Universidad del País Vasco
Euskal Herriko Unibertsitatea

BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

CIENCIA Y TECNOLOGÍA ESPACIAL

TRABAJO FIN DE MÁSTER

VISIÓN E INTELIGENCIA ARTIFICIAL PARA ROBÓTICA DE EXPLORACIÓN

Estudiante *Estévez, Almenzar, Jesús*
Director *Pérez, Hoyos, Santiago*
Departamento *Física Aplicada*
Director *Sánchez, Cubillo, Javier*
Curso académico *2020-2021*

Bilbao, 9 de septiembre, 2021

Resumen

En este trabajo se desarrollan algoritmos de visión por ordenador e Inteligencia Artificial que permiten estimar la posición y localización de un vehículo mediante odometría visual y además reconocer y clasificar objetos del entorno. En concreto, se diseñan algoritmos que permiten calibrar un sistema de cámaras estéreo, estimar la pose de un objeto y generar mapas de disparidad y profundidad con reconocimiento y clasificación de objetos. Se analizan los fundamentos teóricos necesarios para diseñarlos y se muestran los resultados obtenidos en cada uno de los algoritmos. También se discuten las ventajas y desventajas de las técnicas empleadas. Para ello se trabaja en Python con la ayuda de paquetes de visión y segmentación de imagen, como OpenCV y PixelLib. Asimismo, se realiza una revisión acerca del estado del arte en técnicas de posicionamiento de vehículos de exploración y detección y clasificación de objetos. Se hace una estudio sobre las técnicas visuales empleadas en misiones de exploración espacial pasadas y futuras.

Palabras clave: **visión por ordenador**, **odometría visual**, **rover**, **detección de objetos**, **pose**.

Abstract

In this work, computer vision and AI algorithms are developed using visual odometry in order to estimate the position and location of a vehicle and also to recognize and classify objects in the environment. Specifically, algorithms that allow stereo cameras calibration, object pose estimation and disparity and depth map generation with object recognition and classification are designed. Theoretical fundamentals needed to design them are analysed and results in each algorithm are shown. Advantages and disadvantages of these techniques are also discussed. To do this, we work in Python together with vision and image segmentation packages such as OpenCV and PixelLib. A review about the State of the Art in techniques for pose estimation in exploration vehicles and object recognition and classification is done. Visual techniques applied in past and future space exploration missions are also analysed.

Key words: **computer vision**, **visual odometry**, **rover**, **object detection**, **pose**.

Laburpena

Lan honetan ordenagailu bidezko ikusmen- eta adimen artifizialeko algoritmoak garatzen dira, ikusmen-odometria erabiliz ibilgailu baten posizioa eta kokapena balioesteaz gain inguruneko objektuak antzematea eta sailkatzea ahalbidetzen dutenak. Zehazki, estereo kamera sistema bat kalibratzea, objektu baten posea balioztatzea, eta desberdintasun- eta sakontasun-mapak sortzea ahalbidetzen duten eta objektuak antzeman eta sailkatu ditzaketen algoritmoak diseinatzen dira. Horiek diseinatzeko behar diren oinarri teorikoak aztertzen dira eta algoritmo bakoitzean lortutako emaitzak erakusten dira. Erabilitako tekniken abantailak eta desabantailak ere eztabaidatzen dira. Horretarako Python-en lan egiten da, OpenCV eta PixelLib bezalako irudi segmentazio eta ikusmen paketeen laguntzarekin. Halaber, Artea-Egoeraren berrikuspen bat egiten da esplorazio-ibilgailuak kokatzeko eta objektuak detektatu eta sailkatzeko tekniketan. Iraganeko eta etorkizuneko espazio-esplorazioko misioetan erabilitako ikus-teknikei buruz azterketa bat ere egin da.

Hitz-gakoak: **ordenagailu bidezko ikusmena**, **ikusmen-odometria**, **rover**, **objektuak detektatzea**, **pose**.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

La visión por ordenador y la Inteligencia Artificial (IA) se han convertido recientemente en temas de gran relevancia con una gran variedad de aplicaciones, como por ejemplo la automoción, robótica o navegación (ver Fig. 1.0.1). En el ámbito espacial, estas técnicas han ido cobrando cada vez más importancia debido a los avances y soluciones que ofrecen a los problemas que existen actualmente usando otras técnicas. Las aplicaciones son varias: seguimiento de planetas y basura espacial, imágenes de satélites, detección de objetos celestes y maniobras espaciales (Raghavan y Rao 2020). También juega un papel fundamental para misiones interplanetarias presentes y futuras, donde la cooperación entre humanos y rovers de exploración autónomos es necesaria y beneficiosa.

Actualmente, las comunicaciones entre un rover espacial y la Tierra están limitadas por varios factores, como pueden ser las ventanas de comunicación, la potencia disponible y la banda ancha (Larry Matthies *et al.* 2007). Además, las técnicas usadas hoy en día en el seguimiento de rovers producen errores en la posición y orientación (de ahora en adelante pose) a corto o largo plazo. La visión por ordenador permitiría salvar estos inconvenientes, ahorrando tiempo y energía, además de aportar seguridad, rapidez y robustez. A lo largo de estos últimos años se han desarrollado varias técnicas para poder determinar con precisión la pose de un robot o rover de exploración sin necesidad de instrumentos o sensores externos.

(a) Rover prototipo de exploración. Créditos: ZeniaLabs Automation Intelligence S.L / ESMERA Project.

(b) Brazo robótico para recoger muestras. Créditos: Cerilli y Zwick 2019.

Figura 1.0.1: Ejemplos de aplicaciones de visión por ordenador para robótica de exploración.

1.1. Objetivos del trabajo

En este trabajo se propone el desarrollo de algoritmos que permitan la autolocalización de un rover sin necesidad de sistemas GPS/GNSS. Los objetivos de este trabajo son:

- Analizar el estado del arte en algoritmos de visión e IA enfocadas a la robótica de exploración.
- Diseñar algoritmos de visión por ordenador e IA que permitan con total independencia estimar la pose de un rover, analizar los objetos que hay en el entorno, detectándolos y clasificándolos.

Esto es importante, ya que la capacidad de un rover de exploración para autolocalizarse en terrenos mediante algoritmos de visión e IA es crucial para futuras misiones espaciales donde la comunicación GPS/GNSS no es posible.

La estructura del trabajo es la siguiente: en la Sec. 2 se realiza una revisión acerca del estado del arte de las técnicas de localización y posicionamiento para rovers de exploración, analizando sus ventajas y desventajas. Se exploran las principales técnicas de visión por ordenador, de estimación de pose y de detección y clasificación de objetos. En la Sec. 2.5 se exponen diferentes aplicaciones de ejemplos en misiones de exploración en los que se han usado técnicas de visión por ordenador e IA. En la Sec. 3 se repasan los conceptos matemáticos necesarios para diseñar los algoritmos propuestos, desde la calibración de las cámaras hasta la creación de mapas de disparidad y profundidad. En la Sec. 4 se describen los materiales usados durante el desarrollo del trabajo y la metodología seguida en cada uno de los algoritmos diseñados. En la Sec. 5 se analizan y se discuten los resultados obtenidos en cada uno de los algoritmos desarrollados. Por último en la Sec. 6 se exponen las conclusiones del trabajo, acompañado de una visión acerca de trabajos futuros. En el Anexo A se muestra el código del programa usado.

1.2. Relación con el Máster en Ciencia y Tecnología Espacial

Este trabajo esta estrechamente relacionado con Procesado de Datos Espaciales. Las imágenes que se obtienen a partir del conjunto de cámaras son procesadas y analizadas durante la ejecución del algoritmo para generar unos resultados correctos. Por ejemplo, se trabaja con imágenes en escala de grises y se usan filtros WLS para mejorar la calidad de la imagen. El uso de filtros es común en este tema para reducir el ruido en las imágenes. Ejemplos usados por otros autores son filtros bilineales o filtros de Sobel. La convolución, aunque no explicada en este trabajo, se usa de manera implícita al trabajar con el filtro usado. También aparece cuando se hablan de redes neuronales convolucionales, esenciales para detectar, clasificar y segmentar objetos en una imagen. Por otro lado, a la hora de calibrar las cámaras y proyectar imágenes en sus planos, es necesario hablar de transformaciones afines. Estas herramientas son fundamentales ya que permiten, por ejemplo, conocer las posiciones relativas de las cámaras y estimar la pose de los objetos de referencia. También se hace uso de transformaciones no lineales para eliminar la distorsión provocada por las cámaras.

Por otro lado, el uso de una cámara pinhole o un sistema de cámaras estéreo implica conocer cómo funciona la obtención de una imagen y sus características. Es por ello que este trabajo esta también relacionado con Fundamentos de Instrumentación Óptica. Al usar una o varias cámaras, conceptos como la distancial focal o las distorsiones que puede generar una lente son esenciales en este trabajo. Entender qué son y cómo tratar con ellas es importante para obtener buenos resultados. Por otro lado, en este trabajo también se habla de diferentes técnicas de posicionamiento haciendo uso de diferentes tipos de sensores, como los LIDAR o sensores visuales, útiles para estimar la posición de un rover. Es por ello que este trabajo también tiene relación con Detectores y Sensores.

Capítulo 2

Estado del arte

La automatización ha traído grandes avances en varios campos de la ciencia y la ingeniería. Hoy en día hay varias técnicas para determinar la posición de un rover en la superficie. A continuación se muestran las técnicas más usadas hasta el momento para determianr la posición de vehículos robóticos.

2.1. Técnicas de localización y posicionamiento

GNSS/GPS

El GNSS (Global Navigation Satellite System) es un sistema de navegación vía satélite que se ha usado ampliamente por todo el globo para una gran variedad de funciones. Destacan el sistema GPS creado por Estados Unidos; GLONASS, creado por Rusia; y GALILEO, creado por la Unión Europea. La constelación de satélites GPS cuenta con 24 satélites operativos, aunque únicamente hacen falta 4 para determinar la posición de un objeto. Las ventajas que ofrecen estos sistemas son varias. Ofrece una precisión muy alta, de hasta 10 metros (Aqel *et al.* 2016), y un error que no se ve incrementado con el paso del tiempo. Sin embargo, una baja relación señal-ruido, interferencias, bajo ancho de banda, o lugares cerrados como interiores o bajo tierra empeoran la precisión del sistema. Para solventar estos problemas se han propuesto diferentes métodos, como el GPS diferencial (DPGS) o el GPS en tiempo real (RTK-GPS), que mejoran la precisión del sistema GPS y permiten determinar localizaciones en lugares cerrados. Tanto el DGPS como el RKT-GPS proporcionan una precisión de hasta unos pocos centímetros. No obstante, esta técnica requiere una constelación de satélites, lo que hace inviable usarla en otros planetas como Marte.

Sensores láser

Esta técnica permite estimar las distancias a objetos emitiendo un láser y analizando la luz reflejada. Para ello el sensor puede usar técnicas de medida de tiempo de vuelo o de cambio de fase. La primera opción usa pulsos de láser y mide el tiempo que tarda la señal desde que se emite hasta que se vuelve a recibir. A este tipo de técnica se le suele conocer por el nombre de LiDAR (Light Detection And Ranging sensor). La segunda opción usa una señal continua con la que mediante efecto Doppler y comparando la fase recibida respecto a la transmitida calcula la distancia y velocidad del objetivo (Horn y Schmidt 1995 y Takeshi 2007).

La técnica LIDAR se usa ampliamente en la detección de obstáculos, mapeado y captura 3D en movimiento. Se puede usar junto con un sistema GPS o INS para mejorar la precisión en exteriores.

Figura 2.1.1: Odometría de ruedas. Imagen tomada por la cámara MAHLI del MSL. Créditos: NASA.

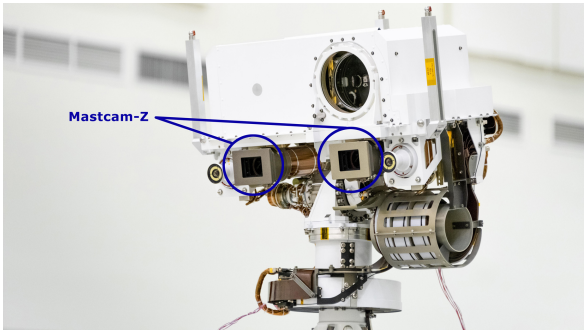


Figura 2.1.2: Sistema de cámaras visuales estéreo (Mastcam-Z) del rover Perseverance. Créditos: NASA/JPL-Caltech.

Sin embargo, esta solución es costosa tanto económica como computacionalmente cuando es comparada con alguna de las técnicas anteriores, pudiendo afectar a aplicaciones en tiempo real (Aqel *et al.* 2016). Además, según varios estudios (Takeshi 2007, Horn y Schmidt 1995 y Lingemann *et al.* 2005), esta técnica falla para materiales transparentes, como el cristal, debido a las reflexiones del láser en sus superficies.

Odometría de ruedas

Es una de las técnicas más usadas por ser sencilla y barata. Consiste en un codificador de ruedas que realiza un conteo de las revoluciones de las ruedas del vehículo, pudiendo determinar la trayectoria y dirección que sigue el rover en todo momento. Sin embargo, esta técnica no es muy precisa. Una terreno complicado como puede ser una superficie deslizante, resbaladiza, fangosa o arenosa puede hacer que las ruedas deslicen con la superficie, dando errores en la medida que se van acumulando e incrementando con el tiempo (Aqel *et al.* 2016), hasta el punto en que la posición ofrecida por la técnica dista mucho de la real.

INS (Inertial Navigation System) es una técnica de posicionamiento relativo que estima la pose de un objeto dado un estado inicial de referencia. Se ayuda de diferentes sensores, como acelerómetros y giróscopos, por lo que no depende de objetos externos para estimar tanto la localización del rover como su velocidad. Sin embargo, esta técnica no es del todo precisa debido a la acumulación de errores en los sensores inerciales, no siendo recomendable su uso durante un largo periodo de tiempo. Cualquier error de medida que se produzca se incrementa con el tiempo, por lo que se debe revisar periódicamente los datos de navegación (Aqel *et al.* 2016).

La técnica INS y odometría de ruedas fueron usadas en la misión Mars Exploration Rover (MER) de NASA para determinar la posición y el movimiento de los rovers marcianos (Y. Cheng, M. Maimone y L. Matthies 2005). Sin embargo, para suplir los errores que se cometían a la hora de estimar correctamente la localización se usó una técnica diferente basada en cámaras ópticas: la odometría visual.

Sensores Visuales

Es una técnica de visión por ordenador que estima la posición de un objeto a partir del análisis de las imágenes capturadas por una, dos o más cámaras. Los resultados que ofrece usar este tipo de sensores son varios. Por un lado, ofrecen unos resultados más precisos que las técnicas anteriormente descritas y una gran cantidad de información a partir de las imágenes, además de ser una técnica de

bajo precio. (Howard 2008; Nistér, Naroditsky y Bergen 2004). Por otro lado, esta técnica es capaz de funcionar en entornos donde un sistema GNSS no está disponible, como lugares cerrados o en exploración planetaria (M. He *et al.* 2020).

Sin embargo, usar sensores visuales tiene un coste computacional alto y la calidad de los resultados depende de las condiciones del entorno, como la luminosidad y los objetos que aparezcan en la escena. En la Tabla 2.1 se muestra un resumen las ventajas y desventajas que ofrece cada técnica:

Técnica	Ventajas	Desventajas
GPS/GNSS	Precisión alta con una incertidumbre conocida	Requiere una constelación de satélites
	No acumulación de errores con el tiempo	Imprecisión en interiores, interferencias o baja relación señal-ruido
Sensores láser	Alta precisión	Solución de alto coste
	Puede calcular distancias de un único punto o varios	Problemas en la reflexión de la señal en ciertas superficies
Odometría de ruedas	Alta precisión a corto plazo	Acumulación de errores con el tiempo
	Técnica barata y sencilla para estimar la pose	Problemas con el deslizamiento de las ruedas y con superficies complicadas
INS	Usa giróscopos para determinar la posición y orientación	Errores en la deriva de la posición
	No está sujeta a interferencias externas	Errores a largo plazo
Sensores visuales	Las imágenes ofrecen alto contenido de información	Requiere técnicas de procesado de imagen y extracción de datos
	Alta precisión en la posición	Alto coste computacional
	Solución de bajo coste	

Tabla 2.1: Ventajas y desventajas de varias técnicas de localilzación (Aqel *et al.* 2016).

Esta técnica se puede enfocar de varias maneras. Los métodos más usados son odometría visual (VO) y SLAM (Simultaneous Localization And Mapping). Ambos se encargan en estimar la posición y orientación en un espacio 3D (de ahora en adelante pose) de un robot o vehículo cualquiera haciendo uso de una o más cámaras ópticas. SLAM tiene como objetivo obtener una estimación robusta y consistente del camino que realiza un vehículo a la vez que reconstruye un mapa del entorno que hay a su alrededor (Servières *et al.* 2021), mientras que VO tiene como objetivo estimar la pose del vehículo analizando la secuencia de imágenes capturadas por una o más cámaras incorporadas, con el fin de seguir la trayectoria del vehículo, detectar obstáculos y rodearlos (Aqel *et al.* 2016).

En el trabajo presentado por Scaramuzza y Fraundorfer 2011 se hace una descripción a fondo de los fundamentos de la VO y , además, analiza la relación que hay entre VO y SLAM. Por un lado, la meta de SLAM es determinar una estimación global y robusta del recorrido que sigue el vehículo. Por el contrario, el objetivo de VO es recuperar de manera incremental el recorrido del rover pose tras pose. Dicho de otra manera, VO se centra únicamente en obtener una buena consistencia local de la trayectoria, mientras que SLAM se centra en la consistencia global del recorrido. La conexión que hay entre estas dos técnicas es que VO puede ser vista como un módulo de SLAM que puede reconstruir la trayectoria de forma incremental. Esto coloca a VO como mejor candidato en algunas

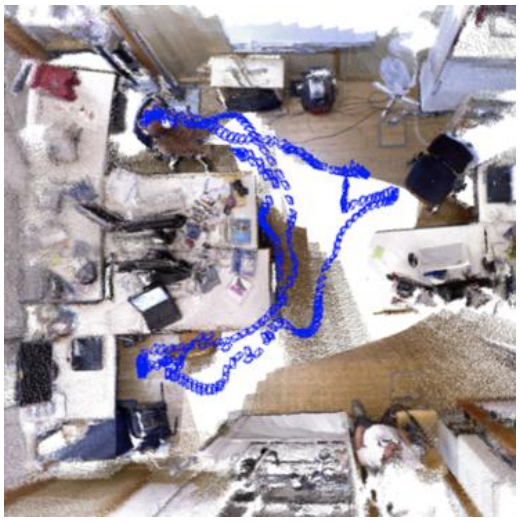


Figura 2.1.3: Nube de puntos de un escenario usando la técnica ORB-SLAM2. Los cuadrados azules indican la trayectoria que ha seguido el vehículo a lo largo de la habitación. Créditos: Mur-Artal y Tardós 2017.

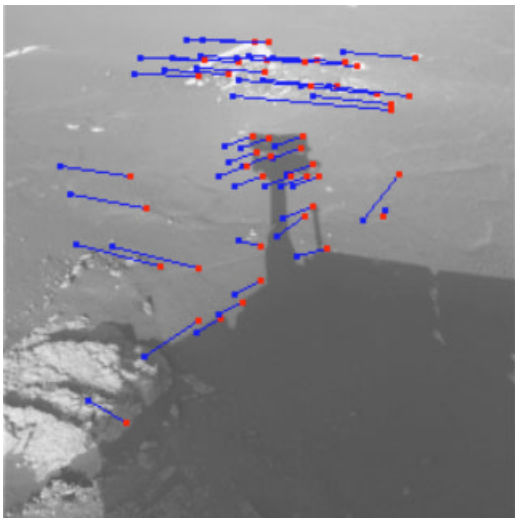


Figura 2.1.4: Imagen de la sombra del rover Opportunity aplicando VO. Los puntos rojos indican rasgos detectados en la imagen, los puntos azules y vectores son sus correspondientes localizaciones en el anterior fotograma. Créditos: Mark Maimone, Yang Cheng y Larry Matthies 2007.

aplicaciones donde se requiera estimar pose en tiempo real (M. He *et al.* 2020).

Dentro de cada método se pueden encontrar variantes (ver Sec. 2.2.1 y 2.2.2) que, dependiendo del uso que se quiera dar, será más recomendable usar una u otra. Sin embargo, ¿cómo se puede determinar la posición de un objeto a partir de un conjunto de imágenes? En la siguiente sección se hace una revisión más profunda sobre VO y SLAM y también se analizan los conceptos y técnicas más importantes en el campo de la visión por ordenador para extraer información útil de las imágenes, con el fin de poder determinar la pose de un objeto.

2.2. Técnicas de visión por ordenador

El problema de determinar la pose de un conjunto de cámaras implementadas en un vehículo de exploración a partir de un conjunto de imágenes es conocido en el campo de la visión por ordenador como Estructura del Movimiento (EdM) (Scaramuzza y Fraundorfer 2011). VO y SLAM son casos particulares de EdM. A continuación se hace un estudio más profundo sobre ambas técnicas.

2.2.1. VO

El origen de la odometría visual viene de la década de los 80, cuando fue por primera vez descrita por Moravec en su tesis doctoral (Moravec 1980). Es una técnica de visión y de localización de vehículos que ha adquirido importancia y popularidad, con un gran abanico de aplicaciones, aunque en sus inicios fue investigada y desarrollada principalmente por NASA para sus rovers de exploración en el programa de misiones de exploración en Marte (Scaramuzza y Fraundorfer 2011).

Como se ha dicho en el anterior apartado, esta técnica se encarga de estimar la pose y movimiento de un vehículo de forma incremental, proporcionando gran cantidad de información con las imágenes que toman una o más cámaras. Dependiendo del número de cámaras que se usen, se habla de un tipo u otro de VO. Principalmente hay dos tipos: monocular y estéreo.

- **Odometría Visual monocular**

  En este caso sólo se hace uso de una sóla cámara. Normalmente se suele usar junto con otras técnicas de posicionamiento como los LiDARs, siendo en esta situación un método híbrido. Sus aplicaciones son varias, aunque están enfocadas a la vehículos de conducción autónomos (Garg y Jain s.f.).

- **Odometría Visual estéreo**

  En este caso se usan dos cámaras que ayudan a determinar la pose del vehículo y a realizar un mapa de profundidades del entorno que le rodea. Para conseguirlo, la geometría epipolar (ver Sec. 3.3) es una herramienta clave que permite determinar la distancia a la que están los objetos que aparecen en escena. Esto permite definir la posición del vehículo a medida que se desplaza por el entorno. Actualmente, la VO estéreo se emplea cada vez más en misiones de exploración espacial (ver Sec. 2.5).

Aunque estas técnicas son las más comunes, se puede aumentar el número de cámaras. En este caso se habla de VO de multicámaras. En la Tabla 2.2 se muestran las principales ventajas y desventajas según que método se use y el número de cámaras que se empleen. Principalmente, usar más cámaras ofrece una mayor robustez a la hora de determinar la pose y las distancias. Sin embargo, hay que tener en cuenta que un mayor número de cámaras aumenta la complejidad del sistema.

	Monocular	**Estéreo**	**Multicámaras**
Complejidad mecánica	Baja	Media	Baja
Complejidad software	Media	Baja-Media	Alta
Robustez	Baja-Media	Media	Alta
Distancia máxima	Alta	Media-Baja	Alta

Tabla 2.2: Ventajas y desventajas de diferentes configuraciones de cámara usando VO (Mohta *et al.* 2017).

2.2.2. SLAM

En 1985, Chatila y Laumond 1985 propusieron un método de localización y mapeo de un entorno cualquiera. Más tarde, esta técnica se llamaría SLAM (Simultaneous Localization And Mapping), convirtiéndose en uno de los principales métodos más usados en el campo de la visión por ordeandor y con una gran variedad de aplicaciones. Como se dijo anteriormente, el objetivo de SLAM es obtener una estimación global del recorrido de un vehículo y reconstruir un mapa global del entorno que hay alrededor del objeto. Con el paso del tiempo, esta técnica ha ido evolucionando y ampliando sus posibilidades, desde aplicaciones en tiempo real hasta realidad aumentada (Servières *et al.* 2021).

Existen diferentes variantes de técnicas SLAM dependiendo de qué materiales se utilicen:

- **vSLAM**

  Cuando se hace uso de cámaras ópticas con las que analizar las imágenes y crear los mapas del entorno, se habla de vSLAM (visual Simultaneous Localization And Mapping) (Teng, Chuo y Hsieh 2018).

- **viSLAM**

  Se habla de viSLAM (visual inertial Simultaneous Localization And Mapping) cuando se combina el uso de cámaras visuales con unidades de medidas inerciales o IMUs. Los trabajos de Durrant-Whyte y Bailey 2006a y Durrant-Whyte y Bailey 2006b aportan un tutorial completo sobre viSLAM.

2.3. Técnicas de estimación de pose

Durante estas últimas décadas se han empleado varios algoritmos de estimación de pose de un vehículo. Los métodos principales son: métodos de seguimiento indirecto basados en la detección de rasgos, métodos de seguimiento directo y métodos de seguimiento semi-directo.

