



POLITÉCNICA

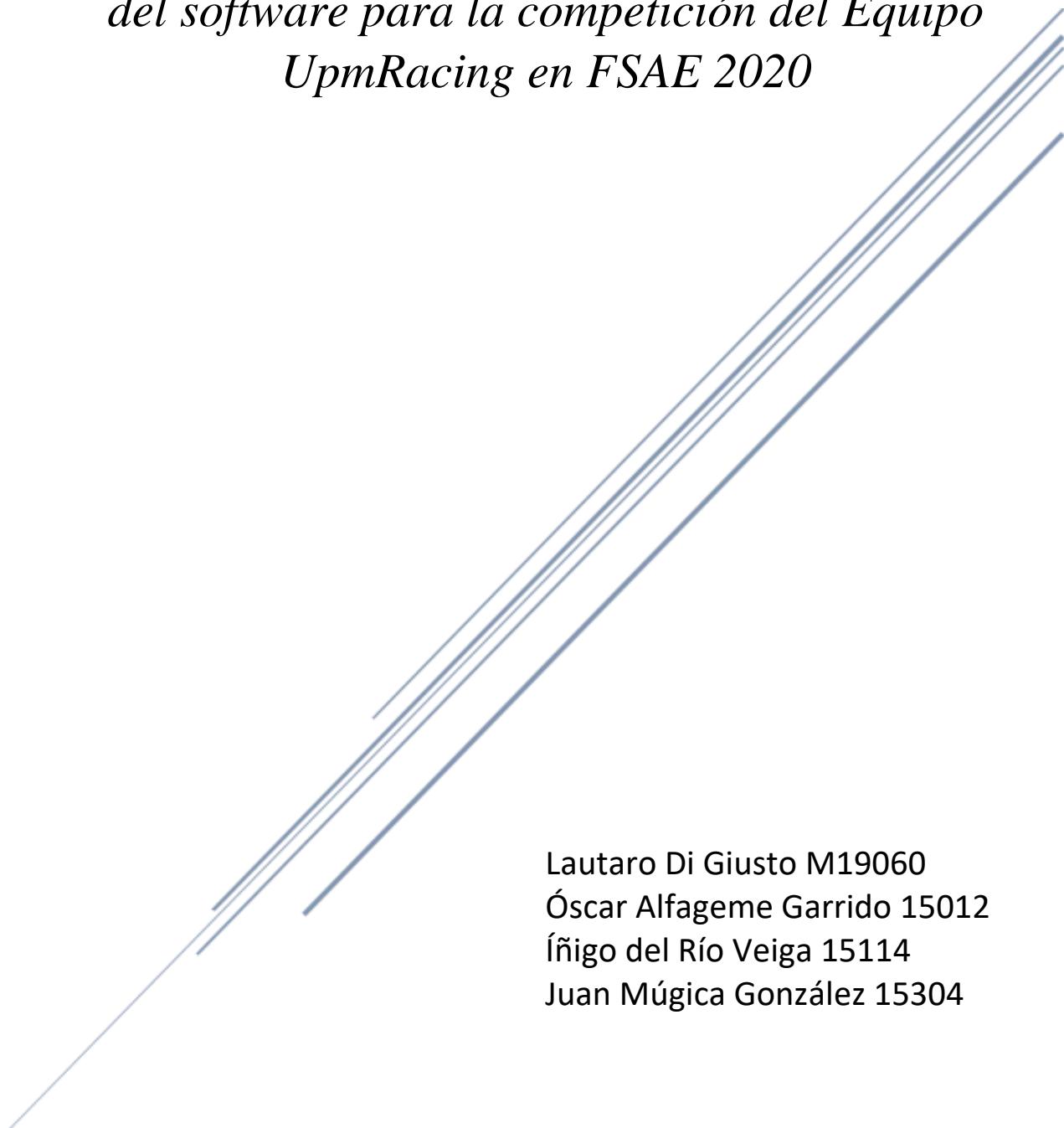
UNIVERSIDAD
POLITÉCNICA
DE MADRID



INDUSTRIALES
ETSII | UPM

Ingeniería FSAE División Driverless:

*detección con LiDAR y control en el desarrollo
del software para la competición del Equipo
UpmRacing en FSAE 2020*



Lautaro Di Giusto M19060
Óscar Alfageme Garrido 15012
Íñigo del Río Veiga 15114
Juan Múgica González 15304



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



INDUSTRIALES
ETSII | UPM



Índice

FIGURAS	5
1. INTRODUCCIÓN	<i>¡Error! Marcador no definido.</i>
1.2 DESCRIPCIÓN DEL ÁMBITO ORGANIZATIVO	7
1.3 SECTOR TECNOLÓGICO	8
1.4 CONTEXTO DEL PROYECTO	9
1.5 PERSPECTIVA DE ANÁLISIS.....	9
2. OBJETIVOS DEL PROYECTO.....	11
2.1 EL PROYECTO	13
3. DETECCIÓN.....	14
3.1 ESTADO DEL ARTE.....	15
3.1.1 PUNTO DE PARTIDA.....	15
3.1.2 SOLUCIONES TECNOLÓGICAS.....	17
3.2 DETECCIÓN CON LIDAR.....	18
3.2.1 El LIDAR.....	<i>¡Error! Marcador no definido.</i>
4.2.2 DEFINICIÓN DE LA ESTRATEGIA	19
1.1. GROUND FILTERING.....	21
1.1.1. RESUMEN	<i>¡Error! Marcador no definido.</i>
1.1.2. OBJETIVOS	21
1.1.3. DESARROLLO.....	23
1.1.4. RESULTADOS	28
1.1.5. CONCLUSIONES.....	34
1.2. CLUSTERING	35
1.2.1. RESUMEN	35
1.2.2. OBJETIVOS	35
1.2.3. DESARROLLO.....	35
1.3. RECONSTRUCTION	45
1.3.1. RESUMEN	45
1.3.2. OBJETIVOS	45
1.3.3. DESARROLLO.....	45
1.3.4. RESULTADOS	49
1.4. NON-CONE FILTERING.....	50
3.4.1 RESUMEN	50
3.4.2 OBJETIVOS	50
3.4.3 DESARROLLO.....	50
REPRESENTACIÓN GRÁFICA DE LOS PUNTOS.....	52
RESULTADOS.....	53



