmente con el número de puntos del mapa (Figura 1.9). Para ello se aborda el problema separando el mapeado de la localización, de tal modo que solo la localización deba funcionar en tiempo real, dejando así que el mapeado trabaje de una manera asíncrona en segundo plano. PTAM hace uso de *keyframes*, es decir, fotogramas clave que se utilizan tanto para la localización como para el mapeado y también de una técnica de optimización mediante ajuste de haces, como en SfM.

[1] http://www.doc.ic.ac.uk/~ajd/

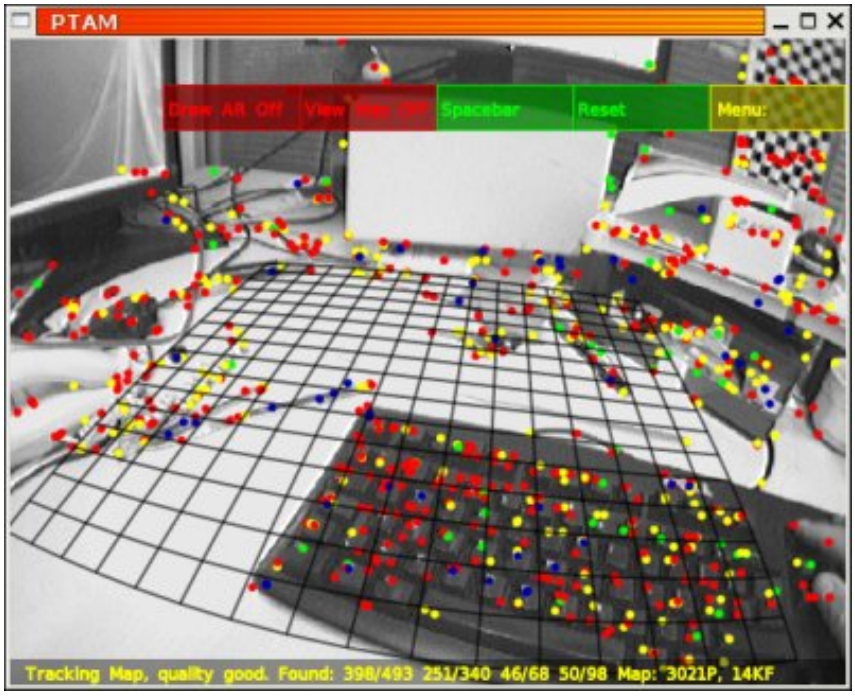


Figura 1.5: PTAM: Funcionamiento sobre un escritorio

1.2.3 Odometría Visual

Dentro de las familias de técnicas pertenecientes a Visual SLAM se encuentra la odometría visual, que es la que abordaremos en este trabajo. Consiste en la estimación del movimiento 3D *incremental* de la cámara en tiempo real. Es decir, el cálculo de la rotación y traslación tridimensionales de la cámara a partir de imágenes consecutivas. Se trata de una técnica incremental ya que se basa en la posición anterior para calcular la nueva.

En este tipo de algoritmos se suelen utilizar técnicas de extracción de puntos de interés, cálculos de descriptores y algoritmos para el emparejamiento. Normalmente el proceso es: una vez calculados los puntos emparejados se calcula la matriz fundamental o esencial y descomponerlas mediante SVD para obtener la matriz de rotación y traslación (RT). Una vez que se conocen estas marices de rotación y traslación, se puede calcular mediante un algoritmo la posición que ocupa nuestra cámara en el espacio.

Dentro de la odometría visual podemos diferenciar diferentes tipos:

- Monocular y estéreo. Según la configuración de la cámara, la odometría visual se puede clasificar como odometría visual monocular (cámara única), odometría visual estéreo (dos cámaras en configuración estéreo).
- Método directo y basado en características. La información visual tradicional de la odometría visual se obtiene mediante el *método basado en características*, que extrae algunos puntos característicos de la imagen y los sigue (*tracking*) en la secuencia

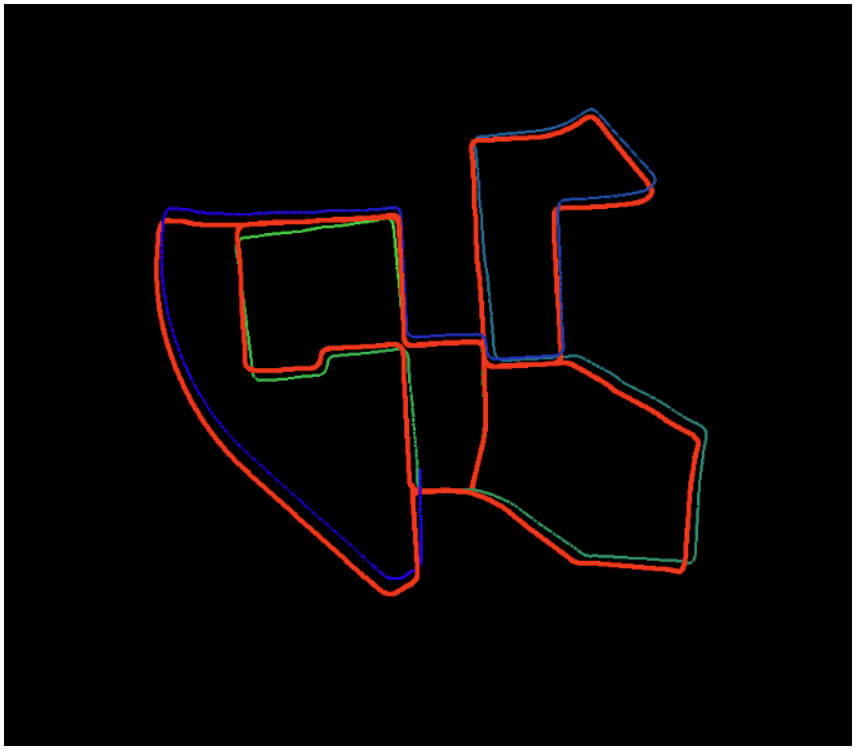


Figura 1.6: Odometría visual 2D. En rojo, la verdad absoluta, frente al resultado de un algoritmo sobre un escenario del dataset de KITTI

de imágenes. Los desarrollos recientes en la investigación en este campo proporcionaron una alternativa, llamada *método directo*, que utiliza la intensidad de todos los píxeles en la secuencia de imágenes directamente como entrada visual. También hay métodos híbridos.

- Odometría inercial visual. Si se utiliza una unidad de medida inercial (IMU) dentro del sistema, esto denomina comúnmente *odometría inercial visual* (VIO).

1.3 Introducción a la Plataforma Unibotics

En los últimos años el uso de plataformas web educativas ha ido incrementando debido a la pandemia del COVID-19, que ha sido un factor por el cual tanto el trabajo como el estudio a distancia ha aumentado considerablemente. Como consecuencia directa el uso de estas plataformas *online* es más demandado. En el área robótica esto no es una excepción. Actualmente se está haciendo especial hincapié en el desarrollo de plataformas *online* que permitan el aprendizaje de la programación de robots. Y aunque Unibotics no nació a causa de la pandemia, sí es una de las pioneras.

Unibotics nace como extensión natural de *Robotics Academy*[5]. Es un entorno docente de robótica universitaria. Este entorno tiene una orientación muy práctica en cuanto al

aprendizaje de la programación de la inteligencia de los robots, ya que utiliza como editor del código fuente el navegador web, el cual también es el interfaz gráfico de la ejecución. La plataforma posee una numerosa colección de ejercicios variados que abarcan muchas de las aplicaciones robóticas que han surgido recientemente: drones, robots aspiradores, coches autónomos, asistente de aparcamiento, control de robots móviles, etc. La diferencia entre Robotics Academy[2] y Unibotics[3] es la forma en la que se ejecutan los ejercicios. La primera es completamente en local, mientras la segunda hace uso de la red para completar los ejercicios.

Figura 1.7: Logo de Unibotics

Toda esta colección de ejercicios son de código abierto lo que proporciona la posibilidad de compartir, modificar y estudiar el código fuente, además de colaborar entre usuarios. Unibotics es un entorno multiplataforma pudiendo utilizarse desde sistemas operativos tales como Linux, Windows y MacOS. La plataforma hace uso del simulador Gazebo y el lenguaje principal con el que se programa es interpretado y multiplataforma, Python.

Algunos de los ejercicios que podemos encontrar en Unibotics relacionados con visión artificial son los siguientes:

- Follow line. El objetivo de *Follow line* es realizar un control reactivo PID que gobierne un coche autónomo para que sea capaz de seguir la línea pintada en el circuito de carreras. Utilizando como sensor exclusivamente la cámara. Pincha aquí para ver el vídeo.
- 3D Reconstruction. En este ejercicio se busca reconstruir una escena 3D a partir de un par estéreo.
- Color Filter. Sirve de introducción al procesamiento de imágenes con OpenCV. Persigue el desarrollo un filtro de color para segmentar algún objeto en la imagen y rastrearlo. Pincha aquí para ver el vídeo.
- Optical Flow Teleop. En este ejercicio se pretende desarrollar un algoritmo de flujo óptico para teleoperar el robot utilizando las imágenes obtenidas de una cámara web. Pincha aquí para ver el vídeo.

[2] http://jderobot.github.io/RoboticsAcademy/
[3] https://unibotics.org/

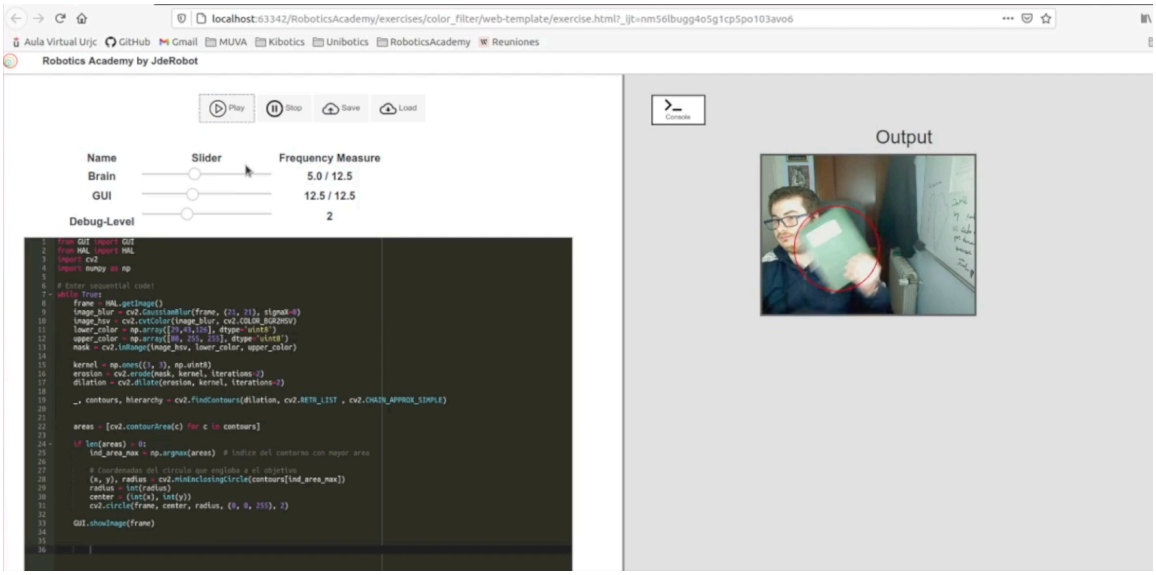


Figura 1.8: Ejercicio *Color Filter* en la plataforma *Unibotics*

- Digit Classification. Aquí el alumnos entrena un modelo de aprendizaje profundo para clasificar números. A demás, el modelo entrenado debe cumplir con las especificaciones de entrada y salida descritas en la documentación.

- Human Detection. Este es un ejercicio de detección de personas usando *deep learning* que incluye la evaluación del rendimiento y la visualización del modelo.

1.4 Estructura de la Memoria

En qwesta sección se describe la organización del resto del documento y qué contenidos se van a encontrar en cada capítulo. La estructura de la memoria se organiza de la siguiente manera:

- En este primer capítulo se hace una breve introducción al proyecto, y en el segundo se describen los objetivos del mismo y se refleja la planificación temporal.

- En el siguiente capítulo se describen el estado del arte así como algunos de los algoritmos utilizados en el desarrollo de este TFM (Capítulo 3).

- En el capítulo 4 se explica una solución de referencia para un problema, y se realiza una evaluación experimental de su rendimiento y precisión.

- En el capítulo 5 se describe la integración del ejercicio de odometría visual 3D en la plataforma educativa Unibotics, destacando su infraestructura.

- Por último, se presentan las conclusiones del proyecto así como los trabajos futuros que podrían derivarse de éste (Capítulo 6).

Capítulo 2

Objetivos y Metodología

El objetivo general de este trabajo es desarrollar un ejercicio educativo de visión artificial en la plataforma Unibotics que permita a los estudiantes aprender y poner en práctica los conceptos de odometría visual tridimensional mediante la creación y prueba de su propio algoritmo. Ofrecerá al estudiante retroalimentación inmediata sobre su rendimiento, mejorando de esta forma su comprensión y habilidades en el tema.

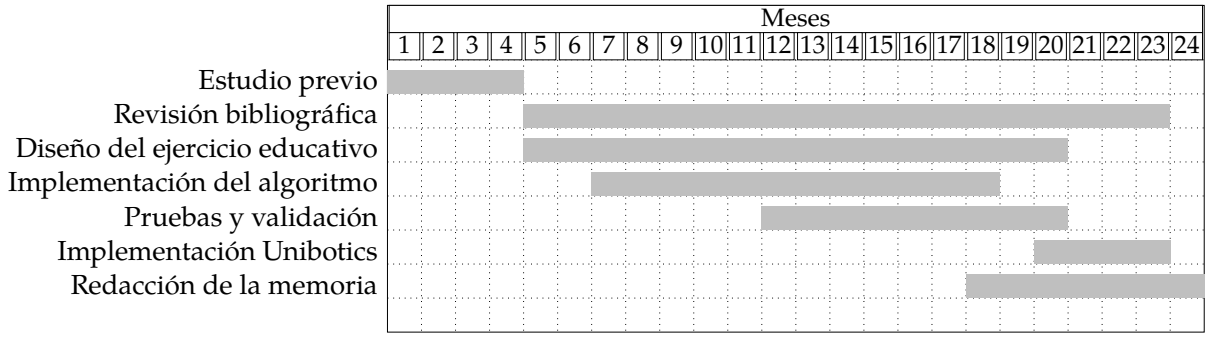
2.1 Objetivos Específicos

El objetivo general se ha articulado en tres subobjetivos parciales:

1. Diseñar un ejercicio educativo que permita a los estudiantes crear y probar su propio algoritmo de odometría visual tridimensional en la plataforma Unibotics.
2. Proporcionar una interfaz visual para que el estudiante pueda ver los resultados de su algoritmo en tiempo real.
3. Implementar un sistema de puntuación para medir el rendimiento del algoritmo desarrollado por el estudiante.

2.2 Planificación Temporal

Esta sección proporciona una visión general de los tiempos previstos para la realización del trabajo. La planificación temporal se representa mediante un diagrama de Gantt, el cual ilustra las diferentes tareas y subtareas que se han llevado a cabo y su duración.



Este diagrama de Gantt representa un proyecto que se ha desarrollado en un periodo de 24 meses. El periodo transcurre entre y enero de 2021 y febrero de 2023. El proyecto comienza con un estudio previo que se lleva a cabo en el primer mes y el cuarto mes. A continuación, se realiza una revisión bibliográfica, diseño del ejercicio educativo y implementación del algoritmo, que se extienden desde el quinto mes hasta el 23 mes.

Durante el periodo de la revisión bibliográfica y diseño del ejercicio educativo, se lleva a cabo también la implementación del algoritmo, que se extiende desde el décimo mes hasta el 18 mes. Durante el 12 y 20 mes, se realizan pruebas y validación del ejercicio educativo.

A partir del 20 mes se procede a la implementación en plataforma educativa Unibotics, y termina en el 23 mes. Mientras se realiza esta tarea, se realiza también la redacción de la memoria, que se extiende desde el 18 mes hasta el 24 mes.

En cuanto al desarrollo temporal del proyecto, se ha llevado a cabo en un periodo aproximado de dos años. Durante este tiempo, se han producido cambios en el tema trabajado y en la planificación original debido a diferentes circunstancias. Además, el proyecto se ha desarrollado en paralelo a otras obligaciones, como el cuidado de familiares dependientes, lo que ha limitado el tiempo disponible para dedicar al trabajo. En total, se ha podido destinar alrededor de 2 horas a la semana, y no todas las semanas, para el desarrollo de este trabajo. A pesar de estos desafíos, se ha logrado cumplir con los objetivos y metas establecidos.

Capítulo 3

Estado del Arte

En este capítulo se presentarán algoritmos de odometría visual, así como técnicas para evaluar su efectividad. Además, se hablará de plataformas web educativas sobre robótica, similares a Unibotics. En ellas es frecuente que se utilicen robots y drones equipados con visión que han de procesar sus imágenes para extraer información útil sobre el entorno o sobre su posición en él.

En concreto, se explicarán los diferentes tipos de algoritmos de odometría visual, así como algunos algoritmos, se hablará de los *datasets* utilizados y se presentarán métodos para evaluar su rendimiento.

3.1 Tipos de Algoritmos de Odometría Visual

La estimación de la posición de un robot móvil puede llevarse a cabo de diversas maneras, una de ellas es mediante la odometría basada en la visión. Esta técnica se puede aplicar de tres formas diferentes: el enfoque basado en características, el basado en apariencia y el híbrido.

3.1.1 Enfoque Basado en Características

Este enfoque utiliza características específicas en el entorno, como puntos o líneas, para estimar la posición del robot. Estas características son seleccionadas porque son fáciles de detectar en la imagen y su posición es conocida con bastante precisión.

Para llevar a cabo el seguimiento de estas características se pueden utilizar diversos

algoritmos de detección y seguimiento. Una vez que se han detectado y seguido en el flujo de video las características, se pueden utilizar técnicas de correlación para determinar el desplazamiento del robot en relación con ellas.

El enfoque basado en características tiene varias ventajas. En primer lugar, es relativamente fácil de implementar y requiere poca potencia de procesamiento. Además, es robusto frente a cambios en la iluminación y en la apariencia del entorno. Sin embargo, tiene algunas desventajas. Una de ellas es que solo es posible seguir un número limitado de características, lo que puede dificultar la estimación de la posición del robot en entornos con pocas características distintivas. Además, la precisión de la estimación de la posición puede ser limitada por la precisión de la detección y seguimiento de las características.

Un trabajo de investigación interesante relacionado con la odometría visual sobre el enfoque basado en características es *ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras* de Mur-Artal et al. (2017). En este trabajo, los autores presentan ORB-SLAM2, un sistema de localización y mapeo (SLAM) basado en características para cámaras monoculares, estereos y RGB-D. ORB-SLAM2 utiliza el algoritmo ORB (Oriented FAST and Rotated BRIEF) para detectar y seguir características en las imágenes capturadas por la cámara. Luego, utiliza estas características para estimar la posición y orientación de la cámara y construir un mapa de la escena. Los autores demuestran que ORB-SLAM2 es capaz de realizar el SLAM de forma precisa y en tiempo real en diversos entornos y con diferentes tipos de cámaras.

3.1.2 Enfoque Basado en Apariencia

Este enfoque utiliza la apariencia general de las imágenes capturadas por la cámara del robot para estimar su posición. En lugar de depender de características específicas, este enfoque utiliza la información *de toda* la imagen para estimar el desplazamiento del robot.

