Trabajo Fin de Máster

Máster en Ingeniería Aeronáutica

Procesamiento de nubes de puntos para el seguimiento de líneas de alta tensión

Autor: Antonio Castro Sánchez

Tutor: José Ramiro Martínez de Dios

Cotutor: Rubén Martín Clemente

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Máster
Máster en Ingeniería Aeronáutica

Procesamiento de nubes de puntos para el seguimiento de líneas de alta tensión

Autor:
Antonio Castro Sánchez

Tutor:
José Ramiro Martínez de Dios
Catedrático de Universidad

Cotutor:
Rubén Martín Clemente
Profesor titular de Universidad

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Trabajo Fin de Máster: Procesamiento de nubes de puntos para el seguimiento de líneas de alta tensión

Autor: Antonio Castro Sánchez

Tutor: José Ramiro Martínez de Dios

Cotutor: Rubén Martín Clemente

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

Agradecimientos

A mi familia, por su apoyo incondicional, sin el cual nunca podría haber llegado hasta aquí. En especial, quiero mandar un saludo a mi sobrina, que ya tiene un añito. Por si algún día lees esto, te quiere tu tito.

A la persona que me ha hecho tan feliz estos últimos años y que siempre ha estado ahí para animarme. Gracias por ayudarme tanto, te quiero.

A mis amigos de siempre, por tantas risas y tantos momentos juntos, y a aquellos compañeros que se han hecho un hueco en mi vida.

Y, por último, a mis tutores, por darme la oportunidad de hacer mi proyecto sobre un tema que tanto me apasiona.

Antonio Castro Sánchez

Ingeniero Aeronáutico

Sevilla, 2021

Resumen

El mantenimiento preventivo de las líneas de alta tensión supone una gran parte de los costes de la red eléctrica de transporte. Para ello, deben llevarse a cabo inspecciones rutinarias, las cuales están consiguiendo automatizarse gracias al empleo de los UAV. Uno de los métodos propuestos consiste en utilizar la información del campo magnético generado por estas líneas para la navegación del dron a lo largo de la misma. Para ello, es necesario correlacionar dichas medidas con la distancia a la línea. El objetivo del proyecto consiste en procesar la información extraída a través de vuelos experimentales en los que un LiDAR nos aporta las nubes de puntos de la escena, y un magnetómetro, en conjunción con un analizador de espectro, nos aporta la densidad espectral de potencia de la señal del campo magnético. Para el procesamiento de nubes de puntos se ha desarrollado una herramienta capaz de leer un archivo *bag* de ROS y arrojar la distancia media al tendido eléctrico para cada nube de la grabación. Para lograr este objetivo, la nube debe ser sometida a una serie de algoritmos de filtrado y segmentación implementados en la librería PCL. Los resultados muestran un elevado porcentaje de éxito de segmentación de líneas en torno al 93%, y una alta correspondencia entre la distancia medida y las observaciones.

Índice

Índice de tablas

Índice de figuras

1 INTRODUCCIÓN

Durante las últimas décadas, la enorme evolución tecnológica que ha acontecido en gran parte del mundo, tanto en los países más desarrollados como en las potencias emergentes, junto al aumento de la población mundial, ha traído consigo el inevitable aumento de la demanda energética. Sólo en España, la producción nacional de energía eléctrica aumentó en un 50% desde el año 2000 hasta el año 2010, de 200 a 300 teravatios-hora (Figura 1). Además, la demanda prevista a escala internacional establece que esta seguirá aumentando en las próximas décadas (Figura 2), lo que conllevará tarde o temprano un despliegue de infraestructuras en todos los niveles: generación, transporte y distribución.

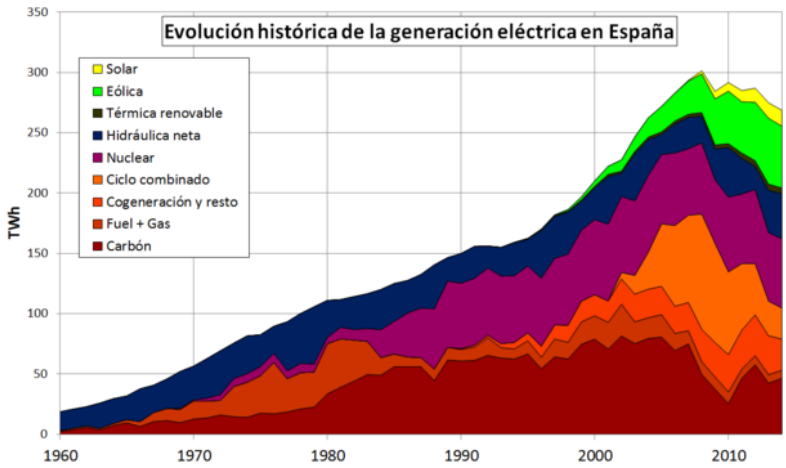


Figura 1. Evolución de la producción eléctrica en España en la última década, desglosada según su origen. Figura extraída de [1] y cortesía de la REE.

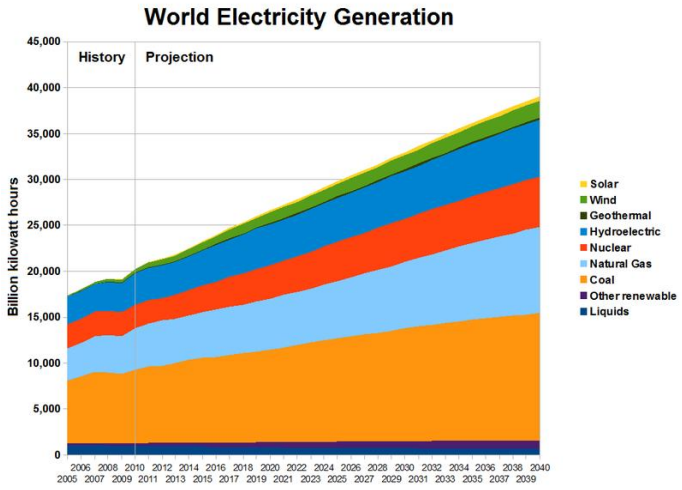


Figura 2. Previsión de la demanda de producción eléctrica a nivel internacional hasta el año 2040, desglosada según su origen y en teravatios-hora. Imagen extraída del International Energy Outlook [2].

Sin lugar a dudas, la gran ventaja de la energía eléctrica, y el motivo por el cual es la forma de energía más consumida junto a las energías fósiles, es su capacidad para transformarse en cualquier tipo de energía útil. Efectivamente, hoy en día la mayoría de los utensilios cotidianos y de las máquinas industriales funcionan mediante electricidad. Sin embargo, una importante desventaja juega en su contra: la energía eléctrica no es

barata de almacenar. Es cierto que las baterías cumplen esta función con solvencia en nuestro día a día, pero resulta inviable almacenar las enormes cantidades de energía producidas en las centrales eléctricas cada día. Esta desventaja es compensada con otro de sus puntos fuertes, que es su facilidad de distribución. No mentimos cuando decimos que la electricidad que consumimos en cada instante acaba de ser producida. Esto supone una serie de retos tecnológicos no obstante ya superados: el primero es la criticidad del sistema de transporte y distribución, y el segundo es la necesidad de ajustarse en cada momento a la demanda de energía por parte de las centrales eléctricas.

Por estos motivos la red de transporte y distribución eléctrica constituye una de las infraestructuras más importantes de cada país, pues es la encargada la suministrar a cada punto del territorio la tan demandada electricidad para la industria y para la ciudadanía.

Figura 3. Mapa de la red eléctrica española en 2016. En rojo, las líneas de 400kV, y en verde las de 220kV. Cortesía del Instituto Geográfico Nacional.

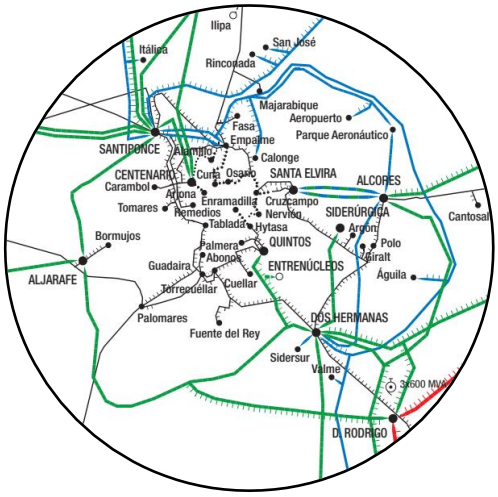


Figura 4. Detalle del tendido eléctrico de alta tensión en el área metropolitana de Sevilla. Nuevamente, en rojo, las líneas de 400kV y en verde las de 220kV. En azul, las líneas entre 150kV y 220kV, y en negro de 110kV o menos. Recorte extraído de [3].

Mientras que la red de distribución es la encargada de suministrar la energía eléctrica desde cada subestación local de distribución hasta el usuario final, la red de transporte se encarga de la transmisión desde las centrales eléctricas hasta estos puntos, generalmente a lo largo de grandes distancias.

La red de trasporte de energía eléctrica consta fundamentalmente de: subestaciones transformadoras, que elevan la corriente a altos niveles de tensión para reducir las pérdidas energéticas debido al efecto Joule y minimizar la cantidad de conductor empleado; y de líneas de transporte que se extienden cientos de kilómetros y que vienen soportadas por torres de alta tensión cada cierta distancia.

En España, el voltaje de las líneas de alta tensión varía según su categoría, atendiendo al Real Decreto 223/2008. Para el transporte se emplean líneas de transmisión correspondientes a la 1ª categoría, para tensiones normalizadas entre 110kV y 150kV; y a la categoría especial, con valores de 220kV y 400kV (ver Figura 3), empleada para la transmisión a largas distancias.

El mantenimiento preventivo es uno de los aspectos más críticos de las líneas de alta tensión, entendiendo por línea tanto el elemento conductor como las torres que lo soportan. Por un lado, la red de transmisión está expuesta continuamente a cualquier condición meteorológica que se presente: sufre de corrosión debido a las condiciones de humedad y temperatura, y de fatiga mecánica como causa del viento. Incluso es frecuente el impacto de rayos. Por el otro, la continuidad del suministro es de suma importancia en una red con escasas redundancias (debido al elevado coste que esto supondría) y en la que un corte prolongado tendría graves consecuencias económicas. Red Eléctrica de España (REE) es la empresa operadora de la red eléctrica de transporte en nuestro país, y por tanto es la encargada de garantizar su continuidad y seguridad.

Figura 5. Estado del vano de la línea 400kV Herrera-Virtus (Palencia-Burgos) tras el temporal Helena (enero 2019). Fotografía extraída de [4].

Este mantenimiento requiere de una monitorización del estado de la red, para lo cual deben llevarse a cabo inspecciones regulares. Estas inspecciones se han llevado a cabo tradicionalmente mediante helicópteros equipados con sensores ópticos, térmicos o ultravioleta, o incluso mediante aviones por SAR (*Synthetic Aperture Radar*) o ALS (*Airborne Laser Scanning*).

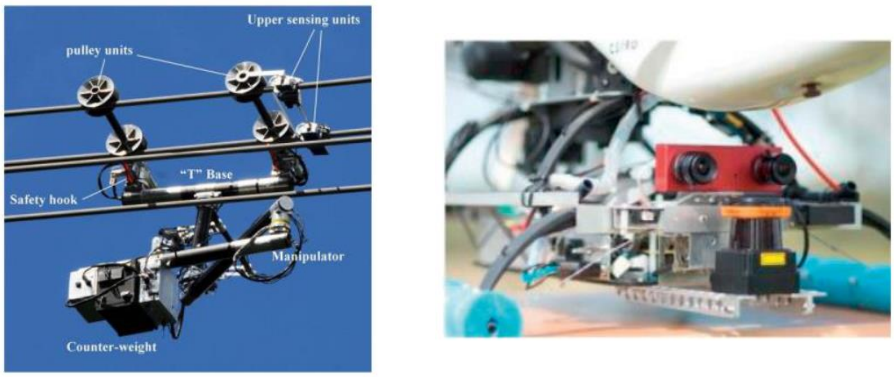
Figura 6. Helicóptero equipado con sensores estabilizados de infrarrojos y ultravioleta para detectar puntos calientes y emisiones del efecto corona.

Este tipo de soluciones suponen unos enormes costes, pueden resultar peligrosas para la tripulación (ya que en numerosas ocasiones las líneas se encuentran rodeadas de obstáculos) y además no proporcionan los mejores resultados. Sí, como se ha visto, se espera un continuo crecimiento de la carga de trabajo de mantenimiento de las líneas, está más que justificado el desarrollo de proyectos de I+D orientados a la automatización de este proceso.

Aquí entra en juego el papel de los UAV, que en la última década ha proliferado su uso en una gran variedad de aplicaciones. En la tarea que nos concierne, su potencial radica no solo en ser capaces de albergar los sensores necesarios para la inspección de la línea, sino que además son muy eficientes en la toma de datos (bajo coste y pequeño tamaño para acceder a zonas repletas de obstáculos) y pueden ser programados para volar de forma autónoma.

1.1 Estado del arte

El uso de los UAV para la inspección automática de líneas de alta tensión ha sido (y lo sigue siendo) una línea de investigación de gran interés en la última década. En [5] se hace una recopilación de los avances más importantes en lo referente a la automatización del proceso de inspección de líneas de alta tensión mediante robots. Los UAV prevalecen como el sistema que presenta un mayor potencial y flexibilidad para esta tarea, aunque otro tipo de robots como los RWR (*Rolling on Wires Robots*) se postulan como alternativa en situaciones específicas [6]. En el ARCAA (*Australian Research Centre for Aerospace Automation*) se ha trabajado en el vuelo autónomo mediante sistemas de percepción, control y navegación. Los sistemas de percepción deben ser capaces de generar un entorno 3D para el UAV, lo cual sirve como *input* para sistemas como el Sistema de detección y franqueamiento de obstáculos. Por su parte, en el Politécnico de Madrid se ha estudiado el seguimiento de líneas mediante procesamiento de imágenes, a partir del seguimiento de características (*feature tracking*) de la propia imagen.

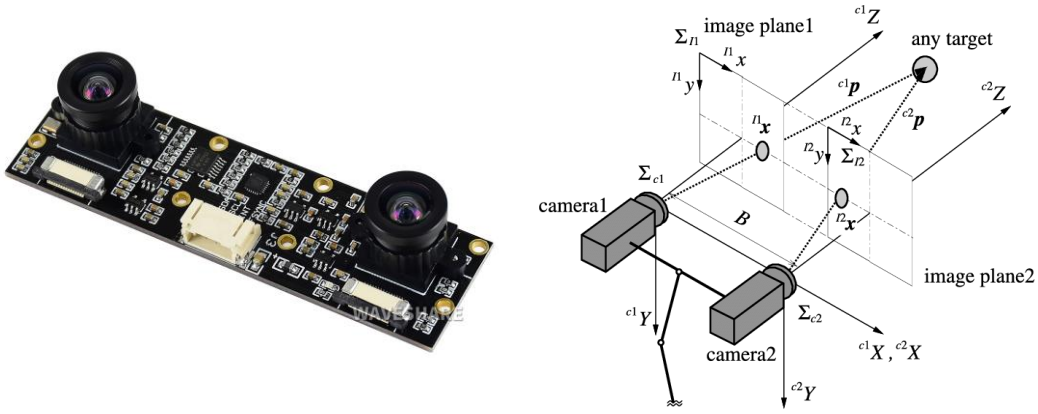


(a) (b)

Figura 7. Ejemplos de configuración de (a) RWR y de (b) UAV. Figuras extraídas de [5].

El procesamiento de imágenes ha sido una de las soluciones preferidas para el seguimiento de la línea, ya que requieren únicamente de una cámara para la toma de datos. Este método requiere sin embargo de algoritmos sofisticados para tratar estas imágenes, que se ejecutan normalmente en tres pasos: filtrado, segmentación y seguimiento. Uno de los inconvenientes principales de este método es la enorme cantidad de ruido al que se exponen estas imágenes, dificultando la tarea de aislar la línea. En [7] se estudia el uso de filtros en forma de red neuronal entrenados para preprocesar el *background* de la imagen y aislar la línea de manera más efectiva. Una vez realizado este preprocesado, se lleva a cabo la segmentación de la línea (esto es, identificar los píxeles que

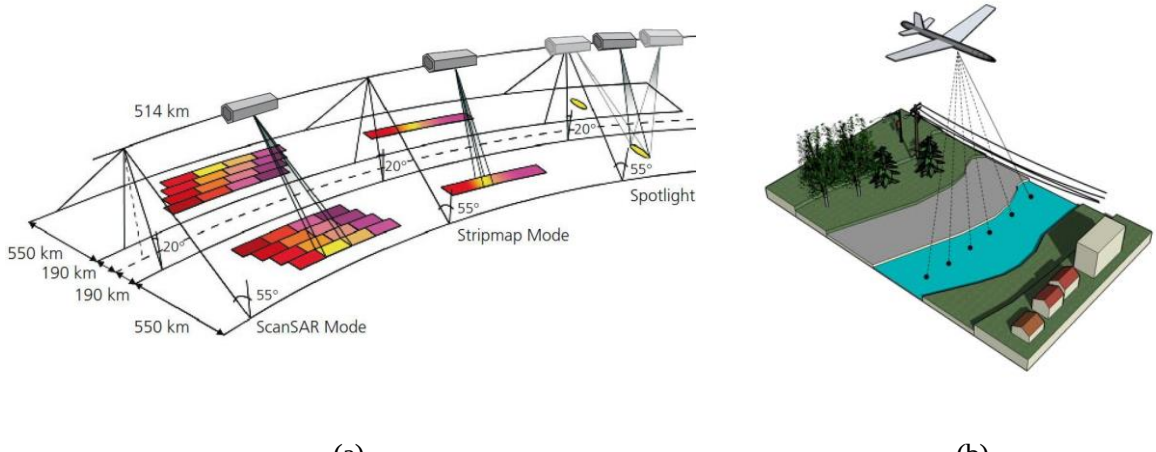
forman parte de ella), para lo cual se puede usar desde un simple detector de bordes hasta otros métodos más precisos, como el LSD (*Line Segment Detector*) [8]. Nuevos algoritmos aún más sofisticados han sido desarrollados en los últimos años [9] [10], la mayoría de ellos basados en la transformada de Hough. Este método sin embargo tiene sus limitaciones, ya que no permite calcular la distancia a la línea, para lo cual se puede recurrir a una configuración de cámaras en serie. Esto se conoce como en el ámbito de la visión por computador como visión estéreo (*stereo vision*) [11].



(a) (b)

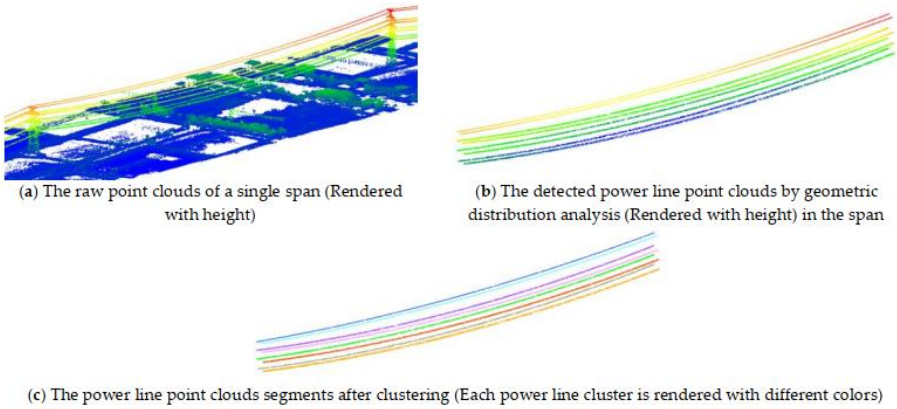
Figura 8. (a) Circuito integrado con un par de cámaras listas para visión estéreo. (b) La visión estéreo permite dar profundidad a la imagen mediante la triangulación que se observa en la figura.

En [12] se implementa un método para determinar si se cumple la distancia de seguridad en las servidumbres de la red de transmisión, es decir, la distancia del pasillo de seguridad de la línea a obstáculos, principalmente vegetación. Esto requiere de una visión global de la red, por lo cual se han venido usando, por un lado, métodos basados en imágenes como: radares SAR (*Synthetic Aperture Radar*), imágenes satélite [13] e imágenes aéreas; y por el otro, métodos que utilizan nubes de puntos: ALS (*Airborne Laser Scanning*) [14], escaneo láser terrestre y escaneo láser mediante UAV. El artículo en cuestión hace uso de un UAV equipado con un sistema LiDAR (*Laser Imaging Detection and Ranging*) y diseña un algoritmo para procesar las nubes de puntos obtenidas usando características geométricas de la propia línea: modela la curva que forma el cable conductor como una catenaria y define una forma característica de las torres. La eficacia de este método depende en gran medida de los algoritmos de filtrado y segmentación a los que se someten las nubes de puntos, al igual que ocurría con el procesamiento de imágenes. Diferentes estudios se han llevado a cabo teniendo en cuenta este aspecto: se han implementado métodos como el MRF (*Markov Random Field*) [14], agrupamiento jerárquico (*hierarchical clustering*), RANSAC (*Random Sample Consensus*) o incluso técnicas de *Deep Learning* [15].



(a) (b)

Figura 9. (a) *Synthetic Aperture Radar*: gracias al propio desplazamiento del radar instalado en el móvil se consigue una mayor apertura de la antena, lo que se conoce como apertura sintética. Esto permite obtener imágenes de mayor resolución. Imagen cortesía del DLR [16]. (b) *Airborne Laser Scanning*: una aeronave con un sistema LiDAR manda pulsos continuamente a la superficie terrestre, donde reflectan y son captados por el receptor, calculando la distancia a partir del tiempo transcurrido. Se puede llegar a tener una densidad de 1-10 puntos por m^2. Imagen extraída de [13].



(a) The raw point clouds of a single span (Rendered with height)

(b) The detected power line point clouds by geometric distribution analysis (Rendered with height) in the span

(c) The power line point clouds segments after clustering (Each power line cluster is rendered with different colors)

Figura 10. Evolución de una nube de puntos tras la aplicación de algoritmos de filtrado y de segmentación, hasta obtener *clusters* de puntos correspondientes a cada cable. Figura extraída de [12].

El sistema UAV donde se integran LiDAR, GPS, IMU, cámaras y otro tipo de sensores ha demostrado tener un enorme potencial de aplicación en una gran diversidad de sectores, especialmente para la obtención de información tridimensional del terreno [17]. Su gran ventaja frente al uso tradicional de aviones de ala fija reside en su mayor resolución espacial y temporal, y frente a los helicópteros, su menor coste operativo. Además, su uso ha cobrado más importancia gracias a los avances tecnológicos en vuelo autónomo y a su regulación.

El equipo LiDAR nos permite determinar con gran precisión distancias en la línea de alta tensión. Sin embargo, presenta un ligero inconveniente para su uso en UAV: se trata de un sistema complejo y más pesado que otro tipo de sensores. Este peso adicional reduce la autonomía del dron y, por tanto, aumenta los costes y el tiempo de inspección.

En [18] se estudia el uso de sensores de flujo magnético para determinar la distancia a la línea de alta tensión. Este concepto se ha implementado con éxito en la localización de cables subterráneos de potencia [19]. Como es sabido, las leyes del electromagnetismo establecen que una corriente eléctrica circulando por un hilo conductor genera un campo electromagnético de valor proporcional a la intensidad de corriente e inversamente proporcional a la distancia. El principal inconveniente de este método reside en el desconocimiento y variabilidad de la intensidad de corriente, ya que responde al consumo de la red eléctrica que se esté haciendo

en ese momento. Es por ello que se implementa un *array* de sensores, con el que podemos deshacernos de esta incógnita. Por otro lado, se debe prestar especial atención a la generación de ruido interno en el propio circuito debido a los fenómenos de acoplamiento capacitivo e inductivo. Los resultados muestran un error del 3% para una distancia de 1m.

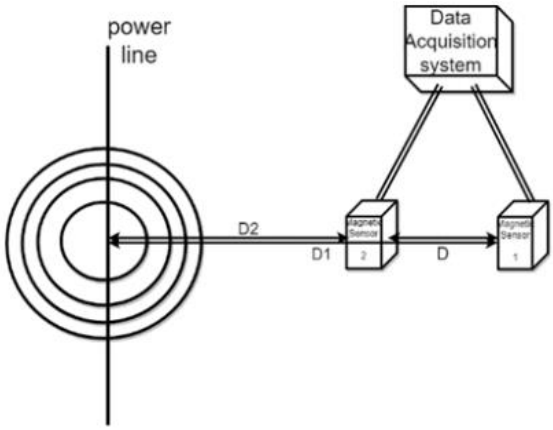


Figura 11. Array de sensores magnéticos para el cálculo de la distancia a la línea. Figura extraída de [18].

La información del campo magnético no sólo permite obtener la distancia del UAV a la línea. En [20] se implementa un método heurístico de reconstrucción magnética de la línea, el cual nos permite determinar los parámetros dinámicos de la corriente (fase y amplitud). Este método ha sido ampliamente tratado en la literatura, tanto para medidas obtenidas en tierra como para medidas tomadas en sensores a bordo de UAV, y se basa en el ajuste de un modelo inverso a partir de las medidas obtenidas y mediante algoritmos de optimización [21]. El modelo inverso describe el campo magnético generado por la línea de alta tensión en función de unos parámetros, pudiendo considerarse corrientes trifásicas. Los diferentes trabajos realizados en este aspecto difieren en el algoritmo de optimización empleado: desde el método de búsqueda simplex [21] hasta métodos más sofisticados de *machine learning* como AIS (*Artificial Inmune System*) [22]. En [23] se utilizan redes neuronales y SVM (*Support Vector Machines*) para situar el dron en uno los 4 cuadrantes en torno a la línea.

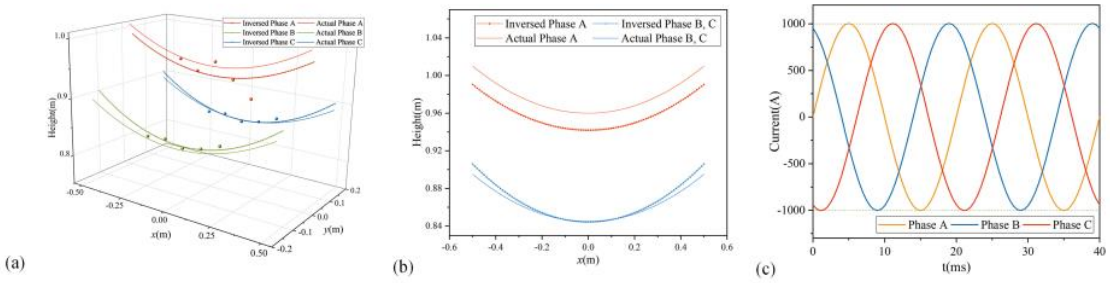


Figura 12. Reconstrucción de los parámetros dinámicos de la corriente y posición de la línea de alta tensión a partir de las medidas de campo magnético. Figuras extraídas de [20].

En definitiva, la integración de todos estos sensores en UAV ha abierto el camino para la inspección cada vez más autónoma e inteligente de las líneas de alta tensión, abaratando enormemente los costes de mantenimiento. En [24] se presentan los avances hasta la fecha en esta materia.

1.2 Objetivo del proyecto

El proyecto aborda la posibilidad de emplear un sistema, basado en un único sensor de campo magnético e integrado en un UAV, que permita calcular su distancia a la línea de alta tensión en su recorrido a lo largo de ella para usarse durante tareas de inspección de la red eléctrica. Como se ha visto, el uso de estos sensores ofrece grandes ventajas respecto a la opción LiDAR: son más sencillos y de menor peso, reduciendo costes.

Usando las nubes de puntos extraídas por el LiDAR y las medidas del campo magnético, la idea consiste en entrenar modelos de *machine learning* para correlacionar ambas medidas. Para lograr este objetivo, es imprescindible llevar a cabo un procesamiento de las nubes de puntos para así poder extraer de estas la distancia media a la línea de alta tensión

El objetivo del proyecto consiste precisamente en llevar a cabo este procesamiento sobre las nubes extraídas durante un vuelo experimental real. Esta tarea no es trivial, pues para llegar a la distancia media final es necesario pasar estas nubes de características tan dispares por una serie de algoritmos de filtrado y segmentación. Respecto a la información del campo magnético, esta no requiere de tratamiento, pues se nos proporciona directamente el espectro de potencia de la señal medida.

El objetivo final del proyecto consiste por tanto en elaborar una herramienta para el procesamiento automático de las medidas extraídas durante los vuelos experimentales, obteniendo así los datos que permitan alimentar los modelos de *machine learning*.

El documento se organiza en base a los siguientes capítulos:

- **Capítulo 1 Introducción:** en este primer capítulo introductorio se pone en contexto el trabajo justificando su interés académico. También se recoge el estado del arte, que constituye un resumen sobre los artículos de investigación más relevantes referentes a la inspección autónoma de líneas de alta tensión, haciendo especial hincapié en el empleo de sensores LiDAR y sensores de campo magnético. Por último, se expone el objetivo del presente trabajo.

- **Capítulo 2 Descripción del entorno y de los sensores:** en primer lugar, se describe el entorno en el que opera el dron, esto es, la línea de alta tensión. Se presta especial importancia a la forma del campo magnético generado por la línea. A continuación, se describen los sensores utilizados (LiDAR y magnetómetro) explicando en detalle su funcionamiento.

- **Capítulo 3 Procesamiento de nubes de puntos:** este capítulo es la base del proyecto, y en él se detalla la metodología seguida para llevar a cabo el procesamiento de las nubes de puntos. En primer lugar, pone en contexto el uso del entorno ROS y la librería PCL, para justo a continuación presentar los algoritmos utilizados y su implementación en el programa desarrollado para el procesamiento de nubes de puntos. El último paso consiste en un postprocesamiento de los datos.

- **Capítulo 4 Resultados:** este es el capítulo que recogen los resultados del programa descrito en el capítulo anterior. Los resultados se muestran paso por paso, visualizando las nubes resultantes en cada uno de ellos. Finalmente se presenta la evolución de la distancia media a la línea con el tiempo, y se hace una comprobación de estos resultados.

- **Capítulo 5 Conclusiones y líneas futuras:** finalmente, se presentan las conclusiones del proyecto y las líneas futuras para su continuación.

2 DESCRIPCIÓN DEL ENTORNO Y DE LOS SENSORES

Antes de comenzar a describir el trabajo realizado respecto al procesamiento de nubes de puntos se ha considerado oportuno hablar sobre el entorno en el que se llevó a cabo el vuelo experimental y sobre los sensores incorporados en el UAV. Esto nos permitirá conocer con más detalle el problema e interpretar correctamente los datos de partida.

Tal y como se ha comentado en el capítulo de introducción, la misión del UAV consiste en un seguimiento de líneas de alta tensión para su inspección. En un primer apartado describiremos las características más relevantes del tendido eléctrico haciendo hincapié en sus componentes más importantes, su geometría y en el campo magnético generado por las líneas conductoras.

En un segundo apartado se describirán generalmente los sensores utilizados para la adquisición de datos durante el vuelo experimental. Estos sensores son el LiDAR (*Laser Imaging Detection and Ranging*) que nos proporciona la información espacial de la escena a través de nubes de puntos; y el sensor de campo magnético, que nos proporciona las medidas de la densidad de flujo magnético (en ambos casos para distintos instantes de tiempo). Para las medidas correspondientes al campo magnético, también se hace uso de un analizador de espectro para obtener la densidad espectral de potencia correspondiente a dicha señal.

2.1 Líneas de alta tensión

Las líneas de alta tensión forman parte de los sistemas de transporte y distribución de la energía eléctrica. La importancia de elevar la tensión radica en la disminución de la intensidad de corriente que circula por la línea para una misma potencia. Esto implica menores pérdidas debido al efecto Joule y menor cantidad de material necesario para los cientos de kilómetros de conductor. Esta elevada tensión también tiene no obstante una serie de desventajas como se verá más adelante.

Además de por los cables conductores, las líneas de alta tensión están constituidas por otro tipo de elementos que cumplen funciones tanto estructurales como eléctricas. El conocer estos elementos y su geometría es fundamental para interpretar correctamente las nubes de puntos obtenidas por el sensor LiDAR.

Por otro lado, describiremos el campo magnético generado por las líneas de alta tensión.

2.1.1 Descripción general

Atendiendo al Real Decreto 223/2008, las líneas de alta tensión en España se clasifican atendiendo al nivel de tensión según las siguientes categorías [25]:

- **Líneas de 3ª Categoría:** también denominada media tensión en la industria eléctrica, abarca tensiones nominales entre 1 y 36kV. Usada para generación y distribución, orientada a clientes industriales.
- **Líneas de 2ª Categoría:** abarca tensiones nominales superiores a 30kV e iguales o inferiores a 66kV. Usada principalmente para distribución.
- **Líneas de 1ª Categoría:** abarca tensiones nominales superiores a 66kV e inferiores a 220kV. Usada en transporte y distribución.
- **Líneas de categoría especial:** tensiones nominales iguales o superiores a 220kV. Usada para el transporte a grandes distancias.

En cualquier caso, la frecuencia de transmisión es de 50Hz.

Podemos distinguir los siguientes elementos fundamentales en las líneas de alta tensión:

- **Cable conductor:** las líneas conductoras de alta tensión generalmente consisten en cables desnudos trenzados de aluminio con un núcleo de acero galvanizado, denominados como ACSR (*Aluminum Conductor Alloy Reinforced*). El núcleo de acero le otorga al conductor una mayor resistencia estructural, mientras que el aluminio tiene la mejor relación entre conductividad, peso y coste.

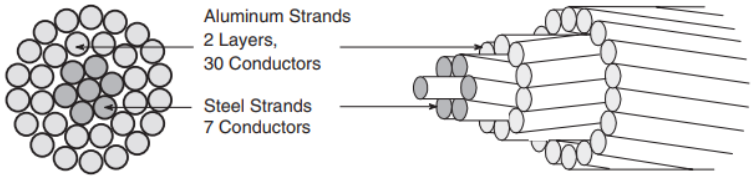


Figura 13. Cable conductor trenzado de tipo ACSR utilizado en las líneas de alta tensión. El trenzado aporta flexibilidad.

Las líneas de alta tensión normalmente portan la energía eléctrica a través de tres fases. Además, para voltajes superiores a 220kV se suele usar más de un conductor por fase, lo que constituye un haz de conductores. El motivo de esto reside en la necesidad de una mayor capacidad portante y en las pérdidas debido al efecto corona. El efecto corona consiste en la ionización del gas que rodea el conductor, que ocurre en presencia de un elevado gradiente del potencial eléctrico en la superficie de este. Las consecuencias son pérdidas importantes de energía e interferencias de radio.

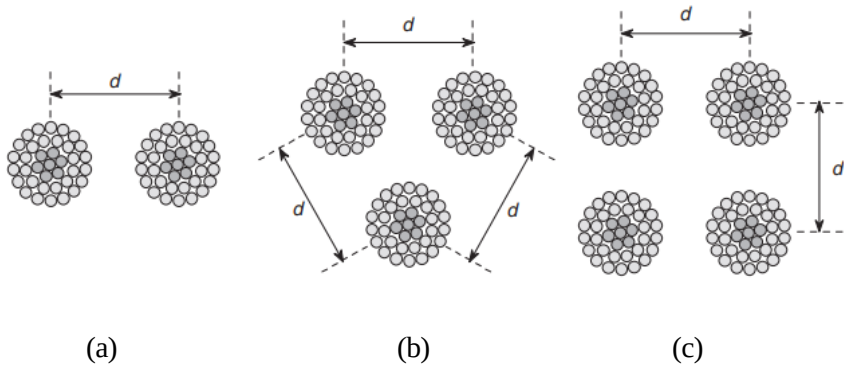


Figura 14. Distintas configuraciones de haces con (a) dos, (b) tres y (c) cuatro conductores por fase.

Los conductores pertenecientes a un mismo haz se mantienen a una distancia aproximadamente constante, lo cual se consigue a través de crucetas o separadores. Así mismo, es crucial mantener separada cada fase entre sí con el fin de evitar efectos de acoplamiento capacitivo e inductivo (hay que tener en cuenta que cada fase se encuentra a un potencial distinto). Estos fenómenos se ven acentuados en líneas de mayor longitud.

Por último, la flexibilidad que aporta el trenzado a estos cables conductores hace que estos estén sometidos casi en su totalidad a esfuerzos de tracción debido a su propio peso. Esto conlleva que los cables cuelguen formando una curva característica denominada catenaria.

- **Apoyos**: los apoyos o torres eléctricas son las estructuras que soportan y anclan el tendido eléctrico a una cierta altura, normalmente comprendida entre 15 y 55m. La forma más común consiste en una

celosía de acero cuyo diseño concreto varía según el número de conductores por fase, la disposición elegida para los conductores, la altura de diseño y la tensión de la línea.

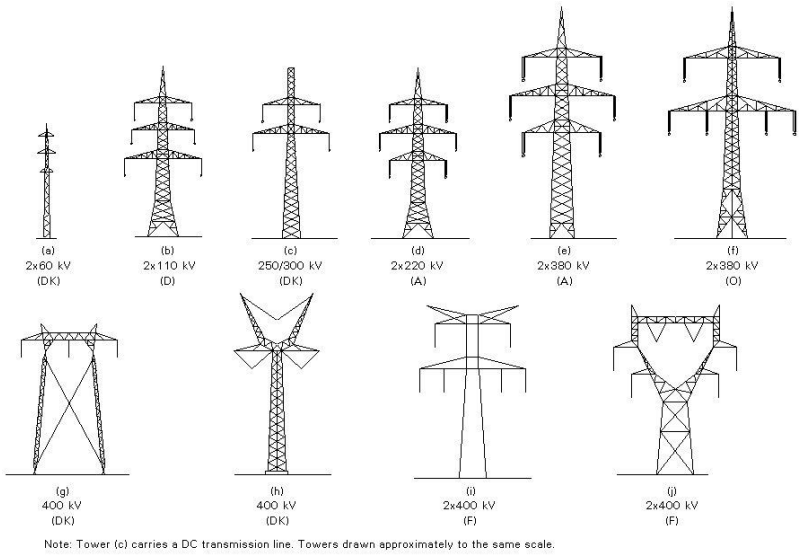


Figura 15. Distintos diseños de torre eléctrica, para los cuales se especifica el nivel de tensión nominal y el número de conductores por fase.

Atendiendo al tipo de apoyo proporcionado a la línea, podemos distinguir entre apoyos de suspensión, apoyos de amarre, apoyos de anclaje (encargados de hacer frente a solicitaciones excepcionales) y apoyos de principio o fin de línea. De igual forma, atendiendo a la posición relativa entre las líneas que se apoyan en la torre distinguimos entre apoyos de alineación y apoyos de ángulo.

(a) (b) (c)

Figura 16. Distintos tipos de apoyo en torres eléctricas: (a) apoyo de suspensión, (b) apoyo de amarre y (c) apoyo de principio o fin de línea.

Además del peso de los conductores, los apoyos deben soportar otro tipo de cargas dinámicas derivadas de agentes externos, como son el viento o la nieve, además de condiciones extremas de temperatura y corrosión.

- **Cable de tierra:** se trata de un cable de acero galvanizado con posible recubrimiento de aluminio, utilizado para absorber el impacto de rayos y proteger los cables conductores. Por tanto, deben situarse en la parte más alta de la torre. Este cable adicional normalmente se aprovecha para establecer comunicaciones entre los distintos puntos de la red de transporte. Para ellos, se hace uso de cables tipo OPGW (*Optical Ground Wire*), con un núcleo formado por fibra óptica.

- **Aisladores:** los aisladores cumplen la doble función de sostener los conductores proporcionando además la resistividad eléctrica suficiente para lidiar con la diferencia de potencial con la torre (no sólo bajo condiciones normales de operación, sino también durante picos de tensión debido al impacto de rayos). Tradicionalmente han estado constituidos por materiales cerámicos como la porcelana o el vidrio, aunque en la actualidad se emplean materiales compuestos de resina epoxi con núcleo de fibra de vidrio.

  Para su uso en líneas de alta tensión estos aisladores vienen en forma de cadenas de suspensión o cadenas de amarre, las cuales proporcionan una excelente resistencia mecánica a tracción.

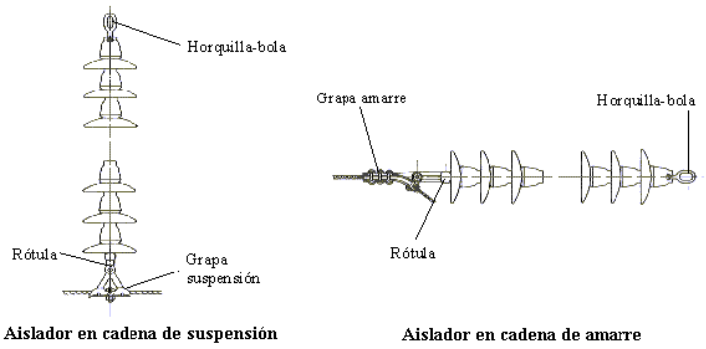
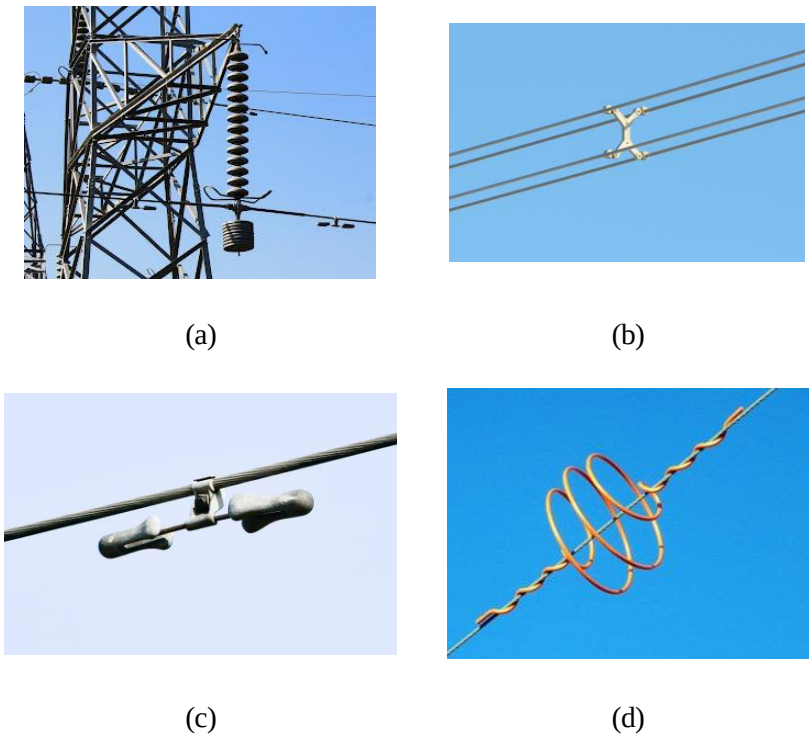


Figura 17. Aisladores en cadena de suspensión y de amarre utilizados en las líneas de alta tensión.

