

Proyecto Fin de Grado  
Ingeniería de las Tecnologías de Telecomunicación

Estudio de Técnicas de Odometría Visual  
Monocular

Autor: Rafael V. Domínguez Pérez

Tutor: Manuel Vargas Villanueva

Dpto. Ingeniería de Sistemas y Automática  
Escuela Técnica Superior de Ingeniería  
Universidad de Sevilla

Sevilla, 2019





Proyecto Fin de Carrera  
Ingeniería de las Tecnologías de Telecomunicación

# **Estudio de Técnicas de Odometría Visual Monocular**

Autor:

Rafael V. Domínguez Pérez

Tutor:

Manuel Vargas Villanueva

Profesor titular

Dpto. de Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2019



Proyecto Fin de Grado: Estudio de Técnicas de Odometría Visual Monocular

Autor: Rafael V. Domínguez Pérez

Tutor: Manuel Vargas Villanueva

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

*A mi familia*

*A mis profesores*



# Agradecimientos

---

*“Emancipate yourselves from mental slavery.  
None but ourselves can free our minds.”  
Robert Nesta Marley, 'Redemption Song'.*

Este trabajo supone el final de una etapa, un camino que no hubiese sido capaz de recorrer sólo.

Por encima de todo, gracias a mi familia: mi padre, mi madre y mis dos hermanas. Sin ellos no hubiera sido posible ni tan siquiera imaginar nada de esto. Los primeros en felicitarme en los buenos momentos y los primeros en consolarme en los malos. Para agradecerles, más que unas líneas, necesitaría un anexo.

A mis amigos, los de siempre y los que aparecieron por el camino. Por los momentos de estudio, los de desconexión y por todo lo que hemos compartido.

A los profesores que he tenido durante estos años. De un modo u otro todos han influido para alcanzar este objetivo, en especial Manuel, cuya asignatura me impulsó a elegir un trabajo como este, por el que también me ha guiado.

*Rafael V. Domínguez Pérez  
Sevilla, 2019*



# Resumen

---

En nuestros días, incorporar la navegación autónoma a vehículos y robots en general, se ha convertido en una temática en pleno auge con potenciales aplicaciones como facilitar operaciones de salvamento y rescate en entornos hostiles, la domótica y hasta la exploración espacial.

Para lograr esta preciada capacidad cobra especial importancia el estudio de la estimación de movimiento y reconstrucción del entorno, cuya investigación viene desarrollándose desde hace ya algunos años.

Existen numerosas formas de enfocar estas materias, entre las que encontramos la odometría visual de una sola cámara, o monocular: el propósito del presente estudio será elaborar un exhaustivo análisis sobre este tema.

Para ello, presentaremos las líneas de investigación dominantes en este campo, clasificando las técnicas más populares del estado del arte. Acto seguido abordaremos la exploración más detallada de una de estas técnicas, tanto teórica como experimentalmente, dando paso a efectuar diversas pruebas en un entorno especialmente dedicado a ello, pudiendo así examinar cómo se implementa en la práctica.



# Abstract

---

Nowadays, incorporating autonomous navigation to vehicles and robots in general has become a booming theme with potential applications such as facilitating rescue operations in hostile environments, home automation and even space exploration.

In order to achieve this precious capacity, the study of the estimation of movement and reconstruction of the environment, whose research has been developing for some years now, is particularly important.

There are several ways to approach these subjects, among which we find monocular visual odometry: the purpose of this study will be to elaborate an exhaustive analysis on this subject.

To do this, we will present the dominant research lines in this field, classifying the most popular techniques of the state of the art. Then we will discuss the more detailed exploration of one of these techniques, both theoretically and experimentally, giving way to carry out various tests in an environment specially dedicated to it, thus being able to examine how it is implemented in practice.

# Índice

---

<b>Agradecimientos</b>	<b>ix</b>
<b>Resumen</b>	<b>xi</b>
<b>Abstract</b>	<b>xiii</b>
<b>Índice</b>	<b>xiv</b>
<b>Índice de Figuras</b>	<b>xvi</b>
<b>1 Introducción</b>	<b>1</b>
<b>2 Técnicas de Estimación del Movimiento 3D</b>	<b>4</b>
2.1. <i>Consideraciones Previas</i>	4
2.2. <i>SLAM y VO. Definición y Comparativa.</i>	5
2.2.1 SLAM	5
2.2.2 Visual Odometry	6
2.2.1 Comparativa	6
2.3. <i>SLAM y VO. Perfil Técnico.</i>	8
2.3.1 SLAM	9
2.3.2 Visual Odometry	11
2.3.3 Elementos comunes: detección y correspondencia de características	15
2.4. <i>Clasificación de las técnicas más destacadas</i>	19
2.4.1 Métodos Densos vs. Métodos Dispersos	19
2.4.2 Métodos Directos vs. Métodos Indirectos	20
2.4.3 Métodos Dispersos Indirectos	21
2.4.4 Métodos Densos Directos	22
2.4.5 Métodos Dispersos Directos	22
<b>3 Semi-Direct Visual Odometry</b>	<b>25</b>
3.1. <i>Presentación</i>	25
3.2. <i>Funcionamiento</i>	26
3.2.1 Motion Estimation	27
3.2.2 Mapping	30
3.2.3 Comentarios Adicionales	31
<b>4 Implementación: Robot Operating System</b>	<b>33</b>
4.1. <i>Introducción a ROS</i>	33
4.2. <i>Estructura y Funcionamiento</i>	35
4.3. <i>Instalación del Software: ROS y SVO</i>	37
<b>5 Pruebas y Resultados</b>	<b>42</b>
5.1. <i>Ejecución de la simulación</i>	42
5.2. <i>Análisis del resultado</i>	45
5.3. <i>Iniciación a pruebas adicionales</i>	60
<b>6 Conclusiones y Líneas Futuras</b>	<b>62</b>
<b>Referencias</b>	<b>64</b>



# Índice de Figuras

---

Figura 1: Robots Sophia, Kiva y Curiosity.	2
Figura 2: Consistencia de mapas VO y SLAM.	7
Figura 3: Esquema de funcionamiento en SLAM.	8
Figura 4: Esquema de funcionamiento en Visual Odometry.	8
Figura 5: Transformaciones entre fotogramas en Visual Odometry.	12
Figura 6: Reconstrucción de la trayectoria en Visual Odometry.	13
Figura 7: Geometría Epipolar.	14
Figura 8: Detección de Características.	15
Figura 9: Comparativa entre Detectores de características.	16
Figuras 10-15: Algoritmo RANSAC.	17
Figura 16: Mapa disperso.	19
Figura 17: Mapa denso.	20
Figura 18: Gráfica resumen de las clasificaciones explicadas.	23
Figura 19: Cronograma relativo a las técnicas introducidas.	24
Figura 20: 6 grados de libertad (6 DoF). Traslación y Rotación.	25
Figura 21: Procesamiento: media y desviación estándar.	25
Figura 22: Diagrama de flujo de SVO.	26
Figura 23: Alineación de imágenes basada en modelo disperso.	28
Figura 24: Alineación de características.	29
Figura 25: Refinamiento de posición y estructura.	29
Figura 26: Estimación de profundidad.	31
Figura 27: Logotipos de Ubuntu y ROS.	33
Figura 28: Establecimiento de la comunicación en ROS.	36
Figura 29: Contenido de un paquete.	37
Figura 30: Configuración de parámetros de la máquina virtual.	38

Figura 31: Directorios en el espacio de trabajo tras la instalación.	40
Figura 32: Directorio “rpg_svo”.	41
Figura 33: Ejecución del comando roscore en nuestro sistema.	42
Figura 34: Apertura de rviz con la configuración apropiada para la simulación.	43
Figura 35: Puesta en marcha de la simulación. Captura de un instante aleatorio	44
Figura 36: Tamaño de las cuadrículas en las que se dividen las imágenes.	45
Figura 37: Valor fijado para la cantidad de keyframes a utilizar.	45
Figura 38: Imagen vista desde el rviz.	45
Figura 39: Vista de la grabación en un instante comprendido entre el primer y segundo fotograma.	46
Figura 40: Instante donde se selecciona el segundo frame.	47
Figura 41: Uso de la herramienta “Measure” del rviz.	47
Figura 42: Reconstrucción de mapa y trayectoria en un instante elegido al azar.	48
Figura 43: Zoom en los ejes de la ruta.	49
Figura 44: Grafo de computación ROS en nuestro sistema.	49
Figura 45: Comando rosnode info para el nodo SVO.	50
Figura 46: Resultado de rosnode list: Listado de nodos activos.	51
Figura 47: Rostopic info en el nodo “/camera/image_raw”.	51
Figura 48: Comparativa visual del contenido de los topics de las imágenes.	52
Figura 49: Comparativa de mensajes publicados en los topics de las imágenes.	53
Figura 50: Búsqueda de una transición que pueda denotar una característica.	53
Figura 51: Datos de los keyframes.	54
Figura 52: Diferenciación entre puntos del mapa y de la ruta.	55
Figura 53: Datos del mensaje para el topic “/svo/points”.	55
Figura 54: Formato de visualization_msg/Marker.msg	56
Figura 55: Verificación de la publicación de puntos de mapa y trayectoria.	57
Figura 56: Ejecución de rostopic echo /svo/keyframes/action.	57
Figura 57: Diferentes ejemplos de eliminación de puntos del mapa.	58
Figura 58: Relación entre los marcos de coordenadas.	59

Figura 59: Visualización de los ejes para la cámara y referencia mundial.	59
Figura 60: Contenido del fichero “.launch”.	60

# 1. INTRODUCCIÓN

---

**E**l desarrollo, potencial e importancia que la Robótica y la Inteligencia Artificial tienen en la sociedad viene creciendo de manera exponencial desde hace ya algunas décadas. Desde su aparición, esta disciplina ha ido superando cualquier expectativa acerca de su impacto y alcance: mientras que los primeros robots estaban enfocados casi exclusivamente a tareas de automatización industrial, en nuestros días son pieza clave en sectores tan diversos como los transportes, la agricultura, la minería, la sanidad, el automovilístico, aplicaciones militares, los emergentes y cada vez más utilizados drones, e incluso, la exploración espacial.

No obstante, aún queda mucho camino por recorrer y parece no haber barreras en el horizonte para la robótica y sus objetivos, siendo cada vez más común que robots lleven a cabo tareas que hasta hace no mucho parecían reservadas únicamente para humanos. Y todo ello sin que, de momento, Isaac Asimov tuviera que preocuparse por el futuro de la humanidad:

- *“Un robot no puede dañar a un ser humano ni, por inacción, permitir que un ser humano sufra daño.”*
- *“Un robot debe cumplir las órdenes de los seres humanos, excepto si dichas órdenes entran en conflicto con la Primera Ley.”*
- *“Un robot debe proteger su propia existencia en la medida en que ello no entre en conflicto con la Primera o la Segunda Ley.”*

Las tres leyes de la Robótica de Isaac Asimov, enunciadas por primera vez en su relato “Círculo Vicioso”, marzo 1942.

Asimov, más allá de su gran influencia en la literatura de ciencia ficción por sus pioneras novelas sobre robots, también es una obligada referencia al tratar estos temas por algunas reflexiones que lanzaría en forma de predicciones. Podemos utilizar algunas muy llamativas datadas en 1964 para hacernos una idea de la evolución que ha experimentado esta disciplina en poco de más cincuenta años:

- *“Los robots no serán ni comunes ni muy buenos en 2014, pero existirán”,* refiriéndose a robots andróides.

Pese a que hemos incorporado en nuestro día a día autómatas de ámbito doméstico sin capacidad de aprendizaje pero capaces de, por ejemplo, limpiar o preparar la comida, es cierto que los robots humanoides no se puede decir que sean algo demasiado común. Sin embargo, el desarrollo de la Inteligencia Artificial y, más concretamente el Machine Learning o aprendizaje automático, ha permitido que su existencia no sea una simple idea utópica más propia de la ciencia ficción. Este es el caso de Sophia, que pasará a la historia como el primer robot en conseguir la ciudadanía de un país (Arabia Saudí, en 2017). Desarrollada por la compañía Hanson Robotics, Sophia emplea tecnologías de reconocimiento de voz y facial que le hacen ser capaz de imitar gestos y expresiones

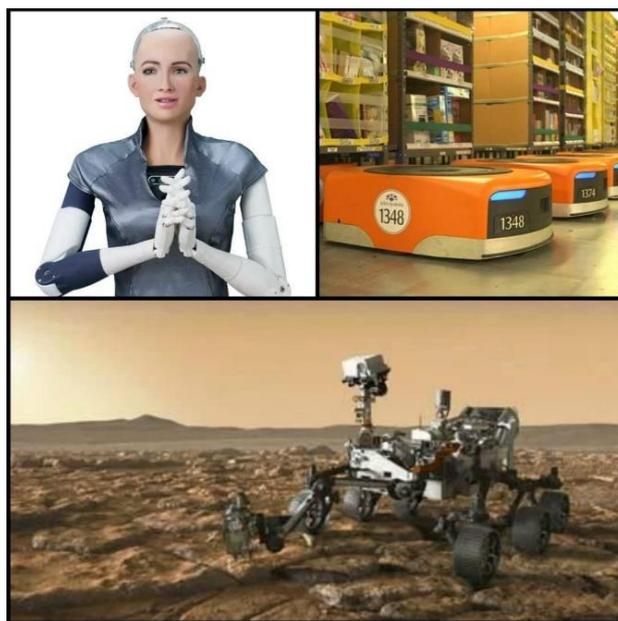
faciales, y lo que es más importante, combina técnicas de Machine Learning y Big Data permitiéndole procesar miles de datos y aprender algo sin que le haya sido explícitamente programado, con lo que se espera que pueda analizar sus conversaciones y mejorarlas. Sin duda, se trata de un gran hito para la robótica.

- *“Habrá pocos trabajos rutinarios que no puedan ser realizados por alguna máquina en vez de por humanos”.*

En este aspecto podemos afirmar que Asimov acertó de pleno. Y es que la automatización de la industria es ya una realidad, siendo habitual la presencia de elementos robóticos en las cadenas de montaje, y que importantes y consolidadas empresas como Apple o Google decidan invertir cada vez más presupuestos en I+D en esta materia. Los populares robots Kiva de Amazon son una buena prueba de ello, ya que dotados de algoritmos que le permiten estimar el tamaño de las cargas, regular la velocidad de sus trayectorias, etcétera, agilizan gran cantidad labores.

- *“Para 2014 habrán aterrizado en Marte naves no tripuladas, aunque ya se estará trabajando en enviar una expedición con humanos”*

Con esta afirmación, el escritor y bioquímico estadounidense de origen ruso tampoco estaba equivocado. El interés por explorar el planeta rojo no es nuevo, como así lo demuestran la gran cantidad misiones soviéticas y norteamericanas que tuvieron lugar a lo largo de la carrera espacial que ambas potencias protagonizaron durante la Guerra Fría, con programas espaciales como Marsnik, Mariner o Vikings en las décadas de los 60 y 70. Con el tiempo, el número de misiones de este tipo fueron multiplicándose y perfeccionándose hasta llegar a los actuales rovers, vehículos espaciales motorizados no tripulados. El robot Curiosity de la NASA, que aterrizó en Marte en agosto de 2012, sea quizás el máximo exponente de estos rovers capaces de recorrer la superficie del planeta vecino. Además, Japón, India o la Agencia Espacial Europea se han ido sumando a esta ambiciosa carrera, en cuyo horizonte asoma incluso una misión de colonización: el proyecto Mars One.



**Figura 1.** Arriba, Sophia y Kiva Amazon. Abajo, el rover Curiosity.

Este último apartado nos servirá como puerta de enlace al tema central que este trabajo aborda. Y es que la comunidad científica, desde hace ya algún tiempo, tiene su punto de mira puesto en ir dotando de mayor autonomía a los robots hasta el punto de que los rovers, y otros más cotidianos como drones y vehículos del sector automovilístico, puedan ejecutar trayectorias de manera completamente autónoma sin que nadie los dirija remotamente. Para ello se hace necesario recoger información del entorno en que se encuentran mediante algún sistema de sensores y, dado que el presente estudio versará sobre herramientas de visión artificial monoculares, asumiremos que dicho sistema viene dado por una sola cámara de abordo, no varias (estéreo).

Técnicas como SLAM (simultaneous localization and mapping) y VO (visual odometry) se ocupan de esta problemática con el fin de que el robot sea capaz de desarrollar un mapa para ubicarse en un entorno a priori desconocido y desplazarse a través de él evitando los obstáculos presentes. Además, cobran mayor importancia si tenemos en cuenta que aplicándolas, en principio puede bastar con sensores visuales para lograr la navegación local, efectuando el procesamiento a partir de las imágenes adquiridas sin necesidad de asociación con mecanismos de posicionamiento global.

Así pues, los objetivos que perseguimos con los sucesivos apartados de este trabajo son, bajo un prisma teórico, analizar y comparar SLAM y VO, profundizar en los fundamentos técnicos de ambas disciplinas, y clasificar y presentar sus principales técnicas; por otra parte, bajo un enfoque experimental nos encargaremos de introducir el sistema ROS (*Robot Operating System*) y emplearlo para estudiar y hacer pruebas con el método elegido. Por último, expondremos los resultados y conclusiones a las que lleguemos tras la realización de esta investigación

# 2. TÉCNICAS DE ESTIMACIÓN DEL MOVIMIENTO 3D

---

## 2.1- Consideraciones previas.

El primer paso para conseguir incorporar la capacidad de realizar vuelos autónomos en drones -denominados a menudo VANTS<sup>1</sup>- parece lógico que sea la construcción de un mapa del entorno en el que se encuentra, para poder estimar su posición dentro del mismo. Las técnicas monoculares de visión artificial que lo permiten son el objeto de estudio de este trabajo.

### - ¿Por qué emplear la Visión Artificial?

El coste y la velocidad de procesamiento son los motivos por los cuales, más allá de la cámara a bordo de la nave, no tendremos en cuenta ningún otro tipo de sensores basados en láser, ondas de radio, ultrasonidos y demás sensores de medición de profundidad o distancia.

Tampoco atenderemos, pues, a señales GPS, la solución más extendida hasta el momento para lograr vuelos autónomos. Sin embargo, estos sistemas no obtienen en recintos interiores los grandes resultados que dan en exteriores, y si además el ambiente es dinámico y tiene numerosos elementos, el dron no queda exento de chocar con alguno que se interponga en su trayectoria.

Así pues, todas estas cuestiones motivan que en este estudio nos centremos en sistemas exclusivamente de visión artificial para lograr el objetivo de estimar el movimiento de la nave. Y tal y como avanzamos en la introducción, este apartado estará dedicado a repasar las técnicas más importantes basadas en SLAM y VO.

### - ¿Por qué un Sistema Monocular?

Analizar las diferencias entre los enfoques monocular y estéreo es algo que ya ha sido ampliamente investigado en numerosos trabajos, valgan como ejemplo [1][2]. A partir de estudios de esta naturaleza, se ha podido concluir que los algoritmos para el caso monocular son algo más complejos. Esto es debido a que no es posible determinar la profundidad a partir de una imagen tomada por una única cámara, con lo que se deberá calcular mediante el análisis de los fotogramas adquiridos en cascada. Sin embargo, un aspecto clave que juega a favor del sistema monocular es que el hardware es más simple, reduciendo el tamaño y el coste de su implementación.

Por otro lado, tal y como exponen Scaramuzza y Fraundorfer [3], un sistema estéreo puede degenerar en el caso monocular si la distancia *baseline* es decir, la distancia que separa ambas cámaras, es considerablemente menor que la distancia a la escena. En

---

<sup>1</sup> Vehículos aéreos no tripulados. También es bastante común designarlos como RPAS o UAVS.

tal situación, por tanto, la opción estéreo resultaría inadecuada. Por estas razones hemos decidido focalizar este trabajo en soluciones monoculares.

### **2.2.- SLAM y VO. Definición y comparativa.**

En primer lugar, describamos de qué se ocupan cada una de estas dos disciplinas para justo a continuación hacer hincapié en los aspectos que las diferencian y hacen únicas. De esta manera conseguiremos tener una visión global de ambas antes de ahondar en sus apartados técnicos.

#### **2.2.1.- SLAM.**

Comencemos por SLAM, que según Durrant-Whyte y Bailey[4], data de 1986, fecha en la que tiene lugar en San Francisco la conferencia del IEEE sobre Robótica y Automática durante la cual se pone de manifiesto la necesidad de abordar la problemática conceptual y computacional que suponía introducir en un robot la habilidad de mapear.

Y es que justamente ése es el objetivo de SLAM, que como sus propias siglas indican (en inglés, *Simultaneous Localisation And Mapping*), se ocupa de la construcción de un mapa consistente y global del entorno en el que opera el robot y, simultáneamente, obtener una estimación de su localización<sup>2</sup> dentro del mismo. Dicho mapa, que se puede generar a través de puntos de interés tales como marcas u obstáculos, es un aspecto clave de SLAM, ya que permite reducir el error cometido en la estimación de la ubicación de la nave. Esto es posible gracias a otro concepto clave, los cierres de bucle ("*loop closures*"): cuando se vuelve a visitar una zona ya conocida, el robot puede restablecer su error de localización y reducir la deriva<sup>3</sup> tanto en el mapa como en la trayectoria[5]. Este proceso de optimización en el que se refinan las posiciones y la estructura final se conoce como *Bundle Adjustment*, y más concretamente para SLAM, *Global Bundle Adjustment*.

Pese a que SLAM ha demostrado funcionar con robots como rovers y drones, los algoritmos pueden incurrir a menudo en fallos en escenarios adversos como entornos muy dinámicos o movimientos bruscos del robot, por lo que no estamos ante un campo de investigación cerrado.

---

<sup>2</sup> En numerosos textos, se denota como "*pose*" (en inglés) al tándem posición-orientación que describe la localización del robot. Por simplicidad, nos referiremos a ella como ubicación o simplemente posición.

<sup>3</sup> A menudo los autores hacen referencia al concepto de deriva para referirse a la desviación entre la trayectoria real y la estimada.

### 2.2.2.- Visual Odometry:

Por su parte, la odometría visual es el proceso en el que se estima incrementalmente, es decir, posición tras posición, y en tiempo real la ubicación del robot. Para ello, se analizan las imágenes tomadas por la cámara a bordo en busca de los cambios que el movimiento va generando en dichas imágenes. La optimización se realiza teniendo en cuenta únicamente las  $n$  últimas posiciones, proceso conocido como *Windowed Bundle Adjustment*.

Esta disciplina nace en la década de los 80 y alcanza su punto álgido con su implementación en los vehículos espaciales Opportunity y Spirit enviados a Marte en 2004.

Para funcionar adecuadamente, VO asume una serie de condiciones[3] que se presentan a continuación<sup>4</sup>:

- Existencia de iluminación suficiente en el entorno.
- Poca presencia de elementos móviles en la escena, a considerar prácticamente estática.
- Imágenes con suficiente textura para extraer el movimiento aparente.
- Suficiente solape/superposición entre fotogramas consecutivos.

VO presenta mejoras respecto a su predecesor *Wheel Odometry*, que en vez de emplear una cámara como sensor de entrada utiliza encoders<sup>5</sup> situados en las ruedas del vehículo. La más destacada es que las estimaciones de VO son más precisas gracias a que no se ven afectadas por deslizamientos de rueda en superficies poco regulares. Así pues, los algoritmos de VO presentan una deriva considerablemente más pequeña, en concreto por debajo del 0.5% de la longitud de la trayectoria[5],[6].

Cabe destacar también que, pese a que a veces se utilicen como sinónimos, VO no es lo mismo SFM (*Structure For Motion*) [3]. Esta técnica es más general y afronta el problema de la reconstrucción 3D del mapa y de las posiciones del robot a partir de un conjunto de imágenes desordenadas. Aplica también *Bundle Adjustment* para optimizar mapa y posiciones, pero a diferencia de VO (y SLAM) el procesamiento no se lleva a cabo en tiempo real, por lo que no es capaz de determinar la ubicación del robot durante la toma de imágenes.

### 2.2.3.- Comparativa:

Una vez presentadas ambas disciplinas es momento de enfatizar en lo que las hace diferentes.

La primera diferencia la encontramos en el proceso de optimización:

---

<sup>4</sup> Estos requerimientos son extensibles a SLAM, puesto que estamos hablando de condiciones mínimas para el procesado de las imágenes (concretamente, detección y correspondencia de características) que tiene lugar en ambas disciplinas.

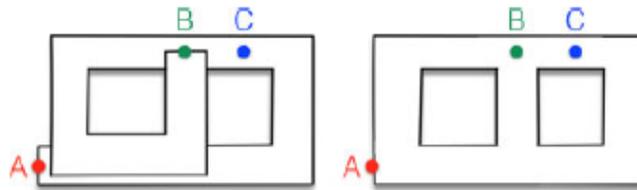
<sup>5</sup> Dispositivo electromecánico que hace las veces de interfaz entre plataformas rotatorias y controlador, dado que permite codificar el movimiento mecánico en un código digital. Por ello también recibe el nombre de codificador de eje.

Dado que VO se centra en la consistencia local de la trayectoria (recordemos que se recupera de manera incremental), este refinamiento se va realizando sólo sobre las últimas  $n$  posiciones. De ahí que reciba el nombre de *Windowed Bundle Adjustment*.

Por su parte, como SLAM no sólo busca la consistencia global de la trayectoria, sino también del mapa, es necesario detectar cuando se producen los cierres de bucles, momento en el que se realizará la optimización. Este proceso se conoce como *Global Bundle Adjustment*.

Pero, ¿qué queremos decir exactamente cuando hablamos de consistencia? Insistamos en este concepto para que no deje lugar a dudas, ya que en él radica la principal diferencia entre SLAM y VO. Para ello emplearemos la **Figura 2**, tomada del estudio de Cadena et al.[5].

Decimos que en VO la consistencia es local porque como la ruta del robot se reconstruye secuencialmente sin tener en cuenta cierres de bucle (**Figura 2-izquierda**), se interpreta que la nave avanza continuamente por áreas desconocidas, como si el entorno se tratase de un pasillo infinito. De esta manera, puntos que en realidad son cercanos (B y C) pueden resultar alejados con esta metodología.



**Figura 2:** A la izquierda, mapa visto por VO. A la derecha, mapa construido en SLAM. El punto A es el inicio del recorrido, siendo B el final del mismo. C es un punto intermedio.

Por el contrario, en SLAM (**Figura 2-derecha**) sí son tenidos en cuenta los cierres de bucle, de forma que pueda plasmarse en el mapa la estructura real de la escena. En este caso, los puntos B y C no quedarían distanciados. Por ello decimos que SLAM se centra en la consistencia global del mapa.