Para llevar a cabo esta técnica se pueden utilizar diversos algoritmos de correlación o de aprendizaje automático. Una vez que se ha comparado la apariencia de la imagen actual con la apariencia de la imagen anterior, se puede utilizar una transformación geométrica para determinar el desplazamiento del robot. Las dos imágenes han de ser próximas en el tiempo.

El enfoque basado en apariencia tiene varias ventajas. En primer lugar, es capaz de funcionar en entornos con pocas características distintivas, ya que utiliza toda la información de la imagen. Además, puede ser más preciso que el enfoque basado en características, ya que utiliza una mayor cantidad de información. Sin embargo, tiene algunas desventajas. Una de ellas es que puede ser más difícil de implementar y requerir más potencia de procesamiento. Además, puede ser menos robusto frente a cambios en la iluminación y en la apariencia del entorno.

Un trabajo de investigación interesante sobre el enfoque basado en apariencia es *DeepVO: Towards End-to-End Visual Odometry with Deep Recurrent Convolutional Neural Networks* de Wang et al. (2017). En este trabajo, los autores presentan una red neuronal recurrente con redes neuronales convolucionales (CNNs) para realizar la odometría visual de forma *extremo a extremo*, utilizando únicamente imágenes de una cámara monocular como entrada. La red neuronal recurrente se entrena para predecir el desplazamiento del robot entre dos imágenes consecutivas y utiliza una arquitectura de *encoder-decoder* para procesar las imágenes. Los autores demuestran que su enfoque es capaz de realizar la odometría visual de forma precisa en diversos entornos y supera a otros métodos basados en apariencia en términos de precisión y robustez.

3.1.3 Enfoque Híbrido

Este enfoque combina tanto el enfoque basado en características como el enfoque basado en la apariencia para obtener una estimación más precisa de la posición del robot.

Para llevar a cabo este enfoque se pueden utilizar diversos algoritmos de detección y seguimiento de características, así como técnicas de correlación o de aprendizaje automático para comparar la apariencia de la imagen actual con la apariencia de la imagen anterior. Una vez que se han detectado y seguido las características y se ha comparado la apariencia de la imagen, se pueden utilizar técnicas de fusión para combinar ambas fuentes de información y obtener así una estimación más precisa de la posición del robot.

El enfoque híbrido tiene varias ventajas. En primer lugar, combina la robustez del enfoque basado en características con la precisión del enfoque basado en apariencia, lo que permite obtener mejor una estimación de la posición del robot. Además, es capaz de funcionar en entornos con pocas características distintivas, ya que utiliza toda la información de la imagen. Sin embargo, también tiene algunas desventajas. Una de ellas es que puede ser más difícil de implementar y requerir más potencia de procesamiento que el enfoque basado en características. Además, puede ser menos robusto frente a cambios en la iluminación y en la apariencia del entorno que el enfoque basado en características.

Un trabajo de investigación interesante sobre el enfoque híbrido es *Towards a Real-Time Visual Odometry using a Hybrid Approach* de Zou et al. (2018). Para llevar a cabo el seguimiento de características, los autores utilizan el algoritmo de detección y seguimiento KLT (Kanade-Lucas-Tomasi). Para la correlación de apariencia, utilizan el algoritmo de correlación normalizada. Luego, fusionan los resultados obtenidos mediante el seguimiento de características y la correlación de apariencia para obtener una estimación más precisa de la posición del robot. Los autores demuestran que su enfoque es capaz de realizar la odometría visual en tiempo real de forma precisa en diversos entornos.

Otro trabajo de investigación interesante sobre el enfoque híbrido es *A Hybrid Approach*

for Visual Odometry based on Monocular Visual SLAM de Zhang et al. (2020). Para llevar a cabo el seguimiento de características, los autores utilizan el algoritmo ORB-SLAM (Oriented FAST and Rotated BRIEF). Para la correlación de apariencia, utilizan una red neuronal convolutional. Luego, fusionan los resultados obtenidos mediante el seguimiento de características y la correlación de apariencia para obtener una estimación más precisa de la posición del robot. Los autores demuestran que su enfoque es capaz de realizar la odometría visual de forma precisa en diversos entornos y supera a otros métodos basados en características y apariencia en términos de precisión y robustez.

3.2 Algoritmos Fundamentales de Odometría Visual

A continuación, se presentarán algunos algoritmos utilizados en la odometría visual y que sirven como fundamento para calcular la matriz fundamental (F), que es una matriz de 3x3 que describe la relación entre los puntos de dos imágenes de una misma escena. Antes de entrar en detalle sobre estos algoritmos, se proporcionará una breve descripción del problema al que se enfrentan estos métodos (como se menciona en la referencia [15]). La matriz fundamental se utiliza a menudo en el procesamiento de imágenes y la robótica para estimar la posición y orientación de una cámara en relación con una escena dada.

$$F = \begin{pmatrix} f_{0,0} & f_{0,1} & f_{0,2} \\ f_{1,0} & f_{1,1} & f_{1,2} \\ f_{2,0} & f_{2,1} & f_{2,2} \end{pmatrix} \tag{3.1}$$

Siendo x un punto de la primera imagen, x' el mismo punto en la segunda imagen, F la matriz fundamental y lx la línea epipolar (ver Figura 3.1) que contiene el punto en la segunda imagen, se cumplen las siguientes ecuaciones.

$$lx'_i = Fx_i \tag{3.2}$$

$$lx_i = F^T x'_i \tag{3.3}$$

Las características más importantes de la matriz fundamental son las siguientes [Faugeras, 1992] :

- La multiplicación del punto en la segunda imagen traspuesto por la matriz fundamental y por el punto en la primera imagen da como resultado 0.

$$x'_i F x_i = 0 \tag{3.4}$$

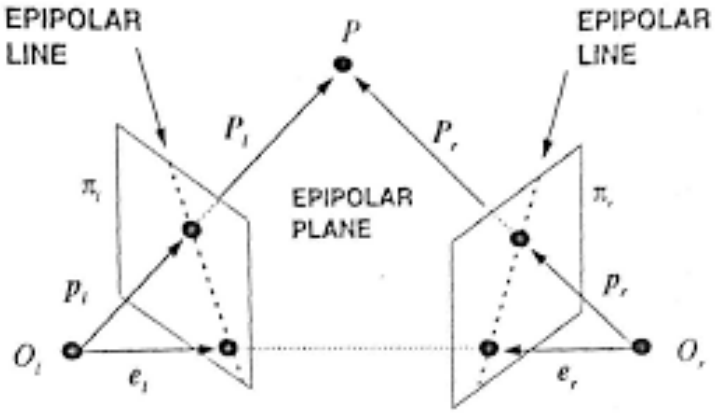


Figura 3.1: Relación entre los puntos de dos imágenes de una misma escena

- El rango de la matriz fundamental es 2, lo que significa que tiene siete grados de libertad, pudiendo calcular dos elementos de la matriz a partir de los otros.
- El determinante es 0.

$$\det(F) = 0 \tag{3.5}$$

- La matriz fundamental es homogénea lo que quiere decir que está definida hasta un factor de escala.

Para calcular la matriz fundamental lo primero que hay que hacer es relacionar de alguna forma los puntos de una imagen con los de la otra, es decir realizar un *emparejamiento*, por ejemplo utilizando el algoritmo de emparejamiento SURF.

Una vez que se tienen los puntos emparejados simplemente hay que desarrollar las ecuaciones y resolver el sistema. La ecuación desarrollada para un punto, siendo i este punto, es la siguiente:

$$\begin{aligned} x'_i x_i f_{0,0} + x'_i y_i f_{0,1} + x'_i f_{0,2} + y'_i x_i f_{1,0} + \\ y'_i y_i f_{1,1} + y'_i f_{1,2} + x_i f_{2,0} + y_i f_{2,1} + f_{2,2} = 0 \end{aligned} \tag{3.6}$$

El sistema se puede resolver de una forma mucho más sencilla si se expresa mediante matrices y se utiliza SVD (*Singular Value Descomposition* por sus siglas en inglés). De esta forma el sistema quedaría definido mediante la siguiente ecuación:

$$Af = 0 \tag{3.7}$$

siendo A una matriz de dimensiones Nx9, cuyas filas se definen así:

$$(x'_i x_i, x'_i y_i, x'_i, y'_i x_i, y'_i y_i, y'_i, x_i, y_i, 1)$$

y f un vector columna definido por los coeficientes de F:

$$(f_{0,0}, f_{0,1}, f_{0,2}, f_{1,0}, f_{1,1}, f_{1,2}, f_{2,0}, f_{2,1}, f_{2,2})^T$$

Dadas las características de la matriz fundamental no es necesario desarrollar un sistema de nueve ecuaciones, sino que pueden seguirse algunas estrategias para utilizar menos puntos. A continuación se describen los dos algoritmos más utilizados para resolver el sistema de ecuaciones.

3.2.1 Algoritmo de Ocho Puntos

Dado que la matriz F sólo está definida hasta un desconocido factor de escala, se impone una restricción adicional sobre la norma de F para eliminar el factor de escala, que consiste en que la norma cuadrática de F sea la unidad:

$$||F||^2 = ||f||^2 = f^t f = 1 \tag{3.8}$$

Para que el sistema de ecuaciones (3.7) admita una solución diferente del vector nulo, la matriz A debería tener un rango máximo de 8. Por lo tanto, el sistema de ecuaciones (3.7) puede ser resuelto mediante la correspondencia entre al menos 8 puntos diferentes en ambas cámaras.

3.2.2 Algoritmo de Siete Puntos

Este algoritmo utiliza la restricción del rango de la matriz fundamental. Dado que el rango de esta matriz es 2, se puede introducir una ecuación más que describe la relación entre dos filas de la matriz. De esta forma se pueden calcular dos términos a partir del resto, teniendo que introducir únicamente 7 puntos en las ecuaciones.

Además de estos algoritmos hay otros dos derivados de estos que utilizan sistemas de optimización para encontrar la matriz que mejor se ajusta a los puntos, eliminando a la vez los posibles outliers: RANSAC y LMEDS.

3.2.3 Optimización con RANSAC

RANSAC, (Fischler y Bolles, 1981)[9], es un algoritmo de optimización iterativo en el que, aplicado a la estimación de la matriz fundamental, en cada iteración se cogen 8 puntos

y se concreta que relaciona esos 8 puntos. Una vez calculada se comprueba qué tal se ajustan el resto de puntos a esta matriz. Si el ajuste es malo la matriz se desecha y si es buena se coge como mejor solución hasta encontrar una mejor o hasta que acabe el algoritmo. El pseudocódigo del método está representado en 1

Algorithm 1 Pseudocódigo de algoritmo de odometría visual optimizado con RANSAC

1: Seleccionar aleatoriamente el número mínimo de puntos necesarios para determinar los parámetros del modelo (*inliers*).
2: Resolver para los parámetros del modelo.
3: Determinar cuántos puntos del conjunto de todos los puntos encajan con una tolerancia predefinida E (*conjunto de consenso*).
4: Si la fracción del número de valores *inliers* sobre el número total de puntos en el conjunto excede un umbral predefinido K, volver a estimar los parámetros del modelo usando todos los valores *inliers* identificados y terminar.
5: De lo contrario, repita los pasos 1 a 4 (máximo de N veces).

3.2.4 Optimización con LMEDS

Least Median of Squares (Rousseeuw, 1984)[17] es un algoritmo de optimización iterativo. En caso de emplearse en la odometría visual, este encontrará una matriz fundamental óptima al conjunto de datos dado. Para ello, intenta minimizar el error cuadrático medio entre el modelo (la matriz fundamental calculada en la iteración anterior) y los puntos, mediante el algoritmo de descenso del gradiente de tal forma que según pasan las iteraciones la matriz fundamental calculada se ajusta mejor a los puntos dados.

Este método es muy resistente a las coincidencias falsas, así como a los valores atípicos debido a una mala localización. Sin embargo, es muy difícil definir el algoritmo completo para la mínima mediana de los cuadrados como una fórmula matemática para un conjunto de datos completo, por lo que un algoritmo que genera una solución

$$\min \underset{i}{med}\, r_i^2 \tag{3.9}$$

de la mínima mediana de los cuadrados para un conjunto de datos que se da a continuación.

La estimación de la mínima mediana de los cuadrados es resistente a los valores atípicos debido a su alto valor de descomposición del 50%. Esta es la fracción de valores atípicos que se pueden tolerar y al mismo tiempo arrojar una buena solución. Sin embargo, el alto valor de ruptura significa que LMEDS no se adapta bien al ruido gaussiano.

Algorithm 2 Pseudocódigo de algoritmo de odometría visual optimizado con LMEDS

1: Se elije m conjuntos aleatorios de puntos con tamaño p del conjunto de datos donde p es el número de parámetros en la ecuación que se está resolviendo.
2: para cada subconjunto se utiliza un método como el de la mínima media de los cuadrados para encontrar una solución para los parámetros de ese conjunto de datos.
3: para cada conjunto de parámetros p obtenidos, se calcula la mediana de los cuadrados de los residuos M con respecto a todo el conjunto de datos:

$$M_J = \underset{i=1,\ldots,n}{med} r_i^2(p_J, m_i) \tag{3.10}$$

4: La solución es p_J para la cual M_J es mínimo entre todos los m M_J .

Figura 3.2: Resultado de aplicar LMEDS para generar líneas epipolares

3.3 Conjuntos de Datos

Los *datasets* de odometría visual son conjuntos de datos que se utilizan para entrenar y evaluar algoritmos de odometría visual. Los *datasets* de odometría visual suelen incluir imágenes o vídeos capturados por una cámara móvil, junto con información de posición y orientación verdadera de la cámara en cada momento. Esta información se utiliza como referencia para evaluar la precisión de los algoritmos de odometría visual que estiman esa posición y orientación continuamente. Algunos ejemplos de *datasets* de odometría visual populares son:

- KITTI: Este *dataset* incluye imágenes y vídeos de un coche equipado con una cámara monocular y un LIDAR, así como información de posición y orientación verdadera obtenida a partir de un sistema de navegación inercial y GPS.

- EuRoC: Este *dataset* incluye imágenes y vídeos capturados por una cámara monocular y un LIDAR montados en un robot móvil, junto con información de posición y orientación verdadera obtenida a partir de un sistema de navegación inercial.

- TUM-RGBD: Este *dataset* incluye imágenes y vídeos capturados por una cámara RGB y un sensor de profundidad, junto con información de posición y orientación verdadera obtenida a partir de un sistema de navegación inercial.

Los *datasets* de odometría visual son muy útiles para la investigación y el desarrollo de algoritmos de odometría visual, ya que proporcionan una forma de evaluar la precisión y el rendimiento de estos sistemas de manera objetiva y cuantitativa. Además, también pueden ser útiles para la validación de sistemas de navegación autónoma en entornos reales.

A continuación se detallan los *datasets* que han sido empleados para la realización de este trabajo.

3.3.1 EuRoC MAV

EuRoC MAV (Micro Aerial Vehicle)[1]. *dataset* es un conjunto de datos de imágenes y metadatos recolectados por un vehículo aéreo no tripulado (UAV) con una cámara estereoscópica y un sistema de navegación inercial. Este *dataset* fue recolectado en diferentes entornos interiores y exteriores, incluyendo edificios, pasillos y patios.

El *dataset* incluye imágenes estereoscópicas a alta resolución, junto con información de posición, orientación y tiempo de captura de cada imagen. También incluye un archivo de trazado de trayectoria, que proporciona información verdadera sobre la posición y orientación del UAV en el momento de la captura de cada imagen.

El *dataset* EuRoC MAV consta de 11 misiones de vuelo, divididas en dos entornos: interior y exterior. Las misiones de interior incluyen un edificio industrial, un edificio de oficinas y un pasillo de un edificio universitario. Las misiones al aire libre incluyen un patio, un parque y un campo abierto.

Cada misión consta de varios cientos de imágenes estereoscópicas y metadatos precisos sobre la posición y orientación del UAV en el momento de la captura de cada imagen. El *dataset* también incluye un archivo de trazado de trayectoria, que proporciona información verdadera sobre la posición y orientación reales del UAV en el momento de la captura de cada imagen.

El *dataset* EuRoC MAV es ampliamente utilizado en investigaciones relacionadas con la navegación autónoma de vehículos aéreos no tripulados, el procesamiento de imágenes estereoscópicas, la detección y seguimiento de objetos en imágenes, y la reconstrucción de escenas 3D. Además, este *dataset* también es utilizado en la evaluación de algoritmos de navegación y control de UAVs.

[1] https://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets

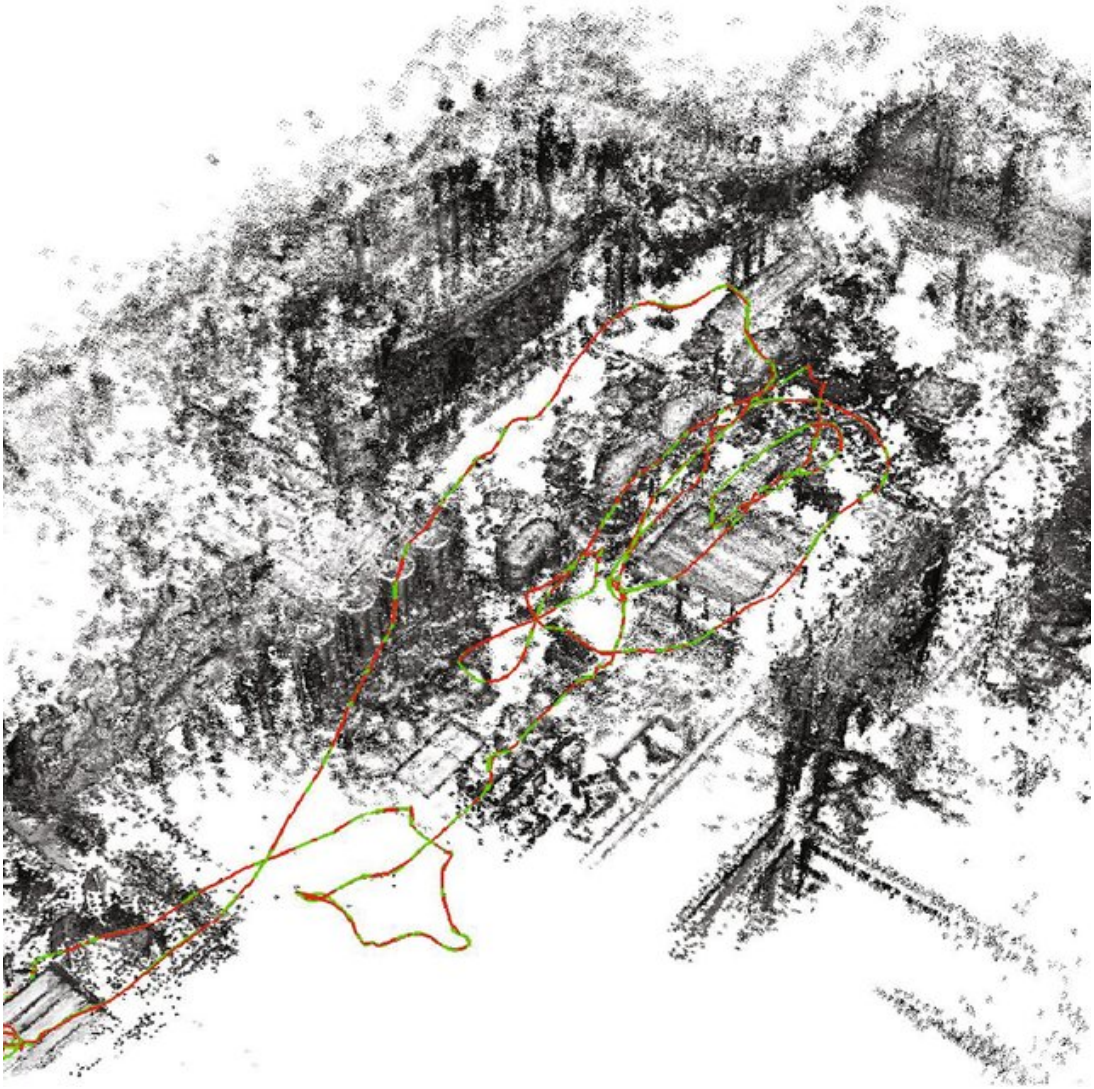


Figura 3.3: Reconstrucción de un escenario de EuRoC MAV utilizando la información de la IMU

3.3.2 KITTI

KITTI (Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago) [10] es un conjunto de datos de imagen y láser para la investigación en visión por computadora y aprendizaje automático. Fue creado por el Karlsruhe Institute of Technology y el Toyota Technological Institute en Chicago para promover la investigación en tecnologías de percepción y localización para sistemas de conducción automatizada.