3.5 IMPLEMENTACIÓN EN ROS2.....	54
3.5.1 RESUMEN	54
3.5.2 OBJETIVOS	54
3.5.3 DESARROLLO.....	55
3.5.4 RESULTADOS.....	63
3.6 FUSION	70
3.6.1 RESUMEN	70
3.6.2 OBJETIVOS	70
3.6.3 DESARROLLO.....	70
3.6.4 RESULTADOS.....	78
4. CONTROL.....	88
4.1 RESUMEN	88
4.2 OBJETIVOS	88
4.3 DESARROLLO	90
4.3.1 CONTROL LONGITUDINAL	91
4.3.2 CONTROL LATERAL.....	92
4.4 RESULTADOS.....	95
4.4.1 SIMULACIÓN	96
4.4.2 PRUEBA EN VEHÍCULO REAL	104
4.4.3 MODIFICACIONES TRAS LA PRUEBA REAL	113
4.5 CONCLUSIONES.....	116
5. CONCLUSIONES DEL PROYECTO Y DIFICULTADES ENCONTRADAS.....	117
6. LÍNEAS FUTURAS	120
6.1 Líneas futuras enfocadas a la mejora de eficiencia y eficacia de lo ya realizado:.....	120
6.2 Líneas futuras enfocadas a la ampliación del proyecto:.....	120
7. REFERENCIAS.....	122



FIGURAS

<i>Figura 1: Driverless Project of the Academic Motorsports Club Zurich (AMZ) of ETH students.</i>	
<i>Fuente: [1].</i>	7
<i>Figura 2: Diagrama de bloques</i>	13
<i>Figura 3: Saltos de puntos correspondientes a dos conos. Fuente: [2].</i>	16
<i>Figura 4: LIDAR Velodyne vlp-16 [8]</i>	18
<i>Figura 5: Prueba LIDAR</i>	19
<i>Figura 6. Etapas de la fase de detección. Fuente: Elaboración propia</i>	19
<i>Figura 7. Pruebas en la captación de datos. Fuente: Elaboración Propia.</i>	20
<i>Figura 8: Frame completo visualizado en Python</i>	28
<i>Figura 9: Nube de puntos de la fase Ground Filter tras filtros de altura y distancia</i>	30
<i>Figura 10: nube de puntos de la fase Ground Filter tras aplicar el filtrado de suelo</i>	31
<i>Figura 11: Plano de suelo obtenido por la fase de filtrado de suelo con los parámetros por defecto</i>	31
<i>Figura 12: Plano de suelo obtenido para n_divisions=2</i>	33
<i>Figura 13: Plano de suelo obtenido para n_divisions=5</i>	33
<i>Figura 14: plano de suelo obtenido para n_iter = 1</i>	34
<i>Figura 15: Imagen explicativa de los tipos de puntos [13]</i>	38
<i>Figura 16: Dos puntos frontera, a y b, densamente conectados desde el punto núcleo c [13]</i>	39
<i>Figura 17: nube de puntos de la fase Ground Filtering tras aplicar el filtrado de suelo</i>	42
<i>Figura 18: nube de puntos de la fase Clustering tras aplicar la división por subnube de puntos</i>	42
<i>Figura 19: Clustering con MinPts = 20</i>	43
<i>Figura 20: Clustering con MinPts = 10</i>	44
<i>Figura 21. Representación en sistema diédrico del círculo.</i>	46
<i>Figura 22. Salida de clústeres sin suelo (arriba) y con suelo (abajo)</i>	49
<i>Figura 23. Representación de los Clusters antes de aplicar el Non-Cone Filter. Fuente: Elaboración propia.</i>	52
<i>Figura 24. Representación de los Clusters después de aplicar el Non-Cone Filter. Fuente: Elaboración propia.</i>	53
<i>Figura 25: Frame de Ground_pubsub a 5 hz</i>	64
<i>Figura 26: Frame de Ground_pubsub a 5 hz</i>	64
<i>Figura 27: Frame de Ground Filtering + Clustering Dataset</i>	65
<i>Figura 28: Frame de clusters obtenidos de Ground Filtering + Clustering</i>	66
<i>Figura 29: Frame de GroundFiltering + Clustering + Reconstruction Datasest</i>	67
<i>Figura 30: Frame de Ground Filtering + Clustering + Reconstruction</i>	67
<i>Figura 31: Frame de Ground Filtering + Clustering + Reconstruction + Non-cone filtering dataset</i>	68
<i>Figura 32: Frame de Ground Filtering + Clustering + Reconstruction + Non-cone filteringç</i>	68
<i>Figura 33: Ejemplo de marco propuesto</i>	72
<i>Figura 34: Resultado referencia de la primera fase de la calibración [7]</i>	75
<i>Figura 35: Resultado esperado tras eliminar discontinuidades [7]</i>	76
<i>Figura 36: Resultado esperado de la tercera etapa del algoritmo de fusión [7]</i>	77
<i>Figura 37: Resultado de aplicar la metodología de calibración en Lidar [7]</i>	77