Un corolario que podemos extraer de las reflexiones vertidas acerca de las diferencias en el proceso de optimización es que VO puede verse como una etapa o bloque constructivo de SLAM. Concretamente, podemos entender que VO es lo que acontece en SLAM antes de un cierre de bucle.

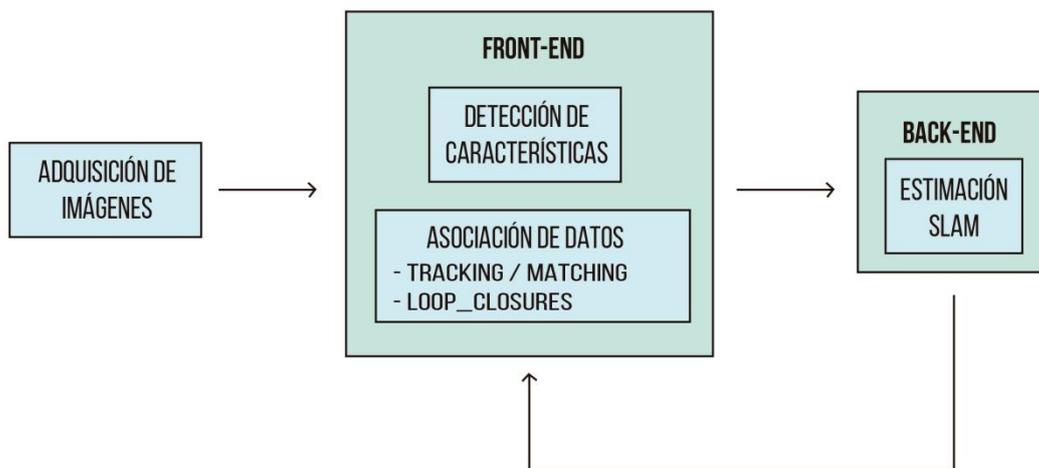
La otra gran diferencia entre estas dos técnicas la encontramos en el coste computacional: los algoritmos SLAM son más costosos y difíciles de implementar.

Por consiguiente, vistas las diferencias, elegir entre una y otra dependerá de donde pongamos el foco, si en el rendimiento o en la consistencia: SLAM da mayor importancia a ésta última, mientras que VO compensa su menor consistencia con un mayor rendimiento en tiempo real, dado que opera con una cantidad menor de fotogramas, haciéndolo computacionalmente más ligero.

### 2.3.- SLAM y VO. Perfil Técnico.

Dedicaremos este apartado a comentar las bases técnicas de estas dos disciplinas de manera general<sup>6</sup>. Más adelante detallaremos más para el método concreto que elijamos de entre los que presentemos en la clasificación del siguiente apartado.

Los diagramas de flujo de SLAM y VO, pese a poseer elementos comunes, difieren en algunos aspectos. Podemos apreciar el cariz que presentan uno y otro en las **Figuras 3 y 4**.



**Figura 3.** Esquema de funcionamiento en SLAM. Fuente: elaboración propia, basada en [5].



**Figura 4.** Esquema de funcionamiento en Visual Odometry. Fuente: Elaboración propia.

En ellas vemos que comparten algunos bloques o tareas del proceso, como la adquisición de imágenes, la extracción de características y la asociación o emparejamiento de los datos. Para no redundar en su explicación los detallaremos una vez que hayamos abordado los apartados que se ocupan de la estimación del movimiento en VO y SLAM, que difieren entre sí. Aprovechando la inercia del apartado anterior, comenzamos por esto último para seguir evidenciando las diferencias entre

---

<sup>6</sup> Ahondaremos algo más en el perfil técnico de VO que en el de SLAM. El motivo es que para la parte experimental de este trabajo se empleará una técnica de odometría visual, concretamente SVO.

estos sistemas. Además, comentaremos algunos matices de los esquemas al completo que nos faciliten su comprensión.

### 2.3.1.- SLAM

Como se aprecia en la **figura 3**, la estructura de un sistema SLAM se compone de dos bloques principales que siguen a la adquisición inicial de datos (imágenes). En el bloque Front-End se combinan la visión por computador y el procesamiento de señal con el objetivo de convertir los datos del sensor en parámetros susceptibles de estimación. A su vez, en el Back-End se emplean probabilidad y geometría para generar las estimaciones de posiciones y mapa a partir de los parámetros producidos en el Front-End.

Nótese que en el módulo de asociación de datos del bloque Front-End se distinguen dos etapas. A corto plazo se tiene como objetivo el emparejamiento de medidas del sensor que representan el mismo punto 3D en fotogramas consecutivos. A largo plazo se pretenderá asociar las nuevas mediciones con las tomadas anteriormente, detectando así los cierres de bucle. Este es el motivo de que exista una realimentación desde el bloque Back-End: reportar feedback que facilite dicha detección de cierres de bucle.

Abordemos ahora el bloque Back-End. Prácticamente la totalidad de los estudios realizados hasta la fecha han planteado la estimación SLAM como un problema probabilístico. Seguiremos el trabajo de Whyte y Bailey[4] para apoyarnos a partir de ahora en su formulación.

Definimos en primer lugar los siguientes parámetros para el instante k-ésimo:

- C<sub>k</sub>**: vector de estado del robot, que define su posición y orientación.
- T<sub>k</sub>**: vector de control. Se aplica al instante k-1 para llevar al robot al estado  $x_k$  en el instante k.
- m<sub>i</sub>**: vector que contiene la ubicación del i-ésimo punto de referencia, cuya ubicación real se considera invariante en el tiempo.
- f<sub>k</sub>**: Observación/medición tomada desde el robot en el instante k (puntos en la imagen).

De manera inmediata pueden definirse los siguientes conjuntos, que agrupan los parámetros anteriores de la siguiente forma:

- C<sub>0:k</sub>** = { C<sub>0</sub> , C<sub>1</sub> , ... , C<sub>k</sub> }. Historial de posiciones del dron.
- T<sub>0:k</sub>** = {T<sub>0</sub> , T<sub>1</sub> , ... , T<sub>k</sub> }. Historial de entradas de control.
- m** = {m<sub>0</sub>, m<sub>1</sub>, ..., m<sub>k</sub>}. Conjunto de puntos de referencia.
- f<sub>0:k</sub>** = { f<sub>0</sub> , f<sub>1</sub> , ... , f<sub>k</sub> }. Conjunto de observaciones.

Con esto, plantear SLAM de manera probabilística requiere resolver, para cada instante k, la función de distribución siguiente:

$$P( C_k, m \mid f_{0:k}, T_{0:k}, C_0 ) \tag{1}$$

que describe la densidad conjunta de las ubicaciones de los puntos de referencia y la posición de la nave en el instante k, conocidas las observaciones y entradas de control hasta dicho instante inclusive, además de la ubicación inicial del robot ( $C_0$ ).

Puesto que lo más conveniente sería una solución recursiva, se comienza calculando la distribución para el instante k-1, y siguiendo el vector de control  $T_k$  y la observación  $f_k$ , se estima el instante posterior k haciendo uso del Teorema de Bayes. Para ejecutar este cálculo resulta necesario definir un modelo para la observación (2) y otro modelo para el movimiento (3), que describa el efecto de las entradas de control para la transición entre dos posiciones. Se presentan ambos a continuación:

$$P(f_k | C_k, m) \quad (2)$$

$$P(C_k | C_{k-1}, T_k) \quad (3)$$

Nótese que en el modelo de movimiento (3) cada localización del robot  $C_k$  depende únicamente de la inmediatamente anterior  $C_{k-1}$  y de la transición  $T_k$ , y por consiguiente es independiente tanto de las observaciones como del mapa.

Una vez que tenemos así modelado el problema, puede implementarse un algoritmo recursivo-secuencial de dos pasos. El paso de predicción es el primero de ellos y se describe en la forma de la ecuación (4). El segundo paso, donde ya se hace uso de la medida actual, se denomina paso de corrección o de actualización, y viene dado por la ecuación (5).

(4)

$$P(C_k, m | f_{0:k-1}, T_{0:k}, C_0) = \int P(C_k | C_{k-1}, T_k) \cdot P(C_{k-1}, m | f_{0:k-1}, T_{0:k-1}, C_0) dC_{k-1} \quad (5)$$

$$P(C_k, m | f_{0:k}, T_{0:k}, C_0) = \frac{P(f_k | C_k, m) P(C_k, m | f_{0:k-1}, T_{0:k}, C_0)}{P(f_k | f_{0:k-1}, T_{0:k})}$$

Estas dos ecuaciones proporcionan un mecanismo recursivo con el que estimar la distribución (1) para mapa y localización del robot en el instante k. A modo de resumen, solucionar el problema probabilístico que SLAM plantea requiere encontrar una representación adecuada para los modelos de observación y movimiento de las ecuaciones (2) y (3), los cuales nos permitirán calcular las distribuciones (4) y (5).

Con el problema estructurado en esta forma, sin lugar a dudas el enfoque más repetido es abordar su solución haciendo uso del Filtro de Kalman Extendido (EKF), como así lo atestiguan la gran cantidad de destacados trabajos consultados [7],[8],[9]. Profundizar en la formulación de SLAM basado en EKF escapa a los límites de este estudio, pero la idea general es la que sigue: permitir que a partir de modelos no lineales como los de observación y movimiento –que con el Filtro de Kalman estándar no podrían trabajarse– se pueda obtener la media y la covarianza de la distribución (1), mediante dos etapas recursivas también denominadas predicción y actualización.

Por su parte, en el estudio de Cadena et al [5], la estimación SLAM del bloque Back-End de la **figura 3** se fundamenta en una estimación MAP (*maximum a posteriori*), que a diferencia del Filtro de Kalman no necesita una diferenciación explícita entre los modelos de observación y movimiento. En este enfoque, se tienen como parámetros la variable  $C$ , que contiene la trayectoria del robot (como un conjunto discreto de posiciones) y la ubicación de los puntos de referencia, y el conjunto de observaciones  $f$ . De esta forma, cada medida puede expresarse como en (6):

$$f_k = h_k(C_k) + \epsilon_k \quad (6)$$

donde  $\epsilon_k$  hace referencia a una medida de ruido aditivo gaussiano, y la función  $h_k(\cdot)$  se supone el modelo de observación conocido. En la estimación MAP se pretende estimar  $C$  cuando se alcanza el máximo ( $C^*$ ) de la función de probabilidad condicional  $P(C | f)$ , que siguiendo el Teorema de Bayes se describe como en (7):

$$C^* = \operatorname{argmax} P(C | f) = \operatorname{argmax} P(f | C) P(C) \quad (7)$$

### 2.3.2.- Visual Odometry

Explicemos ahora cómo trabaja VO, cuyo diagrama de flujo se mostró en la **Figura 4**. Comenzaremos por la formulación del problema que presenta la estimación de movimiento, para lo cual seguiremos la notación de Scaramuzza y Fraundorfer [3].

El conjunto de imágenes tomadas por la cámara a bordo del robot a medida que éste se desplaza por el entorno, al ocuparnos el caso monocular<sup>7</sup>, lo denotamos como:

$$I_{0:n} = \{ I_0, \dots, I_n \} \quad (8)$$

Las posiciones de cámara –y por tanto del vehículo– en dos instantes consecutivos  $k$  y  $k-1$  vienen relacionadas por la transformación:

$$T_{K,K-1} \equiv T_K = \begin{bmatrix} R_{K,K-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix} \quad (9)$$

y el conjunto que agrupa dichas transformaciones:

$$T_{0:n} = \{ T_1, \dots, T_n \} \quad (10)$$

Cabe destacar que el elemento  $R_{K,K-1}$  dentro de la transformación (9) es la matriz de rotación de dimensión 3x3, conformada por la combinación de las matrices  $R_x$ ,  $R_y$ ,  $R_z$

<sup>7</sup> Se trata de una simplificación del caso estéreo, donde habría dos adquisiciones (cámara derecha e izquierda) para cada instante  $k$ , y se plantearían, pues, dos conjuntos:

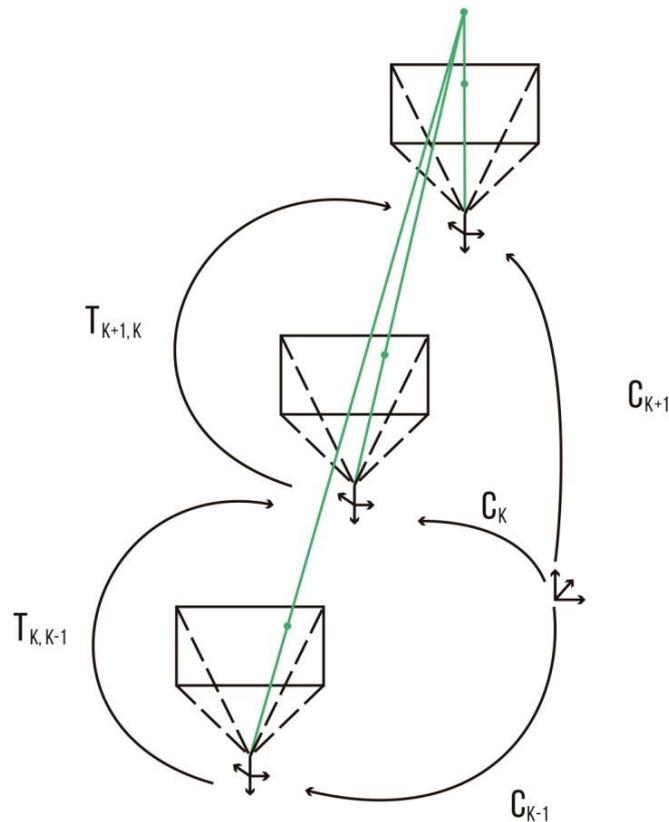
$$I_{L,0:n} = \{ I_{L,0}, \dots, I_{L,n} \} \quad I_{R,0:n} = \{ I_{R,0}, \dots, I_{R,n} \}$$

referentes a los ejes X,Y,Z, respectivamente. El elemento  $t_{k,k-1}$  es el vector de traslación, de dimensión 3x1.

Se define también el conjunto de posiciones, que contiene las transformaciones de la cámara con respecto al punto de partida (fotograma k=0):

$$C_{0:n} = \{ C_0, \dots, C_n \} \quad (11)$$

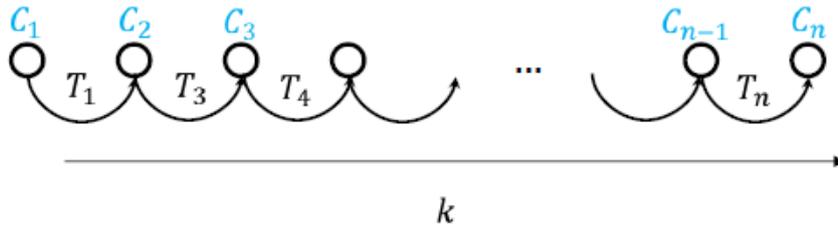
En la **figura 5** se ilustran a modo de resumen estos parámetros.



**Figura 5.** Transformaciones entre fotogramas consecutivos,  $T_K$ , y con respecto al fotograma inicial,  $C_K$ . Fuente: elaboración propia. Adaptación del esquema estéreo en [3].

El objetivo de la estimación de movimiento, por tanto, será calcular las transformaciones  $T_K$  a partir de las imágenes  $I_K$  e  $I_{K-1}$ , para posteriormente concatenarlas y reconstruir la ruta completa de la nave  $C_{0:n}$ , como se indica en la **figura 6**. Así, la posición actual  $C_n$  vendrá dada por la siguiente expresión:

$$C_n = C_{n-1} \cdot T_n \quad (12)$$



**Figura 6.** Reconstrucción de la trayectoria concatenando las sucesivas transformaciones [3].

Actuando de esta forma queda implícito que la reconstrucción se hace incrementalmente posición tras posición que, como insistimos en el apartado anterior, es uno de los aspectos diferenciales de VO.

La estimación de movimiento se puede afrontar desde 3 ópticas distintas, dependiendo de si las correspondencias de características se proporcionan en dos o tres dimensiones:

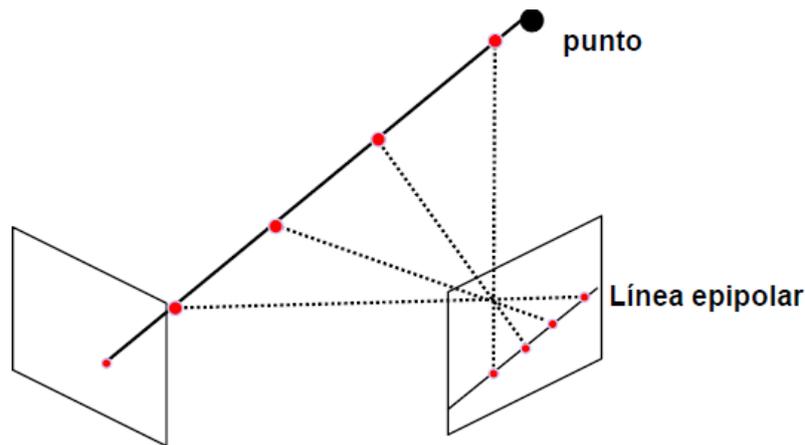
- 2D-2D:  $f_{k-1}$  y  $f_k$ , características en  $I_{k-1}$  e  $I_k$  respectivamente, se especifican ambas en 2D.
- 3D-2D:  $f_{k-1}$  es una característica en  $I_{k-1}$  especificada en 3D, y  $f_k$  es la proyección 2D de esa característica en  $I_k$ . En un sistema monocular, este caso necesita de al menos tres imágenes: la estructura 3D se triangula a partir de dos fotogramas adyacentes, para luego emparejarla con la característica 2D de una tercera imagen.
- 3D-3D: Se especifican ambas características  $f_{k-1}$  y  $f_k$  en coordenadas 3D. Como se necesitan triangular puntos 3D en cada instante temporal, este caso no está disponible para los sistemas monoculares y debe implementarse en estéreo.

Por simplicidad, nos ocuparemos del caso 2D-2D. Para ello se hace necesario introducir la “Matriz Esencial” o matriz “E”, que define las relaciones geométricas entre dos fotogramas adyacentes  $I_{k-1}$  e  $I_k$ . Se describe de la siguiente forma:

$$E_k \approx \hat{t}_k R_k \quad (13)$$

$$\text{Con } \hat{t}_k = \begin{vmatrix} 0 & -t_z & t_y \\ t_z & 0 & t_x \\ -t_y & t_x & 0 \end{vmatrix}, \text{ y } t_k = [t_x, t_y, t_z].$$

Así, la rotación y la traslación se pueden extraer de E, la cual se determina a partir de la correspondencia de características 2D-2D empleando la geometría epipolar. La geometría epipolar nos proporciona la línea (línea epipolar) de la segunda imagen en la que se encuentra un punto/característica de la primera imagen. La **figura 7** ejemplifica este concepto.



**Figura 7.** Objetivo de la geometría epipolar: Proyectar en la segunda imagen el conjunto de puntos reales que se proyecta en la primera imagen sobre el punto en el plano imagen [46].

Cuanto más correspondencias de características tengamos mejor, siendo necesarias al menos 5. El algoritmo de 5 puntos propuesto por Nister en 2003 es el más extendido, pese a que su implementación resulta complicada. Una solución más simple es el algoritmo de Longuet-Higgins para 8 o más puntos no coplanarios, que busca resolver la restricción fundamental que impone la geometría epipolar (14). Esta ecuación es la principal propiedad de la estimación de movimiento 2D-2D y se formula en la forma:

$$P_2 E P_1 = 0 \quad (14)$$

Donde  $P_2$  es la ubicación de una característica en  $I_k$ , y  $P_1$  la ubicación de la correspondiente característica en  $I_{k-1}$ . Ambos se expresan en coordenadas normalizadas en la forma  $P = [\tilde{u}, \tilde{v}, 1]$ , refiriéndose a la proyección sobre el plano de la imagen, y medidos en píxeles.

Una vez obtenida  $E$ , ya es posible extraer<sup>8</sup> de ella tanto la rotación como la traslación. Para ésta última deben calcularse las escalas relativas para las sucesivas transformaciones, por ejemplo, triangulando puntos 3D a partir de dos pares de fotografías consecutivos[10].

Como resumen de lo explicado hasta ahora, presentamos a continuación el algoritmo encargado de la estimación de movimiento en el caso 2D-2D [3]:

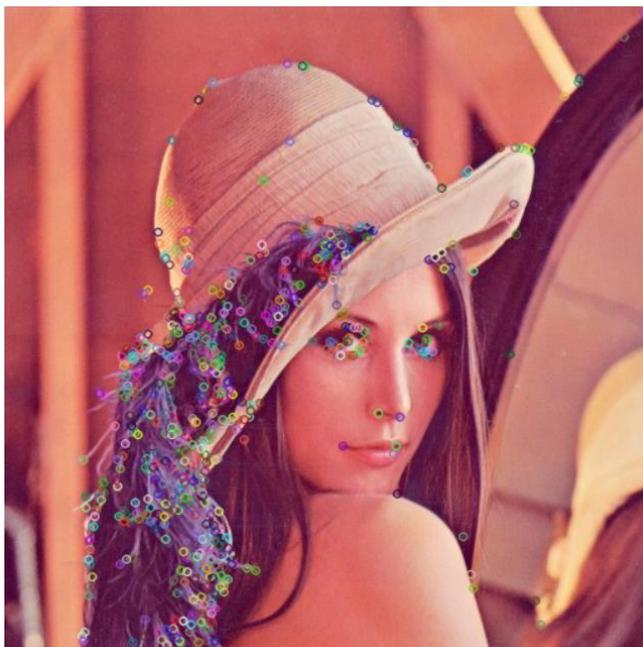
- I. Adquisición del nuevo fotograma  $I_k$ .
- II. Extracción de características y emparejamiento con el fotograma anterior  $I_{k-1}$ .
- III. Calcular la matriz esencial  $E$  para  $I_{k-1}$  y  $I_k$ .
- IV. Descomponer la matriz  $E$  en rotación  $R_k$  y traslación  $t_k$  para formar  $T_k$ .
- V. Calcular la escala relativa para  $t_k$ .
- VI. Concatenar la transformación para obtener la posición actual  $C_k = C_{k-1} \cdot T_k$
- VII. Volver al paso I.

<sup>8</sup> En el algoritmo de Longuet-Higgins se lleva a cabo mediante la técnica de descomposición en valores singulares o SVD.

### 2.3.3.- Elementos comunes: detección y correspondencia de características

Una vez explicada la etapa encargada de la estimación del movimiento, nos ocuparemos ahora de la extracción de características y la asociación de las mismas<sup>9</sup>. Como acabamos de ver en el algoritmo anterior, y tal y como se ilustró en las **figuras 3 y 4**, estas etapas del proceso siguen a la adquisición de la imagen y preceden a la estimación de movimiento.

En primer lugar, se buscan en la imagen características que potencialmente puedan coincidir con las detectadas en otras imágenes, entendiendo “característica” como un punto (o puntos) de la imagen que difiere de sus regiones vecinas en cuestiones de intensidad, color o textura (ver **figura 8**). Las características pueden ser elementos geométricos simples, como por ejemplo bordes, o bien las conseguidas mediante “Detectores de características”, algoritmos especialmente diseñados para la extracción de las mismas. Las principales propiedades de un buen detector de características son, entre otras, la precisión en la localización, la eficiencia computacional, la recurrencia – que se repitan características en las siguientes imágenes–, la robustez ante el ruido, y la invariancia tanto a cambios de iluminación (cambios fotométricos) como a cambios en la rotación o escala (cambios geométricos). Conseguir esto último, es decir, que el detector sea invariante al escalado, es una de las propiedades más deseadas, y consiste en aplicar el detector a un mismo fotograma con distintas versiones de escala.



**Figura 8.** Ejemplo de detección de características típicas.

Existen detectores de características puntuales, siendo los más famosos los detectores de esquinas, más concretamente los detectores de Harris[11], Shi-Tomasi[12] y FAST[13]; y, además, detectores más genéricos (“*blob detectors*”) como son SIFT[14] y SURF[15]. Se presenta en la **figura 9** un cuadro comparativo de ellos:

<sup>9</sup> Nos basaremos en las definiciones del estudio [10] para estas etapas del diagrama de flujo de VO.

	Corner Detector	Blob Detector	Rotation Invariant	Scale Invariant	Affine Invariant	Repeatability	Localization Accuracy	Robustness	Efficiency
Haris	x		x			+++	+++	++	++
Shi-Tomasi	x		x			+++	+++	++	++
FAST	x		x	x		++	++	++	++++
SIFT		x	x	x	x	+++	++	+++	+
SURF		x	x	x	x	+++	++	++	++

**Figura 9.** Comparación entre las propiedades de los detectores citados[10].

Posteriormente, a partir de la región en torno a cada característica detectada se genera un "Descriptor", el cual se intentará comparar con el resto de descriptores referentes a las demás características.

El descriptor más sencillo es la apariencia, entendida ésta como la intensidad de los píxeles en un área alrededor de la característica. Pero en determinadas situaciones, como ante cambios de orientación o de escala, las medidas que usualmente se emplean para comparar intensidades (la correlación cruzada normalizada y la suma de diferencias cuadráticas) fallan, al no ser métodos de medida invariantes ante estos cambios. En tales ocasiones no se comporta como un buen descriptor, estando, pues, limitado a que las imágenes sean tomadas entre posiciones cercanas para que no se den grandes cambios de este tipo.

SIFT y SURF actúan también como descriptores de características, siendo de los más populares, aunque presenten la desventaja de que su uso requiere pagar por su licencia al estar patentados. Otros descriptores cuyo uso está también muy extendido son BRIEF y ORB.

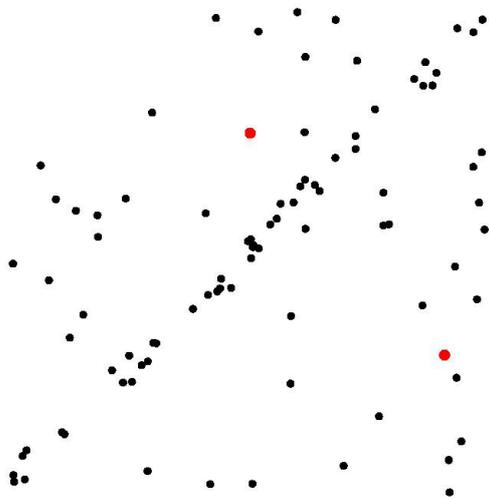
La asociación de datos con la que tratamos de establecer correspondencias entre las características puede afrontarse desde dos perspectivas. Aunque en parte de la literatura se haga referencia a ellas indistintamente, hay un matiz que caracteriza la manera en que se lleva a cabo la correspondencia, y que hace que diferenciamos entre *Matching* y *Tracking*:

- El Matching consiste en detectar características en todas las imágenes de forma independiente, y luego emparejarlas basándonos en alguna medida de similitud entre sus descriptores. Es más indicado para entornos a gran escala, en los que las imágenes se toman más alejadas.
- El Tracking consiste en extraer características en una imagen, para luego tratar de encontrarlas en las siguientes aplicando alguna técnica de búsqueda local, típicamente la correlación, combinada con la geometría epipolar. El tracking es más apropiado para cuando se toman las imágenes desde posiciones próximas entre ellas, por ejemplo, en entornos de pequeña escala.