El *dataset* KITTI incluye imágenes de cámaras de alta resolución y mediciones de escaneo láser para varias tareas de percepción, incluyendo la detección de obstáculos y la estimación de la profundidad. Las imágenes y las mediciones de escaneo láser se capturaron a bordo de un vehículo en movimiento en diferentes escenarios urbanos y rurales en

un área urbano y suburbano de Karlsruhe, Alemania.

Además, está dividido en diferentes conjuntos de datos, cada uno de ellos destinado a una tarea específica de percepción. Por ejemplo, el conjunto de datos *object* incluye imágenes y mediciones de escaneo láser para la detección de obstáculos y el conjunto de datos *odometry* incluye imágenes y mediciones de escaneo láser para la estimación de la posición y orientación del vehículo. El conjunto de datos de odometría consta de 22 secuencias estereoscópicas, guardadas en formato *png* sin pérdida. Se proporcionan 11 secuencias con trayectorias de verdad terrenal para entrenamiento y 11 secuencias sin trayectorias de verdad terrenal para evaluación.

KITTI es ampliamente utilizado en la investigación y el desarrollo de sistemas de conducción automatizada y ha sido utilizado en muchos trabajos de investigación publicados en revistas y conferencias importantes. Si se está interesado en utilizar el *dataset* KITTI para propias investigaciones o proyectos, se puede descargar gratuitamente desde el sitio web del proyecto[2].

Figura 3.4: Circuitos de referencia de KITTI

3.3.3 Virtual KITTI 2

Virtual KITTI 2 [3] es un *dataset* de imágenes y vídeos generadas por ordenador que simula escenas de conducción en un entorno urbano. El *dataset* se basa en el popular *dataset* KITTI, previamente mencionado, pero en lugar de utilizar imágenes y vídeos reales, utiliza un motor de videojuegos para generar las imágenes y vídeos de forma virtual.

El *dataset* contiene un total de 5 escenarios diferentes, cada uno con diferentes condiciones climáticas, iluminación y tráfico, así como imágenes tomadas con diferentes guiñadas. Además, el *dataset* incluye información de sensores como LIDAR, cámaras y GPS, lo que lo hace ideal para el desarrollo de sistemas de conducción autónoma. También incluye una variedad de objetos en las escenas, como vehículos, peatones, señales de tráfico y edificios, lo que lo hace ideal para el desarrollo de sistemas de detección y seguimiento de objetos. El *dataset* también incluye información de la posición y orientación de los objetos en las escenas, lo que lo hace útil para el desarrollo de sistemas de localización y navegación.

Además, el *dataset* Virtual KITTI 2 ha sido diseñado para ser lo más realista posible, con una gran variedad de condiciones climáticas, iluminación y tráfico. Esto significa que

[2] https://www.cvlibs.net/datasets/kitti/eval_odometry.php

los sistemas de conducción autónoma desarrollados utilizando este *dataset* tendrán una mayor capacidad para operar en entornos reales y enfrentar desafíos similares a los que se encontrarían en la vida real.

Figura 3.5: Imágenes del dataset VKITTI2

3.4 Evaluación de Algoritmos de Odometría Visual

La evaluación de algoritmos de odometría visual es una tarea básica en el desarrollo de robots autónomos y sistemas de realidad aumentada. Permite medir la precisión y robustez de los algoritmos utilizados para estimar la posición y orientación de la cámara en tiempo real. Para un alumno de robótica, la evaluación de algoritmos de odometría visual puede ser una herramienta valiosa para comprender cómo funcionan los sistemas de navegación en robótica y cómo mejorarlos. Al aprender a evaluar y comparar diferentes algoritmos, el alumno puede mejorar sus habilidades en programación y análisis de datos, y desarrollar una comprensión más profunda de la odometría visual y sus aplicaciones prácticas.

3.4.1 Herramienta de Evaluación de Algoritmos de SLAM de KITTI

La base de datos KITTI, descrita en la sección anterior, proporciona un conjunto de datos para evaluar algoritmos de odometría visual. Los algoritmos se evalúan en términos de precisión y robustez. La precisión se mide comparando la trayectoria estimada por el algoritmo con la trayectoria real medida por un sistema de referencia. La herramienta de evaluación incluye varias métricas comunes para evaluar la odometría visual, como el porcentaje de deriva de traslación en subsecuencias y el error de rotación en subsecuencias [12].

La robustez se evalúa en función de la capacidad del algoritmo para manejar diferentes

condiciones, como cambios en la iluminación y oclusiones. Los resultados pueden presentarse utilizando odometría visual monocular o estéreo, SLAM basado en láser o algoritmos que combinan múltiples sensores.

La herramienta de evaluación de odometría de KITTI se utiliza para evaluar los resultados de odometría en las 22 secuencias estéreo del conjunto de datos, donde 11 secuencias tienen trayectorias verdaderas. La evaluación se realiza para todas las subsecuencias posibles con longitudes entre 100 y 800 metros.

Existen varias implementaciones disponibles de la herramienta de evaluación de odometría de KITTI, incluyendo una implementación en Python disponible en Github [12].

3.4.2 Herramienta de Evaluación de Algoritmos SLAMTestbed

SlamTestbed (Barcia Mejias, 2019)[2] es una herramienta diseñada para comparar algoritmos SLAM de manera cuantitativa. Se utiliza como una *caja negra* que toma como entrada dos secuencias de puntos 3D orientados. Una de ellas es la verdad absoluta, mientras que la otra es la posición y orientación calculadas por un algoritmo de Visual SLAM que se desea evaluar. La herramienta procesa estas dos secuencias y devuelve las transformaciones estimadas entre la verdad absoluta y las posiciones y orientaciones calculadas, así como un conjunto de estadísticas que miden el error en las estimaciones. Esto permite caracterizar la precisión de los algoritmos de SLAM.

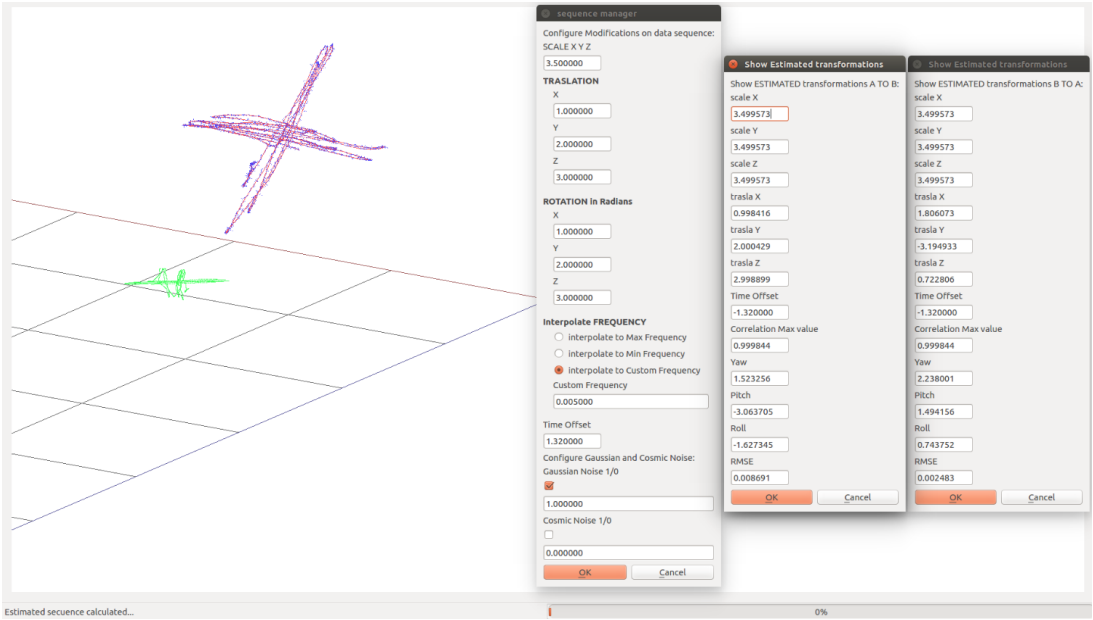


Figura 3.6: SLAMTestbed. Resultados de la estimación de un cambio de escala y traslación, rotación, offset y ruido gaussiano simultáneos.

La herramienta tiene como objetivo principal calcular el error entre una secuencia de

posiciones y orientaciones 3D verdaderas (camino A) y la trayectoria calculada por el algoritmo de SLAM (camino B). Para lograr esto, se eliminan variables que impiden comparar directamente las dos trayectorias, como la escala, el offset temporal y la transformación en 3D entre ellas. Esto se logra mediante el cálculo de un nuevo camino (estimado) *comparable* con el camino A.

Para obtener el camino estimado se utilizan las siguientes funcionalidades:

- **Cálculo de PCA**: permite reducir las dos secuencias a sus componentes principales, lo que posibilita estimar la escala y el offset existente entre ellos.
- **Estimación de escala**: estima la diferencia de escala entre las dos secuencias a partir de los datos proporcionados en el cálculo de componentes principales.
- **Estimación de offset temporal**: permite hallar la diferencia entre marcas de tiempo de las dos secuencias, ya que pueden haber comenzado en periodos de tiempo distintos.
- **Interpolación para igualar frecuencias de muestreo**: se pueden igualar en frecuencia las dos secuencias, en caso de que estas sean distintas.
- **Operaciones de registro para estimar la Rotación y Traslación**: permite estimar la traslación y rotación existentes entre el camino A y el camino B y llevarlos al mismo sistema de referencia espacial, donde ya son directamente comparables.

3.4.3 Análisis de algoritmos de VisualSLAM: un entorno integral para su evaluación

Análisis de algoritmos de VisualSLAM: un entorno integral para su evaluación (Arribas Raigadas, 2016)[1] describe un entorno de evaluación de algoritmos de VisualSLAM (Simultaneous Localization and Mapping). Asimismo el autor desarrolla una herramienta de análisis, en el capítulo 8 de la citada referencia, la cual está caracterizada por los siguientes pasos: arquitectura de evaluación, métricas que emplea para la evaluación, y el medio físico para presentar los resultados.

En cuanto a la arquitectura, el autor presenta tres etapas, la primera etapa comprende la ejecución de un algoritmo de VisualSLAM, del cual se toma su estimación. La segunda etapa comprende el ajuste temporal y espacial, de este modo obtiene dos fuentes comparables de la información de la localización. La tercera etapa implica la evaluación de ambas fuentes, de donde se obtiene las diferentes métricas que propone.

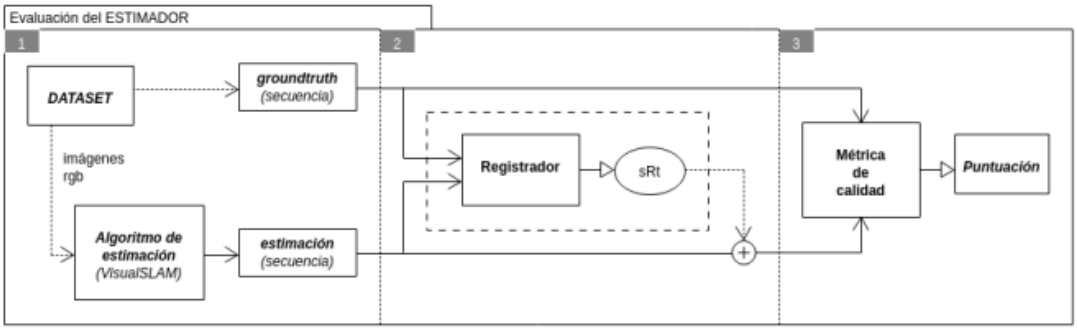


Figura 3.7: Arquitectura de la herramienta propuesta por Víctor Arribas.

3.5 Plataformas Educativas en Robótica

El aprendizaje en robótica se ha vuelto cada vez más popular en la educación y la industria debido a su potencial para desarrollar habilidades en áreas como la programación, la visión artificial y la mecánica. Las plataformas educativas en robótica similares a Unibotics, ofrecen una variedad de cursos, actividades y recursos para aprender robótica de manera interactiva. Además, algunas de estas plataformas también incluyen robots con visión, lo que permite a los estudiantes aprender sobre sistemas de visión y aplicarlos en la programación de robots. En esta sección se comentan algunas de las plataformas educativas más destacadas.

3.5.1 TheConstruct

TheConstruct [3] es una plataforma web educativa especializada en robótica, ROS e inteligencia artificial. Ofrece una versión gratuita con tres cursos básicos: Linux para robótica, Python3 para robótica y C++ para robótica, y una versión de pago con acceso a todos los cursos disponibles. Está diseñada para ser utilizada tanto por principiantes como por profesionales, y no requiere la instalación de ROS. Además, cuenta con una gran comunidad donde se pueden establecer contactos y aprender nuevas formas de programar robots. TheConstruct ofrece la posibilidad de alquilar robots reales y programarlos desde cualquier lugar, y tiene un interfaz de usuario fácil de usar con una sección para la teoría, un espacio para escribir código, un terminal para escribir comandos y una simulación del robot a programar.

[3] https://www.theconstructsim.com/

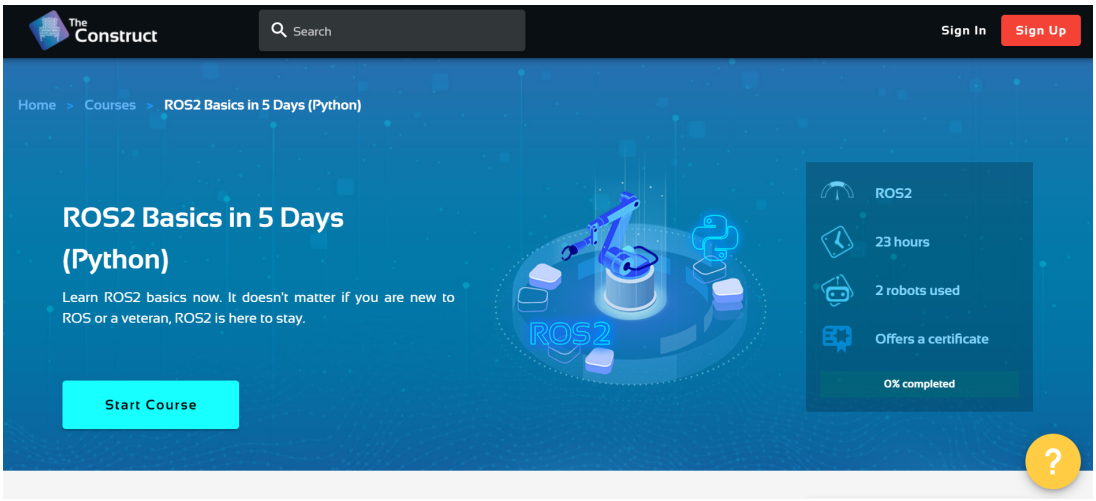


Figura 3.8: Plataforma de educación en robótica en ingeniería TheConstruct

3.5.2 Riders.ai

Riders.ai[4] es una plataforma en línea desarrollada por Acrome Robotic Systems que ofrece educación, simulación y competiciones en robótica. Es una plataforma de pago, donde solo se ofrece la primera lección gratuita. Ofrece dos cursos adicionales con los cuales se obtiene un certificado una vez completados. Los cursos enseñan a programar robots utilizando Python o C++, y se componen de teoría, un interfaz de usuario y el simulador Gazebo. Todo está ubicado en la misma pestaña, como se ve en la Figura 3.9. Riders también cuenta con dos ligas de competición para programar drones voladores y robots móviles.

[4] https://riders.ai/

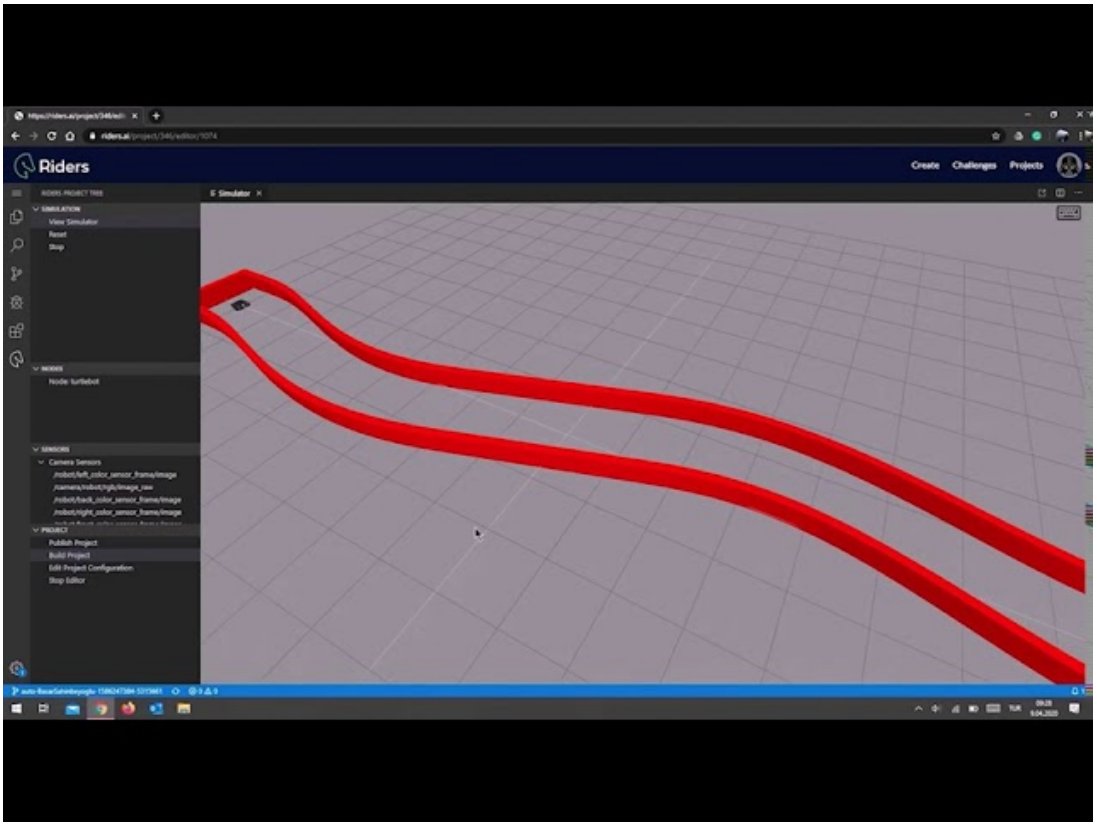


Figura 3.9: Plataforma de robótica educativa Riders.ai

Capítulo 4

Algoritmo Robusto de Odometría Visual 3D

Este capítulo presenta una aplicación de odometría visual 3D desarrollada con una intención educativa, con el objetivo de proporcionar un ejemplo práctico y comprensible de cómo funciona esta técnica de autolocalización visual en robots móviles y vehículos autónomos. El algoritmo se basa en la utilización de técnicas de detección y seguimiento de características 3D, así como en la integración de información de un sensor óptico, una cámara monocular RGB, para realizar la estimación de la posición y orientación tridimensional. Se ha diseñado la aplicación *offline* de manera clara y sencilla para facilitar su uso como herramienta didáctica en el ámbito de la odometría visual y la navegación de robots.