- **Herrajes y accesorios:** además de los elementos principales descritos previamente, las líneas de alta tensión disponen de herrajes para la fijación de unión de estos elementos y de elementos auxiliares como contrapesos (para el equilibrado), amortiguadores (para absorber las vibraciones de los cables), separadores (para fijar la distancia entre conductores de la misma fase), balizas (para visualización), salvapájaros (para prevenir el impacto de aves) y otros.



(a) (b)

(c) (d)

Figura 18. Elementos auxiliares utilizados en las líneas de alta tensión: (a) contrapeso, (b) separador, (c) amortiguador tipo *stockbridge* y (d) salvapájaros.

A modo de resumen se muestran las siguientes fotografías cortesía de [26], donde se muestran estos elementos de manera global.

(a) (b)

Figura 19. Fotografías (a) y (b) donde se indican los elementos principales de las líneas de alta tensión. Cortesía de [26].

2.1.2 Catenaria

La catenaria es la curva que forma un cable sin rigidez a flexión que cae por su propio peso. Por tanto, estos cables están sometidos únicamente a esfuerzos de tracción, característica que permite deducir la ecuación de esta curva. Además, es de gran utilidad en obra civil, pues la curva invertida es utilizada en los denominados arcos catenarios al sólo estar sometidos en este caso a esfuerzos de compresión.

La catenaria es por tanto la curva que formarán nuestras líneas de conductor, lo que hace necesario describirla con detalle. Generalmente la catenaria se expresa como una curva simétrica descrita por la siguiente ecuación:

$$z(x) = a\cosh\frac{x}{a} \tag{2-1}$$

Donde a es un parámetro determinado. Este parámetro deberá determinarse a partir de las características geométricas del cable, como la altura a la que está suspendido o su longitud.

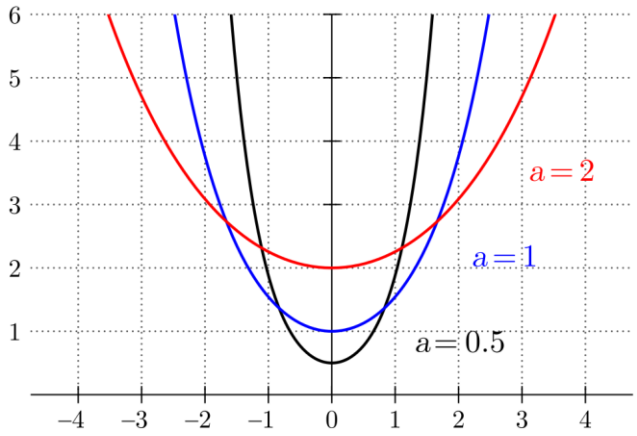


Figura 20. Forma de la catenaria para distintos valores del parámetro a.

En numerosas aplicaciones la catenaria se aproxima a través de una parábola como la de la ecuación (2-2), siendo esta aproximación válida cerca del punto de simetría:

$$z_p(x) = a + \frac{a}{2}\left(\frac{x}{a}\right)^2 \tag{2-2}$$

Para su uso en líneas de alta tensión con apoyos a la misma altura, la ecuación permite redefinirse de la siguiente forma:

$$z(x) = \frac{L}{b}\left(\cosh\frac{bx}{L} - \cosh\frac{b}{2}\right) + h \tag{2-3}$$

Donde L es la distancia entre apoyos, h es la altura de los apoyos y b es un parámetro que determina el *sag* s, esto es, la diferencia entre la altura máxima y mínima del cable.

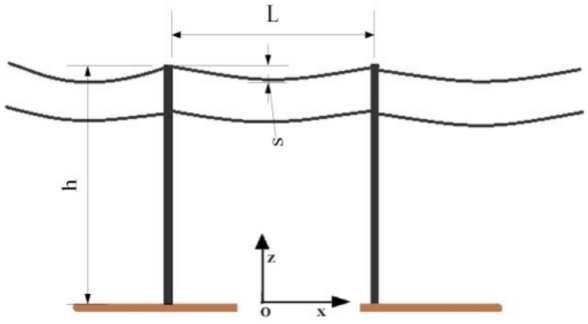


Figura 21. Parámetros de la ecuación de la catenaria para líneas de alta tensión. Figura extraída de [23].

Esta distancia *s* resulta ser:

$$s = h - z(0) = h - \left[\frac{L}{b}\left(1 - \cosh\frac{b}{2}\right) + h\right] = \frac{L}{b}\left(\cosh\frac{b}{2} - 1\right) \tag{2-4}$$

Siendo nulo para *b* también nulo.

Para el propósito de este proyecto resulta interesante determinar si un segmento de cierta longitud de esta curva puede aproximarse localmente como un tramo de pendiente constante. La pendiente de la curva resulta ser:

$$z'(x) = \sinh\frac{bx}{L} \tag{2-5}$$

Considerando la variable $\xi = \frac{bx}{L}$, vamos a obtener el desarrollo en serie de Taylor de dicha pendiente en torno a un punto de referencia ξ_0:

$$\begin{gathered} z'(\xi_0 + \Delta\xi) = z'(\xi_0) + \cosh\xi_0\,\Delta\xi + O(\Delta\xi^2) \rightarrow \\ z'(\xi_0 + \Delta\xi) - z'(\xi_0) \approx \cosh\xi_0\,\Delta\xi \end{gathered} \tag{2-6}$$

Por tanto, la pendiente se mantendrá aproximadamente constante para ξ_0 y $\Delta\xi$ pequeños. Esto quiere decir que la aproximación será mejor para puntos de referencia cercanos al origen y cuando no nos alejemos muchos demasiado del punto de referencia en relación con la distancia entre apoyos, $\frac{\Delta x}{L} \ll 1$. Además, cuanto mayor sea el *sag s* (o, equivalentemente, mayor sea *b*) peor será dicha aproximación.

2.1.3 Campo magnético

En las inmediaciones de las líneas de alta tensión aparece un campo electromagnético originado por la propia intensidad de corriente que circula por los conductores. Este campo electromagnético no es en absoluto despreciable, y es tenido en cuenta dentro del estudio del impacto medioambiental del tendido eléctrico, ya que puede llegar a tener consecuencias negativas para la salud humana.

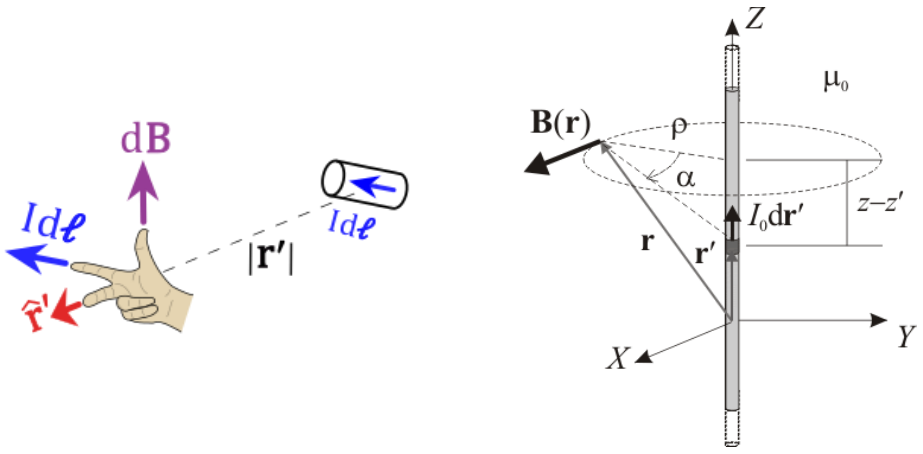
La ley de Biot-Savart establece que un hilo de conductor por el que circula una corriente de intensidad *I* genera a su alrededor un campo magnético $\boldsymbol{B}$ de la forma:

$$\boldsymbol{B}(\boldsymbol{r}) = \frac{\mu_0}{4\pi}\int_C \frac{I d\boldsymbol{l} \times \boldsymbol{r}'}{|\boldsymbol{r}'|^3} \tag{2-7}$$

Donde:

- μ_0 es la permeabilidad magnética del vacío.

- *C* es la curva que forma el hilo conductor.
- *I* es la intensidad de corriente que circula por el conductor.
- ***dl*** es el diferencial de hilo, de dirección tangente al mismo.
- ***r'*** es el vector que va desde el diferencial de hilo hasta el punto en cuestión, $\boldsymbol{r' = r - l}$.



(a) (b)

Figura 22. Diagramas (a) y (b): ley de Biot-Savart.

Hay que tener en cuenta que esta ley, pensada para corrientes estacionarias, también será válida para corrientes cuasi-estáticas de baja frecuencia. Se debe cumplir que la distancia al hilo conductor sea mucho menor que la longitud de onda $\lambda = \frac{c}{2\pi f}$, lo cual se cumple sobradamente para $f = 50Hz$.

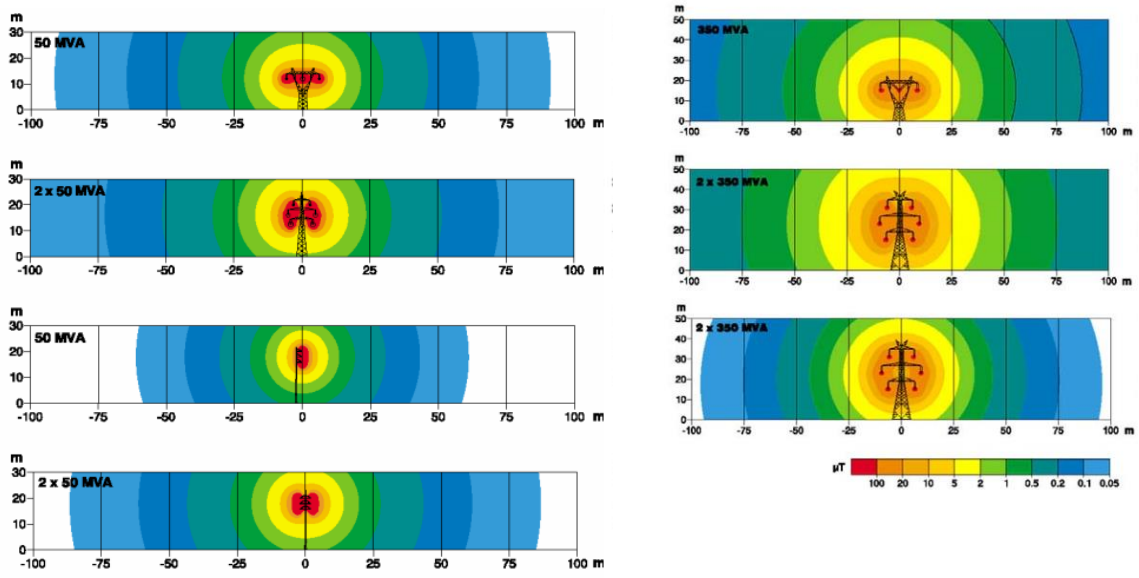
Si aproximamos el campo magnético real por el generado por un hilo rectilíneo e infinito de corriente, el campo magnético resultante tiene la dirección mostrada en la Figura 22 y su módulo:

$$B = \frac{\mu_0 I}{2\pi r} \tag{2-8}$$

Donde en este caso *r* es la distancia perpendicular al hilo conductor. Esta sería la contribución aportada por cada fase del tendido eléctrico. Vemos por tanto que el campo magnético sigue una ley de tal forma que es directamente proporcional a la intensidad de la corriente que circula por el hilo conductor e inversamente proporcional a la distancia. En esta aproximación se ha despreciado tanto el efecto de la curva catenaria como el efecto tierra (modelado mediante corrientes espejo). Sin embargo, estos efectos se vuelven despreciables a medida que nos acercamos a los hilos conductores.

No obstante, hay un obstáculo importante que nos impide conocer el valor del campo magnético a una cierta distancia, y este es el desconocimiento de la intensidad de corriente *I*. Esta corriente varía a lo largo del día, ajustándose a la demanda de energía a través de la línea, y depende de un gran número de parámetros.

En la Figura 23 se muestran los valores típicos medidos para distintos valores de la tensión, potencia aparente y número de conductores por fase. Puede verse que el orden de magnitud es el μT. El campo magnético se hace notar para distancias incluso superiores a 50m, y para una distancia en torno a los 10m, el valor del campo magnético se encuentra en torno a los 5μT. No obstante, el campo magnético puede variar sustancialmente incluso para los mismos valores de tensión nominal y potencia aparente simplemente variando la disposición de los conductores.



(a) (b)

Figura 23. Campo magnético generado por líneas de alta tensión de (a) 150kV de tensión nominal y 50MVA de potencia equivalente por fase; y (b) 400kV de tensión nominal y 350MVA de potencia equivalente por fase. Figuras extraídas de [27].

2.2 Sensores

En esta sección se van a describir con detalle los sensores utilizados para la adquisición de datos durante el vuelo experimental. Estos sensores básicamente son dos: un sensor LiDAR, el cual ha obtenido una serie de nubes de puntos en cada instante; y un sensor de campo magnético, el cual ha hecho lo propio con los valores de la densidad de flujo magnético.

2.2.1 LiDAR

El LiDAR (*Laser Imaging Detection and Ranging* o *Light Detection and Ranging*) es un sensor activo que nos permite realizar escaneos láser 3D de muy alta resolución y a elevadas frecuencias, proporcionando una nube de comprendida en el campo de visión del sensor (FoV, *Field of View*). Es utilizado cada vez más en un gran número de aplicaciones orientadas a la topografía, la navegación y la inspección.

Figura 24. LiDAR embarcado en un UAV.

El sensor nos permite medir la distancia desde un láser hasta el cuerpo al que apunta midiendo el tiempo de retraso desde la emisión del pulso de luz hasta la detección de la señal reflejada. Esta detección se produce a través de fotosensores de tipo CMOS. El fenómeno por el cual parte de una señal reflejada vuelve en la misma dirección de incidencia se conoce como retrodispersión. Despreciando el movimiento del cuerpo al que se apunta y el propio movimiento del sensor en el caso de sistemas embarcados, la distancia se calcula a través de la simple ecuación:

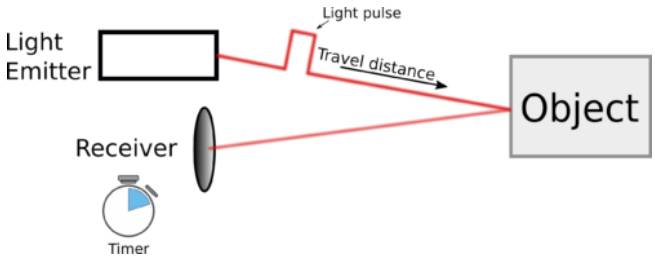


Figura 25. Método *Time of Flight* para el cálculo de la distancia hasta un cuerpo iluminado por un haz de luz.

$$d = \frac{ct}{2} \tag{2-9}$$

Donde *c* es la velocidad de la luz en el vacío y *t* el tiempo de retardo medido. Esta técnica para el cálculo de la distancia a cuerpos iluminados se conoce como ToF (*Time of flight).*

Existen dos tipos de LiDAR, los cuales emplean diferentes estrategias de emisión y recepción:

- **Flash LiDAR:** en el Flash LiDAR el láser diverge a través de unas lentes para iluminar la escena por completo en cada pulso. El receptor, por tanto, está constituido por una matriz de sensores en la que cada uno de ellos obtiene la información correspondiente a la distancia y a la intensidad retrodispersada. El resultado es una imagen en la que en vez de color los píxeles almacenan esta información. Esto se conoce como nube de puntos 2D o nube de puntos ordenada.

  Este tipo de LiDAR es recomendable en aplicaciones en las que exista un desplazamiento relativo entre el sensor y el objeto, pues para velocidades elevadas el LiDAR de escaneo puede llegar a distorsionar la nube de puntos.

- **Scan LiDAR:** en este caso el haz de luz no diverge, sino que es transmite como un haz de luz colimado iluminando un único punto de la escena en cada instante. Se requiere por tanto de un sistema de escáner que nos permita orientar el láser en todas las direcciones del campo de visión del LiDAR. A diferencia

del Flash LiDAR, deja de ser necesario un *array* de sensores para su funcionamiento. En la Figura 26 se muestran las diferencias entre los dos tipos de LiDAR.

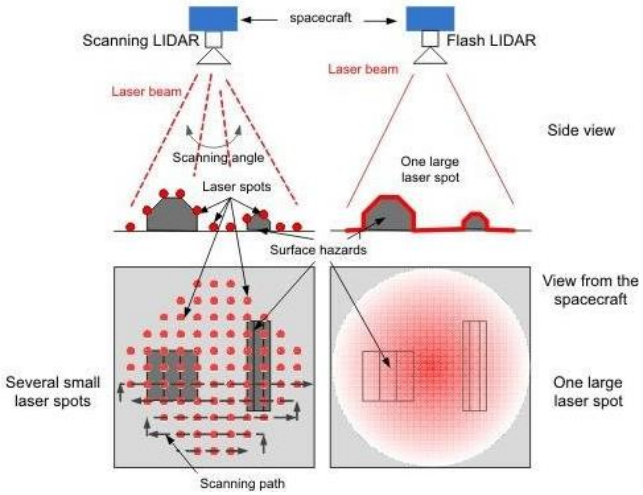


Figura 26. Visualización de los tipos de LiDAR Scan y Flash. Figura extraída de [28].

El LiDAR está constituido de forma general por los siguientes elementos. Estos deben estar sincronizados para que el LiDAR pueda cumplir con su cometido:

- **Láser:** el láser es el emisor del sistema LiDAR, y consiste en dispositivo que emite un haz de luz amplificado mediante emisión estimulada de fotones (de sus siglas en inglés, *light amplification by stimulated emision of radiation*). A diferencia de otras fuentes de luz, el láser emite ondas coherentes: el haz de luz se mantiene para largas distancias recorridas, y su longitud de onda se sitúa en una banda también estrecha. Esta longitud de onda puede corresponderse tanto con luz ultravioleta como luz infrarroja o luz visible (entre los 10µm y los 250nm).

  El tipo de láser utilizado depende de la tarea concreta del LiDAR. Para uso en entornos urbanos se emplea la emisión de micropulsos de menor contenido energético y por tanto más seguros para el ojo humano. En contraposición, su uso para el estudio de la atmósfera requiere de pulsos de alta potencia. La longitud de onda utilizada también tiene en cuenta este aspecto.

- **Escáner:** el escáner es el elemento encargado de dirigir el haz de luz generado por el láser en la dirección deseada. Afecta directamente a la resolución angular del LiDAR, por lo que su diseño es importante. Este es únicamente necesario para el Scan LiDAR.

  Existen dos tipos de soluciones posibles para el escáner. Por un lado, tenemos las soluciones mecánicas clásicas con partes móviles. Por el otro, las más novedosas soluciones no mecánicas (*solid-state*), sin partes móviles.

  Entre las soluciones mecánicas, la más sencilla es sin duda el uso de ejes de rotación y espejos. Este tipo de escáner suele ser también el más voluminoso, pero usando distintas configuraciones de espejo podemos lograr técnicas de escaneo efectivas (Figura 27). Una alternativa que ha surgido en los últimos años son los *Microelectromechanical mirrors*, que son una solución compacta de bajo peso que funcionan tanto en uno como en dos ejes (Figura 28). Estos dispositivos son capaces de rotar un pequeño espejo mediante la aplicación de fuerzas electromecánicas.

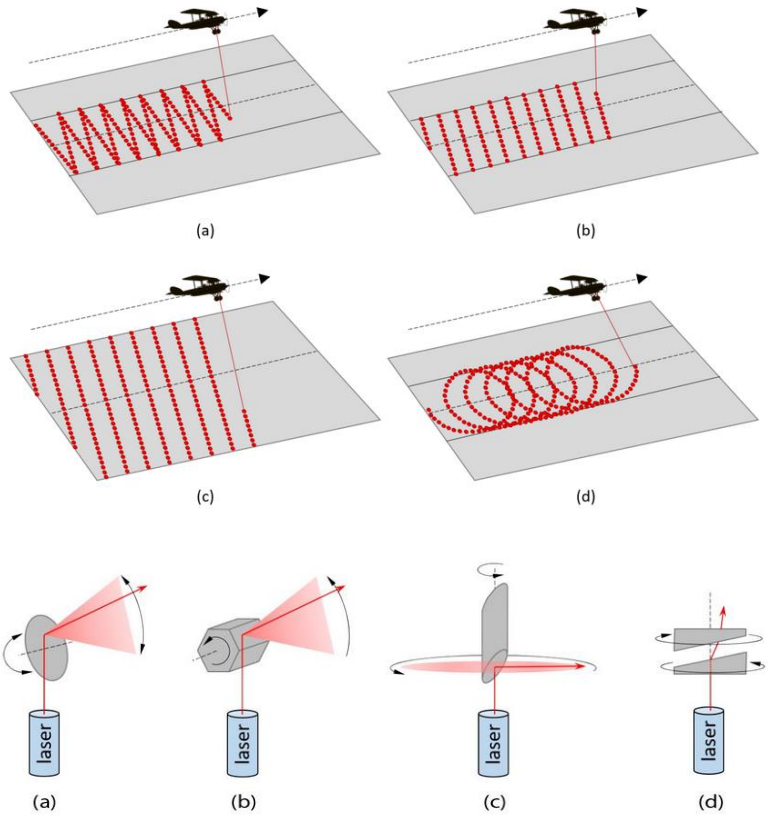
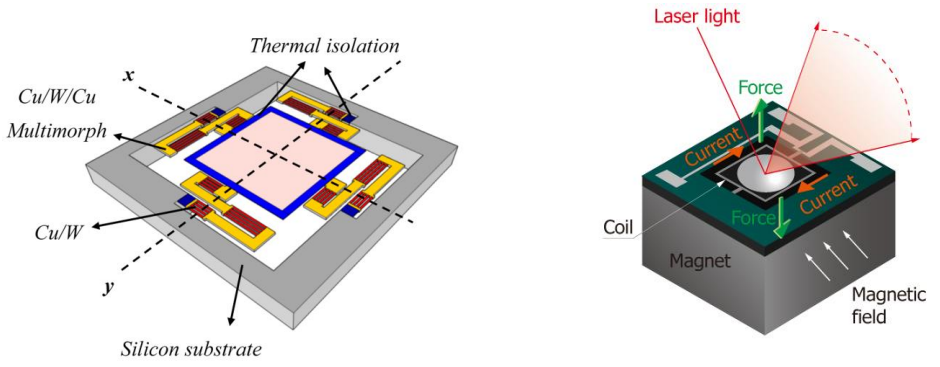


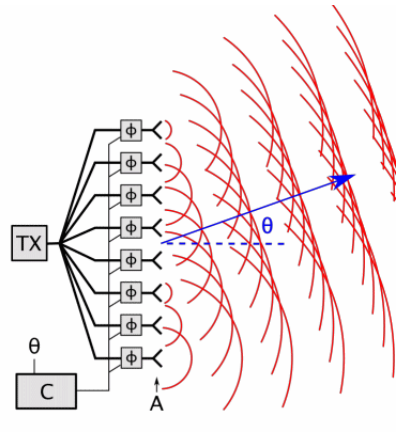
Figura 27. Distintas configuraciones de espejo y los resultados del escaneo a lo largo de una trayectoria rectilínea: (a) espejo oscilante, (b) espejo poligonal en rotación, (c) espejo en rotación y (d) espejo en cuña en rotación. Figuras extraídas de [29].



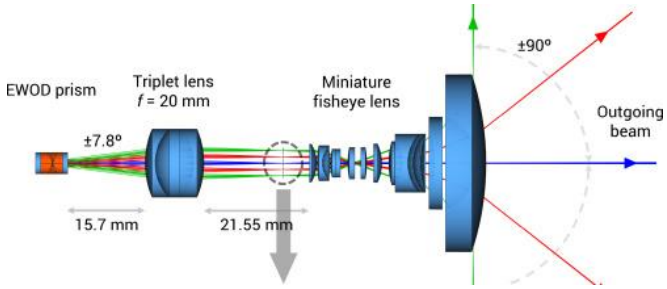
(a) (b)

Figura 28. Distintos diseños *Microelectromechanical mirrors* o microescáneres. (a) Diseño experimental de doble eje en el que las fuerzas aplicadas sobre el espejo son de origen térmico [30]. (b) Diseño comercializado por Hamamatsu en el que las fuerzas aplicadas son de origen magnético.

Entre las soluciones no mecánicas encontramos el uso de antenas en fase. Estas antenas son capaces de direccionar el haz de luz controlando la fase de cada uno de los distintos emisores. Por otro lado, encontramos soluciones como la propuesta en [31], en la que un prisma, junto a una serie de lentes, son capaces de dirigir el haz de luz aplicando una diferencia de potencial sobre dicho prisma. Esto es posible debido al fenómeno conocido como *electrowetting.*



(a)



(b)

Figura 29. Soluciones no mecánicas para el escaneo LiDAR. (a) Antenas en fase y (b) prisma corregido por *electrowetting*.

- **Receptor:** el receptor es el encargado de detectar la reflexión del haz de luz pulsado para calcular el tiempo de desfase y la energía recibida. Está constituido por un único fotosensor o por una matriz de sensores para el caso de la obtención de nubes de puntos 2D (Flash LiDAR). Estos sensores deben funcionar a altas velocidades, equivalentes a la frecuencia del láser. Los fotosensores más utilizados son de tipo CMOS (*Complementary metal oxide semiconductor*).

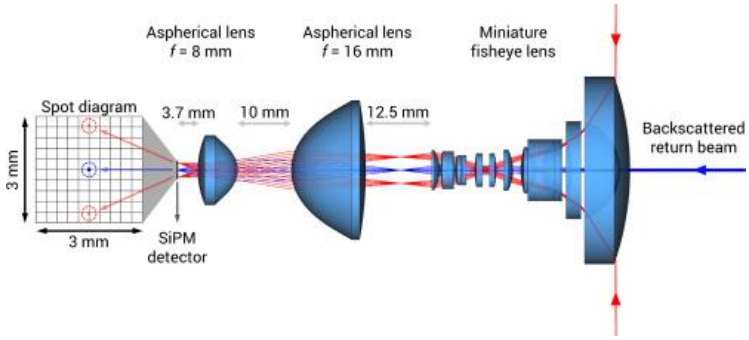


Figura 30. Receptor tipo Flash LiDAR. Puede verse la óptica necesaria para redirigir los haces de luz reflejados, así como la matriz de sensores. Figura extraída de [31].

2.3 Magnetómetro

Un magnetómetro es un sensor capaz de medir la dirección y la intensidad del campo magnético en un punto determinado. El ejemplo más sencillo de magnetómetro que podemos encontrar es el de una brújula, capaz de determinar la dirección del campo magnético terrestre. Cuando hablamos de campo magnético nos referimos generalmente al campo vectorial ***B*** descrito en las ecuaciones de Maxwell, también denominado inducción magnética o densidad de flujo magnético, y cuya unidad en el SI es el Tesla $\left(\mathrm{T} = \frac{\mathrm{kg}}{\mathrm{As}^2}\right)$.

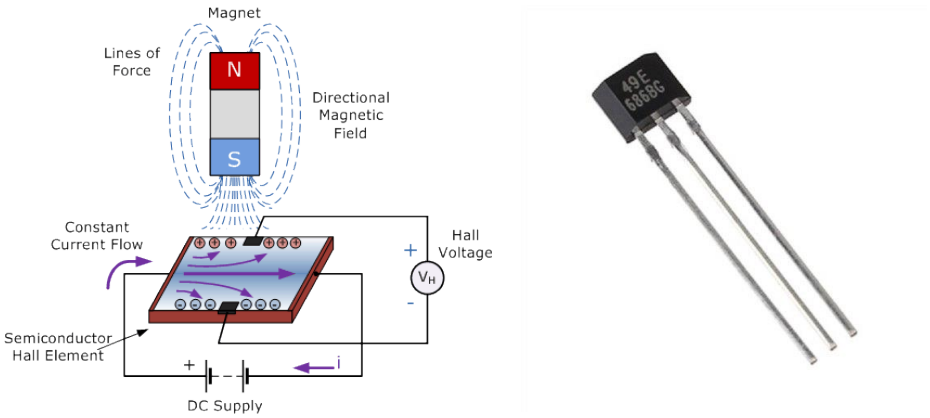
Al tratarse de una magnitud vectorial, podemos distinguir entre dos tipos fundamentales de magnetómetros: escalares y vectoriales. Los primeros nos proporcionan la magnitud del campo magnético, mientras que los segundos nos proporcionan cada una de sus componentes. Para obtener la magnitud absoluta del campo magnético estos sensores deben estar calibrados internamente. En caso contraría tendríamos un variómetro, un sensor capaz de detectar variaciones relativas del campo magnético.

Al estar el magnetismo presente en una gran variedad de fenómenos físicos, existen de igual forma una gran variedad de magnetómetros para muy diversas aplicaciones. Uno de los principales usos de los magnetómetros es en la geofísica, donde se usan sensores como el magnetómetro de precesión protónica o el magnetómetro de bombeo óptico. El primero basa su funcionamiento en el fenómeno de resonancia magnética de núcleos de hidrógeno, mientras que el segundo se basa en la variación con el campo magnético de la capacidad de absorción de un haz de luz por parte de un gas. Aunque estos sensores ofrecen una gran precisión, no están pensados para su uso en UAV.

Hoy en día existen soluciones de magnetómetros compactas y precisas pensadas para su integración en sistemas robóticos incluidos los UAV. Estos sensores normalmente permiten obtener una de las componentes del campo magnético, por lo que debe situarse uno en cada eje para obtener la magnitud vectorial, o bien emplear diseños con esta solución ya integrada. A continuación, se explican con más detalle los más relevantes.

- **Sensor de efecto Hall:** se trata del sensor más usado y carece de partes móviles. Como su nombre indica, su funcionamiento se basa en el conocido efecto Hall. Dicho efecto consiste en la aparición de un voltaje transversal a un conductor por el que circula una corriente y que está en presencia de un campo magnético. Debido a este movimiento de electrones, aparece una fuerza de Lorentz que acumula la carga en los extremos de la placa conductora dando lugar a esta diferencia de potencial (Figura 31).

  Las principales ventajas de este sensor residen en que carece de partes móviles, que tienden a reducir su durabilidad, y en su compacticidad.

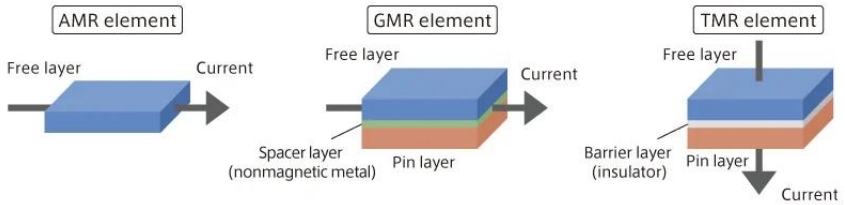


(a) (b)

Figura 31. Sensor de efecto Hall: (a) efecto Hall [32] y (b) ejemplo de sensor.

- **Sensores magnetorresistivos:** su funcionamiento se basa en la propiedad que presentan ciertos materiales con alta permeabilidad magnética denominada magnetorresistencia. Los materiales que presentan esta propiedad como el Permalloy (aleación de níquel y hierro) varían su resistencia magnética con el campo magnético. Esta propiedad puede manifestarse de varias formas, siendo la más simple de esta la que se conoce como AMR (*Anisotropic Magnetoresistance*). Este fenómeno ha sido investigado para desarrollar sensores de altas prestaciones como son los sensores TMR (*Tunnel Magnetoresistance*) y GMR (*Giant Magnetoresistance*).

  Al igual que los sensores de efecto Hall, se tratan de sensores muy compactos y, además, de elevadas frecuencias de muestreo.

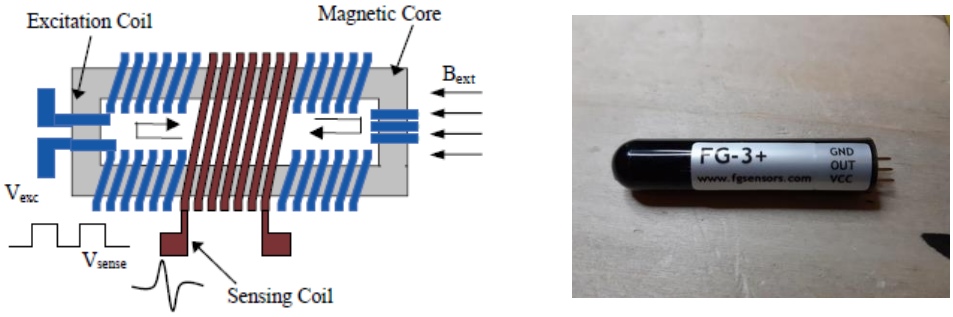


(a) (b) (c)

Figura 32. Distintos tipos de sensores magnetorresistivos: (a) AMR, (b) GMR y (c) TMR. Figura extraída de [33].

- **Magnetómetro de saturación:** este sensor (en inglés, *Fluxgate magnetometer*) consiste en un núcleo de un material de alta permeabilidad magnética como el Permalloy que se envuelve en un par de bobinas de conductor. Este núcleo se somete a ciclos de saturación magnética a través del campo magnético generado por la corriente que circula por una de las bobinas. Este ciclo de saturación es sensible a la presencia de un campo magnético externo. En caso de que este campo externo esté presente, la corriente inducida en la segunda bobina deja de ser la misma que la corriente que circula por la primera (Figura 33).

  Al igual que los casos anteriores, se trata de una solución compacta y de bajo coste.



(a) (b)

Figura 33. (a) Esquema de funcionamiento de un magnetómetro de saturación y (b) ejemplo de sensor. Figuras extraídas de [34].

2.3.1 Analizador de espectro

El analizador de espectro es un dispositivo encargado de obtener la densidad espectral de potencia de una señal determinada. La densidad espectral de potencia PSD (*Power Spectral Density*) o espectro de potencia es una función que distribuye la potencia media de una señal entre las distintas frecuencias *f* de su espectro. De esta forma, integrando dicha densidad a lo largo de todo el rango de frecuencias obtendríamos la potencia media de la señal.

(a) (b)

Figura 34. Distintas formas de un analizador de espectro: (a) de sobremesa y (b) sistema embarcado.

Atendiendo a su definición, la potencia media de una señal se calcula como:

$$P = \lim_{T\to\infty} \frac{1}{T} \int_{t_0-T/2}^{t_0+T/2} |x(t)|^2 dt = \lim_{T\to\infty} \frac{1}{T} \int_{-\infty}^{\infty} |x_T(t)|^2 dt \qquad (2\text{-}10)$$

Donde x_T es nula fuera del intervalo delimitado por la integral. Usando el teorema de Parseval podemos expresar la potencia anterior a partir de la transformada de Fourier de la función x_T:

$$P = \lim_{T\to\infty} \frac{1}{T} \int_{-\infty}^{\infty} |\hat{x}_T(f)|^2 df \qquad (2\text{-}11)$$

Donde el integrando es la definición de densidad espectral de potencia S_{xx}:

$$S_{xx}(f) = \lim_{T\to\infty} \frac{1}{T} |\hat{x}_T(f)|^2 \qquad (2\text{-}12)$$

Observando la fórmula, puede verse que la densidad espectral de potencia coincide con la autocorrelación de la señal $x(t)$. A pesar de esta definición formal, en la práctica se calcula localmente S_{xx} para distintos intervalos de tiempo finitos centrados en t_0. Esto nos permite caracterizar en cada instante de tiempo el espectro de una señal no periódica.

Atendiendo al método de obtención del espectro de potencia, los analizadores de espectro pueden ser de dos tipos:

- **Analógico:** utiliza un filtro pasa banda de frecuencia variable o bien un receptor superheterodino para barrer el rango de frecuencias de interés. El ancho de banda de este filtro delimita la resolución del instrumento.
- **Digital:** el espectro se calcula promediando una serie de periodogramas obtenidos mediante el

algoritmo FFT (*Fast Fourier Transform*). Requiere por tanto de un convertidor AD. El aumento de la capacidad computacional ha hecho que este método sea cada vez más usado al ser el más rápido, aunque tiene como principal inconveniente que sólo es capaz de abarcar un ancho de banda estrecho. Por ello, se ha ideado una versión híbrida entre el barrido analógico y el algoritmo FFT para su uso en aplicaciones en tiempo real.

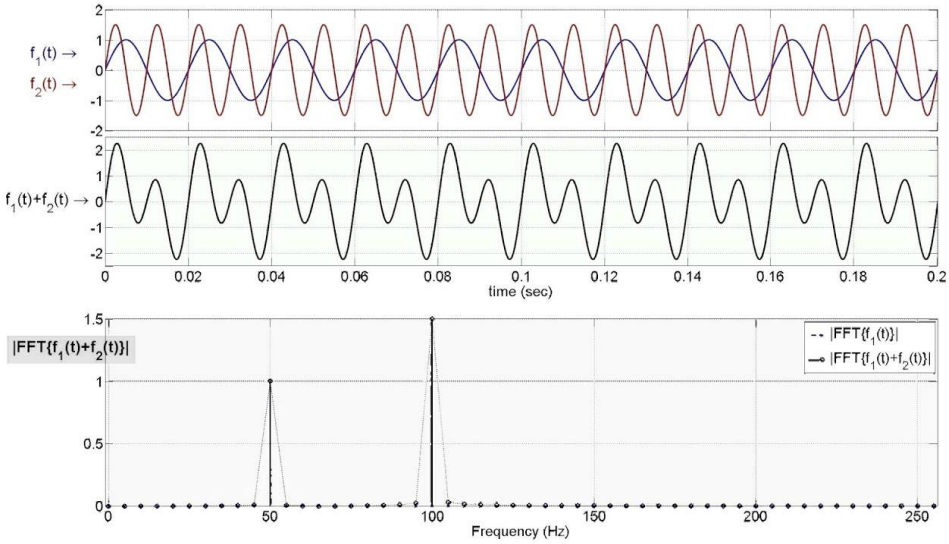


Figura 35. Obtención del espectro de una señal mediante el método FFT.

3 PROCESAMIENTO DE NUBES DE PUNTOS

En este capítulo se van a describir los algoritmos utilizados para el procesamiento de las nubes de puntos obtenidas por el sensor LiDAR durante los vuelos experimentales. Una nube de puntos es un conjunto de puntos n-dimensionales que, además de sus coordenadas XYZ, pueden almacenar otro tipo de información útil.

En este sentido, abarcaremos el procedimiento completo hasta la obtención de la distancia a la línea de alta tensión, que no sólo consiste en aplicar algoritmos de filtrado y segmentación de nubes de puntos, sino que también requiere de una lectura, escritura y postprocesamiento de los datos para su entrada a los modelos de *machine learning*.

Los datos de las nubes de puntos de partida vienen almacenados en archivos *bag*, que son los archivos de almacenamiento de mensajes del entorno ROS. Sin salir de este entorno, se han leído estos archivos y se han extraído las estructuras de datos que representan las nubes de puntos. Una vez tenemos acceso a estos objetos estructurados que contienen los distintos puntos de la nube podemos procesarlos mediante el código desarrollado, el cual implementa diversos algoritmos de la librería PCL. El objetivo final de este procesamiento es el de segmentar las líneas de alta tensión y obtener los coeficientes de los modelos de línea. Por último, se ha llevado a cabo un postprocesamiento de estas líneas extraídas para filtrar errores y calcular la distancia media al tendido de alta tensión, que será el *input* para los modelos de *machine learning.*

3.1 Robot Operating System ROS

ROS (*Robot Operating System*) es un entorno de trabajo (*software framework*) de código abierto constituido por un conjunto de librerías orientadas a la programación modular y flexible de robots. ROS es para el robot como Windows o Linux son para nuestro PC; proporciona los servicios básicos para el desarrollo de plataformas: abstracción de hardware, control a bajo nivel, intercambio de información entre procesos, gestión de paquetes de código y funcionalidades básicas. Además, ROS dispone de librerías para el desarrollo de código propio, convirtiéndose en el entorno ideal para la colaboración y modularidad a través de repositorios.

(a) (b)

Figura 36. (a) Robot Operating System (ROS). (b) ROS Kinetic Kame, distribución de ROS utilizada en el proyecto.

ROS constituye otro entorno de programación de robots entre los tantos disponibles (por poner otro ejemplo, tenemos Microsoft Robotics Developer Studio). ROS nos ofrece, no obstante, una serie de características de interés para nuestro proyecto:

- La arquitectura de ROS se basa en la modularidad, permitiendo el uso de código proveniente de distintas fuentes y usado en aplicaciones muy dispares. De esta forma, ROS ha logrado convertirse en un estándar para el desarrollo de sistemas en el mundo de la robótica.
- Su funcionamiento se basa en un entorno distribuido de procesos denominados nodos. Estos nodos forman un grafo de computación, una red de igual a igual (*peer to peer* o P2P) a través de la cual se intercambian información unos con otros.
- Nos otorga una gran flexibilidad gracias a la compatibilidad con otros entornos de programación de robots y con librerías independientes de ROS. Además, no está restringido a un único lenguaje de programación, siendo compatible tanto con C++ como con otros lenguajes más recientes (véase Python).
- Pone a nuestra disposición herramientas de depuración y de testeo a través de la librería *rostest.*
- Adecuado para sistemas de tiempo de ejecución elevados y para proyectos de desarrollo.

La distribución de ROS utilizada para este proyecto ha sido *Kinetic Kame*, lanzada para Ubuntu 16.04 Xenial. Esta distribución incluye, además de la instalación básica de ROS, las herramientas de visualización de datos rqt y *Rviz*, librerías genéricas de robots y paquetes para navegación y percepción en 2D y 3D.