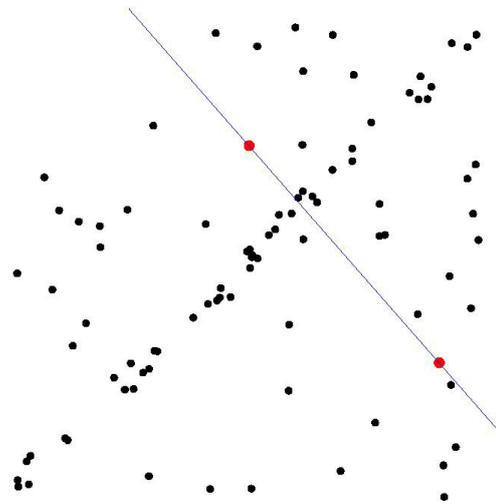
La asociación de características puede verse afectada por falsos emparejamientos, comúnmente denominados *outliers*, que pueden hacer que la estimación del movimiento

derive hacia soluciones erráticas. Para que la estimación de movimiento sea precisa y robusta es necesario eliminar estos outliers, siendo el algoritmo RANSAC (“*Random Sample Consensus*”) el método más utilizado en las técnicas VO para hacer frente a esta delicada tarea. Se trata de un proceso iterativo en el que a partir de muestras aleatorias de puntos (correspondencias) se pretende construir una hipótesis que modele la decisión de cuáles son valores atípicos para poder rechazarlos, y posteriormente verificar la validez de la hipótesis en el resto de muestras. Por tanto, al no ser RANSAC un método determinista, pues depende de la elección inicial aleatoria de las muestras, proporcionaría soluciones distintas para diferentes ejecuciones.

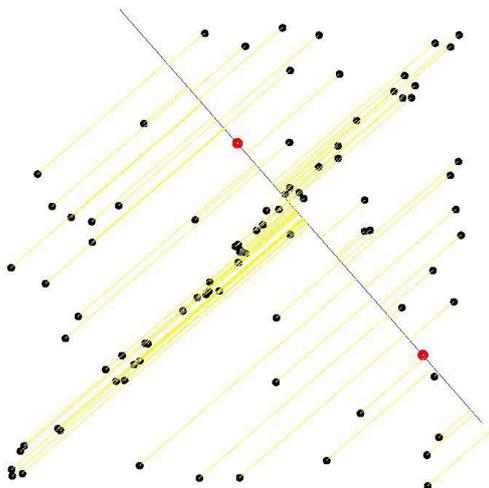
En las **figuras 10-15** [3] mostramos gráficamente cómo actúa RANSAC para una pareja de correspondencias tomada al azar (de manera genérica podría tomarse un conjunto de  $n$  muestras).



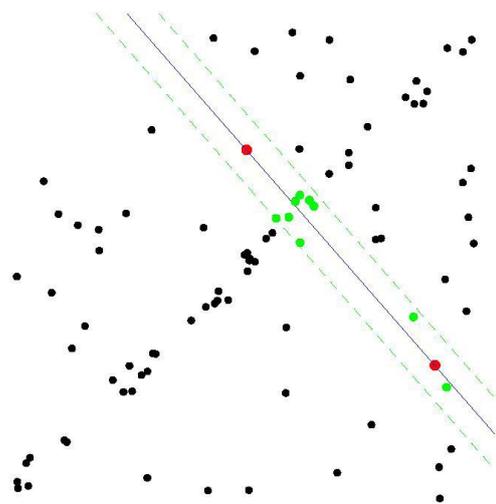
**Figura 10.** Elección de una muestra aleatoria de dos puntos.



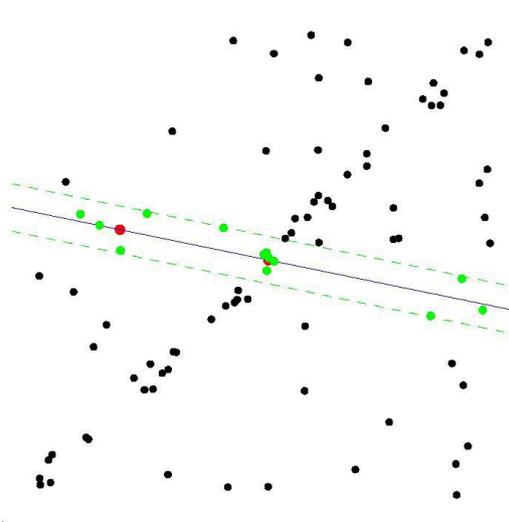
**Figura 11.** Establecer un modelo para la muestra tomada.



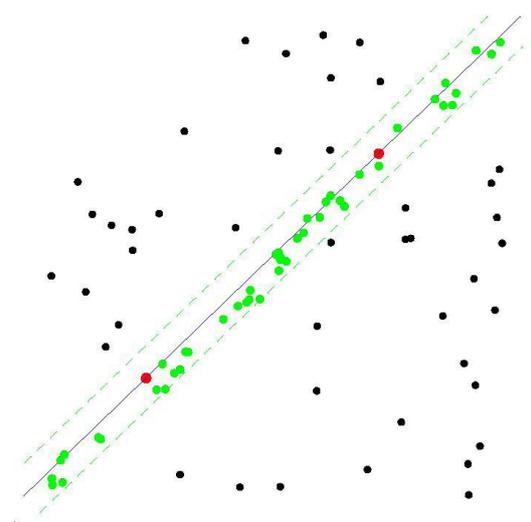
**Figura 12.** Calcular la función de error (distancia) para el resto de puntos.



**Figura 13.** Seleccionar los datos que cumplen con una distancia umbral (conjunto de *inliers*), es decir, los puntos que apoyen la hipótesis del modelo actual.



**Figura 14.** Repetir los pasos anteriores para otros pares de muestras.



**Figura 15.** Seleccionar como solución del problema el modelo con mayor número de *inliers*. Los puntos que no lo satisfacen serán valores atípicos (*outliers*), a descartar como correspondencias.

Pese a todos los esfuerzos que estamos viendo que la odometría visual realiza para que la reconstrucción de la ruta del robot sea robusta y precisa, ésta no será fiel a la real al cien por cien. En cada transformación  $T_{k,k-1}$  llevada a cabo para pasar desde la posición  $k-1$  hasta  $k$ , se introduce una incertidumbre que hace que la trayectoria real y la estimada discrepen. Y puesto que la posición actual  $C_k$  vemos que se forma concatenando las transformaciones, la incertidumbre total del robot dependerá de la incertidumbre de esas transformaciones individuales. Por ello es sumamente importante mantenerlas acotadas, ya que la deriva de la posición irá aumentando al concatenar las sucesivas transformaciones.

Por ello, el diagrama de flujo de VO finaliza con un refinamiento local de dicha deriva, al que ya hicimos referencia en el apartado anterior: *Windowed Bundle Adjustment*. Este ajuste reduce la deriva ya que utiliza correspondencias de características en más de dos cuadros de imagen, empleando una ventana de  $n$  fotogramas. El objetivo es minimizar el error de proyección.

El tamaño de la ventana se elige teniendo en cuenta el coste computacional que traerá consigo este ajuste. Así, un tamaño corto de ventana reduce la cantidad de parámetros a optimizar, lo que permite que este proceso se ejecute en tiempo real.

Con esto damos por finalizado este apartado sobre los perfiles técnicos y damos paso al siguiente, en el que elegiremos el método a escoger para el bloque experimental.

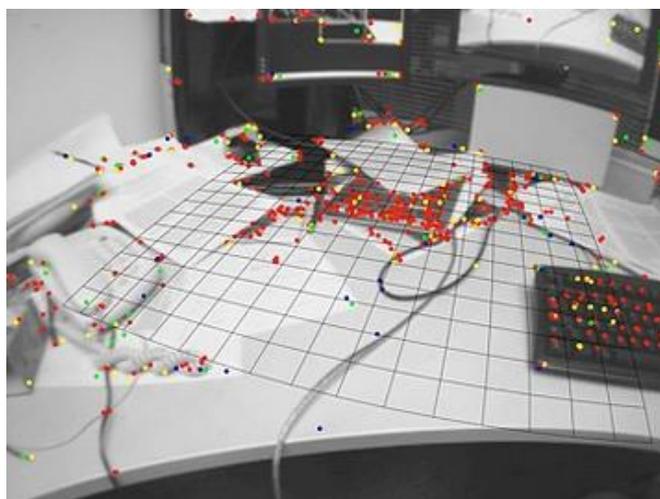
### **2.4.- Clasificación de las técnicas más destacadas.**

Este apartado lo dedicaremos a clasificar las principales técnicas SLAM y VO. Para ello atenderemos a los criterios que siguen, entre otros muchos autores, Engel et al.[16] en función del tratamiento dado a la información extraída de las imágenes. Existen dos clasificaciones posibles que dividen los métodos entre directos o indirectos, y entre densos o dispersos.

#### **2.4.1.- Métodos Densos vs. Métodos Dispersos.**

Esta clasificación atiende a la extensión de la región de imagen utilizada. Los métodos dispersos usan y reconstruyen sólo un pequeño conjunto de píxeles independientes en cada fotograma, generalmente las esquinas. Por su parte, los métodos densos emplean y reconstruyen la mayoría o todos los píxeles de los fotogramas.

Como consecuencia de la diferencia de píxeles y regiones utilizadas, los mapas que generan un método y otro difieren bastante. Los mapas generados por métodos dispersos se reducen a nubes de puntos que representan la escena de manera aproximada, y se usan básicamente para el seguimiento de la posición del robot. Mientras, los mapas generados a partir de métodos densos proporcionan una mayor cantidad de detalle que los mapas dispersos, pero a cambio requieren un procesamiento más potente al utilizar muchos más puntos. Valgan las **figuras 16 y 17** para ilustrar las diferencias entre un mapa disperso y otro denso, respectivamente.



**Figura 16:** Mapa disperso. Los puntos coloreados son los que se reconstruyen en el mapa [17].

Además de la extensión del área de imagen usada, otra diferencia fundamental radica en el empleo o no de un procedimiento geométrico previo. En los métodos densos se aprovecha la vecindad de los píxeles de las regiones de imagen utilizadas, añadiendo una geometría previa a la reconstrucción. De forma contraria, en los métodos dispersos

no se tiene esta posibilidad al no existir conectividad entre las posiciones de los píxeles utilizados.



**Figura 17:** Mapa denso. [5]

#### **2.4.2.- Métodos Directos vs. Métodos Indirectos**

Los métodos indirectos, que también se denominan basados en características, actúan en dos etapas:

En la primera se toman las mediciones en crudo (nivel de intensidad captado en cada píxel por el sensor) y se lleva a cabo un preprocesamiento que da lugar a unos valores intermedios o características. En la segunda etapa se toman esos valores intermedios extraídos en la primera fase, y se rastrean en el resto de imágenes tratando de emparejarlos para obtener correspondencias que faciliten la localización y reconstrucción de la trayectoria del robot.

No sería extraño que, llegados a este punto, el lector haya identificado esta metodología en las explicaciones del apartado anterior sobre los perfiles técnicos de SLAM y VO, ya que los métodos basados en características son los más extendidos y por ese motivo se particularizaron los diagramas de funcionamiento adaptados a ellos. Pero no es la única alternativa, y en el lado opuesto encontramos los métodos directos. Éstos omiten el preprocesamiento y la extracción de características, haciendo uso directamente de los valores de intensidad de los píxeles. Además, a diferencia de los indirectos –que sólo consideran la distancia a la ubicación de las características–, utilizan la magnitud y la dirección del gradiente de intensidad local para la optimización.

La ventaja que presentan los métodos directos es que obvian el costoso procedimiento de una extracción robusta de características en cada fotograma. Pero a cambio, generalmente pierden algo de precisión con respecto a los métodos indirectos o basados en características. Además, estos últimos proveen de una mayor tolerancia ante cambios de condiciones de iluminación, ya que no operan de forma directa con los valores de intensidad de los píxeles.

Otro aspecto diferencial entre ambos enfoques es que mientras los métodos indirectos optimizan un error geométrico –dado que los valores intermedios calculados en la primera fase son cantidades geométricas–, los métodos directos optimizan errores fotométricos<sup>10</sup> –al trabajar directamente con la luz recibida en los píxeles– [16].

Por último, es importante notar que las dos clasificaciones que hemos definido no son excluyentes, es decir, la distinción entre métodos densos y dispersos no es la misma que entre directos e indirectos. Esto hace posible que, combinándolas, existan cuatro posibles tipos de sistemas SLAM/VO, aunque la combinación densa-dispersa sea una formulación aún poco explorada en comparación con el resto, por lo cual no entraremos en detalle en ella.

### 2.4.3.- Métodos Dispersos Indirectos.

Éste es el enfoque más extendido. Estima la estructura tridimensional y la ruta a partir del emparejamiento de puntos clave, por lo que trabaja optimizando un error geométrico (método indirecto), sin explotar conectividades entre esos puntos (método disperso). Algunas de las principales técnicas SLAM se encuentran en este grupo:

- ***MonoSLAM***. Este sistema, desarrollado por Davidson et al. [18], fue la primera metodología SLAM de visión pura con una sola cámara aplicada exitosamente en robótica. Lograba un rendimiento con poca deriva y en tiempo real, algo inaccesible hasta el momento para enfoques del tipo *Structure For Motion* (SFM). Uno de los pilares básicos de esta técnica es un mapeo probabilístico basado en características, que representa en todo instante la estimación actual del estado de la cámara y de las características de interés, así como la incertidumbre de esas estimaciones. El otro principio fundamental es el modelado de movimiento, que tras la puesta en marcha del sistema actualiza el vector de estado (que define las estimaciones de posición) en dos etapas alternas: predicción, para el intervalo ciego entre capturas, y actualización, después de las mediciones y haber logrado las características.
- ***PTAM***. Por las siglas de “*Parallel Tracking and Mapping*”. Esta técnica, ideada por G. Klein y D. Murray [19], propone dividir el seguimiento de la ruta y el mapeado en dos ramas separadas que se procesan paralelamente: una se encarga del seguimiento robusto del movimiento, empleando para ello *keyframes* o fotogramas clave, mientras que la otra genera un detallado mapa 3D de puntos de características, típicamente esquinas extraídas con FAST a partir de los fotogramas adquiridos. Esto permite emplear técnicas de optimización computacionalmente costosas, las cuales no suelen ligarse a la actuación en tiempo real.

---

<sup>10</sup> En caso de que se utilizaran cámaras de profundidad RGB-D, un método directo también podría optimizar un error geométrico.

- ORB-SLAM. Esta técnica, propuesta por Montiel et al. [20], es capaz de operar en tiempo real en diversos escenarios, de interior y de exterior tanto extensos como reducidos. El sistema ORB-SLAM incorpora 3 bloques de funcionamiento que actúan paralelamente: tracking, mapeo local y cierres de bucle. El tracking se encarga de la localización con cada nuevo fotograma adquirido y decide cuando insertar un nuevo *keyframe* o fotograma clave. El mapeo local procesa los nuevos fotogramas clave –encargándose también de eliminar los que sean redundantes–, y realiza una optimización local (*local bundle adjustment*) para reconstruir de manera óptima del entorno de la posición de la cámara. Finalmente, se busca cerrar bucles en cada nuevo *keyframe*, para alinear y fusionar puntos duplicados, obteniendo además información sobre la deriva acumulada en el bucle.

#### 2.4.4.- Métodos Densos Directos.

Esta formulación pretende optimizar errores fotométricos con el fin de estimar una estructura densa o semidensa. Catalogamos aquí otras técnicas SLAM también muy populares, como son DTAM y LSD-SLAM:

- DTAM. Desarrollado por Newcombe et al. [21], DTAM fue el primer sistema generado para el seguimiento y la reconstrucción de la ruta de una cámara en tiempo real que no se basa en la extracción de características, sino que trabaja directamente con los píxeles con un paradigma denso, bajo el cual hace uso de todos los datos en la imagen. Su nombre procede de las siglas “*Dense Tracking And Mapping*” y demuestra cómo un modelo denso permite un alto rendimiento en el seguimiento en movimientos rápidos.
- LSD-SLAM. “*Large-Scale Direct Monocular SLAM*”, por J.Engel, T. Schöps y D. Cremers [22]. Se trata de un algoritmo SLAM monocular directo que, a diferencia del resto de métodos directos hasta su fecha, permite construir un mapa del entorno consistente y de gran escala (por ejemplo, útil en exteriores) Una formulación consciente de la desviación de la escala permite que esta técnica opere ante desafiantes secuencias de fotogramas en las que existan grandes variaciones en la escala de la escena. Junto con una estimación precisa de la posición basada en la alineación directa de las imágenes, el escenario 3D se reconstruye en tiempo real como una representación gráfica de ciertos fotogramas clave.

#### 2.4.5.- Métodos Dispersos Directos.

Esta propuesta trabaja directamente con el error fotométrico de los niveles de intensidad de las imágenes y sin incorporar geometría previa. Destaca aquí una novedosa y reciente técnica de odometría visual, *Direct Sparse Odometry*, concretamente del año 2016.

- DSO. Desarrollado por Engel et al. [16], este método combina un modelo probabilístico totalmente directo (es decir, minimizando el error fotométrico) junto

con una optimización conjunta de todos los parámetros del modelo. Esta optimización se efectúa mediante una ventana que se desplaza manteniendo los fotogramas más recientes, de modo que a medida que se desliza la ventana se obvian las posiciones de cámara antiguas que salen del campo de visión. De esta manera se logra mejorar la consistencia (local).

Además, esta técnica integra una calibración fotométrica de la cámara que incluye aspectos como corrección gamma o tiempos de exposición conocidos, ganando en precisión y robustez.

A lo largo de este apartado hemos visto los pros y los contras que presentan los diferentes enfoques, y dado que este trabajo tiene también una vertiente experimental que abordaremos más adelante, cabría preguntarse cuál de estas técnicas sería la mejor para efectuar pruebas en el entorno ROS. Es, por tanto, momento de introducir la técnica finalmente elegida para ello –que no es ninguna de las anteriores– y justificar dicha decisión. Se trata de SVO o “Semi Direct Visual Odometry”, presentada en 2014 por Forster et al. [23]. Esta propuesta semidirecta aúna los puntos fuertes de los métodos basados en características (asociación de correspondencias, robustez ante condiciones cambiantes, etc.) con la velocidad y precisión de los métodos directos. Esquematisamos en la **figura 18** la clasificación de las técnicas aquí descritas, haciendo hincapié en la condición de “método híbrido” que tiene SVO.

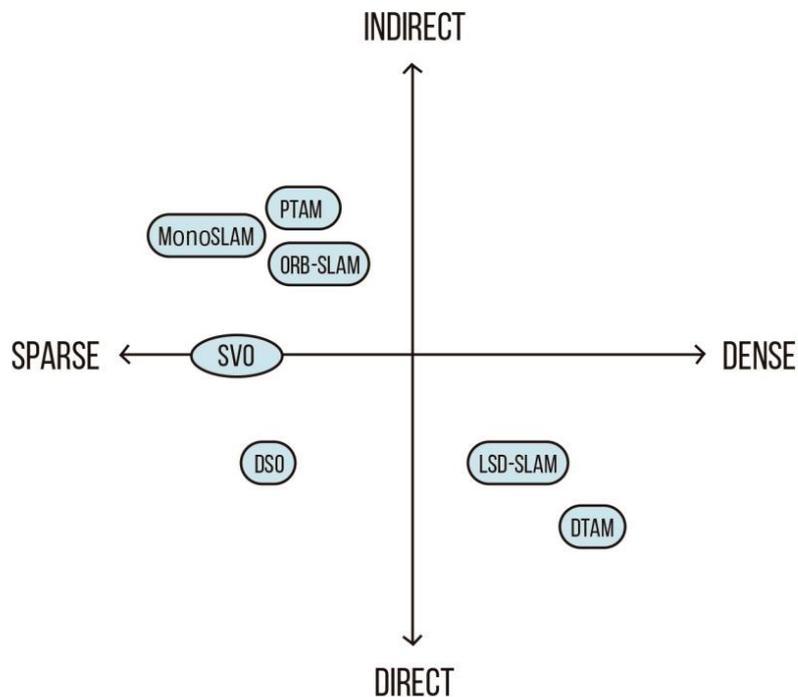
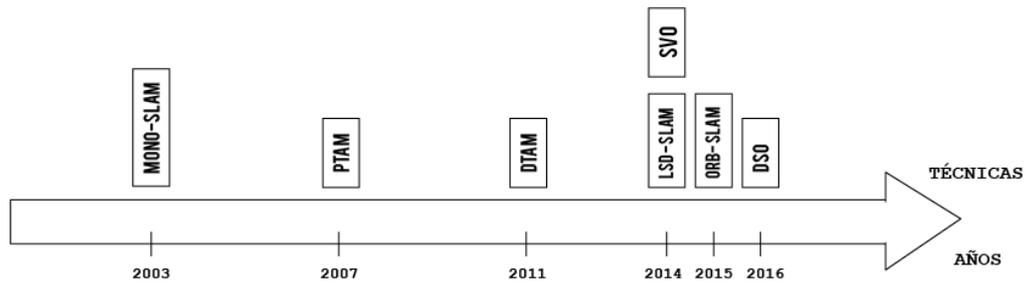


Figura 18. Gráfica resumen de las clasificaciones explicadas. Elaboración propia.

Resulta también interesante comparar en una línea temporal la fecha de desarrollo de estas técnicas. En la **figura 19** ilustramos esto:



**Figura 19:** Cronograma relativo a las técnicas introducidas. Elaboración Propia.

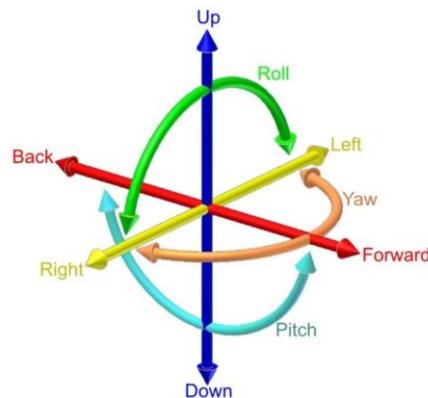
El motivo principal de elegir SVO es que, como sus propios autores indican, es una técnica más rápida y precisa que el resto del estado del arte hasta la fecha de su desarrollo. Esto, unido a la peculiaridad de que se encuentre a medio camino entre métodos directos e indirectos, hace que se trate de un enfoque más que atractivo para su estudio. Nos ocupamos de su análisis a lo largo del siguiente capítulo.

# 3. SEMI-DIRECT VISUAL ODOMETRY

## 3.1- Presentación.

El objetivo de este apartado es presentar la técnica que simularemos dentro del entorno ROS. Como adelantamos al final del apartado anterior, se trata de *Semi Direct Visual Odometry* o SVO, originaria del año 2014 y desarrollada por Christian Forster, Matia Pizzoli y Davide Scaramuzza, miembros del “Grupo de Percepción y Robótica” de la Universidad de Zúrich. El *paper* con el que la dieron a conocer[23] será la fuente de información fundamental en este capítulo.

Lo más interesante de SVO es que emplea un paradigma semidirecto para estimar el movimiento de 6 grados de libertad (ver **figura 20**) de la cámara: utiliza métodos directos para encontrar y triangular los píxeles caracterizados por gradientes de imagen altos, pero usa métodos basados en características para la optimización de estructura y movimiento.



**Figura 20:** 6 grados de libertad (6 DoF). Traslación: adelante/atrás, arriba/abajo, izquierda/derecha. Rotación: cabeceo (pitch), guiñada (yaw) y alabeo (roll). Fuente: Wikipedia.

SVO destaca por su versatilidad, que le permite funcionar en diferentes modelos de cámara, así como por su eficiencia, ya que requiere pocos recursos computacionales en comparación con la mayoría de los algoritmos existentes. En la siguiente figura se presenta una comparativa del tiempo de procesamiento requerido por algunas de las técnicas anteriormente citadas. En ella puede verse como SVO presenta mejoras en cuanto a velocidad.

	Mean	St.D.
SVO Mono	2.53	0.42
ORB Mono SLAM (No loop closure)	29.81	5.67
LSD Mono SLAM (No loop closure)	23.23	5.87

**Figura 21:** Media y desviación estándar del tiempo de procesamiento medido en milisegundos, empleando un portátil con procesador Intel Core i7. Fuente: [24].

Gracias a esa flexibilidad y eficiencia se ha podido implementar en aplicaciones muy diversas entre las que destaca la navegación autónoma en drones, pero también se ha ejecutado con éxito en realidad virtual, escaneo 3D, automoción y demás aplicaciones comerciales.

Cabe destacar también que en el año 2017 los mismos desarrolladores presentaron bajo el nombre de “SVO 2.0” una extensión de la implementación original, con la que se incorporaban novedades como la combinación con IMUs, configuraciones para sistemas estéreo y multicámara, o permitir el uso de cámaras con campo de visión muy grande, como las de ojo de pez o *fisheye*.