4.1 Formulación Matemática

El algoritmo de odometría visual 3D se basa en la detección y seguimiento de características 3D en secuencias de imágenes capturadas por una cámara. La idea principal es estimar la *pose* (posición y orientación) del robot en cada instante de tiempo a partir de la observación de cómo cambian las características 3D en las imágenes y de la *pose* anterior. Enfrentándose de forma principal al problema del factor de escala, sujeto en este tipo de problemas.

4.1.1 Traslaciones y Rotaciones Incrementales

La *pose* del robot en cada instante de tiempo se puede expresar mediante una transformación homogénea $\mathbf{T}_t = \begin{bmatrix} \mathbf{R}_t & \mathbf{t}_t \\ \mathbf{0}^T & 1 \end{bmatrix}$, donde $\mathbf{R}_t$ es la matriz de rotación y $\mathbf{t}_t$ es el

vector de translación que representan la orientación y posición del robot, respectivamente.

La posición del robot en cada instante de tiempo, respecto al sistema de referencia absoluto, que suele ser la posición inicial de la cámara o el robot en la escena, se puede obtener a partir del vector de translación $\mathbf{t}_t = \begin{bmatrix} x_t & y_t & z_t \end{bmatrix}^T$.

Para calcular la pose del robot en el instante de tiempo $t+1$ a partir de la pose en el instante de tiempo t, se utiliza la siguiente ecuación:

$$\mathbf{T}_{t+1} = \mathbf{T}_t \cdot \Delta\mathbf{T}_{t,t+1} \tag{4.1}$$

donde $\Delta\mathbf{T}_{t,t+1} = \begin{bmatrix} \mathbf{R}_{t,t+1} & \mathbf{t}_{t,t+1} \\ \mathbf{0}^T & 1 \end{bmatrix}$ es la transformación que representa el cambio de pose entre el instante de tiempo t y el $t+1$.

Para obtener la posición absoluta estimada (a partir de aquí nos referiremos a la posición absoluta estimada como posición absoluta, no confundir con la posición absoluta real) del robot en cada instante de tiempo, es necesario llevar la pose del robot a un sistema de referencia absoluto. Esto se puede hacer mediante la concatenación de las poses del robot en todos los instantes de tiempo desde el inicio hasta el instante de tiempo actual.

La posición absoluta del robot en el instante de tiempo t, $\mathbf{t}_t^{abs}$, se puede calcular mediante la siguiente ecuación:

$$\mathbf{t}_t^{abs} = \mathbf{t}_0^{abs} \cdot \sum_{i=1}^{t} \Delta\mathbf{t}_{t-1,t} \tag{4.2}$$

donde $\mathbf{t}_0^{abs}$ es la posición absoluta del robot en el instante de tiempo inicial, que se puede establecer de forma arbitraria, y $\Delta\mathbf{t}_{t-1,t}$ es la variación de la pose del robot en cada uno de los instantes de tiempo respecto a su instante de tiempo inmediatamente anterior desde el inicio de la secuencia hasta el instante de tiempo actual.

La posición absoluta del robot en el instante de tiempo t se puede obtener, también, a partir del vector de translación $\mathbf{t}_t^{abs} = \begin{bmatrix} x_t^{abs} & y_t^{abs} & z_t^{abs} \end{bmatrix}^T$ de la pose absoluta $\mathbf{T}_t^{abs}$.

Otra forma de calcular la posición absoluta del robot de manera incremental es utilizando la siguiente ecuación:

$$\mathbf{t}_{t+1}^{abs} = \mathbf{t}_t^{abs} + \mathbf{R}_t^{abs} \cdot \mathbf{t}_{t,t+1} \tag{4.3}$$

Donde:

- $\mathbf{t}_{t+1}^{abs}$ es el vector de translación de la pose absoluta del robot en el instante de tiempo $t+1$.
- $\mathbf{t}_{t}^{abs}$ es el vector de translación de la pose absoluta del robot en el instante de tiempo t.
- $\mathbf{R}_{t}^{abs}$ es la matriz de rotación de la orientación absoluta del robot en el instante de tiempo t.
- $\mathbf{t}_{t,t+1}$ es el vector de translación de la pose del robot entre el instante de tiempo t y el $t+1$.

La ecuación 4.3 se basa en la idea de que la posición absoluta del robot en el instante de tiempo $t+1$ es igual a la posición absoluta del robot en el instante de tiempo t más la translación del robot entre el instante de tiempo t y el incremento de ésta en $t+1$, expresada en el sistema de referencia absoluto. El incremento entre el instante de tiempo t y el $t+1$ se puede calcular a partir de observar cómo cambian las características 3D en las imágenes.

La orientación del robot en cada instante de tiempo se puede obtener a partir de la matriz de rotación $\mathbf{R}_t$ de la pose del robot en ese instante de tiempo, que se puede expresar como una combinación de rotaciones en torno a los ejes x, y y z.

Para obtener la orientación del robot en términos de ángulos de Euler, se pueden utilizar las siguientes ecuaciones:

$$\alpha = atan2(R_{3,2}, R_{3,3}) \quad \beta = asin(-R_{3,1}) \quad \gamma = atan2(R_{2,1}, R_{1,1}) \tag{4.4}$$

donde α, β y γ son los ángulos de Euler en torno a los ejes x, y y z, respectivamente, y $R_{i,j}$ es el elemento de la matriz de rotación $\mathbf{R}_t$ en la fila i y la columna j.

De forma análoga a la ecuación 4.2, la orientación absoluta del robot en el instante de tiempo t se puede obtener a partir de la matriz de rotación $\mathbf{R}_t^{abs}$ de la pose absoluta del robot en ese instante de tiempo, que se puede calcular mediante la siguiente ecuación:

$$\mathbf{R}_t^{abs} = \mathbf{R}_0^{abs} \cdot \prod_{i=1}^{t} \Delta\mathbf{R}_{t,t+1} \tag{4.5}$$

donde $\mathbf{R}_0^{abs}$ es la matriz de rotación de la orientación absoluta del robot en el instante de tiempo inicial, que se puede establecer de forma arbitraria, y $\Delta\mathbf{R}_{t,t+1}$ es la matriz de

rotación de la orientación del robot en cada uno de los instantes de tiempo desde el inicio hasta el instante de tiempo actual.

Asimismo, para calcular la orientación absoluta de forma incremental se puede utilizar la siguiente ecuación:

$$\mathbf{R}_{t+1}^{abs} = \mathbf{R}_{t}^{abs} \cdot \mathbf{R}_{t,t+1} \tag{4.6}$$

donde $\mathbf{R}_{t+1}^{abs}$ es la matriz de rotación de la orientación absoluta del robot en el instante de tiempo $t+1$, $\mathbf{R}_{t}^{abs}$ es la matriz de rotación de la orientación absoluta del robot en el instante de tiempo t y $\mathbf{R}_{t,t+1}$ es la matriz de rotación de la orientación del robot entre el instante de tiempo t y el $t+1$.

Una vez obtenida la matriz de rotación $\mathbf{R}_{t}^{abs}$, se puede utilizar alguna de las ecuaciones que se mencionan en (4.4) para obtener la orientación absoluta del robot en términos de ángulos de Euler.

Se pueden obtener el vector $\mathbf{t}_{t,t+1}$ y la matriz $\mathbf{R}_{t,t+1}$ haciendo uso de la *matriz esencial*.

4.1.2 Matriz Esencial

La *matriz esencial* es una matriz de 3x3 que se utiliza en visión artificial para relacionar los puntos correspondientes en imágenes estéreo. Las imágenes estéreo son un par de imágenes tomadas desde diferentes puntos de vista, y se utilizan para obtener información sobre la profundidad y la estructura 3D de una escena. La matriz esencial se utiliza para calcular la geometría epipolar[1] entre las dos cámaras que tomaron las imágenes estéreo.

La matriz esencial se puede utilizar para calcular la rotación y la traslación entre las dos cámaras que tomaron las imágenes estéreo, que también puede ser la misma cámara física tomando imágenes en dos instantes próximos en tiempo.

Para calcular la rotación y la traslación a partir de la matriz esencial, se puede utilizar la descomposición en valores singulares (SVD) de la matriz esencial. La descomposición en valores singulares de la matriz esencial E se puede escribir como:

$$E = USV^{T} \tag{4.7}$$

donde U y V son matrices ortogonales y S es una matriz diagonal con los valores sin-

[1] La geometría epipolar es la geometría intrínseca entre dos vistas de una misma escena.

gulares de E.

A partir de esta descomposición, se pueden calcular dos posibles soluciones para la rotación y la traslación entre las dos cámaras. Las soluciones para la rotación son:

$$R = U \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} V^T$$

y:

$$R = U \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} V^T$$

Las soluciones para la traslación son:

$$t_1 = u_3$$

y:

$$t_2 = -u_3$$

donde u_3 es la tercera columna de la matriz U.

Otra forma de expresar la matriz esencial es:

$$E = R[t]_x \tag{4.8}$$

donde $[t]_x$ es la matriz resultante del producto cruzado de t.

Estimación de la Matriz Esencial

Para calcular la matriz esencial, se necesitan las correspondencias entre las características 2D en las dos poses y la matriz intrínseca de la cámara. Una vez que se tienen las correspondencias, se pueden utilizar técnicas de optimización para encontrar la matriz esencial que mejor se ajusta a las correspondencias detectadas.

En nuestro caso, utilizaremos el algoritmo RANSAC, implementado en la biblioteca *OpenCV*, para calcular la matriz esencial. Este algoritmo se describe en detalle en el capítulo

3.2.3. Según lo explicado en el capítulo 6 de la referencia [14], RANSAC ofrece un buen equilibrio entre velocidad de ejecución y precisión en la estimación de la matriz esencial. Por esta razón, consideramos que es una buena opción para nuestro caso en particular.

Aquí se puede observar un segmento de código de la solución de referencia en la que se extraen las características de las imágenes y se obtiene la matriz esencial.

```
self.px_ref, self.px_cur = feature_tracking(self.last_frame, self.new_frame, self.px_ref)
E, _ = cv2.findEssentialMat(self.px_cur, self.px_ref, focal=self.focal, pp=self.pp,
    method=cv2.RANSAC, prob=0.999, threshold=1.0)
```

Código 4.1: Cálculo de la matriz esencial

El seguimiento de características entre dos imágenes se realiza mediante la función *feature_tracking*. Toma como entrada una imagen de referencia, *image_ref*, una imagen actual, *image_cur*, y un conjunto de puntos de referencia, *px_ref*. Utiliza la función *calcOpticalFlowPyrLK*, de *OpenCV* también, para calcular el flujo óptico entre las dos imágenes y obtener los puntos clave en la imagen actual *kp2*. Los puntos clave que son encontrados con éxito son seleccionados utilizando el estado *st*. Finalmente, la función devuelve los puntos clave en la imagen de referencia y en la imagen actual que fueron encontrados con éxito en ambas imágenes.

```
def feature_tracking(image_ref, image_cur, px_ref):
  kp2, st, _ = cv2.calcOpticalFlowPyrLK(image_ref, image_cur, px_ref, None, **lk_params)
      #shape: [k,2] [k,1] [k,1]

  st = st.reshape(st.shape[0])
  kp1 = px_ref[st == 1]
  kp2 = kp2[st == 1]

  return kp1, kp2
```

Código 4.2: Función para el seguimiento de características

A continuación se explicará el código que se ha sido necesario desarrollar antes de llevar a cabo la integración en Unibotics, de la *solución de referencia*.

4.2 Solución de Referencia

En esta sección se explicará la solución de referencia haciendo hincapié en la parte referente a odometría visual; se dará a entender la importancia del diseño de la solución de referencia del producto final ya que permite entender la organización del código y cómo está dividido en distintas partes y módulos. Además, al describir el diseño se hará una introducción a los diferentes archivos y scripts incluidos en esta solución y su función específica.

También, se presentan los pseudocódigos del backend y el web, que muestran de manera resumida y esquemática el flujo de ejecución del código y cómo interactúan entre sí las distintas partes de la solución. Esto ayudará a comprender cómo se han implementado las distintas funcionalidades del proyecto y cómo se han integrado posteriormente en la plataforma educativa.[2] Cabe mencionar que dicho código es compatible con los tres datasets que se describen en la sección 3.3.

4.2.1 Diseño

El diseño consta de dos partes principales: *backend* y *web*. El *backend* contiene el código que lanza la aplicación y todos los módulos necesarios para ejecutar esta, además el algoritmo de odometría visual 3D. En el backend se encuentra el archivo *server_app.py*, que es el punto de entrada de la aplicación, y el archivo *requirements.txt*, que especifica las dependencias necesarias para que esta funcione correctamente. También se incluyen varios scripts en *Bash* en la raíz de esta sección *backend* para facilitar la instalación de los datasets y ejecución de esta solución de referencia.

backend/src contiene todos los módulos Python necesarios para ejecutar el algoritmo de odometría visual 3D, que se ha programado usando tecnologías web. Dentro de esta sección se encuentran varias subsecciones, como *maths*, *scales* y *utils*, que almacenan módulos relacionados con las operaciones matemáticas necesarias para el algoritmo, el procesamiento de datos de escala y tiempo entre utilidades diversas, respectivamente. La carpeta *tests* contiene scripts de prueba para los módulos de la carpeta *src*, concretamente para comprobar que la transformación entre matrices de rotación y ángulos de Euler y viceversa sea correcta.

Por otra parte, en *web* se contiene el código de la interfaz gráfica, *gui*, que se utiliza para visualizar los resultados del algoritmo de odometría visual 3D y que se ha programado usando tecnologías web. Aquí se encuentran varios archivos y carpetas, como *index.html*, que es la página principal de la interfaz, y *script.js*, que es el archivo principal de JavaScript que se encarga de la lógica de la interfaz. También se incluye un archivo *style.css* para definir el estilo de la interfaz y una carpeta *static* que almacena recursos estáticos, como texturas y gráficos, utilizados por la interfaz y que atañen a este documento.

web/src contiene los módulos y objetos JavaScript que se utilizan en la interfaz gráfica. Dentro de esta carpeta se encuentra un subdirectorio *3dScene* que almacena el código encargado de generar y controlar la escena 3D que se muestra en la interfaz; *vehicles* que

[2]El código relacionado con esta sección del trabajo puede encontrarse en el repositorio oficial del proyecto en GitHub, que se puede acceder a través del siguiente enlace: https://github.com/RoboticsLabURJC/2020-tfm-pablo-asensio. Además, si se desea obtener una mayor comprensión de la metodología de trabajo utilizada, se puede consultar el repositorio personal del proyecto en GitHub, que se encuentra disponible en el siguiente enlace: https://github.com/PabloAsensio/MonocularVisualOdometry

almacena el código relacionado con los vehículos que se muestran en la escena. *utils* que almacena módulos de utilidad diversa.

Cabe destacar que el visor web se realiza de tal forma que su migración a Unibotics sea lo más cómoda posible.

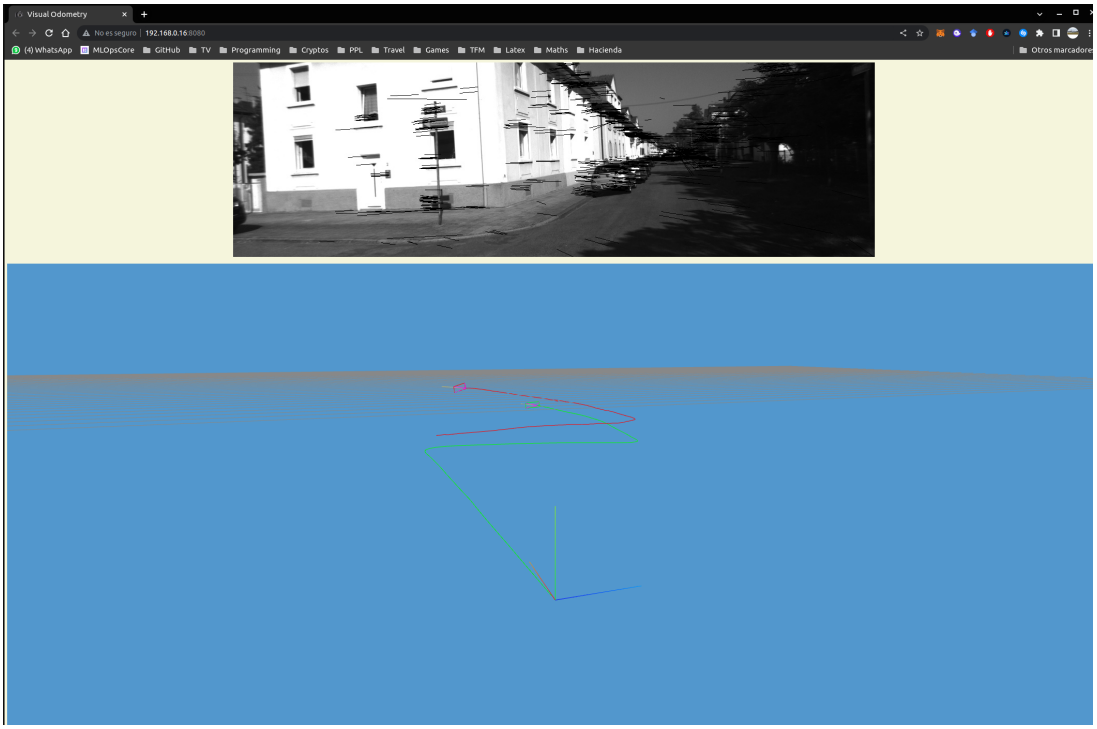


Figura 4.1: Interfaz gráfica de la solución de referencia

Aquí podemos observar la estructura de carpetas en detalle:

```
.
├── backend
│   ├── app.py
│   ├── find_timestamp.py
│   ├── plot_groundtruth.py
│   ├── requirements.txt
│   ├── server_app.py
│   ├── setup.sh
│   ├── src
│   │   ├── maths
│   │   │   ├── cv_maths.py
│   │   │   ├── quaternions.py
│   │   │   ├── rotation_matrix.py
│   │   │   └── rotation_to_euler.py
│   │   ├── pinhole_camera.py
```

```
│   │   ├── scales
│   │   │   ├── find_timestamp.py
│   │   │   ├── grep.py
│   │   │   ├── read_eurocmav_timestamp_groundtruth.py
│   │   │   ├── scales.py
│   │   │   └── true_rotation.py
│   │   ├── utils
│   │   │   ├── load_images.py
│   │   │   └── messages.py
│   │   └── visual_odometry.py
│   └── tests
│       └── maths
│           ├── test_rotation_matrix.py
│           └── test_rotation_to_euler.py
├── dataset.cfg
├── download-datasets.sh
├── web
│   ├── package-lock.json
│   ├── package.json
│   ├── src
│   │   ├── 3dScene
│   │   │   └── scene.js
│   │   ├── index.html
│   │   ├── script.js
│   │   ├── style.css
│   │   ├── utils
│   │   │   ├── CBuffer.js
│   │   │   └── corrector.js
│   │   └── vehicles
│   │       ├── car.js
│   │       └── frame.js
│   └── static
│       └── textures
└── run.sh
```

4.2.2 Funcionamiento General del Backend

A continuación se presenta el pseudocódigo correspondiente a la parte de la aplicación que se ejecuta en segundo plano, el *backend*. El archivo principal encargado de la lógica *backend* es *server_app.py*, llamado así por montar un servidor local que ejecuta la aplicación. Se puede observa en el siguiente pseudocódigo el funcionamiento de este.