<i>Figura 38: Marco de cartón para test de calibración</i>	79
<i>Figura 39: Toma de datos con Lidar algoritmo calibración</i>	80
<i>Figura 40: Marco para la calibración visualizado en Lidar</i>	80
<i>Figura 41: Salida de GroundPF en la primera fase de calibración</i>	81
<i>Figura 42: Salida de la primera fase del algoritmo de Calibración</i>	82
<i>Figura 43: Salida del filtro de profundidad de la segunda fase en calibración</i>	83
<i>Figura 44: Salida de la segunda fase de calibración</i>	84
<i>Figura 45: Salida de la tercera fase de calibración</i>	84
<i>Figura 46: Salida del algoritmo de calibración</i>	85
<i>Figura 47: marco de calibración con los centros hallados</i>	85
<i>Figura 48: Creación de la ruta a partir de los puntos medios de los conos</i>	93
<i>Figura 49: Esquema de los errores Heading y crosstrack</i>	94
<i>Figura 50: Simulación en trayectoria recta empezando en ella</i>	97
<i>Figura 51: Simulación en trayectoria curvilínea, comenzando dentro de ella</i>	98
<i>Figura 52: Error de posición del vehículo en trayectoria curvilínea</i>	98
<i>Figura 53: Trayectoria curvilínea comenzando a 2,5 m de ella</i>	99
<i>Figura 54: Error de posición en trayectoria curvilínea comenzando a 2,5 m de ella</i>	100
<i>Figura 55: Prueba fallida en trayectoria con un lazo</i>	101
<i>Figura 56: Prueba satisfactoria en trayectoria con un lazo</i>	102
<i>Figura 57: Trayectoria en el circuito del INSIA</i>	103
<i>Figura 58: Simulación de trayectoria real</i>	103
<i>Figura 59: Error de posición en simulación de trayectoria real</i>	104
<i>Figura 60: Archivos que se deben encontrar en la carpeta de scripts</i>	106
<i>Figura 61: Códigos para ejecutar el programa en ROS</i>	106
<i>Figura 62: Comando para ejecutar un rosbag</i>	108
<i>Figura 63: Trayectoria seguida en la prueba real</i>	109
<i>Figura 64: Parte de la trayectoria de la prueba real seguida correctamente</i>	112
<i>Figura 65: Error de posición del vehículo en la prueba real en la parte de la trayectoria seguida de forma correcta</i>	112
<i>Figura 66: Análisis del error de posición en la prueba real</i>	114



POLITÉCNICA

UNIVERSIDAD
POLÍTÉCNICA
DE MADRID



INDUSTRIALES
ETSII | UPM

1. INTRODUCCIÓN

Este proyecto se desarrolla con el principal objetivo de crear, dentro de los estándares y requisitos especificados por la sección de Inteligencia Artificial de la Fórmula SAE, un coche autónomo sin conductor con la finalidad de competir en verano de 2020.

Dentro de la competición de Formula Student Artificial Intelligence se concibe este proyecto para participar en la clase Dynamic Driving Task (DDT). Al participar en esta clase, se va a diseñar un sistema de conducción autónoma compuesto por hardware y software que se pueda implementar en un coche equipado y cedido por la Fórmula SAE para la competición.



Figura 1: Driverless Project of the Academic Motorsports Club Zurich (AMZ) of ETH students. Fuente: [1].

Este proyecto pretender cubrir las necesidades y resolver los problemas que se presentan a la hora de desarrollar un prototipo de un coche de conducción autónoma. De esta manera, también, se ha de tener en cuenta que este es un proyecto sin ánimo de lucro ya que es meramente educativo y cuyo objetivo primero es el aprendizaje de los estudiantes que conforman el equipo.

1.1 Descripción del ámbito organizativo

Se desarrolla este proyecto dentro del equipo de UPM Racing de la Universidad Politécnica de Madrid, el cual se encarga del diseño y construcción de un coche de



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



INDUSTRIALES
ETSII | UPM

competición para la Fórmula SAE. Este proyecto de Ingeniería toma parte dentro de la división de DRIVERLESS del equipo y pretende aportar en la búsqueda y aplicación de soluciones para la conducción autónoma.

Para lograr el objetivo de conducción autónoma se reparte el proyecto en distintos equipos, el equipo de control y el equipo de detección, habiendo tomado parte este proyecto de Ingeniería en ambos.

La división de detección se encarga de localizar los conos que delimitan la pista del circuito tanto por la derecha como por la izquierda. Para esto se utilizan dos tipos distintos de detección, un LIDAR de 16 capas y una cámara estéreo, siendo el objetivo de esta división la fusión de ambos y el traslado de esta información a la parte de control.

La división de control es la encargada de, a partir de los datos proporcionados por la detección, crear una trayectoria entre los conos y controlar el giro del volante, accionar el acelerador y el freno.

Dentro de los ámbitos de responsabilidad social el proyecto se enmarca en la categoría (P9) Desarrollo y difusión de tecnologías respetuosas con el medio ambiente.

El margen que se tiene para la toma de decisiones es amplio, ya que, aunque se debe respetar la normativa de la propia competición, se favorece la búsqueda de distintas soluciones para abordar los problemas que surgen de la puesta a punto del coche autónomo.

Este Ingeniería entra dentro de la fase de diseño dentro del ciclo de vida de todo el producto que se está desarrollando. De esta manera, se concibe la solución para la detección y posteriormente el control de la trayectoria y velocidad del coche.

1.2 Sector tecnológico

Se enmarca en el sector de automoción, específicamente en la parte de conducción autónoma, la cual en los últimos años ha avanzado considerablemente, llegando a ser una de las principales áreas de desarrollo de las empresas de automoción.