3.1.1 Fundamentos de ROS

Para entender el proceso de adquisición y lectura de los datos del experimento, es preciso profundizar un poco más sobre el funcionamiento de ROS. Como se ha comentado, ROS funciona en base a un grafo computacional en forma de red de igual a igual (P2P). Este grafo está constituido por diferentes procesos que intercambian información unos con otros, y en él intervienen una serie de elementos que constituyen el alma de ROS. A continuación, se describen someramente estos elementos:

- *Nodes* (Nodos): los nodos del grafo son ejecutables que llevan a cabo una tarea concreta. A diferencia de un sistema monolítico, esta distribución de procesos nos proporciona una gran flexibilidad y simplifica el desarrollo de código. Los nodos están en continuo intercambio de información unos con otros y con el *Parameter Server*. Esta información se transfiere mediante mensajes, que dan la estructura a los datos, y a través de *topics* y/o *services*. En nuestro caso, cada sensor dispone de un nodo de adquisición de datos o serialización que se encarga de interpretar los datos recibidos y publicar los mensajes. Por otro lado, otro nodo se encargará de almacenar los mensajes subscritos en un archivo *bag* para su posterior procesamiento.
- *Messages* (Mensajes): los mensajes son paquetes de datos estructurados que los nodos usan para intercambiar información unos con otros. Los mensajes se transmiten a través de *topics* o *services*, que son los medios de intercambio de mensajes. La estructura del mensaje se la otorgan una serie de campos a los que se les asigna un tipo determinado, permitiéndonos configurar nuestros propios *message types.* En nuestro caso, tenemos dos tipos de mensajes: “bag_data_extraction/Spd”, creado manualmente para las medidas del espectro del campo magnético, y “sensor_msgs/PointCloud2”, mensaje estandarizado para las nubes de puntos.
- *Topics* (Temas): los temas son los *buses* o canales de intercambio de información que usan los nodos para transmitir los mensajes mediante rutinas de publicación y subscripción. Los nodos publican o se subscriben a un cierto tema sin conocer de la existencia de otros nodos. Esto convierte a los temas en verdaderos canales de transmisión de un tipo de información determinada, desacoplando los nodos entre sí. Los protocolos de transmisión pueden ser tanto TCP/IP como UDP. En nuestro caso, cada nodo de

adquisición de datos publica a un tema distinto, mientras que un único nodo de almacenamiento está subscrito a ambos temas.

- *Services* (Servicios): los servicios sustituyen a los temas cuando se requiere un intercambio de mensajes entre nodos. Implementan un RPC (*Remote Procedure Call*), en el que un nodo envía una petición a otro nodo (para el cual se ha definido el servicio) y este le devuelve una respuesta. El servicio por tanto lo constituye un par de mensajes: la petición (*request*) y la respuesta (*reply*).
- *Parameter Server* (Servidor de Parámetros): el Servidor de Parámetros es un diccionario de variables compartido entre los nodos de la red. Se usa para centralizar el acceso a los parámetros del sistema, tales como configuraciones de los sensores, durante el tiempo de ejecución.
- *Master* (Master): el Master es el nodo central de toda esta red. Es el encargado de controlar las interacciones entre el resto de nodos, proporcionando a la red el registro de nombres que identifica cada uno de sus elementos y los organiza jerárquicamente (es decir, cada elemento de la red como mensajes, temas, servicios y parámetros son identificados con un nombre que es registrado en el Master).

Esta descentralización es la que convierte a ROS en una herramienta tan potente y en un estándar para la programación de robots a nivel mundial. Finalmente, en la Figura 37 se muestra un esquema del funcionamiento de ROS particularizado para nuestro proyecto.

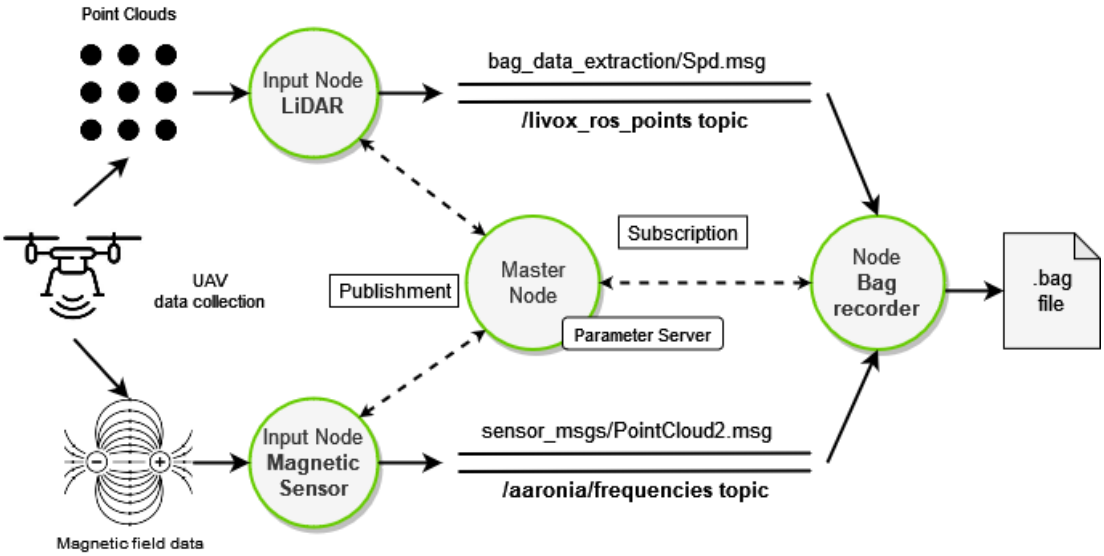


Figura 37. Grafo de procesos de ROS particularizado para la adquisición de datos durante los vuelos experimentales. Los distintos nodos del entorno ROS se comunican entre sí mediante subscripción y publicación en temas.

3.1.2 rqt_bag

La extensión *bag* es usada por ROS para los archivos encargados del almacenamiento de los mensajes, los cuales contienen los datos adquiridos por los sensores. Estos mensajes llegan al *bag recorder* (que almacena la información en el archivo *bag* siguiendo una estructura determinada) a través de los temas publicados por los nodos de la red.

Durante el vuelo de grabación, los sensores de nuestro sistema UAV se han encargado de transmitir a través de un bus de datos las medidas obtenidas para su publicación y registro en el entorno ROS, obteniendo así un archivo *bag* final. El archivo contendrá los mensajes de las medidas organizados en dos temas: uno para el espectro magnético del sensor de campo magnético, y otro para las nubes de puntos del sensor LiDAR.

El primer paso consiste en leer este archivo *bag* y visualizar las medidas extraídas. Por suerte, ROS dispone de

las interfaces gráficas *rqt_bag* para inspeccionar y publicar el contenido de archivos bag, y *Rviz* para visualizar nubes de puntos.

Rqt_bag permite abrir un archivo *bag* e inspeccionar su contenido. En su interfaz (ver Figura 38) pueden observarse los distintos temas que han sido escuchados y, en una línea temporal, los mensajes asociados a cada tema. Estos mensajes pueden reproducirse, como si de una pista de audio se tratase, para su visualización en la propia herramienta o para su publicación a través del tema correspondiente. Esto último es de gran utilidad, ya que *rqt_bag* no incorpora de un visualizador de nubes de puntos. Para ello usaremos la interfaz *Rviz*.

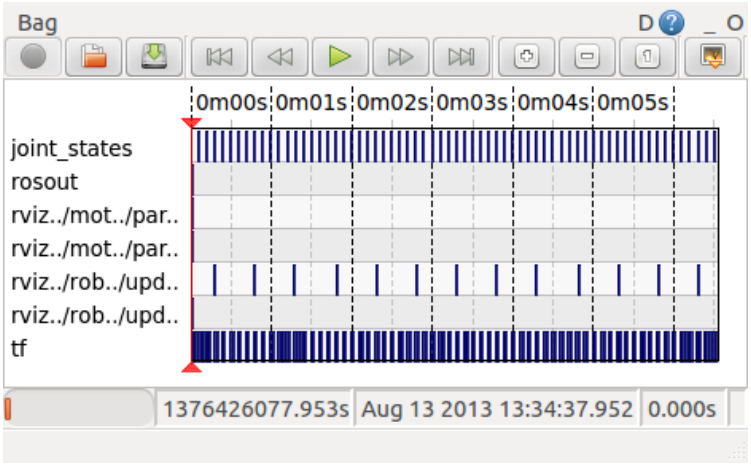


Figura 38. Interfaz de rqt_bag. En una línea temporal se marcan los mensajes correspondientes a cada tema. En la parte superior se pueden apreciar las herramientas para abrir y exportar archivos bag.

3.1.3 RViz

RViz es la herramienta 3D utilizada en ROS para la visualización del entorno del robot. En ella se nos permite definir distintos *displays* o visualizaciones que escucharán un tema determinado y representarán gráficamente el contenido de los mensajes que sean publicados en dicho tema. Las visualizaciones pueden ser de todo tipo, incluyendo modelos del robot, ejes, cámaras, mapas binarios, vectores y todo tipo de marcadores. Además, los sistemas de coordenadas locales (*frames)* asociados a cada mensaje se transforman automáticamente al sistema fijo (*fixed frame)*. Para ello, es necesario definir y publicar esta transformación mediante el paquete *tf* para que le llegue a *RViz*.

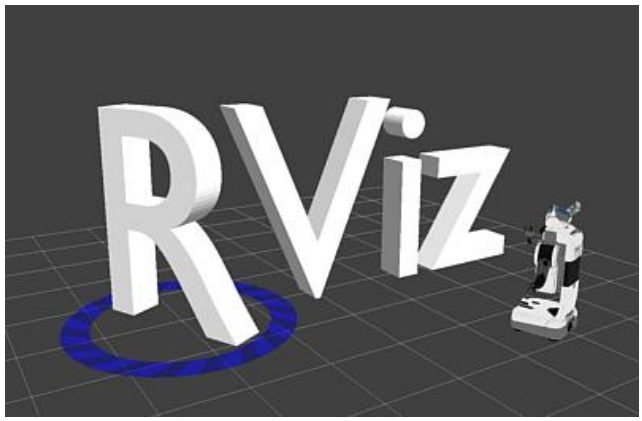


Figura 39. RViz, herramienta de visualización 3D de ROS.

Por supuesto, *RViz* también permite la visualización de nubes de puntos, a través de mensajes tipo *PointCloud2*. La estructura de datos de este mensaje se verá con detalle en el siguiente capítulo, pero en esencia, esta estructura recoge los datos de los distintos puntos de la nube incluyendo, además de sus coordenadas XYZ, otros campos como puede ser la reflectividad. Esta será la herramienta que se usará para visualizar tanto las nubes de puntos de partida como los resultados de los distintos pasos del algoritmo de procesamiento de nubes. Este es un aspecto clave, pues nos permitirá comprobar los resultados, facilitando la detección de errores y el ajuste de los parámetros del algoritmo.

3.1.4 ROS APIs

ROS pone a nuestra disposición distintas APIs (*Application User Interfaces*), es decir, librerías que nos permiten como desarrolladores implementar las distintas funcionalidades de ROS a través de código fuente. Estas librerías vienen escritas en distintos lenguajes de programación, incluyendo *C++* y *Python*. A continuación, se describen brevemente cuales han sido las librerías de interés para el proyecto:

- ***roscpp/rospy**:* escritas en *C++* y *Python* respectivamente, son las librerías que implementan la funcionalidad más básica de ROS, incluyendo la creación de nodos y la publicación y/o subscripción a temas.
- ***rosbag:*** implementa toda la interfaz necesaria para comunicarnos con los archivos *bag*, tanto en lectura como en escritura.
- ***common_msgs:*** define las clases correspondientes a cada tipo de mensaje de ROS para su uso en otras librerías. Los mensajes obtenidos por el sensor LiDAR se corresponden con la clase *PointCloud2*, que se incluye dentro de la categoría *sensor_msgs*. De este modo, y a través de *rosbag*, podemos acceder a las nubes de puntos como objetos de la clase *PointCloud2*, la cual da forma a la estructura de datos que conforman las nubes y nos permite manipularlas.
- ***tf:*** nos permite definir las transformaciones entre *frames* o sistemas de coordenadas en los que se han obtenido los mensajes. Cada mensaje almacena en su cabecero (*header)* la identificación del sistema de referencia en el que se ha obtenido dicho mensaje.
- ***pcl_ros:*** sirve como nexo de conexión entre las librerías propias de ROS y la librería *pcl*. Aquí se incluye la librería *pcl_conversions*, que nos permite transformar la estructura de datos *PointCloud* de un tipo a otro (de ROS a PCL y viceversa).
- ***pcl:*** PCL (*Point Cloud Library*) es una librería independiente integrada en ROS y utilizada para el procesamiento de nubes de puntos. Implementa los algoritmos más actuales de filtrado, estimación de características, segmentación y ajuste de modelos. Por tanto, esta librería es el eje en el que gira la mayor parte del trabajo realizado, y sus algoritmos los que serán explicados con detalle en este capítulo. En la versión de ROS utilizada, esta librería sólo está disponible en *C++*.

Como se ha comentado anteriormente, estas librerías son las que se han usado para desarrollar el algoritmo de procesamiento de nubes de puntos. Puesto que para esta versión de ROS la librería PCL sólo está disponible en *C++*, este es el lenguaje elegido el desarrollo del código. Aunque podría haberse planteado el exportar los datos de las nubes de puntos a otra plataforma para utilizar un lenguaje más sencillo como *Python*, esto supondría salirnos del entorno ROS. La idea del proyecto reside en exprimir al máximo las capacidades de ROS y desarrollar esta tarea íntegramente en su entorno de una manera eficiente, por lo que esta opción se ha descartado.

Por otro lado, el motivo por el cual esta librería está escrita en *C++* está claro: dicho lenguaje resulta mucho más eficiente, y los algoritmos que tratan con nubes densas requieren de esta velocidad de computación adicional, especialmente si se desean implementar a tiempo real.

Eso sí, tras la segmentación de las líneas los datos son exportados para su postprocesamiento externo a ROS, pues el uso de lenguajes más amenos para el programador como *Python* está más que justificado.

3.2 Algoritmos de procesamiento de nubes de puntos

En este apartado se describen con detalle los algoritmos utilizados durante la tarea de procesamiento de nubes de puntos. Estos vienen implementados a través de la librería PCL (*Point Cloud Library*) y escritos en el lenguaje de programación *C++*.

De forma resumida, el algoritmo de procesamiento de nubes de puntos consta de tres bloques: un bloque de filtrado de las nubes en crudo, un bloque de concatenación para el aumento artificial de la resolución del sensor LiDAR y, por último, un bloque de segmentación mediante el cual se identifican las líneas y se obtienen los coeficientes de sus modelos. El desarrollo completo del algoritmo junto con sus diagramas de flujo se deja para el apartado 3.2.

3.2.1 Point Cloud Library (PCL)

Point Cloud Library (PCL) es una librería independiente, multiplataforma y de uso libre para procesamiento de imágenes y nubes de puntos en 2D y 3D. PCL surge como una respuesta al continuo desarrollo de sensores 3D como *Kinect* de *Microsoft*, cada vez más accesibles para el usuario de a pie.

Figura 40. PCL, librería para el procesamiento de nubes de puntos.

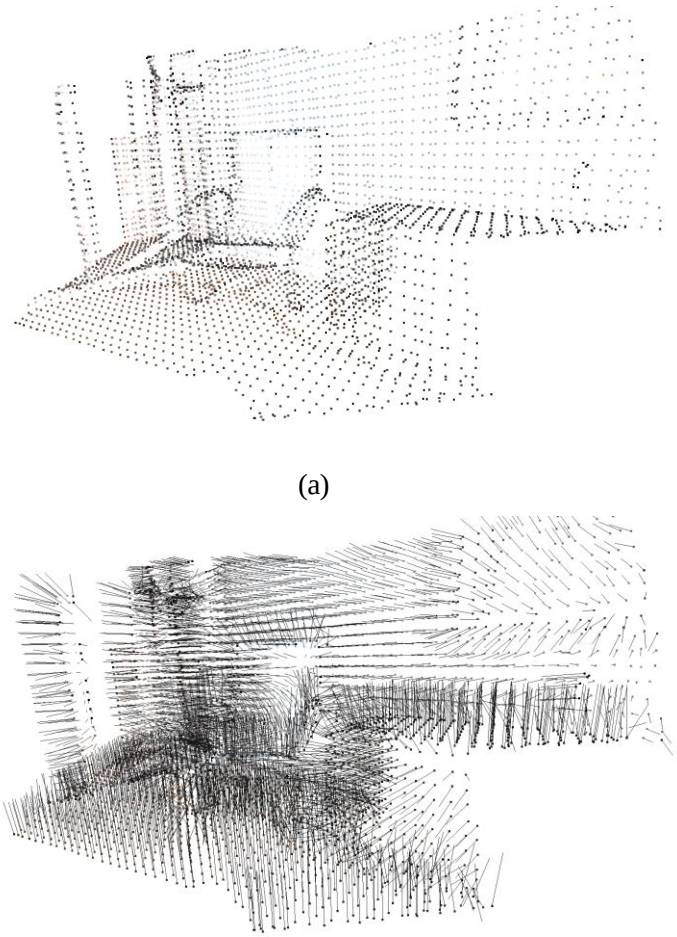
Tal y como vienen definidas por la librería, las nubes de puntos son estructuras de datos que representan una colección de puntos n-dimensionales, capaces de almacenar información adicional además de las coordenadas de cada punto, y que cumplen una función fundamental dentro de los sistemas de percepción de los robots.

La librería consta de algoritmos de diversa índole que forman parte del estado del arte del procesamiento de nubes de puntos. Estos algoritmos se implementan a través de clases que integran toda la secuencia lógica o *pipeline* del mismo.

Con el objetivo de aumentar la flexibilidad y reducir el tamaño de los sistemas, PCL divide sus funcionalidades en distintos módulos independientes entre sí. Cada uno de estos módulos se ajusta a un tipo de tarea concreta, y se describen a continuación:

- ***pcl_filters:*** implementa filtros para el *downsampling*, supresión de ruido, descarte de *outliers* y otro tipo de operaciones como proyecciones. Normalmente, estas operaciones constituyen el primer paso de cualquier algoritmo de procesamiento, pues nos permite simplificar la nube de puntos y reducir el coste computacional.
- ***pcl_features:*** implementa el cálculo de las características locales a cada punto, las denominadas *point features*. La mayoría de estos métodos se basan en tomar los *k* puntos más cercanos para formar un vector descriptor, el cual caracteriza localmente la geometría en torno al punto en cuestión. Los descriptores más básicos son los valores de la curvatura local y la normal, los cuales se obtienen a partir de la solución en autovalores y autovectores de un PCA *(Principal Component Analysis)*. Cuando estos descriptores no son suficientemente discriminativos, existen otros métodos que se basan en usar como descriptor un histograma creado a partir de *features* obtenidas de las normales entre cada par de puntos

del entorno. Ejemplos de estos descriptores son: PFH (*Point Feature Histogram*), FPFH (*Fast Point Feature Histogram*) y VFH (*Viewpoint Feature Histogram*), aunque hay muchos otros.

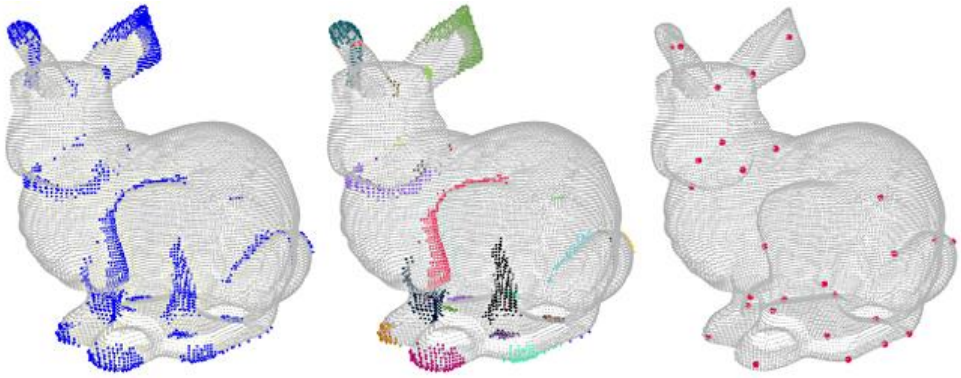


(a)

(b)

Figura 41. (a) Nube de puntos pertenecientes a la escena de una oficina y (b) estimación de las normales para dicha nube. Figuras extraídas de [35].

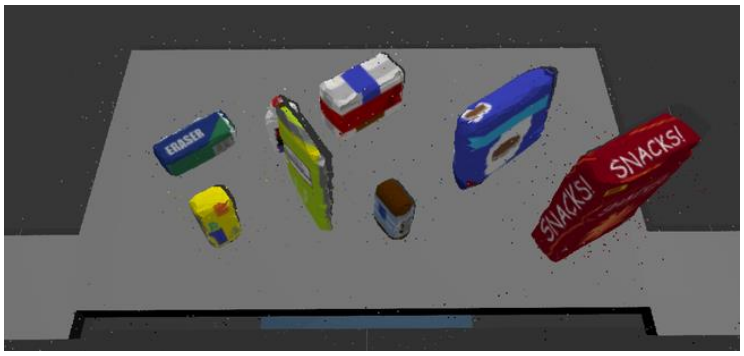
- ***pcl_keypoints:*** implementa los métodos de detección de *keypoints* o puntos de interés, que son puntos representativos de la nube de puntos con características locales distinguibles y que son de gran ayuda para representar de forma compacta la información global de la nube de puntos. Normalmente se usan como paso previo a la extracción de características para segmentación. La librería incluye diversos detectores de *keypoints* como son: Harris, AGAST, BRISK, NARF, etc.



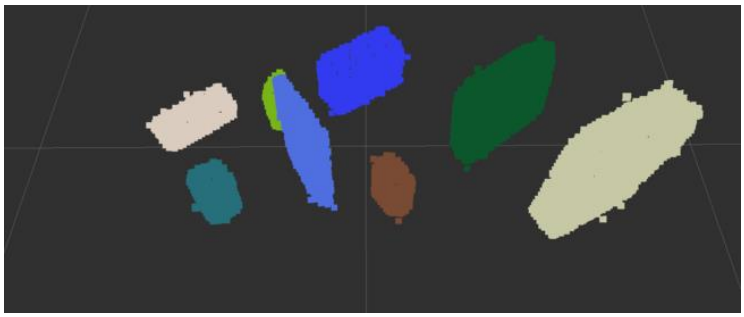
(a) (b) (c)

Figura 42. Proceso de extracción de puntos de interés sobre una nube determinada (a) y (b) hasta obtener los *keypoints* finales (c). Figuras extraídas de [36].

- ***pcl_registration:*** implementa las distintas técnicas de registro de nubes de puntos, que es la tarea de corresponder los puntos de dos nubes distintas pertenecientes a la misma escena. Esto nos permite obtener además la posición y el ángulo del sensor respecto a unos ejes fijos a la escena. La idea general detrás del registro de un par de nubes de puntos consiste en obtener esta correspondencia para una serie de puntos de interés de cada nube (los *keypoints* descritos previamente), y así obtener una matriz de transformación rígida entre ambas nubes. A modo de ejemplo, se incluye el método de registro ICP (*Iterative Closest Point*).
- ***pcl_kdtree:*** implementa los *KdTrees*, que son unas estructuras de datos en forma de árbol binario utilizadas para organizar los puntos pertenecientes a un espacio genérico de *k* dimensiones. Esta partición es utilizada para ejecutar de manera eficiente los algoritmos de búsqueda por rango (*radius search*) y *k* puntos más cercanos (*k-nearest search*).
- ***pcl_octree:*** implementa los *Octrees*, unas estructuras de datos en forma de árbol en la que cada nodo tiene exactamente 8 nodos hijo o ninguno. Esta estructura está pensada para particionar el espacio 3D de manera recursiva en octantes del mismo tamaño. Si un octante contiene más de un punto, este sigue dividiéndose hasta alcanzar el octante o *voxel* de tamaño mínimo permitido o hasta que no contenga ningún punto. Al igual que los *KdTrees*, estas estructuras permiten ejecutar de manera eficiente algoritmos de búsqueda.
- ***pcl_segmentation:*** implementa los algoritmos de segmentación, que nos permiten agrupar los puntos de la nube en distintos *clusters* o grupos. Los distintos métodos disponibles para realizar la segmentación difieren en el criterio elegido, ya sea basándose en la distancia a los puntos más cercanos (*Euclidean clustering*) o cambios bruscos de la normal (*Region growing*).



(a)



(b)

Figura 43. (a) Escena en la que se pueden apreciar distintos objectos. (b) Segmentación de la escena en distintos *clusters*. Figuras extraídas de [37].

- ***pcl_sample_consensus:*** implementa los métodos de segmentación SAC (*Sample Consensus*), que llevan a cabo la segmentación ajustando los puntos de la nube a modelos como líneas, planos, cilindros o esferas. Existen diferentes métodos para encontrar (si los hay) los puntos que se ajustan a un modelo determinado, aunque sin duda uno de los más robustos es el RANSAC (*Random Sample Consensus*).
- ***pcl_surface:*** implementa herramientas para la reconstrucción de superficies cuando las nubes de puntos se ven afectadas por ruido o cuando existen ligeras desviaciones en el registro de nubes. Estas técnicas pueden consistir en: un suavizado de las superficies mediante remuestreo, como ocurre en el MLS (*Moving Least Squares*); en un mallado, esto es, una representación de la superficie mediante elementos como pueden ser traingulares; o bien en obtener la envolvente de una serie de puntos, esto es, una *convex hull*.

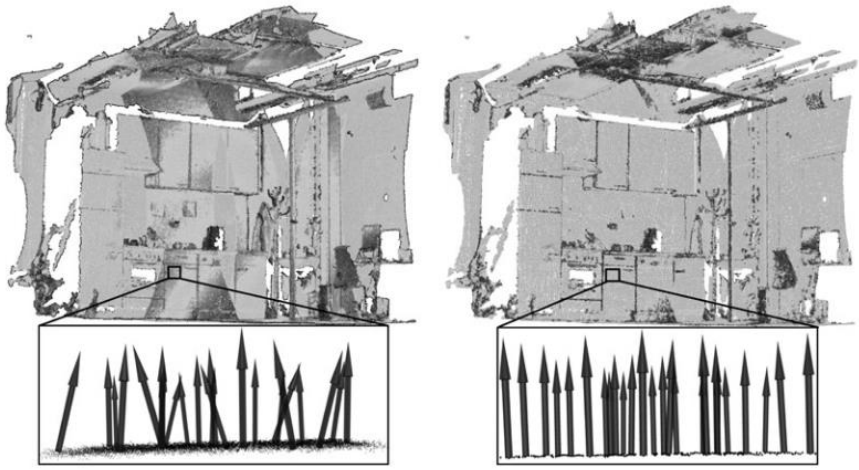


Figura 44. Aplicación del método MLS para suavizar la nube de puntos y mejorar la extracción de las normales. Figura extraída de [38].

- ***pcl_range_image:*** implementa las herramientas para trabajar con nubes de puntos organizadas, es decir, nubes organizadas en una matriz 2D denominada mapa de profundidad en los que cada píxel almacena la información de la distancia desde el origen del sensor.
- ***pcl_recognition:*** implementa algoritmos de reconocimiento de objetos, clasificándolos en distintas clases. Los métodos se basan en la generalización de la *transformada de Hough*, que nos permite realizar una votación en el hiperespacio de las *features* cada punto.

- ***pcl_io:*** implementa las librerías para la lectura y escritura de archivos PCD (*Point Cloud Data*), el estándar utilizado por PCL.
- ***pcl_visualization:*** implementa herramientas de visualización de nubes de puntos basadas en VTK (*The Visualization Toolkit*).
- ***pcl_common:*** es la librería base de PCL que contiene las definiciones de las clases primitivas (estructuras de las nubes de puntos, clases de puntos, normales, colores RGB y un largo etc.) y funciones básicas (cálculo de distancias, medias, covarianzas, transformaciones, etc.)

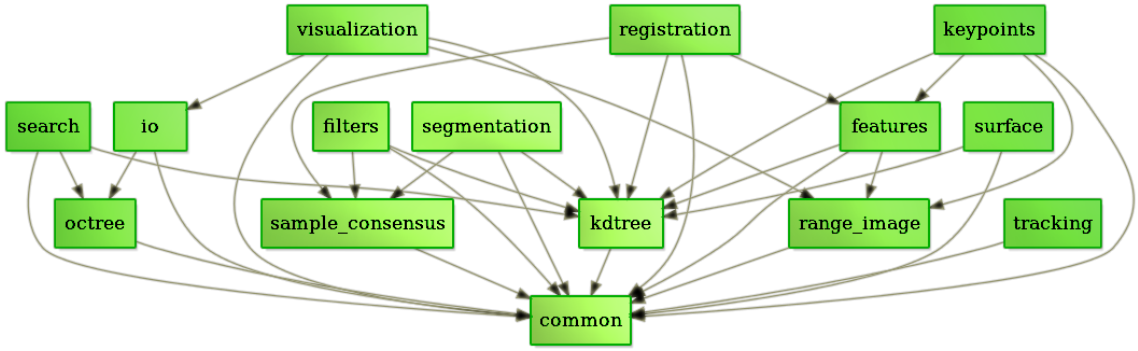


Figura 45. Diagrama de dependencias de la librería PCL.

A su vez, PCL se apoya en otras librerías de terceras partes, como son Eigen (operaciones matriciales), VTK (para el módulo de visualización), Boost (para el uso eficiente de la memoria mediante punteros compartidos), FLANN (para los algoritmos de búsqueda) y otros. Para más información sobre PCL y sobre los algoritmos que implementa, consultar [39] y [40].

A continuación, se van a explicar con más detalle los algoritmos de PCL utilizados en el código para el procesamiento de las nubes de puntos.

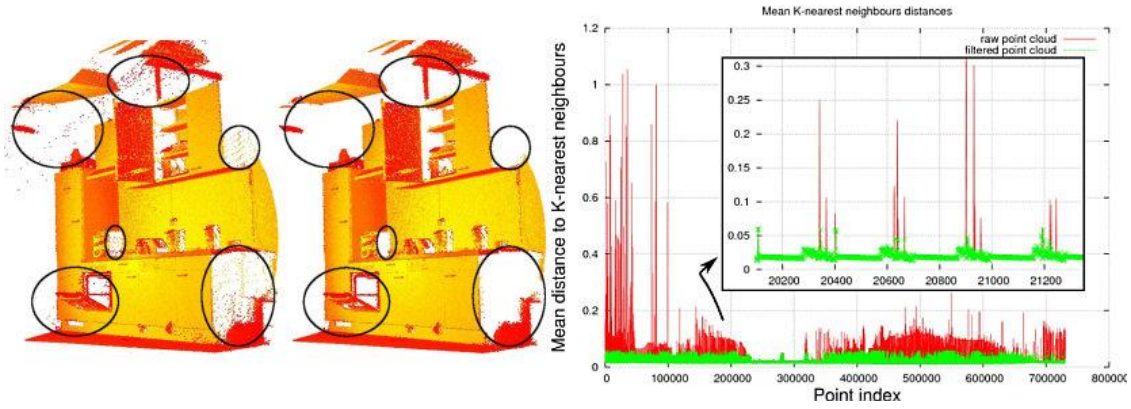
3.2.2 Statistical outlier removal

Perteneciente al módulo de filtrado, el *Statistical outlier removal* se trata de un algoritmo de *downsampling*, esto es, de simplificación de la nube de puntos. Para ello, lleva a cabo un análisis estadístico de la distancia entre cada punto de la nube, a partir del cual decide los puntos que deben ser descartados. El algoritmo se describe a continuación:

- Sea P la nube de puntos inicial, para cada punto $\boldsymbol{p_i} \in P$:
  - Obtener el subconjunto $P_i^k \subseteq P$ conformado por los k puntos más cercanos a $\boldsymbol{p_i}$.
  - Calcular la media $\overline{d_i}$ de las distancias a cada punto $d_{ik} = |\boldsymbol{p_i} - \boldsymbol{p_k}|$.
- Suponiendo una distribución Gaussiana, calcular la media μ y la desviación típica σ de las distancias anteriores $\overline{d_i}$.
- Eliminar del conjunto P los puntos $\{\boldsymbol{p_i} \in P \mid \overline{d_i} \notin [\mu - \alpha\sigma, \mu + \alpha\sigma]\}$, siendo α un parámetro determinado.

Como puede verse, el algoritmo eliminará de la nube de puntos aquellos puntos que se sitúen en zonas con una densidad alejada de los valores medios. Esto es de gran utilidad para lidiar con nubes afectadas por ruido, permitiendo no sólo eliminar *outliers* dispersos, sino además obtener con mayor presión las características

locales en cada punto.



(a) (b)

Figura 46. (a) Antes y después de la aplicación del algoritmo *Statistical outlier removal* a una nube de puntos. En (b) pueden apreciarse las estadísticas de la nube de puntos antes (en rojo) y después (en verde) de aplicar el algoritmo. Cada entrada corresponde a un punto distinto. Figuras extraídas de [38].

3.2.3 VoxelGrid downsampling

Al igual que el algoritmo anterior, se trata de otro algoritmo de *downsampling* para simplificar la nube de puntos. En este caso, el espacio 3D queda subdividido a través de un mallado de *voxels*. Un *voxel* no es más que una celda unitaria en forma de hexaedro que nos permite discretizar el espacio tridimensional. Son estas divisiones las que nos permiten simplificar los puntos contenidos en cada uno de los *voxels* por un único punto localizado en su centroide. El algoritmo se describe a continuación:

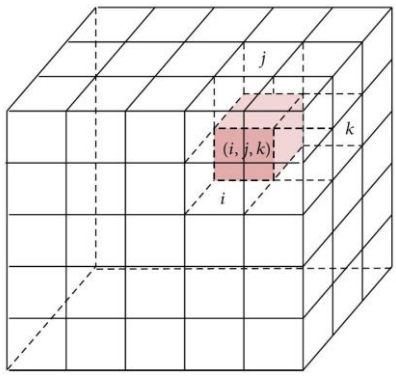
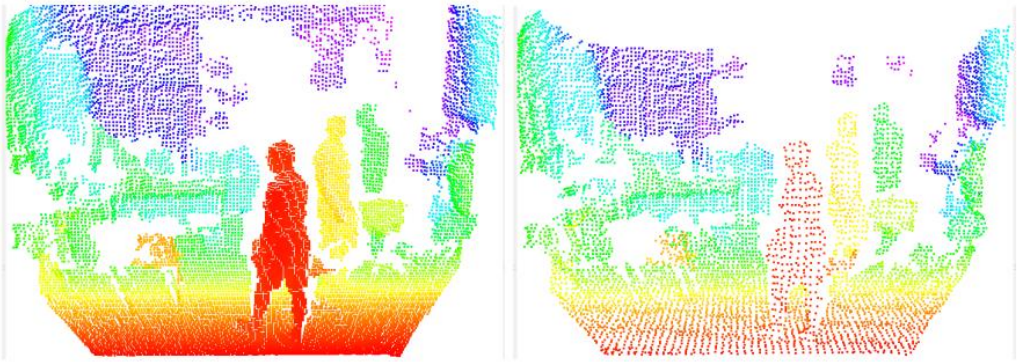


Figura 47. Mallado 3D mediante *Voxels*, paso requerido para aplicar el algoritmo *VoxelGrid downsampling*.

- Dado un tamaño de *voxel* o *leaf size* de $v_x \times v_y \times v_z$, determinar el mínimo volumen V conformado por los *voxels* v_{ijk} de dicho tamaño que encierra la nube de puntos original.
- Para cada punto $\boldsymbol{p_l} \in P$, determinar el *voxel* al que debe ser asignado: $ijk \mid v_{ijk} \in V \wedge \boldsymbol{p_l} \in v_{ijk}$.
- Para cada *voxel*, calcular el centroide de los puntos contenidos en él: $\overline{\boldsymbol{p_l}} \mid \boldsymbol{p_l} \in v_{ijk}$.

Este método permite simplificar de manera simple la nube de puntos imponiendo una densidad de puntos determinada. Además, podemos ajustar el tamaño del *voxel* como queramos, en el caso de que la resolución a lo largo de una dimensión sea más importante que en las demás. Al usar el centroide en cada *voxel* somos capaces de ajustarnos con más precisión a la nube de puntos original sin reducir el tamaño del mallado. Sin embargo, hay que ser conscientes de que este método no conserva ningún punto de la nube original, por lo que una incorrecta selección de los parámetros puede dar lugar a una gran pérdida de información contenida en la geometría original.



(a) (b)

Figura 48. (a) Escena y (b) resultado tras aplicar el VoxelGrid downsampling.

3.2.4 KdTree search

Este constituye uno de los algoritmos de búsqueda eficiente de puntos próximos disponibles en PCL, junto al *Octree search*. Esta búsqueda de puntos cercanos a un punto dado puede entenderse de dos formas distintas: o bien se buscan los k puntos más cercanos al punto en cuestión (k-NN o *k nearest neighbour search*), o bien se buscan todos los puntos situados a una cierta distancia r (*radius search*).

Como se ha comentado anteriormente, este algoritmo de búsqueda se basa en el árbol binario denominado *KdTree*. Esta estructura es ampliamente utilizada dentro de la ciencia de datos para organizar una serie de puntos pertenecientes a un espacio de k dimensiones (de ahí el nombre).

En este árbol, cada uno de los nodos está asignado a un punto en concreto de la nube. El punto elegido es el situado en la mediana de las coordenadas de los puntos correspondientes al eje seleccionado para la división. Para dimensiones $k > 1$, el eje seleccionado para la división va alternándose. De esta forma, cada nodo padre (*parent node*) divide a sus nodos hijos a través de un hiperplano. Para visualizar este árbol, se recomienda observar la Figura 49 y la Figura 50. Esto se repite hasta que no quedan más puntos para realizar la partición. Los nodos de más bajo nivel en cada ramificación se denominan nodos hoja o *leaf nodes*. A continuación, se explica el algoritmo estandarizado para la formación del árbol:

- Sea $Q_{ij} \subseteq P$ el subconjunto de puntos pertenecientes al nodo i del nivel j (N_{ij}), inicializamos el primer nivel con la nube de puntos completa.
- Iterar nivel a nivel en j y ejecutar:
    - Alternar de eje k para realizar la partición en el nivel actual.
    - Obtener la mediana respecto de la coordenada x_k de los puntos Q_{ij} pertenecientes a cada nodo $N_{ij} \forall i$.
    - Asignar el punto $\boldsymbol{p_{ij}}$ situado en la mediana a cada nodo $N_{ij} \forall i$.

- Usando los puntos $\boldsymbol{p_{ij}}$, formar los subconjuntos Q_{ij+1} mediante una partición binaria usando el hiperplano perpendicular al eje x_k y que pasa por $\boldsymbol{p_{ij}}$.
- Cuando se cumpla $Q_{ij+1} = \emptyset \; \forall i$, terminar la iteración.

En la Figura 49 y la Figura 50 se muestran visualmente las particiones mediante hiperplanos para los casos 2D y 3D respectivamente, así como el árbol resultante para el caso 2D. Esto se extrapola de forma análoga para cualquier número de dimensiones.

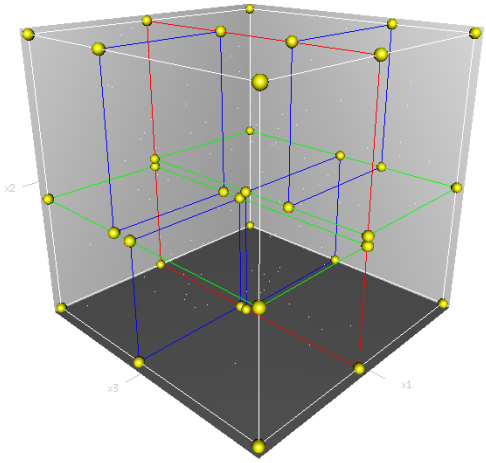
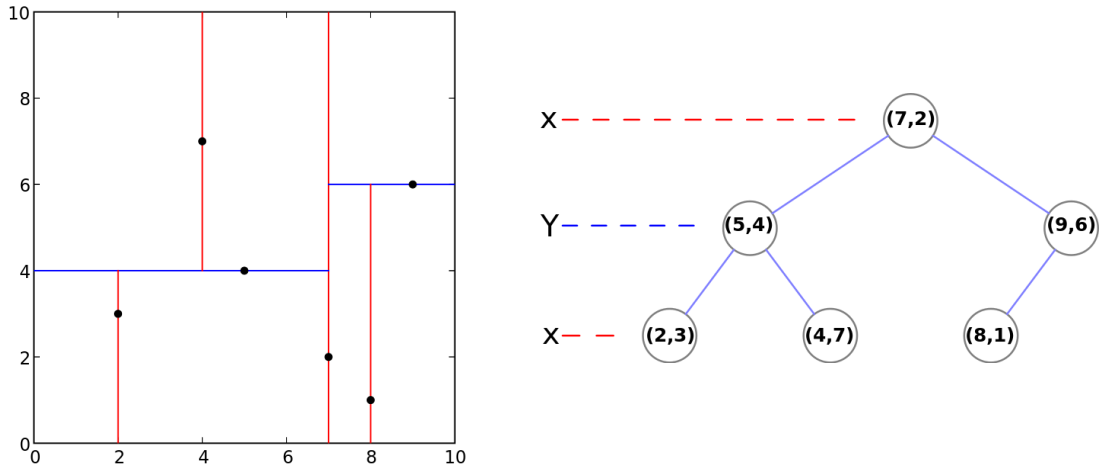


Figura 49. Particiones del árbol *KdTree* visualizado para 3 dimensiones. Puede observarse como los hiperplanos que dividen sucesivamente las regiones se van alternando en cada nivel. Cada color corresponde a un hiperplano en un eje diferente.



(a) (b)

Figura 50. (a) Particiones del árbol KdTree visualizado para 2 dimensiones y (b) el propio árbol.

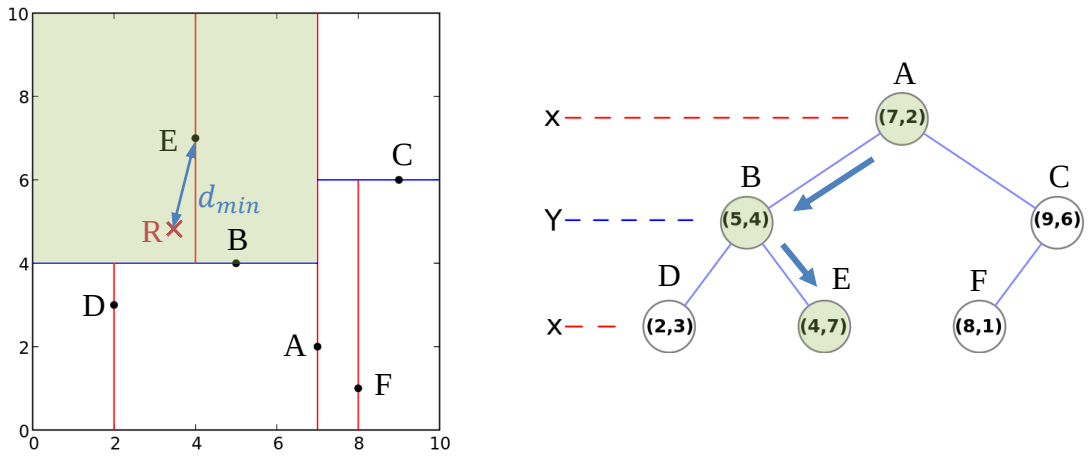
El algoritmo descrito para la creación del árbol da lugar a lo que se denomina como árbol equilibrado, esto es, un árbol en el que todos los nodos hoja están al mismo nivel. Aunque no todos los algoritmos están optimizados para árboles equilibrados, generalmente son más eficientes, y en algunas ocasiones el conveniente equilibrar

árboles cuando se añaden o eliminan una cantidad considerable de nodos.