### 3.2- Funcionamiento.

Una vez presentada, veamos cómo funciona SVO. El algoritmo usa dos bloques o hilos paralelos, uno para estimar el movimiento y el otro para mapear el entorno que va explorándose. Esta separación permite llevar a cabo la correspondencia (tracking) de manera rápida y constante en un bloque, mientras que en el otro se construye el mapa sin atender restricciones de ejecución en tiempo real. En la **figura 22** se muestra el esquema de funcionamiento del sistema:

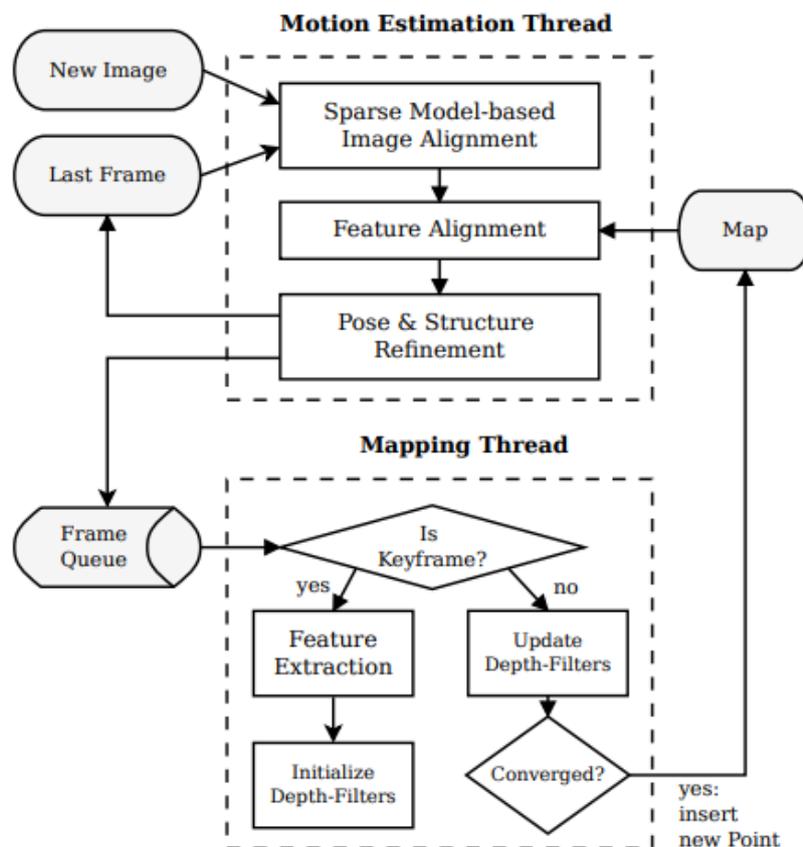


Figura 22: Diagrama de flujo de SVO [24].

Pero antes de entrar en detalle a comentarlo haremos un breve inciso para introducir cierta notación en la que nos apoyaremos, descrita en [23]:

Todo punto 3D,  $p = (x, y, z)^T$ , perteneciente a la superficie de la escena visible (es decir,  $p \in S$ , con  $S \in \mathbb{R}^3$ ) puede mapearse a las coordenadas de la imagen,  $u = (\hat{u}, \hat{v})^T$ , a través del modelo de proyección de la cámara, denotado como en (15):

$$u = \pi({}_k p) \quad (15)$$

donde el subíndice k indica que las coordenadas del punto se expresan para el sistema de referencia de la cámara. La proyección  $\pi: \mathbb{R}^3 \mapsto \mathbb{R}^2$  está determinada por los parámetros intrínsecos de la cámara, los cuales se conocen a partir de la calibración de la misma.

En sentido contrario, la ecuación (16) indica cómo el punto 3D correspondiente a una coordenada de imagen  $u$ , podría recuperarse a partir de la función de proyección inversa  $\pi^{-1}$  y la profundidad  $d_u \in \bar{R}$  (dominio de la imagen en el cual se conoce la profundidad):

$${}_k p = \pi^{-1}(u, d_u) \quad (16)$$

Por otra parte, la posición y orientación de la cámara acoplada a la aeronave para el instante genérico k se expresa con la “*rigid-body transformation*”  $T_{k,w}$  perteneciente al espacio de dimensión tres o  $SE(3)$ . Esto nos permite pasar un punto 3D desde un sistema de referencia “mundial” hasta el sistema de referencia de la cámara:

$${}_k p = T_{k,w} \cdot {}_w p \quad (17)$$

Y la transformación relativa entre dos fotogramas consecutivos se podría calcular como:

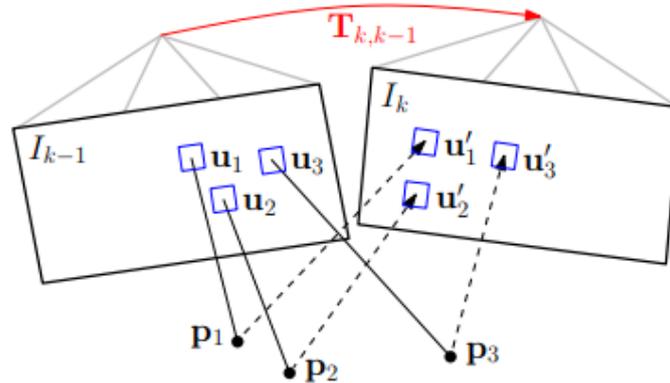
$$T_{k,k-1} = T_{k,w} \cdot T_{k-1,w}^{-1} \quad (18)$$

Ahora, teniendo presente el esquema de SVO de la **figura 22**, procedemos a su explicación. Comenzaremos por el bloque de movimiento, el cual implementa el enfoque semidirecto propuesto para la estimación de la posición relativa y se compone de 3 pasos o etapas.

### 3.2.1.- “Motion Estimation”

El primer paso es inicializar la posición a través de la *alineación de imágenes basada en un modelo disperso*. Es decir, se trata de encontrar la posición relativa de la cámara con relación al fotograma anterior minimizando el error fotométrico entre los píxeles correspondientes a la proyección de los mismos puntos 3D. Dichas diferencias fotométricas entre píxeles que observan el mismo punto 3D se definen como residuos de intensidad ( $\delta I$ ), y se utilizan para su cómputo parches de tamaño 4x4.

En otras palabras, cambiar la posición relativa  $T_{k,k-1}$  entre el fotograma anterior  $I_{k-1}$  y el actual  $I_k$  implica mover la posición de los puntos reproyectados en la nueva imagen; pues el “alineamiento de imágenes basado en modelo disperso” trata de encontrar la transformación  $T_{k,k-1}$  que minimiza la diferencia fotométrica (residuos) entre parches que se corresponden con el mismo punto 3D. Se ilustra esta idea en la **figura 23**.



**Figura 23:** Alineación de imágenes basada en modelo disperso [23].

El residuo de intensidad puede calcularse tomando un punto 2D,  $u$ , del fotograma anterior  $I_{k-1}$  y proyectándolo en el fotograma actual  $I_k$ , lo que matemáticamente se describe como:

$$\delta I(T, u) = I_k \left( \pi \left( T \cdot \pi^{-1}(u, d_u) \right) \right) - I_{k-1}(u) \quad (19)$$

En aras de la simplicidad puede asumirse que los residuos de intensidad siguen una distribución normal de varianza unidad, con lo que encontrar la  $T_{k,k-1}$  que minimice el error fotométrico de todos los parches se asemejaría a un problema de mínimos cuadrados, concretamente con la forma de la ecuación (20):

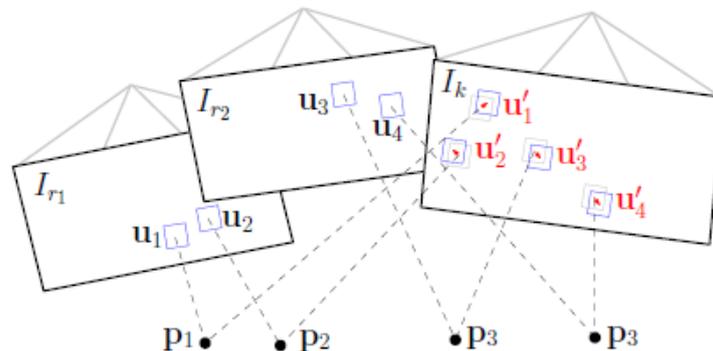
$$T_{k,k-1} = \arg \min_{T_{k,k-1}} \frac{1}{2} \cdot \sum_{i \in \bar{R}} \left\| \delta I(T_{k,k-1}, u_i) \right\|^2 \quad (20)$$

El segundo paso en el bloque de estimación de movimiento es la *alineación de características*. En la etapa anterior que acabamos de ver, se alineaba la cámara con respecto al fotograma previo, y la posición relativa encontrada ( $T_{k,k-1}$ ) define implícitamente una estimación inicial de las posiciones de características de todos los puntos 3D visibles en la nueva imagen. No obstante, esa estimación inicial puede ser mejorada. Y para ello, se debe alinear la posición de la cámara con respecto al mapa en lugar de al fotograma anterior (tal y como se indicaba en la **figura 22**).

Todo punto 3D del mapa visible desde la posición estimada de la cámara se proyecta en la imagen, resultando una estimación de las posiciones de características 2D correspondientes ( $u_i'$  en la **figura 24**). La etapa de alineación de características

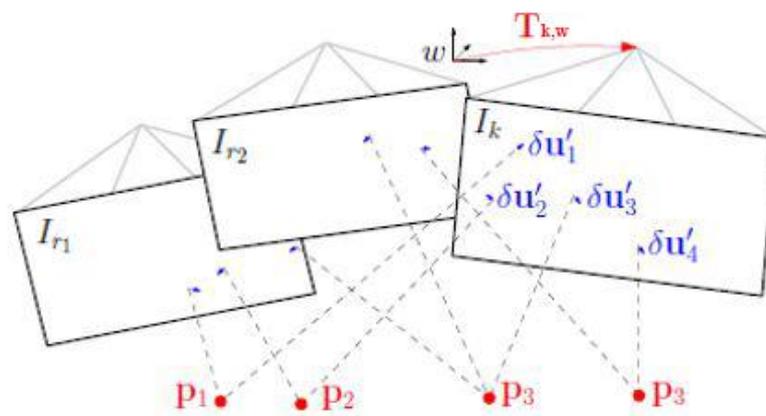
optimiza individualmente todas las posiciones 2D ( $u_i$ ) en la nueva imagen minimizando el error fotométrico del parche en la imagen actual con respecto al parche correspondiente en el fotograma clave<sup>11</sup>  $r_i$ . Por ello se dice que en SVO la correspondencia de características es un resultado implícito de la estimación de movimiento directo (de ahí que reciba el nombre de método semidirecto), en lugar de una extracción y comparación explícita.

Se muestra gráficamente el procedimiento en la **figura 24**:



**Figura 24:** Alineación de características: El error fotométrico entre parches correspondientes en el fotograma actual y los fotogramas clave previos se puede minimizar optimizando individualmente la posición 2D de cada parche [23].

En el tercer y último paso, se optimizan la posición de la cámara y la estructura (es decir, puntos 3D) para minimizar el error de proyección introducido en la etapa anterior de alineación de características. Se ilustra en la **figura 25**, donde los parámetros a optimizar en este proceso se indican en color rojo:



**Figura 25:** Refinamiento de posición y estructura.

<sup>11</sup> Para cada punto proyectado se identifica el fotograma clave  $r_i$  que observa el punto con el ángulo de observación más cerrado.

Como se aprecia en la figura anterior, en este último paso se optimiza nuevamente la posición de la cámara  $T_{k,w}$  para minimizar los residuos, lo cual es modelado de la siguiente forma:

$$T_{k,w} = \arg \min_{T_{k,w}} \frac{1}{2} \cdot \sum_i \|u_i - \pi(T_{k,w} \cdot w p_i)\|^2 \quad (21)$$

Y acto seguido, mediante la minimización del error de proyección, se optimizan las posiciones de los puntos 3D observados (optimización de la estructura).

Nótese que la optimización de posición y estructura efectuada en este paso final no es conjunta. Primero se optimiza la posición de la cámara  $T_{k,w}$ , lo cual se conoce como *motion-only bundle adjustment*, y a continuación se optimiza la posición de los puntos 3D, *structure-only bundle adjustment*.

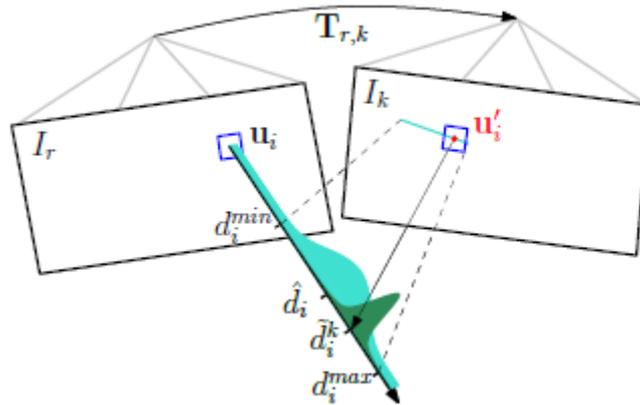
Con esto, podemos dar por concluido el bloque de estimación de movimiento. No obstante, resultaría interesante comentar un último aspecto acerca del mismo. Y es que podría pensarse que para el objetivo de estimar el movimiento de la cámara fuese suficiente con el primer paso de los tres descritos. Sin embargo, en el plano experimental, el uso conjunto de las tres etapas da lugar a una mejora significativa de la deriva en comparación con el uso del primer paso únicamente, que obtiene una desviación más pronunciada. Esta mejora de precisión se debe a la alineación de la nueva imagen también con respecto a fotogramas clave y mapa, en lugar de hacerlo sólo respecto al fotograma anterior.

### 3.2.2.- “Mapping”

Pasamos ahora a comentar el mapeado en SVO. Pese a que ahondar en un análisis pormenorizado del modelado y la actualización del filtro de profundidad en el que se basa este bloque no es objetivo troncal de este trabajo, abordaremos las nociones básicas que nos permitan tener una visión del sistema lo más integral posible.

Este bloque estima la profundidad de las características 2D para las que su correspondiente punto 3D aún no se conoce. La estimación de profundidad de una característica se modela con una distribución de probabilidad, y cada observación posterior sirve para actualizarla. Cuando la varianza de la distribución se vuelve lo suficientemente pequeña, la estimación de profundidad se inserta en el mapa como punto 3D e inmediatamente puede utilizarse para la estimación de movimiento, tal como se indica en el esquema de trabajo de SVO mostrado en la **figura 22**.

Nos apoyaremos en la **figura 26** para estudiar la estimación probabilística de la profundidad. Cada filtro de profundidad se asocia a un fotograma clave (el inicializar nuevos puntos 3D tendrá lugar cuando se seleccione un nuevo keyframe). Y en cada nueva observación tomada, se busca en línea epipolar de la nueva imagen el parche que tenga la mayor correlación con el parche correspondiente a la característica extraída en el fotograma clave con el detector FAST. El punto de mayor correlación  $u_i'$  se corresponde con la profundidad triangulada  $d_i^k$  (como se muestra en la figura), que será utilizada para actualizar la estimación de profundidad bajo un enfoque bayesiano [25] hasta que la varianza sea suficientemente pequeña.



**Figura 26.** Estimación de profundidad  $\hat{d}_i$  para una característica  $i$  en el fotograma clave  $r$  [23].

La profundidad  $d_i^k$  se modela como la mezcla de una distribución gaussiana alrededor de la profundidad real  $d_i$ , y una distribución uniforme en el intervalo  $[d_i^{min}, d_i^{max}]$  que mide los *outliers* o valores atípicos.

### 3.2.3.- Comentarios Adicionales

Acabamos de explicar cómo trabaja el algoritmo SVO, pero antes de probar su simulación vamos a introducir algunos detalles interesantes relativos a su implementación experimental:

- El primero tiene que ver con el hilo del mapeado. Las imágenes se dividen en cuadrículas de tamaño fijo (en la práctica de 30x30 píxeles), y en ellas se inicializará un nuevo filtro de profundidad en la característica extraída con FAST a menos que ya exista una correspondencia 2D-3D. Como consecuencia de esto, se tendrán características uniformemente distribuidas en la imagen. Además, este tamaño de celda también es el utilizado para proyectar el mapa antes del paso de alineación de características en el hilo de estimación de movimiento.

Cabe destacar también que la principal ventaja del mapeado de SVO es que se observan bastantes menos puntos atípicos (*outliers*) debido a que cada filtro de profundidad se somete a numerosas mediciones hasta alcanzar la convergencia, resultando estimaciones más confiables.

- En cuanto al arranque del sistema, este se inicia para obtener la posición de los dos primeros fotogramas clave y el mapa inicial, el cual es triangulado a partir de las dos primeras observaciones.

- Otro aspecto reseñable sobre la implementación es que se requieren movimientos suaves para no incrementar de manera considerable la incertidumbre en profundidad.

- Y para finalizar, comentemos un concepto importante al que hemos venido haciendo referencia reiteradamente, los fotogramas clave o *keyframes*. Podemos definir los

*keyframes* como ciertos fotogramas seleccionados que se usan como referencia, concretamente en SVO para la alineación de características y el refinamiento de la estructura. Además, la extracción de características sólo es necesaria cuando se selecciona un nuevo fotograma clave para inicializar nuevos puntos 3D (revisar **figura 22**). Como consecuencia se tiene una mayor velocidad debido a que no es necesario extraer características en cada adquisición.

Pero SVO no ha sido la primera ni última técnica en emplear estos *keyframes*; ya se habían utilizado anteriormente en PTAM y también poco después en ORB-SLAM. El trabajo de Strasdat et. al[26] demostró que las técnicas basadas en fotogramas clave (que típicamente separan el mapeado de la estimación de movimiento), son más precisas que los enfoques que procesan todos los fotogramas para estimar conjuntamente la posición de la cámara y las ubicaciones de las características del mapa.

La inclusión de fotogramas clave en SVO sigue una política similar a la de PTAM. Los fotogramas clave se insertan con cuidado basándose en criterios de distancia a otros *keyframes*. Esta restrictiva política de inserción se debe a que la complejidad computacional crece con el número de fotogramas clave. Así pues, por razones de eficiencia, el algoritmo SVO mantiene un número fijo de *keyframes* en el mapa, de manera que cuando se inserta un nuevo fotograma clave, se elimina el que esté más alejado de la posición actual de la cámara. La configuración SVO que simularemos tendrá fijado en 10 el número máximo de *keyframes*.

# 4. IMPLEMENTACIÓN: ROBOT OPERATING SYSTEM

---

Previamente a realizar el estudio experimental de SVO mediante la ejecución de una simulación, es conveniente describir detalladamente el entorno en el cual se llevará a cabo dicha prueba. Y como venimos anunciando, ese entorno de trabajo no será otro que *Robot Operating System*. Así pues, dedicaremos este apartado a introducir ROS y explicar las herramientas en las que más nos apoyaremos, además de guiar su instalación.

## 4.1- Introducción a ROS

En primer lugar, tratemos de responder a la pregunta “¿Qué es ROS?”. Diseñar software robótico se antoja una tarea complicada por lo que la comunidad de usuarios dedicados a ello, que en los últimos años ha crecido sustancialmente, tiene entre sus objetivos fomentar la reutilización de códigos, facilitando así la investigación y el desarrollo de la robótica. Parece lógico entonces disponer de un entorno cuyas herramientas y bibliotecas favorezcan el desarrollo de software complejo y robusto, y que además sea fácil de integrar con otras plataformas robóticas. Precisamente éste es uno de los propósitos de ROS, un framework de código abierto ideado en 2007 por la Universidad de Stanford para el despliegue de software para robots.

ROS a menudo se define como un meta-sistema operativo, ya que proporciona la mayoría de los servicios provistos por cualquier sistema operativo, como por ejemplo el control de dispositivos o el paso de mensajes entre procesos. Pero es importante entender que ROS no es un sistema operativo como tal, sino que necesita alojarse o funcionar encima de uno. En la actualidad sólo se implementa sobre sistemas Unix, concretamente está orientado a Ubuntu, aunque poco a poco se está acondicionando también a plataformas como MAC OS X, Windows y otros sistemas Linux como Fedora, Arch o Gentoo [27].



**Figura 27.** Logotipos de Ubuntu (Izquierda) y ROS (Derecha).

El sistema ROS se libera regularmente en versiones conocidas como “Distribuciones”. Están ligadas a distribuciones de Ubuntu, y lanzan el núcleo, herramientas y bibliotecas del sistema. Hasta un total de 12 distribuciones se han difundido hasta la fecha, desde la primera versión, Box Turtle, hasta la más reciente, Melodic Morenia. A nivel comercial podemos mentar una peculiaridad de ROS, que es que las distribuciones no van asociadas a un número de versión, sino a un nombre distintivo que sigue alfabéticamente al anterior. En repositorios y foros de internet la comunidad de usuarios ya anda preguntándose cuál será el nombre de la siguiente distribución.

El siguiente listado muestra todas las distribuciones junto con su fecha de lanzamiento [28]:

<b>DISTRIBUCIÓN</b>	<b>LANZAMIENTO</b>
Box Turtle	02-03-2010
C Turtle	02-08-2010
Diamondback	02-03-2011
Electric Emys	30-08-2011
Fuerte Turtle	23-04-2012
Groovy Galapagos	31-12-2012
Hydro Medusa	04-09-2013
Indigo Igloo	22-07-2014
Jade Turtle	23-05-2015
Kinetic Kame	23-05-2016
Lunar Loggerhead	23-05-2017
Melodic Morenia	23-05-2018

Entre todas ellas nos hemos decantado por la distribución Indigo Igloo, algo antigua pero que goza aún de soporte, tratándose de una versión estable. Además, es una de las distribuciones en las que SVO ya ha sido probada satisfactoriamente.

Siguiendo con su filosofía *open source*, ROS dispone de una amplia gama de librerías que contribuyen al objetivo de colaborar y compartir el software robótico. Es común ver su desarrollo en lenguajes de programación actuales como C++ y Python, aunque también comienza a implementarse cada vez más en Java. Además, su estructura –de la que más tarde hablaremos – hace de ROS un entorno modular y escalado, ideal para sistemas grandes o con numerosos procesos. [27]

Una vez contestada la pregunta “¿Qué es ROS?”, la siguiente cuestión que inexorablemente nos aborda es “¿Cómo funciona ROS?”, a la cual trataremos de dar respuesta en el siguiente apartado.

#### **4.2- Estructura y funcionamiento.**

Para alcanzar una visión funcional de ROS, lo mejor es entenderlo como un ecosistema de procesos distribuidos que sigue una arquitectura de grafos. La definición y relación de sus principales elementos nos servirá para comprender cómo trabaja.

El primer concepto a definir, dado que será uno de los más invocados, son los **nodos**. Un nodo no es más que un proceso que ejecuta un cálculo o computación[29]. Pueden comunicarse entre ellos, estableciendo así un grafo de computación en forma de red peer-to-peer donde se procesan los datos de forma conjunta.

Por tanto, los nodos son quienes soportan la estructura modular de ROS, de manera que cualquier sistema implementado en ROS estará compuesto por múltiples nodos. Este diseño proporciona varias ventajas. La principal es que como se tiene cada proceso por separado, los posibles fallos quedarán aislados en nodos concretos sin afectar a la totalidad del sistema. Adicionalmente, supone también una mayor facilidad de cara al desarrollo del código.

La manera en que los nodos se relacionan nos da pie a introducir dos nuevos elementos: Mensajes y Topics. Podemos decir que los nodos se comunican entre sí a través de la publicación de mensajes en los topics.

Un **mensaje** es una estructura de datos simple, que viene especificada en los archivos de extensión “.msg”. Contiene campos escritos, y admite tipos primitivos convencionales como enteros, coma flotante, booleanos, etc [30].

Los **topics** son buses de datos en los que los nodos publican los mensajes. Decimos “publican” porque siguen un mecanismo del tipo publicación-suscripción. De esta forma, un nodo envía un mensaje publicándolo en un determinado topic, y a su vez, otro nodo interesado en los datos contenidos en ese mensaje se suscribirá a dicho topic. Por lo general, el transporte de mensajes en ROS está basado en TCP-IP, de ahí su denominación TCPROS. Es el transporte predeterminado, aunque también se admite el basado en UDP o UDPROS [31].

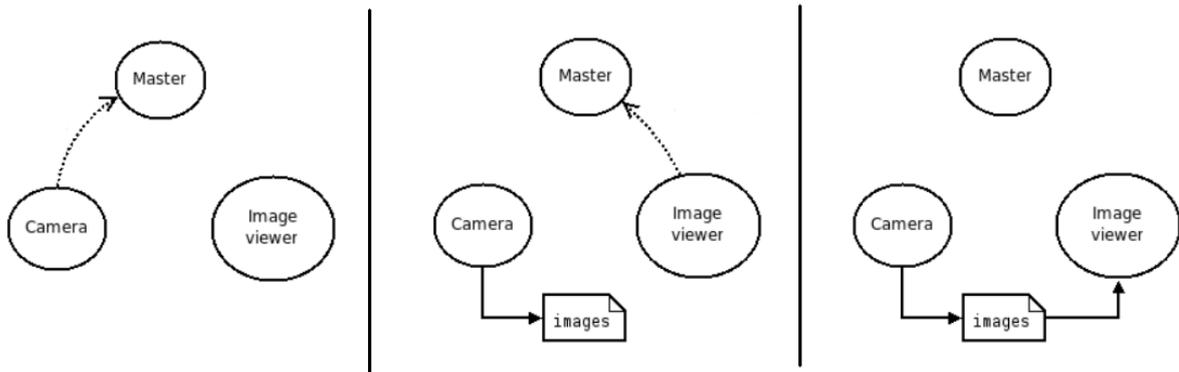
El paradigma publicación-suscripción es ideal para comunicaciones unidireccionales. Pero en caso de que ciertos nodos requiriesen respuestas para los mensajes transmitidos, en vez de topics deberán emplearse los **services**[32], que sí implementan el modelo de comunicación solicitud-respuesta. Para la aplicación que nos ocupa no será necesario recurrir a los ‘services’, haremos uso exclusivo de los topics.

Puede haber varios nodos publicadores y suscriptores para un mismo topic, así como un mismo nodo publicando o suscribiéndose a diferentes topics. En general los nodos, ya sean publicadores o suscriptores, no son conocedores de con qué otros nodos están comunicándose; sólo tienen constancia de los topics en los que publican o a los que se ha suscrito. Entonces, ¿cómo se consigue establecer la comunicación entre nodos? La

respuesta a esta pregunta nos lleva directamente a uno de los elementos fundamentales de ROS, el Maestro o **Master**.

El Master es el punto central donde los nodos en ejecución se registran. En él, además de los nodos, también constan todos los topics (y services, si los hubiera). Su principal función es permitir que los nodos se localicen entre ellos, para lo cual proporciona servicios de denominación y registro[33], además del servidor de parámetros<sup>12</sup>. Por todo ello, es posible trazar una analogía entre un servidor DNS y el Master de ROS.

Para concluir, a modo de resumen, podemos decir que los nodos, para suscribirse a los topics en los que se publiquen mensajes que les interesen, solicitarán al Master la ubicación de los nodos que publican en esos topics, y establecerán la conexión a través del protocolo TCPROS por medio de sockets TCP-IP estándar. Una vez que los nodos se han localizado, se comunican entre sí de igual a igual, es decir, sin pasar por el Master. La **figura 28** ejemplifica gráficamente el proceso.