La relevancia del sector automoción es severa ya que gran parte de la movilidad, tanto de personas como de mercancías, a nivel mundial depende de este. Sin embargo, uno de los inconvenientes que arrastra este sector es su dependencia, tanto del petróleo, como del factor humano. Proyectos como el desarrollado en este Ingenia contribuyen a la evolución del sector automoción en la dirección de su independencia energética y humana.

Las distintas aportaciones que tiene la inteligencia artificial dentro de la automoción son las siguientes:

- Disminución del número de accidentes.
- Alivio de la congestión del tráfico.
- Reducción de costes energéticos.

Estas contribuciones no solamente están pensadas para los turismos, sino también, para replicar al sector del transporte público y el transporte de mercancías.

1.3 CONTEXTO DEL PROYECTO

Este proyecto se realiza durante el 15 aniversario del equipo de UPM Racing, el cual lleva desarrollando coches de competición desde 2004 y es el primer año que se compite en la categoría de DRIVERLESS.

El carácter de este Ingenia es educativo y se desarrolla como una actividad propuesta por la Universidad Politécnica de Madrid. El equipo de UPM Racing principalmente se financia a partir de patrocinadores como Bosch o Endesa y de la propia universidad.

El equipo se encuentra en el INSIA (Instituto Universitario de Investigación del Automóvil), situado en Campus Sur de la UPM, en el sureste de la ciudad de Madrid, España.

1.4 PERSPECTIVA DE ANÁLISIS

Se realiza el análisis de este Ingenia con perspectiva de “Proyecto”. De esta manera, cabe destacar que es un proyecto de investigación y apoyo al equipo UPM Racing,



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



pudiendo contribuir en ambos equipos, tanto en detección como control, acompañando al equipo en su objetivo de terminar con el software a tiempo para la competición.



2. OBJETIVOS DEL PROYECTO

Se distinguen los objetivos generales y específicos del proyecto, marcando estos varios hitos clave en el avance del proyecto.

OBJETIVOS INICIALES:

INICIALMENTE, LOS OBJETIVOS DEL PROYECTO FUERON LOS SIGUIENTES:

- BÚSQUEDA DE INFORMACIÓN
- OBTENER LOS DATOS DEL LiDAR
- FILTRAR LOS PUNTOS QUE NO PROPORCIONASEN INFORMACIÓN RELEVANTE SOBRE LA NUBE
- DETECTAR LOS CONOS
- DETECTAR LA POSICIÓN DE LOS CONOS CON EL LiDAR
- DETECTAR LOS CONOS CON LA CÁMARA
- FUSIONAR CÁMARA Y LiDAR PARA TENER INFORMACIÓN REDUNDANTE
- Lograr el seguimiento de una ruta por medio de un algoritmo de control
- Conectar código y vehículo para mandar órdenes a los actuadores como acelerador, freno o volante.
- Crear la ruta a seguir mediante los conos detectados por los sensores

Objetivo general:

- Este proyecto se desarrolla con el principal objetivo de crear, dentro de los estándares y requisitos especificados por la sección de Inteligencia Artificial de la Fórmula SAE, un coche autónomo sin conductor con la finalidad de competir en verano de 2020.

Objetivos específicos o hitos del proyecto:

- Objetivos de la parte de detección:
 - Comprender el funcionamiento del LiDAR y el tipo de datos se obtienen de este.
 - Comprender el funcionamiento del lenguaje de programación Python, así como la importación de librerías y datos.
 - Representar los datos obtenidos del LiDAR en Python.
 - Filtrar el suelo de la nube de puntos, incluyendo parámetros que permitan la regulación del algoritmo para su mejor ajuste.



- Filtrar alturas mayores a la altura de los conos, incluyendo parámetros que permitan la regulación del algoritmo para su mejor ajuste.
- Detectar los objetos que resulten tras el filtrado de suelo y altura mediante un algoritmo de clustering, incluyendo parámetros que permitan la regulación del algoritmo para su mejor ajuste y representándolos de colores distintos para su mejor interpretación gráfica.
- Reconstruir el suelo de los clústeres obtenidos para tener una mayor precisión de detección de los conos posteriormente, incluyendo parámetros que permitan la regulación del algoritmo para su mejor ajuste.
- Detectar los conos y no-conos de los clústeres con suelo reconstruido, incluyendo parámetros que permitan la regulación del algoritmo para su mejor ajuste.
- Objetivos de la parte de control:
 - Establecer un control longitudinal que permita una circulación a velocidad constante, la cual se alcance de forma suave. Incluir un sistema de frenada progresivo una vez se haya recorrido el circuito.
 - Lograr un control lateral del vehículo, que permita seguir una trayectoria definida con el menor error de posicionamiento posible y que sea estable, de forma que no alcance el objetivo con movimientos bruscos que puedan comprometer el comportamiento dinámico del vehículo.
 - Crear una trayectoria a seguir durante el propio movimiento del vehículo a partir de la posición de los conos detectados con la cámara. Estos conos deben leerse de tal forma que la trayectoria creada al final tenga sentido y no se desvíe del camino, lo que supondría un fallo en la prueba.
 - Leer datos del vehículo proporcionados por el GPS y la IMU, así como la posición de los conos otorgada por la cámara o el Lidar. Además, se debe conectar el programa con el vehículo de tal forma que sea capaz de enviar las órdenes de giro de volante y de pedal de freno o acelerador con una frecuencia suficiente para el correcto funcionamiento.
 - Sincronizar todas las señales enviadas y leídas para que trabajen a una misma frecuencia, de tal forma que diferentes órdenes no se asocien a medidas que corresponden a tiempos ya pasados.