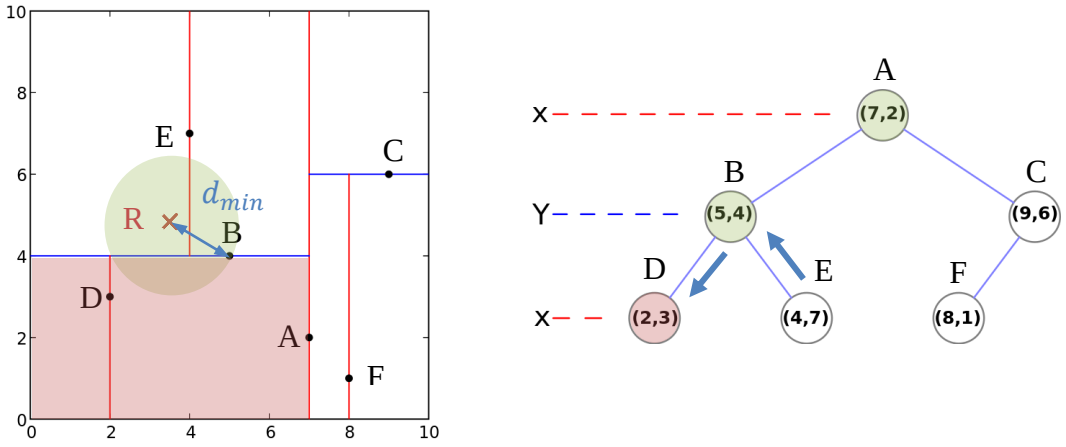
Los algoritmos de búsqueda hacen uso de las propiedades de este árbol para llevar a cabo su cometido de manera eficiente, descartando una gran cantidad de candidatos durante la búsqueda. Aunque cada implementación tiene sus particularidades, todas ellas se basan en la particularización del algoritmo NN (*Nearest Neighbour search*), que se aplica sobre nubes de puntos organizadas mediante *KdTrees*. Este algoritmo se describe a continuación:

- Sea $\boldsymbol{p_0}$ el punto de referencia de y N_{ij} el nodo i en el nivel j en el que nos encontramos, inicializamos para el nodo raíz.
- Inicializamos $d_{min} \leftarrow \infty$.
- Búsqueda hacia adelante. Localizamos el nodo hoja en el que se encuentra nuestro punto. Para ello:
    - Avanzamos por la rama en la que se encuentra nuestro punto, comparando las coordenadas x_{0k} y x_k del punto asignado al nodo actual y eje k. Esto es, se comprueba si $\boldsymbol{p_0}$ está a un lado o al otro del hiperplano definido por N_{ij}.
    - Repetimos hasta llegar al nodo hoja.
- Calculamos distancia $d = |\boldsymbol{p_0} - \boldsymbol{p_{leaf}}|$ y asignamos $d_{min} \leftarrow d$ si $d < d_{min}$.
- Búsqueda hacia atrás.
    - Retrocedemos hacia el nodo padre.
    - Asignamos $d_{min} \leftarrow d$ si $d < d_{min}$.
    - Si la rama alternativa aún no ha sido visitada:
        - Comprobar si el nodo de la rama alternativa puede tener albergar candidatos. Para ello, se crea una hiperesfera de radio d_{min} y centrada en $\boldsymbol{p_0}$.
        - Si dicha esfera penetra la región delimitada por los hiperplanos de dicho nodo, ir a la búsqueda hacia adelante.
    - Repetir hasta regresar al nodo raíz.

Para entender mejor este algoritmo, ver las siguientes figuras:



(a)



(b)

Figura 51. (a) Búsqueda hacia adelante del algoritmo NN *(Nearest Neighbour search*) para el punto de referencia R. Tras avanzar nivel a nivel por el árbol se ha determinado que el primer candidato es el punto E, ya que el punto R queda en la región que divide E. La distancia entre ambos puntos es ahora la distancia mínima obtenida. (b) Búsqueda hacia atrás. En primer lugar, el punto B pasa a ser el nuevo candidato, al encontrarse más cerca que el punto E. Puesto que la hiperesfera interseca con la sección roja sombreada, se debe descender por la rama hasta llegar a un nuevo nodo hijo.

3.2.5 Octree search

Octree search constituye otro de los algoritmos de búsqueda disponibles en PCL junto al *KdTree search.* Al igual que este, hace uso de una estructura de datos en forma de árbol para realizar la búsqueda de forma eficiente, aunque difieren en el tipo de árbol utilizado. Los *Octrees*, como su nombre indica, son estructuras en forma de árbol en las que de cada nodo que no sea nodo hoja (es decir, nodos no últimos) parten exactamente 8 nodos.

Esta propiedad hace que uso sea especialmente indicado para organizar nubes de puntos 3D, al poder dividir un hexaedro en 8 octantes de igual tamaño. Para una resolución dada, el algoritmo creará una malla con esta resolución mediante celdas unitarias denominadas *voxels*. Los puntos se asignarán al *voxel* correspondiente, pudiendo contener cada uno de ellos más de un punto. El resultado final es un árbol en el que los nodos hoja contienen los índices de los puntos contenidos en dicho *voxel*.

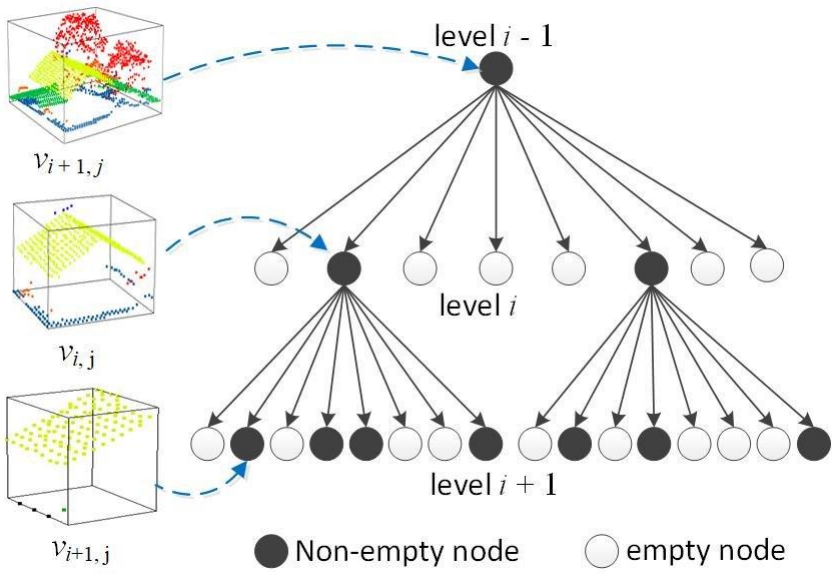


Figura 52. Creación de una estructura de árbol Octree para una escena dada.

El uso de este tipo de árbol está muy extendido, especialmente en aplicaciones de gráficos 3D. Además de su uso para algoritmos de búsqueda (que permite además el método *Neighbours within Voxel Search* respecto a la estructura *KdTree*) también es utilizado en técnicas de *downsampling*, como el algoritmo *VoxelGrid downsampling* visto en apartados anteriores, y en algoritmos de segmentación como el *Euclidean Cluster Extraction* que se verá en el apartado siguiente. Debido a su similitud con estos algoritmos, no se recogerán nuevamente los detalles para este caso.

3.2.6 Euclidean Cluster Extraction

Este algoritmo entra dentro del grupo de los algoritmos de segmentación, encargados de agrupar los puntos de nubes no organizadas en grupos o *clusters* de acuerdo a unos criterios determinados. El método *Euclidean Cluster Extraction* se trata de uno de los métodos más simples de *clustering*, basándose únicamente en la distancia entre puntos vecinos y requiriendo de muy pocos parámetros para su configuración.

Apoyándose en estructuras de árbol como los *Octrees*, este método ejecuta consecutivamente algoritmos de búsqueda NN (*Nearest Neighbour*) para agrupar entre sí puntos que se encuentran a una cierta distancia mínima. Para que dos conjuntos de puntos formen parte del mismo *cluster*, sólo es necesario que un par de puntos se encuentren lo suficientemente próximos. Esto hace que en ocasiones se agrupen cuerpos completamente dispares, siempre que existe un camino de puntos que los una. Puede ser útil aplicar filtros de *downsampling* como paso previo. El algoritmo se describe a continuación:

- Generar el *Octree* a partir de la nube de puntos original P.
- Inicializar una lista vacía C de *clusters* encontrados y un conjunto vacío de puntos $Q = \emptyset$.
- $\forall \boldsymbol{p_i} \in P$, ejecutar:
    - Añadir $\boldsymbol{p_i}$ al conjunto Q
    - $\forall \boldsymbol{p_i} \in Q$, ejecutar:
        - Obtener el conjunto P_i^k de puntos situados a una distancia r de $\boldsymbol{p_i}$.
        - Añadir a Q los puntos que estén en P, es decir, aquellos que aún no hayan sido procesados.
    - Restar el conjunto Q del conjunto P al ser puntos ya procesados.

- Añadir el conjunto Q a la lista C si Q está constituido por un mínino número de puntos n.
- Resetear $Q = \emptyset$.

Puede observarse que el conjunto de *clusters* es mutuamente excluyente, es decir, ningún punto pertenecerá a dos *clusters* distintos. Al basarse únicamente en las distancias euclídeas entre pares de puntos, este método de segmentación a pesar de ser eficiente es mucho más sensible al ruido que el bien conocido método de *k-means clustering*. Sin embargo, su uso está justificado al aplicar previamente filtros para limpiar la nube y al usarse junto con el método de segmentación RANSAC.

Este algoritmo es similar al método DBSCAN (*Density-based spatial clustering of applications with noise*), el cual es considerado un estándar entre los algoritmos de *clustering*, aunque con algunas diferencias importantes. Este último lleva a cabo la segmentación basándose en la densidad localizada en torno a cada punto, descartando puntos aislados. Así mismo, distingue entre puntos del núcleo del *cluster* (*core points*) y puntos de la frontera (*reachable points*). Al igual que el algoritmo descrito anteriormente, únicamente necesita dos parámetros: el radio de búsqueda y el mínimo número de puntos requerido en dicho radio para formar un núcleo (ver la Figura 53).

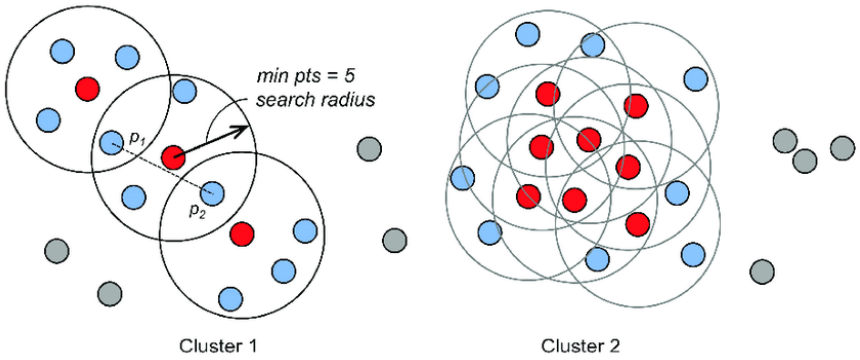


Figura 53. Algoritmo DBSCAN para la segmentación de nubes de puntos. Figura extraída de [41].

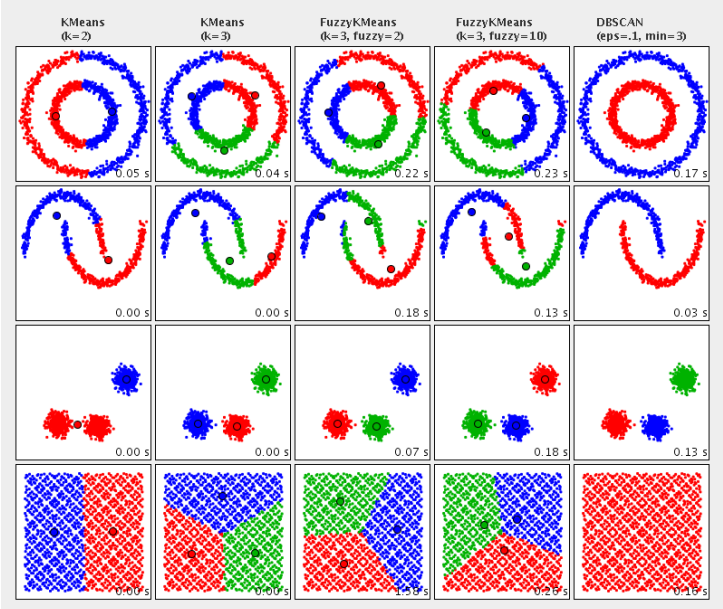


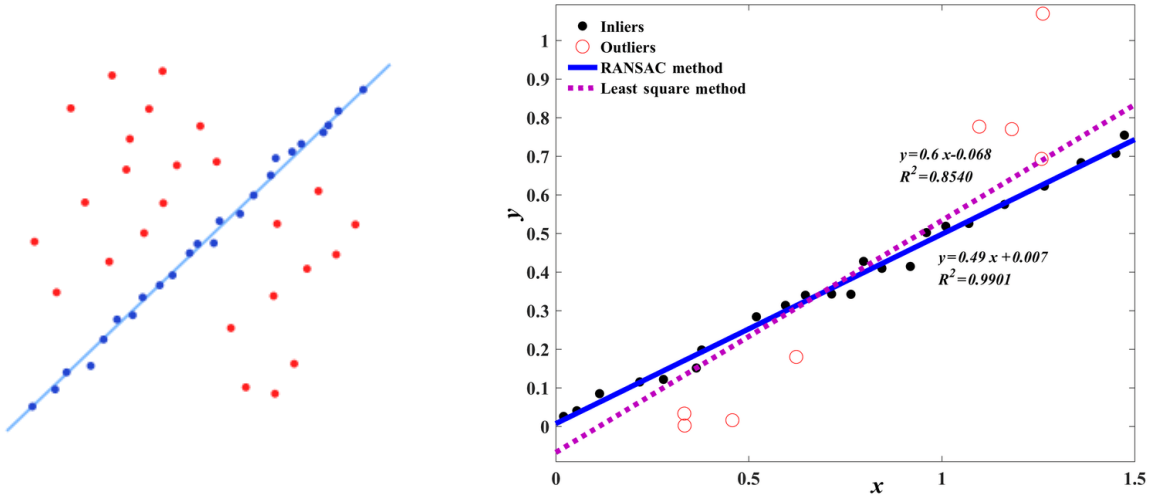
Figura 54. Resultados de la aplicación de distintos algoritmos de segmentación sobre una nube de puntos 2D. Puede verse claramente que el método DBSCAN es el más eficaz para segmentar objetos independientes.

3.2.7 Random Sample Consensus

RANSAC (*Random SAmple Consensus*) es uno de los métodos disponibles para implementar el algoritmo de segmentación SAC. Este algoritmo lleva a cabo la segmentación ajustando un tipo de modelo concreto (como pueden ser líneas, planos, cilindros o esferas) a la nube de puntos de partida, distinguiendo entre *inliers* y *outliers*.

La implementación RANSAC es un método robusto y sencillo de ajustar estos modelos en presencia de una gran cantidad de *outliers* o ruido en la nube de puntos. Consiste en un remuestreo aleatorio de la nube de puntos original, de la que se toma el mínimo número de puntos para formar el modelo. Con estos puntos se ajusta el modelo (esto es, se obtienen los parámetros del modelo) y se obtiene el número de *inliers*, dentro de una tolerancia especificada. Esto se repite un número determinado de veces, y nos quedamos con el modelo con el mayor número de *inliers*. El algoritmo se describe con detalle a continuación:

- Repetir k veces:
  - Seleccionar de manera aleatoria n puntos $\boldsymbol{p_i} \in P$, siendo n el tamaño del vector de parámetros $\boldsymbol{\phi}$ del modelo.
  - Ajustar los n puntos para obtener el vector $\boldsymbol{\phi}$ con los parámetros del modelo.
  - Obtener el número de *inliers* m de la nube que se ajustan al modelo definido por $\boldsymbol{\phi}$ con una cierta tolerancia d. Este es denominado como *consensus set*.
  - Nos quedamos con el modleo $\boldsymbol{\phi}$ tal que m sea máximo.
- Recalcular $\boldsymbol{\phi}$ utilizando los *inliers* obtenidos.



(a) (b)

Figura 55. (a) Resultado del método RANSAC para la segmentación de una línea 2D con un gran número de *outliers*. (b) Comparativa entre el método RANSAC y el LSM. Puede apreciarse como el RANSAC es más robusto ante la presencia de *outliers*.

Como puede observarse, se trata de un algoritmo muy sencillo con un importante factor de aleatoriedad. El principal inconveniente de este método es su sensibilidad ante los parámetros seleccionados. Por un lado, no existe un límite establecido sobre el número iteraciones, y la selección de este parámetro es crucial para conseguir el *trade-off* entre el coste computacional y la precisión de los modelos obtenidos. Por otro lado, la tolerancia para seleccionar los *inliers* juega también un papel fundamental, pues un valor demasiado elevado puede hacer que todos los modelos se puntúen igualmente bien, mientras que un valor demasiado bajo puede hacer que los parámetros de los modelos fluctúen considerablemente. Para seleccionar el máximo número de

iteraciones se debe recurrir a la siguiente fórmula [42]:

$$k = \frac{\ln(1-p)}{\ln(1-u^m)} \tag{3-1}$$

Donde p es la probabilidad de que tras realizar las k iteraciones se haya tomado una muestra donde todos los puntos sean *inliers* (normalmente se fija en 0.99), u es la probabilidad de que al elegir un punto de manera aleatoria este sea un *inlier*, y m es el tamaño del vector de parámetros o el mínimo número de puntos necesarios para construir el modelo.

La gran robustez de este algoritmo lo ha convertido en un estándar en la comunidad científica, y numeras variaciones han surgido en la literatura intentando corregir sus principales defectos. Entre estas versiones mejoradas se encuentran el MSAC (*M-estimator SAmple Consensus*) y el MLESAC (*Maximum Likelihood Estimation SAmple Consensus*), los cuales implementan un método más robusto para evaluar los modelos obtenidos, deshaciéndonos de la tolerancia como parámetro. También encontramos métodos como el R-RANSAC (*Randomized RANSAC*), que busca los *inliers* de cada modelo en un subconjunto reducido de la nube de puntos original, o el PROSAC (*PROgressive SAmple Consensus*), que hace uso de información preliminar sobre la nube de puntos para guiar el proceso de muestreo.

3.3 Descripción del programa para el procesamiento de nubes de puntos

En este apartado se va a explicar con detalle el código desarrollado para el procesamiento de las nubes de puntos extraídas por el UAV. Como se ha comentado previamente, este código se ha escrito en *C++* dentro del propio entorno ROS y haciendo uso de las librerías que nos proporciona. Además de las librerías propias (*core libraries*) de ROS, la esencia del código reside en la librería integrada PCL (*Point Cloud Library*). Esta es la que implementa los propiamente dichos algoritmos de procesamiento de nubes, los cuales han sido explicados con detalle en apartados previos.

El código consta fundamentalmente de cuatro bloques: un bloque de lectura y escritura de nubes de puntos almacenadas en archivos *bag*, y tres bloques de procesamiento de nubes, los cuales se ejecutan uno a continuación del otro, iterando para todos los mensajes del archivo. Tras finalizar cada bloque, se ha optado por guardar en un archivo *bag* las nubes resultantes de los distintos procesos que lo conforman. Esto nos permitirá visualizar de manera sencilla los resultados en el próximo capítulo, así como detectar errores durante la depuración del código y ajustar los parámetros de los algoritmos empleados.

El resultado final de este procesamiento de nubes de puntos son los coeficientes de los modelos de cada línea segmentada. Como puede verse en la nube de muestra de la Figura 64, los vuelos experimentales se han llevado a cabo sobre tendido eléctrico de tres líneas conductoras, una para cada fase. El objetivo por tanto es extraer las ecuaciones de estas tres líneas para cada nube de puntos. La curva catenaria que forman estos cables conductores debido a su propio peso no puede distinguirse debido a la cercanía del dron al tendido y al campo de visión del LiDAR (cabe recordar que el vuelo experimental consiste en un vuelo de seguimiento de la línea, a una distancia relativamente próxima de unos 12m). Debido a esto y a que el objetivo final es obtener la distancia a dichas líneas de transporte, estas se han modelado como líneas rectas en el espacio 3D.

En la Figura 56 se muestra el diagrama de flujo completo del código desarrollado para el proyecto. No se va a indagar sobre los detalles del código, sino que nos centraremos en explicar qué hace cada bloque y por qué. Esto se hará comentando brevemente las clases utilizadas para implementar los algoritmos y los parámetros que se piden, todo ello con el objetivo de facilitar la tarea a cualquier persona que quiera desarrollar un programa similar. El código completo puede encontrarse en el anexo.

A continuación, se va a proceder a explicar cada bloque por separado, siguiendo la toma de decisiones hasta llegar al código final.

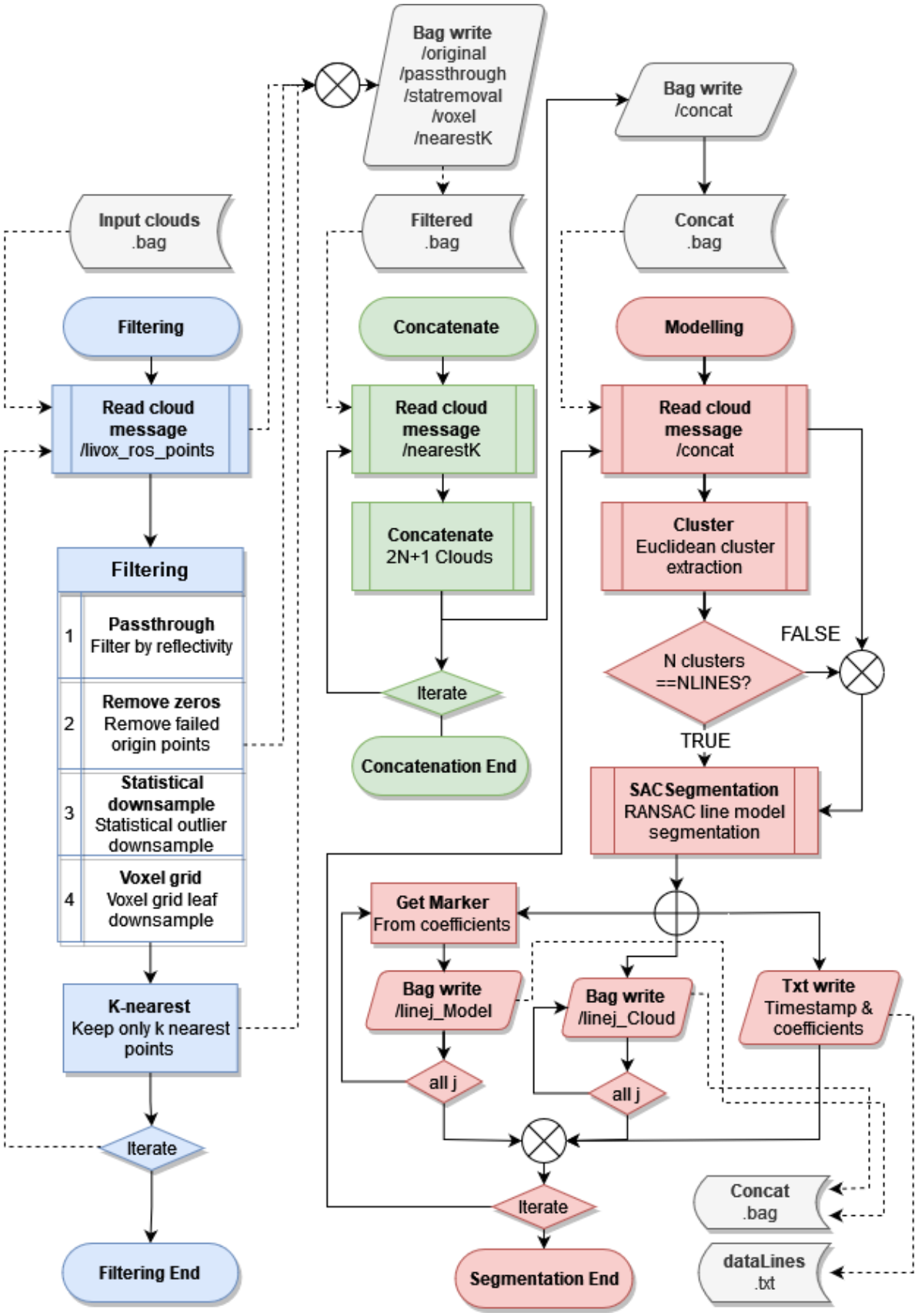


Figura 56. Diagrama de flujo del código escrito en C++ para el procesamiento de las nubes de puntos. Pueden distinguirse con tres colores distintos los bloques de filtrado, concatenación y segmentación. Además, se muestran las operaciones de lectura y escritura de los archivos bag.

3.3.1 Bloque de lectura y escritura

El primer paso de cada bloque consiste en la lectura de los archivos *bag*, que contienen los mensajes ROS del tipo nube de puntos *PointCloud2*.

Para acceder a esta información se debe crear un objeto de la clase *rosbag::View*, el cual funciona como iterador que devuelve uno a uno cada mensaje. A este objeto se le debe asignar un *rosbag::Bag* y un *rosbag::TopicQuery*. El primero apunta al archivo *bag*, mientras que el segundo pide los mensajes pertenecientes a un tema determinado.

Una vez hemos accedido a los mensajes, debemos usar el método *pcl::fromROSMsg()* para transformar el tipo de nube sensor_msgs::PointCloud2 usado por ROS al tipo usado por la propia librería PCL, *pcl::PointCloud<PointXYZINormal>*. Vemos que este último define explícitamente los campos adicionales a las coordenadas XYZ que contienen nuestras nubes. Por su parte, la estructura del mensaje de ROS puede apreciarse en la

Tabla 1. Una vez tenemos acceso a este objeto, estamos listos para aplicarle los distintos algoritmos de la librería PCL.

Por otro lado, tras aplicar cada uno de estos algoritmos escribimos en un archivo *bag* (uno distinto para cada bloque) las nubes de puntos resultantes. Los mensajes se publican en temas que identifican la operación que acaba de ejecutarse sobre dicha nube. La operación se escritura es tan sencilla como aplicar los métodos *Bag::open()*, *Bag::write()* y *Bag::close()* para abrir, escribir y cerrar el archivo, respectivamente.

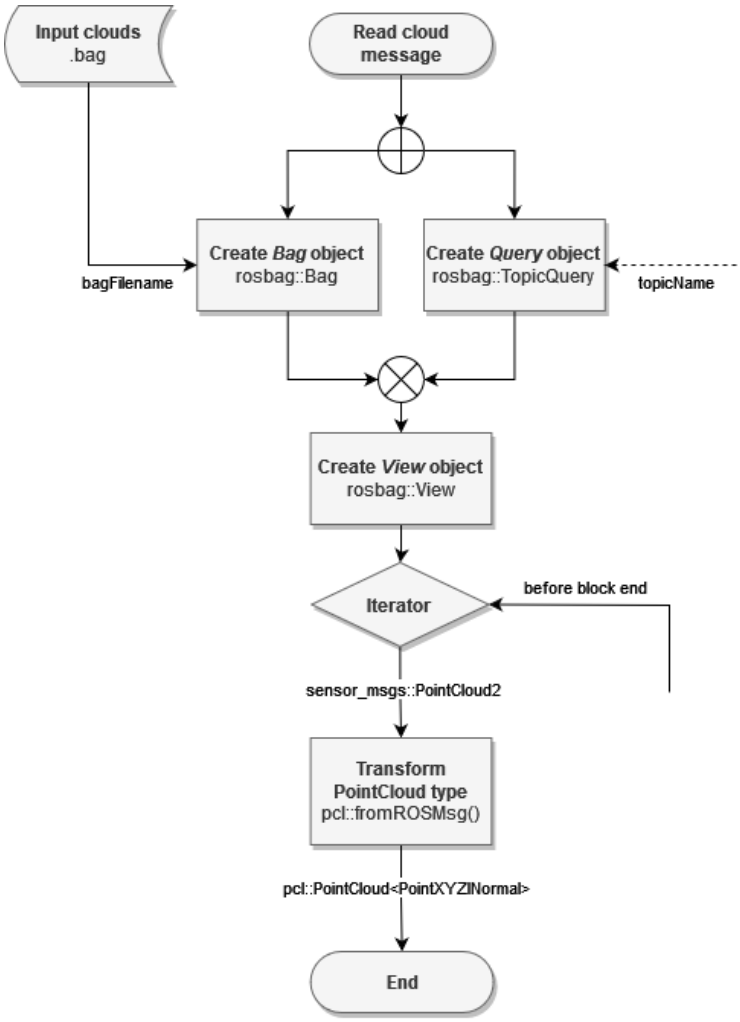


Figura 57. Diagrama de flujo del bloque genérico de lectura. Se debe crear un objeto iterador para acceder a cada nube, para

luego transformarla al tipo de nube de puntos de PCL en cada iteración.

Nombre Campo	Tipo Campo	Descripción Campo	Nombre Subcampo	Tipo Subcampo	Descripción Subcampo
header	Header	Cabecero del mensaje, conteniendo la información fundamental para el registro del mensaje	seq	uint32	ID secuencial del mensaje
			stamp	time	*Timestamp* dividido en segundos y nano-segundos
			frame_id	string	Nombre del sistema de coordenadas
height	uint32	Ancho x Altura para especificar las dimensiones del array de puntos en el caso de puntos ordenados. En el caso de puntos no ordenados (nuestro caso), heigth = 1			
width	uint32				
fields	PointField[]	Estructura que contiene la información de cada punto. Además de las coordenadas (x,y,z), permite almacenar otros campos como la reflectividad	name	string	Nombre del campo
			offset	uint32	Posición del campo en la cadena de *bytes* que representa *data*
			datatype	uint8	Identificación del tipo que representa el campo
			count	uint32	Número de elementos en el campo
is_bigendian	bool	Especifica si los *bytes* son *big-endian* o *little-endian.*			
point_step	uint32	Número de *bytes* por punto			
row_step	uint32	Número de *bytes* por fila. Si los puntos no están ordenados, tamaño total de la nube			
data	uint8[]	Datos de la nube de puntos, conteniendo el valor de cada campo de *fields*			
is_dense	bool	*True* si la nube no contiene valores vacíos en forma de NaN			

Tabla 1. Estructura del mensaje PointCloud2 de ROS. Puede verse como el mensaje admite para cada punto campos adicionales

a las coordenadas XYZ.

3.3.2 Bloque de filtrado

El bloque de filtrado es el primer bloque de procesamiento de nube de puntos, y su misión consiste en reducir el número de puntos de la nube sin que esto afecte a las líneas detectadas. Comienza tras la lectura de las nubes de puntos de partida, y en él se aplican una serie de algoritmos de filtrado y de *downsampling* de manera consecutiva.

En primer lugar, nos encontramos con un subbloque que consta de los siguientes pasos:

1. ***Passthrough*:** se trata de un filtro aplicado a uno de los campos de la estructura de los puntos, en concreto al campo *curvature*, donde el LiDAR ha almacenado el valor de la reflectividad de cada punto. Como se ha comentado en el Capítulo 2, este valor es de gran utilidad para deshacernos de los puntos no pertenecientes a la línea de forma muy eficiente. Este filtro se aplica mediante la clase *pcl::PassThrough<PointT>*, y el intervalo en el que aplicar el filtro se controla mediante el parámetro MAXREFL.
2. ***Remove zeros:*** similar al paso anterior, se trata de un simple filtro para descartar los puntos situados en el origen (0,0,0) como consecuencia de fallos en el receptor del LiDAR. En este caso se usa la clase *pcl::ConditionalRemoval<PointT>* para imponer varias condiciones simultáneamente, y el parámetro ZEROTOL para establecer la tolerancia para que un punto sea considerado situado en el origen.
3. ***Statistical downsample:*** aplica el algoritmo *Statistical outlier removal* para eliminar puntos de la nube mediante un criterio estadístico. Esto nos permite descartar aquellos puntos dispersos que no hayan sido eliminados en los pasos anteriores. Se usa la clase *pcl::StatisticalOutlierRemoval<PointT>*. Respecto a los parámetros, MEANK establece el número de vecinos a tener en cuenta para el análisis estadístico y STDMULT el factor que debe multiplicar la desviación típica.
4. ***Voxel grid:*** aplica el algoritmo *VoxelGrid downsampling* para simplificar la nube de puntos sin perder información sobre la geometría obtenida. La idea reside en simplificar cúmulos de puntos para facilitar la tarea de obtener los modelos de línea. Se usa la clase *pcl::VoxelGrid<PointT>* y el tamaño de *Voxel* se define mediante el parámetro VLEAF.

A continuación, y antes de terminar, se aplica el algoritmo de búsqueda k-NN (*k Nearest Neighbours*) mediante *Octrees* para quedarnos únicamente con los *k* puntos más cercanos al origen. De esta forma, nos quedaremos con el tramo de línea más cercano y obtendremos una mejor estimación de la pendiente local. Esto funcionará siempre que el UAV vuele cerca de la línea y que los filtros previos hayan podido aislarla correctamente. El algoritmo se implementa mediante la clase *pcl::octree::OctreePointCloudSearch<PointT>*, y *k* se define a través de KNEAREST.

Todos estos algoritmos siguen una estructura común a la hora de implementarse en el código:

1. Definición de la clase que implementa el algoritmo.
2. Establecer la nube de puntos de partida.
3. Establecer la salida: o bien una nube de puntos o bien un vector de índices.
4. Configurar los parámetros.
5. Ejecutar el algoritmo.
6. Si la salida son los índices, extraer los puntos de la nube original si es necesario.

Obviamente, este bloque se itera para todas las nubes del archivo *bag*. Una vez completado, el nuevo *bag* generado se guarda y se cierra para su lectura en el bloque siguiente.

3.3.3 Bloque de concatenación

El bloque de concatenación superpone sobre cada nube las n nubes de los instantes justo anteriores y las n nubes posteriores. De esta forma cada nube pasa a estar formada por una concatenación de $2n + 1$ nubes. El valor n se establece mediante el parámetro NCONCAT.

El motivo para realizar esta operación reside en la baja resolución que pueden presentar algunas nubes obtenidas a través del sensor LiDAR. Esto puede dificultar enormemente la extracción de los modelos de línea, o incluso imposibilitarla por completo durante ciertos tramos de la toma de datos. Con esta técnica se consigue un aumento artificial de la resolución del sensor, y es válida siempre que se cumplan ciertos criterios:

- El sensor LiDAR debe funcionar a una frecuencia de muestre lo suficientemente altar para asegurarnos que las nubes concatenadas pertenecen aproximadamente al mismo instante.
- El UAV debe volar a una velocidad no demasiado elevada, pues en caso contrario pueden aparecer distorsiones considerables sobre la geometría de la nube de puntos (ver Figura 81).

Tras este bloque, la nube de puntos está lista para someterse al proceso de segmentación.

3.3.4 Bloque de segmentación

El bloque de segmentación o de modelado es el encargado de aislar cada línea del tendido eléctrico y de obtener para cada una de ellas los coeficientes del modelo de línea 3D. A este bloque llegan las nubes de puntos previamente tratadas, lo cual es fundamental para que la segmentación funcione correctamente. Consta fundamentalmente de cuatro partes:

1. ***Clustering*:** el primer paso consiste en aplicar el algoritmo *Euclidean cluster extraction* descrito en apartados previos. Este trata de segmentar cada una de las líneas utilizando únicamente como criterio la distancia euclídea entre los puntos. La búsqueda de puntos vecinos se lleva a cabo mediante un algoritmo de búsqueda, en este caso el *KdTree Search*. El algoritmo se implementa mediante la clase *pcl::EuclideanClusterExtraction<PointT>*, la cual requiere de la clase *pcl::search::KdTree<PointT>*. Mediante el parámetro EUCLTOL se define la distancia mínima para considerar que dos puntos pertenecen al mismo *cluster*, mientras que el parámetro MINCLUSTSIZE define el mínimo número de puntos necesario para considerar un *cluster* como tal.
2. ***Decision***: el algoritmo descrito anteriormente es muy sensible al ruido y además la presencia de las torres puede poner en compromiso la segmentación. Por lo tanto, y tal como se indica en el diagrama, si el número de *clusters* obtenido mediante este método es distinto al número de líneas observable NLINES (3 en nuestro caso) prescindiremos de estos resultados y pasaremos al siguiente paso con la nube de puntos de partida para este bloque.
3. ***Segmentation:*** en este paso se aplica el algoritmo RANSAC para segmentar las líneas de cada nube. Puesto que previamente se han agrupado los puntos en distintos *clusters*, el código implementado acepta un vector de *clusters* y trata de obtener las líneas contenidas en cada uno de ellos. El resultado final es un vector de nubes de puntos segmentadas, donde cada nube corresponde a una línea distinta. Este viene acompañado de un vector de coeficientes de cada modelo, siendo estos 6 coeficientes: 3 valores representando la posición de un punto de la recta y los otros 3 representando un vector director unitario.

   La clase que implementa el algoritmo es *pcl::SACSegmentation<PointT>*, a la cual se le debe especificar que se usará el método *pcl::SAC_RANSAC* y el modelo de línea 3D *pcl::SACMODEL_LINE*. Respecto a los parámetros, por un lado tenemos RANSAC_MAXIT, que

establece el número de iteraciones a ejecutar. Este parámetro debe seleccionarse cuidadosamente conforme a la ecuación (3-1). Por otro lado, tenemos los parámetros RANSAC_DIST y RANSAC_MININLIERS. Estos especifican la tolerancia para determinar si un punto forma parte del modelo y el mínino número de *inliers* para considerar que un modelo es válido, respectivamente. Se puede apreciar en el diagrama de la Figura 58 que estos parámetros se ajustan en función del número de puntos que conforman la nube a segmentar.

4. ***Save:*** por último, se procede a guardar la información de los modelos obtenidos. Cada línea segmentada se almacena en un archivo *bag* como una nube de puntos independiente. Además, a partir de los coeficientes se han creado los mensajes en forma de vector del tipo *visualization_msgs::Marker* que nos permitirán visualizar en *RViz* cada modelo de recta. No obstante, para su exportación a otra plataforma (donde se aplicará el siguiente paso postprocesamiento) los coeficientes se han almacenado en formato *.txt*. Cada línea de este archivo representa una línea segmentada (valga la redundancia), y en ella se almacenan el *timestamp* y los 6 coeficientes del modelo.

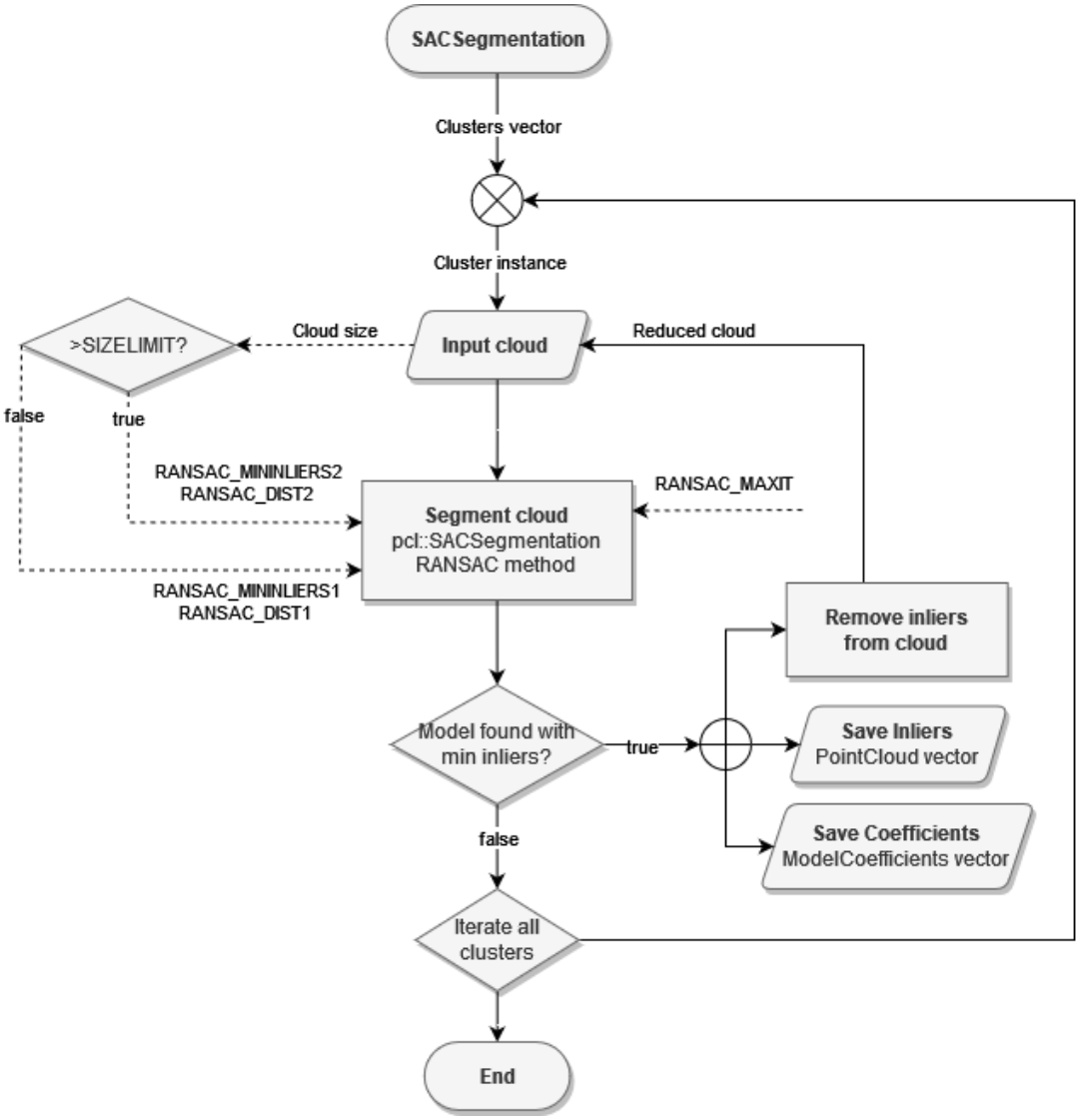


Figura 58. Diagrama de flujo del bloque de segmentación. El código es capaz de segmentar las líneas pertenecientes a distintos *clusters*. Los puntos asociados a cada línea y su s coeficientes se guardan por separado.

Este bloque cierra el código desarrollado en *C++* para el procesamiento de nubes de puntos. Una vez tenemos la información de los modelos en un archivo *txt* podemos pasar a *Python* para ejecutar el siguiente y último paso: el postprocesamiento de las líneas segmentadas.

3.3.5 Resumen del código

A forma de resumen se presentan la Tabla 2 y Tabla 3, que recogen por un lado las clases principales utilizadas en el código y por el otro el listado de parámetros de configuración que sirven de entrada a los algoritmos.