**Figura 28.** El nodo 'Camera' pide al Master publicar en el topic 'images' (Izquierda). Ahora este nodo es publicador del topic, pero aún no hay envío de datos. Mientras, 'Image viewer' solicita suscribirse al topic (Centro). Ahora que el topic cuenta con publicador y suscriptor, el Master informa del establecimiento de la conexión y empieza la transferencia (Derecha) [33].

Otro elemento importante de ROS que utilizaremos en nuestra simulación, son los **bags**. Un bag es un formato de archivo que permite almacenar y reproducir datos de mensajes. Se trata de una de las principales herramientas para el manejo de datos, pues permite etiquetarlos, visualizarlos o almacenarlos para un uso posterior[34].

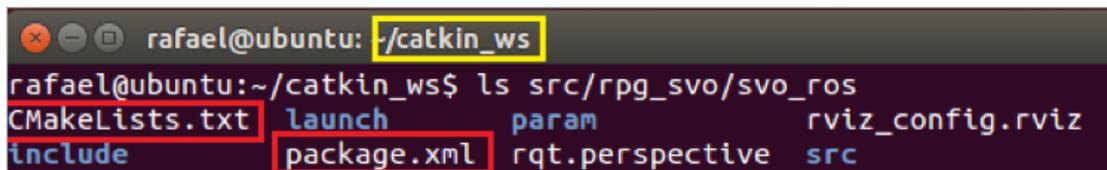
Con esto, ya tendríamos definidos todos los componentes del grafo que explica ROS a nivel de computación. No obstante, este framework también se explica a nivel de la comunidad que lo implementa, lo cual ya fue expuesto en el apartado anterior, y a nivel de sistema de archivos. Ya que hemos hablado tanto del nivel de computación y del nivel de comunidad, introduciremos también ROS en su organización como sistema. En

<sup>12</sup> El servidor de parámetros es un diccionario compartido de múltiples variables. Los nodos ROS acceden a él para almacenar y recuperar parámetros –típicamente de configuración– en tiempo de ejecución. Forma parte del Master.

este sentido, el principal recurso en ROS son los paquetes. Los **paquetes** son la unidad principal de organización de software, y constituyen el elemento más pequeño que se puede construir y distribuir[35]. Se pretende que los paquetes tengan suficiente funcionalidad para que el software resulte útil y fácil de reutilizar, sin que lleguen a ser demasiado pesados.

Cada paquete puede contener nodos, librerías, datasets, scripts, ejecutables, etc. Además de ello, todos los paquetes vienen acompañados por un fichero “package.xml” conocido como “manifiesto de paquete”, que suministra metadatos sobre el paquete como pueden ser su nombre, información sobre la licencia o su versión. Dentro de cada paquete también aparecerá un fichero “CMakeLists.txt”, dedicado a la compilación.

Los conceptos definidos a lo largo de este apartado son los artefactos en los que nos apoyaremos para desenvolvemos dentro del espacio de trabajo de ROS, que recibe el nombre de “catkin\_ws” prácticamente por convención dentro de la comunidad, en referencia a la herramienta de compilación de ROS, denominada catkin.



```
rafael@ubuntu: ~/catkin_ws
rafael@ubuntu:~/catkin_ws$ ls src/rpg_svo/svo_ros
CMakeLists.txt  launch  param  rviz_config.rviz
include         package.xml  rqt.perspective  src
```

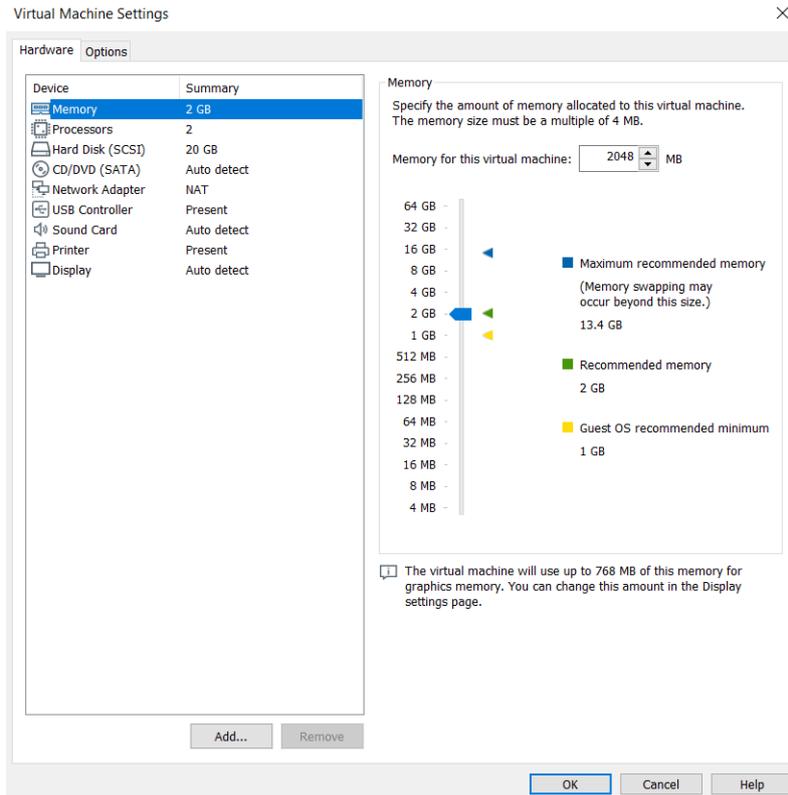
**Figura 29.** Contenido de un paquete, indicando la ruta de este desde el directorio de trabajo, resaltado en amarillo. En rojo se resaltan los mencionados ficheros asociados a todo paquete.

### **4.3- Instalación del Software: ROS Y SVO.**

En el presente apartado describiremos el hardware utilizado para llevar a cabo la prueba experimental de la técnica SVO, así como la enumeración de los pasos dados para la instalación del software pertinente.

El hardware empleado consiste en un portátil Lenovo ideapad modelo 330-15IKB equipado con un procesador de 64 bits Intel i7-7500U con frecuencia base 2.7GHz, memoria RAM de 16GB y disco duro HDD de 1TB. Dado que el sistema operativo por defecto es Windows 10 y que ROS se implementa sobre sistemas Unix, se nos presentan dos opciones: hacer una partición del disco duro o instalar Ubuntu en una máquina virtual. Finalmente nos decantamos por la segunda.

La máquina virtual descargada e instalada en nuestro equipo es VMware Workstation 15 Player, donde cargamos la imagen ISO del sistema operativo Ubuntu 14.04 de 64 bits para su instalación. La configuración de la máquina virtual, mostrada en la **figura 30**, se ha elegido con vistas a ejecutar la simulación en pleno rendimiento, dedicándole dos núcleos del procesador, 20GB de disco duro y 2GB de memoria.



**Figura 30:** Configuración de parámetros de la máquina virtual.

Una vez tenemos disponible Ubuntu en nuestra máquina virtual, lo siguiente será instalar ROS. La distribución escogida, tal y como ya adelantamos, es Indigo Igloo. Así pues, contaremos con la combinación de Ubuntu 14.04 y ROS Indigo Igloo, una de las posibilidades testeadas por los mismos desarrolladores de SVO.

Para ello seguimos las indicaciones compartidas en el apartado dedicado en la propia web de ROS [36]. Ahí se proporcionan diferentes configuraciones de entre las cuales elegimos la *Desktop-Full*, opción recomendada. A continuación, se enumeran los pasos seguidos para la instalación de esta distribución con la citada configuración. Desde el terminal, ejecutamos:

Para preparar al dispositivo a recibir los paquetes de software de las url indicadas:

- `sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'`
- `sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654`

Para asegurar que los paquetes están actualizados:

- `sudo apt-get update && sudo apt-get install dpkg`

Para instalar la configuración *Desktop-Full*, provista de herramientas de visualización, simuladores 2D/3D o bibliotecas genéricas:

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

Para ejecutar algunos componentes necesarios antes de poder utilizar ROS:

- `sudo rosdep init`
- `rosdep update`

Para agregar importantes variables de entorno:

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

Con ROS ya disponible en nuestro sistema Ubuntu, es momento de instalar SVO. Para ello seguimos los pasos suministrados en el GitHub<sup>13</sup> del grupo de la Universidad de Zúrich desarrollador de esta técnica[37]. Se listan a continuación los comandos ejecutados, con los que instalamos herramientas útiles y clonamos directamente los directorios completos del GitHub con los paquetes necesarios ya clasificados convenientemente.

Lo primero es construir el espacio de trabajo:

- `mkdir catkin_ws`

Dentro, clonamos el directorio “Sophus”, contenedor de herramientas necesarias para describir transformaciones:

- `cd catkin_ws`
- `git clone https://github.com/strasdat/Sophus.git`
- `cd Sophus`
- `git checkout a621ff`
- `mkdir build`
- `cd build`
- `cmake ..`
- `make`

Instalamos también el paquete que implementa el detector Fast:

- `cd catkin_ws`
- `git clone https://github.com/uzh-rpg/fast.git`
- `cd fast`
- `mkdir build`
- `cd build`
- `cmake ..`
- `make`

Generamos dentro del espacio de trabajo un directorio donde alojar los paquetes propios del proyecto:

- `cd catkin_ws`
- `mkdir src`

---

<sup>13</sup> GitHub es una plataforma colaborativa online para desarrolladores, que hace las veces de repositorio donde alojar y compartir el código fuente.

Y copiamos en él la carpeta “vikit”, que contiene algunos modelos de cámara y funciones matemáticas necesarias:

- `cd catkin_ws/src`
- `git clone https://github.com/uzh-rpg/rpg_vikit.git`

Instalamos algunas dependencias importantes, para la compilación entre otras utilidades:

- `sudo apt-get install ros-indigo-cmake-modules`

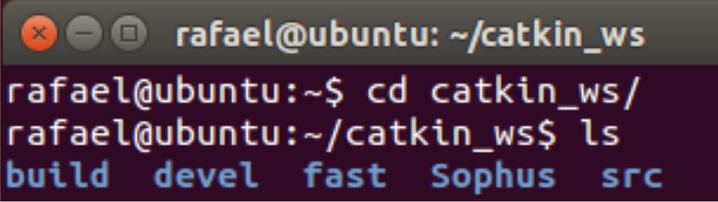
Llegados a este punto, clonamos SVO dentro de nuestro espacio de trabajo:

- `cd catkin_ws/src`
- `git clone https://github.com/uzh-rpg/rpg_svo.git`

Para terminar, compilamos:

- `catkin_make`

Al finalizar la ejecución de esta orden, podremos apreciar que se han generado dos nuevos directorios dentro de nuestro espacio de trabajo: *build* y *devel*. *Build* es donde catkin almacena elementos como bibliotecas o ejecutables C++. Por su parte *devel* contiene varios directorios y archivos entre los que destacan los de configuración, cuya ejecución prepara al sistema para usar este espacio de trabajo y el código en él contenido[38]. La siguiente figura muestra la disposición del directorio de trabajo:

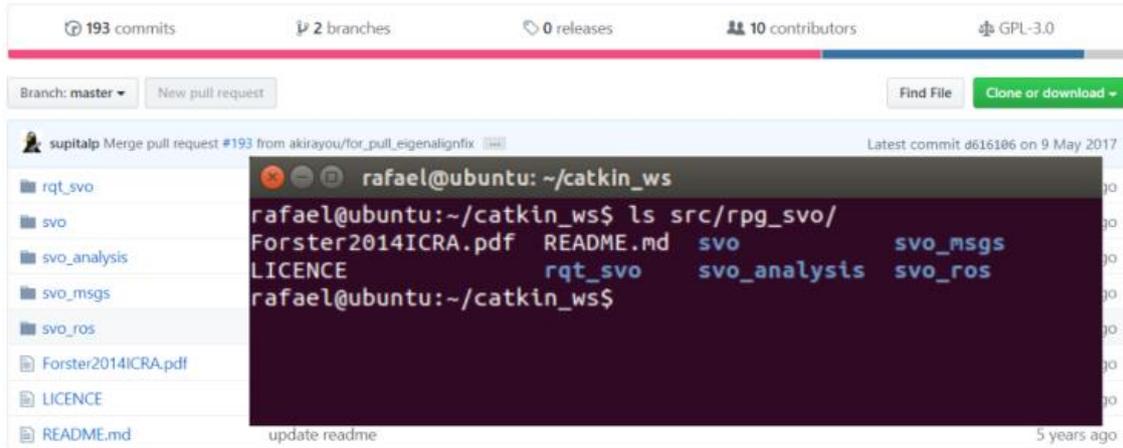


```
rafael@ubuntu: ~/catkin_ws
rafael@ubuntu:~$ cd catkin_ws/
rafael@ubuntu:~/catkin_ws$ ls
build  devel  fast  Sophus  src
```

**Figura 31.** Directorios en el espacio de trabajo tras la instalación.

Podemos ver que además de las mencionadas *build* y *devel*, aparecen las carpetas *Sophus* y *fast*, correspondientes a los paquetes importados durante la instalación, y la carpeta *src* donde a su vez alojamos la carpeta de SVO (“rpg\_svo”) clonándola directamente desde el GitHub correspondiente.

En la siguiente figura podemos observar el resultado de ejecutar el comando *git clone*, ya que al listar el contenido del directorio “rpg\_svo” se aprecia que es idéntico al del GitHub [39]:



193 commits 2 branches 0 releases 10 contributors GPL-3.0

Branch: master New pull request Find File Clone or download

supitalp Merge pull request #193 from akirayou/for\_pull\_eigenalignfix Latest commit d616106 on 9 May 2017

```
rafael@ubuntu: ~/catkin_ws
rafael@ubuntu:~/catkin_ws$ ls src/rpg_svo/
Forster2014ICRA.pdf  README.md  svo        svo_msgs
LICENCE             rqt_svo   svo_analysis  svo_ros
rafael@ubuntu:~/catkin_ws$
```

update readme 5 years ago

**Figura 32.** Detrás, contenido del repositorio web. Superpuesto, el terminal listando el contenido del directorio “rpg\_svo”. El comando git clone funcionó correctamente.

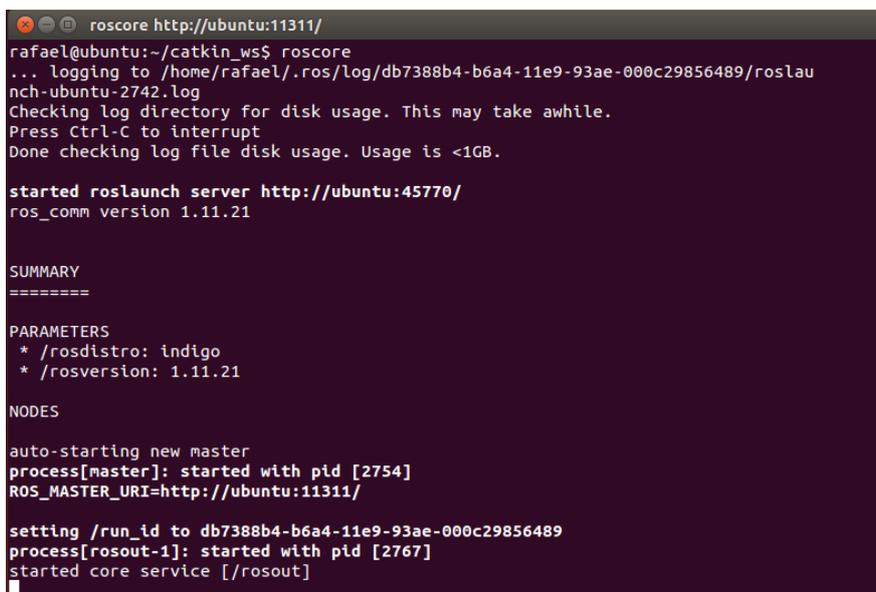
Para concluir, cabe destacar que adicionalmente resultó necesario la instalación de OpenCV. El motivo es que este software ofrece un funcional módulo de visualización 3D requerido para utilizar *rviz*, la herramienta de visualización de ROS. Inicialmente ejecutamos la prueba sin tener instalado OpenCV en Ubuntu, de manera que no podía apreciarse el resultado final. La versión de OpenCV en nuestro caso utilizada ha sido la 2.4.9

# 5. PRUEBAS Y RESULTADOS

Dedicaremos este capítulo a lanzar la simulación de SVO dentro del ecosistema ROS y a comentarla con detalle. Para ello analizaremos los resultados observados combinando la inspección de los mismos y el uso de útiles herramientas que ofrece ROS. La mayoría de estas herramientas son órdenes para el *bash*, el intérprete de comandos (*shell*) de Ubuntu al que venimos refiriéndonos como terminal. Por ello, antes de comenzar, es oportuno mencionar las fuentes a través de las cuales hemos estudiado la aplicación de estos comandos. Destaca el libro “*Programming Robots with ROS*” de Quigley et al. [38], el cual empleamos para entender el uso y la información brindada por cada comando, sin olvidar los espacios dedicados a cada orden en la wiki de ROS o la conferencia del investigador Francisco Martín (miembro del Grupo de Robótica de la URJC) celebrada en el colectivo de experimentación tecnológica *HackLab* de Almería y compartida en su canal de YouTube [40].

## 5.1- Ejecución de la simulación.

Sin más preámbulos, comenzamos describiendo los pasos necesarios[41] para correr la simulación<sup>14</sup>. Será necesario tener abiertos al menos 4 terminales, aunque a medida que vayamos realizando pruebas con las distintas herramientas, el número de terminales en uso aumentará. Lo primero que debemos ejecutar al utilizar ROS es la orden **roscore**, requisito previo para que los nodos puedan comunicarse ya que *roscore* se encarga de iniciar el Master.



```
roscore http://ubuntu:11311/
rafael@ubuntu:~/catkin_ws$ roscore
... logging to /home/rafael/.ros/log/db7388b4-b6a4-11e9-93ae-000c29856489/roslau
nch-ubuntu-2742.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:45770/
ros_comm version 1.11.21

SUMMARY
=====

PARAMETERS
* /rostdistro: indigo
* /rosversion: 1.11.21

NODES

auto-starting new master
process[master]: started with pid [2754]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to db7388b4-b6a4-11e9-93ae-000c29856489
process[rosout-1]: started with pid [2767]
started core service [/rosout]
```

Figura 33. Ejecución del comando *roscore* en nuestro sistema.

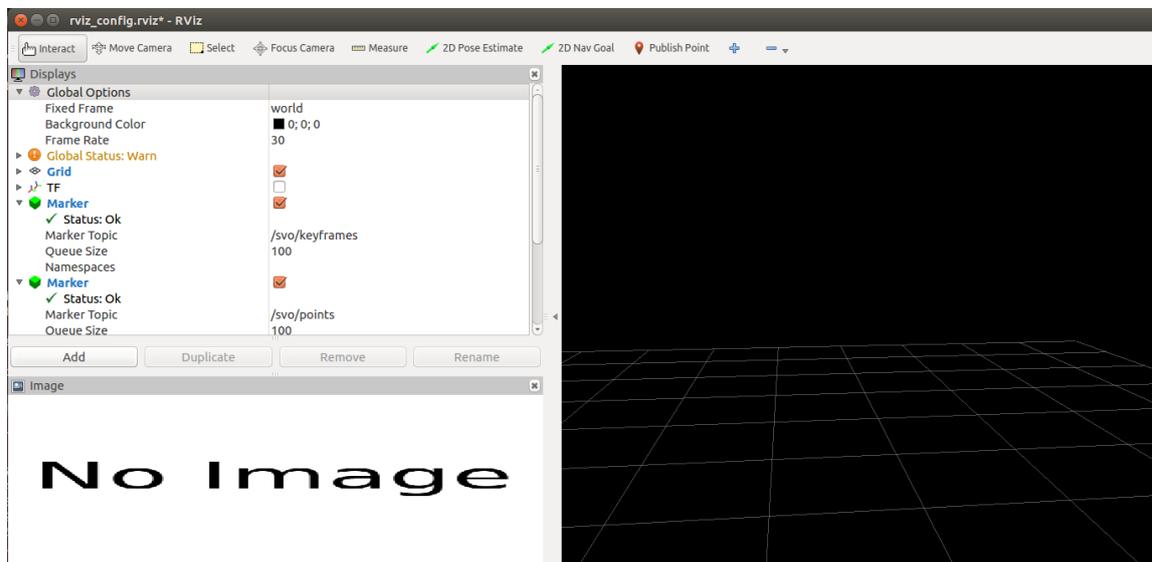
<sup>14</sup> Las órdenes introducidas por línea de comandos que se especifican en este apartado las ejecutamos todas desde el directorio de trabajo, *catkin\_ws*, a menos que se indique lo contrario.

Para aplicaciones de mayor dimensión, al tratarse de un sistema distribuido, la orden `echo $ROS_MASTER_URI` puede ser de ayuda para conocer dónde está el Master ya que devuelve su dirección IP y puerto. En nuestro caso se ejecuta todo en local, dentro del mismo dispositivo, y el resultado de mostrar el contenido de esa variable es “http://localhost:11311”.

A continuación, desde un nuevo terminal, lanzaremos SVO. Pero antes es necesario introducir la orden `source devel/setup.bash` para configurar el espacio de trabajo con el que se desee trabajar. Los scripts del tipo “setup.bash” controlan qué versión de ROS y qué espacio de trabajo están activos, de manera que si no se ejecutan, el bash no sabrá dónde encontrar los códigos. No ingresar este comando fue un error en el que reincidimos en los primeros intentos de simulación.

Una vez hecho esto ya estamos en disposición de lanzar SVO, y para ello ejecutamos `roslaunch svo_ros test_rig3.launch`. `roslaunch` es un recurso de ROS que permite lanzar múltiples nodos de una vez en lugar de lanzarlos uno a uno con la herramienta `roslaunch`. Para ello se le especifica el nombre del paquete y el fichero de configuración XML –con extensión “.launch”– que establece los nodos a iniciar.

El siguiente elemento a iniciar es la herramienta de visualización 3D de ROS, conocida como `rviz`. Desde un tercer terminal ejecutamos la orden `roslaunch svo_ros test_rig3.launch`, y se abrirá el visualizador configurado según el fichero cuya ruta hemos indicado. Lo mostramos en la siguiente figura<sup>15</sup>:



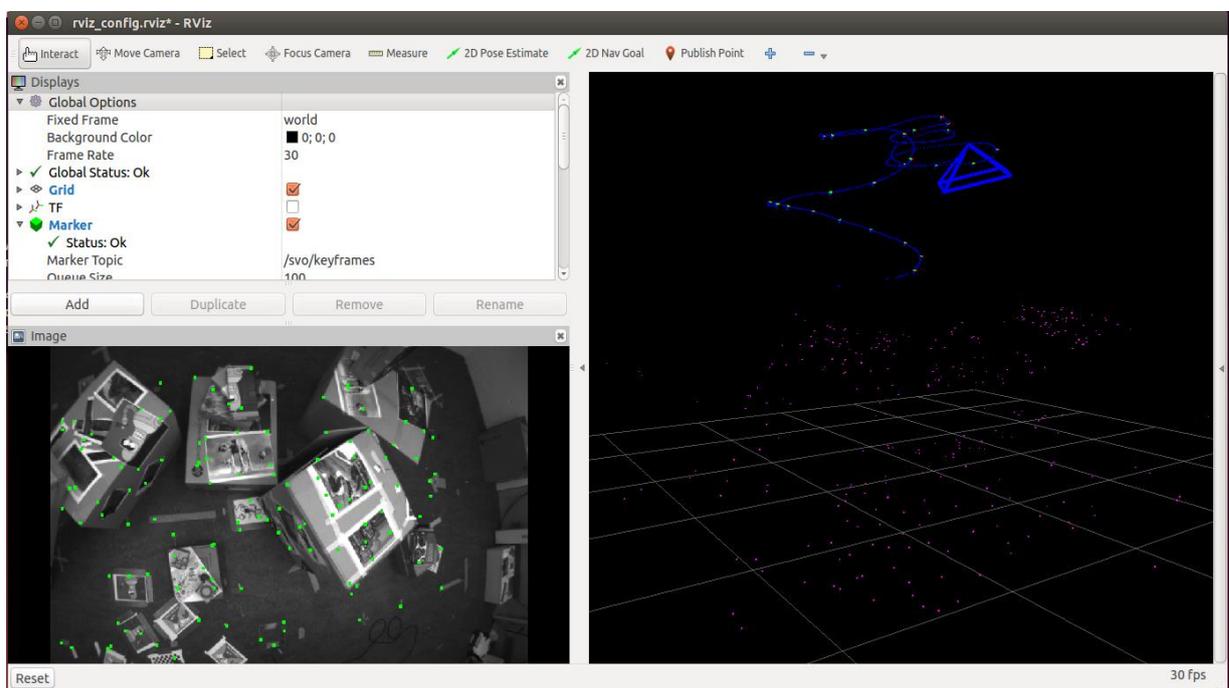
**Figura 34.** Apertura de `rviz` con la configuración apropiada para la simulación. Se distinguen 3 espacios: la ventana donde se representará la reconstrucción (derecha), la ventana donde se visualizará la grabación (izquierda abajo) y las opciones de `rviz` (izquierda arriba).

<sup>15</sup> La **figura 34** no muestra exactamente el visualizador `rviz` tal y como aparece al invocarlo con la configuración del fichero de extensión “.rviz”. Adicionalmente, desde las opciones del propio visor, hemos customizado a nuestro gusto un par de parámetros que facilitan la visualización: hemos cambiado el color de fondo de blanco a negro para que se vean mejor los puntos, y bajado el offset del grid, ya que algunos puntos quedaban por debajo de la rejilla dando sensación de aparecer en el subsuelo (concretamente hemos fijado la z del grid en -1.5 metros).

Como se aprecia en la figura anterior, en este punto todavía no es visible el resultado de la simulación. Esto es debido a que aún no hemos corrido el dataset con el contenido de la grabación, con lo que evidentemente ni el nodo “svo”, encargado de la odometría, ni el nodo ejecutor de rviz están recibiendo imágenes.

El dataset almacena sus datos en un formato tipo *bag* por lo que su manejo debe hacerse por medio de *rosvbag*, una herramienta de línea de comandos dedicada a trabajar con este tipo de formato, y que entre otras cosas permite reproducir el contenido de un bag, resumir su contenido o grabar un nuevo archivo bag con el contenido de los topics deseados. Como en nuestro sistema el dataset se encuentra descargado en el directorio “Downloads”, accedemos a él desde un nuevo terminal y una vez dentro ejecutamos la orden ***rosvbag play <nombre del dataset>***.

Tras esto, la simulación habrá comenzado y podremos observarla en el rviz:



**Figura 35.** Puesta en marcha de la simulación. Captura de un instante aleatorio.

## 5.2- Análisis del resultado.

Es momento de examinar el resultado de la simulación. En el presente apartado trataremos de encontrar el mayor número de evidencias que respalden los detalles sobre SVO y ROS comentados a lo largo del estudio.