2.1 El proyecto

A continuación, se presenta un esquema que contiene el diagrama de bloques en el que se descompone la división de Driverless. Se ha creado este diagrama para facilitar la visualización de todas las partes en las que ha participado este Ingenia.

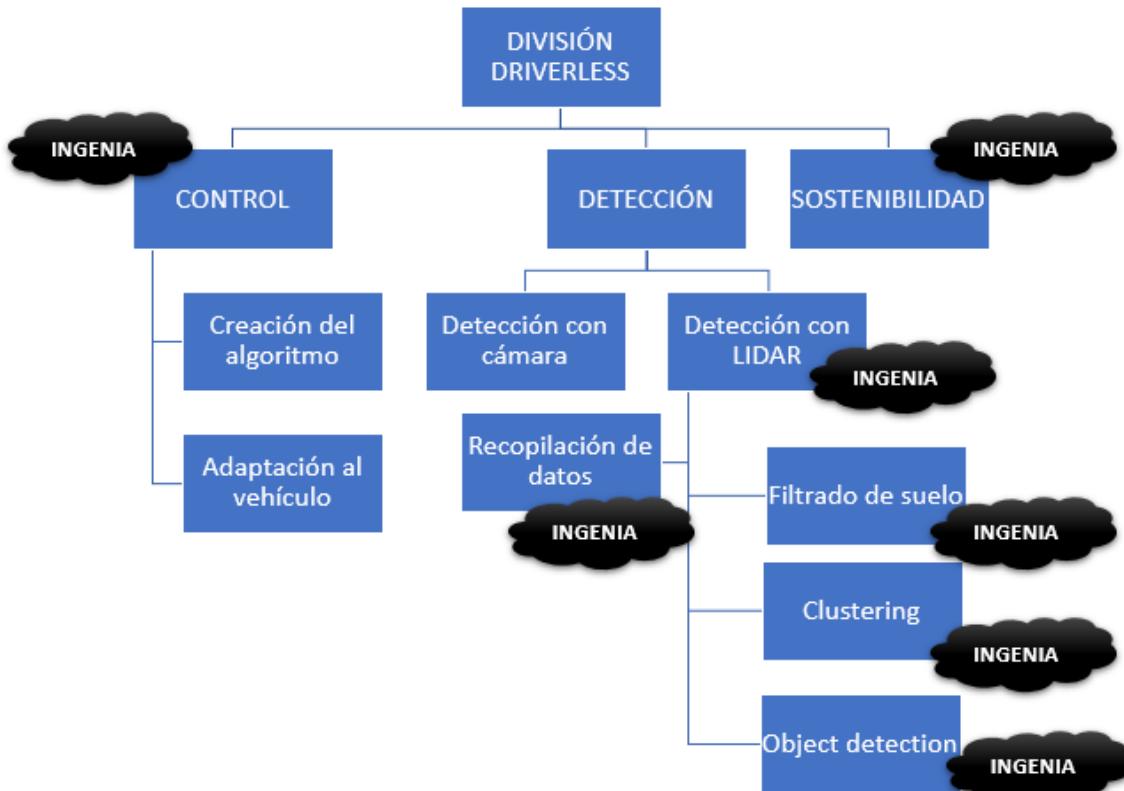


Figura 2: Diagrama de bloques

Como se puede observar en la figura superior, existen muchas ramas diferentes para este proyecto con sus objetivos y entregables y debe existir una sintonía y una relación estrecha entre ellos.

El equipo Helium, nombre que recibe este proyecto de ingenia, ha participado en el desarrollo completo de la detección con LIDAR, habiendo desarrollado desde la fase inicial hasta la final y habiendo conseguido fusionar los datos con la detección con cámara. De la misma manera, también se ha participado en el desarrollo del control de coche autónomo habiendo llegado a la fase de pruebas en pista del control del coche.



3. DETECCIÓN

A la hora de desarrollar un coche autónomo se debe incluir una solución a través de la cual se pueda captar el entorno del coche, en este caso se buscarán los conos que delimitan la pista que debe seguir. Todo esto con el objetivo de crear finalmente una trayectoria que el coche debe seguir para completar el circuito.

Según la normativa de la competición, el coche deberá realizar dos vueltas al circuito, una primera vuelta con el objetivo de detectar los conos y crear la trayectoria y una segunda vuelta para realizar ese camino con la mayor velocidad posible.

El concepto de la detección para un coche autónomo tiene como base fundamental la definición del objeto u objetos que debe detectar y filtrar el resto. Esta filosofía deja un objetivo muy claro para este proyecto que ha sido la detección de los conos. Estos conos están normalizados por la competición y son de color amarillo si se encuentra a la derecha del coche y azul si se encuentran a la izquierda.

En primer lugar, se ha diseñado una estrategia a través de la cual abordar este problema y llegar a soluciones prácticas y funcionales. Se han considerado las posibles limitaciones y se ha llegado a un plan de actuación. Este plan de actuación divide en una primera instancia la detección en dos, la detección con cámara y la detección con LIDAR.

La detección con cámara comprende la utilización de una cámara estéreo. Siempre con el objetivo de encontrar los conos, la estrategia comprende la aplicación de diferentes filtros sucesivos, de color, de forma y otros, para localizar en tiempo real dónde se encuentran estos conos. Sin embargo, en la detección con cámara no se llega a dar la posición de los conos con precisión ya que, aunque es una capacidad de las lentes estéreo, medir la profundidad, no se ha aplicado un algoritmo para buscar la profundidad a la que se encuentran los conos en la imagen ya que esta es una característica clave de la detección con LIDAR. No se va a entrar en detalle en la detección con cámara ya que este Ingenia no ha participado.