2.3.1. Método basado en detección de rasgos

El método de detección de rasgos es el más utilizado a día de hoy (M. He *et al.* 2020). Se basa en detectar a partir de dos pares de imágenes, ya sea mediante una o un conjunto de cámaras, varios puntos que se puedan emparejar fotograma a fotograma con el fin de determinar la pose del vehículo. Estos métodos tienen la capacidad de medir distancias y estimar la geometría 3D de un objeto. Para reducir el error de las medidas, se optimiza lo que se denomina un *error geométrico* (Engel, Koltun y Daniel Cremers 2018), que es un valor que indica la bondad de los resultados que se obtienen. Ahora bien, ¿cuáles son estos puntos y cómo identificarlos? Para dar respuesta a la pregunta, hay que entender dos conceptos importantes: puntos de interés (o rasgos) y detectores y descriptores de rasgos.

Rasgos

Para poder estimar la pose de un objeto primero hay que identificar qué puntos pertenecen a dicho objeto. Identificarlos no es una tarea trivial. La Fig. 2.3.1 se muestra a modo de ejemplo. En ella se observa una escena a campo abierto de un edificio junto con seis muestras recortadas de distintas regiones de la imagen. A y B son dos zonas de la imagen con una superficie plana y sin rasgos que se puedan identificar fácilmente, lo que hace difícil determinar sus posiciones exactas. C y D son dos filos de la azotea del edificio. En este caso es más fácil determinar donde están sus localizaciones aproximadamente, aunque no de manera exacta. Por último, E y F son unas esquinas del edificio. Se puede observar fácilmente el lugar exacto de dichas muestras en la imagen. En el contexto de visión por ordenador, los filos y las esquinas se denominan **rasgos** (***features***) o **puntos de interés** (***keypoints***). Los rasgos por tanto son puntos de una imagen con unas características que les permiten ser identificados fácilmente. Saber qué son los rasgos y cómo identificarlos es importante a la hora de trabajar en el campo de la visión por ordenador.

Detectores y descriptores de rasgos

Una vez se entiende qué son los rasgos, la siguiente pregunta que se puede plantear es: ¿cómo se pueden detectar? Actualmente existen varios detectores de rasgos que permiten identificarlos del resto de una imagen.

Figura 2.3.1: Ejemplo de distintos tipos de rasgos. Créditos: OpenCV[1]

El algoritmo probablemente más usado hasta la fecha (Bay *et al.* 2008) es el **detector de esquinas de Harris** (Harris y Stephens 1988). Es un método popular desarrollado en 1988 que permite identificar esquinas en una imagen analizando la variación de intensidad de brillo alrededor de un píxel. De esta manera, el detector de Harris puede identificar regiones planas, filos y esquinas. Posteriormente en 1994, Jianbo Shi y Tomasi desarrollaron un detector similar pero con ligeras modificaciones que ofrecía mejores resultados que el propuesto por Harris (Shi y Tomasi 1994). Estos detectores funcionan bien incluso bajo rotaciones de imagen. Sin embargo, bajo cambios de escala no son suficientemente buenos.

Fue en 2004, cuando Lowe (Lowe 2004) desarrolló un nuevo detector de rasgos que superó a sus antecesores en varios aspectos, denominado **SIFT** (Scale Invariant Feature Transform). Lo novedoso de este algoritmo es que lleva incorporado un descriptor de rasgos que le permite guardar información valiosa acera de los rasgos que identifica y, aún más importante, que dicha información es invariante bajo transformaciones de imagen. El trabajo realizado por Lowe describe que SIFT también se puede usar de forma aproximada como detector de objetos haciendo uso de los rasgos que detecta en una imagen.

Estos algoritmos, a pesar de funcionar correctamente y dar buenos resultados, no son lo suficientemente rápidos como para poder usarlos en aplicaciones de tiempo real. Es por ello que durante los años siguientes se diseñaron detectores y descriptores de rasgos y esquinas más rápidos y eficientes. Uno de ellos es **SURF** (Speeded-Up Robust Features), que fue propuesto por Herbert Bay (Bay *et al.* 2008) en 2008, que ofrecía mejores resultados y un tiempo de computación notablemente menor que el de sus predecesores. Por otro lado, Edward Rosten y Tom Drummond publicaron en 2006 un artículo (Rosten y Drummond 2006) donde presentaban un nuevo detector de rasgos y puntos de interés cuya rapidez superaba a SURF y a los anteriores detectores, al que llamaron **FAST** (Features from Accelerated Segment Test). El algoritmo consiste en tomar un píxel de la imagen con intensidad I_p y, trazando un círculo de 16 píxeles alrededor de éste, comparar los valores de brillo considerando un valor umbral fijo, t. Si hay un número n de píxeles (inicialmente los autores consideran $n = 12$) con un brillo mayor que $I_p + t$ o menor que $I_p - t$, entonces el píxel I_p es una esquina. Este procecso se aplica a todos los píxeles de la imagen. Los autores demostraron que este método proporciona unos resultados satisfactorios y con un menor tiempo de computación. La Fig. 2.3.2 muestra una imagen a modo de ejemplo de cómo se detecta una esquina aplicando este método.

[1] https://www.docs.opencv.org/master/df/d54/tutorial_py_features_meaning.htm

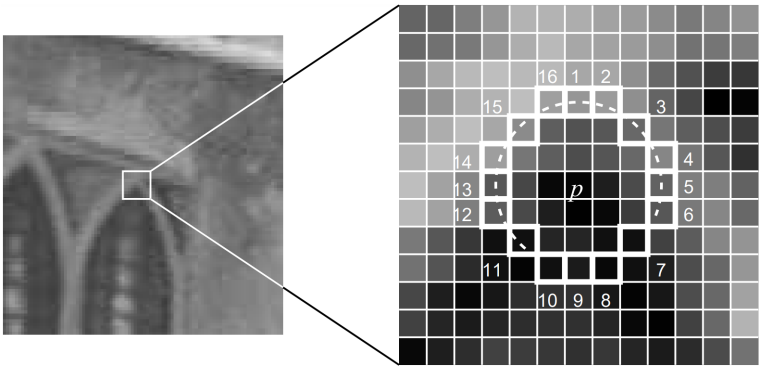


Figura 2.3.2: Ejemplo de detección de una esquina aplicando el método FAST. La línea discontinua representa el círculo trazado por 16 píxeles centrado en el píxel de referencia p con intensidad I_p. Créditos: Rosten y Drummond 2006

Más tarde en 2010, Michael Calonder y su equipo diseñaron un nuevo descriptor de rasgos denominado **BRIEF** (Binary Robust Independent Elementary Features) (Calonder *et al.* 2010), el cual fue comparado con el descriptor de SURF, demostrando que tanto el tiempo de computación como el de ejecución son mucho más rápidos que otros descriptores anteriormente diseñados. Los autores remarcan que hay que tener en cuenta que BRIEF es un descriptor de rasgos, no aporta ningún método para detectar los rasgos. Una desventaja que tiene BRIEF es que los resultados que ofrece con rotaciones de imagen no son del todo satisfactorios (Rublee *et al.* 2011). Debido a este problema y a la necesidad de reducir el ruido de la imagen, en 2011 Rublee y compañía (Rublee *et al.* 2011) diseñaron un nuevo detector eficiente que servía como alternativa a SIFT y SURF. Este nuevo algoritmo, basado en el detector de rasgos de FAST y en el descriptor de rasgos de BRIEF, se llama **ORB** (Oriented FAST and Rotated BRIEF). Este algoritmo combina las ventajas de estas dos técnicas, lo que le permite detectar rasgos con gran robustez en imágenes reescaladas y rotadas, convirtiéndolo en la mejor técnica para trabajar a tiempo real hasta la fecha (M. He *et al.* 2020). En su trabajo, los autores comparan estos tres detectores y demuestran que ORB es un orden de magnitud más rápido que SURF y en torno a dos órdenes de magnitud más rápido que SIFT, con un tiempo de ejecución de 15.3 ms por fotograma, pudiendo operar sin problemas a tiempo real con imágenes de 640×480 píxeles.

Durante los años siguientes se han diseñado más algoritmos con métodos que optimizan los resultados. En M. He *et al.* 2020 se hace una revisión de varios algoritmos que se usan tanto en VO monocular como de multicámaras mediante el método de detección de rasgos. Uno de las técnicas más actuales y exactas es **ORB-SLAM2** (Mur-Artal y Tardós 2017), capaz de trabajar de forma eficaz en cualquier ambiente a tiempo real y con CPUs estándar. Esta técnica está basada en la anterior versión diseñada por el mismo equipo (Mur-Artal, Montiel y Tardós 2015). En este caso hace uso del algoritmo ORB y la técnica vSLAM para detectar y describir rasgos y crear un mapa de la zona de estudio. La Fig. 2.1.3 muestra un ejemplo de los resultados que consiguieron los autores en su trabajo.

2.3.2. Método basado en seguimiento directo y semi-directo

Aunque el método indirecto (detección de rasgos) es el más utilizado hoy en día, existen otros métodos que siguen un camino diferente. Los métodos basados en seguimiento directo y semi-directo

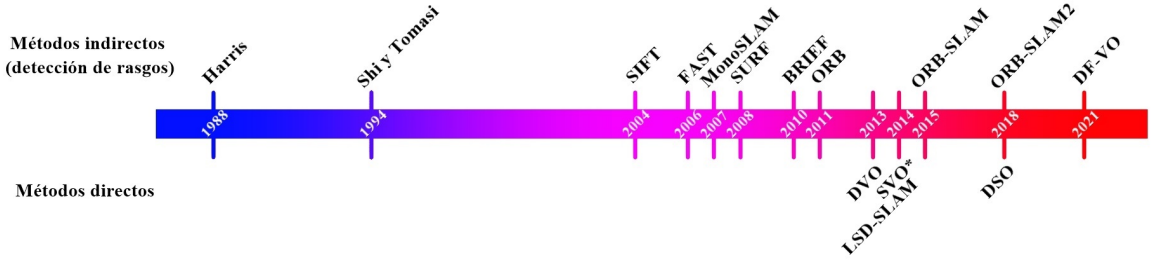


Figura 2.3.3: Cronología de los detectores y descriptores de rasgos.

son técnicas recientes de visión por ordenador aplicadas a la determinación de la pose de un vehículo que no tienen en cuenta los rasgos de una imagen. El método directo estima el movimiento de la cámara minimizando el *error fotométrico* (i.e. minimizando el error geométrico del método de detección de rasgos) de las imágenes analizando los valores fotométricos de los píxeles sin necesidad de detectar rasgos (Engel, Koltun y Daniel Cremers 2018 y Newcombe, Lovegrove y Davison 2011). Algunos de los algoritmos que se han diseñado basados en seguimiento directo son **DVO** (Dense Visual Odometry) (Kerl, Sturm y D. Cremers 2013), **LSD-SLAM** (Large-Scale Direct Monocular SLAM) (Engel, Schöps y Daniel Cremers 2014) y **DSO** (Direct Sparse Odometry) (Engel, Koltun y Daniel Cremers 2018). La diferencia que hay entre el método directo e indirecto es que éste último realiza un paso extra para extraer los rasgos de una imagen (Engel, Koltun y Daniel Cremers 2018). Por otro lado, el método semi-directo o híbrido toma las ventajas del método de detección de rasgos y el de seguimiento directo. **SVO** (Fast Semi-Direct Monocular Visual Odometry) (Forster, Pizzoli y Scaramuzza 2014) es un ejemplo de algoritmo que usa este método.

En la Fig. 2.3.3 se muestra la cronología de las técnicas de estimación de pose con diferentes métodos usados que han sido diseñadas a lo largo de estas últimas décadas. Se puede observar que hay una gran cantidad y variedad de algoritmos diferente, lo que da una idea de la gran importancia que ha adquirido este sector en ámbitos como la automoción, IA o el sector espacial.

2.4. Técnicas de detección y clasificación de objetos

La detección y clasificación de objetos es un tema que se ha popularizado en los últimos años en varios campos, como en la robótica y visión por ordenador. En cuanto al sector de la exploración espacial, existen varias propuestas de modelos enfocados a detectar objetos en la superficie de Marte (ver Sec. 2.5).

Las redes neuronales convolucionales han tenido un papel importante en el desarrollo de este campo, donde varios modelos actualmente conocidos y famosos se basan en esta técnica. Esencialmente, la detección de objetos consiste en una serie de algoritmos que, dada una imagen, es capaz de localizar y clasificar los objetos que aparecen en ella. Se pueden dividir en los siguientes pasos (Russakovsky *et al.* 2015):

- **Clasificación de imagen**: Predice el tipo de clase al que pertenecen los objetos que aparecen en una imagen digital.

- **Localización de objetos**: Localiza los objetos en una imagen digital. Normalmente se indica con un rectángulo en torno al objeto.

- **Detección de objetos**: Este paso engloba a los dos anteriores. Consiste en localizar la posición de un objeto y determinar su tipo de clase.

- **Segmentación de objetos**: Es una extensión del punto anterior. En este caso el algoritmo realza la forma del objeto que se detecta con ayuda de máscaras.

En Russakovsky *et al.* 2015 se hace una revisión sobre la evolución de las técnicas en detección de objetos a lo largo de los últimos años. Además, también compara la precisión de algunos de los métodos más empleados con la precisión del ojo humano. La precisión con la que se mide la bondad de un resultado se calcula estimando la diferencia entre el rectángulo trazado en torno al objeto y el rectángulo esperado. Este valor está comprendido entre 0 y 1, de forma que un buen resultado estará cerca de 1 y viceversa.

Prácticamente todos los modelos que se usan hoy en día hacen uso de redes neuronales convolucionales. A continuación se presentan dos técnicas conocidas y muy usadas que ofrecen buenos resultados.

2.4.1. Método R-CNN

El método **R-CNN** (Girshick *et al.* 2014) fue descrito por Ross Girshick *et al.* en 2014. Se popularizó por sus buenos resultados en detección y segmentación de objetos, demostrando su eficacia y superando las demás técnicas usando el conjunto de datos de VOC-2012 y ILSVRC-2013. La Fig. 2.4.1 muestra una imagen con la vista general del proceso. El diseño del modelo se puede esquematizar de la siguiente manera:

1. Propuesta de regiones: El primer paso consiste en definir una serie de regiones en la imagen como posibles candidatos a clases de objetos. Para ello usan técnicas de visión por ordenador que les permiten realizar búsquedas selectivas de regiones en la imagen.

2. Extracción de rasgos: El siguiente paso se trata de extraer rasgos de cada uno de las regiones propuestas. Para ello hace uso de una red neuronal convolucional.

3. Clasificación de objetos: Por último se encarga de clasificar los objetos detectados de entre la base de datos.

Los resultados que consiguieron fueron de una precisión media superior al 50 %, cuando otros modelos en su momento apenas superaban el 30 %. Sin embargo, el principal inconveniente de este modelo es su lentitud en la detección de objetos.

En 2015, Girshick diseñó una versión mejorada, denominada **Fast R-CNN** (Girshick 2015). Este modelo entrena una red de detección para extraer los rasgos de las imágenes, consiguiendo velocidades mayores y una precisión media de hasta el 68 %. Más tarde en 2016, Shaoqing Ren *et al.* desarrollaron un modelo avanzado: **Faster R-CNN** (Ren *et al.* 2016). En este caso fusionan Fast R-CNN con una red de propuestas de regiones, encargada de localizar los objetos y estimar su precisión simultáneamente. Los resultados superaron los obtenidos por los anteriores modelos con una precisión media superior al 70 %. En 2018, Kaiming He *et al.* presentaron **Mask R-CNN**, una extensión de Faster R-CNN, que incluye la capacidad de segmentar objetos a partir de máscaras de segmentación de alta calidad (K. He *et al.* 2018).

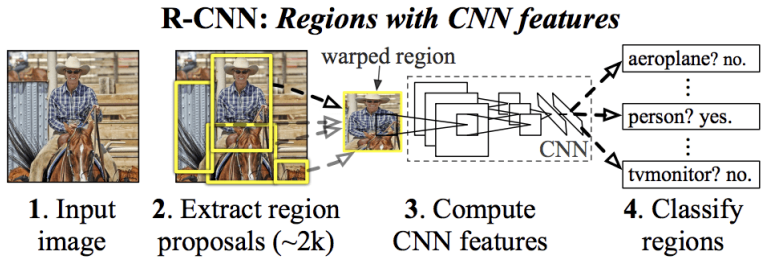


Figura 2.4.1: Visión general del proceso que sigue el modelo R-CNN (Girshick *et al.* 2014).

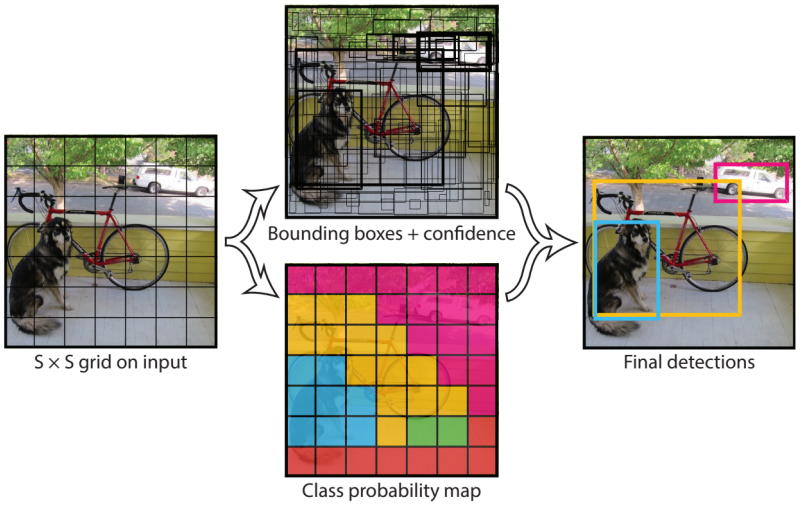


Figura 2.4.2: Esquema del proceso que realiza YOLO.

2.4.2. Método YOLO

En 2016, Joseph Redmon *et al.* diseñaron **YOLO** (You Only Look Once). Este modelo (Redmon, Divvala *et al.* 2016) consiste en única red neuronal que detecta objetos candidatos a partir de rectángulos de un conjunto inicial de rectángulos espaciados con sus respectivas probabilidades.

La técnica comienza dividiendo una imagen en varias celdas. A continuación, cada una de éstas debe predecir un número de objetos, con sus probabilidades y clases. Finalmente estos resultados se combinan formando el conjunto de rectángulos definitivos con sus objetos clasificados. La Fig. 2.4.2 muestra un esquema del funcionamiento. Los resultados demostraron una gran rapidez con respecto a otros modelos, aunque también mayores errores de localización.

En los años siguientes se diseñaron versiones mejoradas de este modelo, como **YOLO9000** (Redmon y Farhadi 2016) y **YOLOv3** (Redmon y Farhadi 2018). Estas versiones actualizadas contienen mejoras en la red neuronal y en la predicción de objetos. En Redmon, Divvala *et al.* 2016 hacen una comparación sobre la precisión media que obtienen diferentes modelos, incluyendo la combinación YOLO + Fast R-CNN.

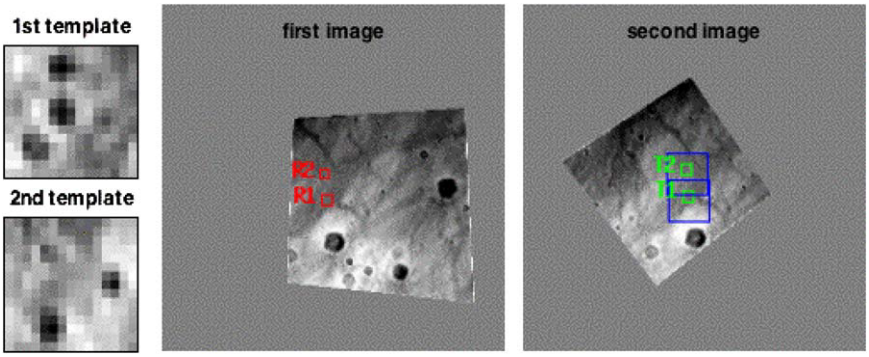


Figura 2.5.1: Par de imágenes consecutivas durante el descenso del rover Spirit. En rojo aparecen las zonas seleccionadas, en azul las ventanas de búsqueda y en verde las localizaciones seleccionadas. Créditos: Larry Matthies *et al.* 2007.

2.5. Aplicaciones en la exploración espacial

Hay varios ejemplos en los que se han usado técnicas visuales para misiones reales de exploración espacial. A continuación se muestran algunos ejemplos de aplicaciones reales en misiones de exploración espacial que ha habido a lo largo de estos años.

Mars Exploration Rovers (MER)

La misión Mars Exploration Rovers (MER) de la NASA fue la primera misión de exploración espacial en usar algoritmos de visión estéreo, VO y seguimiento de objetos de forma exitosa durante la fase de descenso y de navegación de los rovers Spirit y Opportunity (Larry Matthies *et al.* 2007, Scaramuzza y Fraundorfer 2011). En la fase de descenso, la misión MER usó existosamente un algoritmo de visión que estimó la velocidad a la que descendían los rovers analizando las imágenes que se tomaban de la superficie (ver Fig. 2.5.1). Durante la fase de exploración, la misión MER también hizo uso de la visión estéreo y algoritmos de evitación de obstáculos, al ser mejor opción para navegar por la superficie de forma segura frente a otras técnicas de localización, como odometría de ruedas o IMUs (Mark Maimone, Yang Cheng y Larry Matthies 2007). También hizo uso de odometría visual para determinar la posición de los rovers como demostración tecnológica, al no existir un sistema de localización GPS o puntos de referencia visuales (Larry Matthies *et al.* 2007).

En Mark Maimone, Yang Cheng y Larry Matthies 2007 hacen un resumen sobre los resultados de 2 años de operaciones en Marte de la misión MER usando VO. En este artículo muestran, por ejemplo, que el error que se obtiene usando VO en un recorrido deslizante de 2.45 m es menor del 1 %, mientras que el que se obtiene usando IMU y odometría de ruedas supera el 20 %. Sin embargo, destacan algunos de las problemas que hubo con esta técnica pionera. La primera de ellas es el coste computacional debido a la limitación técnológica del hardware[2]. Les tomaba varios minutos en realizar un ciclo completo desde la adquisición de las imágenes hasta determinar la pose de los rovers. Otro problema que detectaron fue que VO no trabaja particularmente bien en escenarios con poca textura. MER se encontró con varios terrenos arenosos, de baja textura. Como futuras mejoras, proponen que el sistema VO sea capaz de buscar y reconocer lugares donde haya estructuras con buena definición. Por otro lado, el trabajo realizado por Di *et al.* 2013 hace uso del repositorio de

[2]Estas misiones contaban con un ordenador de vuelo RAD6000 de 20 MHz con una memoria RAM de 128 MB y 256 MB de memoria flash (Maki *et al.* 2003, Mark Maimone, Yang Cheng y Larry Matthies 2007).

Figura 2.5.2: Marcas que deja el patrón de las ruedas del MSL a medida que se desplaza. Créditos: NASA/JPL-Caltech.

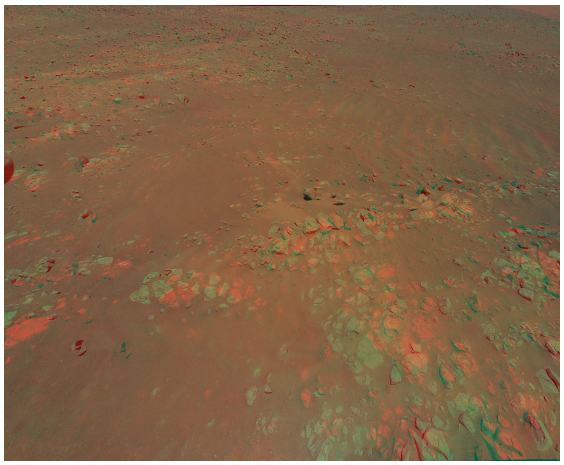


Figura 2.5.3: Anaglifo de la superficie de Marte obtenida con la Mastcam-Z. Créditos: NASA/JPL-Caltech.

datos de nubes de puntos 3D de la misión MER[3]. Utilizan técnicas de segmentación de objetos para detectar y analizar la forma y el tamaño de las rocas en las imágenes.

Mars Science Laboratory (MSL)

Con el objetivo de buscar en Marte evidencias orgánicas y moléculas fundamentales para la vida tal y como se conoce, el rover Curiosity amartizó en la superficie de Marte en el año 2012. Al igual que los rovers de la misión MER, el MSL también dispone de un algoritmo de VO que le permite calcular la distancia que se desplaza a medida que se mueve por la superficie de Marte. Para ello, las ruedas del rover tienen un patrón (concretamente las siglas JPL en código morse) tal que cuando se mueve deja esas marcas en el suelo, que posteriormente son usadas para estimar la distancia que recorre (Whitney Clavin s.f.). En la Fig. 2.5.2 se pueden ver las marcas que deja la rueda izquierda del rover.