Algorithm 3 Pseudocódigo *funcionamiento del backend*

1: Leer el fichero de configuración
2: Pre-cargar las imágenes
3: **while** no sea la última imagen **do**
4: Actualizar la pose del vehículo con la imagen actual
5: Enviar el mensaje al web
6: **end while**

Donde el mensaje se compone de la imagen codificada en *base64* del instante t, y la *pose* calculada y la de referencia.

```
async def app(websocket):
   print("A client just connected")
   try:
      config = configparser.RawConfigParser()
      config.read('../dataset.cfg')
      dataset_info = dict(config.items('TO_READ'))

      run = await websocket.recv()
      if run == "run":
         status = await monocular_visual_odometry(websocket, dataset_info)

         if status == 0:
            print("Finished")
            await websocket.send("close")
            exit(0)

         if status == 1:
            print('Something went wrong.')
            exit(1)

   except websockets.exceptions.ConnectionClosed as e:
      print("A client just disconnected")
      exit(0)

if __name__ == "__main__":
   start_server = websockets.serve(app, "localhost", PORT)
   asyncio.get_event_loop().run_until_complete(start_server)
   asyncio.get_event_loop().run_forever()
```

Código 4.3: Función principal del backend

En este Código 4.3 se puede observar que se habilita el puerto *PORT* para el funcionamiento de la aplicación: función *app*, que se ejecutará haciendo uso de un *event loop* de forma asíncrona. Esta función *app* se encarga de leer el fichero de configuración y esperar hasta que se le de la orden de arranque *run*, la cual hará que se ejecute el algoritmo de odometría visual 3D: *monocular_visual_odometry*

4.2.3 Captura de imágenes y extracción de características

La función asíncrona *monocular_visual_odometry* (ver Código 4.4). Recibe el *websocket* que está habilitado para comunicarse con la interfaz gráfica y los datos de configuración. Se encarga de la extracción de las imágenes (línea 5), precargándolas y guardándolas en un *array*. Después se va iterando sobre este *array*, donde se llama al método *update* de la clase *VisualOdometry*, clase que contiene el *algoritmo de odometría visual*.

```
1  async def monocular_visual_odometry(websocket, info: dict) -> int:
2
3     if info['dataset'] not in ['kitti', 'vkitti2', 'eurocmav']: return 1
4
5     imgs, list_timestamps = load_images(info['images_path'])
6
7     if info['dataset'] == 'eurocmav':
8        print("From eurocmav".upper())
9        camera = PinholeCamera.from_eurocmav(info['calibration_file'])
10       vo = VisualOdometry(camera, info['ground_truth_file'], dataset=info['dataset'])
11
12       vo.frame_timestamps_list = list_timestamps
13       vo.timestamp_groundtruth_list = read_eurocmav_timestamp_groundtruth(info['
             ground_truth_file'])
14
15    if info['dataset'] == 'kitti':
16       print("From kitti".upper())
17       camera = PinholeCamera.from_kitti(info['calibration_file'], width=1241, height=376)
18       vo = VisualOdometry(camera, info['ground_truth_file'], dataset=info['dataset'])
19
20    if info['dataset'] == 'vkitti2':
21       print("From vkitti2".upper())
22       camera = PinholeCamera.from_vkitti2(info['calibration_file'], width=1242, height
             =375, camera=0)
23       vo = VisualOdometry(camera, info['ground_truth_file'], dataset=info['dataset'])
24
25
26    for img_id, img in tqdm(enumerate(imgs), desc="Progress", ascii=True, total=len(imgs))
          :
27
28       if img_id == 0 or img_id == len(imgs):
29          continue
30
31       if vo.dataset == "eurocmav":
32          vo.timestamp = list_timestamps[img_id]
33
34       vo.update(img, img_id)
35
36       message = await create_message(vo, img, img_id)
37       await send_message(websocket, message)
38       cv2.waitKey(30)
39
40    return 0
```

Código 4.4: Función principal de odometría visual

4.2.4 Algoritmo de Odometría Visual 3D Incremental

La clase *VisualOdometry*, que contiene el *algoritmo de odometría visual*, está en el fichero *visual_odometry.py*, ver Pseudocódigo 4. Esta se encarga de calcular la pose del vehículo en función del dataset con el que se esté trabajando. Cada dataset tiene su propio formato, por lo que es necesario tener en cuenta esta variable a la hora de realizar los cálculos. También se encarga de gestionar todas las operaciones necesarias para llevar a cabo el algoritmo de odometría visual 3D de forma efectiva.

Se puede observar en Código 4.5 los atributos de la clase *VisualOdometry*. Todos estos hacen alusión o a configuraciones del dataset o a elementos necesarios para el correcto funcionamiento del algoritmo, como *new_frame, last_frame, detector, init_R, init_t, etc*.

```
class VisualOdometry:

   def __init__(self, camera: PinholeCamera, groundtruth_file: str, dataset: str):

      self.dataset = dataset
      self.camera = camera

      self.frame_stage = 0

      self.new_frame = None
      self.last_frame = None

      self.cur_R = np.eye(3)
      self.cur_t = None
      self.px_ref = None
      self.px_cur = None
      self.focal = camera.fu
      self.pp = (camera.cu, camera.cv)

      self.trueX, self.trueY, self.trueZ = 0, 0, 0
      self.true_R = np.zeros( shape=(3,3) )
      self.true_t = np.zeros((1,3))

      self.init_R = None
      self.init_t = None

      self.detector = cv2.FastFeatureDetector_create(threshold=25, nonmaxSuppression=True
          )
      self.groundtruth = read_groundtruth(groundtruth_file, dataset)

      self.timestamp = None
```

```
31        self.frame_timestamps_list = None
32        self.timestamp_groundtruth_list = None
```

Código 4.5: Declaración de VisualOdometry

El método de la clase llamado *update* se puede entender de la siguiente forma:

Algorithm 4 Pseudocódigo método *update* de la clase *VisualOdometry*

```
1:  if tamaño de imagen es inválido then
2:      devolver error
3:  end if
4:  if es el primer frame then
5:      obtener puntos característicos de la imagen
6:      igualar posición actual a 0
7:      guardar matriz de rotación inicial
8:  else
9:      if es el segundo frame then
10:         calcular puntos de interés
11:         no actualizar pose
12:     else
13:         calcular puntos de interés
14:         actualizar pose
15:     end if
16: end if
```

En Código 4.6 se ve esta diferenciación comentada en Pseudocódigo 4. Al procesar la imágen es dónde se realiza el *cálculo de la matriz esencial y la actualización de la pose*.

```
1  def update(self, img: np.ndarray, frame_id: int, R=None, t=None) -> None:
2     assert (
3        img.ndim == 2
4        and img.shape[0] == self.camera.height
5        and img.shape[1] == self.camera.width
6     ), "Frame: provided image has not the same size as the camera model or image is not
          grayscale"
7
8     self.new_frame = img
9
10    if self.frame_stage == STAGE_DEFAULT_FRAME:
11       self.process_frame(frame_id)
12
13    elif self.frame_stage == STAGE_SECOND_FRAME:
14       self.process_second_frame()
15
16    elif self.frame_stage == STAGE_FIRST_FRAME:
17       self.process_first_frame()
18
19    self.last_frame = self.new_frame
```

Código 4.6: Método update

4.2.5 Cálculo de la Matriz Esencial y Actualización de la Pose

El procesamiento de la imagen se puede entender dentro de Pseudocódigo 5. Donde se detalla de una mejor forma el proceso de actualización de la *pose*.

Algorithm 5 Pseudocódigo actualización pose

1: Realizar el seguimiento de los puntos de interés
2: utilizar *findEssentialMat* de OpenCV para calcular la matriz esencial
3: utilizar *recoverPose* de OpenCV para obtener a partir de la matriz esencial, las matrices de rotación y el vector posición
4: calcular la escala
5: **if** la escala pasa un umbral **then**
6: actualizar la posición absoluta con la formula incremental corregida por la escala
7: actualizar la matriz de rotación con la formula incremental
8: **end if**
9: **if** los puntos de interés son menos que un umbral **then**
10: calcular nuevos puntos de interés
11: **end if**

Figura 4.2: Flujo de características entre imágenes

```
def process_frame(self, frame_id: int) -> None:

   self.px_ref, self.px_cur = feature_tracking(self.last_frame, self.new_frame, self.
       px_ref)
   E, _ = cv2.findEssentialMat(self.px_cur, self.px_ref, focal=self.focal, pp=self.pp,
       method=cv2.RANSAC, prob=0.999, threshold=1.0)
   _, R, t, _ = cv2.recoverPose(E, self.px_cur, self.px_ref, focal=self.focal, pp = self.
       pp)
   absolute_scale = self.calculate_absolute_scale(frame_id)
```

```
    self.calculate_true_rotration(frame_id)

    if(absolute_scale > 0.1):
        self.cur_t = self.cur_t + absolute_scale * self.cur_R @ t
        self.cur_R = R @ self.cur_R

    if(self.px_ref.shape[0] < kMinNumFeature):
        self.px_cur = self.detector.detect(self.new_frame)
        self.px_cur = np.array([x.pt for x in self.px_cur], dtype=np.float32)

    self.px_ref = self.px_cur
```

Código 4.7: Método process_frame

El método *process_frame* hace uso de *feature_tracking* que se ha explicado en Código 4.2 para la correlación de características. Después simplemente se extrae la matriz esencial, seguido de la rotación y traslación y cálculo de la escala. Finalmente se actualiza la posición y orientación haciendo uso de Ecuación 4.3 y Ecuación 4.6

4.2.6 Interfaz Gráfica

El funcionamiento de la parte gráfica, directorio *web*, es muy sencillo en este caso, simplemente ha de representar la imagen, y las poses en la escena 3D.

Algorithm 6 Pseudocódigo interfaz gráfica

1: **if** llega un mensaje nuevo **then**
2: actualizar escenario 3D
3: actualizar imagen
4: **end if**

```
socket.onmessage = async function (event) {
    let data = JSON.parse(event.data);

    frame.rotation.x = data.pose.yaw
    frame.rotation.y = data.pose.pitch
    frame.rotation.z = data.pose.roll
    frame.position.x = data.pose.x * 1.0
    frame.position.y = data.pose.y * 1.0
    frame.position.z = data.pose.z * 1.0
    tracker.push(frame.position.clone())

    frameGt.rotation.x = data.poseGt.yaw
    frameGt.rotation.y = data.poseGt.pitch
    frameGt.rotation.z = data.poseGt.roll
    frameGt.position.x = data.poseGt.x * 1.0
    frameGt.position.y = data.poseGt.y * 1.0
    frameGt.position.z = data.poseGt.z * 1.0
```

```
    trackerGt.push(frameGt.position.clone())

    if (!(track === [])) {
        scene.remove(track)
        track = createTrack(tracker, USER_COLOR)
        trackGT = createTrack(trackerGt, GT_COLOR)
        scene.add(track)
        scene.add(trackGT)
    }

    // Parse Image Data
    var image_data = data.img_data,
        source = await decode_utf8(image_data.image),
        shape = image_data.shape;

    image_canvas.src = '' + source;
    image_canvas.width = shape[1];
    image_canvas.height = shape[0];
};
```

Código 4.8: Actualización de la interfaz gŕafica

Donde *frame* y *frameGt* son los vehículos o cámaras que usan los resultados del algoritmo y de *groundtruth* respectivamente.

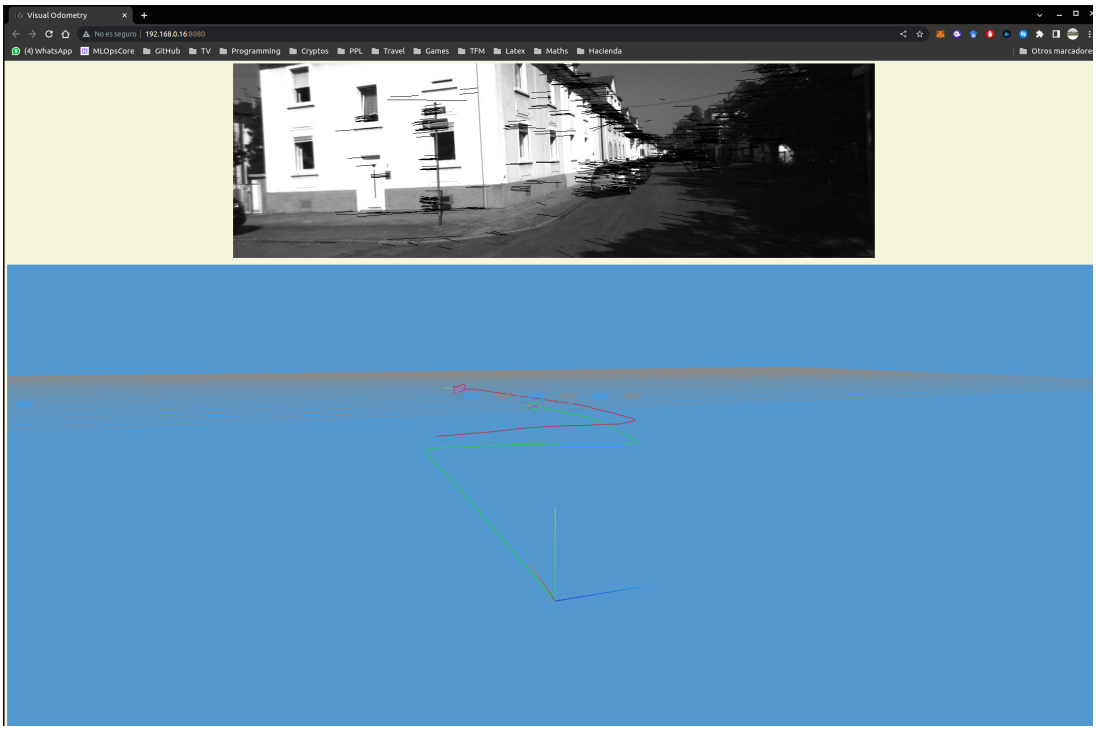


Figura 4.3: Interfaz gráfica de la solución de referencia

Aquí se puede ver un vídeo ilustrativo de la solución de referencia: https://youtu.be/nQrEuxnBdTM

4.3 Tests y Validación Experimental

Se han llevado a cabo varios tests para validar el correcto funcionamiento del algoritmo de odometría visual 3D. Estos tests se han dividido en dos categorías: tests de unidad y tests visuales.

Los tests de unidad se han enfocado en verificar el correcto funcionamiento de aquellos elementos relacionados con las matrices de rotación y ángulos de Euler, que son esenciales en el algoritmo. Estos tests se han escrito utilizando la librería `pytest` de Python y se han ejecutado mediante el comando `pytest`. Además, se ha creado un *workflow* de GitHub que permite ejecutar estos tests antes de aceptar un parche (*pull request*) en la rama principal de desarrollo.

Por otro lado, los tests visuales se han enfocado en verificar el funcionamiento correcto del algoritmo en su conjunto, sin preocuparse tanto por la precisión, ya que no es el objetivo principal del algoritmo. Para ello, se han utilizado diversos conjuntos de datos de prueba que incluyen imágenes y datos de posición y orientación obtenidos en situaciones reales. Estos resultados se han comparado cuantitativamente con los datos de verdad para evaluar el desempeño del algoritmo.

Además, se ha llevado a cabo una validación experimental utilizando un enfoque cualitativo visual. En este caso, se ha comparado la trayectoria estimada por el algoritmo con la trayectoria real del vehículo obtenida a partir de los datos de *ground truth*. Aunque este método no es tan preciso, es suficiente para evaluar si el algoritmo es capaz de seguir correctamente la trayectoria del vehículo. Además, no es necesario que el algoritmo sea completamente preciso, sino que simplemente debe ser capaz de seguir de manera razonable la trayectoria del vehículo.

A continuación, se muestran los resultados obtenidos del algoritmo utilizando diferentes conjuntos de datos:

4.3.1 Tests sobre dataset KITTI

Se ha probado el algoritmo programado en el dataset KITTI en las escenas disponibles desde la primera hasta la tercera. En la Figura 4.4 se puede observar que el algoritmo funciona bien en el plano por el que circula el vehículo.

Sin embargo, la herramienta visual de test puede resultar insuficiente para evaluar completamente el algoritmo en toda la dimensión de la escena, como se muestra en la Figura 4.6.

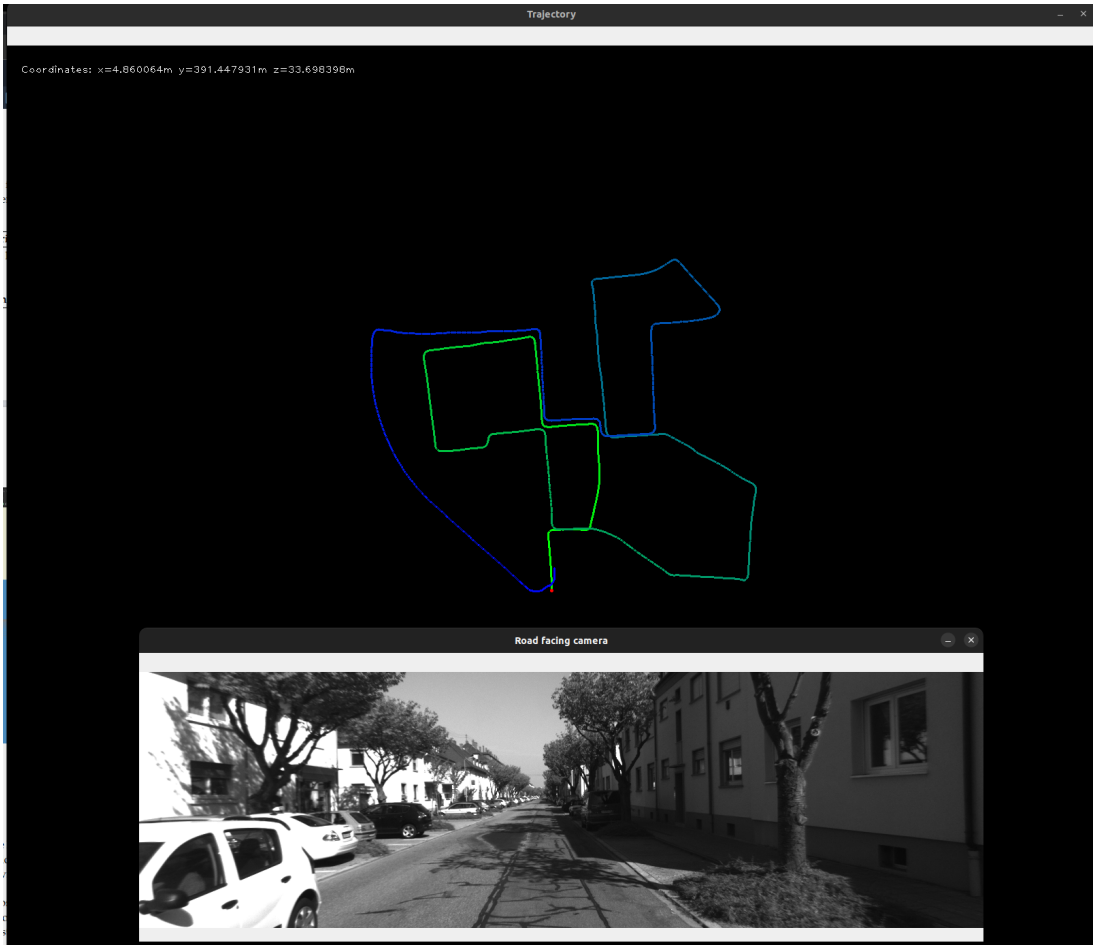


Figura 4.4: Plano azimutal del algoritmo sobre la primera escena de KITTI

Para verificar la precisión del algoritmo se ha optado por la medición del *error máximo* y del *error al final de la escena* tanto en plano, es decir en el suelo, como en el espacio tridimensional.