La detección con LIDAR se centra en la utilización de este dispositivo para obtener la posición exacta de los conos. El LIDAR de 16 capas que se ha utilizado se un dispositivo



que emite 16 laser 365 grados con inclinaciones de entre +150. El output de este dispositivo es un archivo CSV con las coordenadas de miles de puntos y su reflectividad. Al igual que en el caso de detección con cámara, al recibir una gran cantidad de datos se aplican sucesivos filtros para eliminar el exceso de puntos y obtener finalmente la posición de los conos.

Toda la fase de detección comprende una fase final muy importante que es la de la incorporación de todos los algoritmos desarrollados en ROS. ROS o Robot Operating System es la opción por la que se ha optado para poder obtener datos y ejecutar los algoritmos en tiempo real y transmitir esta información posteriormente a Control.

3.1 Estado del arte

Se ha realizado un estudio del estado del arte en el que se han analizado todas las opciones tecnológicamente posibles para abordar el desarrollo, se han estudiado los avances realizados por el equipo de Ingeniería en años previos y, finalmente, se ha establecido una estrategia. Aún con todo esto, a lo largo del desarrollo del proyecto se ha continuado con la investigación debido a la aparición de problemas y a la búsqueda constante de soluciones.

3.1.1 Punto de partida

Se tomaron como punto de partida dos direcciones.

Análisis del trabajo realizado por el equipo Ingeniería de ediciones previas.

Este análisis resultó en una compresión de los métodos y tecnologías utilizadas. Se pudo observar que, para la fase de detección se utilizaba un LIDAR de 16 capas. Igualmente, se había desarrollado un algoritmo para el tratamiento de las nubes de puntos que proporcionaba el LIDAR. Este algoritmo se centraba en el filtrado de los puntos asociando las discontinuidades de las líneas creadas por los haces del LIDAR a conos.

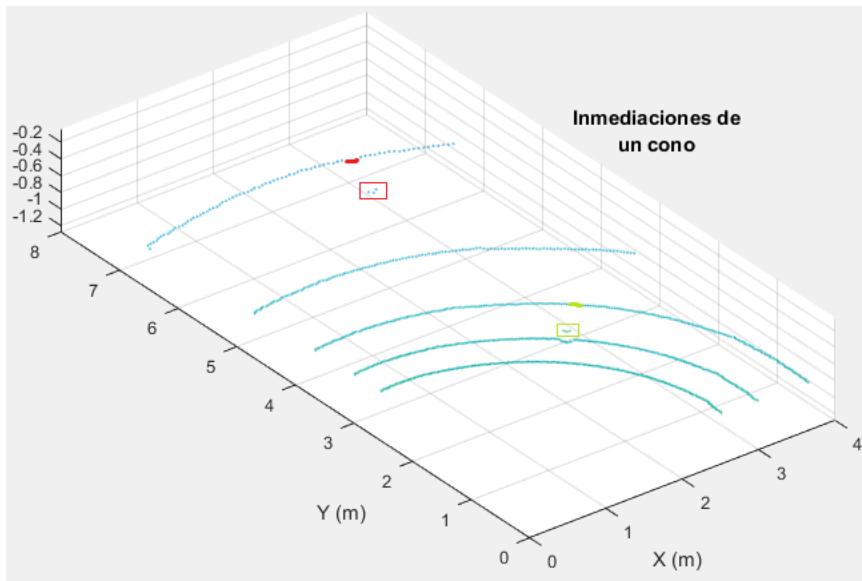


Figura 3: Saltos de puntos correspondientes a dos conos. Fuente: [2].

Tras una investigación sobre la practicidad de esta solución, se vio que no era posible asociar a conos las discontinuidades de las líneas de suelo creadas por los haces del LIDAR, ya que, cualquier elemento que se encuentre en la pista puede inducir a error y ocasionar un mal funcionamiento del sistema.

De la misma manera, se habían planteado los algoritmos en Matlab. Esta plataforma permite, como se hizo en este caso, la ejecución del programa para varios frames, pero, no se permite la ejecución en tiempo real que es el objetivo final del proyecto. A esto se suma el hecho de que Matlab no se puede implementar en ROS2 por lo que los algoritmos requerían de una traducción completa y no se dispuso de acceso a ellos.

Análisis de publicaciones científicas de equipos de éxito de la competición

En segundo lugar, se ha realizado un análisis de las publicaciones en revistas científicas hasta la fecha, tomando como referencia principal el documento de AMZ [1]. En este documento se recogen fielmente todas las etapas que componen el sistema de un coche autónomo de competición, la percepción del ambiente, la estimación de la posición, el SLAM y el control del coche.

Con el documento de Arnau Roche [3] se observa la realización de un proyecto que en gran medida tiene relación con este Ingeniería y comprende un desarrollo de toda la parte



de detección del coche autónomo, desde la captación de los datos con un LIDAR de 32 capas hasta la implementación en ROS.

Del documento [4] [5] se obtiene el método de cómo realizar el tratamiento de la nube de puntos a partir de una segmentación correcta de la misma para poder hacer el filtrado de suelo y clustering. La segmentación de los puntos permite un mejor comportamiento del algoritmo, con mayor precisión y rendimiento.

Siguiendo los pasos propuestos en estos documentos y otros como [6] [7] se ha establecido la estrategia para desarrollar el proyecto. En primer lugar, se desarrollan los algoritmos en Python y se trasladan a ROS2 (Robot Operating System), siendo este último el sistema operativo más utilizado actualmente para ejecutar algoritmos en tiempo real.

3.1.2 Soluciones tecnológicas

La primera decisión que hubo que tomar fue si la detección se realizaba con cámara, con LIDAR o con ambas.