También cuenta con las cámaras de navegación NavCam y MastCam, que les permiten realizar imágenes estéreo del entorno y trabajar con datos 3D del entorno. Varios trabajos se han publicado haciendo uso de las imágenes ofrecidas por estas cámaras. Por ejemplo, el trabajo realizado por Tao y Muller 2013 demuestra que usando la información aportada por estas cámaras se puede identificar y caracterizar las diferentes capas de sedimentos y rocas del suelo marciano. Para ello usan herramientos de visión, como detectores de rasgos y filtros de suavizado. Otro ejemplo es el trabajo realizado por Kerner *et al.* 2019. En este caso desarrollan un nuevo detector en imágenes multiespectrales llamado SAMMIE (Selections based on Autoencoder Modeling of Multispectral Image Expectations), que consta de una red neuronal convolucional con la que clasifica las rocas que detecta.

Mars 2020

El rover Perseverance de la misión Mars 2020, la última misión de la NASA en llegar a Marte, dispone del sistema de cámaras Mastcam-Z, que ofrece imágenes multicolor y estéreo. Consta de

[3] https://an.rsl.wustl.edu/mer/

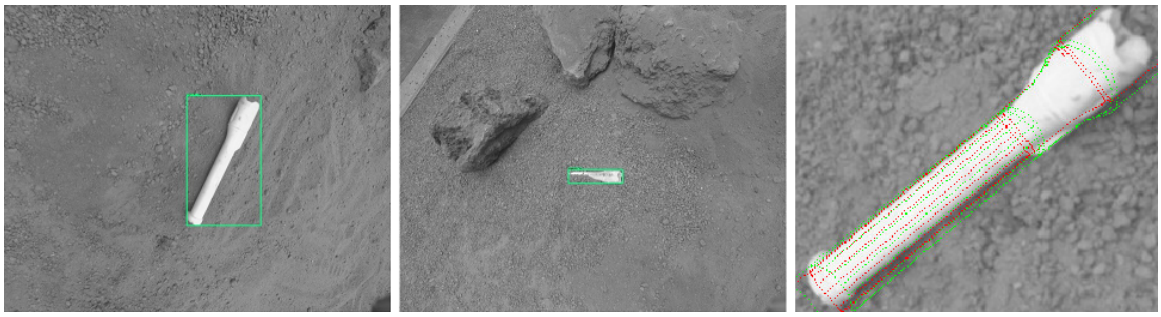


Figura 2.5.4: Ejemplos de detección de un tubo de muestra en diferentes posiciones. Izquierda: Detección completamente descubierta. En medio. Detección parcialmente cubierta. Derecha: Estimación de la pose. Créditos: Cerilli y Zwick 2019.

un par de cámaras que pueden hacer zoom en objetos lejanos, tomar imágenes 3D y grabar vídeos de alta velocidad. La Fig. 2.5.3 muestra uno de los varios de ejemplos de imágenes 3D que ya ha tomado el rover ayudándose de las dos cámaras de navegación de Mastcam-Z.

Otras misiones

Tanto la NASA como la ESA tienen como objetivo hacer uso de VO en misiones futuras de exploración en Marte, como ExoMars Rover o Sample Fetch Rover (SFR). La misión SFR tiene como objetivo enviar un rover encargado de recolectar las muestras dejadas por el rover Perseverance y entregarlas a un lanzador para enviarlas a la Tierra. Aquí es donde entra en juego el papel de VO. Analizar la superficie y buscar los recipientes con muestras del suelo de Marte es una tarea que requiere de algoritmos de visión y de reconocimiento de objetos. Hay varias propuestas para estas misiones que incluyen VO. Una de ellas es el sistema de odometría visual OVO (Churchill y Newman 2012), desarrollado por el Instituto de Robótica en la Universidad de Oxford. Este algoritmo usa el detector de rasgos FAST y el descriptor de rasgos BRIEF con el fin de poder trabajar en ambientes que cambian con el tiempo de manera repentina o gradualmente. Varios estudios hacen uso de este algoritmo (Shaw *et al.* 2013 y Townson, Woods y Carnochan 2018), demostrando su eficacia en terrenos marcianos simulados y considerándolo apto para futuras misiones espaciales como ExoMars o SFR.

Por otro lado, el estudio realizado por Cerilli y Zwick 2019 se centra en la detección y recolección de objetos en la superficie de Marte, en concreto para la misión SFR que será lanzada en los próximos años. En este trabajo hacen uso de la red neuronal artificial R-CNN (Ren *et al.* 2017) para detectar las muestras en la superficie (ver Fig. 2.5.4) y el algoritmo D^2CO (Imperoli y Pretto 2015) para determinar la pose de los tubos y poder recogerlos, incluso aún estando ocultados parcialmente, obteniendo unos resultados satisfactorios con un 70 % de tasa de éxito.

Capítulo 3

Fundamento teórico

Una vez realizada la revisión acerca del estado del arte de la visión por ordenador e IA enfocada a la robótica de exploración, el objetivo es diseñar un algoritmo que, haciendo uso de estas herramientas, sea capaz de estimar la pose de un vehículo con un par de cámaras estéreo a partir de objetos que haya en escena. Para ello, a continuación se hace una revisión del fundamento teórico de la visión por ordenador. El planteamiento es el siguiente:

1. **Calibración de las cámaras**: Se analiza las posibles distorsiones que pueden tener una o un conjunto de cámaras y el proceso de calibración que se sigue para eliminarlas. Este paso es fundamental ya que permite tomar imágenes correctamente sin errores para posteriormente estimar la pose de los objetos que haya en escena.

2. **Proyección 3D en cámaras**: A continuación se explica el proceso de proyección de imágenes 3D en cámaras. Es decir, entender cómo se obtienen imágenes 2D a partir de objetos reales 3D. Este paso es muy importante pues aparecen términos clave que servirán más adelante para determinar la pose de la cámara con respecto a objetos del entorno.

3. **Geometría epipolar**: El tercer apartado está dedicado a la geometría epipolar, qué es y cómo funciona. Un problema que surge cuando se usan cámaras es que al proyectar objetos 3D en una imagen 2D se pierde información acerca de la profundidad. La herramienta que ayuda a recuperar esa información perdida es la geometría epipolar.

4. **Mapas de disparidades y profundidades**: Al usar un par de cámaras estéreo es posible crear mapas de profundidades que aporten información valiosa sobre las distancias a las que se encuentran objetos en escena. Esto permite, por ejemplo, sortear obstáculos que puedan suponer un problema para el rover explorador. Para conseguir esto, anteriormente se debe crear un mapa de disparidades. En este punto se explica qué es y qué se necesita para crear un mapa de profundidades. Además, con las herramientas adecuadas, también es posible crear un mapa de puntos 3D del entorno. En este punto también se analiza esta cuestión.

3.1. Calibración de la cámara

La calibración de las cámaras tiene como objetivo determinar los parámetros geométricos intrínsecos y extrínsecos (ver Sec. 3.1.2) en el proceso de formación de imágenes (Laureano *et al.* 2015). Es el primer paso que hay que hacer y es fundamental en el campo de la visión por ordenador. La

correcta calibración de las cámaras permite eliminar posibles distorsiones que se generen y minimiza los errores a la hora de obtener resultados. A continuación se hace un repaso de los principales tipos de distorsión que se pueden encontrar.

3.1.1. Tipos de distorsiones

Algunas cámaras pueden introducir distorsiones importantes a las imágenes. Las más importantes son la **distorsión radial** y la **distorsión tangencial**. En la Fig. 3.1.1 se muestra un tablero de ajedrez (Fig. 3.1.1a) junto a estos tipos de distorsiones. Se ha elegido un tablero de ajedrez ya que, al estar formado por rectas paralelas y perpendiculares, es fácil detectar las distorsiones que puede generar la cámara. Por un lado, la Fig. 3.1.1b muestra un ejemplo distorsión radial. Esta distorsión provoca que las líneas rectas parezcan curvadas. Este efecto se incrementa cuanto más lejos se esté del centro de la imagen. La distorsión radial en cada punto de la imagen se puede expresar de la siguiente manera:

$$\begin{aligned} x_{distorsión} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{distorsión} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{3.1}$$

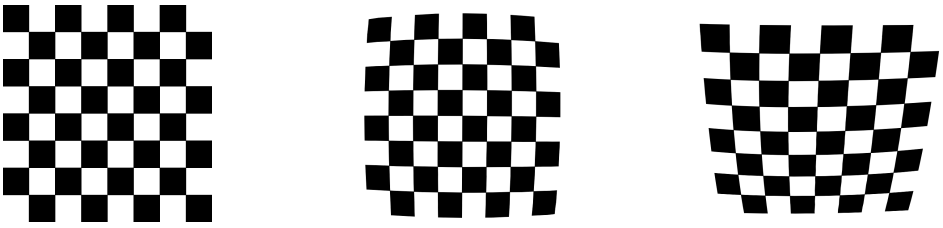
Por otro lado, la Fig. 3.1.1c muestra la distorsión tangencial, que es la causa de que diferentes zonas del tablero parezcan más grandes o pequeñas de lo que realmente son. Esto se debe a que la imagen no está alineada con el plano de la imagen de la lente. En este caso, la distorsión tangencial se calcula de la siguiente manera:

$$\begin{aligned} x_{distorsión} &= x[2p_1 xy + p_2(r^2 + 2x^2)] \\ y_{distorsión} &= y[p_1(r^2 + 2y^2) + 2p_2 xy] \end{aligned} \tag{3.2}$$

Donde x e y son las posiciones reales de los puntos del tablero, $x_{distorsión}$ e $y_{distorsión}$ son las posiciones de los puntos distorsionados, k_1, k_2 y k_3 son los coeficientes de distorsión radial y p_1 y p_2 son los coeficientes de distorsión tangencial. A partir de este conjunto de ecuaciones 3.1 y 3.2 se puede definir el vector de distorsión:

$$\boldsymbol{v_{distorsión}} = (k_1, k_2, k_3, p_1, p_2) \tag{3.3}$$

Que es un vector cuyos elementos están formados por los coeficientes de distorsión radial y tangencial. Es necesario obtener un vector de distorsión por cada cámara que se use. Sin embargo, con los vectores de distorsión no basta para calibrar la cámaras. En el siguiente apartado se estudian los parámetros geométricos necesarios para eliminar las distorsiones.



(a) Sin distorsión. (b) Con distorsión radial. (c) Con distorsión tangencial.

Figura 3.1.1: Tipos de distorsiones en un tablero de ajedrez. Imagen 3.1.1a: Tablero original sin distorsión. Imagen 3.1.1b: Tablero con distorsión radial. Imagen 3.1.1c: Tablero con distorsión tangencial.

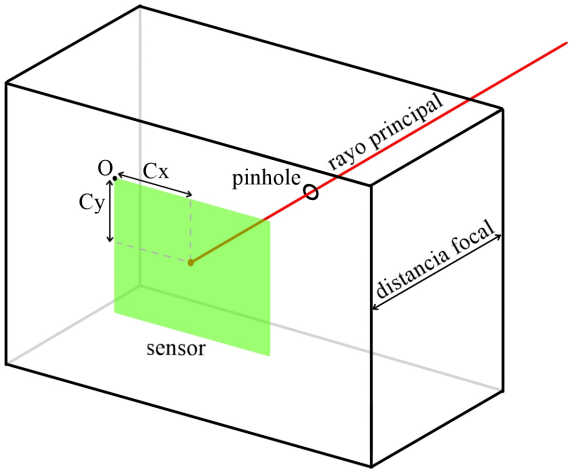


Figura 3.1.2: Diagrama de una cámara pinhole con sus respectivos parámetros intrínsecos.

3.1.2. Parámetros de calibración

Para realizar correctamente la calibración de la lente, es necesario conocer los coeficientes de distorsión anteriormente vistos y además, los denominados parámetros geométricos intrínsecos y extrínsecos de las cámaras.

Los **parámetros intrínsecos** son aquellos referidos a la propia cámara y son:

- La distancia focal (f_x, f_y): f_x y f_y son las distancias focales de la cámara a lo largo del eje x e y en píxeles, respectivamente. Normalmente f_x=f_y pero es posible que sean diferentes. Las razones pueden ser varias: defectos en los sensores de las cámaras, distorsiones no intencionadas, errores de fabricación, aberraciones...

- Los centros ópticos (c_x, c_y): También conocidos como puntos principales, son los puntos que resultan de la intersección del rayo principal con el plano imagen. c_x y c_y son las coordenadas en píxeles del centro óptico de la cámara respecto al origen del plano imagen (normalmente la esquina superior izquierda).

- Sesgo del eje (s): Es un coeficiente entre 0 y 1 que indica el ángulo de torsión de los píxeles. Está referido a la posible distorsión de cizalladura en las cámaras. Sin embargo, su valor generalmente es 0, por lo que se puede omitir.

En la Fig. 3.1.2 se muestra un diagrama de una cámara con sus parámetros intrínsecos. Cada uno de estos parámetros es responsable de una transformación 2D de la imagen en el plano imagen de la cámara. Por ejemplo, variando el centro óptico se consigue trasladar de sitio el objeto proyectado. Alterando el valor de la distancia focal se consigue aumentar o disminuir el tamaño del mismo. Por otro lado, el sesgo realiza una transformación de cizalladura. Con estos parámetros se define lo que se conoce como **matriz de cámara**, $\boldsymbol{K}$, que es clave para eliminar las distorsiones de las lentes de una cámara y para la posterior proyección en cámaras (ver Sec. 3.2). La matriz de cámara puede ser vista como una secuencia de 3 matrices que representan transformaciones 2D de traslación, escalado y cizalladura:

$$\boldsymbol{K} = \underbrace{\begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Traslación 2D}} \times \underbrace{\begin{bmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Escalado 2D}} \times \underbrace{\begin{bmatrix} 1 & s/f_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Cizalladura 2D}} = \begin{bmatrix} f_x & s/f_x & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

Por otro lado, los **parámetros extrínsecos** hacen referencia a los elementos de rotación y traslación que permiten pasar del sistema de coordenadas real (3D) al sistema de coordenadas imagen (2D). En concreto, para la rotación se tiene una matriz 3×3 y para la traslación un vector de 3 elementos:

$$\boldsymbol{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \qquad \boldsymbol{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \tag{3.5}$$

Con estos elementos se puede analizar ahora cuál es el proceso de proyección de objetos 3D en el plano imagen 2D de una cámara.

3.2. Proyección en cámaras

La Fig. 3.2.1 resume el proceso de la proyección de un punto 3D en el plano imagen de una cámara. En esta imagen, el punto P está localizado en el espacio real 3D y tiene como coordenadas (X, Y, Z) respecto a un sistema de coordenadas cartesianas arbitrario. Este mismo punto, al proyectarlo sobre el plano de imagen de la cámara, pasa a ser 2D con coordenadas de píxeles (u, v).

El proceso que se sigue para pasar de coordenadas 3D a coordenadas de píxeles es el siguiente:

1. Partiendo del punto P del espacio real 3D con coordenadas (X, Y, Z), se pasa al espacio cámara con coordenadas (X_C, Y_C, Z_C), que es el espacio de coordenadas cuyo sistema de referencia es la cámara. Para ello, hay que multiplicar (X, Y, Z) por la matriz de transformación que permita pasar del espacio real euclidiano al espacio cámara. A esta matriz se le denomina **matriz de transformación real-cámara**. La transformación viene dada por la matriz de rotación y el vector de traslación que se vio en la anterior sección. Recordándolo otra vez:

$$\boldsymbol{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \qquad \boldsymbol{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \tag{3.6}$$

   Aplicando la rotación y la traslación a las coordenadas reales se obtienen las coordenadas respecto al sistema de referencia de la cámara:

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} = \boldsymbol{R} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \boldsymbol{t} \tag{3.7}$$

   Esta expresión se puede simplificar si se reescriben las coordenadas reales en coordenadas homogéneas[1]:

[1] Las coordenadas homogéneas son un sistema de coordenadas que permiten realizar transformaciones afines y representarlas fácilmente en forma matricial. De forma general, un punto cualquiera en coordenadas homogéneas (x, y, z, w) es el mismo punto $(x/w, y/w, z/w)$ en coordenadas cartesianas.

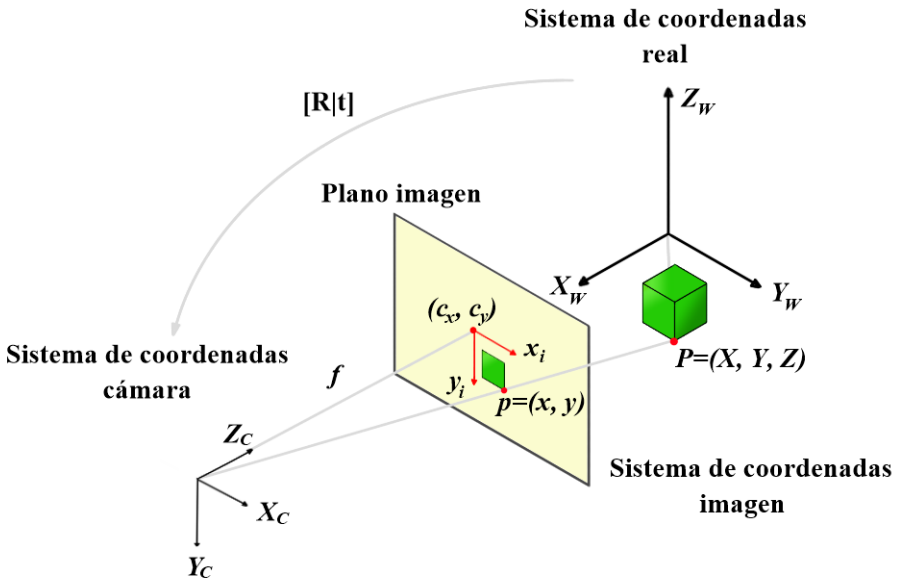


Figura 3.2.1: Proyección de un punto en el plano imagen de una cámara.

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = [\boldsymbol{R}|\boldsymbol{t}] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \longrightarrow \boldsymbol{X}_C = [\boldsymbol{R}|\boldsymbol{t}]\, \boldsymbol{X} \tag{3.8}$$

2. El siguiente paso es proyectar las coordenadas del punto respecto al sistema de referencia de la cámara al plano de la imagen de la cámara. Para ello se hace uso de la **matriz de cámara**. Esta matriz, como se vio en el anterior apartado, está formada la distancia focal f (f_x y f_y) y los centros ópticos, c_x y c_y (se considera que el valor del sesgo es $s = 0$). Su expresión, como ya se vio, es la siguiente:

$$\boldsymbol{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

Proyectando $\boldsymbol{K}$ sobre (X_C, Y_C, Z_C) se obtienen las coordenadas del punto sobre el plano imagen de la cámara:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} \longrightarrow \boldsymbol{X'} = \boldsymbol{K}\boldsymbol{X}_C \tag{3.10}$$

Para obtener las coordenadas (x, y) en píxeles simplemente hay que pasar de coordenadas homogéneas a coordenadas cartesianas. Es decir:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \longrightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x'/z' \\ y'/z' \end{bmatrix} = \begin{bmatrix} f_x \frac{X_C}{Z_C} + c_x \\ f_y \frac{Y_C}{Z_C} + c_y \end{bmatrix} \tag{3.11}$$

En resumen, la transformación que hay que hacer para pasar de las coordenadas originales en 3D a las coordenadas proyectadas sobre el plano imagen de la cámara viene dada por la siguiente expresión:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \boldsymbol{K} \times [\boldsymbol{R}|\boldsymbol{t}] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \longrightarrow \boldsymbol{X'} = \boldsymbol{PX} \tag{3.12}$$

Donde $\boldsymbol{P}$ es una matriz de 3x4 y se define como la **matriz de proyección** del espacio 3D euclidiano al espacio imagen 2D. Nótese que al proyectar un objeto 3D en el plano imagen 2D de una cámara se pierde la información de la coordenada z (ver Fig. 3.2.1). Como se verá a continuación, la geometría epipolar servirá de ayuda para recuperar esa información perdida.

3.3. Geometría Epipolar

Un problema que surge cuando se proyectan objetos 3D en un plano 2D usando una cámara es la pérdida de información espacial del entorno y, en concreto, de la profundidad. Para resolver este problema hay que usar 2 o más cámaras que ofrezcan una visión estéreo. Esto permite triangular la posición de los objetos y determinar sus distancias a las cámaras. Esto permite tener una visión estéreo, que es similar a lo que hace el ojo humano para medir profundidades. La geometría que explica los conceptos básicos de la visión estéreo se denomina geometría epipolar.

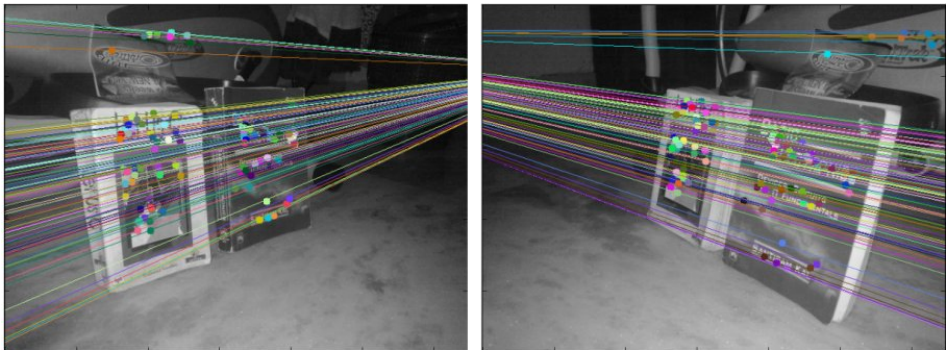


Figura 3.3.1: Par de imágenes con puntos correspondientes y sus respectivas líneas epipolares (en colores). La relación que hay entre las dos imágenes (es decir, la posición de las cámaras) viene dada por una traslación y rotación. Créditos: OpenCV[2].

La geometría epipolar describe la relación geométrica entre dos sistemas de cámaras. Es independiente de la escena y depende únicamente de los parámetros internos de las cámaras y sus posiciones

[2] https://docs.opencv.org/3.2.0/da/de9/tutorial_py_epipolar_geometry.html

relativas. El propósito de esta herramienta es la búsqueda de puntos que se correspondan (ver Fig. 3.3.1) en un sistema de cámaras estéreo (Hartley y Zisserman 2004).

La Fig. 3.3.2 muestra un esquema sencillo donde resume lo más importante de la geometría epipolar. Sean dos cámaras con centros C y C' y planos imagen $\mathcal{I}$ y $\mathcal{I}'$. Un punto M cualquiera es proyectado en m en la cámara C trazando un rayo r que une ambos puntos. La idea es localizar dónde se proyecta M en la segunda cámara conociendo únicamente m. En principio, se podría buscar dicha proyección en toda la imagen. Sin embargo, este proceso es muy costoso. La geometría epipolar ayuda a simplificar este problema. Volviendo a la Fig. 3.3.2, como el rayo r que une M y C pasa por m, la proyección de r en $\mathcal{I}'$ también pasará por la proyección de M en la segunda cámara, m'. Por otro lado, la intersección de la recta que une los centros de las cámaras con los planos imagen da lugar a dos puntos, e y e', denominados **puntos epipolares**. Un epipolo es las proyección del centro de una cámara en la otra cámara. La distancia que separa C y C' se le denomina **baseline** (B). Observando la imagen, se puede ver que tanto la baseline como r están contenidos en un plano, denominado **plano epipolar** (Π). Además, se tiene que la proyección de r en $\mathcal{I}'$ es l'_m, que pasa por el punto epipolar e' y está contenido en Π. A esta recta se le denomina **línea epipolar**, y en ella está contenida la proyección de M en la segunda cámara. Por tanto, la búsqueda del punto correspondiente a m en la segunda cámara se reduce a una sola línea (ver Fig. 3.3.1). A este hecho se le denomina restricción epipolar. Esto ocurre de forma general para cualquier punto real del espacio.

Una manera de englobar todo lo que se ha comentado anteriormente es haciendo uso de la **matriz fundamental F**. Es una matriz 3×3 que engloba la geometría intrínseca del sistema y relaciona las proyecciones de un punto real de una misma escena en dos imágenes estéreo. Contiene información sobre las matrices de cámara y sobre la traslación y rotación relatia entre las cámaras.

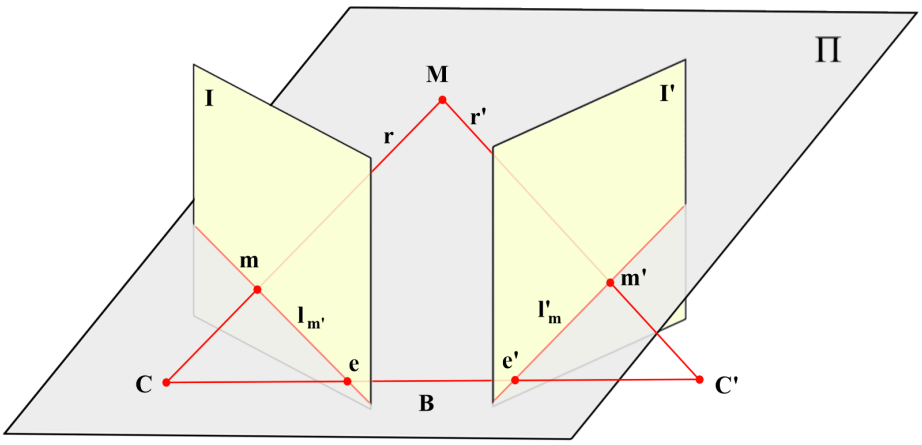


Figura 3.3.2: Esquema de la geometría epipolar con un sistema de dos cámaras estéreo. Créditos: Zhang 2014.