Para las escenas primera a tercera de KITTI, se han obtenido los siguientes resultados cuantitativos:[3]

[3]Para ver datos completos: https://github.com/RoboticsLabURJC/2020-tfm-pablo-asensio/tree/develop/metrics

Escena y Tipo de Error	Error 2D [m]	Error 3D [m]
Primera escena: Final de la escena	18,06	400,39
Primera escena: Máximo	18,06	400,39
Primera escena: Promedio	9,57	195,72
Segunda escena: Final de la escena	350,80	376,25
Segunda escena: Máximo	352,93	378,00
Segunda escena: Promedio	143,75	160,34
Tercera escena: Final de la escena	115,66	379,09
Tercera escena: Máximo	115,66	379,09
Tercera escena: Promedio	47,53	165,57

Tabla 4.1: Medición del error en escenas de KITTI

Estas variaciones en el error sobre el plano y el volumen se deben la problema de la *deriva*, que es natural a este tipo de algoritmos clásicos y que se aprecia muy bien en el vídeo[4] de la solución de referencia. Pero se puede afirmar que *la implementación del algoritmo es correcta*. Asimismo se puede observar que en las escenas primera y tercera, ambas en entorno urbano, que este algoritmo funciona mejor, ya que a la hora de extraer características, pueden ser estas más favorables para un buen seguimiento. También puede favorecer a que haya menor error el hecho que estas secuencias sean cerradas (fin en el inicio). En cambio, en la segunda escena, abierta, se observa que hay un mayor error sobre el plano en promedio. Esto puede ser ya que se trata de un entorno de carretera donde las características extraídas no sean de buena calidad.

[4] https://youtu.be/nQrEuxnBdTM

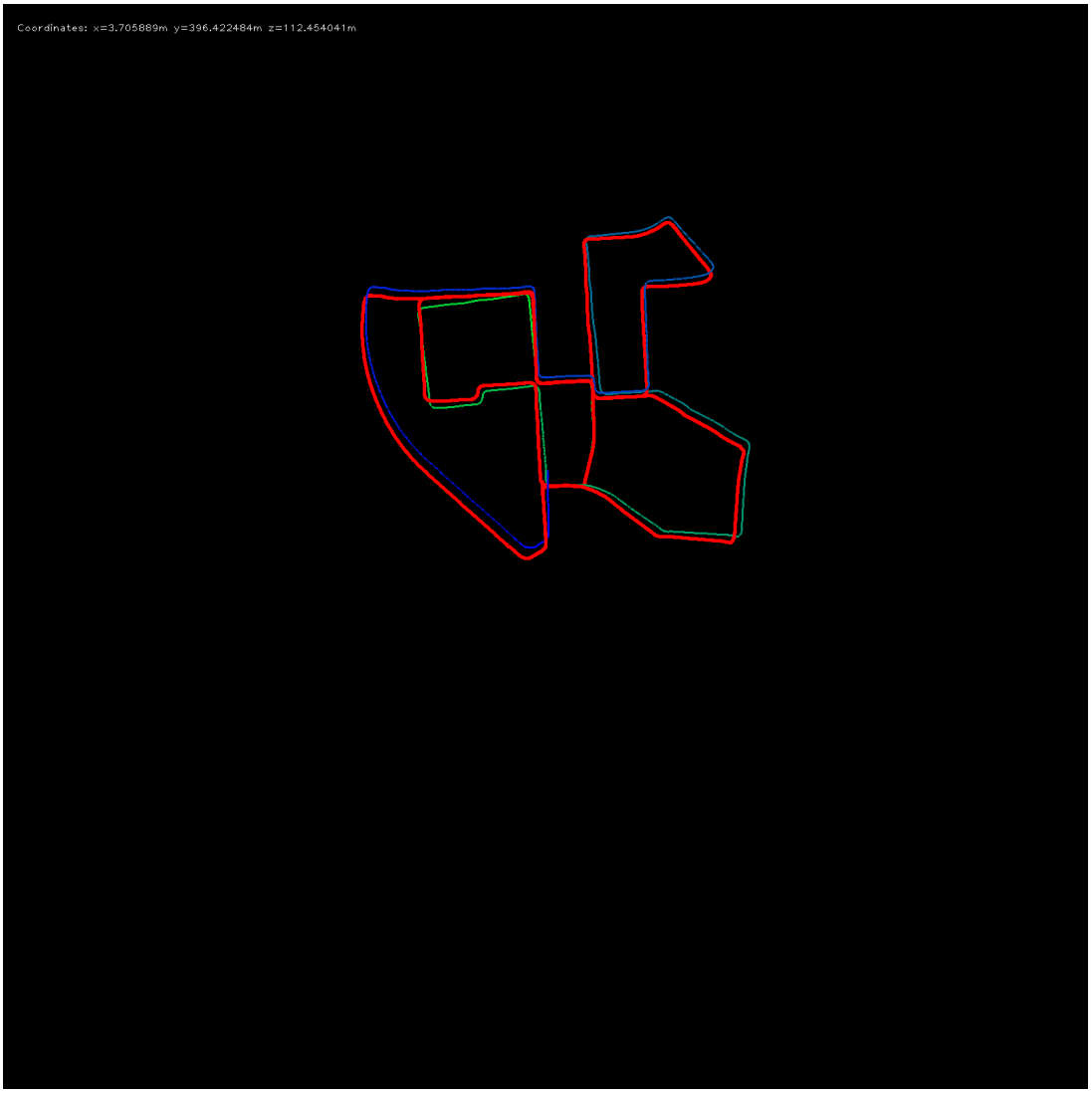


Figura 4.5: Primera escena de KITTI con *groundtruth*

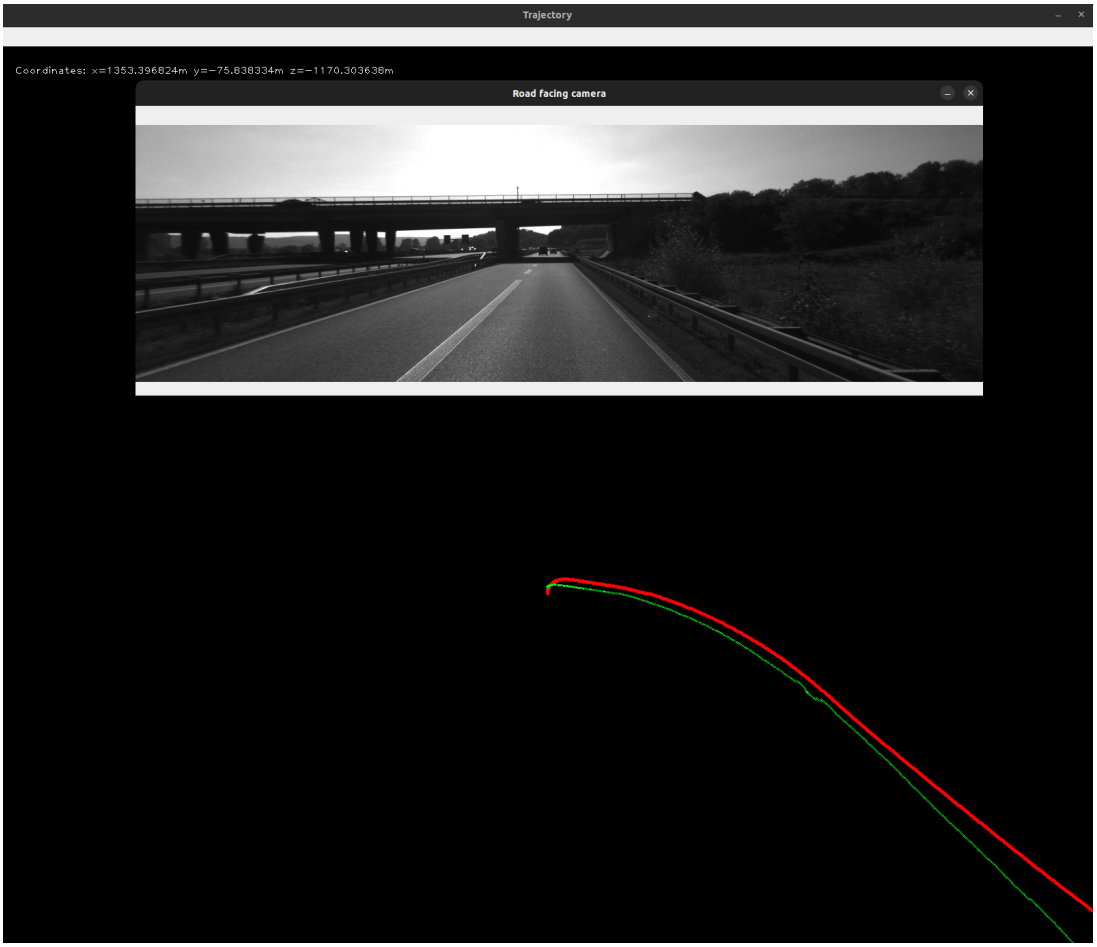


Figura 4.6: Segunda escena de KITTI

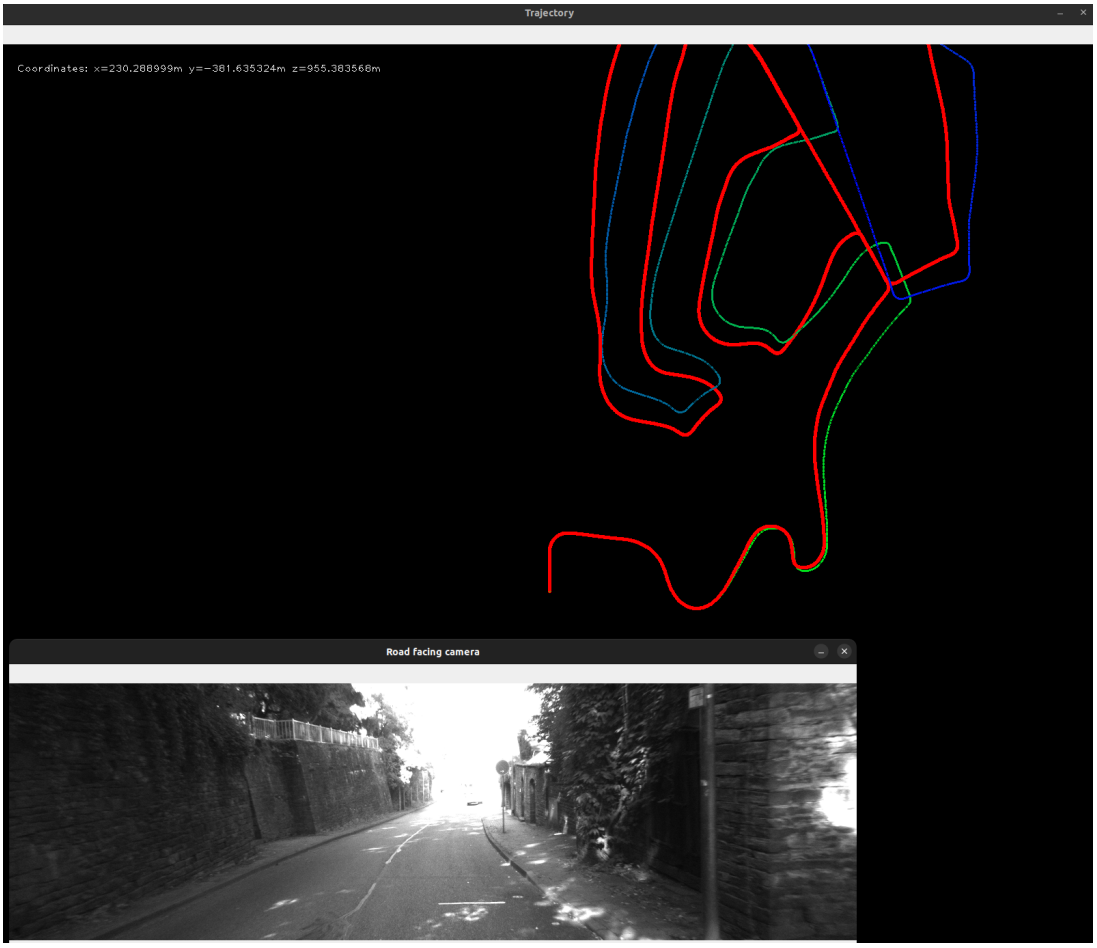


Figura 4.7: Tercera escena de KITTI

4.3.2 Tests sobre dataset VKITTI2

También se ha probado el algoritmo en el dataset VKITTI2, específicamente en la vigésima escena. Aunque este dataset no está destinado a la odometría visual, proporciona secuencias de escenas sintéticas extraídas de KITTI.

En la Figura 4.8 se observa que el inicio de la secuencia es bueno, pero a medida que avanza, no es tan preciso, esto se debe que el error en este tipo de algoritmos es de carácter *acumulativo*, es decir, que hace que el error se incremente de una forma muy rápida por cada iteración. Además también se trata de una escena abierta en carretera.

En esta secuencia se han obtenido unos errores de:

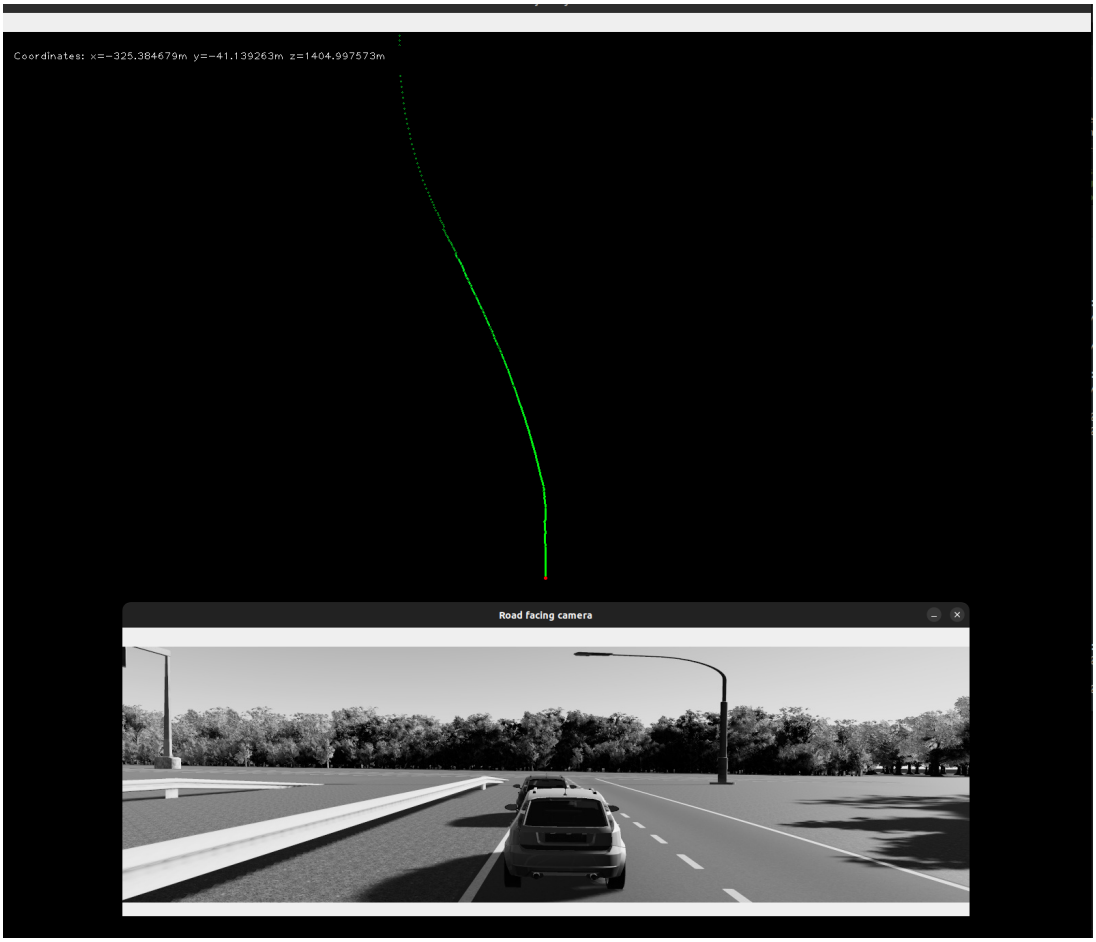


Figura 4.8: Plano azimutal del algoritmo sobre VKITTI2

Tabla 4.2: Medición del error en la vigésima escena de VKITTI2

Tipo de Error	**Error 2D [m]**	**Error 3D [m]**
Final de la escena	755,34	822,67
Máximo	763,62	832,33
Promedio	188,51	394,34

4.3.3 Tests sobre EuRoC MAV

Al intentar probar el algoritmo en la escena *MH_01_easy*, se observa que no funciona correctamente, como se puede apreciar en la Figura 4.9. El algoritmo no muestra desplazamiento y permanece estático en toda la escena de interior. Por lo tanto, se descarta utilizar este dataset, EuRoC MAV, en la versión de Unibotics.

Conviene distinguir entre los errores del algoritmo y errores de la programación y fallos en la secuencia de datos. En esta validación se han ratificado problemas estructurales del

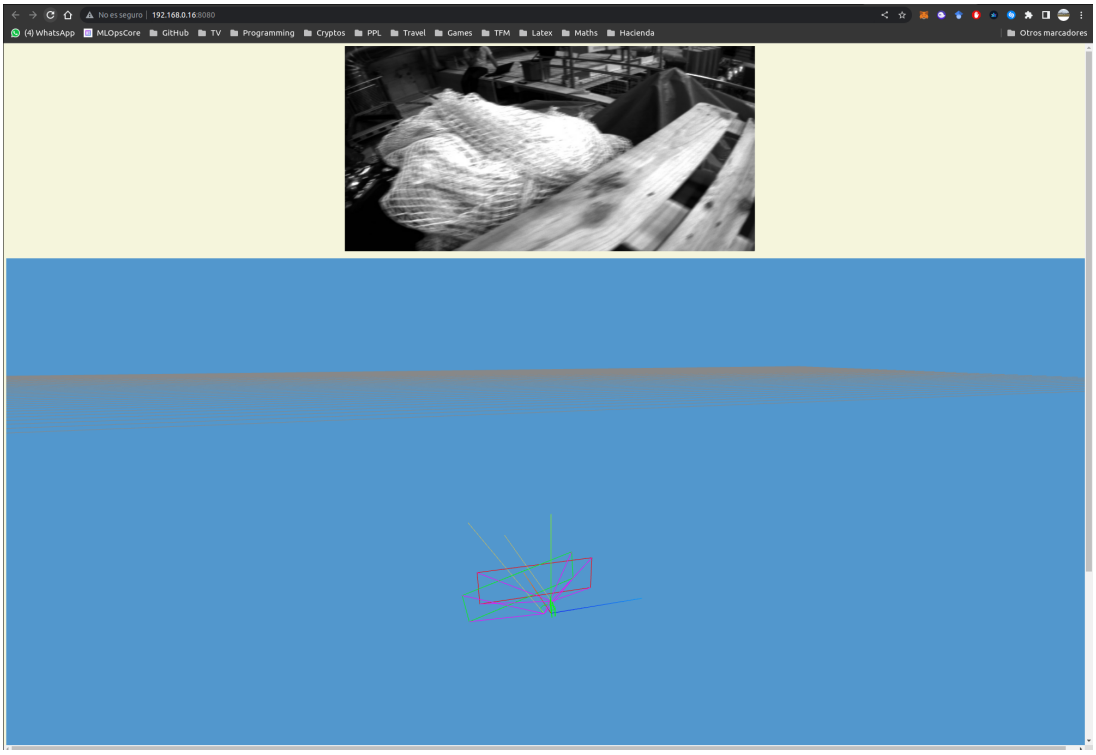


Figura 4.9: Visor web del algoritmo sobre EuRoC MAV

algoritmo de odometría visual 3D, como la deriva. Se ha verificado que la programación realizada es correcta (funciona satisfactoriamente en entornos reales **no** complejos). Y se ha contrastado que hay secuencias en las cuales el algoritmo tiene muchas limitaciones (como en las de EuRoCMAV). También ha permitido seleccionar el dataset real **no** complejo a utilizar en la enseñanza del algoritmo clásico de odometría visual 3D, siendo esta escena la primera de KITTI, ya que es el dataset y secuencia que menor error muestra sobre el plano; cosa que considero de especial importancia para la transmisión de conocimiento al ser más fácil de visualizar mentalmente y, por ende, de comprender.