- Se valora la utilización exclusiva de cámara por la simplificación que suponía utilizar solo un método de los dos. A esto se sumó el hecho de que la cámara es mucho más barata que el LIDAR, por lo que el equipo UPM Racing podría comprar una y tener independencia del INSIA. Igualmente, se valoró la posibilidad de incorporar una cámara de una sola lente o una cámara estéreo.
- Se valora la utilización exclusiva del LIDAR por el simple hecho de que puede obtener la posición exacta de los conos en contraposición a la cámara que solo puede estimar su posición. Un impedimento del LIDAR es la gran cantidad de datos que obtiene por lo que se necesita una mayor capacidad de computación. Además, el coste del LIDAR es muy alto y se depende de que el INSIA proporcione uno.
- Por último, se valora a utilización de ambos, lo que llevaría a un resultado mucho mejor, ya que se estimaría la posición de los conos con la cámara y se obtendría la posición exacta con el LIDAR, disminuyendo la necesidad de computación. Debido a los buenos resultados que promete, al gran valor educativo y pese a los inconvenientes económicos, esta es la solución que finalmente se elige.

3.2 Detección con LiDAR

3.2.1 El LiDAR

En primer lugar, definir qué es el LIDAR, porqué se ha utilizado en este proyecto y porque se utiliza en gran parte de coches autónomos.



Figura 4: LiDAR Velodyne vlp-16 [8]

El LIDAR (Laser Imaging Detection and Ranging) que se ha utilizado para la realización de este proyecto es el LIDAR Velodyne vlp-16. Este LIDAR adquiere datos 360° emitiendo simultáneamente 16 haces de luz pulsada que se distribuyen homogéneamente en vertical con una orientación entre los +15° y los -15°, mientras gira sobre su centro a una velocidad de 600 rpm. A esta velocidad tiene una resolución horizontal de 0.2° y una resolución vertical de 1.875°.

De esta manera, el LIDAR genera haces con una longitud de onda de entre 896 y 910 nm y los pulsos que emite son de 6 ns de duración. El resultado es la generación de una nube de puntos a partir del tiempo que tarda el haz en regresar y su reflectividad.

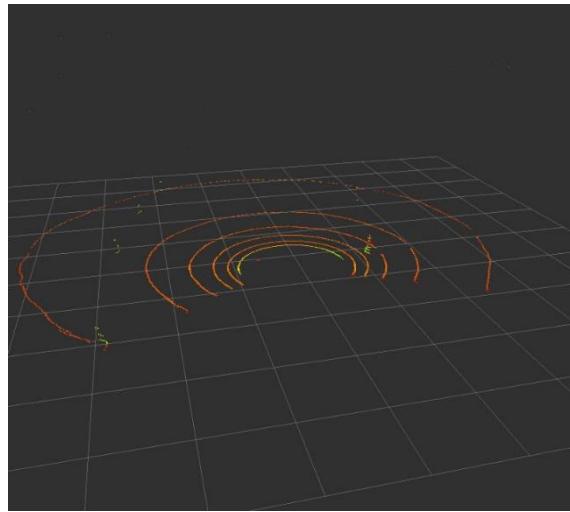


Figura 5: Prueba LIDAR

Esta imagen, ha sido obtenida de las pruebas en ROS que se han realizado y los datos especificados en el apartado 3.5.3.6 DATASETS REALES CON ROS2 del documento. Se pueden observar las distintas líneas creadas por los haces de LIDAR según el ángulo.

3.2.2 Definición de la estrategia

En lo que se refiere a la detección, tras haber realizado un primer reconocimiento de los métodos utilizados en publicación científicas. Se establece la estrategia para desarrollar la fase de detección según las siguientes etapas.

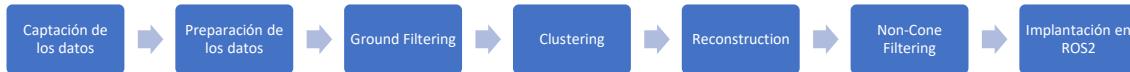


Figura 6. Etapas de la fase de detección. Fuente: Elaboración propia

Las fases en la que se divide la detección son:

1. **Captación de los datos:** La fase de captación de datos se centra en la utilización del dispositivo de detección que es el LIDAR. A la hora de la toma de datos para el desarrollo de los algoritmos se realizó un experimento como el de la imagen en el que se fueron tomando datos para distancias entre LIDAR y cono cada vez mayores. Cada toma de datos consiste guarda unos segundos de frames tomados por el LIDAR para, posteriormente, trabajar con el frame que más interesa.



Figura 7. Pruebas en la captación de datos. Fuente: Elaboración Propia.

2. **Preparación de los datos:** Esta fase tiene como objetivo preparar los datos que salen del LIDAR para poder aplicar posteriormente los algoritmos deseados. En este caso, la aplicación de Velodyne permite la opción de exportar a un archivo csv todos los puntos obtenidos de un frame de LIDAR. Esta es la opción que se utiliza para la fase de desarrollo.
En la fase de ejecución, se conecta un ordenador al LIDAR a través de un puerto Ethernet. Un driver denominado *3.5.3.5.1.1 Velodyne_driver* en ROS2 interpreta la información y se ejecuta los algoritmos posteriores. La explicación de la implementación en ROS2 se puede encontrar en el apartado *3.5.3.4 IMPLANTACIÓN DE LOS ALGORITMOS DE DETECCIÓN EN ROS2*.
3. **Ground Filtering:** Esta fase tiene como objetivo el filtrado de suelo de todos los frames que llegan del LIDAR para dejar únicamente los objetos que puedan ser los conos que delimiten la pista.
4. **Clustering:** Esta fase busca la agrupación de puntos que se encuentren cercanos para crear objetos y posteriormente decidir sobre ellos. Se ha utilizado un algoritmo de aprendizaje supervisado y se han diferenciado con colores los diferentes clusters para su mejor visualización.
5. **Reconstruction:** Esta fase pretende reconstruir, con puntos de suelo previamente filtrados, los clusters creados en la fase anterior para poder completar los puntos que los formaban en primera instancia.