Las primeras observaciones que podemos destacar saltan a la vista ya en el terminal desde el que lanzamos SVO mediante la orden *roslaunch*. Al ejecutarla, se muestran mensajes de información sobre configuraciones de parámetros mientras SVO se inicializa. Recordemos que dijimos que el algoritmo dividía las imágenes en rejillas de tamaño fijo, concretamente de 30x30 píxeles. Relativo a esto, uno de los mensajes que podemos apreciar en el terminal es el siguiente:

```
[ INFO ] [1565026614.172548698]: Found parameter: svo/grid_size, value: 30
```

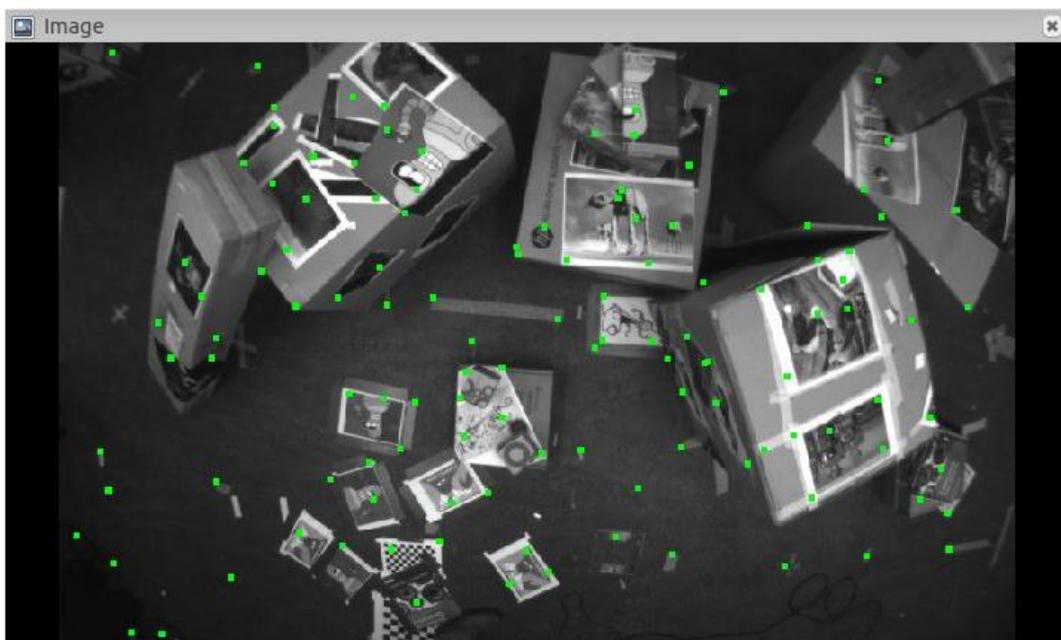
**Figura 36.** Tamaño de las cuadrículas en las que se dividen las imágenes. Captura del terminal.

Otro mensaje que nos ayuda a contrastar la información sobre SVO tiene que ver con el número de *keyframes*. Comentábamos que por motivos de eficiencia SVO mantiene fijado el número de keyframes, de manera que al insertar uno nuevo se elimine el más alejado de la posición en ese momento, y que ese valor está fijado en 10. Así pues, el siguiente mensaje, impreso en el mismo terminal, corrobora este tema:

```
[ INFO ] [1565026614.188977749]: Found parameter: svo/max_n_kfs, value: 10
```

**Figura 37.** Valor fijado para la cantidad de *keyframes* a utilizar. Captura del terminal.

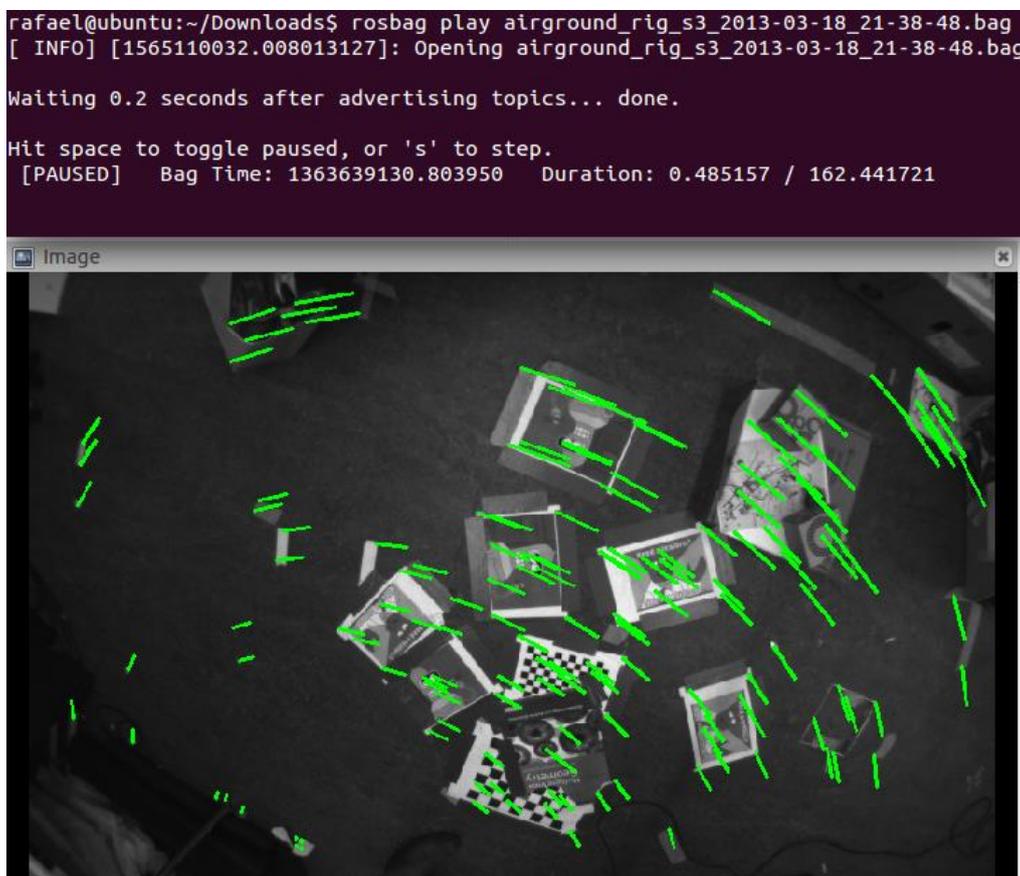
Analicemos ahora los resultados que apreciamos en el rviz una vez que la simulación está en marcha. En la ventana inferior izquierda podemos observar las imágenes capturadas con la particularidad de que en ellas aparecen resaltadas en color verde las características extraídas. Mostramos en la siguiente figura una de esas imágenes para un instante arbitrario:



**Figura 38.** Imagen vista desde el rviz. Por defecto incluye las características resaltadas.

Podemos apreciar cómo las características en su gran mayoría se corresponden con esquinas en la imagen, y también, aunque en menor medida, con bordes. En definitiva, se seleccionan los puntos con mayor gradiente. Este resultado se explica por la utilización de FAST que, recordemos, se trata de un detector de esquinas entre cuyas bondades se encuentra la varianza al escalado y a la rotación. Además, como consecuencia de que FAST actúe en cada cuadrícula de 30x30 píxeles que dividen la imagen, las características tienden a encontrarse uniformemente distribuidas a lo largo de esta, como también podemos intuir en la figura anterior.

Ahora bien, la claridad con la que se aprecian las características no es en todo momento como acabamos de mostrar en la **figura 38**. En el intervalo entre la toma del primer y segundo fotograma –hay que estar muy fino para detener la simulación justo al arrancar– las imágenes en el rviz se presentan como en la **figura 39**, que muestra un instante correspondiente al breve lapso de tiempo entre la selección de ambos:



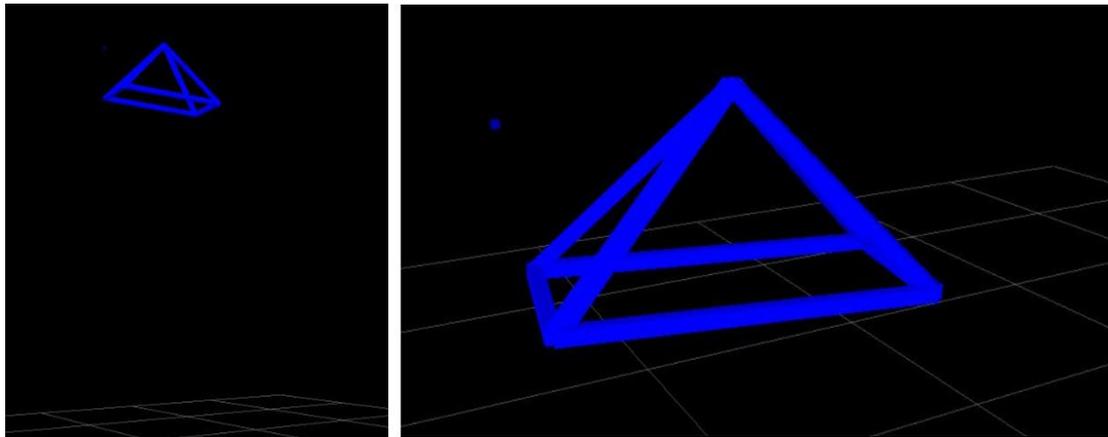
**Figura 39.** Vista de la grabación en un instante comprendido entre el primer y segundo fotograma. Nótese el efecto que presentan las características en estos momentos.

Esto efecto se debe a que entre el primer y segundo *frame* las características no pueden ser emparejadas.

Además, en su momento comentábamos que el mapa inicial es triangulado a partir de los dos primeros fotogramas, lo cual es refrendado ahora en el terminal encargado de ejecutar SVO, mediante el mensaje “[INFO] [1565457684.318221339]: Init: Selected second frame, triangulated initial map”. Pero ¿qué implica esto a nivel de visualización? En primer lugar, en el intervalo entre ambos no se inserta ningún punto en el mapa. Y

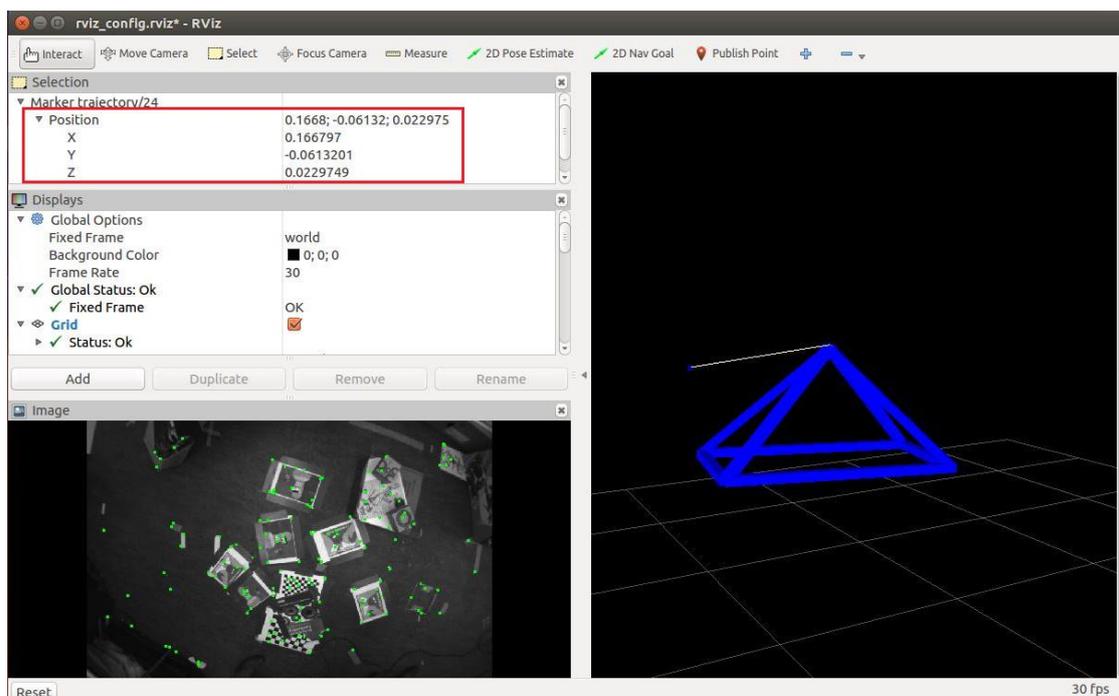
lógicamente, tampoco se puede estimar la ruta entre las posiciones donde se toman el primer y segundo *frame*. Por tanto, en esos instantes la reconstrucción de trayectoria y mapa aún está en ciernes.

La **figura 40** corresponde al preciso instante en el que se selecciona el segundo fotograma. En ella podemos notar que entre el punto azul inicial (punto de partida) y la posición actual, efectivamente, no se ha reconstruido la ruta seguida y tampoco hay aún ningún punto mapeado.



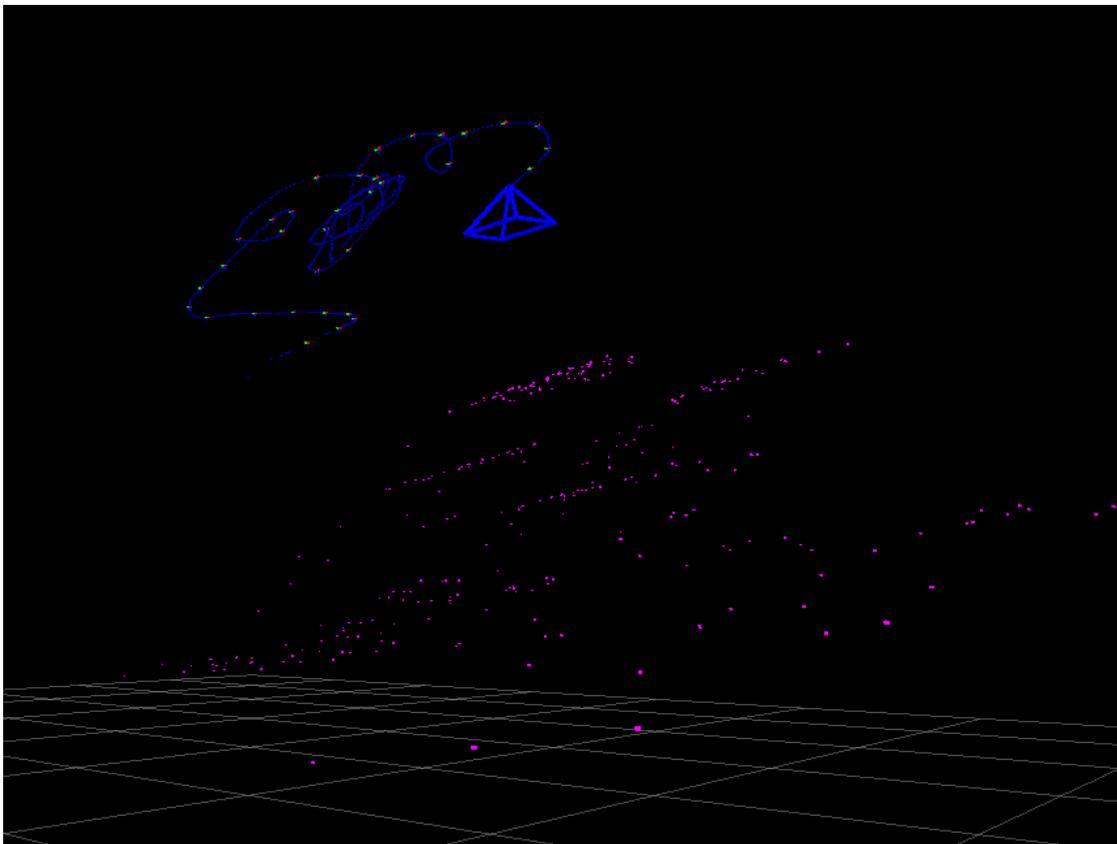
**Figura 40.** Instante donde se selecciona el segundo *frame*. A la izquierda, vista desde abajo. Como el origen se aprecia tímidamente, en la derecha se muestra el resultado de hacer zoom.

Como curiosidad, aprovechando las herramientas del rviz, hemos podido trazar la distancia que tiene la trayectoria no reconstruida: las coordenadas de la posición en la que se selecciona el segundo fotograma serán la distancia entre ambos puntos, al ser el primer frame el origen (0,0,0). Se resalta recuadrado en rojo en la siguiente figura:



**Figura 41:** Uso de la herramienta "Measure" del rviz para medir la posición del segundo frame.

A partir de este momento –la toma del segundo *frame*–, en el rviz ya visualizaremos los resultados de manera estable. Es decir, la ventana de la grabación presentará las imágenes con las esquinas detectadas señaladas (como en la **figura 38**), mientras que en la ventana de la reconstrucción apreciaremos tanto la trayectoria como el mapa. Valga como ejemplo de ello la **figura 42**.



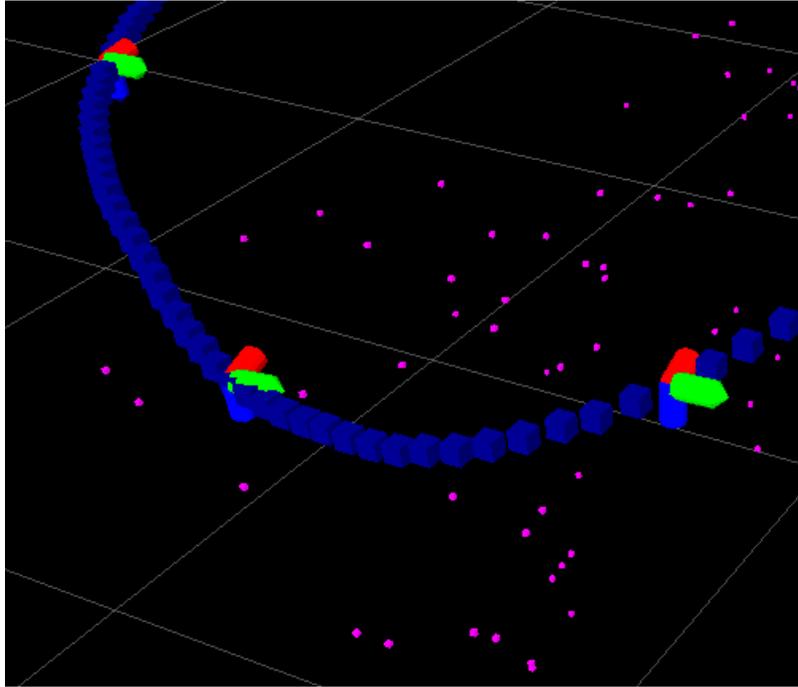
**Figura 42.** Reconstrucción de mapa y trayectoria para un instante elegido al azar.

Los puntos de color rosa en esta ventana conforman el resultado del mapeado, mientras que los puntos azules denotan la reconstrucción de la trayectoria. Nótese que en la ruta también podemos ver las posiciones en las que se seleccionan los *keyframes*: aparecen destacadas como ejes tricolores. En la **figura 43** ampliamos la imagen para mostrarlos más detalladamente.

Existe un interesante resultado a comentar referente a ellos. Observando la simulación de corrido durante algunos instantes podemos darnos cuenta de que cada vez que se inserta un nuevo fotograma clave aparecen nuevos puntos rosas en el mapa al tiempo que desaparecen otros. Este hecho sostiene la idea ya expuesta de que cuando se inserta un nuevo *keyframe* en el mapa, se elimina el más alejado de la posición de la cámara en dicho momento (manteniendo así un número fijo de *keyframes* en el mapa). Lamentablemente no podemos incluir pruebas que apoyen gráficamente este resultado, ya que no se apreciaría mediante capturas de instantes individuales, sino que requiere un visionado continuo<sup>16</sup>.

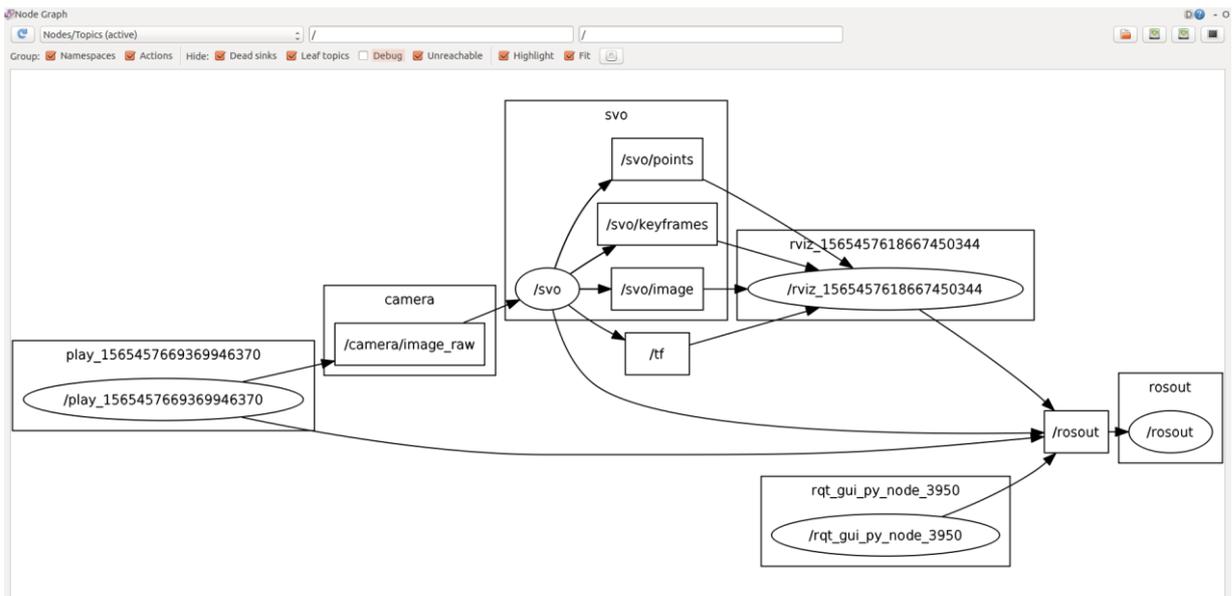
---

<sup>16</sup> Trataremos de demostrar este hecho más adelante mediante una vía alternativa.



**Figura 43.** Zoom en los ejes de la ruta, que denotan las posiciones donde se toman los keyframes. A diferencia del resto de puntos, se almacenan con orientación.

Abordemos ahora SVO en lo relativo a su implementación en ROS. Una útil herramienta con la que extraer conclusiones es *rqt\_graph*, que nos muestra cómo queda el grafo de computación en nuestro sistema. Para iniciarla sólo tenemos que introducir la orden *roslaunch rqt\_graph* por línea de comandos en un terminal independiente (nuevo proceso):



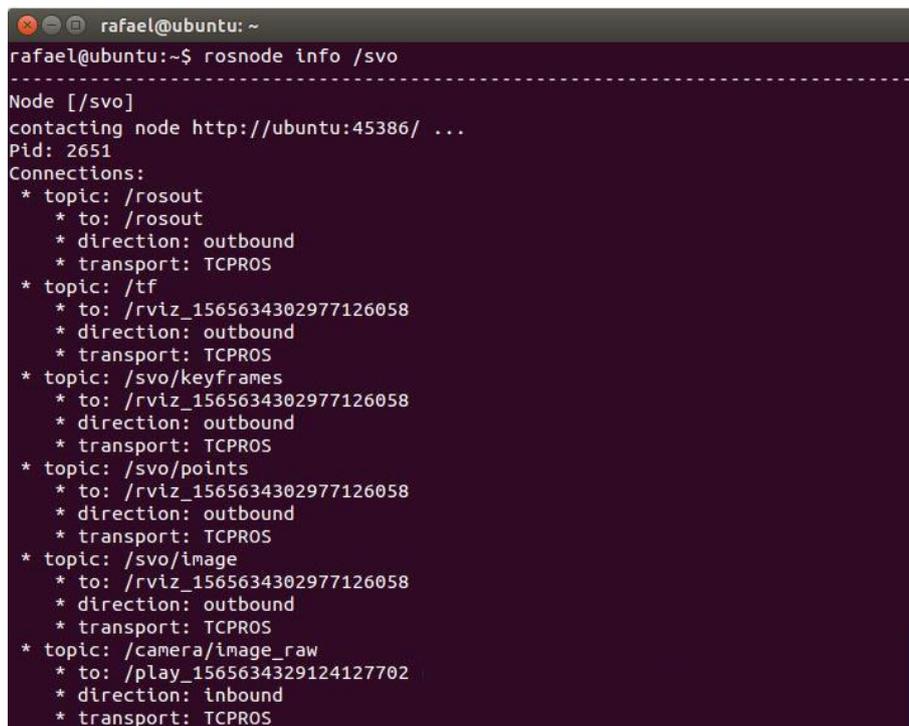
**Figura 44.** Grafo de computación ROS en nuestro sistema: Nodos y Topics activos.

Interpretemos este esquema, en el cual los nodos vienen representados por elipses y los topics mediante rectángulos, y el flujo de trabajo va de izquierda a derecha:

A la izquierda del todo aparece el nodo “/play\_156(...)370”, que fue lanzado con la orden *roslaunch play* y es el encargado de la publicación de las imágenes almacenadas en el dataset. Dicha publicación se lleva a cabo en el topic “/camera/image\_raw” que, como su propio nombre indica, contiene las imágenes en crudo. A continuación, el nodo “/svo” se suscribe a este topic con vistas a obtener las imágenes con las que ejecutar el algoritmo.

A su vez, el nodo SVO se comunica con el nodo encargado de lanzar el rviz mediante la publicación de mensajes en tres topics diferentes que dictan la información que se visualizará en el rviz.