Clase	Bloque	Librería	Descripción
sensor_msgs::PointCloud2	Común	*common_msgs*	Tipo ROS para nubes de puntos
pcl::PointCloud<T>	Común	*pcl_common*	Tipo PCL para nubes de puntos
rosbag::Bag	Lectura	*rosbag*	Objeto que apunta al directorio del archivo *bag*
rosbag::Query	Lectura	*rosbag*	Usado para pedir unos temas determinados al archivo *bag*
rosbag::View	Lectura	*rosbag*	Iterador para acceder a los mensajes contenidos en el archivo *bag*
pcl::fromROSMsg()	Lectura	*pcl_ros*	Método para transformar entre tipos de nubes de puntos
pcl::PassThrough<T>	Filtrado	*pcl_filters*	Algoritmo simple de filtrado por rango aplicado a uno de los campos de los puntos
pcl::ConditionalRemoval<T>	Filtrado	*pcl_filters*	Clase para aplicar varios filtros de rango simultáneamente
pcl::StatisticalOutlierRemoval<T>	Filtrado	*pcl_filters*	Algoritmo de *downsampling* estadístico
pcl::VoxelGrid<T>	Filtrado	*pcl_filters*	Algoritmo de *downsampling* mediante mallado de *voxels*
pcl::octree::OctreePointCloudSearch<T>	Búsqueda	*pcl_octree*	Algoritmo de búsqueda mediante la estructura de árbol *Octree*
pcl::search::KdTree<T>	Búsqueda	*pcl_kdtree*	Algoritmo de búsqueda mediante la estructura de árbol *KdTree*
pcl::EuclideanClusterExtraction<T>	Segmentación	*pcl_ segmentation*	Algoritmo de segmentación mediante distancias euclídeas
pcl::SACSegmentation<T>	Segmentación	*pcl_sample_ consensus*	Algoritmo de segmentación mediante ajuste de modelos

Tabla 2. Listado de clases utilizadas en el código de procesamiento de nubes de puntos.

Parámetro	Bloque	Algoritmo	Descripción
MAXREFL	Filtrado	*PassThrough*	Límite superior del intervalo para filtrar los puntos a través de la intensidad recibida
ZEROTOL	Filtrado	*Conditional Removal*	Tolerancia para eliminar los puntos cercanos al origen
MEANK	Filtrado	*Statistical Outlier Removal*	Número de puntos utilizado para calcular los parámetros estadísticos
STDMULT	Filtrado	*Statistical Outlier Removal*	Multiplicador de la desviación típica para determinar el intervalo de *downsampling*
VLEAF	Filtrado	*VoxelGrid Downsampling*	Tamaño del *voxel* para el *downsampling*
KNEAREST	Filtrado	*k-Nearest Search*	Número de puntos más cercanos con los que quedarnos
NCONCAT	Concatenación	-	Número de nubes anteriores y posteriores para concatenar
EUCLTOL	Segmentación	*Euclidean Cluster Extraction*	Tolerancia para agrupar puntos en un mismo *cluster*
MINCLUSTSIZE	Segmentación	*Euclidean Cluster Extraction*	Tamaño mínimo del *cluster* para que se considere válido
RANSAC_MAXIT	Segmentación	*RANSAC Segmentation*	Número de iteraciones del algoritmo RANSAC
RANSAC_DIST	Segmentación	*RANSAC Segmentation*	Tolerancia para determinar si un punto es *inlier* del modelo obtenido
RANSAC_MININLIERS	Segmentación	*RANSAC Segmentation*	Número mínimo de *inliers* para considerar válido el modelo

Tabla 3. Listado de parámetros para configurar el programa de procesamiento de nubes de puntos.

3.4 Descripción del programa para el postprocesamiento de las líneas segmentadas

Durante el procesamiento de las nubes de puntos es posible que algunas de ellas no sean segmentadas correctamente y, por tanto, que los modelos de línea 3D no representen la disposición real del tendido eléctrico respecto al dron. Si esto ocurre las distancias obtenidas no tendrían nada que ver con las distancias reales al tendido eléctrico. Mientras esto ocurra para un porcentaje reducido del total de nubes de puntos extraídas, el código de procesamiento de nubes sigue siendo útil. Sin embargo, es necesario disponer de una herramienta para detectar estos fallos de segmentación y descartar los modelos erróneos. Por otro lado, para las líneas extraídas correctamente se debe implementar un código para extraer la distancia desde el origen hasta la línea, así como la distancia media al tendido eléctrico.

Se ha elegido *Python* para desarrollar este código de postprocesamiento. La entrada es el archivo *txt* que contiene línea a línea los valores del *timestamp* y los coeficientes de cada modelo. Para lectura y postprocesamiento se han usado *Numpy* y *Pandas*, que son las librerías estándar de computación en la comunidad científica. Para la representación de los resultados se ha usado *matplotlib*.

(a) (b)

Figura 59. (a) Numpy y (b) Pandas, las librerías de Python más usadas para computación.

En la Figura 61 se muestra el diagrama de flujo para este proceso. El código se describe con más detalle a continuación:

1. **Lectura:** el primer paso consiste en leer las líneas segmentadas que han sido almacenadas en el archivo de texto. Este archivo contiene líne a línea el *timestamp* y los 6 coeficientes que caracterizan cada recta en el espacio 3D. El formato de este archivo es similar al CSV (*Comma Separated Values*), aunque los valores vienen separados por espacios en vez de por comas. Para su lectura recurrimos a los *Numpy Structured Arrays*. Estos son vectores estructurados, es decir, vectores que no almacenan un tipo simple como puede ser un entero o un *float*, sino que almacenan una estructura de datos. La estructura de datos definida para esta ocasión se recoge en la Tabla 4. El resultado final es un vector con esta estructura, donde cada elemento del vector se corresponde con una línea segmentada distinta.

Campo	Tipo	Tamaño	Descripción
timestamp	*uint64*	Escalar	Nanosegundos desde la época de referencia
line	*float32[]*	6	Coeficientes del modelo: punto sobre la linea (x_0, y_0, z_0) y vector director (v_x, v_y, v_z)

Tabla 4. Estructura de datos resultado del procesamiento de las nubes.

2. **Agrupamiento**: partiendo de este *array* de líneas segmentadas procedemos a agrupar las líneas de acuerdo a dos criterios. En primer lugar, agrupamos las líneas que comparten el mismo *timestamp*, es decir, aquellas que formaban parte de la misma nube. A continuación agrupamos las líneas por paralelismo, es decir, se agrupan las líneas que se correspondan con el mismo instante y seas paralelas entre sí. La condición de paralelismo se obtiene a partir del ángulo que forman los vectores directores de las líneas. Se define una tolerancia a partir del parámetro PTOL, que es el coseno del ángulo máximo para que las líneas sean consideradas paralelas.
3. **Transformación del *timestamp:*** por comodidad se reemplaza el *timestamp* por su transformada a segundos, para lo cual se toma como referencia el *timestamp* más bajo entre los disponibles.
4. **Filtros:** el siguiente paso consiste en deshacernos de grupos formados por líneas que no han sido segmentadas correctamente. Para ello se aplican dos filtros: el primero simplemente consiste en eliminar los grupos formados por un número de líneas distinto al esperado (en nuestro caso, sabemos que el tendido eléctrico grabado durante los vuelos experimentales está formado por 3 líneas); el segundo consiste en eliminar grupos que contengan líneas coincidentes. Decimos que dos líneas son coincidentes cuando sus vectores directores son paralelos y cuando sus vectores ortogonales son coincidentes. Los vectores ortogonales son los vectores perpendiculares a la línea y que pasan por el origen (son los vectores cuyo módulo coincide con la distancia desde el origen a la línea, ver Figura 60). Nuevamente, se utiliza una tolerancia definida a través del parámetro CTOL para comprobar si los vectores son coincidentes.

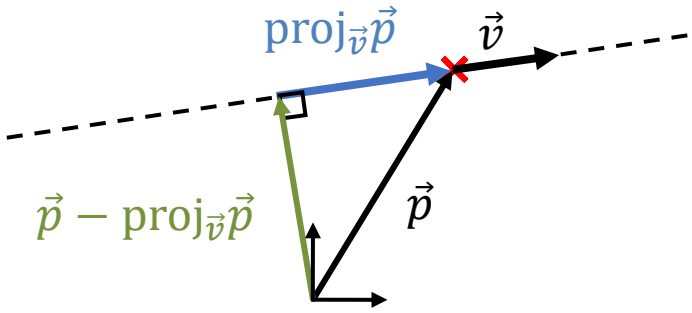


Figura 60. Obtención del vector ortogonal a la línea a partir del vector director y de un punto contenido en la misma.

5. **Obtención de la línea media:** para definir la línea media de las 3 líneas segmentadas debemos obtener su vector director y un punto de la misma. El vector director se obtiene promediando los vectores directores de las líneas. Un punto de la misma se obtiene igualmente promediando los vectores ortogonales calculados, dando como resultado el punto situado en el baricentro del triángulo que forman

las líneas.

6. **Cálculo de distancias:** se calculan las distancias desde el origen hasta cada una de las cuatro líneas (las tres líneas segmentadas y la línea media). La distancia a las líneas es simplemente el módulo del vector ortogonal. La distancia media al tendido eléctrico se define como la distancia a la línea media.
7. **Tracking:** este paso nos permite comprobar que las distancias calculadas forman una función aproximadamente continua. Par ello se compara la distancia media entre instantes sucesivos, comprobando que no existen saltos importantes en el valor de la distancia. Si se produce un salto o no se dispone de información de las líneas para un intervalo de tiempo dado por el parámetro TTOL, las líneas se separan en grupos distintos para su representación gráfica.
8. **Representación gráfica:** por último, se representan gráficamente los resultados, principalmente la distancia media al tendido eléctrico frente al tiempo.

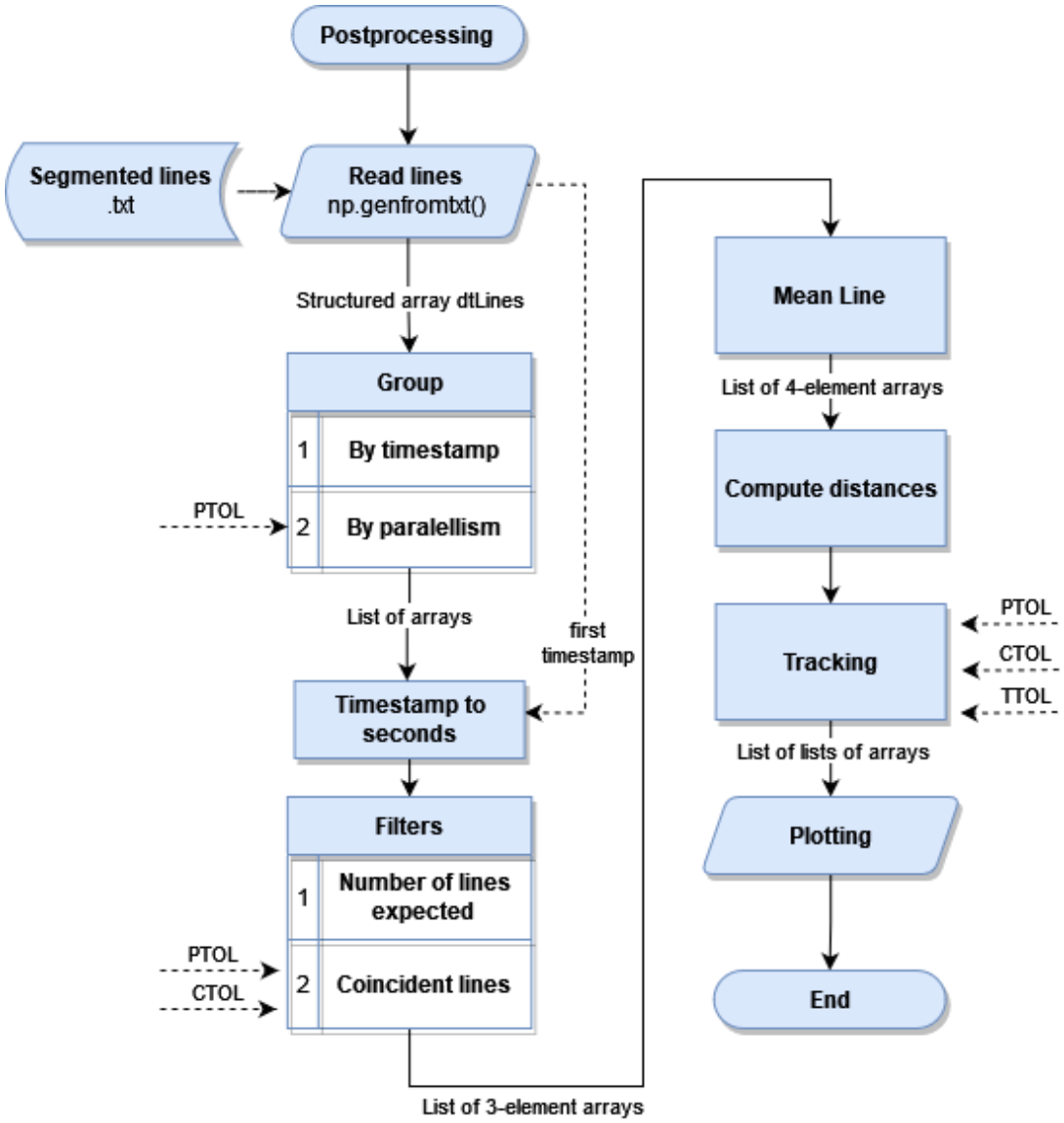


Figura 61. Diagrama de flujo del bloque del postprocesamiento de las líneas segmentadas.

3.5 Extracción de los datos del campo magnético

Aunque no se ha mencionado en los apartados anteriores, en paralelo al procesamiento y postprocesamiento de las nubes de puntos se han leído igualmente los mensajes de ROS correspondientes al campo magnético. Las medidas registradas en el archivo *bag* corresponden a los valores de la densidad espectral de potencia (*Power Spectral Density* PSD) obtenida por el analizador del espectro del campo magnético. La estructura del mensaje ROS correspondiente se muestra en la Tabla 5. Se trata de un mensaje no estándar.

Nombre Campo	**Tipo Campo**	**Descripción Campo**
header	Header	Cabecero del mensaje, conteniendo la información fundamental para el registro del mensaje
frame_id	string	Nombre del sistema de coordenadas
Frequencies	float64[]	Vector de 51 elementos que contiene las frecuencias para las que se conoce la PSD
FrequenciesValues	float64[]	Vector de 51 elementos que contiene los valores del PSD en μT^2

Tabla 5. Estructura del mensaje ROS conteniendo la información del campo magnético.

Los mensajes conteniendo esta información han sido iterados de forma análoga a como se ha hecho con las nubes de puntos. La diferencia es que estos mensajes no requieren de ningún tipo de procesado o de postprocesado. De igual forma tendremos otro archivo de texto que nos permitirá exportar los datos a *Python.* Esta exportación es necesaria si queremos entrenar modelos de *machine learning.* En la TABLA se muestra la estructura de datos exportada para su lectura mediante *Numpy.*

Campo	**Tipo**	**Tamaño**	**Descripción**
timestamp	*uint64*	Escalar	Nanosegundos desde la época de referencia
frequencies	*float32[]*	51	Frecuencias para las que se conoce el valor de la PSD
power	*float32[]*	51	Valores de la PSD

Tabla 6. Estructura de datos utilizada para exportar la información del campo magnético a Python.

4 RESULTADOS

En este capítulo se presentan los resultados obtenidos en los distintos pasos del programa desarrollado para el procesamiento de nubes de puntos y descrito en el capítulo anterior. En este sentido, se representarán en el espacio 3D las nubes de puntos correspondientes: desde la nube de puntos de partida hasta los modelos de línea finales segmentados, pasando por cada uno de los algoritmos de filtrado, concatenación y segmentación. También se mostrará el espectro de potencia de la señal obtenida por el sensor de campo magnético.

En primer lugar, se procede a la lectura del archivo *bag* y a la visualización de las medidas de los sensores a través del entorno ROS. A continuación, se presenta paso a paso el resultado de aplicar cada algoritmo a las nubes de puntos, hasta conseguir las líneas segmentadas y los coeficientes de los modelos de línea. Por último, el código de preprocesado de datos prepara y filtra los datos para alimentar los modelos de *machine learning*, obteniendo la distancia media a cada conductor y el espectro del campo magnético para un conjunto de instantes determinado.

4.1 Visualización del archivo bag

El primer paso consiste en leer el archivo *bag* y visualizar las medidas extraídas. Por suerte, ROS dispone de las interfaces gráficas *rqt_bag* (para inspeccionar y publicar el contenido de archivos bag), y *RViz* (para visualizar nubes de puntos). El proceso es el siguiente:

1. Abrimos el archivo *bag* mediante *rqt_bag*, que nos permite inspeccionar inmediatamente los mensajes que contiene el archivo, organizados por tema. Los mensajes se muestran en una línea temporal como la de la Figura 80, donde se aparece el tiempo de registro en el archivo bag (en adelante, *bag timestamp*). Este instante de registro no debe confundirse con el *timestamp* almacenado internamente para cada mensaje (en adelante, *message timestamp*, o simplemente *timestamp*). Este último es el instante en el que se obtuvo la medida de la nube de puntos o del campo magnético, y es por tanto el valor que nos interesa para correlacionar las. En cambio, el primero es el instante en el que ROS almacena el mensaje en el archivo *bag*, y en principio no estará estar relacionado con el anterior. En la Tabla 7 se resume el contenido de los archivos *bag* disponibles.

Archivo *bag*	Tipo de mensaje	Nº de mensajes	Inicio	Fin	Duración [min: s]
pc_n_emf.bag	sensor_msgs/PointCloud2	4688	18/01/2021 11:46:39 GMT	18/01/2021 11:54:28 GMT	7:49
pc_n_emf.bag	beginner_tutorials/aaronia	411	18/01/2021 11:43:10 GMT	18/01/2021 11:54:51 GMT	11:41
livox_pcd_points_burguillos_exp_002_single.bag	sensor_msgs/PointCloud2	6381	18/01/2021 11:28:30 GMT	18/01/2021 11:39:09 GMT	10:38
emf_burguillos_exp_002_single.bag	beginner_tutorials/aaronia	453	18/01/2021 11:26:18 GMT	18/01/2021 11:39:09 GMT	12:53

Tabla 7. Resumen del contenido de los archivos bag procesados.

Puede observarse que la frecuencia de adquisición de datos es muy diferente de un tipo de mensaje a otro. Mientras que el LiDAR funciona a una tasa media de 10Hz, el analizador de espectro lo hace a una tasa de 0.586Hz. Esto se debe a que este último debe tomar un intervalo de tiempo proporcional al periodo de la señal para estimar su espectro.

Esta diferencia convierte al analizador de espectro en el cuello de botella del sistema de adquisición de datos para los modelos de aprendizaje, obteniéndose dichos datos cada 1.7s como valor promedio. Por otro lado, el disponer de una gran densidad de nubes de puntos es beneficioso para detectar *outliers*, es decir, valores erróneos obtenidos por el algoritmo de procesamiento de dichas nubes de puntos.

2. Analizamos el contenido de los mensajes, con el objetivo de visualizar la estructura de datos. Las nubes de puntos se almacenan en el mensaje estándar de ROS *sensor_msgs/PointCloud2*, mientras que el espectro del campo magnético se almacena en el mensaje *beginner_tutorials/aaronia*, el cual se ha definido manualmente. La estructura de estos mensajes se detalló en la Tabla 1 y Tabla 5. Ambos comparten un campo común, la estructura denominada *header* o cabecero del mensaje. Este cabecero identifica inequívocamente el mensaje, proporcionando una ID secuencial, un *timestamp* y un sistema de referencia o *frame*.

   En la Tabla 8 se recoge el contenido de uno de los mensajes *PointCloud2* a modo de ejemplo. De la estructura de datos correspondiente a las nubes de puntos queda claro que una tabla no nos permite visualizar la nube de puntos. Sí puede verse en cambio el tipo de información que se almacena y que tenemos un total 24000 puntos por nube. Los mensajes serán publicados en los temas correspondientes para su visualización a través de *RViz*.

   Respecto a la información del campo magnético, se ha determinado que el ancho de banda del analizador de espectro comprende entre 45 y 65Hz, con una resolución de 0.4Hz. También podemos apreciar el orden de magnitud del espectro de potencia (en torno a 2 y 5μT^2).

Nombre Campo	Nombre Subcampo	Valor	Comentarios
header	seq	0	-
	stamp	secs: 1610970399 nsecs: 251136541	-
	frame_id	"livox_frame"	Sistema de referencia del LiDAR en el que están expresadas las coordenadas de los puntos
height	-	1	El valor unidad aplica a nubes de puntos no organizadas
width	-	24000	Número de puntos en la nube
fields	name	[x, y, z, intensity, normal_x, normal_y, normal_z, curvature]	Además de las coordenadas xyz, sólo se utilizan *intensity* para almacenar información del escáner y *curvature* para almacenar la intensidad del haz de luz reflejado
	offset	[0, 4, 8, 32, 16, 20, 24, 36]	-
	datatype	[7, 7, 7, 7, 7, 7, 7, 7]	Tipo float32
	count	[1, 1, 1, 1, 1, 1, 1, 1]	-
is_bigendian	-	False	-
point_step	-	48	48 *bytes* por punto
row_step	-	115200	115kB para la nube completa
data	-	[18, 131, 32, 63, …]	Palabras de 8 bits expresadas en decimal y que contienen los datos de la nube (no nos aportan ninguna información en este formato)
is_dense	-	True	-

Tabla 8. Valores de los campos de la estructura del mensaje PointCloud2 de ROS para una nube de muestra.

3. Configuramos *RViz* para la visualización de las nubes de puntos. Además de los parámetros gráficos, debemos añadir los temas correspondientes para recibir los mensajes y el sistema de referencia *livox_frame*, que es el sistema de referencia del LiDAR definido respecto al UAV (Figura 62). De esta forma, podremos visualizar las nubes de puntos en el sistema de referencia fijo al UAV.

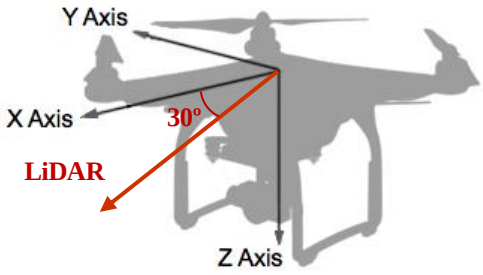


Figura 62. Orientación del LiDAR respecto a los ejes del UAV. El LiDAR se encuentra girado 30º de cabeceo mirando al suelo.

En la Figura 63, Figura 64 y Figura 65 se pueden observar unas muestras de las nubes de partida extraídas de los archivos de la Tabla 7. El mapa de colores se corresponde con el valor de la intensidad de la señal reflejada en cada punto. Puede verse a simple vista que dicho valor es de gran utilidad para segmentar las líneas de alta tensión. También se puede apreciar en los puntos pertenecientes al suelo tanto la técnica de escaneo utilizada como la "sombra" que dejan la torre o las propias líneas.

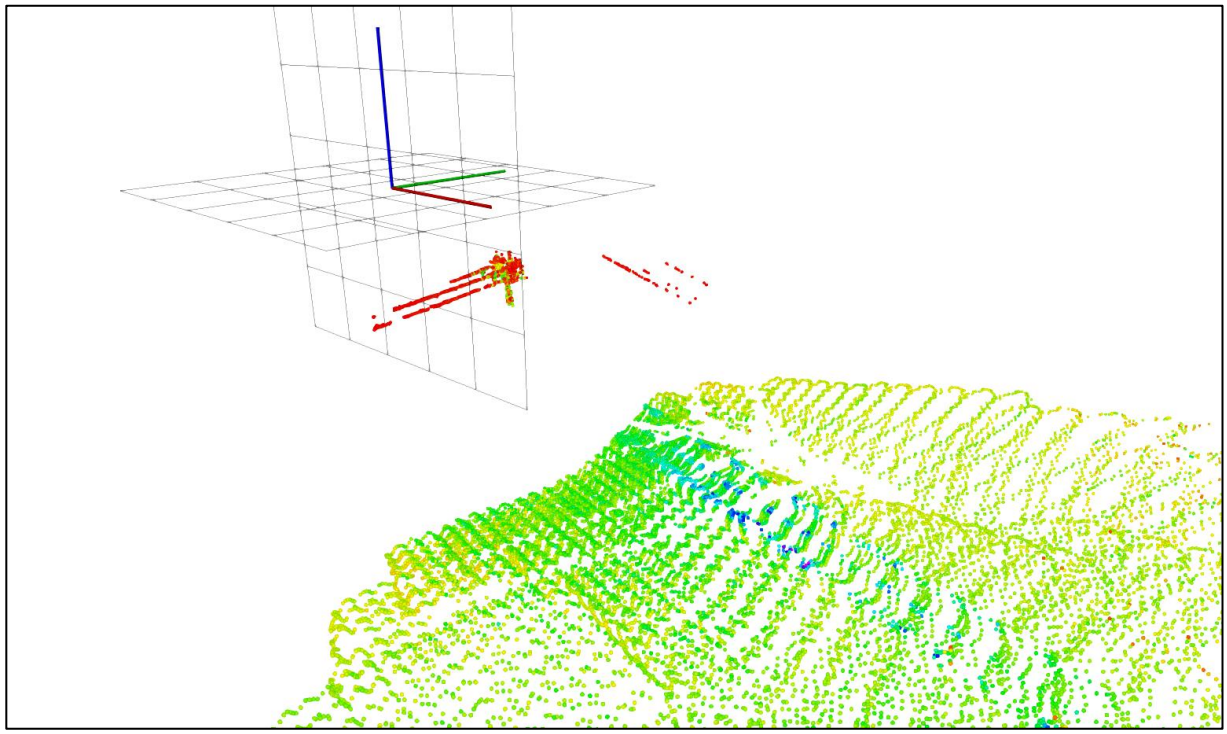


Figura 63. Nube de muestra perteneciente al archivo pc_n_emf.bag. En este instante el LiDAR ha detectado dos tendidos eléctricos independientes.

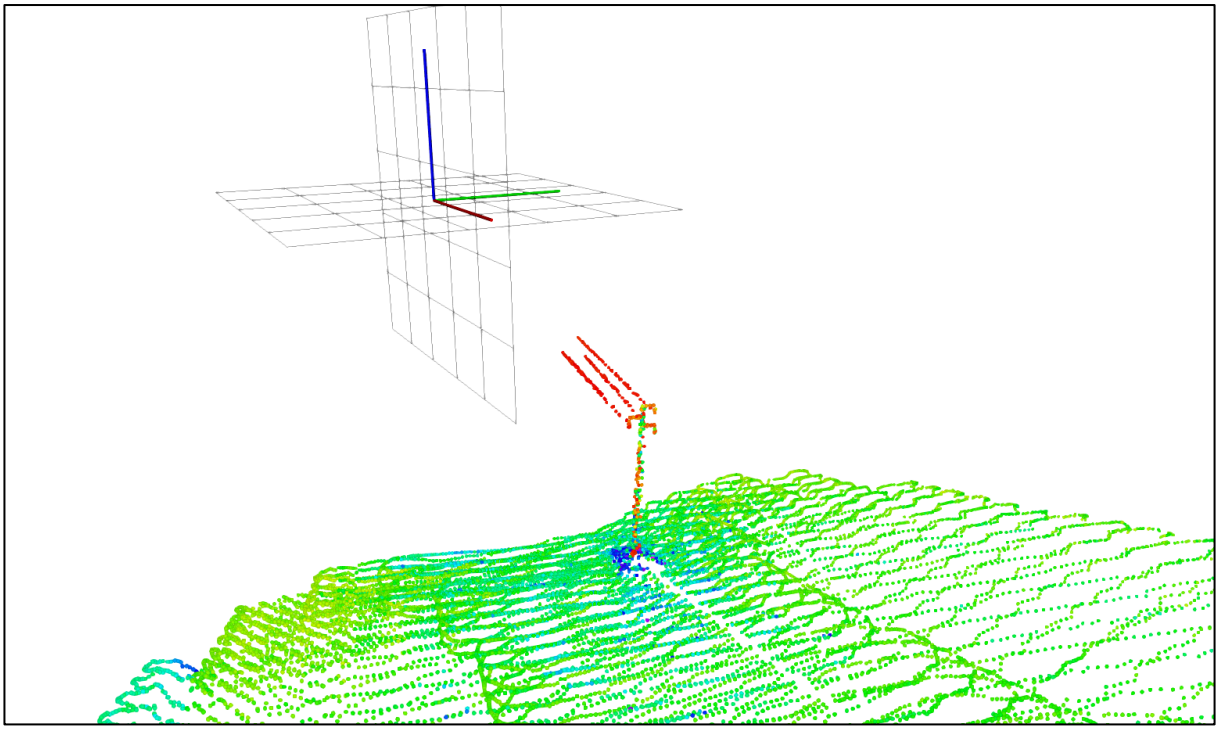


Figura 64. Nube de muestra perteneciente al archivo livox_pcd_points_burguillos_exp_002_single.bag. En la escena puede apreciarse el tendido eléctrico y la torre.

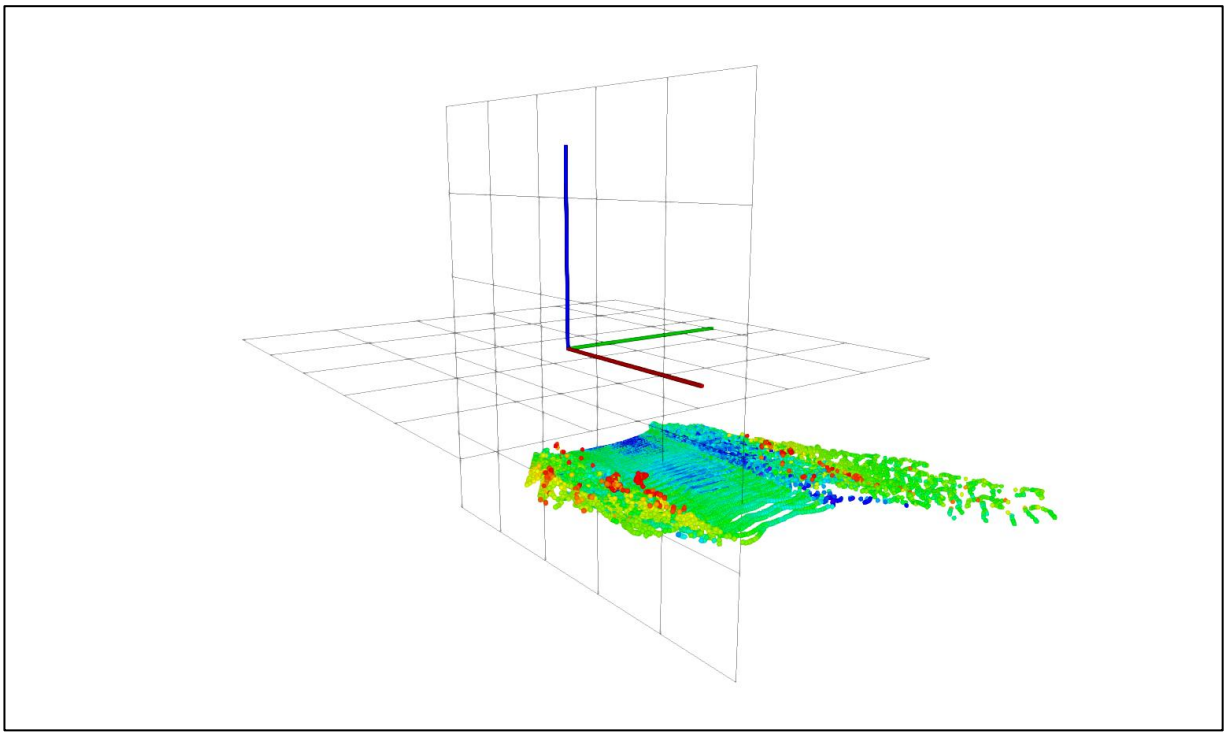


Figura 65. Nube de muestra perteneciente al archivo livox_pcd_points_burguillos_exp_002_single.bag. La escena corresponde a momentos posteriores al despegue.

A continuación, se mostrarán los resultados de aplicar los distintos pasos del procesamiento de nubes de puntos, de tal forma que visualizaremos la evolución de las nubes hasta llegar a la segmentación final y a la obtención de los modelos de línea. Para mostrar este proceso utilizaremos las nubes del archivo *livox_pcd_points_burguillos_exp_002_single.bag,* pues la presencia de torres eléctricas las hace más

interesantes para la evaluación de los resultados. Por medio de una inspección visual se ha extraído un recorte de este archivo, descartando los tramos de despegue y aterrizaje. Se seleccionarán dos nubes de referencia para el seguimiento, en adelante Nube 1 y Nube 2. Dichas nubes se muestran en la Figura 66 y Figura 67 en su estado inicial.

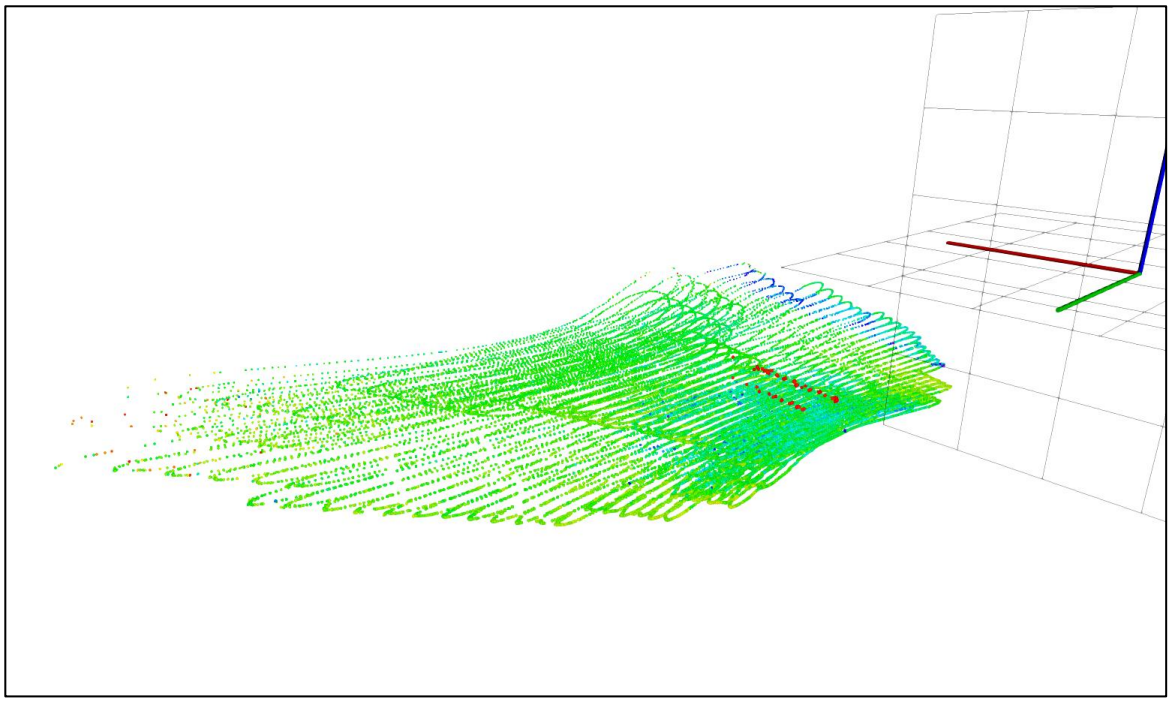


Figura 66. Nube 1 inicial para el seguimiento de los resultados del programa. Pueden apreciarse tanto el suelo como las tres líneas de alta tensión.

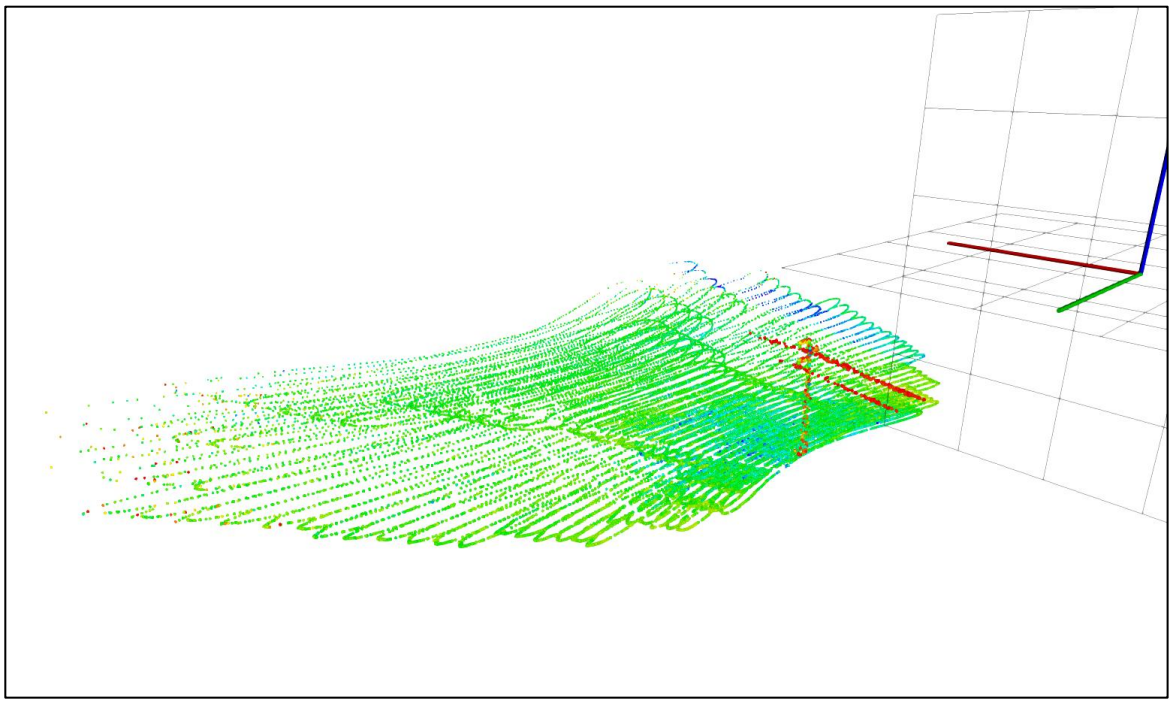


Figura 67. Nube 2 inicial para el seguimiento de los resultados del programa. En este caso también aparece una torre de apoyo.

Los resultados que se van a mostrar se corresponden con la selección de parámetros del programa que ha dado mejores resultados para nuestras nubes de puntos. Esta selección se justificará en el apartado posterior (ajuste paramétrico). En la

Tabla 9 se recogen los valores de los parámetros usados para obtener los resultados finales.

4.2 Resultados del procesamiento de nubes de puntos

4.2.1 Filtrado

De las figuras anteriores y del número de puntos de partida resulta evidente que no es viable emplear algoritmos complejos de segmentación hasta que no hayamos reducido la cantidad de puntos de la nube. Esta es la tarea de este primer bloque de filtrado y *downsampling*.

4.2.1.1 Passthrough y Zero removal

En este paso se aplican dos filtros a la nube: en primer lugar, usando el valor de la intensidad en cada punto y, justo a continuación, eliminando los puntos situados en el origen del sistema de referencia (puntos obtenidos erróneamente por el sensor). Los resultados son excelentes, pudiéndose ver en la Figura 68 y Figura 69 como se han eliminado prácticamente la totalidad de los puntos correspondientes el suelo sin afectar a los puntos de las líneas y de las torres. Hemos pasado de tener del orden de 24000 puntos a 227 puntos para la Nube 2. Aun así, todavía pueden apreciarse puntos aislados, los cuales se tratarán en el siguiente paso.

El umbral seleccionado de MAXREFL=0.1 resultar ser la mejor solución para eliminar la mayor cantidad de puntos sin afectar a la línea. Para la eliminación de los puntos en el origen se ha tomado una tolerancia de ZEROTOL=0.5.

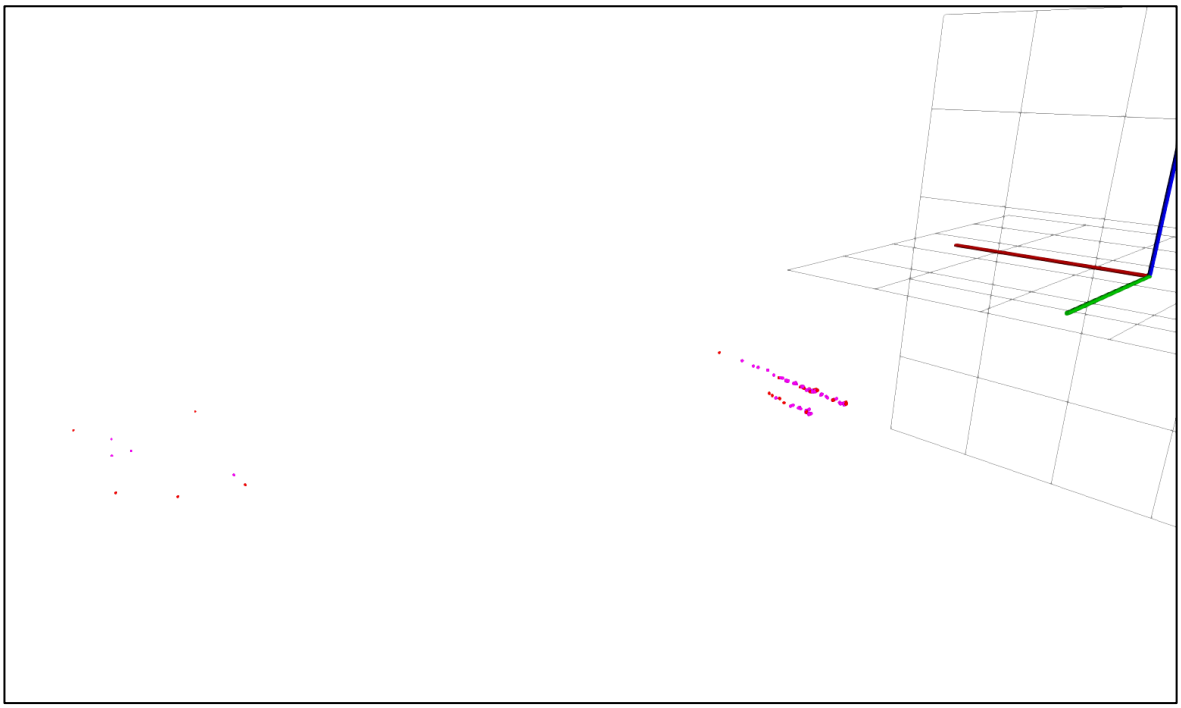


Figura 68. Nube 1 tras aplicar los algoritmos *Passthrough* y *Zero removal*.