La Figura 4.10 pretende captar la mala estimación del algoritmo en este *dataset*. En verde se ve la trayectoria oscilatoria (arriba y abajo) del *groundtruth* y en rojo, la del algoritmo, se queda inmóvil.

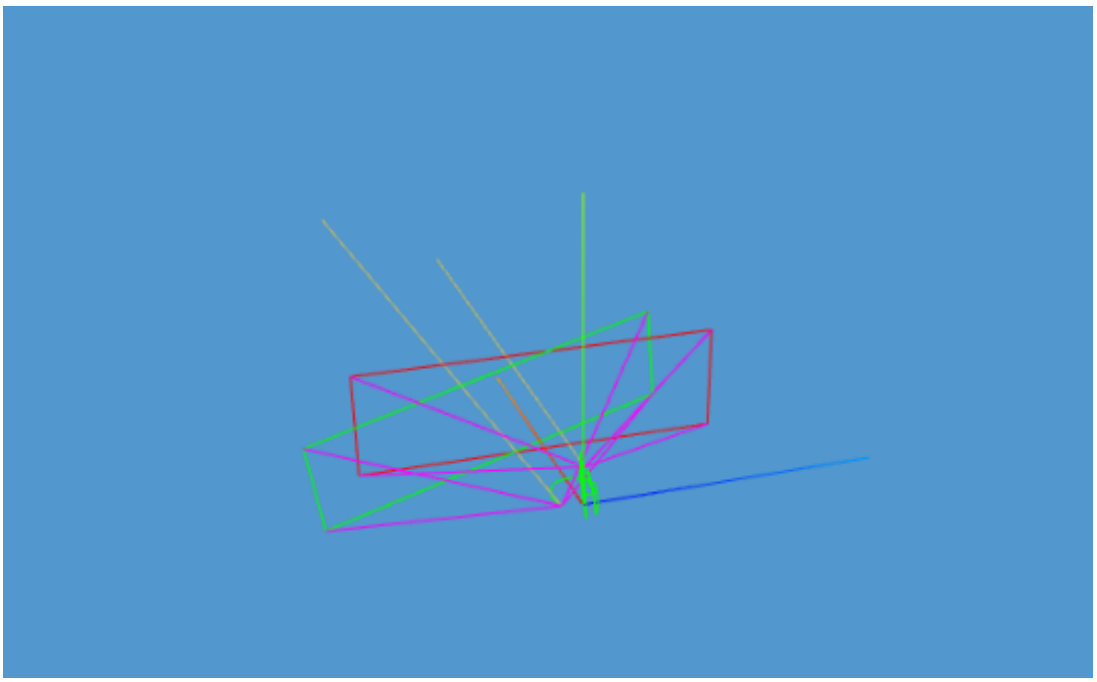


Figura 4.10: Zoom del visor web del algoritmo sobre EuRoC MAV

Capítulo 5

Integración en la Plataforma Unibotics

En este capítulo se describe el proceso de integración de la aplicación de odometría visual 3D en la plataforma educativa Unibotics. Se explica cómo se ha adaptado la solución de referencia para poder ser utilizada como un ejercicio más por los estudiantes de forma sencilla y cómo se ha integrado en la plataforma de manera que sea accesible desde ella.

Una vez desarrollado y validado el algoritmo de odometría visual 3D en el capítulo anterior, el siguiente paso es integrarlo en la plataforma web Unibotics como un ejercicio más de visión artificial. Para ello, se ha llevado a cabo una serie de adaptaciones tanto del código del algoritmo como de la interfaz web que se encarga de visualizar los resultados.

Primero se proporciona un editor de código fuente integrado para que los alumnos puedan escribir y ejecutar su propio algoritmo de odometría visual 3D a través de una *API* proporcionada en el enunciado del ejercicio.

Segundo, se incorpora el *GUI* desarrollado como *frontend* del ejercicio para visualizar la trayectoria 3D estimada y la verdadera. Se ha añadido un campo en la interfaz de la actividad que ofrece una métrica de evaluación del algoritmo, basada en la distancia en el plano entre la trayectoria estimada por el algoritmo del estudiante y la trayectoria real del vehículo. Esta métrica permite a los alumnos evaluar el rendimiento de su algoritmo y compararlo con el de otros compañeros.

Con esta integración, se espera que los alumnos puedan poner en práctica los conocimientos adquiridos durante el curso y desarrollar sus propias soluciones a problemas reales de odometría visual 3D.

Antes de explicar cómo se realiza la creación de un nuevo ejercicio, se pasa a explicar la arquitectura software de Unibotics.

Tercero, se ejecuta el código fuente del estudiante usando como fuente de imágenes la

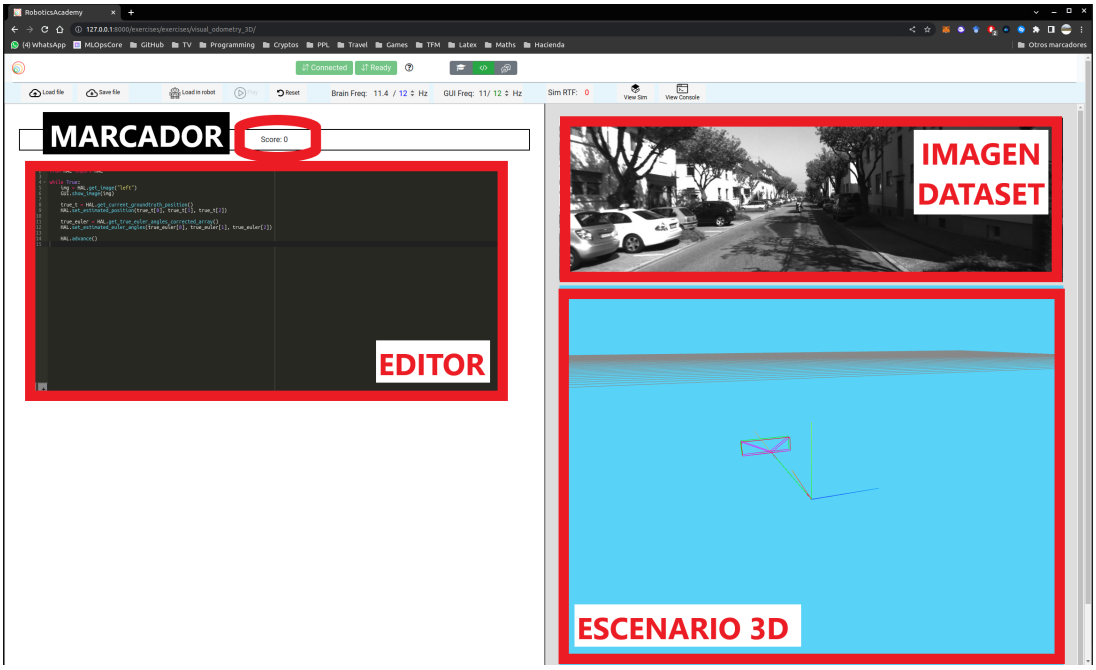


Figura 5.1: Ejercicio de Odometría Visual en Unibotics

secuencia *00* del dataset de KITTI, tal y como se explicó en el capítulo anterior.

En la Figura 5.1 se puede apreciar dos partes claramente diferenciadas de forma vertical. La que se encuentra en la mitad izquierda se trata del editor, donde el estudiante escribe su código fuente, y, a la derecha, el visor *GUI*, donde se muestra la trayectoria del algoritmo del estudiante y la verdad en la mitad inferior de esta, y en la mitad superior, la imagen de la secuencia.

5.1 Arquitectura Software de la Plataforma Unibotics

La plataforma Unibotics, al igual que Robotics Academy, cuenta con un servidor web especialmente diseñado para ofrecer ejercicios de programación de robots, denominado Robotics Academy Docker Image (RADI). Este servidor permite el uso de varios ejercicios que utilizan Gazebo y STDR, y ofrece un puerto websocket (8765) y un protocolo de comunicación (Robotics Academy Manager Protocol, o RAMP) para solicitar e interactuar con estos ejercicios. Cada ejercicio abre 1, 2 o más websockets para interactuar específicamente con el ejercicio y recibir datos.

El RAMP incluye los siguientes comandos:

- *open* para iniciar un ejercicio especificado en el campo *ejercicio*

- *stop* para detener la simulación robótica
- *resume* para reanudar la simulación robótica
- *reset* para reiniciar la simulación robótica
- *evaluate* para solicitar una evaluación sintáctica del código enviado en el campo *código*
- *startgz* para abrir el visualizador GZClient
- *stopgz* para cerrar el visualizador GZClient
- *Ping* o *PingDone* para enviar mensajes Ping y comunicar que una orden ha sido ejecutada (después de los comandos resume, reset o stop)

Cada ejercicio está compuesto por una página web (`exercise.html`) y una plantilla de Python (`exercise.py`). El `exercise.py` se ejecuta dentro del RADI, mientras que el `exercise.html` proviene del navegador. Ambos se comunican a través de *websockets*: canales de comunicación bidireccionales que permiten la comunicación entre diferentes lenguajes de programación. Cada websocket de ejercicio (típicamente uno para la GUI y otro para el cerebro del robot) tiene su propio protocolo. Los primeros cinco caracteres se utilizan para identificar el tipo de mensaje.

5.1.1 Comunicación entre Backend Robótico y Frontend

La comunicación entre el *backend* robótico y el *frontend* consta de estos elementos:

- Websocket manager: se encarga de solicitar los ejercicios y controlar la simulación
- Websocket código: interactúa con el cerebro del robot
- Websocket GUI: recibe datos del robot
- GZClient VNC: interactúa y visualiza la simulación
- Console VNC: muestra mensajes de depuración y salida.

Websocket Código y Websocket GUI

El protocolo de comunicación entre la plantilla de Python, `exercise.py`, que se ejecuta en el servidor RADI, y el navegador web que utiliza el alumno para interactuar con el ejercicio, incluye dos websockets: el websocket de código, que se utiliza para enviar el código

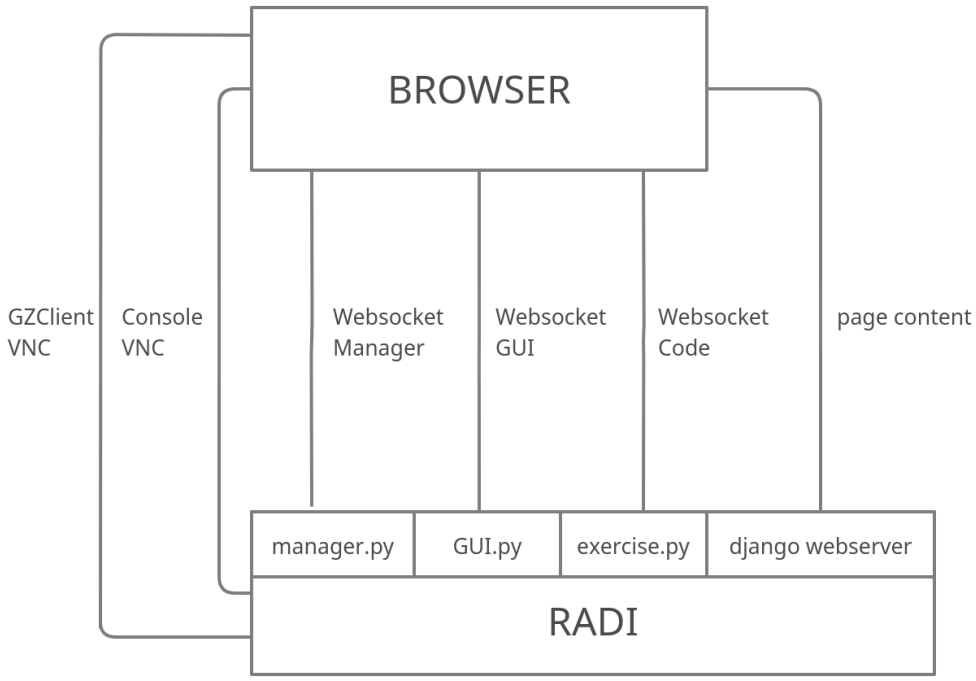


Figura 5.2: Comunicación entre el backend y el frontend

fuente del alumno al ejercicio y controlar su ejecución; y el websocket GUI, que se utiliza para enviar datos y imágenes desde el backend (`exercise.py`) al frontend (navegador web).

El websocket de código tiene un protocolo estandarizado que incluye los siguientes comandos:

- *#freq*: para establecer la frecuencia de ejecución del código del alumno y del GUI
- *#code*: para actualizar el código del alumno en el ejercicio
- *#play*: para iniciar la ejecución del código
- *#stop*: para detener la ejecución
- *#reset*: para detener y reiniciar la ejecución
- *#ping*: para enviar mensajes de ping

El servidor responde con los mensajes *#exec* cuando se ha cargado el último código enviado con *#code*, *#freq* en cada iteración del código con los campos *brain* (frecuencia del código del alumno), *gui* (frecuencia del GUI) y *rtf* (factor de tiempo real), y *#ping* como respuesta a los mensajes de ping.

El *websocket GUI*, por otro lado, se utiliza para enviar datos y imágenes desde el backend al frontend. Este websocket varía según las necesidades de cada ejercicio, pero suele incluir campos como #*image* con imágenes obtenidas de la cámara del robot y #*map* con la posición y rotación del robot.

Protocolo entre manager.py y el Navegador

El *websocket del manager* se encarga de solicitar los ejercicios y controlar la simulación (en el puerto 6080, escuchando al servidor VNC en el puerto 5900). Por ejemplo, inicia/detiene/pausa/reanuda/mata la simulación de Gazebo del ejercicio elegido. También inicia el servidor VNC. El código escrito por el usuario se envía primero desde el navegador al proceso `manager.py` a través de los websockets del manager. Luego, `manager.py` comprueba el código con Pylint y devuelve el resultado al navegador. Puede manejar los siguientes comandos:

- *Open*: Mata la simulación anterior y comienza una nueva dependiendo del ejercicio elegido y si la simulación acelerada está habilitada o no.
- *Resume*: Despausa la física de Gazebo.
- *Stop*: Pausa la física de Gazebo.
- *Evaluate*: Comprueba el código del usuario enviado a manager.py y devuelve una matriz vacía si no hay errores.
- *Evaluate_Style*: Comprueba el código del usuario enviado a manager.py y devuelve una matriz vacía si no hay errores. En este caso, también devuelve advertencias.
- *Start*: Despausa la física de Gazebo.
- *Reset*: Tipo de reinicio = predeterminado. Si el ejercicio está incluido en el array DRONE_EX, mata el exercise.py y reinicia el drone. Si el ejercicio está incluido en HARD_RESET_EX, se requiere tirar todo y volver a construirlo.
- *Soft reset*: Tipo de reinicio = suave. En este caso, ya sea que el ejercicio esté incluido en el array DRONE_EX o en el HARD_RESET_EX, el comportamiento es el mismo: se pausa y se reinicia la física.
- *Stopgz*: Detiene el cliente de Gazebo.
- *Startgz*: Configura el ancho y el alto de la pantalla del navegador para el gzclient. También inicia el cliente Gazebo.

5.1.2 Procesamiento de Código del Usuario

La procesamiento del código del usuario sigue los siguientes pasos:

1. El código es enviado desde el editor ACE del navegador al proceso `manager.py` del RADI.
2. El código es revisado por Pylint y los errores son devueltos al navegador. Si el navegador recibe un error, éste se muestra en un modal y el código no se envía al cerebro del robot.
3. Si no hay errores, el código es enviado desde el navegador al archivo `exercise.py` a través del websocket de código. `exercise.py` recibe el código fuente del usuario como texto bruto y lo pone en funcionamiento.
4. El código del usuario es separado en dos partes: la parte secuencial (ejecutada una vez) y la parte iterativa (ejecutada cada intervalo de tiempo del cerebro). El código es separado por el primer bucle *while True* encontrado. Dentro de la parte iterativa, el cerebro mide el tiempo después de cada iteración, lo que se llama gestión del código, para no sobrecargar el CPU y mantener un ritmo controlado de iteraciones por segundo para dejar el CPU libre para otras tareas del navegador. La parte iterativa se inserta en otra plantilla junto con código adicional que controla las iteraciones por segundo que se realizan (esqueleto computacional), de modo que este motor computacional está vinculado al código del usuario para garantizar que el código se ejecute a una frecuencia nominal. El código del usuario también se enriquece con algunos elementos de control de ejecución para pausar, reiniciar y cargar un nuevo código en el cerebro del robot.

5.2 Creación de un Nuevo Ejercicio de Odometría Visual

Para incluir un nuevo ejercicio en la plataforma se siguen los siguientes pasos:

1. Añadir la carpeta con el contenido del ejercicio en `exercises/static/exercises` siguiendo las convenciones de nombres de archivos.
2. Crear una entrada en `db.sqlite3`. Una forma sencilla de hacerlo es a través de la página de Django admin:
   (a) Ejecutar `python3.8 manage.py runserver`.
   (b) Acceder a `http://127.0.0.1:8000/admin/` en un navegador y inicia sesión con ”user” y ”pass”.

(c) Hacer clic en "add exercise" y rellena los campos: id de ejercicio (nombre de la carpeta), nombre (nombre que se mostrará), estado, idioma y descripción (descripción que se mostrará). Guarda y sal.