Este problema se puede simplificar aún más cuando los planos imagen de las cámaras se alinean. Con esto lo que se consigue es que las líneas epipolares en ambas imágenes sean horizontales (ver Fig. 3.3.3). Es decir, en este caso los pares de puntos correspondientes tienen las mismas coordenadas verticales, reduciendo el problema de encontrar puntos correspondientes a únicamente buscar píxeles

en una misma fila. Esto es clave para poder formar mapas de disparidades y de profundidades a partir de dos cámaras, como se verá a continuación.

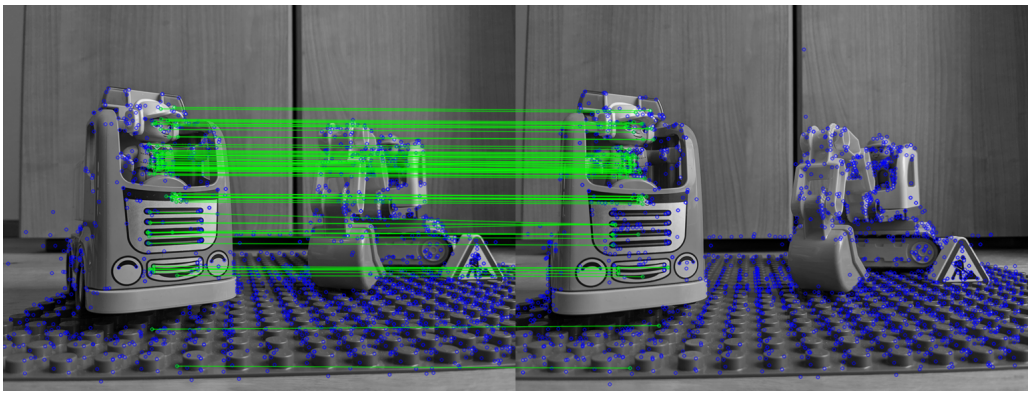


Figura 3.3.3: Caso especial de geometría epipolar donde el par de cámaras están alineadas. Los puntos epipolares (en verde) correspondientes en ambas imágenes tienen las mismas coordenadas verticales. Las rectas representan las líneas epipolares (en verde). Créditos: Andreas Jakl, 2020.

3.4. Mapas de disparidades y profundidades

Se ha visto que cuando se alinean el sistema de cámaras estéreo el problema de encontrar puntos correspondientes en ambas imágenes se simplifica enormemente. En este caso lo que ocurre es que los objetos más cercanos tienen una separación aparente mayor que los objetos que están más alejados. En la Fig. 3.4.1 se muestra un esquema de esto mismo. Al desplazamiento relativo de un objeto visto desde dos cámaras alineadas horizontalmente se le denomina disparidad. A partir de la disparidad, la focal de las cámaras (f) y la baseline (B) se puede determinar la distancia a la que está un objeto según la siguiente ecuación:

$$disparidad = x - x' = \frac{Bf}{Z} \tag{3.13}$$

Con esta ecuación, se puede determinar la profundidad a la que se encuentra cada píxel de la imagen y representarlo en un mapa de profundidades. Hoy en día existen librerías con determinados algoritmos que permiten, a partir de los parámetros intrínsecos del sistema de cámaras y programas externos, crear una nube de puntos 3D a partir de una imagen 2D. Para ello se hace uso de la **matriz de reproyección**. Esta matriz calcula las coordenadas (X, Y, Z) de un punto dada su posición (x, y) con valor de disparidad $d(x, y)$ en el mapa de disparidad de la siguiente manera:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = Q \begin{bmatrix} x \\ y \\ d(x,y) \\ 1 \end{bmatrix} \tag{3.14}$$

Nótese que este resultado está expresado en coordenadas homogéneas, por lo que la posición 3D del punto en coordenadas cartesianas es $(X/W, Y/W, Z/W)$. Por otro lado, la matriz de reproyección viene definida de la siguiente forma:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & -1/B & (c_x - c'_x)/B \end{bmatrix} \tag{3.15}$$

Donde c_x, c_y, c'_x y c'_y son los centros ópticos de la cámara izquierda y derecha respectivamente, B la baseline y f la focal de las lentes.

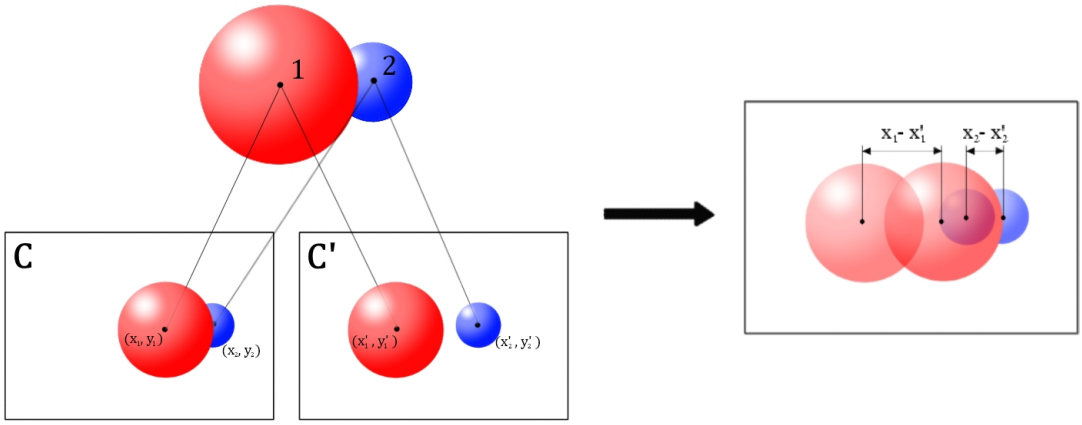


Figura 3.4.1: Disparidad en un sistema de dos cámaras estéreo. Los objetos cercanos tienen un desplazamiento relativo mayor que los objetos lejanos.

Capítulo 4

Metodología

El trabajo tiene como objetivo desarrollar algoritmos de visión por ordenador e IA con el fin de determinar la localización de un rover mediante la estimación de la pose y el reconocimiento de objetos. En esta sección se detalla la metodología seguida en el diseño de los algoritmos. Primero se explica el material utilizado, tanto hardware como software. Seguidamente, se detallan los algoritmos que se han diseñado.

4.1. Materiales

Para trabajar con algoritmos de visión se ha usado un par de cámaras Anvask Full HD 1080p 30 fps formando un sistema de visión estéreo. Se ha elegido usar un par de cámaras frente a un sistema monocular debido a las ventajas que ofrece. Por un lado, usar dos cámaras es idóneo para desarrollar una visión estéreo y crear mapas de profundidades 3D a partir de mapas de disparidades. Además, como se explicó en la Tabla 2.2, un sistema estéreo ofrece resultados más robustos sin aumentar la complejidad del diseño del software. Sin embargo, el uso de dos cámaras aumenta la complejidad mecánica del sistema, donde la calibración de ambas cámaras juega un papel fundamental, como se verá a continuación. Las características principales a tener en cuenta de las cámaras usadas son las siguientes:

Anvask Full HD 1080p 30 fps		
Adquisición de imagen	Sensor de imagen	1/3" sensor de 2 millones de píxeles
	Resolución de pantalla	2.0 MP, 1080P
	Lentes	HD, f=3.6 mm, FOV= 110°
Vídeo	Resolución	1920×1080
	Fotogramas	30 fps
Parámetros físicos	Dimensiones (L×W×H)	91 mm×70 mm×35 mm
	Peso	104 g
	Tamaño del sensor (W×H)	4.8 mm×3.6 mm

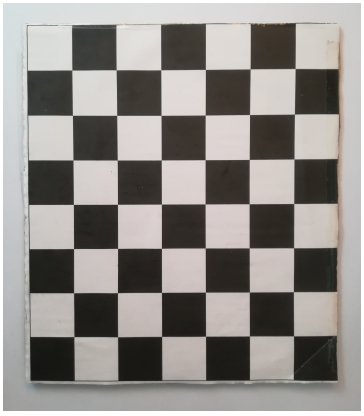
Tabla 4.1: Características de las cámaras usadas.

El par de cámaras fueron atornilladas a un soporte a modo de sujeción para facilitar su manejo y evitar lo máximo posible desplazamientos relativos entre ambas cámaras (ver Fig.4.1.1a), con una

baseline de 8.7 cm. Por otro lado, a la hora de calibrar las cámaras se ha hecho uso de un tablero de ajedrez de 7×8 celdas (ver Fig. 4.1.1b). Esta elección es debido a que detectar los bordes de las celdas y sus esquinas es sencillo. Como se verá a continuación, se puede usar un detector de esquinas para rectificar las imágenes de las cámaras y eliminar sus distorsiones.

El lenguaje de programación usado ha sido Python 3.8, junto con varias librerías de visión por ordenador e IA. Fundamentalmente se ha usado OpenCV[1]. Es una librería dedicada a la visión por ordenador en tiempo real y al procesado de imágenes. Dispone de una gran variedad de herramientas y funciones con las que trabajar. Para este trabajo, se ha usado OpenCV tanto para la calibración de las cámaras como para la determinación de la pose y creación de mapas de disparidades debido a su gran versatilidad, robustez y completitud. En cuanto al reconocimiento y clasificación de objetos se ha usado la librería PixelLib[2]. Esta librería esta desarrollada específicamente para tratar con segmentación de imagen y vídeo, lo que lo hace idóneo para este trabajo. Adicionalmente, también se han usado otras librerías matemáticas como NumPy o Matplotlib. Este trabajo se ha realizado en un ordenador portátil de uso privado con un procesador Intel i5-10300H 2.5 GHz y memoria RAM de 8 GB.

(a) Sistema de cámaras estéreo Anvask Full HD 1080p 30 fps.



(b) Tablero de ajedrez de 7×8 usado para calibrar las cámaras y estimar la pose.

Figura 4.1.1: Material usado para el diseño de los algoritmos.

4.2. Algoritmos

En total se han desarrollado 5 algoritmos, cada uno con un propósito y una finalidad diferente. En esta sección se explica en qué consisten y los pasos que siguen. Las explicaciones vienen acompañadas de diagramas de flujo para visualizar las estructuras de cada algoritmo. La Fig. 4.2.1 muestra la consecución de los algoritmos diseñados con una breve descripción de sus funciones. Para más información, el Anexo A contiene las líneas de código de cada algoritmo. A continuación se detalla la metodología seguida en cada uno de ellos:

1. **Calibración de las cámaras**: Es el algoritmo centrado en la calibración de las cámaras. Es el primer paso a realizar y consiste, a partir de un objeto de referencia, rectificar las imágenes

[1] https://docs.opencv.org/3.4/index.html
[2] https://pixellib.readthedocs.io/en/latest/index.html

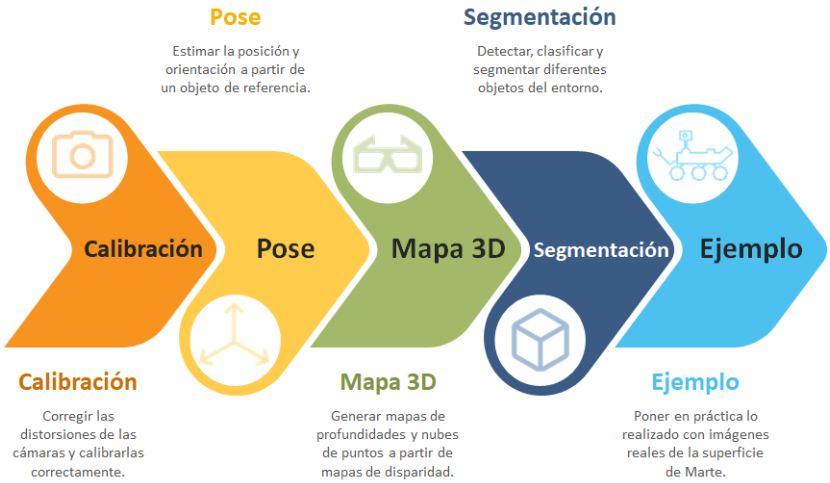


Figura 4.2.1: Diagrama general seguido en los algoritmos.

de ambas cámaras, eliminar las posibles distorsiones que existan y determinar los parámetros intrínsecos de cada cámara. En este caso se ha elegido como objeto de referencia un tablero de ajedrez de 7×8 celdas. OpenCV dispone de funciones específicas para realizar cada uno de estos pasos.

El algoritmo comienza encendiendo las cámaras y capturando las imágenes grabadas cada 500 ms. Seguidamente estas imágenes son pasadas a escala de grises ya que en este caso únicamente interesa la intensidad de brillo de cada píxel y no su color. El siguiente paso es detectar el tablero de ajedrez. Para ello hay que colocarlo frente a las cámaras en diferentes posiciones y orientaciones, ocupando el máximo campo visible de las dos cámaras. Es necesario que ambas detecten el tablero a la vez, pues esto es un requisito para posteriormente realizar la calibración estéreo del sistema. OpenCV tiene una función específica para detectar las esquinas de las celdas del tablero de ajedrez, `cv.findChessboardCorners()`[3]. Esta función detecta las esquinas de las celdas en las imágenes (puntos imagen) con la ayuda de una matriz de referencian (puntos objeto). En este caso se ha elegido una matriz de referencia de 6×7 elementos, representando cada uno de ellos las esquinas de las celdas a detectar en el tablero. Así pues, cuando se muestra el tablero y se detecta un patrón de 6×7 esquinas en ambas cámaras (o lo que es lo mismo, de 7×8 celdas), se dice que el algoritmo ha encontrado el tablero. Cuando lo hace, se optimiza la localización de las esquinas usando la función `cv.cornerSubPix()` y después se almacenan en un vector. Este proceso se repite cada vez que el algoritmo detecta el tablero de ajedrez, capturando tantas imágenes como se necesiten. Un mayor número de fotos bien realizadas ofrece mejores resultados, aunque el tiempo de computación aumenta. Una vez ya se han tomado todas las fotos, el algoritmo pasa a calibrar las cámaras. El primer paso es calibrar las cámaras individualmente haciendo uso de la función `cv.calibrateCamera()`. Esta función toma los puntos objeto y puntos imagen de las esquinas detectadas y devuelve para cada cámara los vectores de distorsión y la matriz de cámara (ver Sec. 3.1.2 y 3.1.1). Esto se consigue proyectando los puntos objeto en el tablero y minimizando el error de reproyección

[3]La función `cv.findChessboardCorners()` de OpenCV empleada para detectar las esquinas de las celdas del tablero está basada en el detector de Harris (Duda y Frese 2018).

con los puntos imagen. A continuación se optimiza la matriz de cámara usando la función `cv.getOptimalNewCameraMatrix()`. Una vez se han calibrado individualmente las cámaras, el algoritmo realiza la calibración estéreo con la función `cv.stereoCalibrate()`. Esta función devuelve los parámetros intrínsecos y extrínsecos del sistemas de cámaras, como la matriz fundamental y la matriz de rotación y vector de traslación que permite pasar del sistema de referencia de una cámara al otro (ver Sec.3.3).

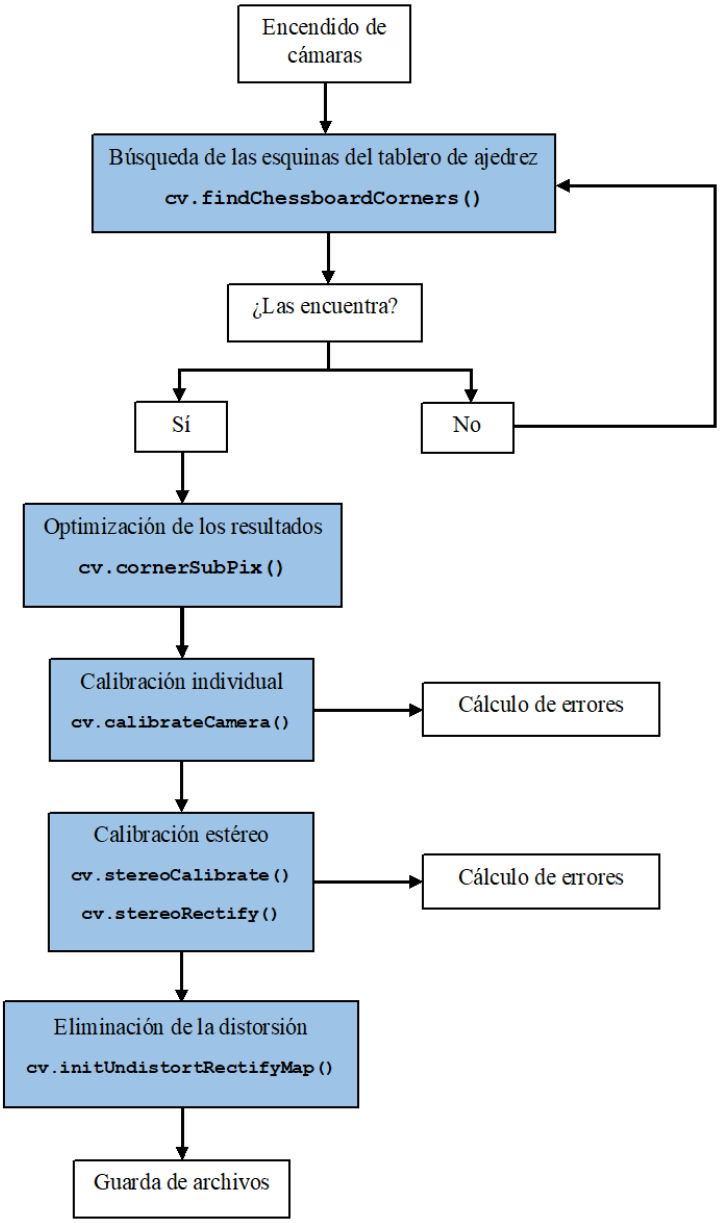


Figura 4.2.2: Diagrama de flujo del algoritmo de calibración del sistema de cámaras.

Por último, se realiza la rectificación estéreo con la función `cv.stereoRectify()`. Esta función se encarga de alinear los planos de las cámaras con los parámetros anteriormente obtenidos, lo que se traduce en que ahora las líneas epipolares aparecen horizontales. Además, devuelve como resultado adicional la matriz de reproyección 2D-3D. Esta matriz es importante a la hora de crear mapas 3D a partir de una imagen 2D, como se verá más adelante. Tras esto, se usa la función `cv.initUndistortRectifyMap()` para crear los mapas de rectificación y eliminación de la distorsión que deben ser aplicados a las imágenes de las cámaras para calibrarla correctamente. Para ello se usa la función `cv.remap()`. Una vez finalizado el proceso, el algoritmo devuelve el error cometido tanto en la calibración individual como en la calibración estéreo. Finalmente los parámetros de calibración se guardan en un fichero. Así, cada vez que se inicien las cámaras no hay que volver a calibrarlas de nuevo. En la Fig. 4.2.2 se muestra un diagrama de flujo de la estructura del algoritmo, junto con las correspondientes funciones usadas de OpenCV.

2. **Determinación de la pose del objeto de referencia**: Este segundo algoritmo tiene como objetivo determinar la pose (*i.e.* traslación y rotación) del objeto de referencia, en este caso el tablero de ajedrez. A modo de comprobación, en las imágenes se representa un sistema de ejes cartesianos en uno de los vértices del tablero junto con los vectores de rotación y traslación.

   El algoritmo comienza cargando los parámetros de calibración anteriormente obtenidos. A continuación, una vez encendidas las cámaras, elimina las distorsiones en ambas cámaras con la función `cv.remap()`. Posteriormente, siguiendo el mismo proceso que en el algoritmo de calibración, se detectan las esquinas de las celdas del tablero de ajedrez. Para ello, hay que colocar el tablero de forma que sea visible para las dos cámaras. Cuando lo consigue, pasa a calcular la pose (vectores de traslación y rotación) con la función `cv.solvePnP()`. Esta función estima la pose del tablero a partir de los puntos objeto (matriz de referencia), sus correspondientes puntos imagen (las esquinas detectadas en cada imagen) y los parámetros intrínsecos. Esencialmente se basa en lo explicado en la Sec. 3.2 (Ec. 3.8). Con esta información y conocidos los parámetros internos de las cámaras (matrices de cámara y coeficientes de distorsión) la función devuelve la pose del tablero de ajedrez en tiempo real. Finalmente, en cada una de las imágenes se representa un sistema de ejes cartesianos en una de las esquinas del tablero, junto con su pose. En concreto, el algoritmo devuelve un vector de traslación y de rotación de 3 elementos cada uno. Inicialmente, las unidades del vector de traslación vienen dadas en términos de la matriz de referencia, es decir, en celdas. Para pasarlo a cm simplemente se multiplica el resultado obtenido por el tamaño real de una celda del tablero de ajedrez. Por otro lado, el vector de rotación se muestra en radianes, aunque luego se pasa a grados. La Fig. 4.2.3 muestra un diagrama de flujo de la estructura del algoritmo.

3. **Creación de mapas de disparidad y profundidades**: Este algoritmo consiste en crear un mapa de disparidad del entorno que hay alrededor del sistema. En este caso se distinguen dos modos: el primero, sin procesar los datos de las imágenes; y el segundo, aplicando un post-procesado mediante un filtro de mínimos cuadrados ponderados (mejor conocido como filtro WLS). Finalmente también se crea un mapa de puntos 3D a partir de los datos obtenidos de la disparidad. Para visualizar los resultados se ha usado el programa MeshLab.

   Recordando que durante la calibración los planos imágenes de las cámaras se alinean, basta con seguir el procedimiento explicado en la Sec. 3.4 para crear un mapa de disparidad. OpenCV cuenta con varias funciones dedicadas a la creación de mapas de disparidad a partir de dos imágenes alineadas y rectificadas. En este caso se usa la función `cv.stereoSGBM`. Se encarga de buscar la correspondencia entre puntos vistos desde ambas cámaras analizando las imágenes

usando una ventana de un determinado número de píxeles. El proceso de búsqueda y emparejamiento de píxeles entre imágenes que sigue esta función viene descrito con más detalle en Birchfield y C. Tomasi 1998.

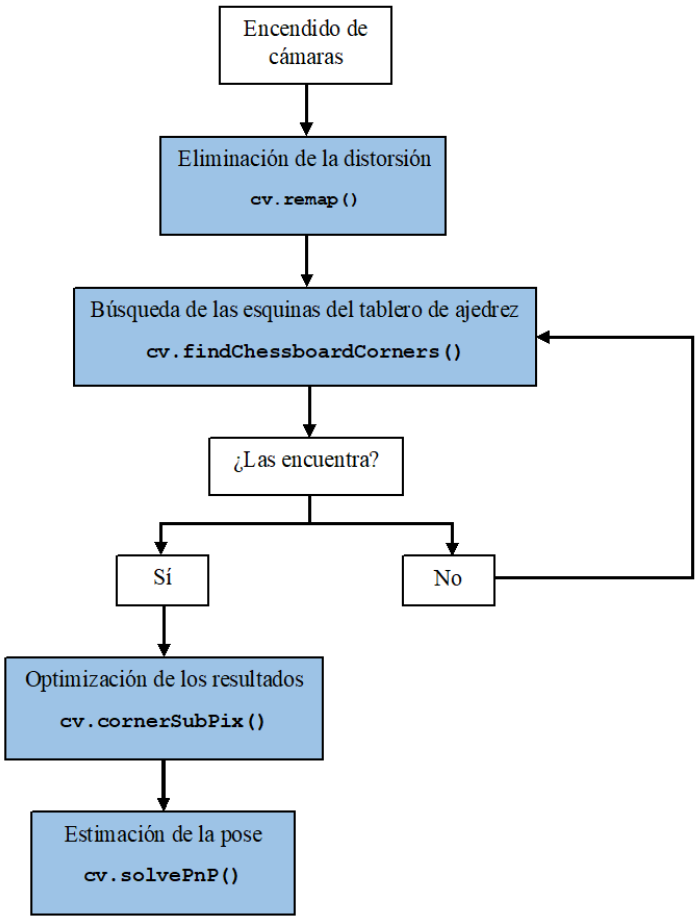


Figura 4.2.3: Diagrama de flujo del algoritmo de estimación de la pose del tablero de ajedrez.

Esta función dispone de varios parámetros con los que mejorar la calidad del resultado. Los principales parámetros usados son los siguientes:

- `numDisparities`: Es el valor máximo de disparidad menos el valor mínimo. Aumentarlo ayuda a visualizar mejor la disparidad en objetos cercanos.
- `blockSize`: Es el tamaño de la ventana de píxeles dedicado a buscar la correspondencia entre píxeles de ambas imágenes. Un valor pequeño aporta resolución a la imagen pero le añade ruido y viceversa.
- `minDisparity`: Es el valor mínimo de la disparidad que puede medir la función. Puede ser negativo o positivo.