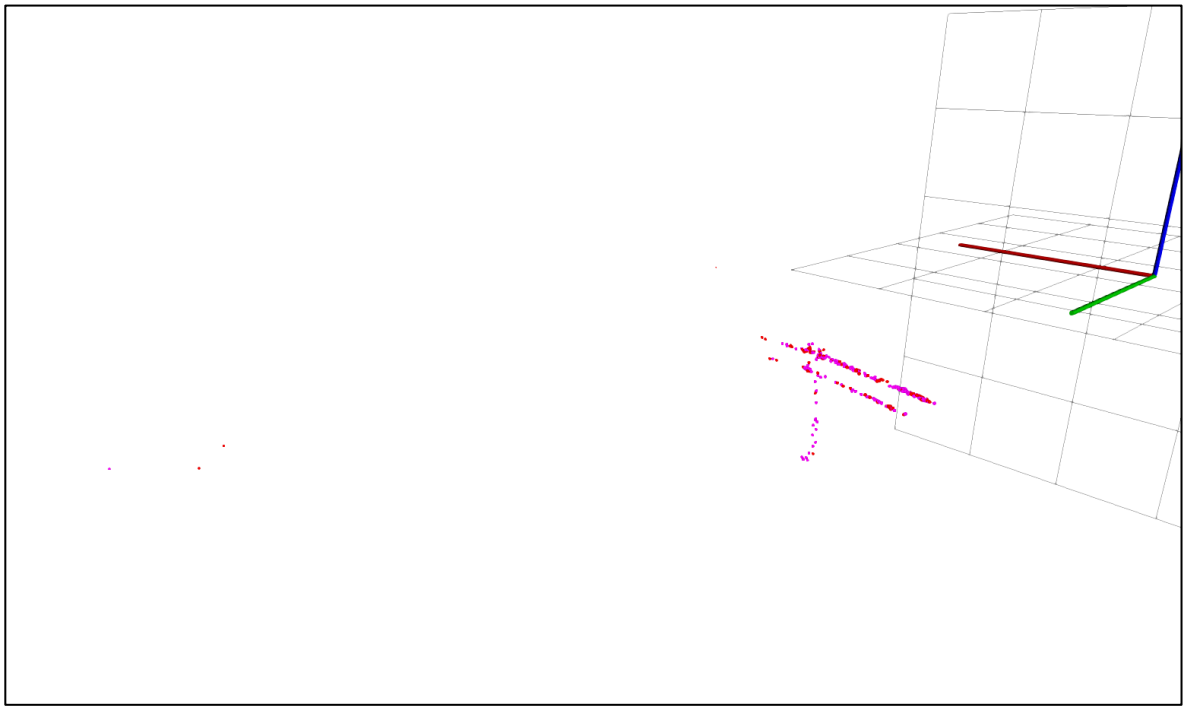


Figura 69. Nube 2 tras aplicar los algoritmos *Passthrough* y *Zero removal*.

4.2.1.2 Statistical downsample

En este paso se aplicará un filtro estadístico para eliminar puntos aislados y quedarnos así únicamente con los puntos pertenecientes al tendido eléctrico. Los resultados se muestran en la Figura 70 y Figura 71.

Los valores seleccionados para los parámetros han sido MEANK=25 y STDMULT=1.0.

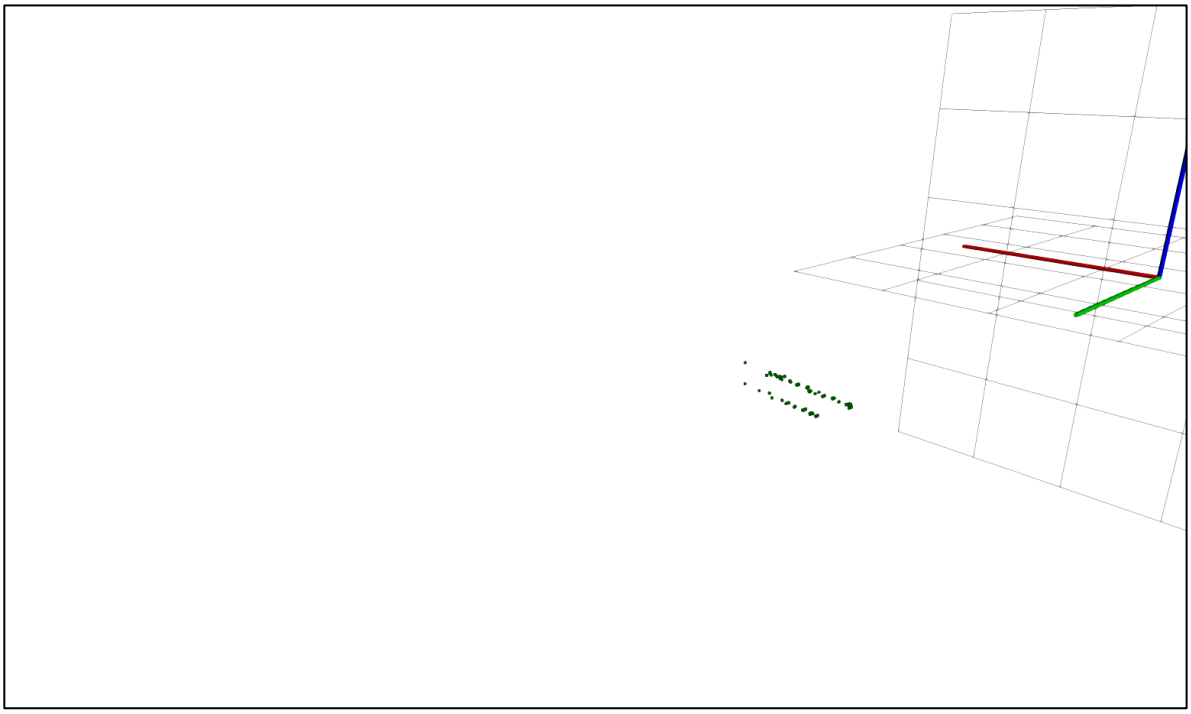


Figura 70. Nube 1 tras aplicar el algoritmo *Statistical removal*.

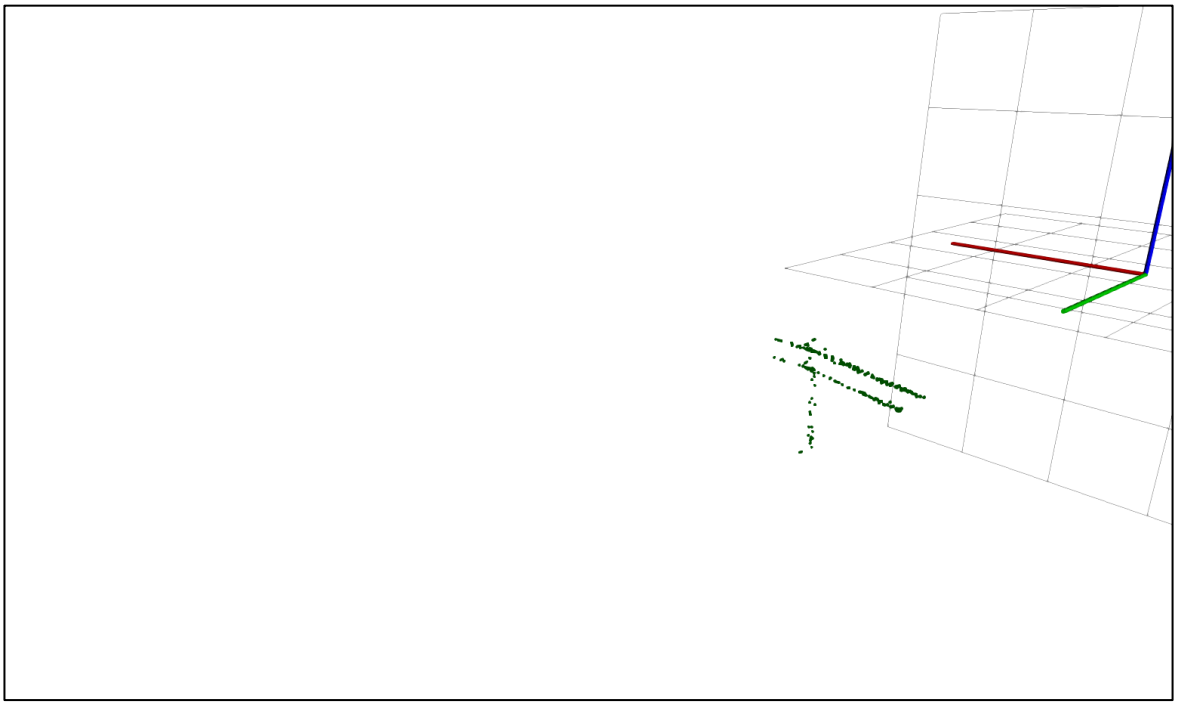


Figura 71. Nube 2 tras aplicar el algoritmo *Statistical removal*.

4.2.1.3 VoxelGrid downsample

El objetivo de este paso es simplificar la nube reduciendo la densidad de puntos. Esto facilitará el proceso de segmentación de líneas mediante RANSAC, pues dispondremos de una geometría más limpia y reconocible. Los resultados se muestran en la Figura 72 y Figura 73. Para la Nube 2 nos quedamos con unos 100 puntos.

El tamaño de *voxel* elegido para el *downsampling* ha sido VLEAF=0.5. Este valor debe mantenerse por debajo de la distancia mínima entre líneas (aproximadamente de 1.5m) para evitar que se promedien líneas entre sí, pero un valor demasiado bajo no tendría ningún efecto sobre la nube.

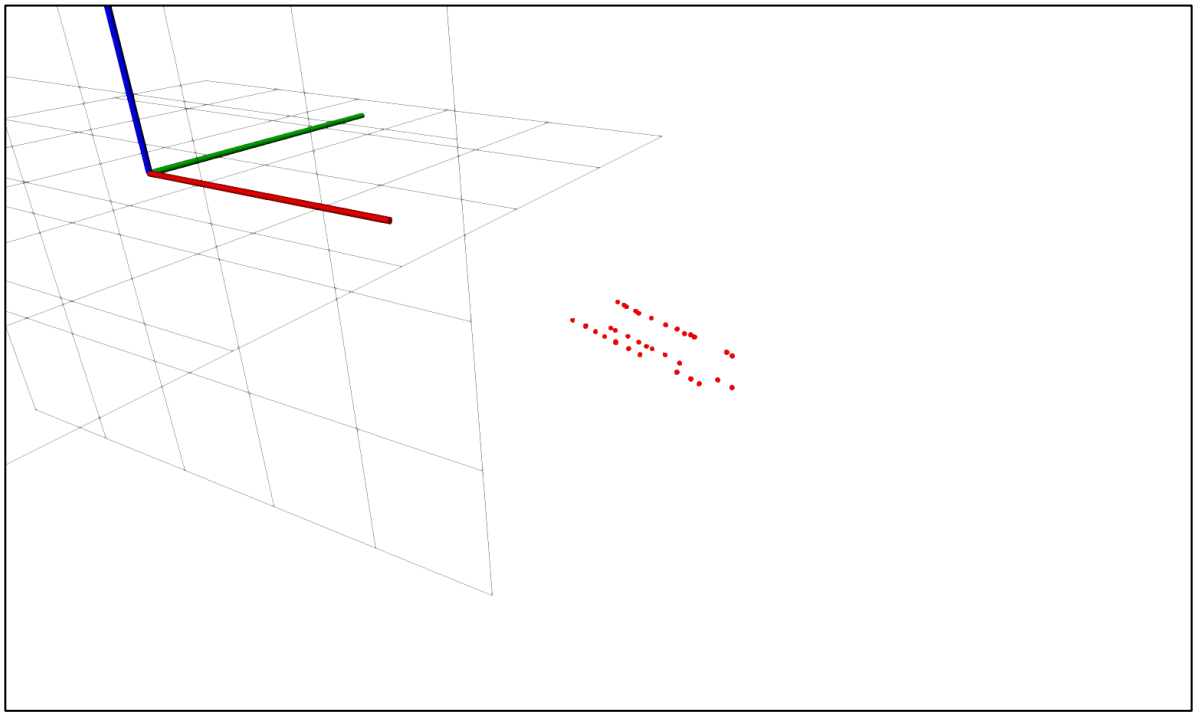


Figura 72. Nube 1 tras aplicar el algoritmo *VoxelGrid downsample*.

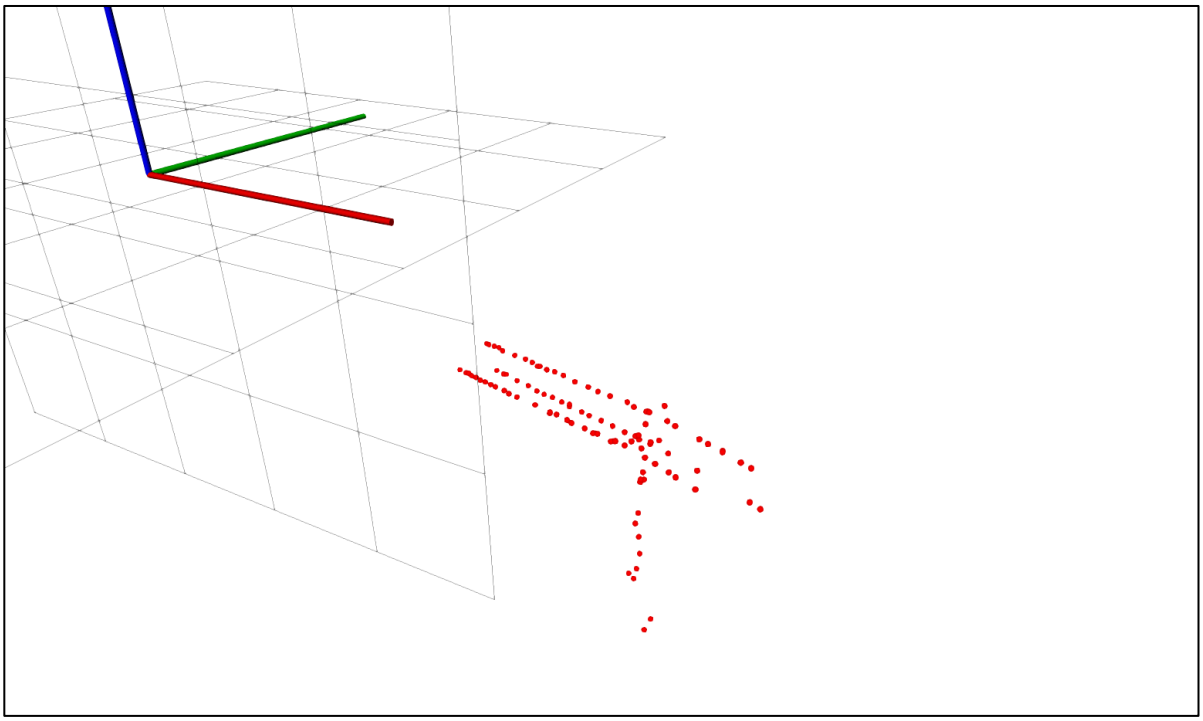


Figura 73. Nube 2 tras aplicar el algoritmo *VoxelGrid downsample*.

4.2.1.4 K-nearest search

El último paso del bloque de filtrado consiste en quedarnos con los *k* puntos más cercanos al origen. Esto nos permite obtener mejores modelos de línea y reduce en muchos casos el ruido que genera la presencia de las

torres de alta tensión. Los resultados se muestran en la Figura 74 y Figura 75.

El número de puntos elegido es KNEAREST=50, aproximadamente la mitad de puntos disponibles en presencia de las torres (Nube 2). Este valor no puede ser demasiado bajo, pues podríamos perder información de alguna de las tres líneas.

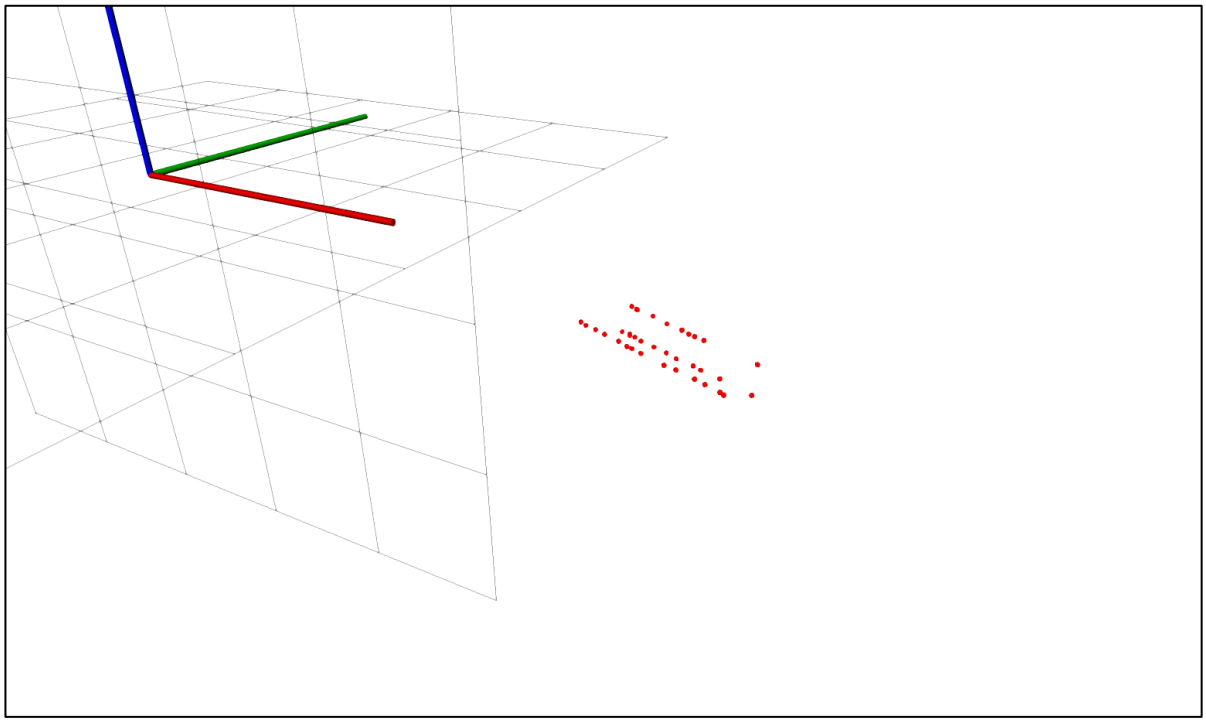


Figura 74. Nube 1 tras aplicar el algoritmo *K-nearest search*.

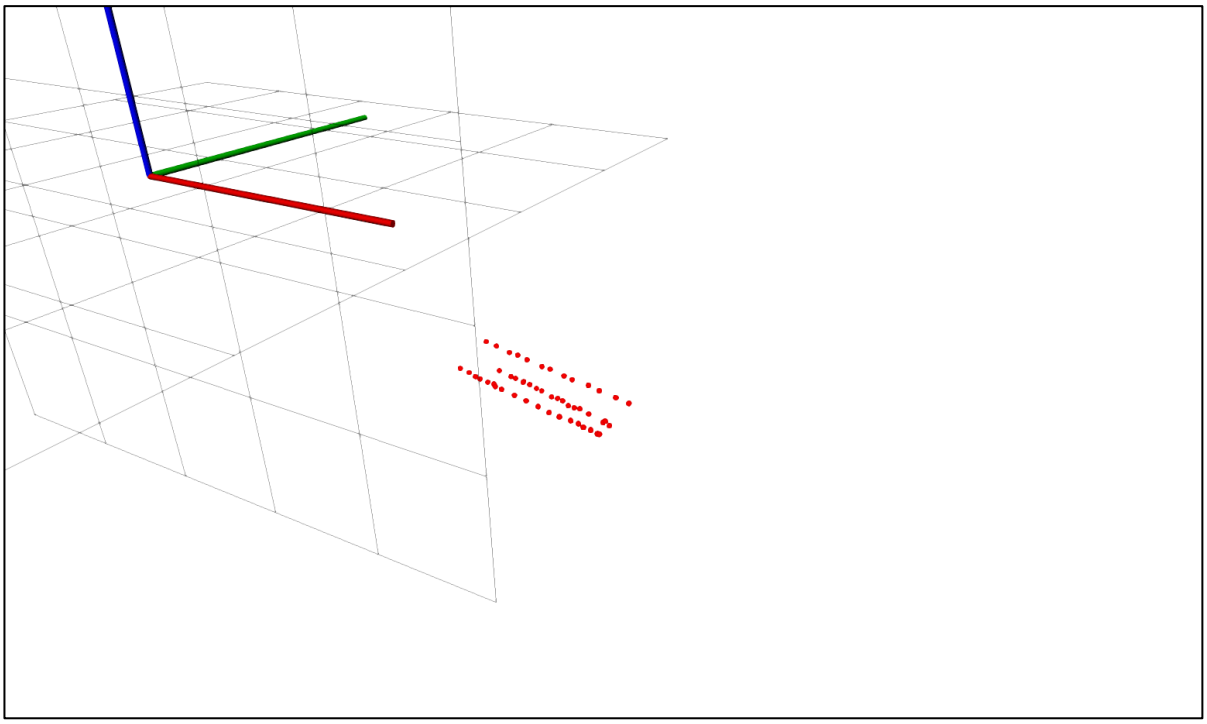


Figura 75. Nube 2 tras aplicar el algoritmo *K-nearest search*.

4.2.2 Concatenación

El bloque de filtrado nos ha proporcionado nubes limpias que representan fielmente la geometría de las líneas y de las torres. No obstante, como se aprecia en la Figura 74, es posible que se hayan perdido algunos puntos pertenecientes a las líneas bien a lo largo del proceso de filtrado o bien como consecuencia de la resolución limitada del LiDAR. Es por ello que, antes de pasar al bloque de segmentación, es conveniente aumentar artificialmente esta resolución concatenando las nubes de instantes justamente anteriores y posteriores. En la Figura 76 y Figura 77 se muestran los resultados de este proceso.

El número de nubes anteriores y posteriores a concatenar ha sido NCONCAT=1. Como se aprecia en la Figura 81, valores superiores han dado lugar a distorsiones en la nube de puntos debido al movimiento del dron.

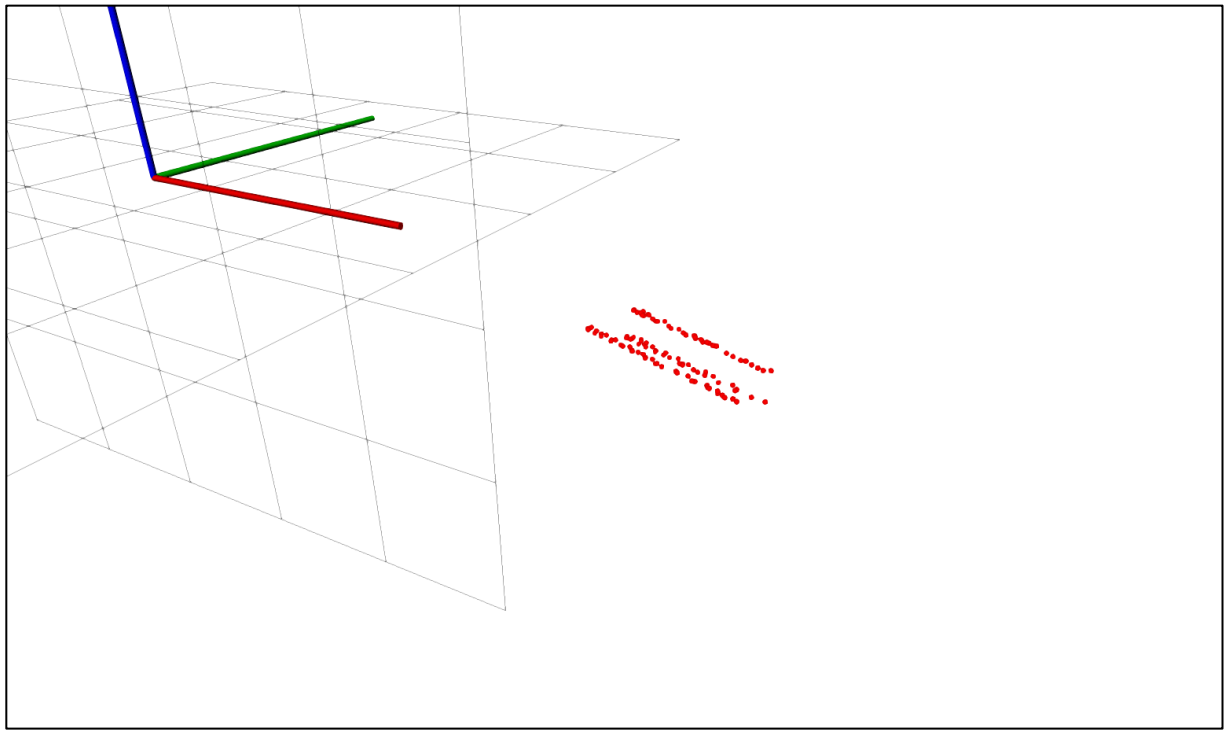


Figura 76. Nube 1 tras la concatenación de nubes anteriores y posteriores.

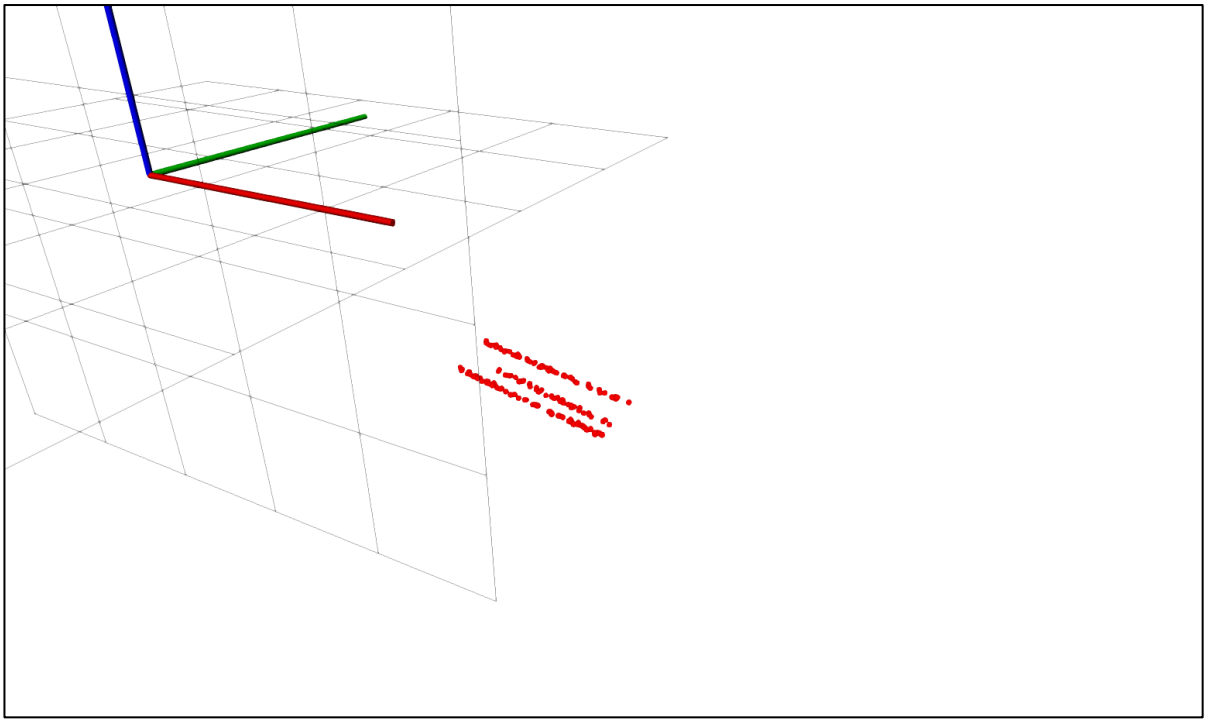


Figura 77. Nube 2 tras la concatenación de nubes anteriores y posteriores.

4.2.3 Segmentación

El ultimo bloque consiste en la segmentación de las líneas. El programa trata inicialmente de separar las líneas en tres *clusters* distintos, lo cual aceleraría la tarea de segmentación. Lo consiga o no, se aplica el método RANSAC para obtener las nubes finales segmentadas y los coeficientes de sus modelos. Los resultados finales se muestran en la Figura 78 y Figura 79. Puede verse el éxito de la segmentación, mostrándose cada *cluster* de un color diferente y el modelo de línea 3D ajustado para cada uno de estos.

Los parámetros para el *clustering* han sido EUCLTOL=1.1 y MINCLUSTSIZE=25. Respecto a los parámetros del algoritmo RANSAC, estos se recogen en la

Tabla 9. Para elegir estos parámetros se han realizado varias iteraciones, pues hay que encontrar un equilibrio entre el mínimo número de puntos que debe tener el modelo de línea y la precisión exigida al ajuste del modelo. Esta discusión se hará en el apartado siguiente.

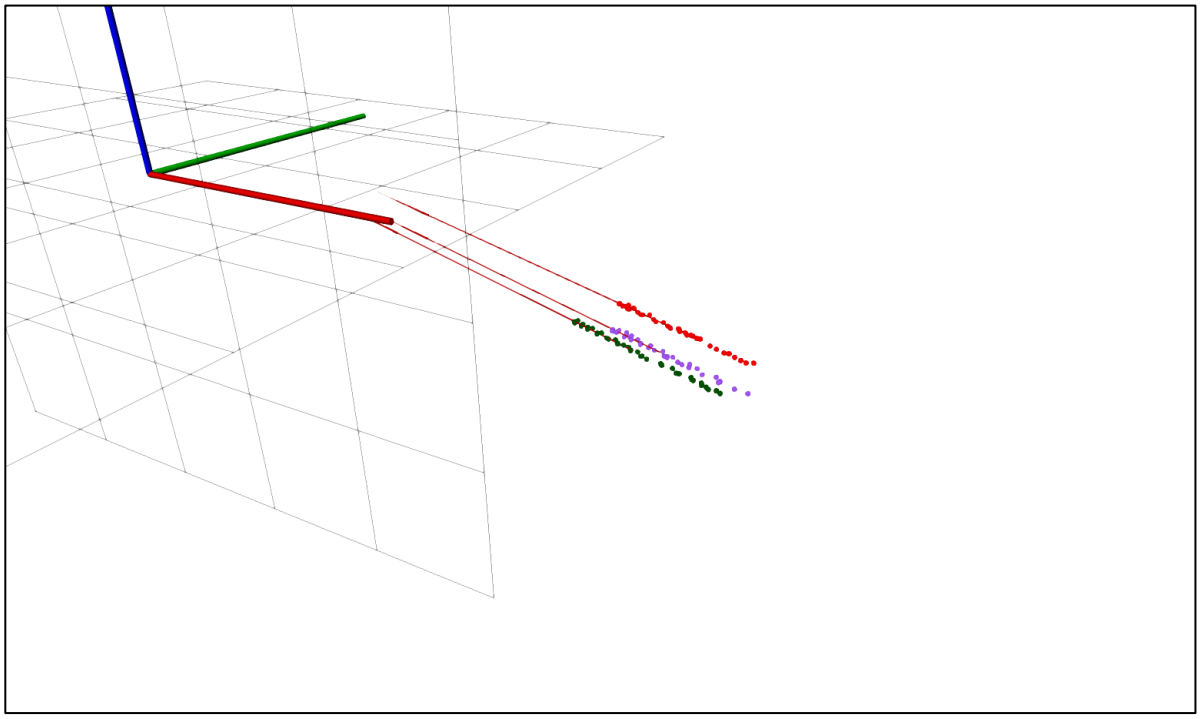


Figura 78. Nube 1 tras la segmentación de las líneas. Se muestra cada línea segmentada en un color distinto.

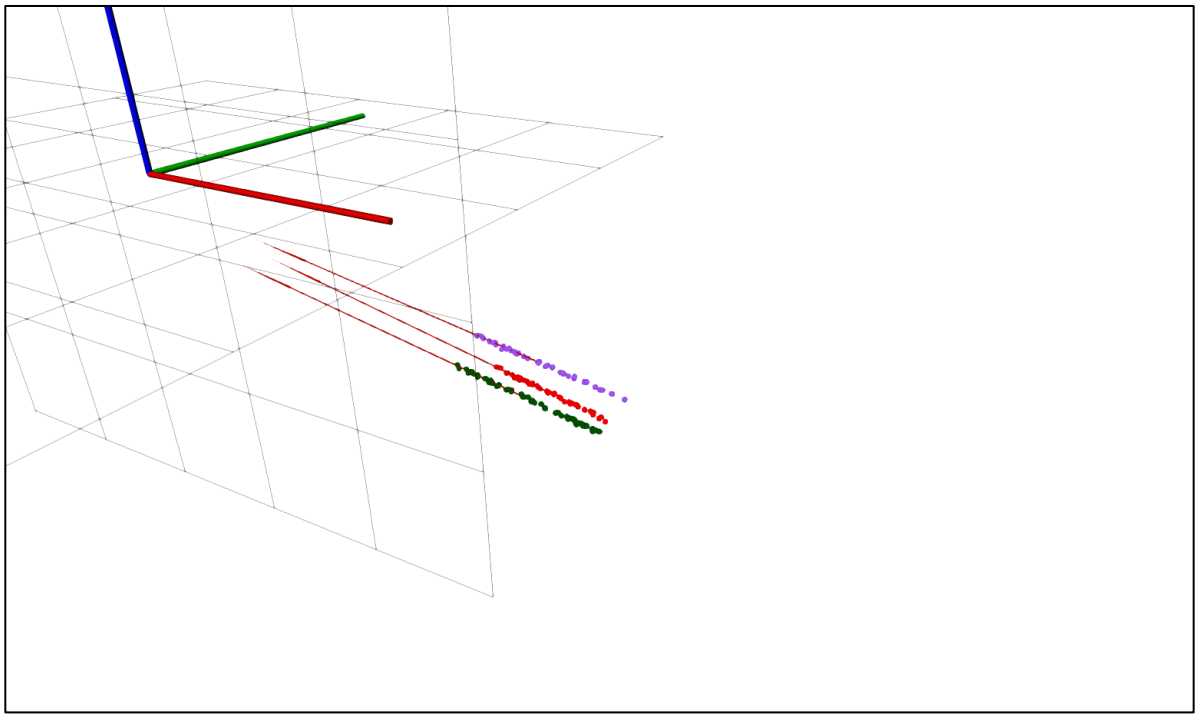


Figura 79. Nube 2 tras la segmentación de las líneas. Se muestra cada línea segmentada en un color distinto.

En la Figura 80 se muestra la línea temporal de las nubes resultado de cada uno de los pasos descritos. Pueden distinguirse pequeños tramos en los que se reconocen hasta cuatro modelos de línea (que se corresponden con el sobrevuelo de las torres) y en los que se reconocen sólo dos, aunque en términos generales el programa es capaz de segmentar las tres líneas sin dificultad.

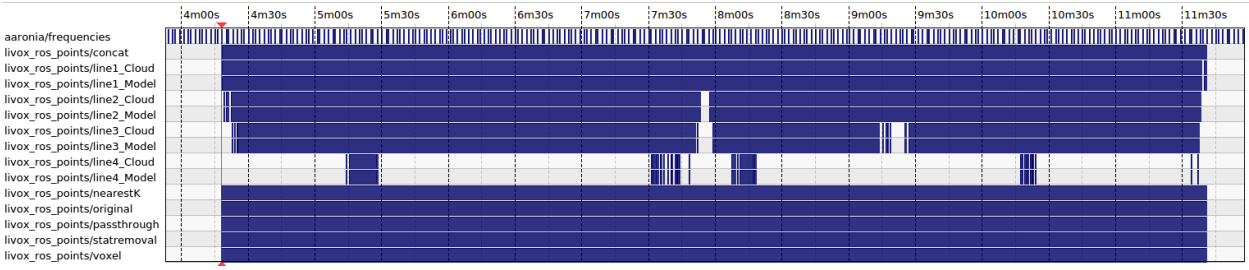


Figura 80. Línea temporal de las nubes resultado de los distintos pasos del procesamiento de nubes de puntos.

4.3 Ajuste paramétrico

Como se comentó en el capítulo previo, los algoritmos utilizados durante el procesamiento de nubes de puntos requieren de unos parámetros para su configuración. Este apartado trata de justificar la elección de los parámetros, mostrando la problemática más relevante que surge para una mala elección de estos. La descripción de estos parámetros se encuentra en la Tabla 3.

Parámetro	Valor	Justificación
MAXREFL	0.1	Límite establecido mediante inspección del mapa de color de la Figura 66
ZEROTOL	0.5	Debe ser menor que la distancia mínima a la que se acercará el UAV a la línea
MEANK	25	Suficiente para extraer el promedio de la distancia a los puntos vecinos
STDMULT	1.0	Suficiente para eliminar los puntos más dispersos
VLEAF	0.5	Debe ser inferior a la mínima distancia entre líneas de 1.5m
KNEAREST	50	Suficientemente grande para evitar perder información de las líneas más lejanas
NCONCAT	1	Un valor superior da lugar a distorsiones debido al movimiento del UAV
EUCLTOL	1.1	Ligeramente inferior que la mínima distancia entre líneas de 1.5m
MINCLUSTSIZE	25	Aproximación al repartir 100 puntos en 3 líneas distintas
RANSAC_MAXIT	200	Mayor que el número obtenido a través de la ecuación (3-1)
RANSAC_DIST1	0.5	Determinado iterativamente para mantener equilibrio con RANSAC_MININLIERS
RANSAC_DIST2	0.4	Determinado iterativamente para mantener equilibrio con RANSAC_MININLIERS
RANSAC_MININLIERS1	15	Determinado iterativamente para mantener equilibrio con RANSAC_DIST
RANSAC_MININLIERS2	20	Determinado iterativamente para mantener equilibrio con RANSAC_DIST

Tabla 9. Valores finales de los parámetros elegidos para el programa de procesamiento de nubes de puntos y su justificación.

En la Tabla 9 se recogen los parámetros finales utilizados y su justificación. A esto, caben añadir los siguientes comentarios sobre los principales problemas encontrados:

- **Distorsión de la nube tras concatenación:** como se ha comentado previamente, si se usan demasiadas nubes para la concatenación aparecerán distorsiones debido al movimiento del UAV. Esto puede visualizarse en la Figura 81.

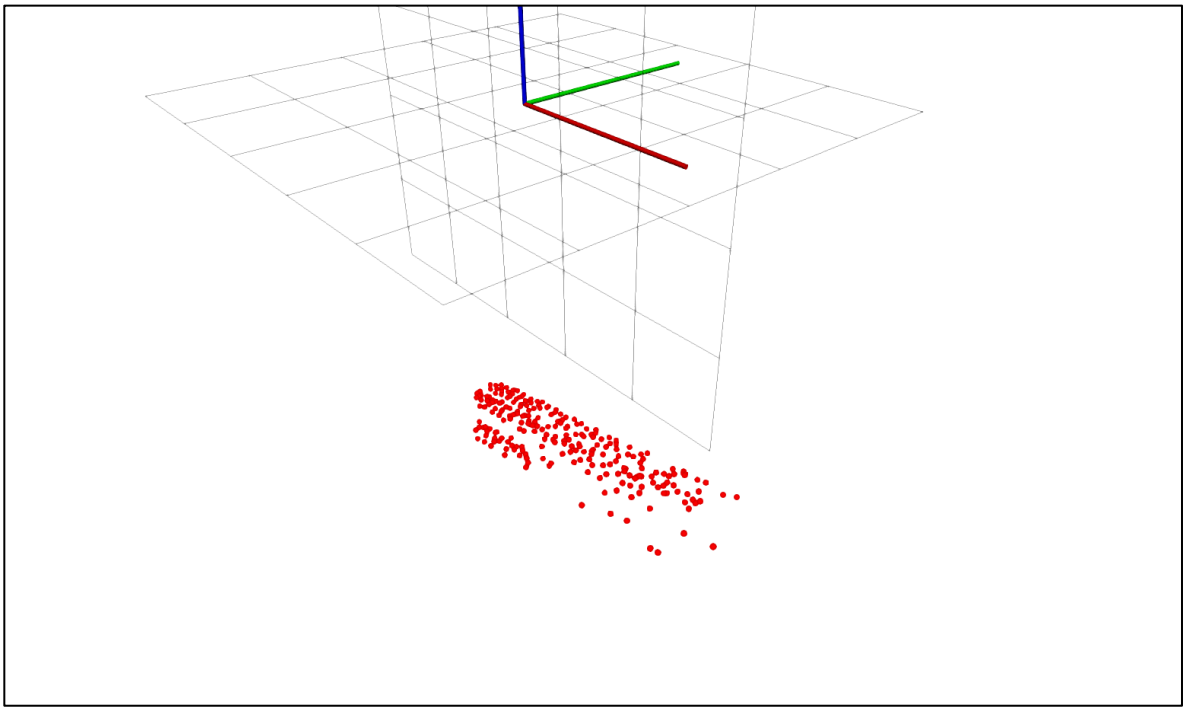


Figura 81. Distorsión de las líneas en una nube de puntos concatenada debido a un valor elevado del parámetro NCONCAT y al movimiento rápido del UAV.

- **Equilibrio entre precisión y número de puntos de los modelos RANSAC:** un modelo segmentado a través del algoritmo RANSAC se considerará válido si tiene un número mínimo de *inliers* dentro de una tolerancia determinada. Este equilibrio se consigue ajustando los parámetros RANSAC_MININLIERS y RANSAC_DIST a través de una serie de iteraciones. Por ejemplo, dado un determinado número de *inliers* mínimo, una tolerancia demasiado estrecha hará que encontremos menos líneas de las disponibles, mientras que una tolerancia demasiado amplia hará que aparezcan líneas espurias (como sucede en la Figura 82).