3. Hacer la confirmación de cambios (*commit*) de los cambios en `db.sqlite3`.

4. Editar el archivo `manager.py` e `instructions.json` si es necesario.

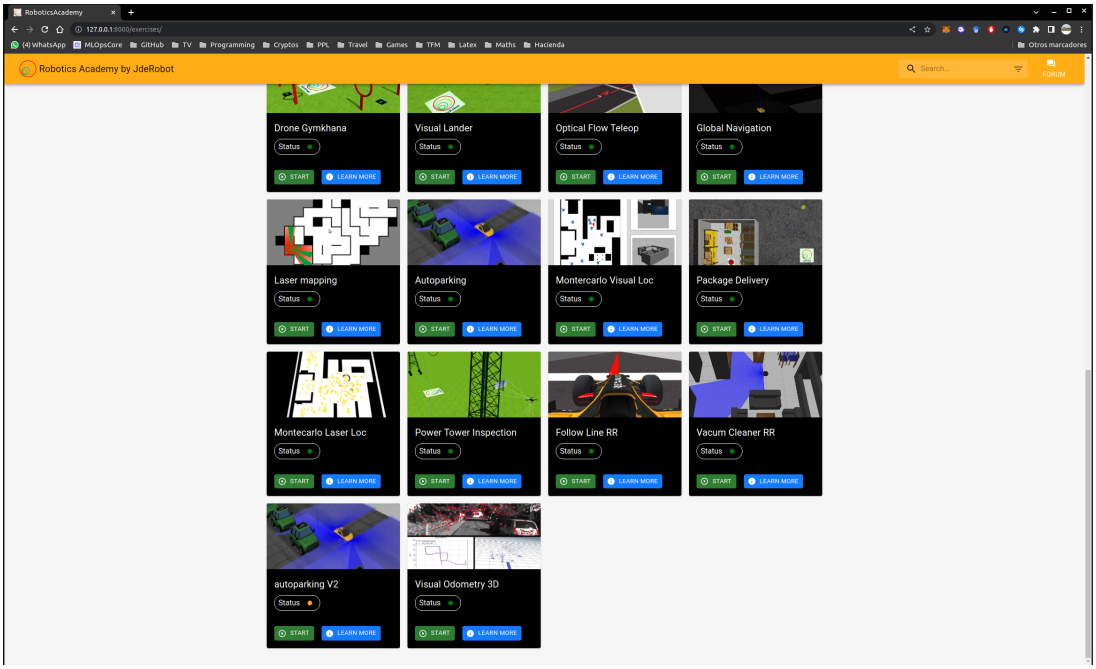


Figura 5.3: Ejercicio de odometría visual en el selector de ejercicios

Para realizar la integración del ejercicio en la plataforma Unibotics, es necesario incluir los archivos `hal.py`, `gui.py`, `exercise.py` en la ruta *exercises/static/exercises/tu_ejercicio/* y `exercise.html`, en *exercises/templates/exercises/tu_ejercicio/*. Estos archivos son necesarios para que el ejercicio se integre correctamente en la plataforma y pueda ser ejecutado por el alumno a través de la interfaz web.

Es aconsejable basarse en uno de los ejercicios existentes, como ha sido en nuestro caso *3d_reconstruction*, para facilitar la creación del ejercicio. De esta forma, podremos ver cómo están estructurados estos archivos y cómo debemos adaptarlos a nuestro ejercicio de odometría visual.

5.2.1 Archivos Principales

hal.py

El archivo `hal.py` contiene la clase HAL, que es la encargada de gestionar los datos como imágenes, posiciones del *groundtruth*, etc. Esta clase cuenta con variables privadas y métodos que pueden ser expuestos a la API del ejercicio. De esta forma, se controla qué información se le da al alumno y cómo puede utilizarla en su algoritmo. Por ejemplo, la clase HAL puede tener una variable privada que almacene el estado del vehículo, y un método que permita al alumno obtener la información de este estado a través de la API. De esta forma, el alumno puede utilizar la información del estado del vehículo en su algoritmo para calcular la odometría visual 3D.

gui.py

El archivo `gui.py` se encarga de la comunicación entre el *backend* robótico del ejercicio y la interfaz web que ve el alumno a través de un websocket. Esto se realiza mediante la clase GUI, que se encarga de inicializar y gestionar el websocket, y de enviar y recibir mensajes a través de él. Además, esta clase también se encarga de actualizar la interfaz web del alumno con la información que se envía desde el backend. Es importante tener en cuenta que toda la información que se envía o recibe a través del websocket debe estar en formato JSON.

exercise.py

El archivo `exercise.py` es el encargado de gestionar el servidor del ejercicio, es decir, se encarga de exponer la API del ejercicio a través de la cual el alumno podrá interactuar con el ejercicio. También se encarga de analizar el código del alumno, es decir, de comprender el código escrito por el alumno y hacer que sea ejecutable por el ejercicio.

Además, este archivo se encarga de definir la frecuencia de funcionamiento del ejercicio, es decir, la velocidad a la que se ejecutará en cada iteración del bucle infinito del código del alumno. Esto es importante ya que en algunos casos el ejercicio puede consumir demasiados recursos de la máquina del alumno, y así se puede reducir dicho consumo de recursos.

Resumiendo, el archivo `exercise.py` es el encargado de gestionar el servidor del ejercicio y hacer que el código del alumno sea ejecutable de forma correcta.

exercise.html

El archivo `exercise.html` es el encargado de la parte del frontend que se muestra al alumno. Se encarga de generar la interfaz gráfica a través de la cual el alumno puede interactuar con el ejercicio. Además, este archivo se encarga de calcular el error en la estimación 3D de la posición que hay en el algoritmo del alumno y de realizar la conversión al sistema de puntos. El sistema de puntos es una medida de la calidad del algoritmo del alumno, y se consiguen mejores resultados cuanto menor sea la puntuación obtenida.

Para la realización de página web se ha utilizado en gran parte el visor web que se desarrolló para lo solución de referencia en la subsección 4.2.6 (Ver Figura 4.3 y Figura 5.1).

Cabe destacar que la lectura de imágenes se realiza leyendo las imágenes de forma secuencial, sin precargarlas como se había realizado en la solución de referencia. Y se almacenan en un directorio del alumno. Este directorio se tiene que pasar como argumento al RADI. Aquí un ejemplo:

```
docker run -it --rm --name RADI -v ~/datasets/:/datasets -p 7681:7681 -p 2303:2303 -p
    1905:1905 -p 8765:8765 -p 6080:6080 -p 1108:1108 -p 8000:8000 tfm
```

Código 5.1: Comando docker para la ejecución correcta del ejercicio de odometría visual

En Código 5.1 se ve que haciendo uno de la opción *-v* se indica que el directorio local */datasets* lo monte en la ruta */datasets* del RADI. Siendo *RADI* el nombre del contenedor y *tfm* la imágen personalizada utilizada en el desarrollo de este ejercicio.

5.3 Sistema de Evaluación Automática

La evaluación automática utilizada en este trabajo se basa en la asignación de una puntuación al código del alumno en función de la distancia entre la posición estimada por el algoritmo del estudiante y la posición real del vehículo, obtenida a partir de los datos de *groundtruth*. Esta puntuación se utiliza para determinar el grado de acierto del algoritmo y proporcionar una retroalimentación al alumno sobre el rendimiento de su código. La puntuación se asigna mediante una función sigmoide que penaliza de forma más severa a medida que la distancia entre la posición estimada y la real aumenta. De esta forma, se busca fomentar la implementación de algoritmos precisos por parte del alumno.

Para una distancia d, la puntuación s se rige por la siguiente ecuación:

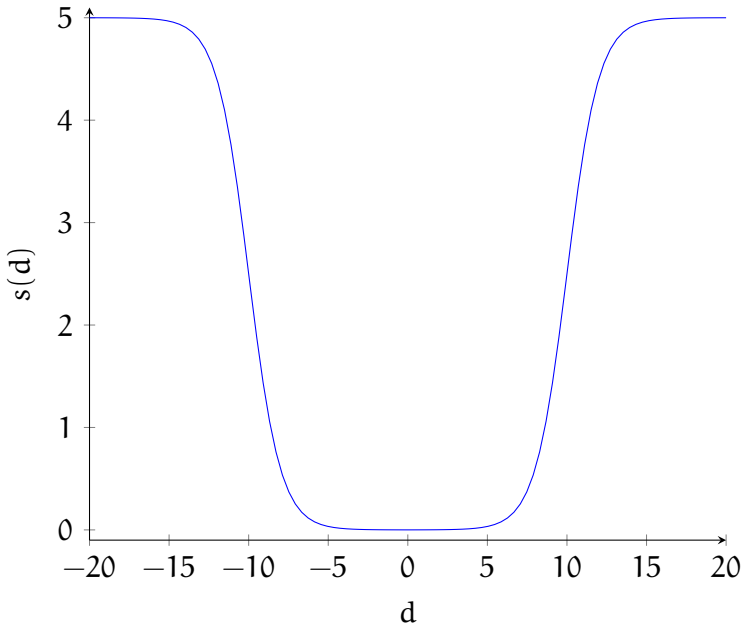
$$s(d) = \frac{5}{1 + e^{-(|d|-10)}} \tag{5.1}$$

Donde d es la distancia relativa entre la posición calculada y la real en el mismo instante de tiempo, y k es un parámetro que determina la pendiente de la función. Se ha elegido un valor de $k = 1$, lo que hace que la función tenga una pendiente bastante pronunciada cerca del valor de $d = 10$. De esta forma, se consigue que la puntuación caiga rápidamente a medida que la distancia d aumenta.

La puntuación total quedaría de la siguiente forma:

$$p = \sum_i s(d_i) \tag{5.2}$$

Donde p es la puntuación.



La corrección del algoritmo del alumno se realiza por cada fotograma, cada vez que se recibe el mensaje en el frontend desde el RADI. De esta forma, se puede evaluar en tiempo real la precisión del algoritmo de odometría visual 3D y proporcionar una retroalimentación inmediata al alumno. Esto permite que el alumno pueda ir mejorando su algoritmo de forma progresiva y obtener una mejor puntuación en el ejercicio. En la Figura 5.4 se puede ver un ejemplo ficticio en el que se observa que la puntuación ha incrementado ya que el error lo ha hecho.

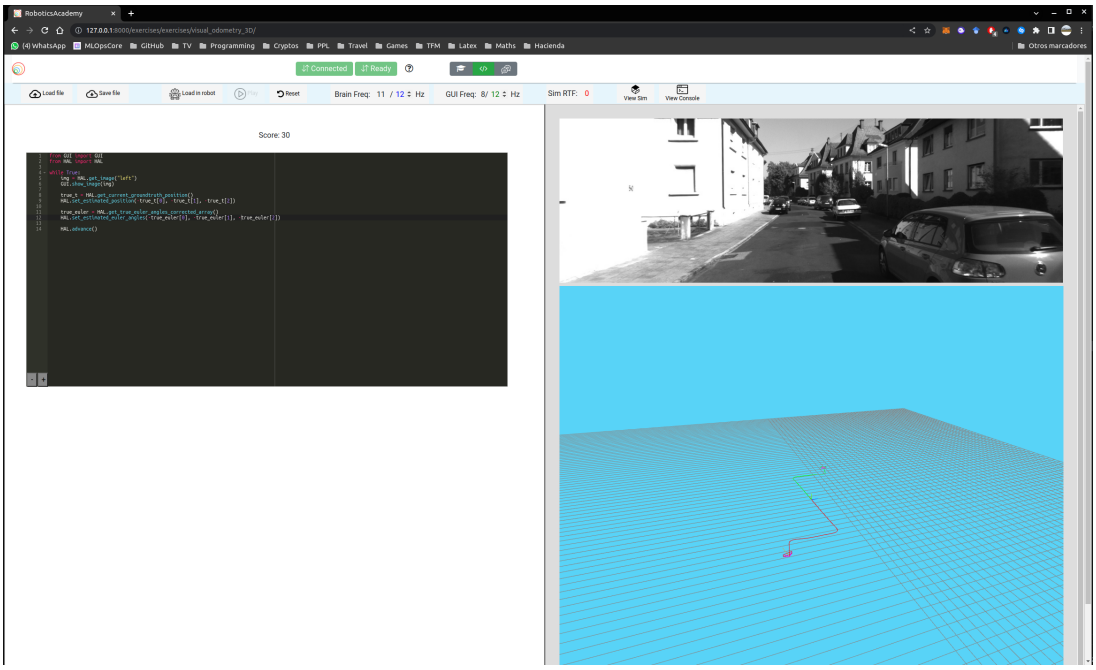


Figura 5.4: Ejemplo de evaluación automática

Capítulo 6

Conclusiones y Trabajos Futuros

En el presente trabajo, se ha desarrollado un ejercicio de odometría visual 3D para la plataforma educativa Unibotics. A través de este ejercicio, se ha ofrecido a los estudiantes una herramienta de aprendizaje práctica y didáctica, que les permita comprender y aplicar los conceptos teóricos de odometría visual de manera sencilla y amena.

6.1 Repaso de Objetivos

Una vez finalizado el ejercicio, podemos concluir que se han cumplido los objetivos propuestos en su desarrollo.

Repasamos los objetivos descritos en la Sección 2.1:

1. *Diseñar un ejercicio educativo que permita a los estudiantes crear y probar su propio algoritmo de odometría visual tridimensional en la plataforma Unibotics.*

2. *Proporcionar una interfaz visual para que el estudiante pueda ver los resultados de su algoritmo en tiempo real.*

3. *Implementar un sistema de puntuación para medir el rendimiento del algoritmo desarrollado por el estudiante.*

En primer lugar, se ha logrado diseñar un ejercicio educativo que permita a los estudiantes crear y probar su propio algoritmo de odometría visual tridimensional en la plataforma Unibotics. Para ello ha sido necesario desarrollar una solución de referencia donde probar el algoritmo de odometría visual 3D (Sección 4.2). El algoritmo de odometría visual

ha sido programado satisfactoriamente haciendo uso de un detector de puntos de interés (Fast, ver Código 4.5), haciendo un emparejamiento de las características (ver Código 4.2), calculando la matriz esencial con RANSAC y posteriormente calcular los incrementos en la posición y orientación de la cámara mediante las ecuaciones incrementales 4.3 y 4.6 (Código 4.7)

En segundo lugar, se ha desarrollado una interfaz web (Sección 4.2.6) sobre la cual el alumno puede ver por cada iteración, la trayectoria obtenida por su algoritmo y la verdadera. También se integró en Unibotics gran parte de este trabajo de la interfaz gráfica (Capítulo 5), optimizando tiempos de trabajo. Para ello fue necesario realizar un editor de texto, usar imágenes de diferentes *datasets*, y crear un escenario 3D donde poder visualizar la trayectoria estimada por el algoritmo del alumno y la verdadera del *dataset* (ver Código 4.8).

Además, se implementó un sistema de puntuación para medir el rendimiento del algoritmo desarrollado por el estudiante en tiempo real, es decir, por cada iteración del algoritmo (Sección 5.3).

Durante el desarrollo del ejercicio se han encontrado algunos problemas, como la dificultad para sincronizar la odometría visual con la situación del robot sobre la escena o la lectura de los *groundtruth* de los diferentes *datasets*, estos problemas han sido solucionados mediante la implementación de medidas adecuadas, como el uso de matrices de rotación y realizando traslaciones.

Se observó que el algoritmo clásico funciona mejor sobre entornos urbanos en los que se empieza y termina la secuencia en la misma posición.

También se grabó un vídeo ilustrativo de la solución de referencia: `https://youtu.be/nQrEuxnBdTM`

6.2 Líneas Futuras

En cuanto a los trabajos futuros, se podría considerar la posibilidad de ampliar el ejercicio con nuevas funcionalidades y opciones de configuración, con el objetivo de hacerlo aún más completo y atractivo para los estudiantes. También se podría explorar la posibilidad de integrar el ejercicio con otras tecnologías y plataformas, con el fin de expandir su alcance y beneficios para la comunidad educativa. También se podría modificar el ejercicio para que trabajase con diferentes *datasets*, y crear nuevos ejercicios con otros algoritmos de autolocalización visual sobre los mismos *datasets*, como localización probabilística (Montecarlo) o VisualSLAM (PTAM o MonoSLAM, SVO, ...).

En resumen, la creación de este ejercicio de odometría visual en la plataforma Unibotics ha supuesto un éxito en la consecución de sus objetivos, y abre la puerta a futuros trabajos y mejoras que permitan seguir ofreciendo a los estudiantes una herramienta de aprendizaje de calidad y valor.

Referencias

[1] Víctor Arribas Raigadas. «Análisis de algoritmos de VisualSLAM: un entorno integral para su evaluación». https://gsyc.urjc.es/jmplaza/students/tfm-visualslam_evaluation-victor_arribas-2016.pdf. Tesis de mtría. Universidad Rey Juan Carlos, 2016.

[2] Elías Barcia Mejias. «Herramienta de evaluación cuantitativa de algoritmos Visual SLAM». https://gsyc.urjc.es/jmplaza/students/tfm-visualslam_slamtestbed-elias_barcia-2019.pdf. Tesis de mtría. Universidad Rey Juan Carlos, 2019.

[3] Yohann Cabon, Naila Murray y Martin Humenberger. *Virtual KITTI 2*. 2020. arXiv: 2001.10773 [cs.CV].

[4] John Canny. «A computational approach to edge detection». En: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), págs. 679-698.

[5] José M Cañas y col. «A ROS-based open tool for intelligent robotics education». En: *Applied Sciences* 10.21 (2020), pág. 7419.

[6] Andrew J Davison y col. «MonoSLAM: Real-time single camera SLAM». En: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), págs. 1052-1067.

[7] Hugh Durrant-Whyte y Tim Bailey. «Simultaneous localization and mapping: part I». En: *IEEE robotics & automation magazine* 13.2 (2006), págs. 99-110.

[8] Olivier D Faugeras. «What can be seen in three dimensions with an uncalibrated stereo rig?» En: *European conference on computer vision*. Springer. 1992, págs. 563-578.

[9] Martin A Fischler y Robert C Bolles. «Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography». En: *Communications of the ACM* 24.6 (1981), págs. 381-395.

[10] Andreas Geiger, Philip Lenz y Raquel Urtasun. «Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite». En: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012.

[11] Chris Harris, Mike Stephens y col. «A combined corner and edge detector». En: *Alvey vision conference*. Vol. 15. 50. Citeseer. 1988, págs. 10-5244.

[12] *Huangying-Zhan/kitti-odom-eval: KITTI Odometry Evaluation Toolbox - Github*. https://github.com/Huangying-Zhan/kitti-odom-eval.

[13] Georg Klein y David Murray. «Parallel tracking and mapping for small AR workspaces». En: *2007 6th IEEE and ACM international symposium on mixed and augmented reality*. IEEE. 2007, págs. 225-234.

[14] Ignacio San Román Lana. «Odometría visual 3D para autolocalización de una cámara móvil en tiempo real». https://gsyc.urjc.es/jmplaza/students/tfm-visualodometry-isanroman-2015.pdf. Tesis de mtría. Universidad Rey Juan Carlos, 2015.

[15] Quang-Tuan Luong y col. «On determining the fundamental matrix: Analysis of different methods and experimental results». Tesis doct. Inria, 1993.

[16] Raúl Mur-Artal, J. M. M. Montiel y J. D. Tardós. «ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras». En: *IEEE Transactions on Robotics* 33.5 (2017), págs. 1255-1262.

[17] Peter J Rousseeuw. «Least median of squares regression». En: *Journal of the American statistical association* 79.388 (1984), págs. 871-880.

[18] D Scaramuzza y F Fraundorfer. «Visual odometry [Tutorial]». En: *IEEE Robotics & Automation Magazine* 18.4 (2011), págs. 80-92.

[19] David Valiente García, J M Montiel y J Dorado. «A review of visual odometry methods for mobile robots». En: *Sensors* 12.12 (2012), págs. 16093-16158.

[20] Sen Wang y col. «Deepvo: Towards end-to-end visual odometry with deep recurrent convolutional neural networks». En: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, págs. 2043-2050.

[21] Y Zhang y col. «A Hybrid Approach for Visual Odometry based on Monocular Visual SLAM». En: *2020 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE. 2020, págs. 1764-1769.

[22] Q Zou, Y Chen e Y Gao. «Towards a Real-Time Visual Odometry using a Hybrid Approach». En: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, págs. 6373-6379.