Es necesario ajustar estos parámetros según convenga para obtener un mapa de disparidades correcto. Por otro lado, el algoritmo diseñado ofrece a su vez la posibilidad de usar un filtro para suavizar los resultados obtenidos en el mapa de disparidad. Esto es debido a que originalmente y en ciertas condiciones losresultados obtenidos contienen un alto nivel de ruido. El filtro usado se denomina filtro WLS (Weighted Least Squares) (Farbman *et al.* 2008). Este filtro tiene la particularidad de suavizar los valores de los píxeles preservando los bordes de los objetos en una imagen, lo que aporta una mejor calidad y resolución al resultado final. OpenCV también dispone de una función específica para crear este filtro: `cv.ximgproc.createDisparityWLSFilter()`. Se ha usado para comparar los resultados con el mapa de disparidades sin filtrar y analizar las ventajas y desventajas que tiene. Una vez se crea, se aplica directamente al mapa de disparidad. Al igual que en el caso anterior, esta función tiene varios parámetros. Los que se han usado son:

- `lambda`: Define la regularización del filtrado. Un valor grande hace que los bordes del mapa de disparidad filtrado se adhieran más a los bordes de la imagen original.
- `sigmaColor`: Este parámetro define la sensibilidad del filtrado. Un valor grande conlleva a bordes de bajo contraste, mientras que un valor pequeño lo vuelve sensible al ruido.

Ajustando los parámetros (con o sin filtrar), el algoritmo calcula la disparidad de cada píxel en función de los parámetros introducidos. Acto seguido los valores de la disparidad se normalizan entre 0 (color negro) y 255 (color blanco) y se representan en un mapa en escala de grises. Los colores oscuros indican puntos lejanos y los colores claros puntos cercanos. Es necesario prestar especial atención las condiciones del entorno, tales como la luminosidad, calidad de la imagen, texturas, etc, ya que una mala elección aumenta el nivel de ruido y empeora la calidad de la imagen. Una vez que ya se han creado los mapas de disparidades y se aprueba el resultado, el algoritmo finaliza creando una nube de puntos y guardándolo en un archivo. Para ello lo que hace es reproyectar la información 2D del mapa de disparidad en un mapa 3D. Para ello usa la matriz de reproyección 2D-3D obtenida en el proceso de calibración. En ella vienen incluidas los parámetros intrínsecos de las cámaras. Para visualizar la nube de puntos se usa el programa externo MeshLab[4], un sistema de procesado de objetos 3D. Este programa permite mover, rotar, hacer zoom en la nube de puntos y medir distancias entre puntos.

4. **Reconocimiento y clasificación de objetos**: Este algoritmo se encarga de reconocer, clasificar y enmascarar objetos. Para ello se hace uso de un modelo específico del sistema de segmentación de imagen Mask R-CNN[5] (K. He *et al.* 2018). En cuando a las librerías usadas, además de OpenCV en este caso también se ha usado la librería PixelLib.

   Lo novedoso de usar Mask R-CNN es que también genera una máscara de los objetos reconocidos mediante segmentación de imagen. Por lo que a parte de poder detectarlos y clasificarlos, también se puede crear una máscara del objeto u objetos detectados. El modelo usado contiene hasta 81 clases de objetos diferentes. Entre ellos se incluyen vehículos, personas, animales, aparatos, electrodomésticos, alimentos, herramientas, etc.

   Este algoritmo está implementado en el anterior, por lo que a parte de generar mapas de disparidad y profundidad también ofrece la posibilidad de detectar y clasificar los objetos que hay en escena. La Fig. 4.2.4 muestra un diagrama de flujo de los pasos que realiza el algoritmo.

[4] https://www.meshlab.net/

[5] El modelo que se ha usado es mask_rcnn_coco.h5, que se puede obtener más información en la página de descarga: https://github.com/matterport/Mask_RCNN/releases/tag/v2.0

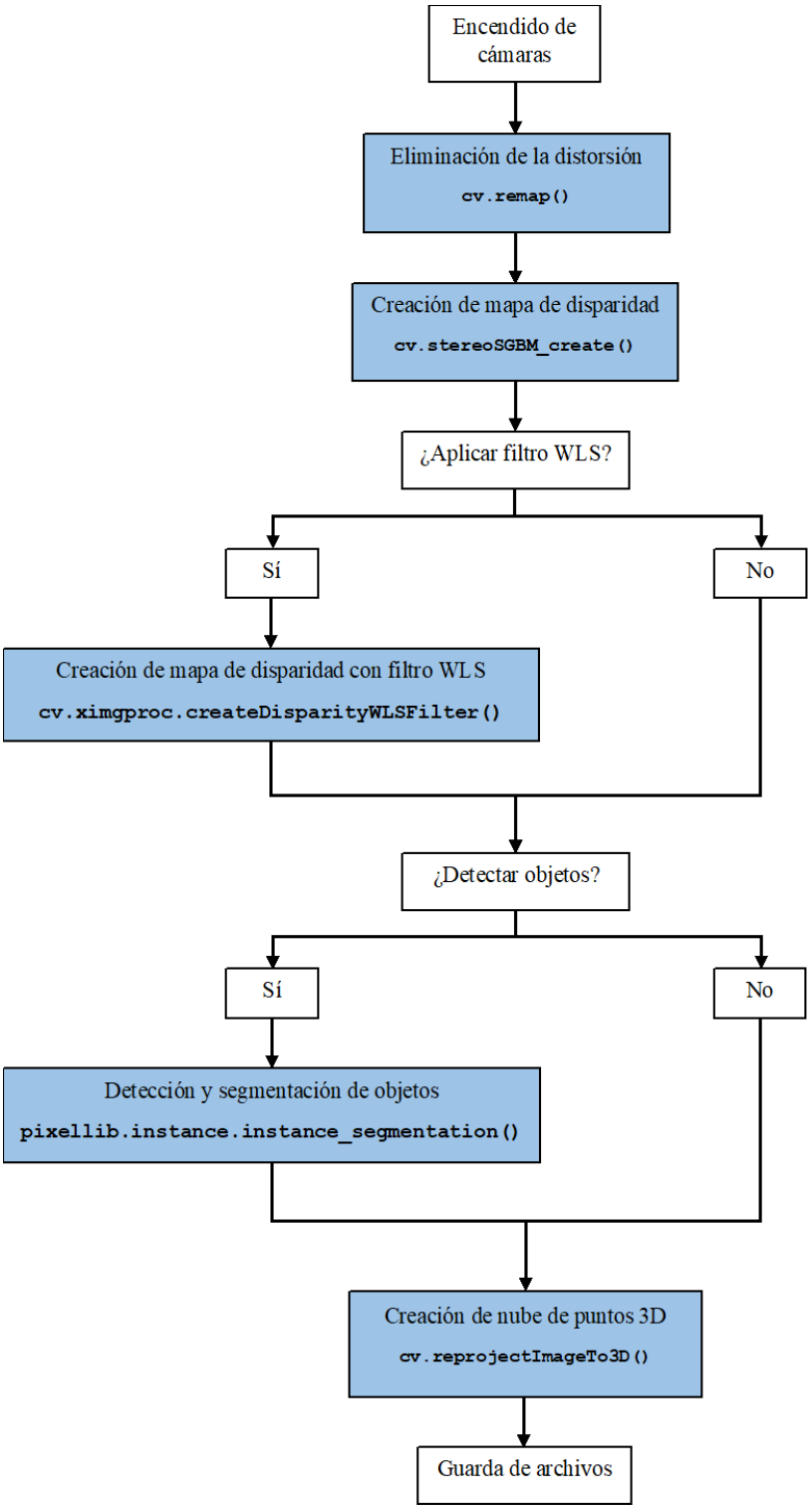


Figura 4.2.4: Diagrama de flujo del algoritmo de creación de mapas de disparidades y nubes de puntos 3D con detección y segmentación de objetos.

Esta implementación hace posible eliminar aquellas regiones de la imagen que no interesen y puedan introducir ruido al resultado final, como zonas sin textura u objetos no interesantes. Lo primero que hace es descargar el modelo y analizar las imágenes de las cámaras. Cuando detecta un objeto se generan automáticamente tres parámetros: el tipo de clase al que pertenece, la probabilidad de pertenecer a esa clase (o umbral de confianza) y la máscara del objeto. Posteriormente comprueba si la probabilidad es mayor que un valor dado, en este caso 0.7. Si es así, el algoritmo escribe en pantalla la clase a la que pertenece y su probabilidad. Por último, el algoritmo también aplica las máscaras generadas de los objetos reconocidos en el par de imágenes. Así, a la hora de generar los mapas de disparidad y de profundidad se puede elegir si visualizar todo el entorno o únicamente los objetos detectados por el algoritmo.

5. **Algoritmo de ejemplo**: En estos últimos años la visión por ordenador ha cobrado cada vez más importancia en el ámbito de la exploración espacial. En este algoritmo se hace un ejemplo con imágenes propias de los rovers marcianos, en concreto del rover Perseverance. Dado que su cámara de navegación MastCam-Z le permite tomar fotografías de la superficie marciana, se han elegido algunas imágenes de ejemplo para evaluar la técnica seguida en este trabajo con muestras reales. La página oficial de las cámaras MastCam-Z[6] ofrece, a parte de información técnica acerca de las cámaras, un repositorio de las imágenes que va tomando el rover Perseverance sus cámaras de navegación. En este caso se han buscado imágenes que muestren la superficie marciana vista desde las dos cámaras. Así, este algoritmo genera un mapa de disparidad siguiendo la misma metodología que en los anteriores casos. Sin embargo se han tenido en cuenta algunas consideraciones iniciales:

   - Las imágenes escogidas están tratadas y rectificadas correctamente, sin ningún tipo de distorsión, por lo que en estos casos no se ha tenido que realizar el proceso de calibración.
   - Dado que las imágenes escogidas fueron realizadas por otro sistema de cámaras (como la MastCam-Z), se ha tenido que tener en cuenta los parámetros intrínsecos de las cámaras a la hora de generar un mapa de profundidades.

   Conociendo la focal usada por las cámaras y la baseline, se puede calcular el mapa de profundidades de la escena elegida según la Ec. 3.13.

[6] https://mastcamz.asu.edu/

Capítulo 5

Resultados

En esta sección se detallan los pasos seguidos explicado en la metodología y se analizan los resultados obtenidos en cada uno de los algoritmos diseñados. También se discuten los inconvenientes y problemas que han surgido al trabajar con estos algoritmos, junto con posibles soluciones. Finalmente se hace una visión a futuro de la proyección que puede tener este trabajo y cómo podría mejorarse.

5.1. Calibración de las cámaras

En este apartado se muestra los pasos y resultados conseguidos con el algoritmo A.1.1. Para la calibración de las cámaras se ha usado un tablero de ajedrez de 7×8 celdas. La Fig. 5.1.1 muestra 4 pares de imágenes del tablero capturadas por las cámaras izquierda y derecha desde diferentes perspectivas. Esto se hace para captar mejor las distorsiones de las cámaras. En las figuras se han dibujado una serie de líneas verdes paralelas para visualizarlas. De este conjunto de imágenes se pueden destacar varios puntos importantes. Por un lado, se puede observar que ambas cámaras tienen distorsión radial y tangencial. Éstas aumentan en las zonas exteriores y bordes de la imagen. El tablero ayuda a visualizar estas distorsiones. Por otro lado, también se puede observar que los planos imágenes de las cámaras no están alineados correctamente. Esto se puede observar comprobando que un rasgo cualquiera visto desde las dos cámaras esté sobre una recta horizontal. En este caso no se cumple la correspondencia entre píxeles. En la Fig. 5.1.2 se muestran los mismos pares de imágenes con la detección de las esquinas de los tableros. Se puede observar que la correspondencia entre esquinas y el orden que sigue es la misma en cada par de imágenes. La información acerca de la posición de cada esquina de cada imagen es almacenada y usada posteriormente para obtener los parámetros de calibración.

El proceso de calibración se ha repetido varias veces con el fin de determinar la bondad de los resultados. Por ejemplo, las imágenes mostradas pertenecen a una prueba de calibración donde se han tomado en total 21 imágenes por cada cámara. Cada una de estas imágenes se ha tomado desde un ángulo y una posición diferente, intentando abarcar el máximo campo de visión que ofrecían las dos cámaras para así detectar las distorsiones. Tras realizar la calibración con estas imágenes los principales resultados arrojados muestran un valor aproximado de la focal de las lentes de 4.4 mm. Este valor está algo alejado del valor teórico ofrecido por el vendedor (ver Tabla 4.1), 3.6 mm. Esto representa un error relativo de más del 20 %. Por tanto, ha sido necesario realizar varias pruebas de calibración hasta conseguir unos resultados óptimos, que aseguren eliminar las distorsiones de las cámaras y rectificar las imágenes.

Figura 5.1.1: Toma de imágenes para la calibración de las cámaras. Se puede apreciar la distorsión de las cámaras y el alineamiento incorrecto de los planos imagen.

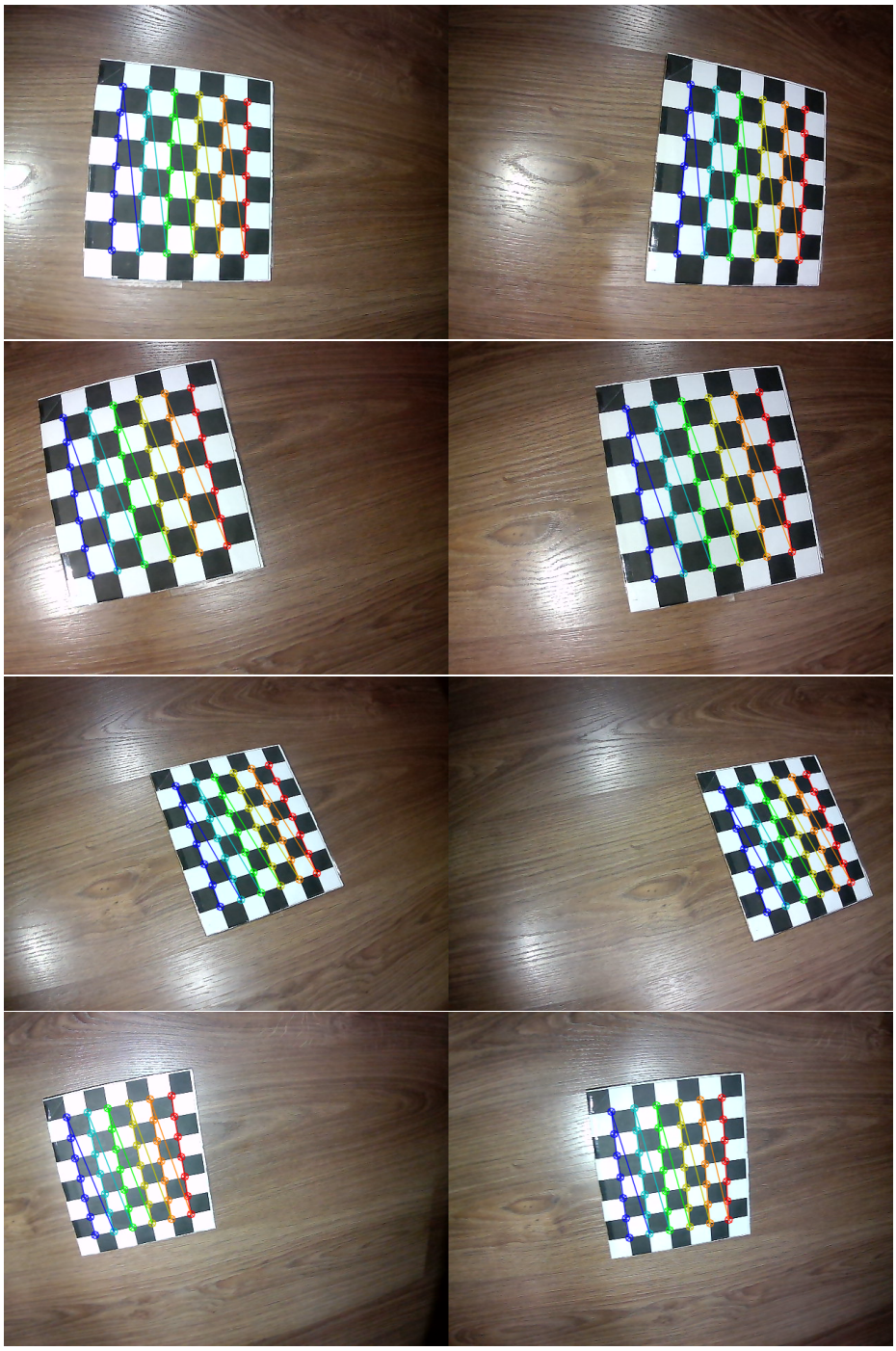


Figura 5.1.2: Detección de las esquinas de las celdas del tablero. Las esquinas detectadas siguen el mismo orden, representado por las rectas que las unen, en ambas cámaras.

Prueba	Nº imágenes	f_{izq} (mm)	f_{der} (mm)	ε_{ind} (%)	ε_{est} (%)	ε_{rel} (%)
1	21	4.4	4.4	0.04	1.02	22.2
2	43	3.5	3.5	0.04	0.6	1.8
3	64	4.6	4.4	0.03	0.8	25
4	67	4.6	4.5	0.03	0.6	26.4
5	68	4.5	4.4	0.06	1.01	23.6

Tabla 5.1: Resultados de las focales obtenidas en 5 pruebas realizadas junto con sus respectivos errores de calibración.

La Tabla 5.1 recopila los valores de las focales obtenidas para cada cámara en diferentes pruebas realizadas. Se puede observar que la prueba 2 arroja unos resultados muy cercanos a los valores teóricos de la focal, dando un error relativo (ε_{rel}) por debajo del 2 %. También es interesante ver que aunque el número de imágenes es menor, ofrece resultados mejores que las pruebas 3, 4 y 5, en las que se tomaron más imágenes. La razón es debido a que la calidad de los datos no depende únicamente de cuántas fotos se tomaron, sino también cómo. Imágenes movidas, con mala iluminación o posiciones desfavorables repercuten en el proceso de calibración y cálculo de parámetros. Continuando con la Tabla 5.1, se muestran también los errores de reproyección obtenidos durante la calibración, además del error relativo. En concreto, el error de calibración individual en las cámaras (ε_{ind}) y el error de calibración estéreo (ε_{est}). Los errores de calibración individual y estéreo se calculan a la vez que se determinan los parámetros intrínsecos. Se obtienen automáticamente calculando la suma total de las distancias entre los puntos imagen de las esquinas del tablero detectadas y los respectivos puntos reproyectados de la matriz de referencia usada. Por tanto, cuanto menor sea el error de reproyección mejor será la calibración. Los errores de reproyección obtenidos en todas las pruebas son casi despreciables, por lo que el proceso de calibración es robusto y eficaz, dando buenos resultados.

Para realizar los posteriores algoritmos se han usado los parámetros de calibración obtenidos en la prueba 2. Aunque realmente las demás pruebas también funcionan correctamente, hay que destacar que los valores de la focal obtenidos en la prueba 2 son los que más se aproximan al valor teórico ofrecido por el vendedor. Algunos de los parámetros de calibración usados en los siguientes algoritmos son los siguientes:

$$\boldsymbol{K}_{\text{izquierda}} = \begin{pmatrix} 470{,}4 & 0 & 319{,}5 \\ 0 & 471{,}8 & 239{,}5 \\ 0 & 0 & 1 \end{pmatrix}, \qquad \boldsymbol{K}_{\text{derecha}} = \begin{pmatrix} 470{,}9 & 0 & 319{,}5 \\ 0 & 471{,}8 & 239{,}5 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.1}$$

$$\boldsymbol{v}_{\text{distorsión izquierda}} = \begin{pmatrix} -0{,}418 & 0{,}156 & 0{,}0014 & 0{,}0016 & 0{,}042 \end{pmatrix} \tag{5.2}$$

$$\boldsymbol{v}_{\text{distorsión derecha}} = \begin{pmatrix} -0{,}402 & 0{,}160 & -0{,}0002 & -0{,}0003 & -0{,}018 \end{pmatrix} \tag{5.3}$$

Donde las matrices 5.1 son la matrices de cámaras y los vectores 5.2 y 5.3 son los coeficientes de distorsión de cada cámara obtenidos en la prueba 2. Las matrices de cámara están representadas en píxeles, por lo que si se desea expresar el valor de la focal en mm basta con multiplicar estos valores por el tamaño del píxel (0.0075 mm/píxel, se puede deducir a partir del tamaño del sensor y la resolución con la que se trabaja, Tabla 4.1). De esta manera se obtiene que el valor de la focal de ambas cámaras mide 3.5 mm, como se mostró anteriormente. Por otro lado, los coeficientes de

distorsión son adimensionales. Otro parámetro importante que se ha obtenido tras la calibración es la matriz de reproyección (ver Sec. 3.4):

$$\boldsymbol{Q} = \begin{pmatrix} 1 & 0 & 0 & -319{,}5 \\ 0 & 1 & 0 & -239{,}5 \\ 0 & 0 & 0 & 471{,}1 \\ 0 & 0 & -0{,}3 & 0 \end{pmatrix} \tag{5.4}$$

Esta matriz será importante en el algoritmo dedicado a la creación de mapas de disparidad y de profundidades, pues permite obtener una nube de puntos (mapa 3D) a partir de una imagen 2D.

Con los parámetros de calibración calculados, se puede comprobar el resultado final de la rectificación. En la Fig. 5.1.3 se muestra una imagen del tablero usado visto con el par de cámaras. Se puede observar que las distorsiones han sido corregidas exitosamente y ahora las celdas del tablero aparecen rectas y paralelas entre sí. Además, se puede visualizar que los planos imagen de ambas cámaras están alineados. Una prueba de ello es que las esquinas de las celdas vistas desde ambas cámaras caen sobre las mismas rectas verdes horizontales. Es decir, cada punto correspondido entre las dos imágenes tiene igual coordenada vertical. Esto simplifica el problema de la geometría epipolar y es clave para usarlo correctamente en el algoritmo de creación de mapas de disparidad.

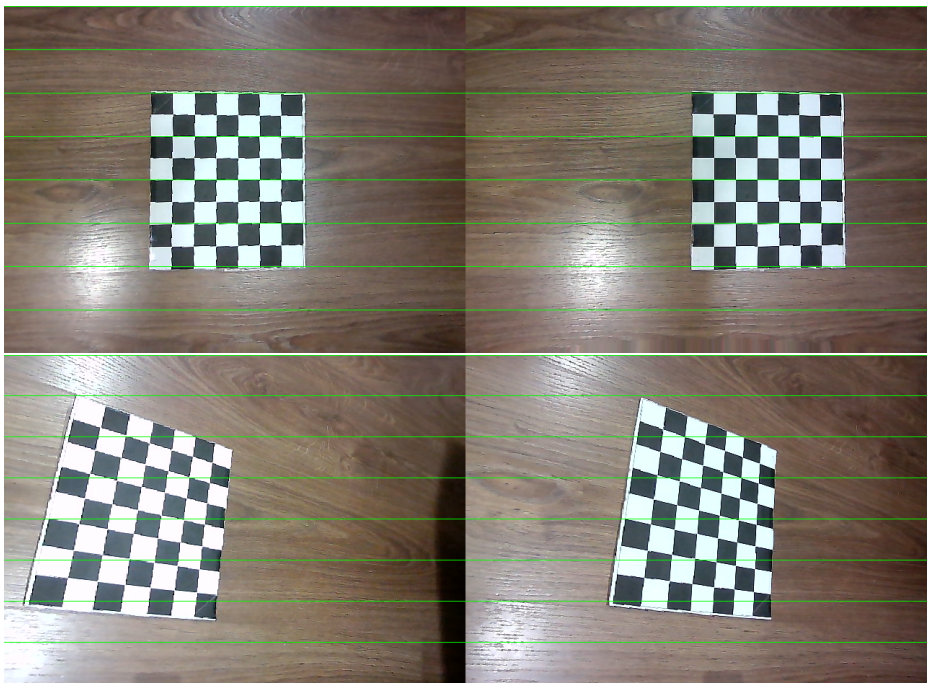


Figura 5.1.3: Pares de imágenes tras el proceso de rectificación y calibración. Se observa que las distorsiones han sido eliminadas y los planos imagen alineados correctamente.

5.2. Estimación de la pose

En este apartado se muestran los resultados obtenidos por el algoritmo A.1.2 usando el tablero como referencia. También se analizan y se discuten las diferentes medidas realizadas.