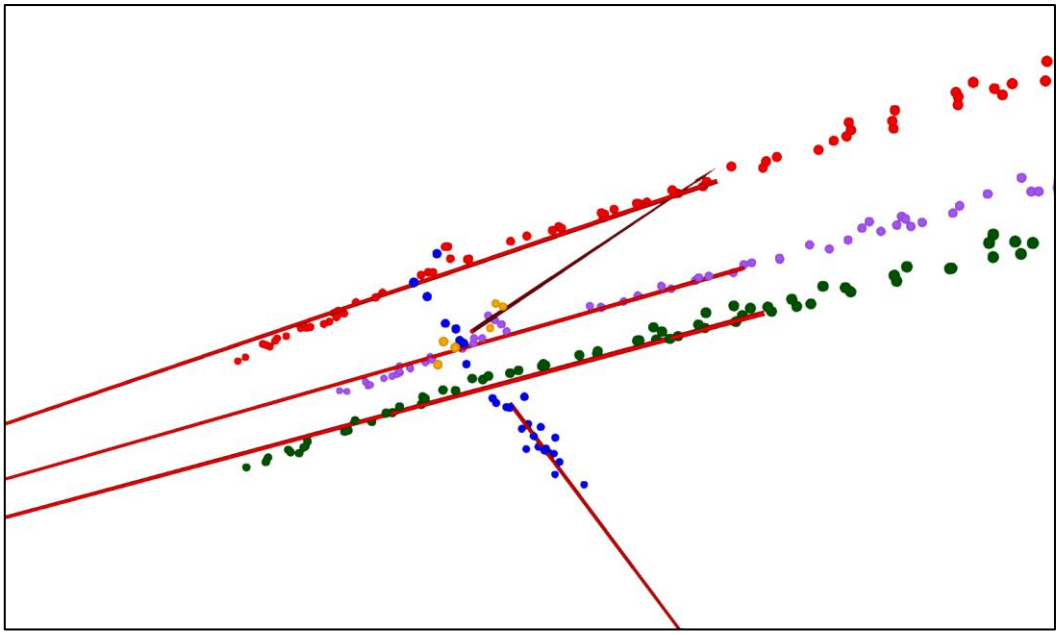


Figura 82. Segmentación fallida de una nube debido a un valor demasiado pequeño del parámetro RANSAC_MININLIERS. Se segmenta un conjunto de puntos (en color amarillo) que no se corresponde con una línea.

4.3.1 Proceso iterativo

Para la selección de los parámetros RANSAC_MININLIERS y RANSAC_DIST se ha llevado a cabo un proceso iterativo hasta encontrar la combinación que devuelve los mejores resultados respecto a porcentaje de líneas segmentadas con éxito. Cabe recordar del diagrama de la Figura 58 que estos parámetros pueden adoptar dos valores distintos atendiendo al número de puntos restantes a segmentar, luego a efectos prácticos debemos iterar un total de cuatro parámetros. Se ha considerado el valor SIZELIMIT=20 como tamaño de la nube para tomar esta decisión.

Esta iteración se ha realizado a modo de prueba y error, y los distintos parámetros y resultados se muestran en la Tabla 10. El resto de parámetros permanecen constantes e iguales a los vistos previamente.

Iteración	**1**	**2**	**3**	**4**
RANSAC_DIST1	0.80	0.40	0.50	0.50
RANSAC_DIST2	0.60	0.30	0.40	0.40
RANSAC_MININLIERS1	15	8	5	15
RANSAC_MININLIERS2	25	15	20	20
% Éxito	90%	92%	88%	93%

Tabla 10. Proceso iterativo para encontrar los parámetros óptimos de segmentación. El % de éxito es el porcentaje de nubes para las cuales se han segmentado 3 o 4 líneas (correspondientes a las tres líneas y a la torre).

Puede verse que el programa es capaz de segmentar las líneas con un gran porcentaje de éxito. También es capaz de segmentar las torres eléctricas por separado, dando lugar a una línea aproximadamente perpendicular al suelo (Figura 83). Esto no supone problema alguno, pues durante el postprocesamiento podremos descartar estas líneas

adicionales.

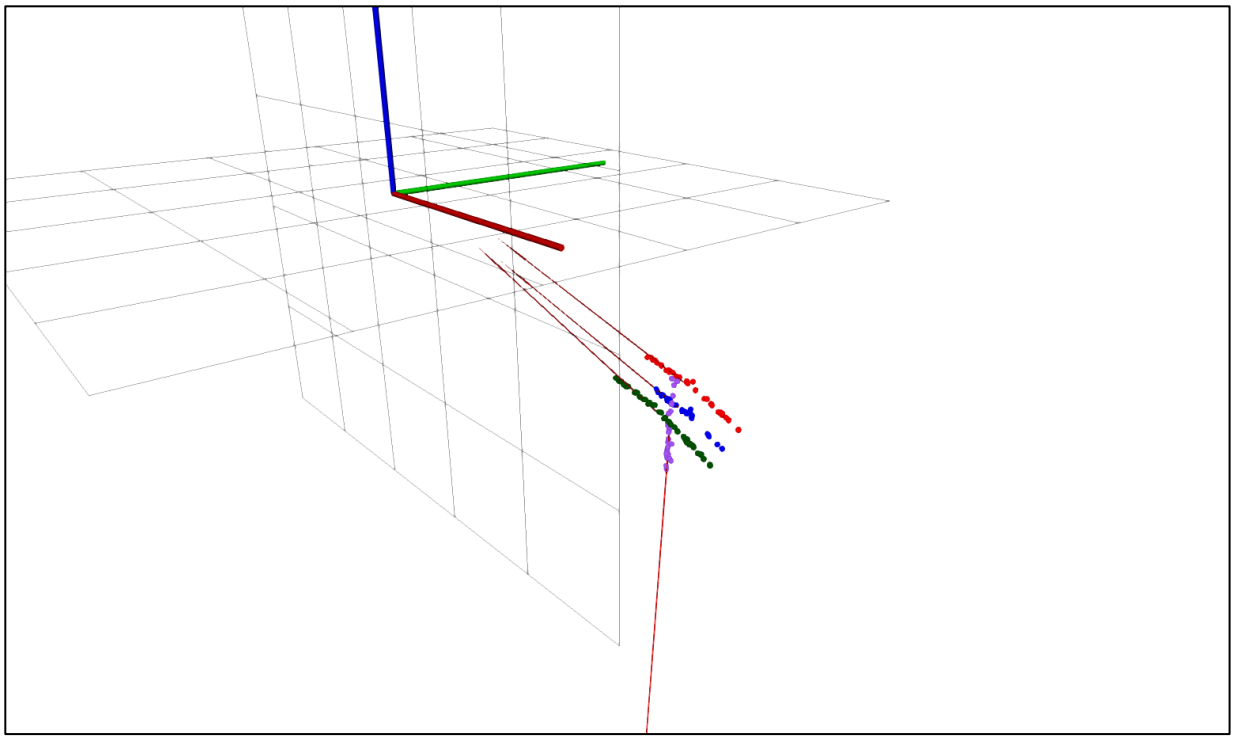


Figura 83. Segmentación de una nube en presencia de una torre. Puede verse que el programa es capaz de segmentar cada una de las líneas y la torre por separado.

4.4 Resultados tras postprocesamiento

El resultado del programa anterior, más allá de la visualización de las nubes, es un archivo de texto donde se guardan los coeficientes obtenidos línea por línea. Siguiendo el procedimiento descrito en el capítulo anterior, estas líneas han sido leídas para su postprocesamiento en un programa de *Python*. El resultado del postprocesamiento es la evolución temporal de la distancia media del dron al tendido eléctrico, y puede verse en la Figura 84 y Figura 85 para cada uno de los archivos procesados. Para el archivo del cual se han mostrado se han descartado 297 de las 4425 nubes segmentadas que no cumplían con los requisitos. El tiempo está referenciado al primer instante del que se tiene información del campo magnético.

En la primera gráfica, si descartamos el primer tramo en el que se sobrevuela por un tendido secundario, se puede apreciar que el UAV sigue las líneas manteniéndose a una distancia comprendida entre 12 y 16m. Así mismo, el seguimiento de la distancia es prácticamente continuo salvo pequeños intervalos.

En la segunda gráfica, que se corresponde con el vuelo experimental que sobrevuela las torres de alta tensión, tenemos una mayor variación de las distancias. Concretamente se puede apreciar el tramo inicial de acercamiento a la línea seguido de tres aproximaciones repentinas hasta los 8m.

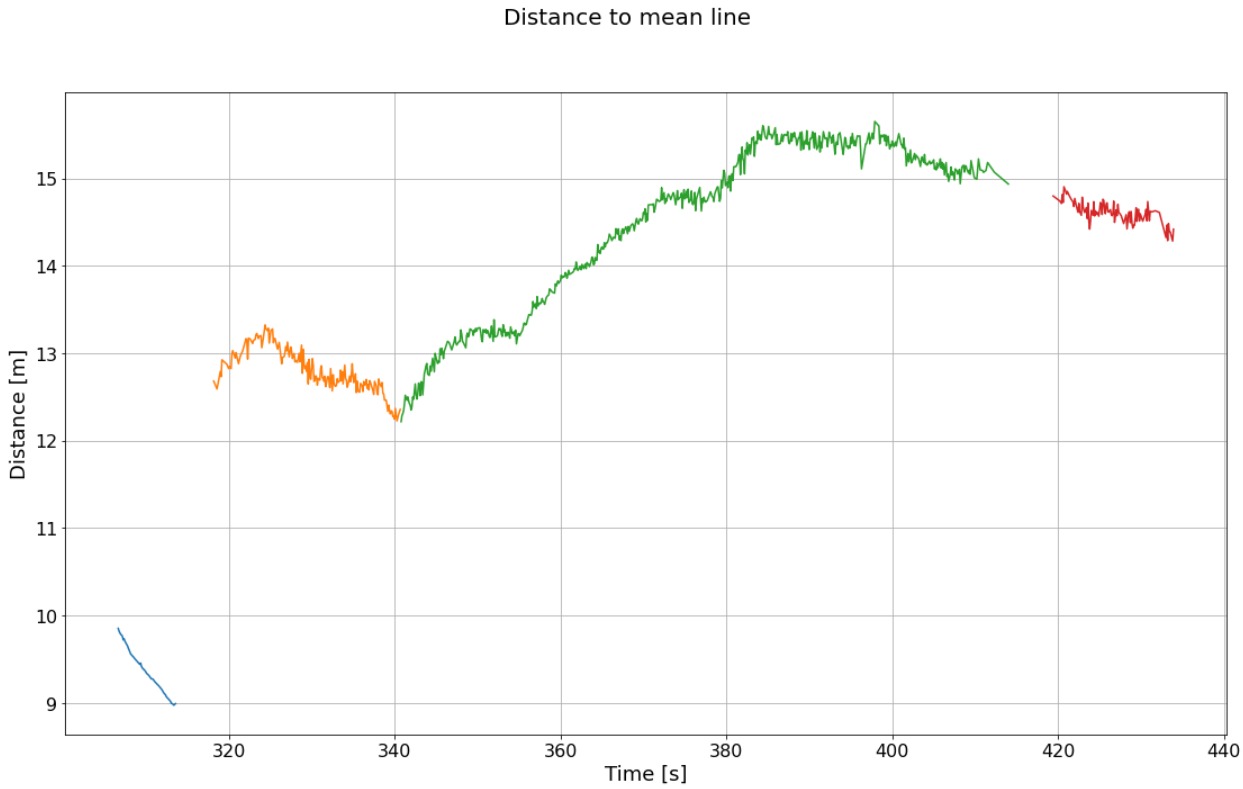


Figura 84. Distancia media al tendido eléctrico frente al tiempo obtenida para las nubes procesadas del archivo pc_n_emf.bag. Puede apreciarse un pequeño tramo inicial en el que se sobrevuela un téndido eléctrico próximo.

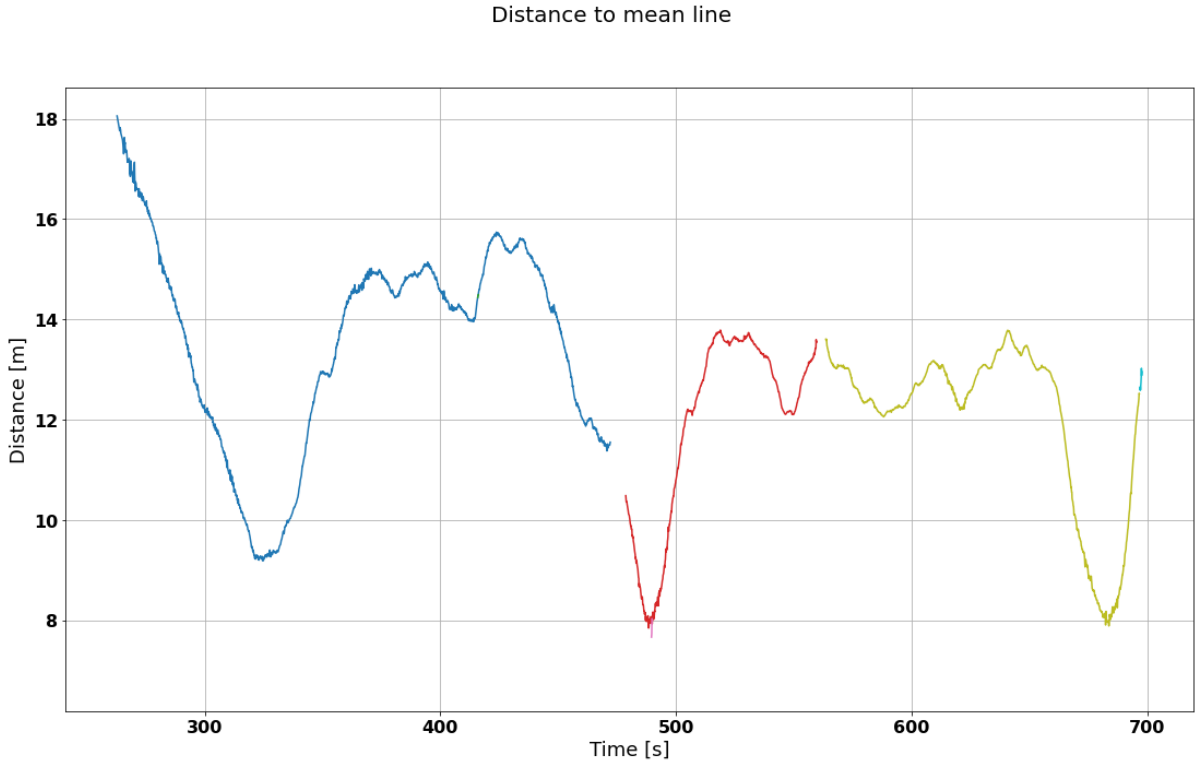


Figura 85. Distancia media al tendido eléctrico frente al tiempo obtenida para las nubes procesadas del archivo livox_pcd_points_burguillos_exp_002_single.bag.

En paralelo al cálculo de la distancia media de las líneas, en este programa de postprocesamiento también se ha leído la información del campo magnético. Puesto que estos datos no han requerido de ningún tratamiento especial, simplemente se recoge una muestra del espectro de potencia de la señal para distintos instantes de tiempo:

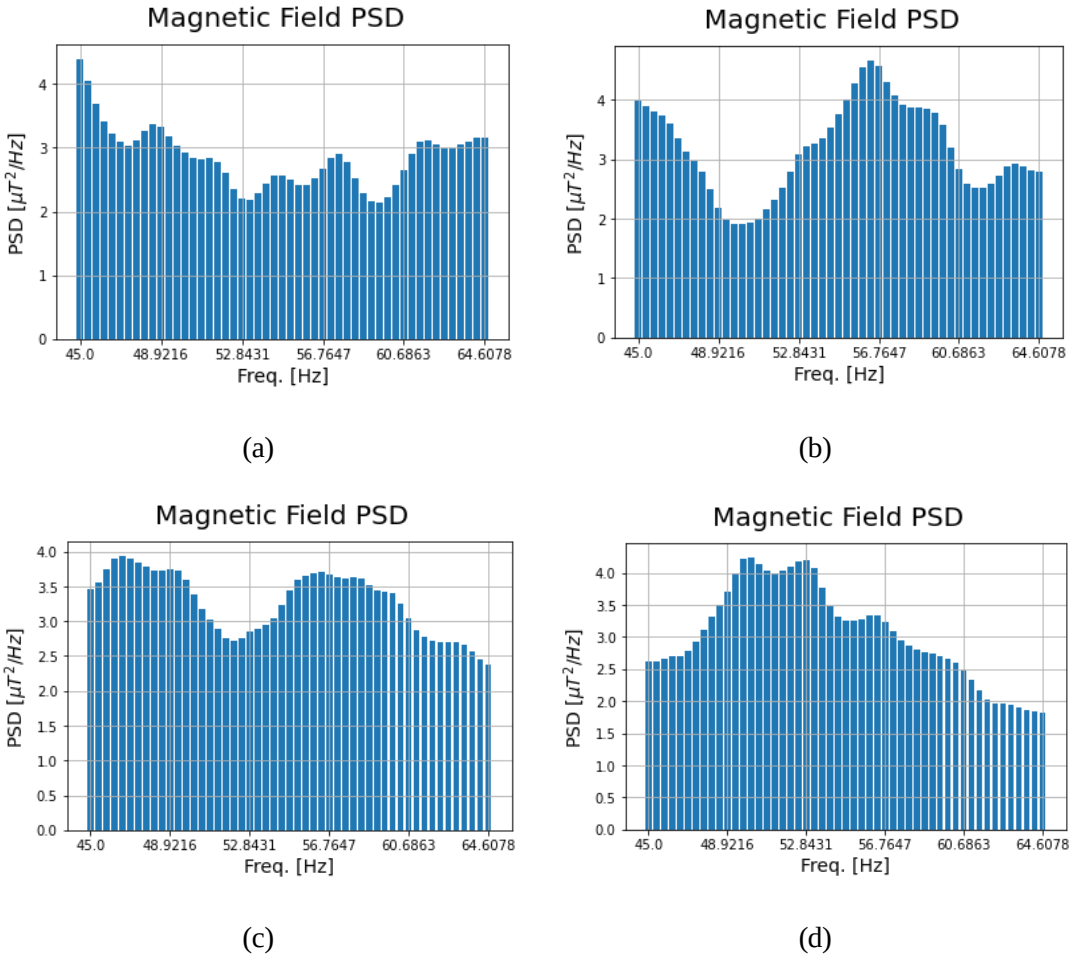


Figura 86. Espectro de potencia de la señal de campo magnético para distintos instantes de tiempo del seguimiento de las líneas de alta tensión.

4.4.1 Comprobación de resultados

A modo de comprobación, visualizaremos y estimaremos manualmente la distancia para distintos *timestamps* de la Figura 85 usando nuevamente la herramienta *RViz*. Este proceso se recoge en las siguientes figuras:

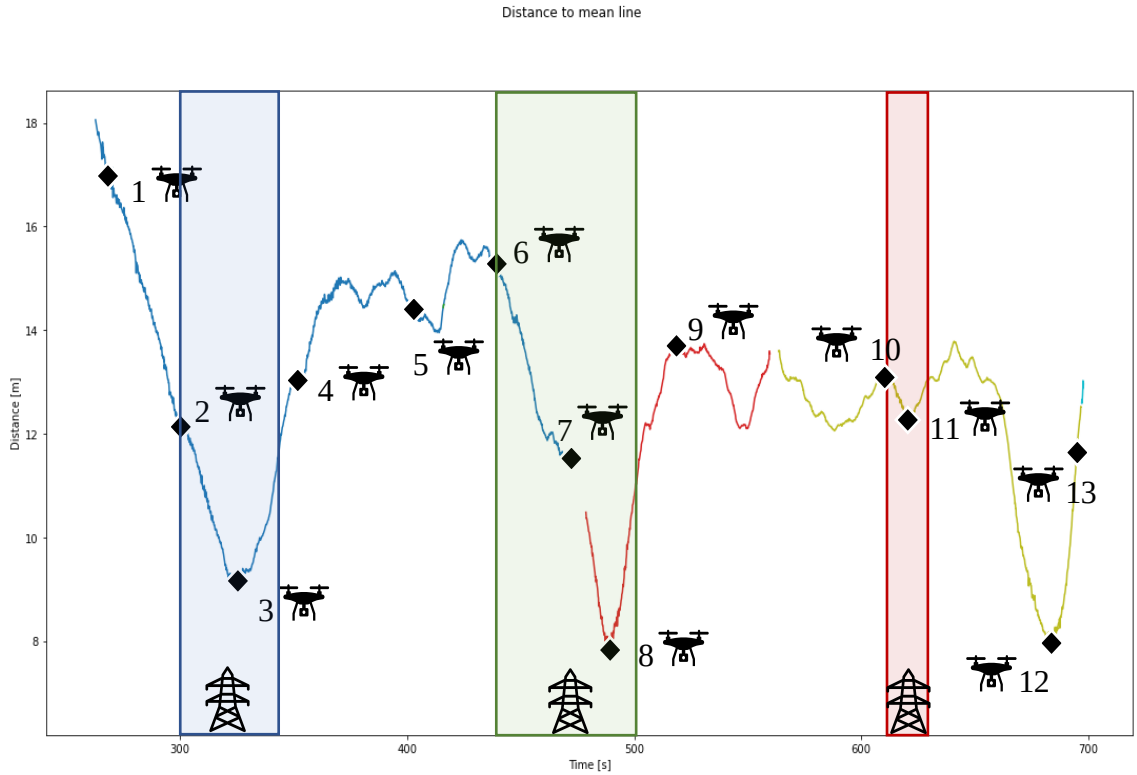
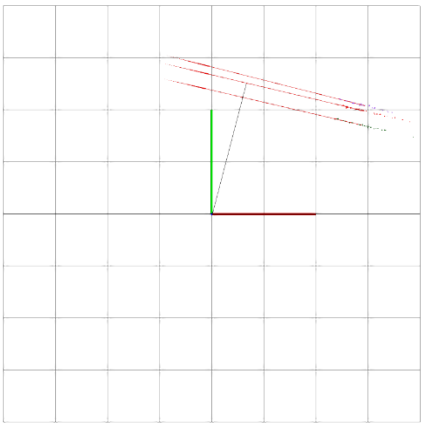
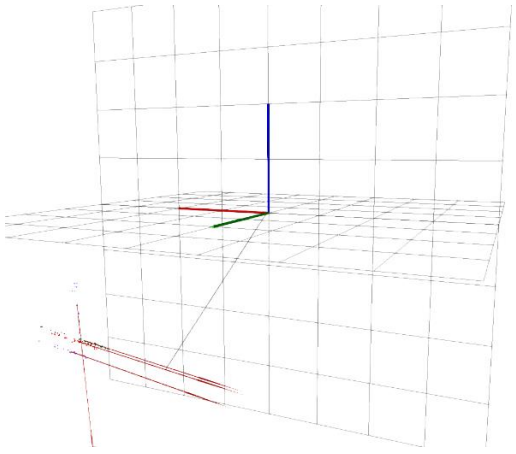
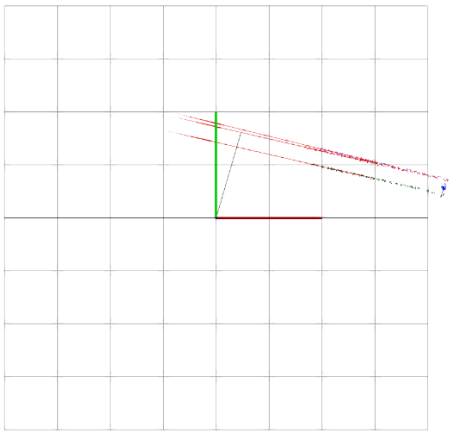
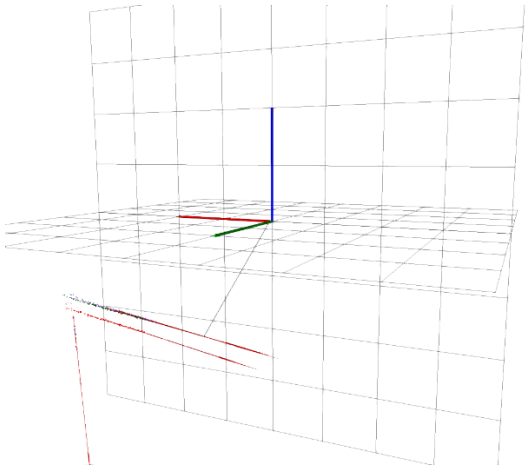


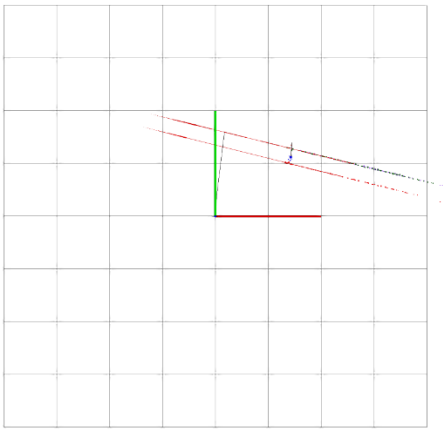
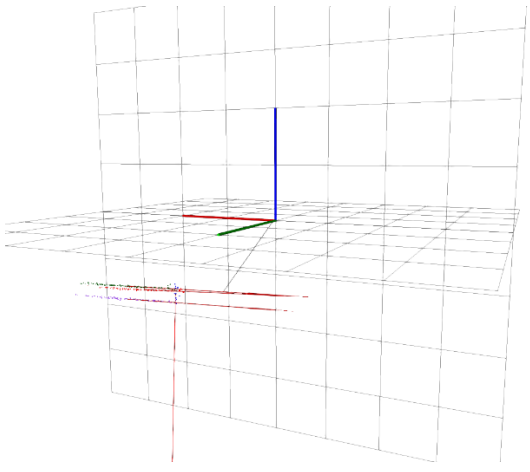
Figura 87. Enumeración de los distintos timestamps usados para la comprobación de los resultados de la distancia media frente al tiempo. Los tramos de colores se corresponden con el sobrevuelo de torres eléctricas.

Timestamp 1: 270s (d≈17m)	
Vista superior	Vista lateral
	

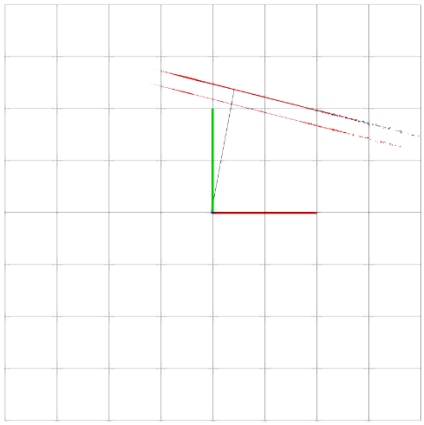
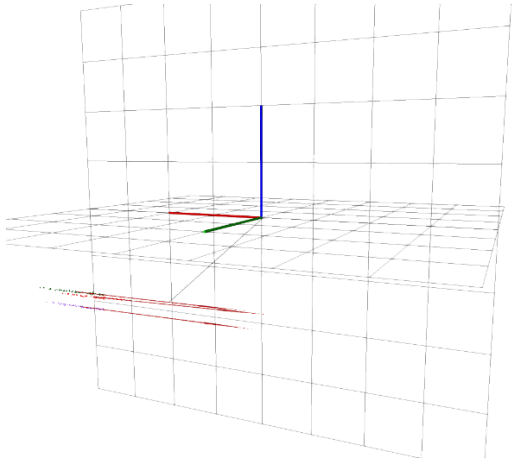
(a)

Timestamp 2: 300s (d≈12m)	
Vista superior	Vista lateral
	

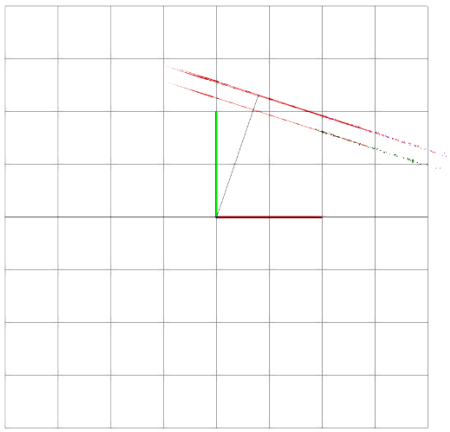
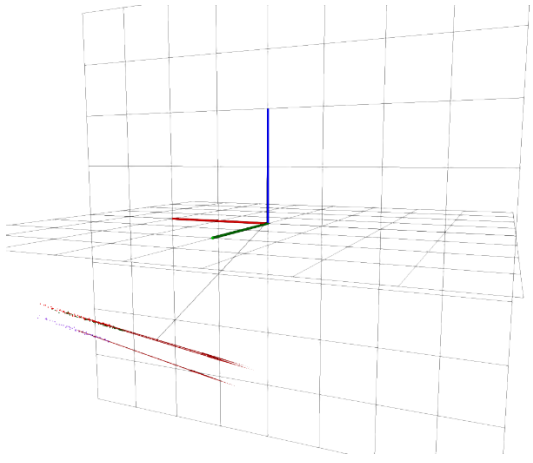
(b)

Timestamp 3: 325s (d≈9m)	
Vista superior	Vista lateral
	

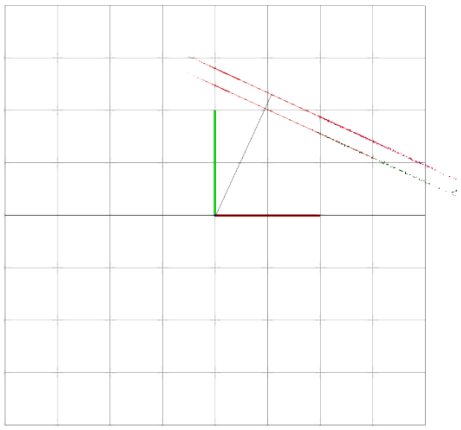
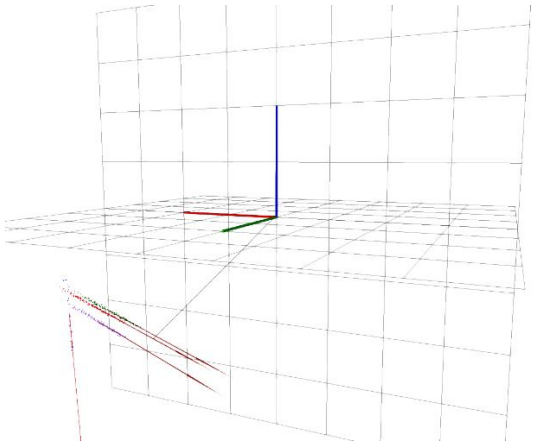
(c)

Timestamp 4: 350s (d≈13m)	
Vista superior	Vista lateral
	

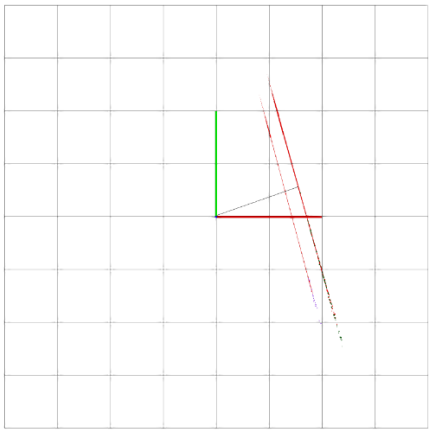
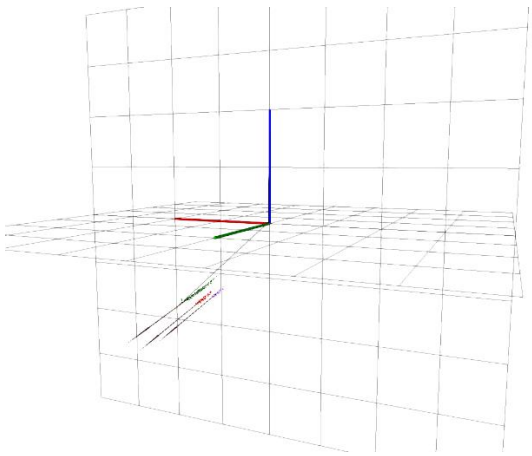
(d)

Timestamp 5: 400s (d≈14m)	
Vista superior	Vista lateral
	

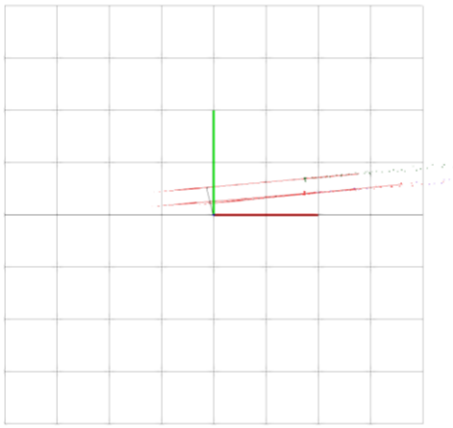
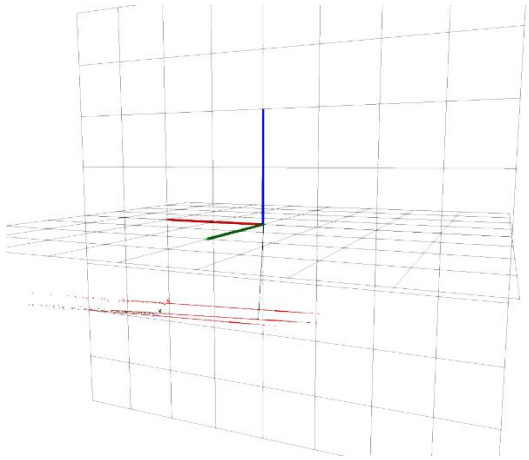
(e)

Timestamp 6: 430s (d≈15m)	
Vista superior	Vista lateral
	

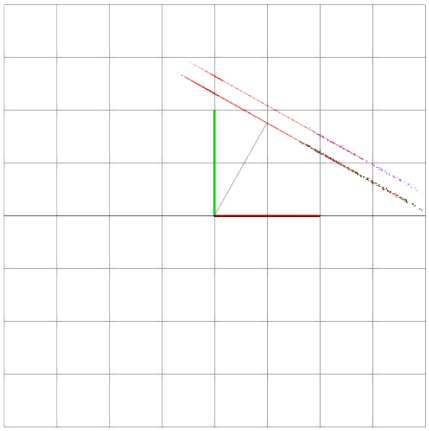
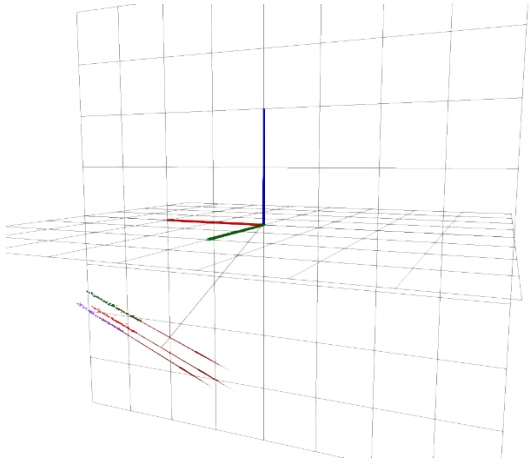
(f)

Timestamp 7: 470s (d≈11.50m)	
Vista superior	Vista lateral
	

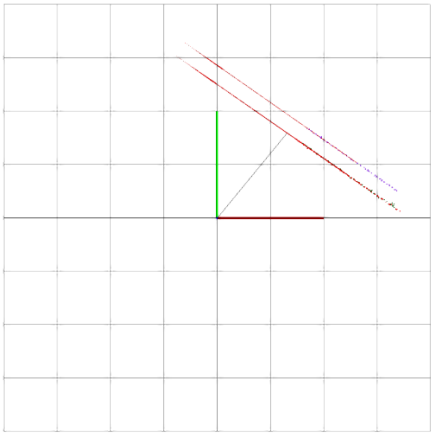
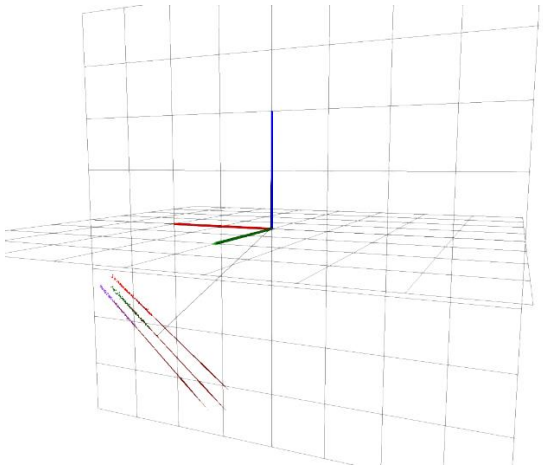
(g)

Timestamp 8: 485s (d≈8m)	
Vista superior	Vista lateral
	

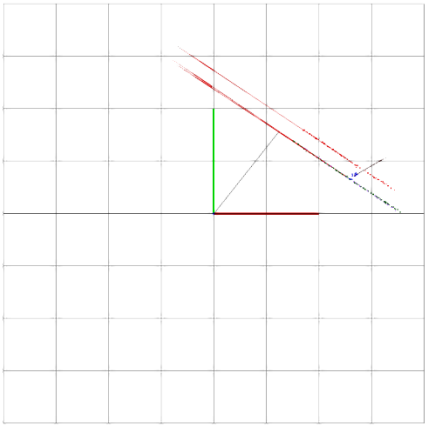
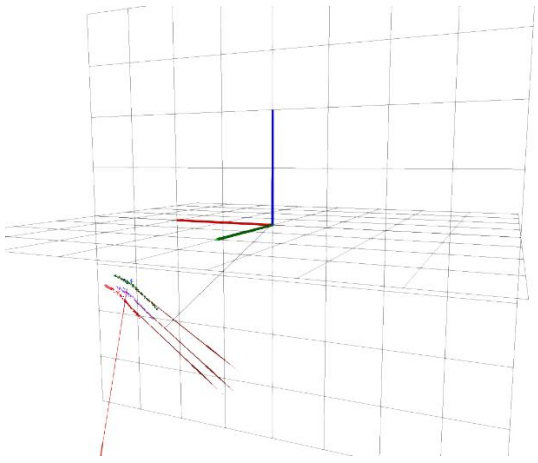
(h)

Timestamp 9: 520s (d≈13.50m)	
Vista superior	Vista lateral
	

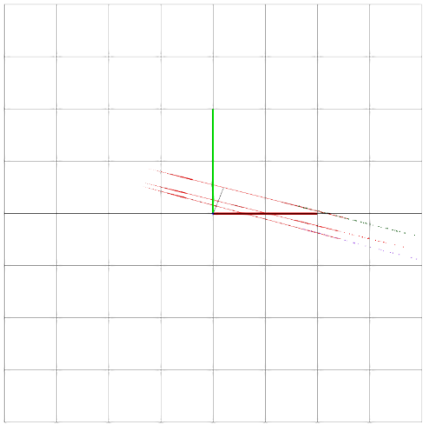
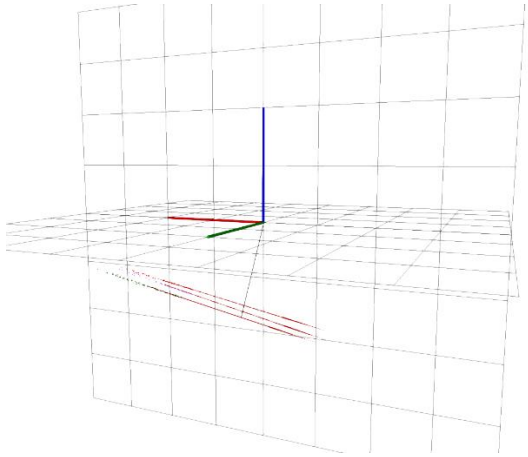
(i)

Timestamp 10: 610s (d≈13m)	
Vista superior	Vista lateral
	

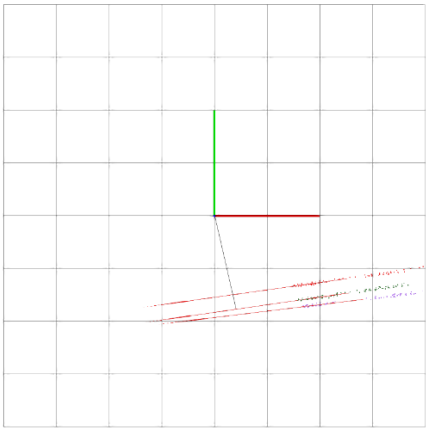
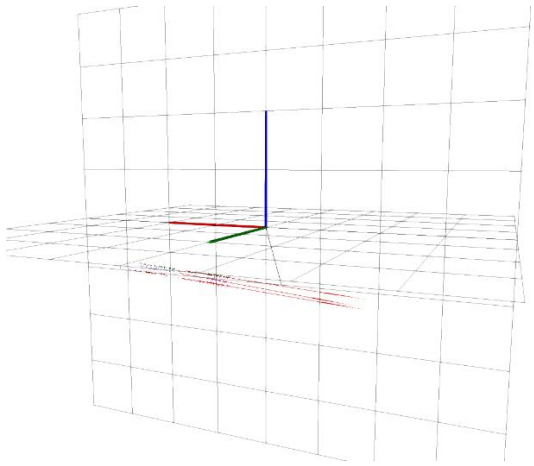
(j)

Timestamp 11: 615s (d≈12.50m)	
Vista superior	Vista lateral
	

(k)

Timestamp 12: 680s (d≈8.50m)	
Vista superior	Vista lateral
	

(l)

Timestamp 13: 695s (d≈11.50m)	
Vista superior	Vista lateral
	

(m)

Figura 88. Vistas de las líneas segmentadas para los distintos *timestamps* representados en la Figura 87 (a)-(m).

Inspeccionando las figuras anteriores se llega a la conclusión de que existe una correspondencia entre la distancia obtenida manualmente con este procedimiento y la representada en la Figura 87.

No obstante, es preciso destacar un par de consideraciones que pueden afectar a la precisión de los resultados obtenidos:

- **Efecto de la catenaria:** tal y como se describió en el Capítulo 3, se ha empleado el modelo de línea 3D para modelar los cables conductores ya que la cercanía del UAV al tendido eléctrico y el campo de visión limitado del LiDAR hacían imposible distinguir la curva catenaria. No obstante, la aproximación hecha afectará a la precisión de la distancia calculada puesto que el LiDAR no apunta perpendicularmente a las líneas (ver Figura 89).

- **Efecto del sobrevuelo de torres de alta tensión:** debido precisamente al efecto de la catenaria, el procedimiento empleado tendrá dificultades para calcular la distancia media al tendido eléctrico en las zonas cercanas a torres de alta tensión. Esto se debe al cambio de pendiente de los cables conductores a un lado y al otro de la torre. Este fenómeno se visualiza en la Figura 89, y se hace notar en el tramo rojo de la Figura 87 como una oscilación de la distancia calculada.

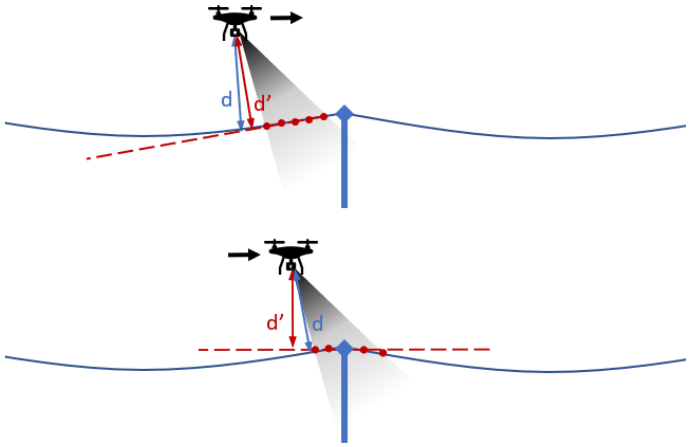


Figura 89. Visualización de las dificultades para calcular la distancia a la línea en proximidades a las torres. Conforme avanza el UAV se producirán oscilaciones de la pendiente que afectarán a la distancia calculada. En las nubes disponibles el LiDAR mira hacia adelante en la dirección de avance del UAV, y no perpendicular a la línea.