Podemos cerciorarnos de que un nodo cuenta con las publicaciones y suscripciones que la **figura 44** ilustra haciendo uso de la herramienta *rostopic*. Introduciéndola por línea de comandos nos proporciona información sobre el nodo dado como parámetro de entrada. Por ejemplo, para el nodo de SVO:

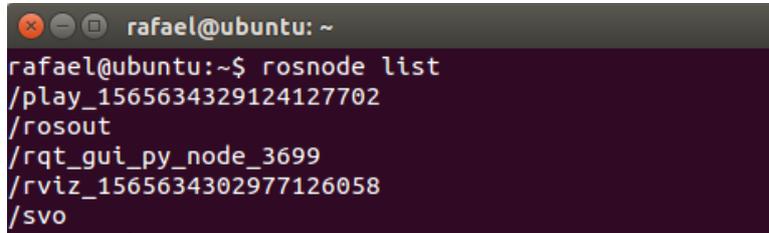


```
rafael@ubuntu:~$ rostopic info /svo
-----
Node [/svo]
contacting node http://ubuntu:45386/ ...
Pid: 2651
Connections:
* topic: /rosout
  * to: /rosout
  * direction: outbound
  * transport: TCPROS
* topic: /tf
  * to: /rviz_1565634302977126058
  * direction: outbound
  * transport: TCPROS
* topic: /svo/keyframes
  * to: /rviz_1565634302977126058
  * direction: outbound
  * transport: TCPROS
* topic: /svo/points
  * to: /rviz_1565634302977126058
  * direction: outbound
  * transport: TCPROS
* topic: /svo/image
  * to: /rviz_1565634302977126058
  * direction: outbound
  * transport: TCPROS
* topic: /camera/image_raw
  * to: /play_1565634329124127702
  * direction: inbound
  * transport: TCPROS
```

**Figura 45.** Comando *rostopic info* para el nodo SVO.

Observamos como efectivamente este nodo está suscrito (“*direction: inbound*”) al topic “/camera/image\_raw”, que le comunica con el nodo “/play\_156(...)370”. Y nótese también que todas las conexiones se efectúan mediante el protocolo TCPROS, como comentamos en el capítulo anterior. Por otra parte, vemos que el nodo “/svo” publica (“*direction: outbound*”) en hasta cinco nodos, de los cuales tres de ellos le comunican con el nodo de rviz. De esta manera, hemos podido contrastar el contenido del grafo de computación.

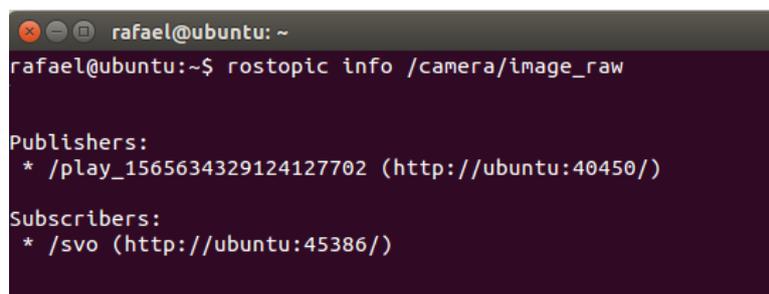
Existen otras formas de verificar el workflow que sigue el grafo de computación. Por ejemplo, *roscout* también acepta la opción *list*, que saca el listado de los nodos activos. Al ejecutarla, comprobamos que esta información concuerda con el número de nodos (elipses) en la **figura 44**:



```
rafael@ubuntu: ~
rafael@ubuntu:~$ roscout list
/play_1565634329124127702
/rosout
/rqt_gui_py_node_3699
/rviz_1565634302977126058
/svo
```

**Figura 46.** Resultado de *roscout list*: Listado de nodos activos.

Y también podemos comprobar de manera individual para cada topic los nodos que publican en él, así como los nodos suscriptores, gracias a la herramienta *rostopic*. Su opción *info* es de otra forma de ratificar, de manera local, las conexiones representadas en el grafo de computación. Por ejemplo, para el topic “camera/image\_raw”:



```
rafael@ubuntu: ~
rafael@ubuntu:~$ rostopic info /camera/image_raw

Publishers:
* /play_1565634329124127702 (http://ubuntu:40450/)

Subscribers:
* /svo (http://ubuntu:45386/)
```

**Figura 47.** *Rostopic info* en el nodo “/camera/image\_raw”.

Las herramientas de ROS nos permiten ir un paso más allá en el análisis, pudiendo incluso acceder a los mensajes que los nodos publican en los correspondientes topics. Vamos a estudiar su contenido en busca de más resultados interesantes. Para ello haremos uso nuevamente del comando *rostopic*, en esta ocasión con la opción *echo*, la cual imprime en pantalla los mensajes publicados en el topic dado como parámetro de entrada.

En primer lugar, vamos a inspeccionar el contenido de los topics que contienen información de las imágenes, estos son “/camera/image\_raw” y “/svo/image”. La diferencia entre ellos es que en el primero se publican las imágenes del dataset en escala de grises, mientras que en el segundo se publican estas mismas imágenes, pero ya con las características resaltadas en verde.

La **figura 48** ilustra la diferencia entre configurar el rviz para visualizar el topic “/svo/image” (arriba) o el topic “/camera/image\_raw” (abajo)<sup>17</sup>:

<sup>17</sup> Elegir mostrar este topic en el rviz implica una ligera modificación del grafo de computación ilustrado en la **figura 44**: el nodo que ejecuta el visualizador se suscribirá a este topic en lugar de hacerlo al topic “/svo/image”.



**Figura 48.** Comparativa visual del contenido de los topics de las imágenes.

Consecuentemente, los mensajes contenidos en estos topics deben ser análogos, siendo esto lo que examinaremos con la orden *rostopic echo*. La **figura 49** muestra el resultado de ejecutar este comando tanto con el topic “/camera/image\_raw” (arriba) como con “/svo/image” (abajo). Podemos apreciar como los mensajes en uno y otro topic son de la misma naturaleza, tratándose básicamente de una ristra de valores que denotan los niveles de intensidad de los pixeles de las imágenes.

Obviamente estos valores no nos proporcionan información espacial. Sin embargo, inspeccionándolos, podemos atrevernos a deducir en torno a cuáles de ellos podrían colocarse características, aunque no sepamos con cuales se corresponderían posicionalmente en la imagen. Así pues, los puntos candidatos serán aquellos donde encontremos un salto significativo en sus valores de intensidad con respecto a los de los vecinos, asumiendo que se emplea FAST para detectar las características, y que estas se corresponderán con las direcciones de máxima variación (gradiente), dado que podemos entender bordes y esquinas como las transiciones entre dos áreas con niveles de gris cuantitativamente diferenciables.

En la **figura 50** hemos marcado un conjunto que cumple con tales especificaciones.

```

rafael@ubuntu: ~
1, 32, 37, 39, 41, 42, 44, 41, 43, 45, 45, 44, 41, 41, 43, 40, 40, 40, 39, 39, 39, 39
, 40, 42, 43, 43, 42, 44, 44, 43, 44, 45, 42, 40, 40, 38, 38, 33, 31, 31, 31, 29, 27
, 25, 26, 24, 21, 20, 21, 21, 21, 22, 22, 22, 25, 23, 23, 22, 23, 24, 24, 24, 25
, 27, 28, 27, 26, 27, 29, 29, 31, 31, 29, 32, 31, 32, 34, 34, 32, 31, 35, 31, 31, 33, 4
1, 41, 39, 37, 31, 31, 33, 32, 29, 32, 32, 31, 30, 30, 32, 33, 33, 33, 32, 31, 30, 30
, 34, 34, 32, 31, 29, 29, 32, 33, 33, 34, 34, 35, 34, 33, 33, 33, 36, 35, 34, 33, 33
, 34, 35, 35, 33, 35, 35, 35, 36, 34, 35, 34, 32, 31, 31, 31, 32, 31, 30, 25, 17, 20
, 22, 26, 29, 29, 31, 31, 32, 30, 32, 34, 35, 34, 34, 36, 35, 36, 34, 34, 33, 34, 32, 3
6, 40, 37, 40, 38, 37, 37, 37, 40, 39, 38, 35, 33, 22, 30, 40, 42, 41, 38, 35, 34, 35
, 36, 35, 33, 33, 35, 34, 32, 28, 29, 30, 28, 27, 26, 28, 26, 28, 30, 30, 34, 33, 34
, 37, 36, 31, 34, 32, 35, 33, 37, 39, 38, 34, 34, 39, 37, 34, 32, 34, 34, 34, 33, 35
, 35, 34, 33, 33, 36, 36, 35, 34, 39, 42, 40, 36, 39, 40, 39, 39, 40, 41, 35, 36, 36, 3
7, 37, 36, 38, 38, 37, 35, 36, 36, 34, 33, 35, 35, 34, 28, 31, 24, 31, 34, 34, 32, 35
, 33, 33, 31, 35, 37, 35, 35, 36, 36, 35, 35, 35, 33, 34, 32, 31, 32, 33, 32, 34, 32
, 33, 31, 32, 31, 28, 29, 28, 29, 31, 30, 31, 29, 32, 32, 33, 33, 32, 34, 32, 35
, 33, 32, 35, 30, 31, 29, 30, 30, 31, 29, 30, 28, 29, 30, 30, 32, 29, 29, 31, 31, 30, 3
1, 30, 29, 29, 29, 31, 30, 31, 30, 33, 32, 31, 30, 29, 29, 31, 33, 31, 31, 30, 30, 30
, 30, 31, 30, 31, 29, 29, 29, 27, 29, 29, 29, 30, 28, 29, 29, 28, 29, 29, 29, 29, 29
, 29, 32, 31, 29, 30, 31, 30, 31, 33, 31, 30, 29, 28, 28, 29, 28, 29, 28, 27, 27, 28
, 27, 25, 26, 26, 26, 26, 28, 26, 27, 25, 27, 24, 26, 27, 28, 27, 27, 27, 26, 26, 25, 2
4, 26, 23, 24, 24, 25, 25, 24, 24, 25, 24, 24, 23, 22, 23, 24, 24, 23, 23, 21, 21, 21
, 21, 21, 21, 21, 21, 21, 20, 21, 19, 21, 20, 19, 20, 19, 19, 20, 19, 20]
---
rafael@ubuntu:~$ rostopic echo /camera/image_raw
30, 30, 30, 31, 31, 31, 30, 30, 30, 30, 30, 29, 29, 29, 27, 27, 27, 29, 29, 2
9, 31, 31, 31, 33, 33, 33, 31, 31, 31, 32, 32, 32, 32, 32, 33, 33, 33, 30, 30
, 30, 29, 29, 29, 30, 30, 30, 28, 28, 28, 27, 27, 29, 29, 29, 28, 28, 28, 30, 3
0, 30, 29, 29, 29, 29, 29, 29, 29, 26, 26, 26, 28, 28, 28, 29, 29, 29, 29, 29
, 29, 29, 29, 29, 28, 28, 28, 27, 27, 27, 28, 28, 28, 30, 30, 30, 30, 30, 2
7, 27, 27, 29, 29, 28, 28, 28, 29, 29, 29, 27, 27, 27, 29, 29, 29, 27, 27, 27
, 26, 26, 26, 27, 27, 27, 27, 27, 28, 28, 27, 27, 27, 27, 27, 27, 27, 27, 2
7, 29, 29, 29, 28, 28, 28, 28, 28, 27, 27, 27, 26, 26, 26, 27, 27, 27, 26, 26
, 26, 27, 27, 27, 27, 27, 27, 27, 28, 28, 26, 26, 26, 26, 26, 26, 27, 2
7, 27, 26, 26, 26, 26, 26, 25, 25, 25, 25, 25, 27, 27, 26, 26, 26, 26
, 26, 26, 25, 25, 25, 26, 26, 26, 25, 25, 25, 24, 24, 24, 25, 25, 25, 24, 24, 2
4, 24, 24, 22, 22, 22, 24, 24, 24, 24, 24, 24, 26, 26, 26, 26, 26, 24, 24, 24
, 26, 26, 23, 23, 23, 24, 24, 24, 25, 25, 25, 23, 23, 23, 24, 24, 23, 23, 2
3, 23, 23, 25, 25, 25, 25, 25, 25, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25
, 25, 24, 24, 26, 26, 26, 26, 26, 28, 28, 28, 40, 40, 40, 51, 51, 51, 57, 5
7, 57, 60, 60, 60, 63, 63, 63, 61, 61, 61, 62, 62, 62, 60, 60, 60, 61, 61, 60
, 60, 60, 58, 58, 58, 58, 58, 58, 59, 59, 59, 58, 58, 58, 59, 59, 59, 56, 56, 56, 5
5, 55, 55, 55, 55, 53, 53, 53, 51, 51, 51, 47, 47, 47, 42, 42, 42, 38, 38, 38
, 33, 33, 33, 30, 30, 30, 31, 31, 30, 30, 29, 29, 29, 29, 29, 29, 29, 29, 2
9, 28, 28, 28, 28, 28, 29, 29, 29, 28, 28, 28, 27, 27, 27, 27, 27, 23, 23
, 23, 26, 26, 23, 23, 23, 24, 24, 24, 23, 23, 23, 20, 20, 20, 19, 19, 18, 1
8, 18, 20, 20, 20, 20, 20, 22, 22, 22, 22, 22, 22, 22, 22, 24, 24, 24]
---
rafael@ubuntu:~$ rostopic echo /svo/image
5, 25, 23, 24, 24, 26, 25, 25, 24, 24, 23, 24, 23, 27, 24, 25, 25, 27, 28, 26, 2
8, 25, 26, 25, 26, 26, 25, 26, 25, 25, 26, 27, 26, 25, 28, 27, 26, 25, 26, 25, 2
7, 25, 26, 27, 27, 27, 25, 26, 26, 26, 27, 28, 26, 28, 25, 26, 27, 28, 29, 2
8, 29, 28, 25, 27, 26, 26, 25, 27, 27, 28, 27, 27, 29, 27, 27, 27, 26, 26, 26, 2
6, 23, 27, 28, 26, 29, 27, 27, 29, 28, 27, 27, 28, 26, 26, 27, 29, 30, 30, 33, 3
3, 30, 29, 30, 28, 27, 28, 27, 24, 23, 23, 25, 23, 24, 28, 28, 27, 26, 24, 24, 2
4, 24, 23, 22, 21, 22, 21, 21, 21, 20, 22, 23, 23, 24, 23, 25, 28, 28, 28, 27, 2
7, 26, 30, 28, 28, 28, 29, 29, 29, 29, 28, 28, 30, 29, 26, 26, 24, 20, 20, 18, 1
7, 19, 23, 24, 25, 26, 26, 22, 19, 18, 19, 19, 19, 18, 18, 18, 18, 19, 18, 20, 2
0, 23, 25, 27, 29, 30, 31, 33, 34, 34, 37, 39, 39, 39, 41, 42, 48, 45, 40, 41, 4
0, 38, 39, 39, 38, 38, 38, 36, 36, 34, 34, 34, 34, 36, 35, 35, 31, 30, 36, 4
6, 48, 53, 67, 123, 133, 137, 139, 139, 133, 95, 71, 64, 68, 66, 70, 94, 131, 13
4, 136, 136, 130, 120, 84, 76, 75, 75, 78, 86, 114, 131, 140, 130, 122, 105, 75
, 48, 39, 36, 35, 32, 32, 31, 31, 32, 30, 30, 31, 30, 32, 32, 32, 31, 29, 30, 30
, 32, 31, 31, 30, 31, 29, 27, 28, 28, 28, 30, 28, 28, 29, 30, 27, 27, 27, 29
, 27, 27, 26, 26, 27, 29, 28, 28, 28, 27, 23, 20, 25, 28, 27, 27, 27, 26, 27, 27
, 27, 28, 28, 28, 28, 27, 28, 28, 29, 29, 30, 30, 30, 32, 32, 31, 30, 30, 29, 30
, 31, 31, 30, 30, 30, 29, 30, 28, 31, 27, 28, 28, 30, 29, 28, 29, 27, 26, 27, 28
, 26, 26, 27, 27, 25, 26, 26, 24, 22, 21, 19, 21, 21, 23, 23, 23, 24, 24, 25, 25
, 29, 27, 29, 28, 26, 24, 24, 23, 21, 27, 26, 23, 19, 18, 16, 19, 19, 17, 17, 16
, 16, 18, 20, 20, 20, 19, 19, 19, 19, 18, 19, 17, 16, 17, 16, 20, 23, 21, 20, 18
, 17, 20, 18, 19, 21, 21, 23, 23, 23, 22, 23, 21, 20, 22, 22, 26, 27, 25, 26, 27
, 26, 27, 25, 27, 27, 26, 27, 27, 29, 29, 29, 28, 29, 29, 30, 28, 27, 29, 30, 31
, 31, 30, 29, 32, 30, 30, 31, 29, 29, 31, 31, 32, 30, 33, 31, 32, 33, 34, 38,

```

Figura 49. Comparativa de los mensajes publicados en los topics de las imágenes.

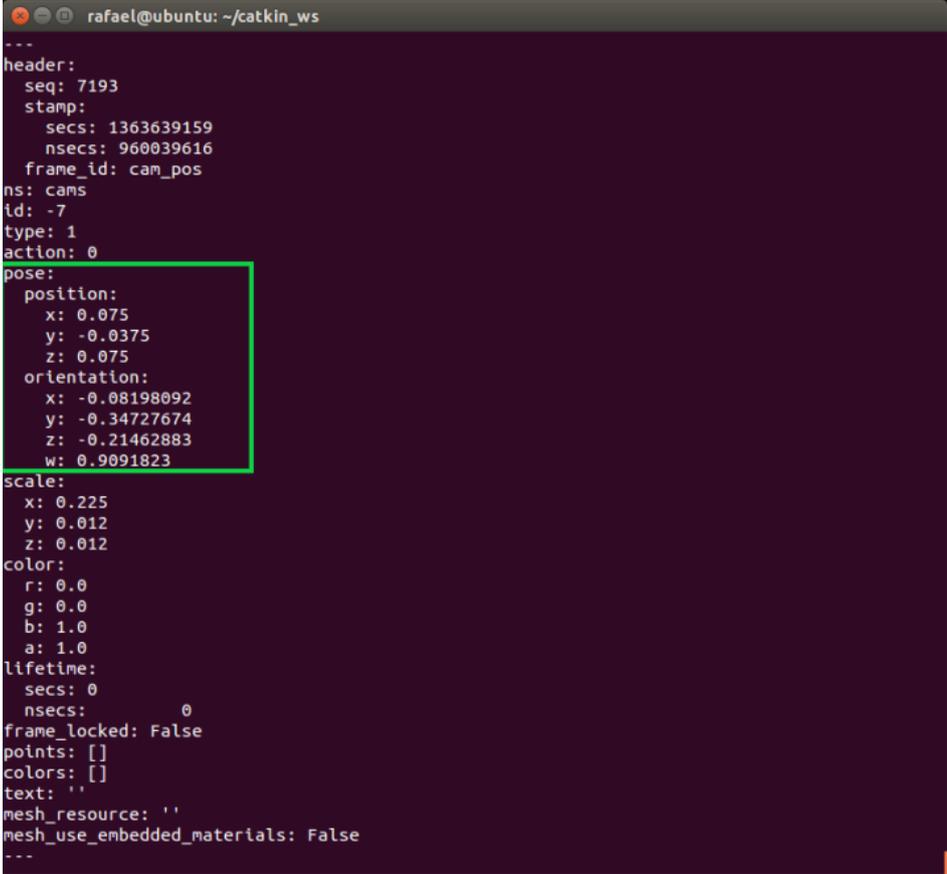
```

rafael@ubuntu: ~
5, 25, 23, 24, 24, 26, 25, 25, 24, 24, 23, 24, 23, 27, 24, 25, 25, 27, 28, 26, 2
8, 25, 26, 25, 26, 26, 25, 26, 25, 25, 26, 27, 26, 25, 28, 27, 26, 25, 26, 25, 2
7, 25, 26, 27, 27, 27, 25, 26, 26, 26, 27, 28, 26, 28, 25, 26, 27, 28, 29, 2
8, 29, 28, 25, 27, 26, 26, 25, 27, 27, 28, 27, 27, 29, 27, 27, 27, 26, 26, 26, 2
6, 23, 27, 28, 26, 29, 27, 27, 29, 28, 27, 27, 28, 26, 26, 27, 29, 30, 30, 33, 3
3, 30, 29, 30, 28, 27, 28, 27, 24, 23, 23, 25, 23, 24, 28, 28, 27, 26, 24, 24, 2
4, 24, 23, 22, 21, 22, 21, 21, 21, 20, 22, 23, 23, 24, 23, 25, 28, 28, 28, 27, 2
7, 26, 30, 28, 28, 28, 29, 29, 29, 29, 28, 28, 30, 29, 26, 26, 24, 20, 20, 18, 1
7, 19, 23, 24, 25, 26, 26, 22, 19, 18, 19, 19, 19, 18, 18, 18, 18, 19, 18, 20, 2
0, 23, 25, 27, 29, 30, 31, 33, 34, 34, 37, 39, 39, 39, 41, 42, 48, 45, 40, 41, 4
0, 38, 39, 39, 38, 38, 38, 36, 36, 34, 34, 34, 34, 36, 35, 35, 31, 30, 36, 4
6, 48, 53, 67, 123, 133, 137, 139, 139, 133, 95, 71, 64, 68, 66, 70, 94, 131, 13
4, 136, 136, 130, 120, 84, 76, 75, 75, 78, 86, 114, 131, 140, 130, 122, 105, 75
, 48, 39, 36, 35, 32, 32, 31, 31, 32, 30, 30, 31, 30, 32, 32, 32, 31, 29, 30, 30
, 32, 31, 31, 30, 31, 29, 27, 28, 28, 28, 30, 28, 28, 29, 30, 27, 27, 27, 29
, 27, 27, 26, 26, 27, 29, 28, 28, 28, 27, 23, 20, 25, 28, 27, 27, 27, 26, 27, 27
, 27, 28, 28, 28, 28, 27, 28, 28, 29, 29, 30, 30, 30, 32, 32, 31, 30, 30, 29, 30
, 31, 31, 30, 30, 30, 29, 30, 28, 31, 27, 28, 28, 30, 29, 28, 29, 27, 26, 27, 28
, 26, 26, 27, 27, 25, 26, 26, 24, 22, 21, 19, 21, 21, 23, 23, 23, 24, 24, 25, 25
, 29, 27, 29, 28, 26, 24, 24, 23, 21, 27, 26, 23, 19, 18, 16, 19, 19, 17, 17, 16
, 16, 18, 20, 20, 20, 19, 19, 19, 19, 18, 19, 17, 16, 17, 16, 20, 23, 21, 20, 18
, 17, 20, 18, 19, 21, 21, 23, 23, 23, 22, 23, 21, 20, 22, 22, 26, 27, 25, 26, 27
, 26, 27, 25, 27, 27, 26, 27, 27, 29, 29, 29, 28, 29, 29, 30, 28, 27, 29, 30, 31
, 31, 30, 29, 32, 30, 30, 31, 29, 29, 31, 31, 32, 30, 33, 31, 32, 33, 34, 38,

```

Figura 50. Búsqueda de una transición que pueda denotar una característica.

Analicemos ahora el topic “/svo/keyframes”, en el cual el nodo del algoritmo SVO publica los puntos de la trayectoria que referencian los lugares donde se insertan los fotogramas clave. Como ya comentamos anteriormente, estos puntos sí son almacenados con un valor para la orientación, a diferencia del resto de puntos de la ruta y los puntos del mapa. Si ejecutamos la orden *rostopic echo* proporcionando este topic como entrada, observamos en pantalla los mensajes en él publicados. Presentan el siguiente formato:



```
rafael@ubuntu: ~/catkin_ws
---
header:
  seq: 7193
  stamp:
    secs: 1363639159
    nsecs: 960039616
  frame_id: cam_pos
ns: cams
id: -7
type: 1
action: 0
pose:
  position:
    x: 0.075
    y: -0.0375
    z: 0.075
  orientation:
    x: -0.08198092
    y: -0.34727674
    z: -0.21462883
    w: 0.9091823
scale:
  x: 0.225
  y: 0.012
  z: 0.012
color:
  r: 0.0
  g: 0.0
  b: 1.0
  a: 1.0
lifetime:
  secs: 0
  nsecs: 0
frame_locked: False
points: []
colors: []
text: ''
mesh_resource: ''
mesh_use_embedded_materials: False
---
```

**Figura 51.** Datos de los keyframes. En verde se recuadran los valores que los ubican.

La figura anterior nos sirve para notar cómo ROS almacena la orientación para los keyframes, que viene dada por medio de cuatro componentes (x, y, z, w). Se trata de una representación especial de la orientación llamada “*quaternion*”, una abstracción matemática que difiere de la forma en que la entendemos los humanos, basándonos en los ángulos de Euler (roll, pitch, yaw). La conversión se realiza de forma transparente al usuario.

Por su parte, en el topic “/svo/points” son publicados los puntos mapeados del entorno, así como los puntos que reconstruyen la trayectoria de las posiciones por las que pasa la cámara. Pese a que desde el propio rviz se indique esto (como se ilustra en la **figura 52**), demostrar el hecho de que en este topic se publiquen ambos ‘tipos’ de puntos no es algo trivial.



Figura 52. Diferenciación entre puntos del mapa (*pts*) y de la ruta (*trajectory*).

Y es que si tratamos de observar los mensajes aquí publicados haciendo nuevamente uso de *rostopic echo*, puede darse la situación de que pensemos que estamos ante una contradicción entre lo que nos indica el rviz –**figura 52**– y el resultado que arroja este comando –**figura 53**–. El motivo es que las últimas salidas de esta orden para este topic pueden estar referidas, por ejemplo, sólo a puntos del trayecto, sin que encontremos a simple vista información para puntos los del mapa, lo cual podría llevarnos a creer que en este topic solo se publican puntos de la trayectoria:

```

rafael@ubuntu: ~/catkin_ws
---
header:
  seq: 5295
  stamp:
    secs: 1566664185
    nsecs: 51198915
  frame_id: /world
ns: trajectory
id: 1023
type: 1
action: 0
pose:
  position:
    x: 1.39957611914
    y: -0.0266905642737
    z: 0.625026692214
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 0.0
scale:
  x: 0.006
  y: 0.006
  z: 0.006
color:
  r: 0.0
  g: 0.0
  b: 0.5
  a: 1.0
lifetime:
  secs: 0
  nsecs: 0
frame_locked: False
points: []
colors: []
text: ''
mesh_resource: ''
mesh_use_embedded_materials: False
---
```

Figura 53. Datos del mensaje para el topic “/svo/points”. La marca de color verde nos indica que esta última salida –al igual que las anteriores– viene referida para puntos de la trayectoria.