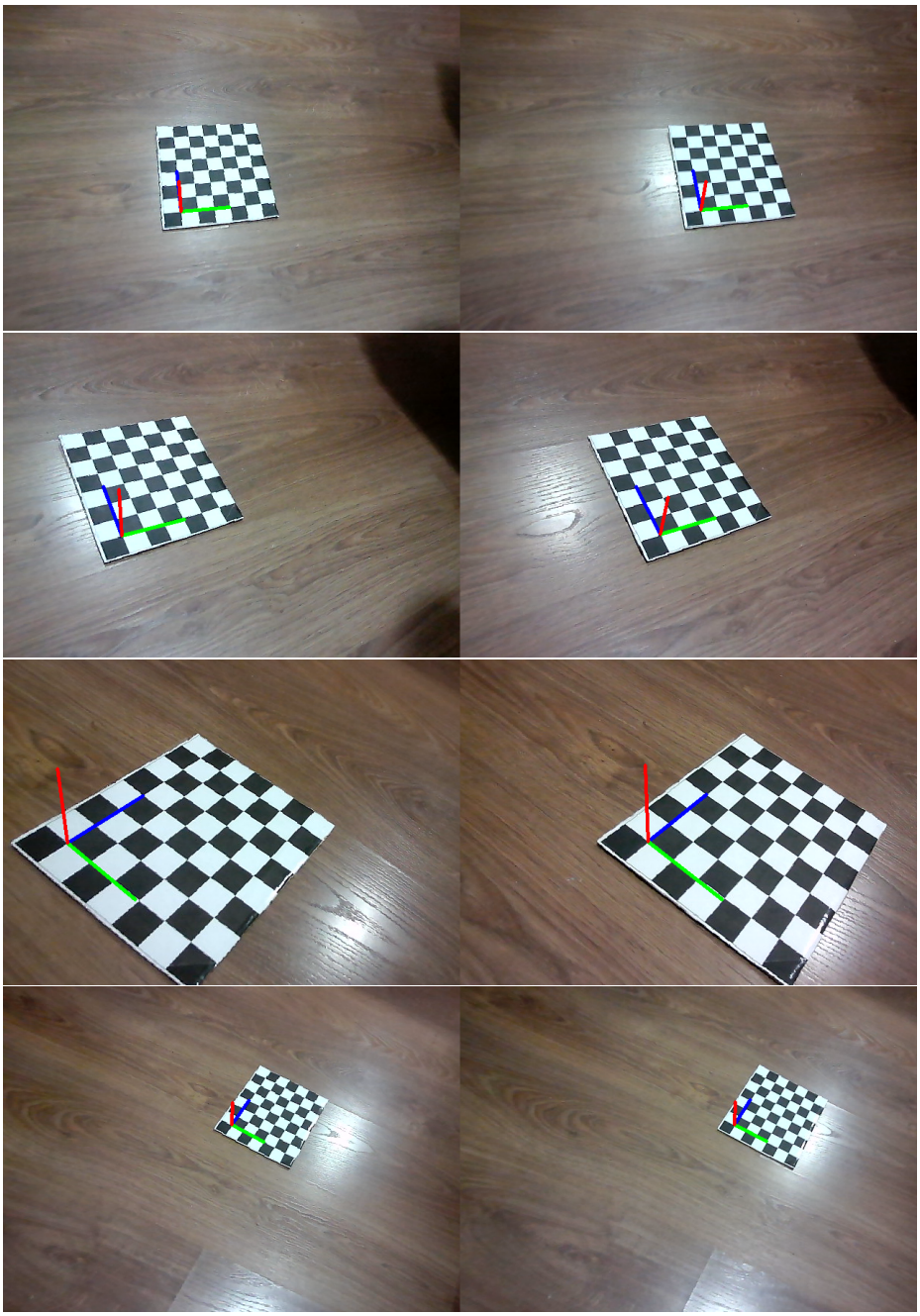


Figura 5.2.1: Estimación de la pose del tablero desde diferentes puntos de vista.

En la Fig. 5.2.1 se muestran varios pares de imágenes con la estimación de la pose del tablero. El motivo por el que se usa el tablero de nuevo es porque ya se ha usado anteriormente en el algoritmo de calibración y se puede estimar su posición conociendo sus dimensiones. La figura muestra cuatro capturas diferentes de la posición del par de cámaras respecto al tablero. Se puede observar que en

una de sus esquinas se ha dibujado un sistema de ejes cartesianos. El eje verde representa el eje X, el azul el eje Y y el rojo el eje Z. Esa esquina es la que sirve de referencia para estimar la pose del tablero, o lo que es lo mismo, la pose del sistema de cámaras respecto al tablero. Las distancias reales que hay entre las cámaras y el tablero en cada una de las imágenes son, respectivamente, 61 cm, 47 cm, 30 cm y 84 cm. Los resultados acerca de la pose del tablero de las imágenes de la Fig. 5.2.1 aparecen en la Tabla 5.2. Observando los datos obtenidos se puede destacar varios resultados relevante. Por un lado, la coordenada X muestra la posición en la coordenada horizontal del sistema de ejes cartesianos. La diferencia relativa que hay entre la posición de los ejes visto desde una cámara y otra es la disparidad. Por otro lado, la coordenada Y muestra la posición en la coordenada vertical. Se puede observar que los valores entre la cámara izquierda y derecha de cada par de imágenes prácticamente coinciden. Esto no es ninguna sorpresa, ya que este resultado verifica que los planos imágenes de ambas cámaras están alineados correctamente. Por último, la coordenada Z muestra la distancia que hay del sistema de cámaras al sistema de ejes del tablero. Comparando la media de los resultados obtenidos con los resultados exactos, se puede calcular el error relativo (ver Tabla 5.3). El error relativo general obtenido es menor al 5 %. El algoritmo también calcula el vector de rotación, representado por α (rotación en el eje X), β (rotación en el eje Y) y γ (rotación en el eje Z).

Imagen	Cámara	X (cm)	Y (cm)	Z (cm)	α (°)	β (°)	γ (°)
1	I	-9.4	6.4	61.2	-22.7	-35.7	-93.5
	D	2.3	6.4	62.3	-23.7	-33.6	-93.1
2	I	-14.3	4.4	41.1	-11.1	-49.1	-108.0
	D	-4.1	4.6	47.4	-12.1	-42.2	-108.1
3	I	-15.5	1.1	30.6	-36.2	-25.1	-46.7
	D	-4.0	1.0	32.3	-36.0	-21.3	-45.5
4	I	0.1	-8.3	84.9	-24.5	-15.9	-57.7
	D	11.4	-8.1	82.7	-28.7	-11.1	-56.9

Tabla 5.2: Resultados de la pose obtenidos de las imágenes de la Fig.5.2.1. Por cada par de imágenes se muestran las coordenadas y ángulos medidos por cada cámara.

Distancia real (cm)	Distancia medida (cm)	ε_{rel} (%)
61	61.8	1.31
47	44.3	5.74
30	31.5	5.0
84	83.8	0.24

Tabla 5.3: Comparación entre la distancia real y distancia medida.

5.3. Mapas de disparidad y profundidades

En este apartado se muestran los resultados obtenidos aplicando el algoritmo A.1.3. Para poner en práctica el algoritmo se ha montado una escena con varios objetos de diferentes tamaños distanciados entre sí (ver Fig. 5.3.1). Estos objetos se han puesto a diferentes distancias para determinar la bondad de la técnica seguida en este trabajo. En la propia imagen se indica los objetos usados y en la Tabla 5.4 las distancias reales entre éstos.

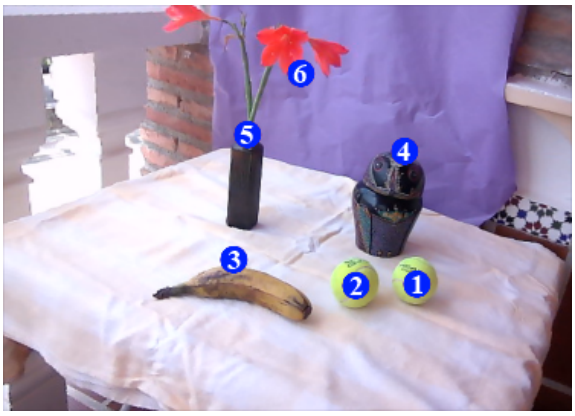


Figura 5.3.1: Escena con objetos a diferentes distancias.

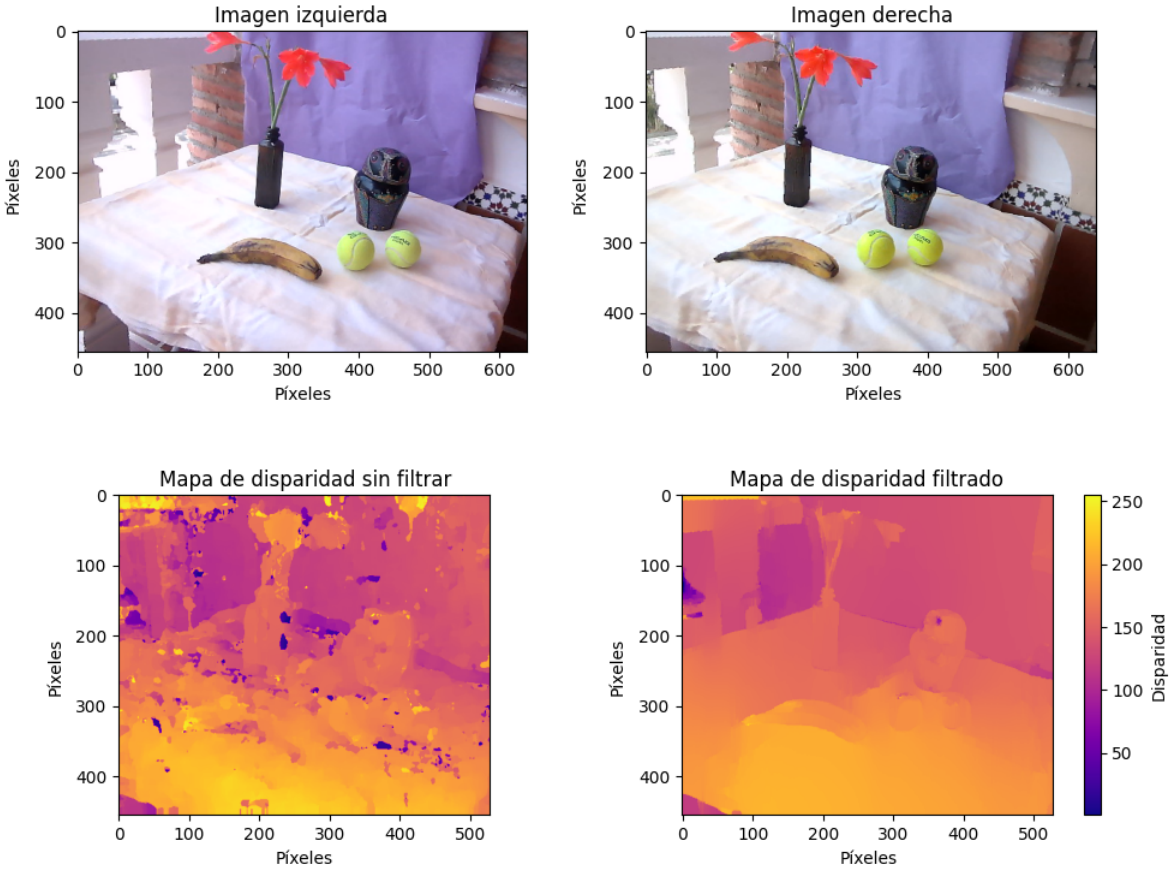


Figura 5.3.2: Arriba: Imágenes izquierda y derecha originales del escenario. Abajo: Mapas de disparidad sin filtrar y filtrado.

La Fig. 5.3.2 muestra los mapas de disparidad obtenidos, tanto sin filtrar como filtrado. Las dos gráficas superiores muestran las imágenes originales observadas por las cámaras izquierda y derecha, respectivamente. Las dos gráficas de abajo son los resultados obtenidos en los mapas de disparidad junto con una barra de colores, indicando el valor de la disparidad en cada caso, de 0 a 255. A la izquierda se muestra el mapa sin filtrar y a la derecha, el filtrado mediante el filtro WLS ya usado anteriormente.

Se pueden observar claramente las diferencias entre ambos resultados. Observando el mapa de disparidad sin filtrar, se puede observar que ciertas zonas de la imagen tienen un alto nivel de ruido. Este ruido se corresponde con valores excesivamente altos y bajos en el mapa generado. Esto es debido a las regiones donde el brillo o las texturas de los objetos no son idóneas para estimar el valor de la disparidad. Aún así, en este mapa se pueden distinguir los principales objetos que hay en escena. Por otro lado, el resultado obtenido aplicando el filtro WLS muestra la reducción del ruido respetando los bordes de los objetos y de forma general, el valor de la disparidad en cada píxel. En este caso se aprecia que la imagen está mucho más suavizada, consecuencia de usar el filtro. Sin embargo, usar el filtro puede perjudicar a la hora de determinar las distancias reales que hay entre los píxeles.

Para cada mapa de disparidad generado, el algoritmo reproyecta los valores en una imagen 3D, creando una nube de puntos. En la Fig. 5.3.3 se muestra las nubes de puntos 3D que se han generado a partir de los mapas de disparidad. La imagen superior se trata de la nube de puntos creada con el mapa de disparidad sin filtrar y la inferior con el filtro WLS. Las distancias que hay entre los puntos en ambos mapas tuvo que ser reescalada para devolver los valores en centímetros. Los resultados de las distancias entre los diferentes objetos se muestran en la Tabla 5.4. Las dos imágenes que se muestran son dos perspectivas diferentes de la escena. Analizando las imágenes, se puede apreciar la profundidad de los objetos y las separaciones que hay entre ellos. En general, los resultados que se obtienen con el mapa de puntos generado a partir del mapa de disparidades filtrado contienen menos ruido que los mapas generados sin filtrado. También se midió la separación aproximada entre los objetos para determinar la bondad de los valores que ofrece el algoritmo al generar la nube de puntos. Las distancias medidas son las mismas que las representadas en la Fig. 5.3.1. Comparando los resultados se puede ver que ambos mapas dan buenos resultados y cercanos a la realidad. Además, los resultados obtenidos por el mapa sin filtrar son más fieles a la realidad que el filtrado, aún con una mayor presencia de ruido. Esto puede deberse al proceso de filtrado que sigue el filtro WLS. La pérdida de ruido que consigue lo hace a costa de perder calidad en las medidas de los puntos.

El mayor inconveniente al generar el mapa de disparidades de un entorno es el ruido. Lugares con poca resolución u objetos de baja textura suelen dar problemas. Por ejemplo, objetos o superficies planas de un único color son una fuente de problemas para esta técnica. Una solución consiste en eliminar estas superficies indeseadas y detectar únicamente aquellos objetos que interesen. Para ello se realizaron varias pruebas con distintos tipos y a su vez se generaron mapas de disparidad para comprobar la calidad de los resultados. En la Fig. 5.3.4 se muestra un ejemplo de el resultado conseguido. En este caso se observan varios detalles. Las imágenes superiores muestran la visión de las cámaras izquierda y derecha, respectivamente. Los resultados se muestran de tal forma que la cámara izquierda detecta todas las clases de objetos que sea capaz de reconocer, mientras que la cámara de la derecha sólo detecta las plantas (esto mismo se podría realizar con cualquier tipo de clase). En ambas imágenes se pueden observar las clases que se detectan con sus respectivas probabilidades. Sin embargo en la imagen izquierda hay algunas detecciones erróneas con bajas probabilidades. Para evitar este problema, el algoritmo se encarga de almacenar las máscaras cuyas probabilidades son mayor a 0.7. Las imágenes inferiores son los mapas de disparidad con la máscara elegida aplicada (en este caso la de la planta detectada por la cámara derecha). Ahora el fondo y todos los demás objetos que no interesan no aparecen.

Figura 5.3.3: Nube de puntos 3D del escenario usando MeshLab. Arriba: Generado a partir del mapa de disparidad sin filtrar. Abajo: Generado a partir del mapa de disparidad con filtro WLS.

Puntos	Distancia real (cm)	Distancia sin filtrar (cm)	Distancia con filtro (cm)
1–2	10	9.8	10.2
2–3	15	14.5	12.9
2–4	20	20.4	18.5
4–5	25	24.9	22.7
5–6	14	13.0	12.4

Tabla 5.4: Comparación de distancias medidas entre los objetos.

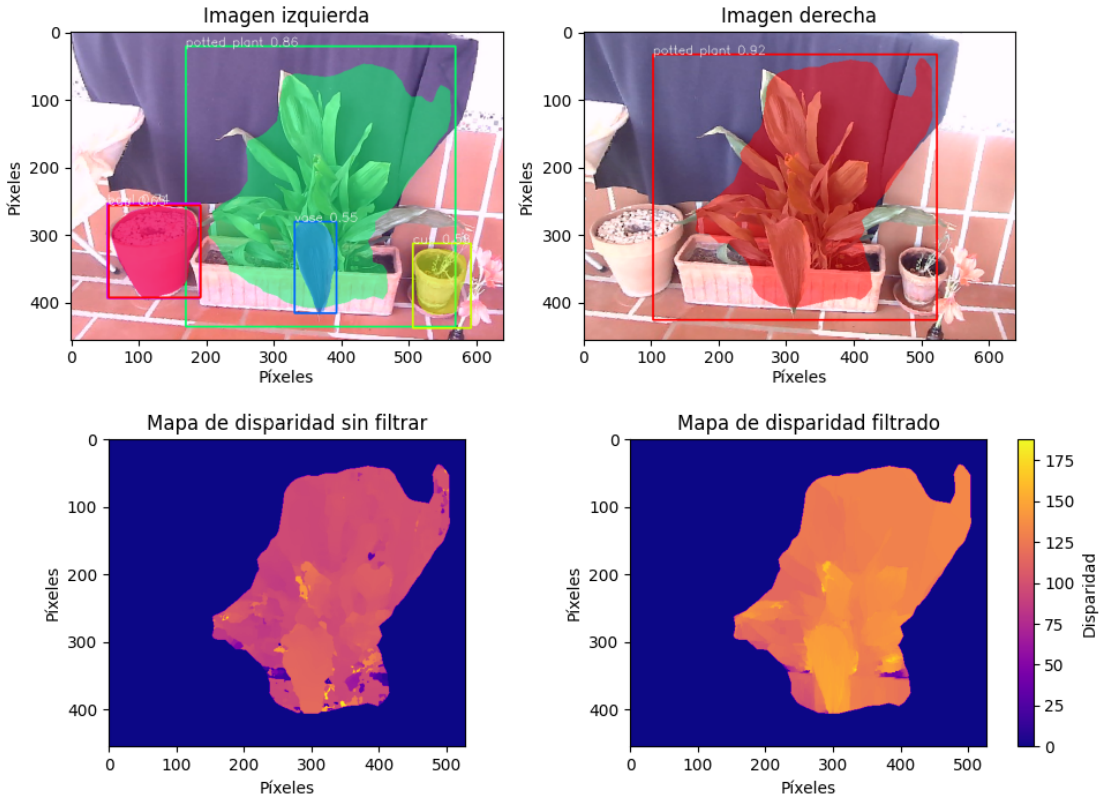


Figura 5.3.4: Ejemplo de detección y segmentación de objetos.

5.4. Aplicación con imágenes del rover Perseverance

Ya se ha visto como trabaja el algoritmo diseñado en un escenario con diferentes objetos. Sin embargo, esta técnica aún no se ha podido observar en situaciones reales de exploración espacial. El algoritmo A.1.4 diseñado hace uso de un par de imágenes obtenidas por el rover Perseverance (Fig. 5.4.1) en la superficie de Marte para poner en práctica lo desarrollado en este trabajo, la generación de mapas de disparidad y profundidad. En cuanto a la detección de objetos, hoy en día no hay disponible una base de datos robusta de imágenes de la superficie de Marte para desarrollar modelos de redes neuronales convolucionales que permitan la detección y segmentación de objetos. Por ejemplo, en Li *et al.* 2020 desarrollan un modelo que trabaja con las imágenes obtenidas por el rover Curiosity. Sin embargo, debido a que el conjunto de datos de imágenes válidas son insuficientes, usan métodos como transformaciones afines y combinaciones entre imágenes para aumentar la cantidad de datos. En este caso se centrará únicamente en la generación de mapas de disparidad y profundidad a partir del par de imágenes ofrecidas por el Perseverance.

Las imágenes usadas se han obtenido de la página oficial de la NASA y en ellas se muestra la superficie marciana con el helicóptero Ingenuity en escena. También se observan varias rocas de diferentes tamaños y a varias distancias. Las fotos se tomaron con las cámaras izquierda y derecha de la MastCam-Z.

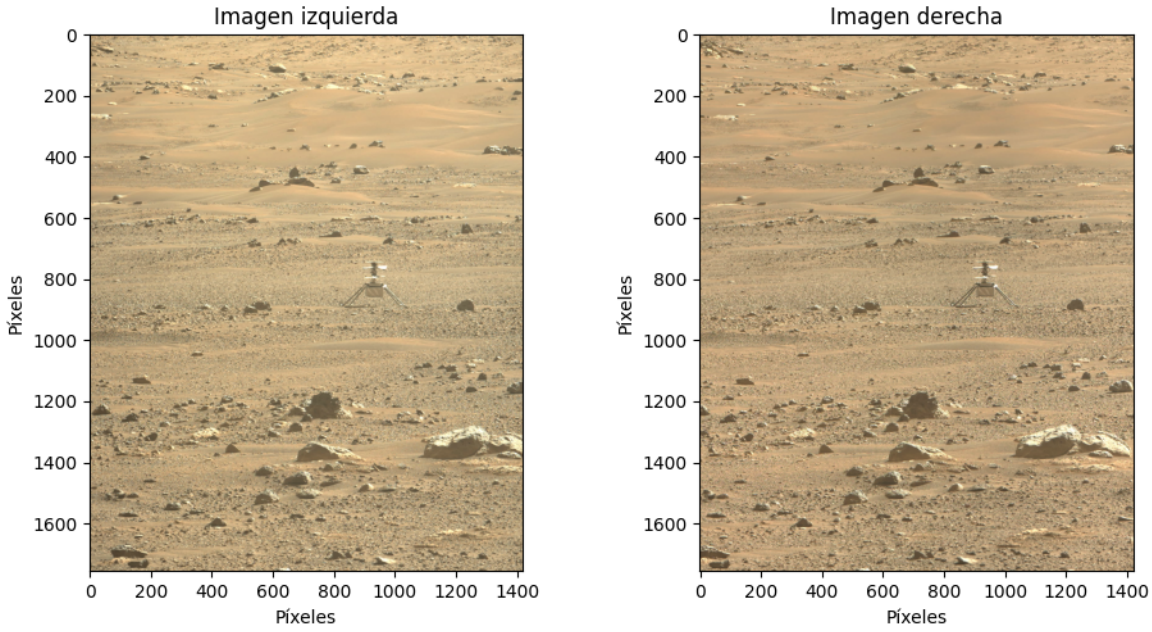


Figura 5.4.1: Imágenes obtenidas por la MastCam-Z del rover Perseverance en la superficie de Marte. Créditos: NASA/JPL.

Aplicando el algoritmo A.1.4 se genera un mapa de disparidad tanto sin filtrar como filtrado por el filtro WLS y un mapa de profundidades, tal y como se ha ido realizando durante todo el trabajo. La Fig. 5.4.2 muestra esto mismo. A la izquierda se visualizan los mapas de disparidad sin filtrar y filtrado. Se puede observar a ambos lados de las imágenes las bandas carentes de información producidas por los parámetros introducidos en la función `cv.stereoSGBM` a la hora de crear los mapas. También se aprecia ruido en los laterales, producido posiblemente porque en esas zonas no es capaz de determinar la disparidad correctamente. Sin embargo, se puede ver que el mapa de disparidad filtrado corrige parte de ese ruido, resultando en una imagen más limpia. A la derecha se muestra el mapa de profundidades generado a partir del mapa filtrado, ya que este al contener menos ruido produce menos errores. Además, la imagen ha sido recortada por su margen derecho para eliminar el ruido. Para generar el mapa de profundidades es necesario conocer tanto la focal de las cámaras como la baseline. La baseline está fijada en 24.4 cm, mientras que las focales con las que trabaja MastCam-Z varían entre 26 mm y 110 mm (Bell *et al.* 2021). En este par de imágenes, las cámaras trabajaron con una focal de 110 mm. Por tanto, siguiendo la Ec. 3.13 se puede determinar la distancia a cada píxel de la imagen. Sin embargo, los valores iniciales variaban entre 1 y 13 m. Estos números no concuerdan con la perspectiva de la imagen, donde las rocas más lejanas parecen estar a mucha más distancia. Para solventar este problema se puede tener un objeto de referencia, que conocidas sus dimensiones y/o distancia a las cámaras, sirva para estimar la distancia a cualquier otro punto de la imagen. Esto se traduce en que habría que añadir un factor de escalado a la ecuación 3.13. En la Fig. 5.4.2, se ha supuesto que el helicóptero Ingenuity está situado a 20 m. Sabiendo esto, se puede generar el mapa de profundidades que se observa aplicando el factor correspondiente. Los resultados muestran unos valores más realistas que varían entre 8 y 90 m. En la imagen se puede observar claramente la silueta del helicóptero con gran detalle, desde su base hasta las aspas.