5 CONCLUSIONES Y LÍNEAS FUTURAS

5.1 Conclusiones

- Del estudio del estado del arte presentado y de las numerosas referencias consultadas puede concluirse que la inspección de líneas de alta tensión para su mantenimiento preventivo mediante sistemas UAV es de gran interés actual para la comunidad investigadora. En concreto, las soluciones que emplean LiDAR y magnetómetros han demostrado tener un enorme potencial para la caracterización de la línea y la navegación a lo largo de ella.
- Ha quedado expuesto todo el potencial de la librería PCL y su uso en conjunto con el entorno ROS, el estándar a nivel global para la programación de robots. Esta librería, una vez dominada, pone a nuestra disposición una serie de herramientas para implementar de forma rápida todo tipo algoritmos punteros en el tratamiento de nubes de puntos. El Capítulo 3 busca ser una introducción para cualquier persona que quiera iniciarse en el uso de estas herramientas.
- Se ha programado una herramienta en C++ que permite, de manera robusta y eficiente, procesar las nubes de puntos contenidas en un archivo *bag* de ROS y correspondientes a un vuelo experimental de sobrevuelo de líneas de alta tensión. Los diversos algoritmos de filtrado y segmentación pertenecientes a la librería PCL han demostrado ser una herramienta eficaz para obtener los coeficientes de las líneas segmentadas. Así mismo, se ha creado un esquema de lectura y escritura para poder visualizar los resultados de estos algoritmos en cada paso.
- En este sentido, el filtro basado en el valor de la intensidad LiDAR en cada punto ha probado ser con diferencia la mejor herramienta para aislar el tendido eléctrico del resto de puntos.
- El programa desarrollado ha sido utilizado para procesar dos archivos de experimentos reales con excelentes resultados. El programa ha sido capaz de procesar más de 6000 nubes en menos de un minuto en un equipo de sobremesa. Los resultados revelan un éxito de segmentación de nubes del 93%, proporcionándonos un seguimiento casi continuo de la distancia media a la línea.
- La evolución de la distancia media con el tiempo se corresponde con las observaciones realizadas, verificando así los resultados obtenidos y validando la herramienta para su uso.
- Por último, y no menos importante, la consecución de este proyecto ha supuesto para el alumno adentrarse en una temática poco familiar, lo que le ha permitido crecer académica, personal y profesionalmente.

5.2 Líneas futuras

- La continuación más inmediata y obvia de este proyecto es la de implementar modelos de *machine learning* a partir de los datos extraídos de la distancia media al tendido eléctrico y el espectro de potencia. Estos modelos tratarán de buscar una correlación entre la distancia y el espectro, usando este último como variable explicativa. Atendiendo al volumen de datos disponible se pueden idear modelos más sencillos como SVMs (*Support Vector Machines*) o *Random forests*, o modelos más complejos como redes neuronales (*Neural Networks*), para los cuales se necesita un gran volumen de datos.
- El propósito final de esto último sería el de implementar un sistema de navegación capaz de utilizar las medidas del campo magnético para seguir el tendido eléctrico durante tareas de inspección. Siguiendo esta línea hay una gran cantidad de trabajo por delante en forma de estudios y de vuelos experimentales.
- Así mismo, podrían considerarse líneas de mejora para el programa desarrollado. Podrían considerarse algoritmos de segmentación más sofisticados como la Transformada de Hough aplicada a 3D o descriptores de características locales como los PFHs (*Point Feature Histograms*). Por otro lado, sería interesante incorporar un tratamiento especial para las torres eléctricas evitando así que interfieran en la estimación de la distancia, o bien considerar modelos de catenaria locales.
-

REFERENCIAS

[1] Wikipedia, la enciclopedia libre, «Energía Eléctrica en España,» [En línea]. Available: https://es.wikipedia.org/wiki/Energ%C3%ADa_el%C3%A9ctrica_en_Espa%C3%B1a. [Último acceso: 16 Octubre 2021].

[2] US Energy Information Administration, «International Energy Outlook,» [En línea]. Available: https://www.eia.gov/outlooks/ieo/. [Último acceso: 17 Octubre 2021].

[3] Red Eléctrica de España, «Mapa transporte ibérico,» 2018. [En línea]. Available: https://www.ree.es/sites/default/files/01_ACTIVIDADES/Documentos/Mapas-de-red/mapa_transporte_iberico_2018.pdf.

[4] Red Eléctrica de España, «Red Eléctrica de España,» [En línea]. Available: https://www.ree.es/es. [Último acceso: 18 Octubre 2021].

[5] A. Pagnano, M. Höpf y R. Teti, «A roadmap for automated power line inspection. Maintenance and repair.,» *Procedia CIRP*, vol. 12, pp. 234-239, 2013.

[6] J. Katrasnik, F. Pernus y B. Likar, «New Robot for Power Line Inspection,» de *2008 IEEE Conference on Robotics, Automation and Mechatronics*, 2008.

[7] Z. Li, Y. Liu, R. Hayward, J. Zhang y J. Cai, «Knowledge-based power line detection for UAV surveillance and inspection systems,» de *2008 23rd International Conference Image and Vision Computing New Zealand*, 2008.

[8] L. Fu y S. Lu, «Obstacle detection algorithms for aviation,» de *2011 IEEE International Conference on Computer Science and Automation Engineering*, 2011.

[9] G. Zhou, J. Yuan, I.-L. Yen y F. Bastani, «Robust real-time UAV based power line detection and tracking,» de *2016 IEEE International Conference on Image Processing (ICIP)*, 2016.

[10] F. Tian, Y. Wang y L. Zhu, «Power line recognition and tracking method for UAVs inspection,» de *2015 IEEE International Conference on Information and Automation*, 2015.

[11] C. Sun, R. Jones, H. Talbot, X. Wu, K. Cheong, R. Beare, M. Buckley y M. Berman, «Measuring the distance of vegetation from powerlines using stereo vision,» *Isprs Journal of Photogrammetry and Remote Sensing*, vol. 60, nº 4, pp. 269-283, 2006.

[12] C. Chen, B. Yang, S. Song, X. Peng y R. Huang, «Automatic Clearance Anomaly Detection for Transmission Line Corridors Utilizing UAV-Borne LIDAR Data,» *Remote Sensing*, vol. 10, nº 4, p. 613, 2018.

[13] Y. Kobayashi, G. Karady, G. Heydt y R. Olsen, «The Utilization of Satellite Images to Identify Trees Endangering Transmission Lines,» *IEEE Transactions on Power Delivery*, vol. 24, nº 3, pp. 1703-1709, 2009.

[14] G. Sohn, Y. Jwa y H. B. Kim, «AUTOMATIC POWERLINE SCENE CLASSIFICATION AND RECONSTRUCTION USING AIRBORNE LIDAR DATA,» *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, nº 1, pp. 167-172, 2012.

[15] H. B. Kim y G. Sohn, «Point-based Classification of Power Line Corridor Scene Using Random Forests,» *Photogrammetric Engineering and Remote Sensing*, vol. 79, nº 9, pp. 821-833, 2013.

[16] German Aerospace Center, «German Aerospace Center (DLR) Deutsches Zentrum für Luft- und Raumfahrt,» 2021. [En línea]. Available: https://www.dlr.de/EN/Home/home_node.html. [Último acceso: 18 10 2021].

[17] M. Nagai, T. Chen, R. Shibasaki, H. Kumagai y A. Ahmed, «UAV-Borne 3-D Mapping System by Multisensor Integration,» *IEEE Transactions on Geoscience and Remote Sensing*, vol. 47, nº 3, pp. 701-708, 2009.

[18] D. Rosner, C. Trifu, C. Tranca, I. Vasilescu y F. Stancu, «Magnetic Field Sensor for UAV Power Line Acquisition and Tracking,» de *2018 17th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, 2018.

[19] P. Wang, K. F. Goddard, P. L. Lewin, S. G. Swingler, P. Atkins y K. Y. Foo, «Magnetic Field Measurement to Detect and Locate Underground Power Cable,» de *International Conference on Pipelines and Trenchless Technology 2011*, 2011.

[20] Y. Wu, G. Zhao, J. Hu, Y. Ouyang, S. X. Wang, J. He, F. Gao y S. Wang, «Overhead Transmission Line Parameter Reconstruction for UAV Inspection Based on Tunneling Magnetoresistive Sensors and Inverse Models,» *IEEE Transactions on Power Delivery*, vol. 34, nº 3, pp. 819-827, 2019.

[21] F. Gao, S. Wang, Y. Wu, G. Zhao, B. Wang y J. Hu, «A novel inverse method for automatic UAV line patrolling with magnetic sensors,» de *2018 IEEE International Symposium on Electromagnetic Compatibility and 2018 IEEE Asia-Pacific Symposium on Electromagnetic Compatibility (EMC/APEMC)*, 2018.

[22] A. H. Khawaja y Q. Huang, «Estimating Sag and Wind-Induced Motion of Overhead Power Lines With Current and Magnetic-Flux Density Measurements,» *IEEE Transactions on Instrumentation and Measurement*, vol. 66, nº 5, pp. 897-909, 2017.

[23] Y. Wu, Y. Luo, G. Zhao, J. Hu, F. Gao y S. Wang, «A novel line position recognition method in transmission line patrolling with UAV using machine learning algorithms,» de *2018 IEEE International Symposium on Electromagnetic Compatibility and 2018 IEEE Asia-Pacific Symposium on Electromagnetic Compatibility (EMC/APEMC)*, 2018.

[24] H. Guan, X. Sun, Y. Su, T. Hu, H. Wang, H. Wang, C. Peng y Q. Guo, «UAV-lidar aids automatic intelligent powerline inspection,» *International Journal of Electrical Power & Energy Systems*, vol. 130, p. 106987, 2021.

[25] Wikipedia, la enciclopedia libre, «Alta tensión eléctrica,» [En línea]. Available: https://es.wikipedia.org/wiki/Alta_tensi%C3%B3n_el%C3%A9ctrica. [Último acceso: 28 Noviembre 2021].

[26] A. B. Rusiñol, «Materiales líneas alta tensión,» 25 Junio 2015. [En línea]. Available: https://electricidad-viatger.blogspot.com/2015/06/materiales-lineas-alta-tension.html. [Último acceso: 28 Noviembre 2021].

[27] E. Karabetsos, D. Filippopoulos, D. Koutounidis, C. Govari y N. Skamnakis, «ELF Electric and Magnetic fields measurements in Greece,» de *International Seminar on The Role of Dosimetry in High-Quality EMF Risk Assessment*, Ljubljana & Zagreb, 2006.

[28] B. Lohani, S. Chacko, S. Ghosh y S. Sasidharan, «Surveillance system based on Flash LiDAR,» Diciembre 2013.

[29] Q. Pentek, *Contribution à la génération de cartes 3D-couleur de milieux anturels à partir de données d'un système multicapteur pour drone,* Montpellier, 2020.

[30] X. Zhang, L. Zhou y H. Xie, «A Fast, Large-Stroke Electrothermal MEMS Mirror Based on Cu/W Bimoroph,» *micromachines*, 2015.

[31] M. Zohrabi, W. Y. Lim, R. H. Cormack, O. D. Supekar, V. M. Bright y J. T. Gopinath, «Lidar system with nonmechanical electrowetting-based wide-angle beam steering,» *Optics Express*, vol. 27, nº 4, pp. 4404-4415, 2019.

[32] Electronic Tutorials, [En línea]. Available: https://www.electronics-tutorials.ws/. [Último acceso: 29 Noviembre 2021].

[33] TDK, «Tech Library, Product review, TMR angle sensors,» [En línea]. Available: https://product.tdk.com/en/techlibrary/productoverview/tmr-angle-sensors.html. [Último acceso: 29 Noviembre 2021].

[34] PhysicsOpenLab, «Fluxgate magnetometer,» 6 Junio 2019. [En línea]. Available: https://physicsopenlab.org/2019/06/06/fluxgate-magnetometer/. [Último acceso: 29 Noviembre 2021].

[35] Open3D: A Modern Library for 3D Data Processing, «Open3D docs,» [En línea]. Available: http://www.open3d.org/docs/release/. [Último acceso: 23 Noviembre 2021].

[36] FernUniversität in Hagen: Faculty of Mathematics and Computer Science, «Research: Keypoint Detection in 3D Point Clouds,» [En línea]. Available: https://www.fernuni-hagen.de/mci/research/2_CompVis/keypoints. [Último acceso: 23 Noviembre 2021].

[37] Robotics Knowledgebase: The Wiki for Robot Builders, «Robotics Knowledgebase Wiki,» [En línea]. Available: https://roboticsknowledgebase.com/wiki/. [Último acceso: 23 Noviembre 2021].

[38] R. B. Rusu, Z. C. Marton, N. Blodow, M. Dolha y M. Beetz, «Towards 3D Point cloud based object maps for household environments,» *Robotics and Autonomous Systems*, vol. 56, nº 11, pp. 927-941, 2008.

[39] R. Bogdan Rusu y S. Cousins, «3D is here: Point Cloud Library (PCL)».

[40] Point Cloud Library (PCL), «Main site,» [En línea]. Available: https://pointclouds.org/. [Último acceso: 24 Noviembre 2021].

[41] P.-M. DiFrancesco, D. Bonneau y D. Hutchinson, «The Implications of M3C2 Projection Diameter on 3D Semi-Automated Rockfall Extraction from Sequential Terrestrial Laser Scanning Point Clouds,» *remote sensing*, 2020.

[42] K. G. Derpanis, «Overview of the RANSAC Algorithm,» 2010.

[43] Autor, «Este es el ejemplo de una cita,» *Tesis Doctoral,* vol. 2, nº 13, 2012.

[44] O. Autor, «Otra cita distinta,» *revista,* p. 12, 2001.

[45] L. Zhu y J. Hyyppä, «Fully-Automated Power Line Extraction from Airborne Laser Scanning Point Clouds in Forest Areas,» *Remote Sensing,* vol. 6, nº 11, pp. 11267-11282, 2014.

[46] A. Wehr y U. Lohr, «Airborne laser scanning—an introduction and overview,» *Isprs Journal of Photogrammetry and Remote Sensing,* vol. 54, nº 2, pp. 68-82, 1999.

[47] I. Said, N. Rahman, H. Hussain, A. Farag y T. Juhana, «Evaluation of magnetic field from different power transmission line configurations in Malaysia,» de *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No.04CH37513)*, 2004.

[48] Z. Wang, H. Liu, Y. Qian y T. Xu, «Real-time plane segmentation and obstacle detection of 3d point clouds for indoor scenes,» de *ECCV'12 Proceedings of the 12th international conference on Computer Vision - Volume 2*, 2012.

[49] Wikipedia, la enciclopedia libre, «k-d tree,» [En línea]. [Último acceso: 24 Noviembre 2021].

[50] Hipparchus: a mathematics Library, «Clustering,» [En línea]. Available: https://www.hipparchus.org/hipparchus-clustering/. [Último acceso: 24 Noviembre 2021].

ANEXO

En este anexo se recoge el código en C++ utilizado para el procesamiento de nubes de puntos. El código se organiza en dos programas independientes: el primero, **filtering.cpp,** implementa los bloques de filtrado y concatenación. El segundo, **modelling.cpp** implementa la segmentación y extracción de los modelos de línea.

filtering.cpp

```
//INCLUDES

#include <iostream>
#include <fstream>

#include <ros/ros.h>
#include <visualization_msgs/Marker.h>
#include <bag_data_extraction/Spd.h>

#include <tf/tf.h>

#include <rosbag/bag.h>
#include <rosbag/view.h>
#include <rosbag/query.h>

#include <pcl_ros/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>

#include <pcl/point_types.h>
#include <pcl/ModelCoefficients.h>

#include <pcl/filters/passthrough.h>
#include <pcl/filters/conditional_removal.h>
#include <pcl/filters/statistical_outlier_removal.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/segmentation/extract_clusters.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/octree/octree_search.h>


//TYPEDEFS
```

```
//definition of pcl point structures: it includes 2 extra fields, "curvature"
field containing reflectivity
typedef pcl::PointXYZINormal PointStructure;
typedef pcl::PointXYZ PointStructureSimp;
typedef pcl::PointCloud<PointStructure> PclPointCloud;



//FILENAMES

//directory where bag files are located
const std::string bagDirectory{"/home/antonio/bagfiles/"};

//pointcloud bag file
const std::string
bagFilenameCloud{bagDirectory+"livox_pcd_points_burguillos_exp_002_single_extrac
ted.bag"};

//pointcloud topic name
const std::string pointCloudTopic{"/livox_ros_points"};

//bag filenames for storing filtered clouds
const std::string bagFilenameFiltered{"filtered.bag"};
const std::string bagFilenameConcat{"concat.bag"};

//topics for identifying each filtering step in the bag file
//filtered.bag
const std::string originalTopic{"/livox_ros_points/original"};
const std::string passthroughTopic{"/livox_ros_points/passthrough"};
const std::string statremovalTopic{"/livox_ros_points/statremoval"};
const std::string voxelTopic{"/livox_ros_points/voxel"};
const std::string nearestKTopic{"/livox_ros_points/nearestK"};
const std::string filteredTopic{nearestKTopic};

//concat.bag
const std::string concatTopic{"/livox_ros_points/concat"};



//PARAMETERS

//Passthrough max reflectivity
const double MAXREFL = 0.1;
//Zero removal tolerance
const double ZEROTOL = 0.5;

//Number of neighbours to take into account in statistical removal
const int MEANK = 25;
```

```
//Standard deviation threshold multiplier for statistical removal
const double STDMULT = 1.0;

//Voxel downsampling leaf size
const float VLEAF = 0.5f;

//K nearest default points
const int KNEAREST = 100;
//Octree resolution for search algorithm
const double OCTRES = 1.0;

//Previous and latter N clouds to concatenate
const int NCONCAT = 1;



PclPointCloud::Ptr parseMessage(const rosbag::MessageInstance message)
{
  // Parse message instance to PointCloud2 ros message
  sensor_msgs::PointCloud2::ConstPtr
rosPointCloud{message.instantiate<sensor_msgs::PointCloud2>()};

  // Transform PointCloud2 ros message to PointCloud<pcl::PointXYZINormal>
  PclPointCloud::Ptr pclPointCloud{new PclPointCloud};
  pcl::fromROSMsg(*rosPointCloud,*pclPointCloud);

  return pclPointCloud;
}

void passThrough(PclPointCloud::Ptr &cloud)

{
  //PASSTHROUGH CODE

  // Create the filtering object
  pcl::PassThrough<PointStructure> pass;
  pass.setInputCloud (cloud);
  pass.setFilterFieldName ("curvature");
  pass.setFilterLimits (0.0, MAXREFL);
  //pass.setFilterLimitsNegative (true);

  PclPointCloud::Ptr filteredCloud{new PclPointCloud};
  pass.filter (*filteredCloud);
  cloud = filteredCloud;

}

void removeZeros(PclPointCloud::Ptr &cloud)
```

```
{
  //CONDITIONAL REMOVAL

  pcl::ConditionOr<PointStructure>::Ptr condition(new
pcl::ConditionOr<PointStructure>);

  condition->addComparison(pcl::FieldComparison<PointStructure>::ConstPtr(new
pcl::FieldComparison<PointStructure>("x", pcl::ComparisonOps::GT, ZEROTOL)));
  condition->addComparison(pcl::FieldComparison<PointStructure>::ConstPtr(new
pcl::FieldComparison<PointStructure>("x", pcl::ComparisonOps::LT, -ZEROTOL)));

  condition->addComparison(pcl::FieldComparison<PointStructure>::ConstPtr(new
pcl::FieldComparison<PointStructure>("y", pcl::ComparisonOps::GT, ZEROTOL)));
  condition->addComparison(pcl::FieldComparison<PointStructure>::ConstPtr(new
pcl::FieldComparison<PointStructure>("y", pcl::ComparisonOps::LT, -ZEROTOL)));

  condition->addComparison(pcl::FieldComparison<PointStructure>::ConstPtr(new
pcl::FieldComparison<PointStructure>("z", pcl::ComparisonOps::GT, ZEROTOL)));
  condition->addComparison(pcl::FieldComparison<PointStructure>::ConstPtr(new
pcl::FieldComparison<PointStructure>("z", pcl::ComparisonOps::LT, -ZEROTOL)));

  pcl::ConditionalRemoval<PointStructure> filter;
  filter.setCondition(condition);
  filter.setInputCloud(cloud);

  PclPointCloud::Ptr filteredCloud{new PclPointCloud};
  filter.filter(*filteredCloud);
  cloud = filteredCloud;

}

void statisticalRemoval(PclPointCloud::Ptr &cloud)
{

  //STATISTICAL REMOVAL CODE

  // Create the filtering object
  pcl::StatisticalOutlierRemoval<PointStructure> filter;
  filter.setInputCloud(cloud);
  filter.setMeanK(MEANK);
  filter.setStddevMulThresh(STDMULT);

  PclPointCloud::Ptr filteredCloud{new PclPointCloud};
  filter.filter(*filteredCloud);
  cloud = filteredCloud;
}

void nearestKSearch(PclPointCloud::Ptr &cloud, int k = KNEAREST)
{
```

```
  pcl::octree::OctreePointCloudSearch<PointStructure> octree{OCTRES};
  octree.setInputCloud(cloud);
  octree.addPointsFromInputCloud();

  PointStructure origin{};
  pcl::IndicesPtr indices{new std::vector<int>};
  std::vector<float> distances{};

  octree.nearestKSearch(origin, KNEAREST, *indices, distances);

  pcl::ExtractIndices<PointStructure> extract;
  extract.setInputCloud(cloud);
  extract.setIndices(indices);

  PclPointCloud::Ptr filteredCloud{new PclPointCloud};
  extract.filter(*filteredCloud);

  cloud = filteredCloud;
}

void voxelGrid(PclPointCloud::Ptr &cloud)

{
  //VOXEL GRID CODE

  // Create the filtering object
  pcl::VoxelGrid<PointStructure> voxel;
  voxel.setInputCloud (cloud);
  voxel.setLeafSize(VLEAF, VLEAF, VLEAF);
  voxel.setDownsampleAllData(false);
  //voxel.setMinimumPointsNumberPerVoxel(5);

  PclPointCloud::Ptr filteredCloud{new PclPointCloud};
  voxel.filter (*filteredCloud);
  cloud = filteredCloud;
}

int main (int argc, char *argv[])
{
  //BAG INPUT CODE
  rosbag::Bag bagCloud{bagFilenameCloud};
  rosbag::TopicQuery query{pointCloudTopic};
  rosbag::View viewCloud{bagCloud,query};


  //OPEN BAG FILE FOR SEGMENTED CLOUDS STORING

  std::remove(bagFilenameFiltered.c_str());
```

```
  rosbag::Bag bagFiltered;
  bagFiltered.open(bagFilenameFiltered, rosbag::bagmode::Write);


  //ITERATE THROUGH POINTCLOUDS
  //write segmented clouds to bag files for late visualization
  int nCloudsFiltered{0};

  for(const rosbag::MessageInstance message: viewCloud)
  {
    // Parse message instance to PointCloud2 ros message
    sensor_msgs::PointCloud2::ConstPtr
rosPointCloud{message.instantiate<sensor_msgs::PointCloud2>()};
    // Transform PointCloud2 ros message to PointCloud<pcl::PointXYZINormal>
    PclPointCloud::Ptr pclPointCloud{new PclPointCloud};
    pcl::fromROSMsg(*rosPointCloud,*pclPointCloud);

    //write original cloud
    bagFiltered.write(originalTopic, rosPointCloud->header.stamp,
pclPointCloud);


    //PASSTHROUGH FILTER

    passThrough(pclPointCloud);
    //REMOVE ZEROS
    removeZeros(pclPointCloud);
    //write filtered clouds
    bagFiltered.write(passthroughTopic, rosPointCloud->header.stamp,
pclPointCloud);


    //STATISTICAL REMOVAL DOWNSAMPLING

    statisticalRemoval(pclPointCloud);
    //write downsampled clouds
    bagFiltered.write(statremovalTopic, rosPointCloud->header.stamp,
pclPointCloud);


    //VOXEL GRID DOWNSAMPLING

    voxelGrid(pclPointCloud);
    //write voxel downsampling
    bagFiltered.write(voxelTopic, rosPointCloud->header.stamp, pclPointCloud);



    //NEAREST K SEARCH FILTER
```

```
    nearestKSearch(pclPointCloud);
    //write nearestK clouds
    bagFiltered.write(nearestKTopic, rosPointCloud->header.stamp,
pclPointCloud);


    ++nCloudsFiltered;
    std::cout << nCloudsFiltered << " Clouds filtered" << std::endl;

  }

  bagFiltered.close();


  //CLOUD CONCATENATION

  //OPEN FILTERED BAG
  bagFiltered.open(bagFilenameFiltered);
  rosbag::TopicQuery queryFiltered{filteredTopic};
  rosbag::View viewFiltered{bagFiltered,queryFiltered};

  int nCloudsConcat{0};

  //OPEN BAG FILE FOR CONCATENATED CLOUDS STORING

  std::remove(bagFilenameConcat.c_str());
  rosbag::Bag bagConcat;
  bagConcat.open(bagFilenameConcat, rosbag::bagmode::Write);

  //ITERATE THROUGH POINTCLOUDS
  //write concatenated clouds to bag files for late visualization

  //this is the vector that will store 2n+1 clouds to perform concatenation
  std::vector<PclPointCloud::Ptr> concatCloudsVector;
  //this is the vector containing corresponding timestamp
  std::vector<ros::Time> rosTimeVector;

  for(const rosbag::MessageInstance message: viewFiltered)
  {

    // Parse message instance to PointCloud2 ros message
    sensor_msgs::PointCloud2::ConstPtr
rosPointCloud{message.instantiate<sensor_msgs::PointCloud2>()};

    // Transform PointCloud2 ros message to PointCloud<pcl::PointXYZINormal>
    PclPointCloud::Ptr pclPointCloud{new PclPointCloud};
    pcl::fromROSMsg(*rosPointCloud,*pclPointCloud);
```

```
    // Add current cloud to vector
    concatCloudsVector.push_back(pclPointCloud);
    rosTimeVector.push_back(rosPointCloud->header.stamp);

    // Erase firs element of cloud vector if we exceeded max length of 2n+1
    if(concatCloudsVector.size()>2*NCONCAT+1)
    {
      concatCloudsVector.erase(concatCloudsVector.begin());
      rosTimeVector.erase(rosTimeVector.begin());
    }

    // Concatenate clouds only if we have 2n+1 clouds already stored. Write only
in that case.
    if(concatCloudsVector.size()==2*NCONCAT+1)
    {
      PclPointCloud::Ptr concatCloud{new PclPointCloud};
      // We copy-initialize the cloud so it keeps essential information,
including timestamp
      *concatCloud = *concatCloudsVector[NCONCAT];
      for(PclPointCloud::Ptr cloud: concatCloudsVector)
      {
        int flag{0};

        if (flag != NCONCAT)
         {
           //concatenation
           *concatCloud += *cloud;
           ++flag;
          }

      }

      //we would perform here other operations if needed

      //write
      bagConcat.write(concatTopic, rosTimeVector[NCONCAT], concatCloud);

      ++nCloudsConcat;
      std::cout << nCloudsConcat << " Clouds concatenated with NCONCAT=" <<
NCONCAT << std::endl;
    }

  }

  bagConcat.close();

  std::cout << "Final segmentation results:" << std::endl;
  std::cout << "Total clouds filtered successfully: " << nCloudsFiltered <<
std::endl;
```

```
  std::cout << "Final number of concatenated clouds: " << nCloudsConcat <<
std::endl;

  return (0);
}
```

modelling.cpp

```
#include <iostream>
#include <fstream>

#include <ros/ros.h>
#include <visualization_msgs/Marker.h>
#include <bag_data_extraction/Spd.h>

#include <tf/tf.h>

#include <rosbag/bag.h>
#include <rosbag/view.h>
#include <rosbag/query.h>

#include <pcl_ros/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>

#include <pcl/point_types.h>
#include <pcl/ModelCoefficients.h>

#include <pcl/filters/passthrough.h>
#include <pcl/filters/conditional_removal.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/segmentation/extract_clusters.h>
#include <pcl/filters/voxel_grid.h>

typedef pcl::PointXYZINormal PointStructure;
typedef pcl::PointCloud<PointStructure> PclPointCloud;

const std::string bagDirectory{"/home/antonio/bagfiles/"};

//segmented bag file containing filtered clouds
const std::string bagFilenameCloud{"concat.bag"};
//bag file containing magnetic field spectrum
const std::string
bagFilenameSPD{bagDirectory+"emf_burguillos_1_experimento002_single.bag"};

//input data topics
const std::string pointCloudTopic{"/livox_ros_points/concat"};
const std::string SPDTopic{"/aaronia/frequencies"};

//bag filenames for storing segmented clouds
```

```
const std::string bagFilenameSegmented{"segmented.bag"};

//topics for identifying each message in output bag
//segmented.bag
const std::vector<std::string> lineCloudsTopics{
"/livox_ros_points/line1_Cloud",
"/livox_ros_points/line2_Cloud",
"/livox_ros_points/line3_Cloud",
"/livox_ros_points/line4_Cloud",
"/livox_ros_points/line5_Cloud",
"/livox_ros_points/line6_Cloud"
};
const std::vector<std::string> lineModelsTopics{
"/livox_ros_points/line1_Model",
"/livox_ros_points/line2_Model",
"/livox_ros_points/line3_Model",
"/livox_ros_points/line4_Model",
"/livox_ros_points/line5_Model",
"/livox_ros_points/line6_Model"
};

//txt files for saving data
const std::string dataFilenameLines{"burguillos/dataLines.txt"};
const std::string dataFilenameSPD{"burguillos/dataPSD.txt"};



//PARAMETERS

//Cluster distance tolerance. It must be less than distance between lines ~1.5m
const double EUCLTOL = 1.1;
//Min cluster size
const int MINCLUSTSIZE = 25;

//RANSAC max iterations
const int RANSAC_MAXIT = 200;
//RANSAC distance threshold
double RANSAC_DIST{0};
//RANSAC min inliers, variable depending on cluster size
int RANSAC_MININLIERS{0};
//RANSAC point cloud size threshold for min inliers & distance threshold

std::vector<pcl::PointIndices> cluster(PclPointCloud::Ptr cloud)
{
  // create search object
  pcl::search::KdTree<PointStructure>::Ptr tree{new
pcl::search::KdTree<PointStructure>};
  tree->setInputCloud(cloud);
```

```
  // initialize indices vector
  std::vector<pcl::PointIndices> cluster_indices;

  // Euclidean Cluster Extraction parameters
  pcl::EuclideanClusterExtraction<PointStructure> ec;
  ec.setClusterTolerance(EUCLTOL);
  ec.setMinClusterSize(MINCLUSTSIZE);
  ec.setSearchMethod(tree);
  ec.setInputCloud(cloud);

  ec.extract(cluster_indices);

  return cluster_indices;
}

std::vector<PclPointCloud::Ptr> extract(PclPointCloud::Ptr cloud
,std::vector<pcl::PointIndices> clusterVector)
{
  //Extract cluster clouds from indices and place in cloud vector

  std::vector<PclPointCloud::Ptr> cloudVector{};

  pcl::ExtractIndices<PointStructure> extract;
  extract.setInputCloud(cloud);


  for(pcl::PointIndices cluster: clusterVector)
  {
    pcl::PointIndicesPtr clusterPtr{new pcl::PointIndices};
    *clusterPtr = cluster;
    extract.setIndices(clusterPtr);

    PclPointCloud::Ptr line{new PclPointCloud};

    extract.filter(*line);

    cloudVector.push_back(line);
  }

  return cloudVector;
}

void clusterFilter(std::vector<PclPointCloud::Ptr> clusterClouds,
std::vector<pcl::ModelCoefficients::Ptr> &coefficientsVector,
std::vector<PclPointCloud::Ptr> &lineCloudsVector)
{
  //Function that processes all clouds in cluster vector, any number of
ModelCoefficients from them
```

```
  //A cluster can provide from none to any number of lines models
  //Lines are extracted by lineModel function


  //declare lineModel function for obtaining coefficients
  pcl::ModelCoefficients::Ptr lineModel(PclPointCloud::Ptr, PclPointCloud::Ptr);

  for (PclPointCloud::Ptr clusterCloud: clusterClouds)
  {
    //infinite loop that will exit when:
    // 1) With the remaining points no model is feasible
    // 2) Cloud got fully emptied (no remaining points)
    while (true)
    {
      PclPointCloud::Ptr lineCloud{new PclPointCloud};
      pcl::ModelCoefficients::Ptr coefficients{lineModel(clusterCloud,
lineCloud)};

      //check if a line model could be obtained
      if (coefficients)
      {
        coefficientsVector.push_back(coefficients);
        lineCloudsVector.push_back(lineCloud);

        //we stop when there are no remaining points
        if (clusterCloud->points.size() == 0)
        {
          break;
        }
      }
      else break;
    }
  }
}

pcl::ModelCoefficients::Ptr lineModel(PclPointCloud::Ptr clusterCloud,
PclPointCloud::Ptr lineCloud)
{
    //Function that obtains line model coefficients from input cloud
    //Also returns cloud of inliers

    //RANSAC definition and parameters
    pcl::SACSegmentation<PointStructure> seg;
    seg.setOptimizeCoefficients(true);
    seg.setModelType(pcl::SACMODEL_LINE);
    seg.setMethodType(pcl::SAC_RANSAC);
    //seg.setDistanceThreshold(RANSAC_DIST);
    seg.setMaxIterations(RANSAC_MAXIT);
```

```
//seg.setProbability(0.3);
seg.setInputCloud(clusterCloud);

if (clusterCloud->points.size()<=20)
  {
    RANSAC_MININLIERS = 15;
    RANSAC_DIST = 0.50;
  }
else
  {
    RANSAC_MININLIERS = 20;
    RANSAC_DIST = 0.40;
  }

seg.setDistanceThreshold(RANSAC_DIST);


//Apply RANSAC segmentation
pcl::ModelCoefficients::Ptr coefficients{new pcl::ModelCoefficients};
pcl::PointIndices::Ptr inliers{new pcl::PointIndices};
seg.segment(*inliers, *coefficients);


//Check if the minimum number of inliers is accomplished
if (inliers->indices.size() <= RANSAC_MININLIERS)
{
  return nullptr;
}
else
{
  //In positive case, extract inliers and remove them from cloud

  pcl::ExtractIndices<PointStructure> extract;
  extract.setInputCloud(clusterCloud);
  extract.setIndices(inliers);
  {
    //First extract inliers
    PclPointCloud::Ptr filteredCloud{new PclPointCloud};
    extract.filter(*filteredCloud);
    *lineCloud = *filteredCloud;
  }

  {
    //Next extract remaining points cloud
    extract.setNegative(true);
    PclPointCloud::Ptr filteredCloud{new PclPointCloud};
    extract.filter(*filteredCloud);
    *clusterCloud = *filteredCloud;
  }
```

```
    }

    //A pointer is returned in either case: it could be the pointer to
coefficients, or a null pointer
    //A null pointer indicates a failed model extraction
    return coefficients;


}

void saveDataLines(sensor_msgs::PointCloud2::ConstPtr cloud,
std::vector<pcl::ModelCoefficients::Ptr> coefficientsVector)
{
  std::ofstream dataFile;
  dataFile.open(dataFilenameLines.c_str(), std::ios_base::app);


  for (pcl::ModelCoefficients::Ptr coefficients: coefficientsVector)
  {
    dataFile << cloud->header.stamp.toNSec();
    for (float coefficient: (*coefficients).values)
    {
      dataFile << " " << coefficient;
    }
    dataFile << "\n";
  }

  dataFile.close();

}

void saveDataSPD(bag_data_extraction::Spd::ConstPtr spd)
{
  std::ofstream dataFile;
  dataFile.open(dataFilenameSPD.c_str(), std::ios_base::app);

  dataFile << spd->header.stamp.toNSec();

  for (double freq: spd->Frequencies)
  {
    dataFile << " " << freq;
  }

  for (double power: spd->FrequenciesValues)
  {
    dataFile << " " << power;
  }
```

```
  dataFile << "\n";
  dataFile.close();

}

visualization_msgs::Marker getMarker(pcl::ModelCoefficients::Ptr coefficients,
int i)
{
  visualization_msgs::Marker marker;

  marker.header.frame_id = "livox_frame";
  //marker.header.stamp = ros::Time::now();
  marker.type = visualization_msgs::Marker::ARROW;
  marker.action = visualization_msgs::Marker::ADD;

  marker.pose.position.x = coefficients->values[0];
  marker.pose.position.y = coefficients->values[1];
  marker.pose.position.z = coefficients->values[2];

  tf::Vector3 axis_vector(-coefficients->values[3], -coefficients->values[4], -
coefficients->values[5]);
  tf::Vector3 up_vector(1.0, 0.0, 0.0);
  tf::Vector3 right_vector = axis_vector.cross(up_vector);//
  right_vector.normalized();//
  tf::Quaternion q(right_vector, -1.0*acos(axis_vector.dot(up_vector)));//
  q.normalize();//
  geometry_msgs::Quaternion line_orientation;
  tf::quaternionTFToMsg(q, line_orientation);

  marker.pose.orientation.x = line_orientation.x;
  marker.pose.orientation.y = line_orientation.y;
  marker.pose.orientation.z = line_orientation.z;
  marker.pose.orientation.w = line_orientation.w;

  marker.id = i;
  marker.scale.x = 20;
  marker.scale.y = 0.05;
  marker.scale.z = 0.05;
  // Set the color -- be sure to set alpha to something non-zero!
  marker.color.r = 1.0f;
  marker.color.g = 0.0f;
  marker.color.b = 0.0f;
  marker.color.a = 1.0f;
  //marker.lifetime = ros::Duration();

  return marker;
}
```

```
int main (int argc, char *argv[])
{

  //BAG READ
  rosbag::Bag bagCloud{bagFilenameCloud};
  rosbag::Bag bagSPD{bagFilenameSPD};

  rosbag::TopicQuery cloudQuery{pointCloudTopic};
  rosbag::TopicQuery SPDQuery{SPDTopic};

  rosbag::View viewCloud{bagCloud, cloudQuery};
  rosbag::View viewSPD{bagSPD, SPDQuery};

  // clear output data files
  {
  std::remove(dataFilenameLines.c_str());
  std::ofstream dataFile(dataFilenameLines.c_str());
  dataFile.close();
  }

  {
  std::remove(dataFilenameSPD.c_str());
  std::ofstream dataFile(dataFilenameSPD.c_str());
  dataFile.close();
  }

  //OPEN BAG FILE FOR SEGMENTED CLOUDS STORING

  std::remove(bagFilenameSegmented.c_str());
  rosbag::Bag bagSegmented;
  bagSegmented.open(bagFilenameSegmented, rosbag::bagmode::Write);

  //ITERATE THROUGH POINTCLOUDS

  int nCloudsSegmented{0};
  std::vector<int> nLinesModeled{};

  for(const rosbag::MessageInstance message: viewCloud)
  {
    // Parse message instance to PointCloud2 ros message
    sensor_msgs::PointCloud2::ConstPtr
rosPointCloud{message.instantiate<sensor_msgs::PointCloud2>()};
    // Transform PointCloud2 ros message to PointCloud<pcl::PointXYZINormal>
    PclPointCloud::Ptr pclPointCloud{new PclPointCloud};
    pcl::fromROSMsg(*rosPointCloud,*pclPointCloud);


    //CLUSTERING
```

```
    // Cluster input cloud, obtaining indices
    std::vector<pcl::PointIndices> clusterIndices{cluster(pclPointCloud)};

    // Get clouds from PointIndices vector
    std::vector<PclPointCloud::Ptr> clusterClouds{extract(pclPointCloud,
clusterIndices)};

    //only use the clusters if we get exactly 3 clusters
    //RANSAC is a more powerful segmentation tool if it considers the whole
cloud
    if (clusterClouds.size() != 3)
    {
      clusterClouds.clear();
      clusterClouds.push_back(pclPointCloud);
    }

    //LINE MODEL COEFFICIENTS
    std::vector<pcl::ModelCoefficients::Ptr> coefficientsVector{};
    std::vector<PclPointCloud::Ptr> lineCloudsVector{};

    clusterFilter(clusterClouds, coefficientsVector, lineCloudsVector);


    //WRITE DATA TO BAG FILE

    //write inliers clouds
    int j{0};
    for(PclPointCloud::Ptr cloud: lineCloudsVector)
    {
      bagSegmented.write(lineCloudsTopics[j], rosPointCloud->header.stamp,
cloud);
      j++;
    }

    //write line markers
    j=0;
    for(pcl::ModelCoefficients::Ptr coefficients: coefficientsVector)
    {
      bagSegmented.write(lineModelsTopics[j], rosPointCloud->header.stamp,
getMarker(coefficients,j));
      j++;
    }

    ++nCloudsSegmented;
    nLinesModeled.push_back(coefficientsVector.size());

    std::cout << "Number of lines modeled: " << coefficientsVector.size() << ".
Total clouds segmented: " << nCloudsSegmented << std::endl;
```

```
    //WRITE DATA TO TXT FILE
    saveDataLines(rosPointCloud, coefficientsVector);


  }

std::cout << "Number of clouds with 0 lines: " <<
std::count(nLinesModeled.begin(), nLinesModeled.end(), 0) << std::endl;
std::cout << "Number of clouds with 1 line: " <<
std::count(nLinesModeled.begin(), nLinesModeled.end(), 1) << std::endl;
std::cout << "Number of clouds with 2 lines: " <<
std::count(nLinesModeled.begin(), nLinesModeled.end(), 2) << std::endl;
std::cout << "Number of clouds with 3 lines: " <<
std::count(nLinesModeled.begin(), nLinesModeled.end(), 3) << std::endl;
std::cout << "Number of clouds with 4 lines: " <<
std::count(nLinesModeled.begin(), nLinesModeled.end(), 4) << std::endl;
std::cout << "Number of clouds with 5 lines: " <<
std::count(nLinesModeled.begin(), nLinesModeled.end(), 5) << std::endl;
std::cout << "Number of clouds with 6 lines: " <<
std::count(nLinesModeled.begin(), nLinesModeled.end(), 6) << std::endl;

//ITERATE THROUGH SPD

    for(const rosbag::MessageInstance message: viewSPD)
  {
    // Parse message instance to PointCloud2 ros message
    bag_data_extraction::Spd::ConstPtr
spd{message.instantiate<bag_data_extraction::Spd>()};

    //WRITE DATA TO BAG FILE
    bagSegmented.write(SPDTopic, spd->header.stamp, spd);

    //WRITE DATA TO TXT FILE
    saveDataSPD(spd);

  }

bagSegmented.close();

  return (0);
}
```