Haber discurrido en esta errónea idea supuso un considerable consumo de tiempo hasta dar con la solución. Para ello es necesario realizar un análisis aún más pormenorizado de la situación, hasta el punto de estudiar el formato del mensaje. Podemos acceder a él desde la web de ROS[42], o bien con la orden `rosmmsg show visualization_msg/Marker`. Tiene el siguiente aspecto:

```

uint8 ARROW=0
uint8 CUBE=1
uint8 SPHERE=2
uint8 CYLINDER=3
uint8 LINE_STRIP=4
uint8 LINE_LIST=5
uint8 CUBE_LIST=6
uint8 SPHERE_LIST=7
uint8 POINTS=8
uint8 TEXT_VIEW_FACING=9
uint8 MESH_RESOURCE=10
uint8 TRIANGLE_LIST=11

uint8 ADD=0
uint8 MODIFY=1
uint8 DELETE=2
uint8 DELETEALL=3

Header header
string ns
int32 id
int32 type
int32 action
geometry_msgs/Pose pose
geometry_msgs/Vector3 scale
std_msgs/ColorRGBA color
duration lifetime
bool frame_locked

# header for time/frame information
# Namespace to place this object in... used in conjunction with id to create a unique name for the object
# object ID useful in conjunction with the namespace for manipulating and deleting the object later
# Type of object
# 0 add/modify an object, 1 (deprecated), 2 deletes an object, 3 deletes all objects
# Pose of the object
# Scale of the object 1,1,1 means default (usually 1 meter square)
# Color [0.0-1.0]
# How long the object should last before being automatically deleted. 0 means forever
# If this marker should be frame-locked, i.e. retransformed into its frame every timestep

#Only used if the type specified has some use for them (eg. POINTS, LINE_STRIP, ...)
geometry_msgs/Point[] points
#Only used if the type specified has some use for them (eg. POINTS, LINE_STRIP, ...)
#number of colors must either be 0 or equal to the number of points
#NOTE: alpha is not yet used
std_msgs/ColorRGBA[] colors

# NOTE: only used for text markers
string text

# NOTE: only used for MESH_RESOURCE markers
string mesh_resource
bool mesh_use_embedded_materials

```

**Figura 54.** Formato de `visualization_msg/Marker.msg`, mensaje que soporta los datos publicados en los topics `“/svo/points”` y `“/svo/keyframes”`. Puede comprobarse que presenta los mismos campos que la información impresa en los terminales de las **figuras 51** y **53**.

La clave aquí para distinguir entre puntos del mapa y puntos de la trayectoria son los parámetros `ns` (*namespace*) e `id` de la figura anterior, que usados combinadamente identifican cada elemento individual (*Marker*) publicado en el rviz. Sabiendo esto, si mostramos ahora a qué *namespace* pertenecen los puntos podremos verificar que están publicándose tanto unos como otros.

Para ello ejecutamos la orden `rostopic echo /svo/points/ns`, cuyo resultado se adjunta en la **figura 55**, que nos demuestra que los puntos de ambos *namespaces* `–pts` y `–trajectory–` se están insertando correctamente, sólo que con diferente frecuencia de publicación.

```

rafael@ubuntu: ~/catkin_ws
---
pts
---
trajectory
---
trajectory
---
trajectory
---
trajectory
---
trajectory

```

**Figura 55.** Verificación de la publicación de puntos de mapa y trayectoria.

Con la orden anterior hemos conseguido especificar que deseamos inspeccionar información a nivel de campos concretos del mensaje. Sin duda se trata de una potente posibilidad, que podemos utilizar para concluir el testeo de SVO contrastando alguna otra idea subyacente.

Por ejemplo, en la **página 48** dejamos pendiente de demostración la desaparición de puntos en el mapa a consecuencia de la eliminación del keyframe más alejado cuando se inserta uno nuevo. Este resultado no podía distinguirse sin ver la simulación ininterrumpidamente durante algunos instantes, sin embargo, inspeccionando el parámetro *action* del mensaje (reparar **figura 54**) dentro del topic “/svo/keyframes”, ahora estamos en condiciones de demostrarlo.

```

rafael@ubuntu: ~/catkin_ws
---
0
---
0
---
0
---
0
---
0
---
0
---
2
---
2
---
2
---
2
---
2
---
2
---
2

```

**Figura 56.** Ejecución de *rostopic echo /svo/keyframes/action*.

Como podemos apreciar, los valores para este campo son 0 y 2, que como vimos en el cuerpo del mensaje (reparar comentario adyacente al campo *action*) se corresponden con las acciones para añadir y eliminar elementos, respectivamente. Así, queda demostrado que existen puntos que acaban siendo borrados, tal y como afirmábamos que sucedía para mantener un número fijo de keyframes en el mapa.

Si además queremos confirmar que los puntos que se eliminan efectivamente son los del mapa, podemos buscar entre los resultados de ejecutar *rostopic echo /svo/points* que los puntos con el parámetro *action* igual a 2 además tengan como *ns* el namespace *pts*, lo cual verificaría esta hipótesis:

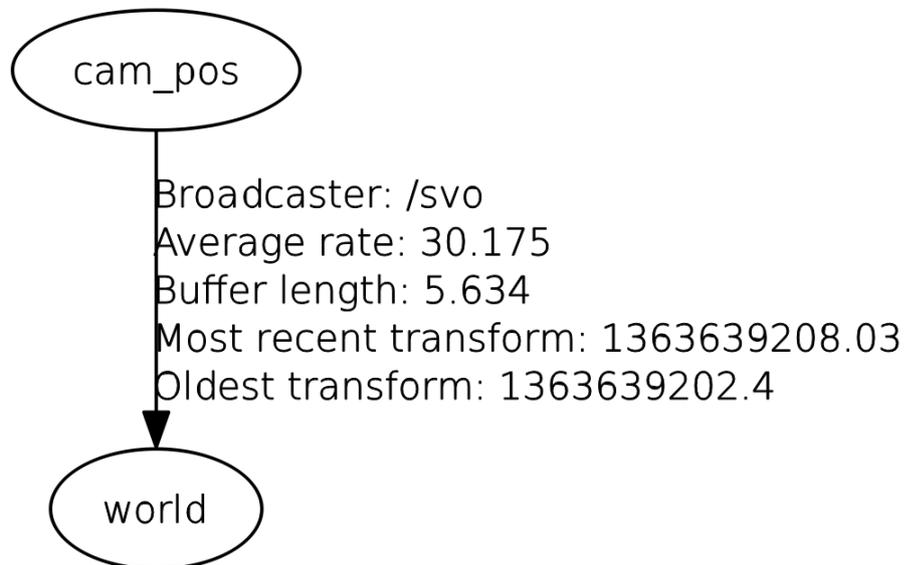
<pre>header:   seq: 10260   stamp:     secs: 1566669078     nsecs: 596217874   frame_id: /world ns: pts id: 5026 type: 1 action: 2</pre>	<pre>header:   seq: 10258   stamp:     secs: 1566669078     nsecs: 596214989   frame_id: /world ns: pts id: 5060 type: 1 action: 2</pre>	<pre>header:   seq: 10257   stamp:     secs: 1566669078     nsecs: 596213625   frame_id: /world ns: pts id: 4997 type: 1 action: 2</pre>
--	--	--

**Figura 57.** Diferentes ejemplos de eliminación (*action=2*) de puntos del mapa (*ns='pts'*).

A lo largo de este apartado hemos venido analizando exhaustivamente la técnica SVO mediante su simulación en el entorno ROS, aportando numerosas pruebas de distinta índole que asientan y complementan su explicación. Por consiguiente, podemos dar por concluido su estudio, no sin antes comentar alguna pincelada final sobre tres elementos no citados del grafo de computación que fue mostrado en la **figura 44**, para así cerrar apartado sin dejar ningún fleco suelto:

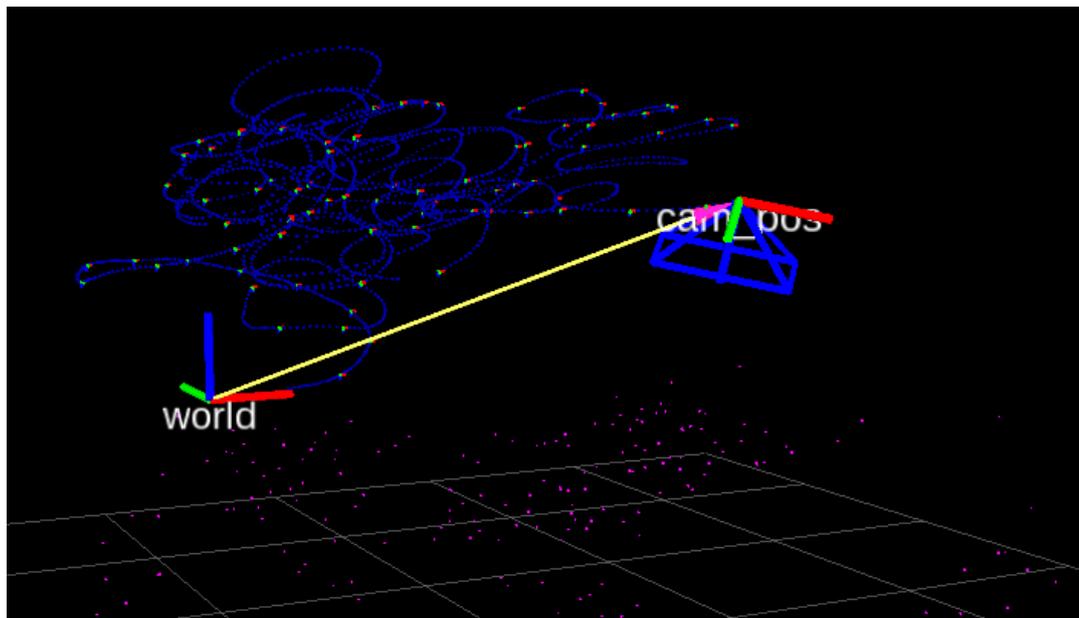
- En la parte inferior derecha de dicha figura, se mostraba el nodo *rq\_gui\_node*, pasado por alto en su momento. No procede profundizar mucho en él puesto que no es más que el proceso encargado de ejecutar el propio grafo, que como está activo para su visionado, aparece representado.
- Observamos también el elemento *tf* entre los nodos de *svo* y *rviz*. Se trata de un *topic* existente por defecto que sirve para efectuar un seguimiento de *n* marcos de coordenadas 3D a lo largo del tiempo, manteniendo la relación entre ellos en el tiempo en una estructura de árbol almacenada en un *buffer*[43]. *Tf* es un paquete muy útil en sistemas robóticos grandes, que por lo general contienen múltiples marcos de coordenadas. Por ejemplo, para un robot articulado se tendría un marco de referencia mundial, otro para la cabeza, uno para cada extremidad, etc. Gracias a *tf* se puede realizar el seguimiento en el tiempo de todos estos marcos.

En nuestro sistema tenemos dos marcos de coordenadas 3D, uno de ellos fijo (anclado al origen o punto de partida) que sirve como referencia mundial, y el otro, que cambia su posición con el tiempo respecto al mundo, para la posición de la cámara. La herramienta *rqt\_tf\_tree* nos suministra la relación entre los diferentes marcos de coordenadas, que en nuestra aplicación queda reducida a la siguiente estructura:



**Figura 58.** Relación entre los marcos de coordenadas presentes en nuestra aplicación: *cam\_pos* y *world*, para cámara y referencia global, respectivamente.

Además, es posible configurar el rviz para su mostrar la evolución de la posición de la cámara acoplada a la aeronave con respecto a la referencia fijada:



**Figura 59.** Visualización de los ejes para la cámara y referencia mundial, y la transformación entre ellos, que va cambiando a lo largo del tiempo.

- Para finalizar, hablemos sobre *rosout*, el nodo con el que el resto de nodos de nuestro grafo de computación se comunican de manera directa a través del topic homónimo. El motivo es que *rosout* hace las veces de salida estándar y salida de errores del sistema, es decir, un proceso genérico usado para presentar mensajes en los terminales[44]. Este nodo siempre se lanza por defecto con la ejecución de *roscore*.

### 5.3- Iniciación a pruebas adicionales.

La finalidad de este apartado es conducir al lector hacia los pasos que sería necesarios dar para poder lanzar la simulación de una grabación propia. Exploramos esta posibilidad pese a que no hemos llegado implementarla, dado que SVO requiere tener calibrada la cámara a emplear en un modelo específico denominado *ATAN* que se configura en un fichero de extensión “.yaml”, y por cuestiones de alcance y tiempo no hemos abordado esta temática. También el modelo *Pinhole* es soportado, aunque el recomendado es el anterior por ofrecer mayor velocidad.

Una vez calibrada, deberíamos guardar nuestro fichero de calibración, llamado por ejemplo “calibración\_propia.yaml”, en una ruta accesible para referenciarlo fácilmente. Lo ideal sería hacerlo en el directorio “svo\_ros/params/” dentro de nuestro espacio de trabajo, donde ya se encuentran alojados otros ficheros de esta naturaleza.

Acto seguido se debe modificar el fichero que lanza el nodo de SVO, que recordemos se ejecuta mediante la orden *roslaunch*. La corrección a realizar es cambiar el enlace al fichero de calibración, incorporando el nuestro. En la siguiente figura se recuadra en rojo dónde deberíamos incluir nuestro fichero “calibración\_propia.yaml”:



```
17 lines (10 sloc) | 522 Bytes
Raw Blame History
1 <launch>
2
3 <node pkg="svo_ros" type="vo" name="svo" clear_params="true" output="screen">
4
5 <!-- Camera topic to subscribe to -->
6 <param name="cam_topic" value="/camera/image_raw" type="str" />
7
8 <!-- Camera calibration file -->
9 <rosparam file="$(find svo_ros)/param/camera_atan.yaml" />
10
11 <!-- Default parameter settings: choose between vo_fast and vo_accurate -->
12 <rosparam file="$(find svo_ros)/param/vo_fast.yaml" />
13
14 </node>
15
16 </launch>
```

Figura 60. Contenido del fichero “.launch”.

Nótese también que hemos resaltado en verde el topic al que se suscribe el nodo svo dado que este parámetro también debe ser modificado, colocando en su lugar el topic pertinente: si hubiéramos optado por transmitir el recorrido seguido por la cámara a través de una grabación guardada con *rosbag record*, deberíamos colocar ahí el topic

donde el bag publica el contenido de las imágenes, para posteriormente iniciar la simulación con *rosvbag play* tal y como venimos haciendo hasta ahora. Si por el contrario optásemos por una transmisión en vivo, habría que sustituirlo por el topic donde publique el nodo encargado de ejecutar el proceso de la cámara.

En ambos casos la inicialización de SVO ya no se llevaría a cabo lanzando el fichero *test\_rig3.launch*, si no el de la figura anterior (*live.launch*) ejecutando *roslaunch svo\_ros live.launch*.

# 6. CONCLUSIONES Y LÍNEAS FUTURAS

---

Trazar la línea argumental de este estudio no supuso una decisión inmediata debido a que, dada la cantidad de información existente relativa a la navegación autónoma basada en visión monocular, existía el riesgo de acabar exponiendo un conglomerado de ideas sin mucha conexión entre sí.

Por tanto, en aras de cumplir los objetivos propuestos al inicio, hemos basado la estructura del presente trabajo en un recorrido a través de la odometría visual, partiendo de lo general (conceptos, comparaciones, clasificaciones...) hasta llegar a lo particular, focalizando el análisis en la técnica SVO.

Tras su elaboración podemos arrojar algunas conclusiones en forma de claves principales de esta técnica, que podrían servir como resumen general para alguien que nunca hubiera oído hablar de ella. Las podemos sintetizar en las siguientes sentencias:

- Se trata de un sistema autónomo basado puramente en visión, sin uso de GPS o teleoperación.
- A diferencia de la mayoría de técnicas del estado del arte, no requiere detectar cierres de bucle o *bundle adjustment*.
- Resultados con menos *outliers* gracias al uso de filtros de profundidad en el bloque del mapeado, ya que cada filtro se somete a numerosas mediciones hasta alcanzar la convergencia.
- El enfoque semi-directo evita el proceso de extracción de características en todo fotograma, derivando en una mayor eficiencia computacional. Destaca también por su robustez, permitiendo el seguimiento de características débiles o no tan destacadas.
- Desacoplar la estimación del movimiento y el mapeado en bloques separados repercute en un mejor rendimiento, liberando a este último de la restricción de actuar en tiempo real, de manera que la inserción de nuevos puntos en el mapa esté desencadenada por la toma de un nuevo fotograma clave en vez de hacerlo en cada frame.

Estas ventajas, unidas al particular enfoque semi-directo, hicieron de es SVO una técnica más que atractiva para su estudio y fueron el detonante que motivaron su elección por delante del resto de técnicas presentadas en la parte final del capítulo 2.

Por otra parte, al abordar su análisis experimental, hemos podido comprobar en primera persona las bondades del entorno ROS para el desarrollo de software robótico. Su estructura ha sido especialmente ideada para esta empresa y es de extrañar la técnica

actual que no disponga de implementación para esta plataforma. En vista de su cada vez mayor número de usuarios y del potencial de las herramientas que ofrece, podemos atrevernos a vaticinar que en los próximos años gozará aún de mayor peso y popularidad.

Y aprovechando que hablamos de desempeños venideros, volvemos a la odometría visual para cerrar este estudio preguntándonos cuáles serán las líneas y trabajos futuros dominantes:

Los sistemas de visión monoculares han sido de gran interés hasta el momento debido a su tamaño, bajo coste y configuración software. No obstante, esta línea de investigación se encuentra en una continua búsqueda de progreso y a las anteriores ventajas trata de añadirle mejoras, como una mayor precisión al determinar la posición del robot. Así, todo apunta a que la tendencia preponderante en el futuro más inmediato consistirá en la fusión del sensor visual con una unidad de medición inercial (IMU). De hecho, se trata de una posibilidad ya explorada y que está cobrando fuerza en los últimos años. Valga como ejemplo el ya citado “SVO 2.0”, que permitía la asociación del SVO original con una IMU, o el estudio “VINS-Mono” elaborado en 2018 por Quin et. al [45] en la Universidad de Ciencia y Tecnología de Hong-Kong, que propone un sistema de odometría monocular visual-inercial (VIO) de bajo coste que puede valernos como resumen de las contribuciones que aportan estos enfoques:

- Alto rendimiento en tiempo real para navegación de micro aeronaves.
- Detección de bucles y localización.
- Robustez y versatilidad, actuando en escenarios de pequeña y media escala.
- Optimización de la posición de 4 DoF (*degrees of freedom*/grados de libertad) con vistas a asegurar la consistencia.
- Por lo general, implementación *open source* compatible con ROS.

La metodología seguida para llevar a cabo la optimización nos da pie a introducir las dos maneras que vienen dándose en sistemas VIO para tratar con las medidas visuales e inerciales adquiridas. La forma más simple es a través de la fusión de sensores mediante un “acoplamiento flexible”, donde la IMU se trata como un módulo independiente para auxiliar a las estimaciones de la posición estrictamente visuales.

Paralelamente también se están desarrollando algoritmos “estrechamente acoplados”, como el anteriormente citado (VINS-Mono), donde las mediciones de la cámara y la IMU se optimizan conjuntamente.

# REFERENCIAS

---

- [1] T. Lemaire, C. Berger, I. K. Jung, and S. Lacroix, "Vision-based SLAM: Stereo and monocular approaches," *Int. J. Comput. Vis.*, vol. 74, no. 3, pp. 343–364, 2007.
- [2] J. Solà, A. Monin, and M. Devy, "BiCamSLAM: Two times mono is more than stereo," *Proc. - IEEE Int. Conf. Robot. Autom.*, pp. 4795–4800, 2007.
- [3] D. Scaramuzza and F. Fraundorfer, "Tutorial: Visual odometry," *IEEE Robot. Autom. Mag.*, vol. 18, no. 4, pp. 80–92, 2011.
- [4] M. Bailey and H. Durrant-whyte, "Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms" *Robotics Automation Magazine IEEE*, vol. 13, no. 2, pp. 99–110, 2006.
- [5] C. Cadena *et al.*, "Past , Present , and Future of Simultaneous Localization And Mapping : Towards the Robust-Perception Age," vol. 32, no. 6, pp. 1–27, 2015.
- [6] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, "On-Manifold Preintegration for Real-Time," *IEEE Trans. Robot.*, vol. 33, no. 1, pp. 1-21, 2017.
- [7] M. Dissanayake, P. Newman, S. Clarke, H. Durrant-whyte , "A solution to the simultaneous localisation and map building problem," *IEEE Trans. Robot. Autom.*, vol. 17, no. 3, pp. 229–241, 2001.
- [8] T. Bailey, J. Nieto, J. Guivant, M. Stevens, and E. Nebot, "Consistency of the EKF-SLAM algorithm," *IEEE Int. Conf. Intell. Robot. Syst.*, no. May 2014, pp. 3562–3568, 2006.
- [9] A. J. Davison and D. W. Murray, "Simultaneous localization and map-building using active vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 865–880, 2002.
- [10] D. Scaramuzza and F. Fraundorfer, "Visual Odometry Part II," *IEEE Robot. Autom. Mag.*, vol. 19, no. 2, pp. 78–90, 2012.
- [11] C. Harris, M. Stephens, "A Combined Corner and Edge Detector", *Alvey vision conference*, vol. 15, pp. 147-151, 1988.
- [12] Shi, J., and Tomasi, C. "Good features to track", In *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 593–600, 1994.
- [13] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection" In *Proc. 9th European Conference on Computer Vision (ECCV'06)*, Graz, 2006.
- [14] D.G. Lowe, "Object recognition from local scale-invariant features", *Int. Conf. on Computer Vision*, vol.2, pp. 1150–1157, 1999.

- [15] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *European Conference on Computer Vision*, May 2006.
- [16] J. Engel, V. Koltun, and D. Cremers, “DSO Conference: Direct Sparse Odometry,” In *arXiv:1607.02555*, July 2016.
- [17] Kudan (2017), “Different Types of Visual SLAM Systems”, [Online] Disponible en: <https://www.kudan.io/post/different-types-of-visual-slam-systems>
- [18] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse, “MonoSLAM: Real-time single camera SLAM,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 1052–1067, 2007.
- [19] G. Klein, D. Murray, "Parallel tracking and mapping for small AR workspaces", *Proc. IEEE ACM International Symposium on Mixed Augmented Reality*, pp. 225-234, 2007.
- [20] J. M. M. Montiel, R. Mur-Arta, and J. D. Tardos, “ORB-SLAM : A Versatile and Accurate Monocular,” *IEEE Trans. Robot.*, vol. 31, no. 5, pp. 1147–1163, 2015.
- [21] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison, “DTAM: Dense Tracking and Mapping in Real-Time”, *ICCV (International Conference on Computer Vision)*, pages 2320-2327, 2011.
- [22] J. Engel, T. Schops, D. Cremers, “LSD-SLAM: Large-Scale Direct Monocular SLAM”, *Proc. IEEE Int. Conf. Robot. Autom.*, pp. 846-853, Jun. 2014.
- [23] C. Forster, M. Pizzoli, and D. Scaramuzza, “SVO: Fast semi-direct monocular visual odometry,” In *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.
- [24] C. Forster, M. Gassner, D. Scaramuzza, Z. Zhang, and M. Werlberger, “SVO: Semidirect Visual Odometry for Monocular and Multicamera Systems,” *IEEE Trans. Robot.*, vol. 33, no. 2, pp. 249–265, 2016.
- [25] G. Vogiatzis and C. Hernández, “Video-based, real-time multi-view stereo,” *Image Vis. Comput.*, vol. 29, no. 7, pp. 434–441, 2011.
- [26] H. Strasdat, J. M. M. Montiel, and A. J. Davison, “Visual SLAM: Why filter?” *Image and Vision Computing*, vol. 30, no. 2, pp. 65–77, 2012.
- [27] ROS.org (2018), “ROS Introduction”, [Online] Disponible en: <https://wiki.ros.org/ROS/Introduction>
- [28] ROS.org (2019), “Distributions”, [Online] Disponible en: <https://wiki.ros.org/Distributions>
- [29] ROS.org (2018), “ROS Graph Concepts: Nodes”, [Online] Disponible en: <https://wiki.ros.org/Nodes>
- [30] ROS.org (2016), “ROS Graph Concepts: Messages”, [Online] Disponible en: <http://wiki.ros.org/Messages>
- [31] ROS.org (2019), “ROS Graph Concepts: Topics”, [Online] Disponible en:

- <https://wiki.ros.org/Topics>
- [32] ROS.org (2019), "ROS Graph Concepts: Services", [Online] Disponible en: <https://wiki.ros.org/Services>
- [33] ROS.org (2018), "ROS Graph Concepts: Master", [Online] Disponible en: <https://wiki.ros.org/Master>
- [34] ROS.org (2015), "ROS Graph Concepts: Bags", [Online] Disponible en: <https://wiki.ros.org/Bags>
- [35] ROS.org (2019), "ROS Filesystem Concepts: Packages", [Online] Disponible en: <http://wiki.ros.org/Packages>
- [36] ROS.org (2019), "Ubuntu install of ROS Indigo", [Online] Disponible en: <http://wiki.ros.org/indigo/Installation/Ubuntu>
- [37] Robotics and Perception Group (Department of Informatics, University of Zurich) (2018), "*uzh-rpg/rpg\_svo: ROS Installation*" GitHub [Online], Disponible en: [https://github.com/uzh-rpg/rpg\\_svo/wiki/Installation:-ROS](https://github.com/uzh-rpg/rpg_svo/wiki/Installation:-ROS)
- [38] M. Quigley, B. Gerkey, and W. D. Smart, "Programming Robots with ROS", Northern California: Ed. O'Reilly, 2015.
- [39] Robotics and Perception Group (Department of Informatics, University of Zurich) (2019), "*uzh-rpg/rpg\_svo: Semi-direct Visual Odometry*" GitHub [Online], Disponible en: [https://github.com/uzh-rpg/rpg\\_svo](https://github.com/uzh-rpg/rpg_svo)
- [40] Francisco Martín, "Conferencia-demo: programación de robots con ROS", En HackLab Almería, Abril 2017.
- [41] Robotics and Perception Group (Department of Informatics, University of Zurich) (2019), "*uzh-rpg/rpg\_svo: Run SVO with ROS*" GitHub [Online], Disponible en [https://github.com/uzh-rpg/rpg\\_svo/wiki/Run-SVO-with-ROS](https://github.com/uzh-rpg/rpg_svo/wiki/Run-SVO-with-ROS)
- [42] ROS.org (2019), "ROS Message Types: Marker", [Online] Disponible en: [http://docs.ros.org/api/visualization\\_msgs/html/msg/Marker.html](http://docs.ros.org/api/visualization_msgs/html/msg/Marker.html)
- [43] ROS.org (2017), "ROS Geometry tf", [Online] Disponible en: <http://wiki.ros.org/tf>
- [44] Wyatt S. Newman, "A Systematic Approach to Learning Robot Programming with ROS", Florida: Ed. Taylor & Francis Group, 2018.
- [45] T. Qin, P. Li, S. Shen, "VINS-mono: A robust and versatile monocular visual-inertial state estimator", IEEE Trans. Robot., vol. 34, no. 4, pp. 1004-1020, Aug. 2018.
- [46] Apuntes de la asignatura Visión Artificial (Departamento de Ingeniería de Sistemas y Automática). 2018. GITT. ETSI Sevilla.