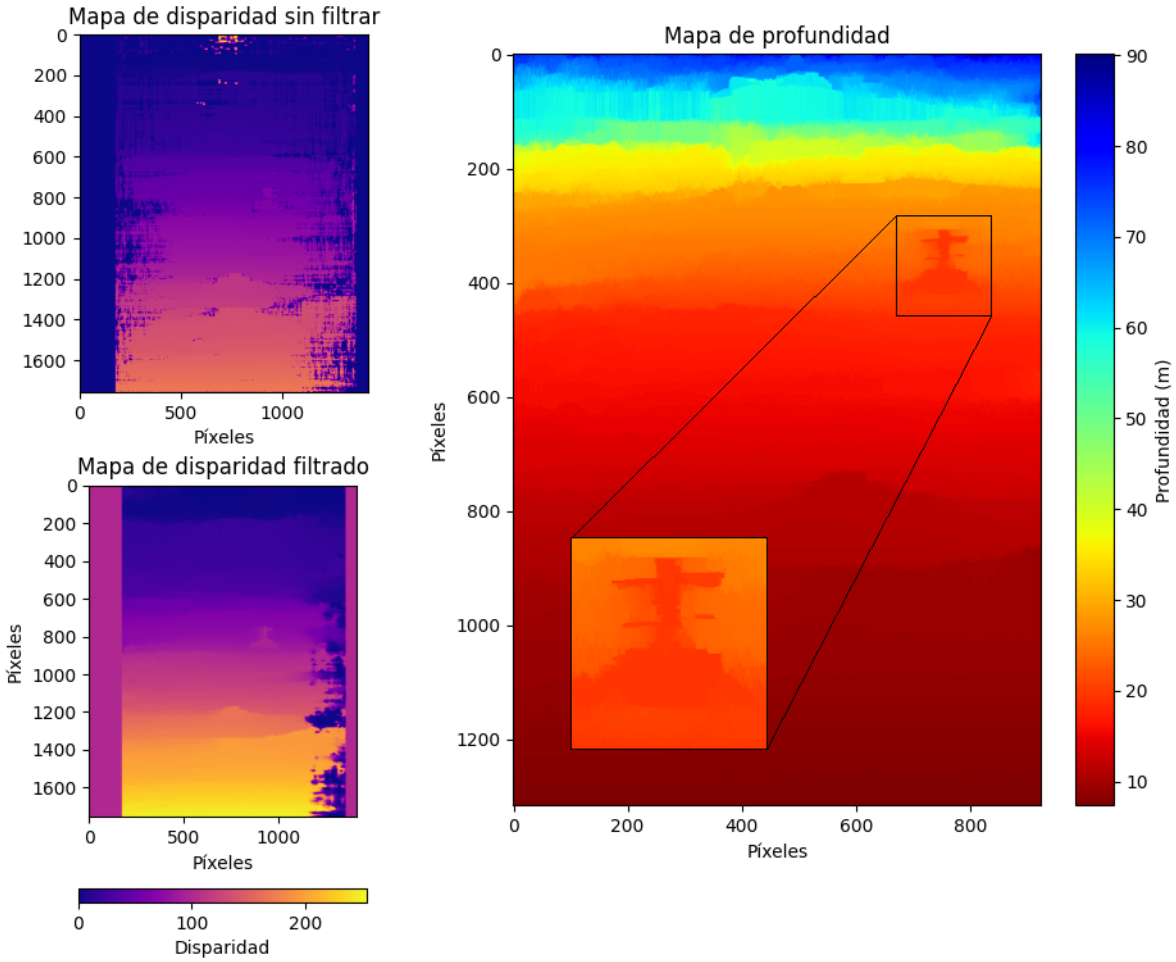


Figura 5.4.2: Mapas de disparidad y profundidad de las imágenes del rover Perseverance.

Capítulo 6

Conclusiones

La visión por ordenador ha cobrado mucha importancia estos últimos años en varios campos de la ciencia y la tecnología: robótica, automoción y últimamente, espacio. En este trabajo se ha realizado una revisión acerca del estado del arte en algoritmos de visión por ordenador e Inteligencia Artificial (IA). También se ha diseñado un conjunto de algoritmos para estimar la pose de un sistema de cámaras estéreo y generar mapas de profundidad usando odometría visual y segmentación de objetos. Los algoritmos han conseguido calibrar las cámaras, estimar la pose con buena precisión y realizar mapas de disparidad y profundidad con la capacidad de detectar y clasificar objetos del entorno. Para ellos se ha programado en Python y se ha usado principalmente la librería de OpenCV, para la visión por ordenador, y PixelLib, para la detección, clasificación y segmentación de objetos. Las principales conclusiones son las siguientes:

1. Se ha analizado el estado del arte en algoritmos de visión e IA para robótica de exploración. La odometría visual y SLAM destacan por ser las técnicas más usadas para estimar la pose de objetos y para generar mapas del entorno. Se ha realizado una revisión sobre la evolución de algoritmos de detección y descripción de rasgos. Existe una gran variedad de técnicas, desde el detector de esquinas de Harris hasta los más modernos actualmente, como ORB-SLAM2. También se han explorado los distintos métodos de detección y segmentación de objetos, siendo R-CNN y YOLO los más populares. Por último, se ha hecho una revisión de las diferentes técnicas de visión e IA llevadas a cabo en misiones de exploración espacial, como las misiones MER, MSL, Mars2020 o futuras.

2. Se han diseñado un conjunto de algoritmos que permiten estimar la pose de un sistema de cámaras estéreo a partir de un objeto de referencia, con la capacidad de detectar y clasificar objetos del entorno. Para ello, anteriormente se ha seguido un proceso de calibración para rectificar las cámaras.

3. El proceso de calibración y rectificación es un paso fundamental y necesario para eliminar las distorsiones que puedan tener las cámaras. Para ello, en este trabajo se ha usado un tablero de ajedrez como objeto de referencia. Se han obtenido adecuadamente los parámetros intrínsecos y extrínsecos de las cámaras, corrigiendo las distorsiones correctamente con un error de calibración menor del 1 %.

4. La estimación de la pose se ha realizado con el mismo tablero de ajedrez. Se ha podido comprobar que el algoritmo ofrece buenos resultados acerca de la posición y rotación del tablero,

sirviendo como referencia para estimar la pose del sistema de cámaras. Los valores obtenidos para varias pruebas arrojan un error medio relativo en la distancia del 3 %.

5. La generación de mapas de disparidad es una técnica potente que puede ofrecer grandes resultados en la determinación de la distancia de cada píxel de la imagen, en la creación de mapas de profundidad y nubes de puntos 3D. Se ha demostrado que ofrece unos resultados muy cercanos a la realidad, midiendo la separación entre diferentes objetos de la escena. Sin embargo, es susceptible a las propias cámaras y a las condiciones del entorno. La luminosidad, objetos de poca textura o el movimiento relativo entre las cámaras perjudica notablemente la calidad de los resultados. El filtro WLS usado ha demostrado eliminar gran parte del ruido generado inicialmente, preservando los bordes de los objetos. No obstante, los resultados en los valores de las distancias son mejores sin el uso del filtro, debido a que el filtrado suaviza los valores de los píxeles, perdiendo calidad en la medida. En cuanto a la segmentación de objetos, se ha probado que el algoritmo es capaz de detectar una determinada clase de objeto y enmascararla, eliminando así aquellos objetos innecesarios de la imagen.

6. Se ha podido probar lo desarrollado en este trabajo con un par de imágenes tomadas por el rover Perseverance. En este caso se ha podido crear un mapa de disparidad y de profundidad de buena calidad, pudiendo ser capaces de determinar las distancias de los píxeles con buena precisión conociendo los parámetros necesarios.

6.1. Trabajos futuros

El papel que tienen los algoritmos de visión e IA en varios campos de la ciencia y la tecnología es cada vez más importante, con grandes aplicaciones en sectores como el espacio, la robótica, la automoción y la navegación autónoma. Tras la realización de estos algoritmos, hay varias direcciones que se pueden tomar de cara a futuros trabajos.

Por un lado, estos algoritmos se pueden implementar en un rover para ponerlos en práctica en un entorno real. Se podría analizar el error en la precisión de la pose del rover a medida que avanza por una superficie complicada y compararlo con otras técnicas de posición, como la odometría de ruedas. Para ello, existen varios kits de desarollo enfocados a la robótica e IA, como el módulo Jetson Nano, diseñado para trabajar con esta clase de algoritmos en rovers de exploración y robots.

Por otro lado, también se pueden explorar nuevas técnicas de detección de esquinas que permitan obtener mejores resultados en la estimación de la pose de los objetos. Para mejorar la calidad de los mapas de disparidad y reducir el ruido, también se pueden incorporar otros sensores encargados de medir la distancia, como sensores LIDAR o infrarrojos. En cuanto a la detección de objetos, si bien se quiere emplear en misiones de exploración espacial, es necesario utilizar un modelo de reconocimiento de objetos adecuado. Para ello, se podría investigar y desarrollar modelos encargados en la detección y segmentación de rocas en la superficie marciana a partir de conjuntos de imágenes estéreo. Esto permitiría detectarlas, clasificarlas y obtener información acerca de su tamaño, haciendo posible esquivarlas si suponen un peligro para la navegación del rover.

Bibliografía

Aqel, M. *et al.* (2016). "Review of visual odometry: types, approaches, challenges, and applications". En: *SpringerPlus* 5.1897. DOI: http://dx.doi.org/10.1186/s40064-016-3573-7.

Bay, Herbert *et al.* (2008). "Speeded-Up Robust Features (SURF)". En: *Computer Vision and Image Understanding* 110.3. Similarity Matching in Computer Vision and Multimedia, págs. 346-359. ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2007.09.014. URL: https://www.sciencedirect.com/science/article/pii/S1077314207001555.

Bell, J. *et al.* (feb. de 2021). "The Mars 2020 Perseverance Rover Mast Camera Zoom (Mastcam-Z) Multispectral, Stereoscopic Imaging Investigation". En: *Space Science Reviews* 217. DOI: 10.1007/s11214-020-00755-x.

Birchfield, S. y C. Tomasi (1998). "Depth discontinuities by pixel-to-pixel stereo". En: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, págs. 1073-1080. DOI: 10.1109/ICCV.1998.710850.

Calonder, Michael *et al.* (sep. de 2010). "BRIEF: Binary Robust Independent Elementary Features". En: vol. 6314, págs. 778-792. ISBN: 978-3-642-15560-4. DOI: 10.1007/978-3-642-15561-1_56.

Cerilli, Gianluca y Martin Zwick (2019). "VISUAL SERVOING FOR SAMPLE TUBE DETECTION AND PICK-UP ON MARS". En:

Chatila, Raja y Jean-Paul Laumond (abr. de 1985). "Position referencing and consistent world modeling for mobile robots". En: vol. Vol. 2, págs. 138-145. DOI: 10.1109/ROBOT.1985.1087373.

Cheng, Y., M. Maimone y L. Matthies (2005). "Visual odometry on the Mars Exploration Rovers". En: *2005 IEEE International Conference on Systems, Man and Cybernetics* 1, 903-910 Vol. 1. DOI: 10.1109/ICSMC.2005.1571261.

Churchill, W. y P. Newman (2012). "Practice makes perfect? Managing and leveraging visual experiences for lifelong navigation". En: *2012 IEEE International Conference on Robotics and Automation*, págs. 4525-4532.

Di, Kaichang *et al.* (feb. de 2013). "Automated Rock Detection and Shape Analysis from Mars Rover Imagery and 3D Point Cloud Data". En: *Journal of Earth Science* 24. DOI: 10.1007/s12583-013-0316-3.

Duda, Alexander y Udo Frese (sep. de 2018). "Accurate Detection and Localization of Checkerboard Corners for Calibration". En:

Durrant-Whyte, H. y T. Bailey (2006a). "Simultaneous localization and mapping: part I". En: *IEEE Robotics Automation Magazine* 13.2, págs. 99-110. DOI: 10.1109/MRA.2006.1638022.

— (2006b). "Simultaneous localization and mapping: part I". En: *IEEE Robotics Automation Magazine* 13.2, págs. 99-110. DOI: 10.1109/MRA.2006.1638022.

Engel, Jakob, Vladlen Koltun y Daniel Cremers (2018). "Direct Sparse Odometry". En: *IEEE transactions on pattern analysis and machine intelligence* 40(3), págs. 611-625. DOI: https://doi.org/10.1109/TPAMI.2017.2658577.

Engel, Jakob, Thomas Schöps y Daniel Cremers (2014). "LSD-SLAM: Large-Scale Direct Monocular SLAM". En: *Computer Vision – ECCV 2014*. Ed. por David Fleet *et al.* Cham: Springer International Publishing, págs. 834-849. ISBN: 978-3-319-10605-2.

Farbman, Zeev *et al.* (ago. de 2008). "Edge-Preserving Decompositions for Multi-Scale Tone and Detail Manipulation". En: *ACM Trans. Graph.* 27.3, págs. 1-10. ISSN: 0730-0301. DOI: 10.1145/1360612.1360666. URL: https://doi.org/10.1145/1360612.1360666.

Forster, Christian, Matia Pizzoli y Davide Scaramuzza (mayo de 2014). "SVO: Fast Semi-Direct Monocular Visual Odometry". En: DOI: 10.1109/ICRA.2014.6906584.

Garg, Chirayu y Utkarsh Jain (s.f.). *Stereo Visual Odometry*. URL: https://github.com/cgarg92/Stereo-visual-odometry.

Girshick, Ross (2015). *Fast R-CNN*. arXiv: 1504.08083 [cs.CV].

Girshick, Ross *et al.* (2014). *Rich feature hierarchies for accurate object detection and semantic segmentation*. arXiv: 1311.2524 [cs.CV].

Harris, C. y M. Stephens (1988). "A Combined Corner and Edge Detector". En: *Proc. AVC*, págs. 23.1-23.6. DOI: 10.5244/C.2.23.

Hartley, Richard y Andrew Zisserman (2004). *Multiple View Geometry in Computer Vision*. 2.ª ed. Cambridge University Press. DOI: 10.1017/CBO9780511811685.

He, Kaiming *et al.* (2018). *Mask R-CNN*. arXiv: 1703.06870 [cs.CV].

He, Ming *et al.* (mayo de 2020). "A review of monocular visual odometry". En: *The Visual Computer* 36. DOI: 10.1007/s00371-019-01714-6.

Horn, Joachim y Günther Schmidt (1995). "Continuous localization of a mobile robot based on 3D-laser-range-data, predicted sensor images, and dead-reckoning". En: *Robotics and Autonomous Systems* 14.2. Research on Autonomous Mobile Robots, págs. 99-118. ISSN: 0921-8890. DOI: https://doi.org/10.1016/0921-8890(94)00023-U.

Howard, A. (2008). "Real-time stereo visual odometry for autonomous ground vehicles". En: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, págs. 3946-3952.

Imperoli, Marco y Alberto Pretto (2015). "D^{2}CO: Fast and Robust Registration of 3D Textureless Objects Using the Directional Chamfer Distance". En: *Computer Vision Systems*. Ed. por Lazaros Nalpantidis *et al.* Cham: Springer International Publishing, págs. 316-328. ISBN: 978-3-319-20904-3.

Kerl, C., Jürgen Sturm y D. Cremers (2013). "Dense visual SLAM for RGB-D cameras". En: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, págs. 2100-2106.

Kerner, Hannah *et al.* (ene. de 2019). "Novelty Detection for Multispectral Images with Application to Planetary Exploration". En: DOI: 10.5281/zenodo.1486196.

Laureano, Gustavo Teodoro *et al.* (2015). "Chapter 34 - A topological approach for detection of chessboard patterns for camera calibration". En: *Emerging Trends in Image Processing, Computer Vision and Pattern Recognition*. Ed. por Leonidas Deligiannidis y Hamid R. Arabnia. Boston: Morgan Kaufmann, págs. 517-531. ISBN: 978-0-12-802045-6. DOI: https://doi.org/10.1016/B978-0-12-802045-6.00034-X. URL: https://www.sciencedirect.com/science/article/pii/B978012802045600034X.

Li, Jialun *et al.* (sep. de 2020). "Autonomous Martian rock image classification based on transfer deep learning methods". En: *Earth Science Informatics* 13. DOI: 10.1007/s12145-019-00433-9.

Lingemann, Kai *et al.* (jun. de 2005). "High-speed laser localization for mobile robots". En: *Robotics and Autonomous Systems* 51, págs. 275-296. DOI: 10.1016/j.robot.2005.02.004.

Lowe, D.G. (2004). "Distinctive Image Features from Scale-Invariant Keypoints". En: *International Journal of Computer Vision* 60, págs. 91-110. DOI: 10.1023/B:VISI.0000029664.99615.94.

Maimone, Mark, Yang Cheng y Larry Matthies (2007). "Two years of Visual Odometry on the Mars Exploration Rovers". En: *Journal of Field Robotics* 24.3, págs. 169-186. DOI: https://doi.org/

10.1002/rob.20184. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.20184. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.20184.

Maki, J. N. *et al.* (2003). "Mars Exploration Rover Engineering Cameras". En: *Journal of Geophysical Research: Planets* 108.E12. DOI: https://doi.org/10.1029/2003JE002077. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2003JE002077. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2003JE002077.

Matthies, Larry *et al.* (oct. de 2007). "Computer Vision on Mars". En: *International Journal of Computer Vision* 75, págs. 67-92. DOI: 10.1007/s11263-007-0046-z.

Mohta, Kartik *et al.* (dic. de 2017). "Fast, Autonomous Flight in GPS-Denied and Cluttered Environments". En: *Journal of Field Robotics* 35. DOI: 10.1002/rob.21774.

Moravec, H. (1980). "Obstacle avoidance and navigation in the real world by a seeing robot rover". Tesis doct. Stanford University, Stanford, CA.

Mur-Artal, Raúl, J. M. M. Montiel y Juan D. Tardós (2015). "ORB-SLAM: A Versatile and Accurate Monocular SLAM System". En: *IEEE Transactions on Robotics* 31.5, págs. 1147-1163. DOI: 10.1109/TRO.2015.2463671.

Mur-Artal, Raúl y Juan D. Tardós (2017). "ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras". En: *IEEE Transactions on Robotics* 33.5, págs. 1255-1262. DOI: 10.1109/TRO.2017.2705103.

Newcombe, Richard A., S. Lovegrove y A. Davison (2011). "DTAM: Dense tracking and mapping in real-time". En: *2011 International Conference on Computer Vision*, págs. 2320-2327.

Nistér, D., O. Naroditsky y J. Bergen (2004). "Visual odometry". En: *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.* 1, págs. I-I.

Raghavan, D. y B. Rao (oct. de 2020). "Computer Vision for Space Exploration". En: *INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT)* 09, págs. 68-70.

Redmon, Joseph, Santosh Divvala *et al.* (2016). *You Only Look Once: Unified, Real-Time Object Detection.* arXiv: 1506.02640 [cs.CV].

Redmon, Joseph y Ali Farhadi (2016). *YOLO9000: Better, Faster, Stronger.* arXiv: 1612.08242 [cs.CV].

— (2018). *YOLOv3: An Incremental Improvement.* arXiv: 1804.02767 [cs.CV].

Ren, Shaoqing *et al.* (2016). *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* arXiv: 1506.01497 [cs.CV].

— (2017). "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6, págs. 1137-1149. DOI: 10.1109/TPAMI.2016.2577031.

Rosten, E. y T. Drummond (2006). "Machine learning for high-speed corner detection". English. En: *Computer Vision - ECCV 2006, Proceedings, Part 1 - Lecture Notes in Computer Science.* Ed. por Ales Leonardis, Horst Bischof y Axel Pinz. Vol. 3951. European Conference on Computer Vision 2006, ECCV 2006 ; Conference date: 07-05-2006 Through 13-05-2006. Germany: Springer-Verlag London Ltd., págs. 430-443. ISBN: 3540338322. DOI: 10.1007/11744023. URL: https://link.springer.com/book/10.1007/11744023.

Rublee, Ethan *et al.* (2011). "ORB: An efficient alternative to SIFT or SURF". En: *2011 International Conference on Computer Vision*, págs. 2564-2571.

Russakovsky, Olga *et al.* (2015). *ImageNet Large Scale Visual Recognition Challenge.* arXiv: 1409.0575 [cs.CV].

Scaramuzza, Davide y Friedrich Fraundorfer (dic. de 2011). "Visual Odometry [Tutorial]". En: *IEEE Robot. Automat. Mag.* 18, págs. 80-92. DOI: 10.1109/MRA.2011.943233.

Servières, Myriam *et al.* (2021). "Visual and Visual-Inertial SLAM: State of the Art, Classification, and Experimental Benchmarking". En: *Journal of Sensors* 2021, pág. 26. DOI: 10.1155/2021/2054828.

Shaw, Andy *et al.* (2013). "Robust visual odometry for space exploration". En: *12th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*.

Shi, Jianbo y Tomasi (1994). "Good features to track". En: *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, págs. 593-600. DOI: 10.1109/CVPR.1994.323794.

Takeshi, T. (2007). "2 D localization of outdoor mobile robots using 3 D laser range data". En: *Carnegie Mellon University*.

Tao, Y. y J. -P. Muller (sep. de 2013). "New geological products from MSL using machine vision". En: *European Planetary Science Congress*, EPSC2013-606.

Teng, Chin-Hung, Kai-Yuan Chuo y Chen-Yuan Hsieh (nov. de 2018). "Reconstructing Three-Dimensional Models of Objects Using a Kinect Sensor". En: *Vis. Comput.* 34.11, págs. 1507-1523. ISSN: 0178-2789.

Townson, Daniel, Mark Woods y Stuart Carnochan (jun. de 2018). "EXOMARS VISLOC – THE INDUSTRIALISED, VISUAL LOCALISATION SYSTEM FOR THE EXOMARS ROVER". En:

Whitney Clavin, JPL (s.f.). *Rover Leaves Tracks in Morse Code*. URL: https://mars.nasa.gov/news/1329/rover-leaves-tracks-in-morse-code/?site=msl.

Zhang, Zhengyou (2014). "Epipolar Geometry". En: *Computer Vision: A Reference Guide*. Ed. por Katsushi Ikeuchi. Boston, MA: Springer US, págs. 247-258. ISBN: 978-0-387-31439-6. DOI: 10.1007/978-0-387-31439-6_128. URL: https://doi.org/10.1007/978-0-387-31439-6_128.

Apéndice A

Anexos

A.1. Códigos

A.1.1. Algoritmo de calibración

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

def errorCalib(objpoints, rvecsL, rvecsR, tvecsL, tvecsR, mtxL, mtxR, distL, distR):

    mean_errorL = 0
    mean_errorR = 0

    for i in range(len(objpoints)):
        imgpointsL2, _ = cv2.projectPoints(objpoints[i], rvecsL[i], tvecsL[i], mtxL, distL)
        imgpointsR2, _ = cv2.projectPoints(objpoints[i], rvecsR[i], tvecsR[i], mtxR, distR)

        errorL = cv2.norm(imgpointsL[i], imgpointsL2, cv2.NORM_L2) / len(imgpointsL2)
        errorR = cv2.norm(imgpointsR[i], imgpointsR2, cv2.NORM_L2) / len(imgpointsR2)

        mean_errorL += errorL
        mean_errorR += errorR

    return mean_errorL, mean_errorR


criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 60, 0.000001)

objp = np.zeros((6*7, 3), np.float32)
objp[:,:2] = np.mgrid[0:7, 0:6].T.reshape(-1,2)

objpoints = []
imgpointsL = []
imgpointsR = []
disparity = []

size = (480, 640)

```

```
print('Iniciando camaras...')

imagesL = cv2.VideoCapture(0)
imagesR = cv2.VideoCapture(2)

photos = 1

print('Buscando tablero de ajedrez...')

while True:

    trueL, imgL = imagesL.read()
    trueR, imgR = imagesR.read()

    grayL = cv2.cvtColor(imgL, cv2.COLOR_BGR2GRAY)
    grayR = cv2.cvtColor(imgR, cv2.COLOR_BGR2GRAY)

    istrueL, cornL = cv2.findChessboardCorners(grayL, (7, 6),
                                               cv2.CALIB_CB_ADAPTIVE_THRESH +
                                               cv2.CALIB_CB_FAST_CHECK +
                                               cv2.CALIB_CB_NORMALIZE_IMAGE)

    istrueR, cornR = cv2.findChessboardCorners(grayR, (7, 6),
                                               cv2.CALIB_CB_ADAPTIVE_THRESH +
                                               cv2.CALIB_CB_FAST_CHECK +
                                               cv2.CALIB_CB_NORMALIZE_IMAGE)

    if istrueL and istrueR:

        objpoints.append(objp)

        cornersL = cv2.cornerSubPix(grayL, cornL, (11,11), (-1, -1), criteria)
        cornersR = cv2.cornerSubPix(grayR, cornR, (11, 11), (-1, -1), criteria)

        imgpointsL.append(cornersL)
        imgpointsR.append(cornersR)

        cv2.drawChessboardCorners(imgL, (7,6), cornersL, trueL)
        cv2.drawChessboardCorners(imgR, (7,6), cornersR, trueR)

        print(photos)
        photos = photos + 1

    cv2.imshow('cornersL', imgL)
    cv2.imshow('cornersR', imgR)

    if cv2.waitKey(2000) & 0xff == ord('q'):
        break

print('Calibrando las camaras individualmente...')

retL, mtxL, distL, rvecsL, tvecsL = cv2.calibrateCamera(objpoints, imgpointsL, grayL.
    shape[::-1], None, None)
retR, mtxR, distR, rvecsR, tvecsR = cv2.calibrateCamera(objpoints, imgpointsR, grayR.
    shape[::-1], None, None)

print('Optimizando la matriz de ambas camaras...')

hL,  wL = imgL.shape[:2]
hR,  wR = imgR.shape[:2]
```

```
93
94  newcameramtxL, roiL = cv2.getOptimalNewCameraMatrix(mtxL, distL, (wL, hL), 1, (wL, hL
        ), True)
95  newcameramtxR, roiR = cv2.getOptimalNewCameraMatrix(mtxR, distR, (wR, hR), 1, (wR, hR
        ), True)
96
97  print('Calculando el error del calibrado individual...')
98
99  mean_errorL, mean_errorR = errorCalib(objpoints, rvecsL, rvecsR, tvecsL, tvecsR, mtxL
        , mtxR, distL, distR)
100
101 print("Error total de la calibracion de la camara izquierda: {}".format(mean_errorL /
         len(objpoints)))
102 print("Error total de la calibracion de la camara derecha: {}".format(mean_errorR /
        len(objpoints)))
103
104 print('Calibracion estereo...')
105
106 retval, mtxL, distL, mtxR, distR, R, T, E, F = cv2.stereoCalibrate(objpoints,
        imgpointsL, imgpointsR, mtxL, distL, mtxR, distR, size)
107
108 retval, mtxL, distL, mtxR, distR, R, T, E, F, rms_error1 = cv2.
        stereoCalibrateExtended(objpoints, imgpointsL, imgpointsR, mtxL, distL, mtxR,
        distR, size, R, T)
109
110 rms_error = np.sum(rms_error1)/len(rms_error1)
111 print("Error total de la calibracion estereo: {}".format(rms_error))
112
113 print('Rectificacion estereo...')
114
115 RL, RR, PL, PR, Q, roiL1, roiR1 = cv2.stereoRectify(mtxL, distL, mtxR, distR, size, R
        , T)
116
117 print('Eliminando distorsion...')
118
119 imagesL = cv2.VideoCapture(0)
120 imagesR = cv2.VideoCapture(2)
121
122 # Eliminamos la distorsion
123 mapL1, mapL2 = cv2.initUndistortRectifyMap(mtxL, distL, RL, PL, (640, 480), m1type=
        cv2.CV_32FC1)
124 mapR1, mapR2 = cv2.initUndistortRectifyMap(mtxR, distR, RR, PR, (640, 480), m1type=
        cv2.CV_32FC1)
125
126 imagesL.release()
127 imagesR.release()
128
129 cv.destroyAllWindows()
```

A.1.2. Algoritmo de estimación de pose

```
1  import numpy as np
2  import cv2 as cv
3  import time
4
5  # Funcion que elimina y prepara las imagenes
6  def imgrect(img0, img1, mapL1, mapL2, mapR1, mapR2, roiL, roiR):
7
8      # Elimino las distorsiones de ambas camaras
```

```
    dst0 = cv.remap(img0, mapL1, mapL2, interpolation=cv.INTER_NEAREST, borderMode=cv.BORDER_CONSTANT)
    dst1 = cv.remap(img1, mapR1, mapR2, interpolation=cv.INTER_NEAREST, borderMode=cv.BORDER_CONSTANT)

    # Recortamos la imagen

    dst0 = dst0[0:455, 0:640]
    dst1 = dst1[0:455, 0:640]

    gray0 = cv.cvtColor(dst0, cv.COLOR_RGB2GRAY)
    gray1 = cv.cvtColor(dst1, cv.COLOR_RGB2GRAY)

    return dst0, dst1, gray0, gray1


# Funcion que dibuja los corners y los puntos de los ejes para dibujar un sistema de ejes 3D
def draw(img, corners, imgpts):
    corner = tuple(corners[0].ravel())

    img = cv.line(img, corner, tuple(imgpts[0].ravel()), (255, 0, 0), 3)
    img = cv.line(img, corner, tuple(imgpts[1].ravel()), (0, 255, 0), 3)
    img = cv.line(img, corner, tuple(imgpts[2].ravel()), (0, 0, 255), 3)

    return img

criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 120, 0.00001)

objpL = np.zeros((6 * 7, 3), np.float32)
objpL[:, :2] = np.mgrid[0:7, 0:6].T.reshape(-1, 2)

objpR = np.zeros((6 * 7, 3), np.float32)
objpR[:, :2] = np.mgrid[0:7, 0:6].T.reshape(-1, 2)

axis = np.float32([[3, 0, 0], [0, 3, 0], [0, 0, -3]]).reshape(-1, 3)

t1 = time.time()

imagesL = cv.VideoCapture(0)
imagesR = cv.VideoCapture(2)

t2 = time.time()
print('Tiempo de encendido de camaras: ', t2-t1, 's')
photo = 1

while True:

    t_inicio = time.time()

    ret0, imgL = imagesL.read()
    ret1, imgR = imagesR.read()

    t1 = time.time()


    # Rectificacion de imagenes
    dstL, dstR, grayL, grayR = imgrect(imgL, imgR, mapL1, mapL2, mapR1, mapR2, roiL1, roiR1)
    t2 = time.time()
```

```
    print('Tiempo de rectificacion de las imagenes de ambas camaras: ', t2 - t1, 's')

    t1 = time.time()

    retL, cornersL = cv.findChessboardCorners(grayL, (7, 6), cv.CALIB_CB_ACCURACY)  # Busca los corners de cada imagen
    retR, cornersR = cv.findChessboardCorners(grayR, (7, 6), cv.CALIB_CB_ACCURACY)

    t2 = time.time()
    print('Tiempo en encontrar las celdas del tablero de ajedrez: ', t2 - t1, 's')

    if retL == True:  # Si los encuentra calcula los rvecs y tvecs y dibuja los ejes

        t1 = time.time()

        corners2_L = cv.cornerSubPix(grayL, cornersL, (11, 11), (-1, -1), criteria)  # Mejora los corners encontrados

        t2 = time.time()
        print('Tiempo en mejorar las celdas del tablero de ajedrez: ', t2 - t1, 's')

        t1 = time.time()

        # Encuentra los vectores de rotacion y traslacion
        retLL, rvecsL, tvecsL = cv.solvePnP(objpL, corners2_L, newcameramtxL, distL, flags=cv.SOLVEPNP_IPPE)

        t2 = time.time()
        print('Tiempo en determinar la pose del tablero de ajedrez: ', t2 - t1, 's')

        t1 = time.time()

        # Proyecta los puntos 3D en el plano imagen
        imgptsL, jacL = cv.projectPoints(axis, rvecsL, tvecsL, newcameramtxL, distL)
        dstL = draw(dstL, corners2_L, imgptsL)  # Llama a la funcion draw() para dibujar los ejes

        t2 = time.time()
        print('Tiempo en proyectar eje cartesiano en el tablero de ajedrez: ', t2 - t1, 's')

    cv.imshow('dstL', dstL)

    if retR == True:  # Si los encuentra calcula los rvecs y tvecs y dibuja los ejes

        corners2_R = cv.cornerSubPix(grayR, cornersR, (11, 11), (-1, -1), criteria)  # Mejora los corners encontrados

        # Encuentra los vectores de rotacion y traslacion
        retRR, rvecsR, tvecsR = cv.solvePnP(objpR, corners2_R, newcameramtxR, distR, flags=cv.SOLVEPNP_IPPE)

        # Proyecta los puntos 3D en el plano imagen
        imgptsR, jacR = cv.projectPoints(axis, rvecsR, tvecsR, newcameramtxR, distR)

        dstR = draw(dstR, corners2_R, imgptsR)  # Llama a la funcion draw() para dibujar los ejes

    cv.imshow('dstR', dstR)

```

```
    t_final = time.time()

    print('Tiempo total: ', t_final-t_inicio, 's')

    if cv.waitKey(1) & 0xff == ord('s'):
        cv.imwrite('estimacion_pose' + str(photo) + '.png', concatenate)
        print(photo)
        print('-------------------------')
        print('rvecsL: ', 180 / np.pi * rvecsL.transpose())
        print('tvecsL: ', 3.1 * tvecsL.transpose())
        print('-------------------------')
        print('rvecsR: ', 180 / np.pi * rvecsR.transpose())
        print('tvecsR: ', 3.1 * tvecsR.transpose())

        photo = photo + 1

    elif cv.waitKey(1) & 0xff == ord('q'):
        break

imagesL.release()
imagesR.release()

cv.destroyAllWindows()
```

A.1.3. Algoritmo de creación de mapas de disparidad y profundidad y reconocimiento de objetos

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
from pixellib.instance import instance_segmentation

ply_header = '''ply
format ascii 1.0
element vertex %(vert_num)d
property float x
property float y
property float z
property uchar red
property uchar green
property uchar blue
end_header
'''

class_names = ["BG", "person", "bicycle", "car", "motorcycle", "airplane",
                       "bus", "train", "truck", "boat", "traffic light", "fire
    hydrant", "stop sign",
                       "parking meter", "bench", "bird", "cat", "dog", "horse", "
    sheep", "cow", "elephant", "bear",
                       "zebra",
                       "giraffe", "backpack", "umbrella", "handbag", "tie", "
    suitcase", "frisbee", "skis",
                       "snowboard",
                       "sports ball", "kite", "baseball bat", "baseball glove", "
    skateboard", "surfboard",
                       "tennis racket",
                       "bottle", "wine glass", "cup", "fork", "knife", "spoon", "
    bowl", "banana", "apple", "sandwich",
```

```
                           "orange",
                           "broccoli", "carrot", "hot dog", "pizza", "donut", "cake", "chair", "couch", "potted plant",
                           "bed",
                           "dining table", "toilet", "tv", "laptop", "mouse", "remote", "keyboard", "cell phone",
                           "microwave",
                           "oven", "toaster", "sink", "refrigerator", "book", "clock", "vase", "scissors", "teddy bear",
                           "hair dryer",
                           "toothbrush"]

def imagSegmentation(imagL, imagR):

    new_imagL = cv2.cvtColor(imagL, cv2.COLOR_RGB2BGR)
    new_imagR = cv2.cvtColor(imagR, cv2.COLOR_RGB2BGR)

    # Detecta los objetos
    segmaskL, new_imagL = instance_video.segmentFrame(new_imagL, show_bboxes=True)
    segmaskR, new_imagR = instance_video.segmentFrame(new_imagR, show_bboxes=True)

    # Cargo la clase de objeto y su valor de confianza
    ids = segmaskL['class_ids']
    score = segmaskL['scores']

    # Creo las mascaras
    maskL = segmaskL['masks'] * 1.0
    maskR = segmaskR['masks'] * 1.0

    # Genero una mascara para cada imagen que englobe todos los objetos detectados por cada camara
    mascaraL = np.zeros((455, 640, 3))
    mascaraR = np.zeros((455, 640, 3))

    for i in range(maskL.shape[2]):
        mascaraL[maskL[:, :, i] == 1] = 1

        # Si su valor de confianza es mayor a 0.7 muestro en pantalla la clase y su valor
        if score[i] >= 0.7:
            number = ids[i]
            print(class_names[number])
            print(score[i])

    for i in range(maskR.shape[2]):
        mascaraR[maskR[:, :, i] == 1] = 1

    imgL[mascaraL == 0] = 0
    imgR[mascaraR == 0] = 0

    return imgL, imgR, mascaraL, mascaraR, maskL, maskR, ids, score


def nothing(x):
    pass


# Escribe la nube de puntos en un archivo .ply
def write_ply(fn, verts, colors):
    l = len(verts)
```

```
82
83      if len(verts) % 3 == 1:
84
85          verts = verts[:len(verts) - 1]
86          colors = colors[:len(colors) - 1]
87
88      elif len(verts) % 3 == 2:
89
90          verts = verts[:len(verts) - 2]
91          colors = colors[:len(colors) - 2]
92
93      verts = verts.reshape(-1, 3)
94      colors = colors.reshape(-1, 3)
95
96      verts = np.hstack([verts, colors])
97
98      with open(fn, 'wb') as f:
99          f.write((ply_header % dict(vert_num=len(verts))).encode('utf-8'))
100         np.savetxt(f, verts, fmt='%f %f %f %d %d %d ')
101
102
103 criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 60, 0.000001)
104
105 size = (480, 640)
106
107 disparity = []
108 a = []
109 b = []
110
111 # Encendido de camaras
112 print('Iniciando camaras...')
113
114 cv2.namedWindow('disp',  cv2.WINDOW_FREERATIO)
115 cv2.resizeWindow('disp', (10000, 100))
116
117 imagesL = cv2.VideoCapture(0)
118 imagesR = cv2.VideoCapture(2)
119
120 print(' Usar filtro en el mapa de disparidad? No: 0, Si: 1')
121 filtro = int(input())
122
123 print(' Usar segmentacion de imagen? No: 0, Si: 1')
124 segment = int(input())
125
126 print(' Aplicar mascaras? No: 0, Si: 1')
127 segment2 = int(input())
128
129 if bool(segment) is True:
130     # Iniciamos el modelo de segmentacion de imagen "mask_rcnn_coco.h5"
131     instance_video = instance_segmentation(infer_speed="rapid")
132     instance_video.load_model("mask_rcnn_coco.h5")
133
134 print('Creando mapa de disparidad...')
135
136 cv2.namedWindow('disp', cv2.WINDOW_AUTOSIZE)
137 cv2.resizeWindow('disp', (640, 480))
138
139 cv2.createTrackbar('numDisparities', 'disp', 7, 30, nothing)
140 cv2.createTrackbar('blockSize', 'disp', 9, 100, nothing)
141 cv2.createTrackbar('disp12MaxDiff', 'disp', 20, 100, nothing)
```

```
142 cv2.createTrackbar('minDisparity', 'disp', 100, 200, nothing)
143 cv2.createTrackbar('mode', 'disp', 3, 4, nothing)
144 cv2.createTrackbar('lambda', 'disp', 8000, 80000, nothing)
145 cv2.createTrackbar('sigma', 'disp', 30, 50, nothing)  # luego se divide entre 10
146
147 stereo = cv2.StereoSGBM_create()
148
149 disparity = []
150 disparityFiltered = []
151
152 while True:
153
154     # Leo los frames de ambas camaras
155     trueL, imagL = imagesL.read()
156     trueR, imagR = imagesR.read()
157     imgL = imagL
158     imgR = imagR
159
160     # Elimino las distorsiones de ambas camaras
161     imgL = cv2.remap(imagL, mapL1, mapL2, interpolation=cv2.INTER_NEAREST, borderMode
    =cv2.BORDER_REPLICATE)
162     imgR = cv2.remap(imagR, mapR1, mapR2, interpolation=cv2.INTER_NEAREST, borderMode
    =cv2.BORDER_REPLICATE)
163
164     # Recortamos la imagen
165     imgL = imgL[0:455, 0:640]
166     imgR = imgR[0:455, 0:640]
167
168     # Cambio a escala de grises
169     dstL = cv2.cvtColor(imgL, cv2.COLOR_BGR2GRAY)
170     dstR = cv2.cvtColor(imgR, cv2.COLOR_BGR2GRAY)
171
172     # Parametros del mapa de disparidades
173     numDisparities = cv2.getTrackbarPos('numDisparities', 'disp') * 16  # Numero
    maximo de disparidades (maxDis - minDis)
174     blockSize = cv2.getTrackbarPos('blockSize', 'disp') * 2 + 1  # Tama o del bloque
     que matchea puntos
175     disp12MaxDiff = cv2.getTrackbarPos('disp12MaxDiff', 'disp')
176     minDisparity = cv2.getTrackbarPos('minDisparity', 'disp')  * -1 # Valor minimo de
     disparidad al matchear puntos
177     mode = cv2.getTrackbarPos('mode', 'disp')
178     lmbda = cv2.getTrackbarPos('lambda', 'disp')
179     sigma = cv2.getTrackbarPos('sigma', 'disp') / 10
180
181     stereo.setNumDisparities(numDisparities)
182     stereo.setBlockSize(blockSize)
183     stereo.setDisp12MaxDiff(disp12MaxDiff)
184     stereo.setMinDisparity(minDisparity)
185     stereo.setMode(mode)
186
187     # Creo el mapa de disparidades sin filtrar con la camara izquierda como
    referencia
188     disparity = stereo.compute(dstL, dstR).astype(np.float32) / 16.0
189
190     if bool(segment) is True:
191         # Segmentacion de imagen
192         _, _, mascaraL, mascaraR, maskL, maskR, ids, score = imagSegmentation(imgL,
    imgR)
193
194     # Disparidad con filtro WLS
```

```
    if bool(filtro) is True:

        # Creo el matcher para la camara derecha
        matcher = cv2.ximgproc.createRightMatcher(stereo)
        # Creo el filtro WLS (Weighted Least Squares)
        wls_filter = cv2.ximgproc.createDisparityWLSFilter(matcher_left=stereo)
        # Creo el mapa de disparidades con la camara derecha como referencia
        disparityR = matcher.compute(dstR, dstL).astype(np.float32) / 16.0

        # Parametros del filtro WLS
        wls_filter.setLambda(lmbda)
        wls_filter.setSigmaColor(sigma)

        # Creo el mapa de disparidades filtrado
        disparityFiltered = wls_filter.filter(disparity, dstL, disparity_map_right=
    disparityR)

        # Normalizado el mapa de disparidades filtrado
        disparityFiltered = cv2.normalize(disparityFiltered, disparityFiltered, beta
    =255, alpha=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
        disparityFiltered = (256 - disparityFiltered)
        disparityR = cv2.normalize(disparityR, disparityR, beta=255, alpha=1,
    norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
        disparityR = (256 - disparityR)

        # Proyecto el mapa de disparidades en un mapa de puntos 3D
        factorcalibracion = 2.478
        pointsFiltered = - cv2.reprojectImageTo3D(disparityFiltered, Q) *
    factorcalibracion
        pointsFiltered = pointsFiltered[:, 12:540]

        # Los convierto a formato 8U
        disparityFiltered = np.uint8(disparityFiltered)
        disparityR = np.uint8(disparityR)

        # Muestro el mapa de disparidades filtrado
        if bool(segment2) is True:
            disparityFiltered[mascaraL[:,:,1] == 0] = 0

        # Muestro el mapa de disparidades filtrado
        cv2.imshow('disp', disparityFiltered)

    # Normalizado el mapa de disparidades
    disparity = cv2.normalize(disparity, disparity, beta=255, alpha=1, norm_type=cv2.
    NORM_MINMAX, dtype=cv2.CV_8U)
    disparity = (256 - disparity)

    # Convierto los valores maximos y minimos (producto del ruido, baja calidad de la
     imagen, zonas sin textura, etc.)
    # al valor de la mediana del total de la imagen
    threshold1 = 250
    threshold2 = 5
    disparitymedian = disparity[threshold1 > disparity]
    disparitymedian = disparitymedian[disparitymedian > threshold2]
    m = np.median(disparitymedian)
    disparity[disparity > threshold1] = m
    disparity[disparity < threshold2] = m

    factorcalibracion1 = 3.421504915
    factorcalibracion2 = 2.750382425
```

```
249     depth = factorcalibracion1 * 8.7 * 586 / disparity
250     depth = depth[:, 12:540]
251     cv2.imshow('depth', depth)
252     # Proyecto el mapa de disparidades en un mapa de puntos 3D
253     disparity = np.uint8(disparity)
254     points = - cv2.reprojectImageTo3D(disparity, Q) * factorcalibracion2
255     points = points[:, 12:540]
256 
257     # Muestro los resultados
258     #disparity = cv2.applyColorMap(disparity, cv2.COLORMAP_OCEAN)
259 
260     if bool(segment2) is True:
261         disparity[mascaraL[:,:,1] == 0] = 0
262         disparityR[mascaraR[:,:,1] == 0] = 0
263 
264     cv2.imshow('left', disparity)
265 
266     if bool(filtro) is True:
267         cv2.imshow('right', disparityR)
268 
269     cv2.setMouseCallback("disp", on_EVENT_LBUTTONDOWN)
270     cv2.setMouseCallback("combined", on_EVENT_LBUTTONDOWN)
271     cv2.setMouseCallback("left", on_EVENT_LBUTTONDOWN)
272 
273     if cv2.waitKey(1) & 0xff == ord('q'):
274         break
275 
276 colors = cv2.cvtColor(imgL, cv2.COLOR_RGB2BGR)
277 colors = colors[:, 12:540]
278 
279 disparity = disparity[:, 12:540]
280 mask = disparity > disparity.min()
281 out_points = points[mask]
282 out_colors = colors[mask]
283 
284 if bool(filtro) is True:
285     disparityFiltered = disparityFiltered[:, 12:540]
286     maskFiltered = disparityFiltered > disparityFiltered.min()
287     out_pointsFiltered = pointsFiltered[maskFiltered]
288     out_colorsFiltered = colors[maskFiltered]
289 
290 out = 'out.ply'
291 write_ply(out, out_points, out_colors)
292 
293 if bool(filtro) is True:
294     outFiltered = 'outFiltered.ply'
295     write_ply(outFiltered, out_pointsFiltered, out_colorsFiltered)
296 
297 # Muestro la imagen izquierda y derecha, el mapa de disparidades sin filtrar y el
        filtrado
298 imgL = cv2.cvtColor(imgL, cv2.COLOR_BGR2RGB)
299 imgR = cv2.cvtColor(imgR, cv2.COLOR_BGR2RGB)
300 
301 plt.subplot(221), plt.imshow(imgL), plt.title('dstL')
302 plt.subplot(222), plt.imshow(imgR), plt.title('dstR')
303 plt.subplot(223), plt.imshow(disparity, cmap=plt.get_cmap('plasma')), plt.title('
        disparity non filtered')
304 
305 if bool(filtro) is True:
306     plt.subplot(224), plt.imshow(disparityFiltered, cmap=plt.get_cmap('plasma')), plt
```

```
    .title('disparity filtered')

plt.show()

imagesL.release()
imagesR.release()

cv2.destroyAllWindows()
```

A.1.4. Algoritmo de ejemplo

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

disparity = []
a = []
b = []

# Cargo las imagenes
imgL = cv2.imread('PIA24670-Ingenuity_Helicopter_in_3D-left.png')
imgR = cv2.imread('PIA24670-Ingenuity_Helicopter_in_3D-right.png')

# Preparo las imagenes
imgL = imgL[110:1866, 110:1530]
imgR = imgR[110:1866, 110:1530]

dstL = cv2.cvtColor(imgL, cv2.COLOR_BGR2GRAY)
dstR = cv2.cvtColor(imgR, cv2.COLOR_BGR2GRAY)

dstL = cv2.normalize(dstL, dstL, beta=0, alpha=255,
                                  norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8U)
dstR = cv2.normalize(dstR, dstR, beta=0, alpha=255,
                                  norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8U)

# Creo el mapa de disparidades
print('Creando mapa de disparidades...')

stereo = cv2.StereoSGBM_create(minDisparity=-65,
                               numDisparities=15*16,
                               blockSize=3,
                               P1=1*8*11*11,
                               P2=1*32*11*11,
                               disp12MaxDiff=20,
                               preFilterCap=0,
                               uniquenessRatio=5,
                               speckleWindowSize=50,
                               speckleRange=1,
                               mode=3)

disparity = stereo.compute(dstL, dstR)

# Aplico el filtro WLS
if True:
    matcher = cv2.ximgproc.createRightMatcher(stereo)
    wls_filter = cv2.ximgproc.createDisparityWLSFilter(matcher_left=stereo)
    disparityR = matcher.compute(dstR, dstL)
```

```
49
50      lmbda = 80000
51      sigma = 3.0
52      wls_filter.setLambda(lmbda)
53      wls_filter.setSigmaColor(sigma)
54
55      disparity = np.int16(disparity)
56      disparityR = np.int16(disparityR)
57
58      disparityFiltered = wls_filter.filter(disparity, dstL, disparity_map_right=
        disparityR)
59      disparityFiltered = cv2.normalize(disparityFiltered, disparityFiltered, beta=0,
        alpha=255,
60                                        norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)
61
62  disparity = np.int16(disparity)
63  disparity = cv2.normalize(disparity, disparity, beta=0, alpha=255, norm_type=cv2.
        NORM_MINMAX, dtype=cv2.CV_32F)
64  disparityR = cv2.normalize(disparityR, disparityR, beta=0, alpha=255, norm_type=cv2.
        NORM_MINMAX, dtype=cv2.CV_32F)
65
66  factorcalibracion = 7
67  depth = factorcalibracion * 24.4 * 11 / disparityFiltered
68  depth = depth[440:1756, 175:1100]
69  print(np.max(depth))
70  disparity = np.uint8(disparity)
71  disparityR = np.uint8(disparityR)
72  disparityFiltered = np.uint8(disparityFiltered)
73
74  # Muestro los resultados obtenidos
75  plt.subplot(121), plt.imshow(cv2.cvtColor(imgL, cv2.COLOR_BGR2RGB)), plt.title('
        Imagen izquierda'), plt.xlabel('P xeles'), plt.ylabel('P xeles')
76  plt.subplot(122), plt.imshow(cv2.cvtColor(imgR, cv2.COLOR_BGR2RGB)), plt.title('
        Imagen derecha'), plt.xlabel('P xeles'), plt.ylabel('P xeles')
77  plt.subplot(122), plt.imshow(disparity, cmap=plt.get_cmap('plasma')), plt.title('Mapa
         de disparidad sin filtrar'), plt.xlabel('P xeles'), plt.ylabel('P xeles')
78  plt.subplot(121), plt.imshow(disparityFiltered, cmap=plt.get_cmap('plasma')), plt.
        title('Mapa de disparidad filtrado'), plt.xlabel('P xeles'), plt.ylabel('
        P xeles')
79  plt.colorbar(label="Disparidad", orientation="horizontal")
80  plt.subplot(122), plt.imshow(depth, cmap=plt.get_cmap('jet_r')), plt.title('Mapa de
        profundidad'), plt.xlabel('P xeles'), plt.ylabel('P xeles')
81  plt.colorbar(label="Profundidad (m)", orientation="vertical")
82
83
84  plt.show()
```