6. **Non-Cone Filtering:** En esta fase se llega a diferenciar si un cluster es un cono o no lo es. De esta manera, se llegan a filtrar todos los puntos no pertenecientes a conos y tenemos como resultado final los conos con sus posiciones.
7. **Implantación en ROS2:** La última fase se centra en implementar todos los algoritmos desarrollados en Python en ROS2. ROS2 es el sistema operativo que va a permitir la ejecución de estos algoritmos en tiempo real.

3.3 Ground filtering

3.3.1 Resumen

El objetivo del Filtrado de Suelo es constituir la primera etapa de filtrado de nubes de puntos. Partiendo de la totalidad de los puntos del Lidar (o en su defecto tras un filtro previo), se eliminarán todos los puntos constituyentes del terreno, restando únicamente los objetos sobre la pista.

Además, en esta etapa se realizará un filtrado por distancia, altura y azimuth de los puntos, de esta forma se eliminarán todos los puntos que no aporten la competición, incrementando de esta forma el tiempo de ejecución de los algoritmos.

Para el filtrado de suelo, existen diferentes prácticas disponibles. Además, existen librerías y software online que facilitan el proceso, llegándolo a finalizar en algunos casos. Una de estas librerías es PointCloud (PCL) [9]. Esta librería no ha sido testeada por lo que no se han comprobado sus resultados.

En esta sección se detalla cómo se realizó el primer Filtrado de Suelo con Lidar, partiendo de un algoritmo desarrollado en la Universidad de Delphi [10]. Una explicación de los objetivos del código, seguida por su desarrollo detallado, será concluida con resultados a lo largo de la sección.

3.3.2 Objetivos

1. Filtro de altura que elimine todos los puntos situados a partir de una altura umbral.
2. Filtro de distancia que elimine todos los puntos situados a mayor distancia de otro umbral.
3. Segmentación precisa de la nube de puntos 3D proporcionada por un sensor Lidar en un número n de divisiones según el azimuth de cada punto.
4. Eliminar todo punto 3D constituyente del terreno.
5. Alta frecuencia de ejecución del algoritmo. Siendo la velocidad de giro estándar del VLP16 de 600rpm (10Hz) se busca aproximarse lo mayor posible a esta última.



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



6. Mínima complejidad de computación, para dedicar los recursos de hardware a otras funciones del sistema. Este objetivo va muy ligado al objetivo 5.



3.3.3 Desarrollo

EXPLICACIÓN DEL ALGORITMO

Existen múltiples técnicas para realizar un Filtrado de Suelo. Dichas técnicas conllevan complejidades, precisiones y tiempos de ejecución diferentes.

Hay que tener en cuenta que el suelo está compuesto por infinitos planos de diferentes pendientes, por lo que una correcta detección no resulta de limitar los puntos inferiores a una altura h determinada.

Tal como mencionan los autores de la segmentación por parte de la Universidad de Delphi [10], la mayoría de los puntos devueltos por el Lidar corresponden a impactos del haz láser en el suelo. Es por ello por lo que un filtrado correcto de este conllevará una disminución significativa de la nube de puntos, con el correspondiente incremento de la velocidad de procesamiento del resto de algoritmos que continúan.

La identificación de los puntos pertenecientes al suelo cuenta con dos importantes ventajas:

- a. Dichos puntos pertenecen a planos, objetos geométricos fácilmente parametrizados matemáticamente.
- b. Los puntos cuya cota de altura sea menor, pertenecerán con mayor probabilidad al suelo.

Como mencionábamos anteriormente, un solo plano no suele ser suficiente para representar la superficie de suelo. Además, el ruido en las medidas del Lidar incrementa con la distancia, por lo que se necesita un método más sofisticado.

El modelo propuesto segmenta el rango total de 360° en un número finito de segmentos, de forma que para cada segmento se calculará un plano diferente. Dichos segmentos podrían ser divididos a su vez en subsegmentos en función de la distancia r al Lidar, lo que no parece necesario para una primera aproximación.

A partir de este punto, el proceso contará con un número finito de iteraciones, calculando planos sucesivos para cada segmento, saliendo del algoritmo el último plano de suelo calculado. La primera iteración o cálculo del primer plano parte de extraer un número finito de puntos cuyas cotas de altura sean menores, con los que se estimará el primer plano.

Tras esto, será evaluada la distancia de cada punto del segmento a la proyección ortogonal de este sobre el plano. Dicha distancia es comparada a un umbral Th_{dist} , decidiendo si dicho punto pertenece a la superficie solar o no. Los puntos considerados pertenecientes compondrán las semillas del nuevo plano generado, iterando dicho proceso



un número N_{iter} de veces. Finalmente, los planos generados para cada uno de los segmentos constituirán la superficie total de suelo.

¿Cómo estimar las semillas para el plano inicial?

A partir de un número finito de puntos cuya cota de altura sea menor, se generará un nuevo punto N_{LPR} , siendo este una media de los anteriores. Esto permitirá reducir el ruido de las medidas. A partir de este punto, el algoritmo se quedará con los puntos que estén a menor distancia del plano que el umbral Th_{seeds} .

¿Cómo estimar cada plano a partir de la nube de puntos?

Se partirá de un modelo lineal:

$$ax + by + cz + d = 0$$

$$n^T x = -d$$

$$n = [a \ b \ c]^T$$

$$x = [x \ y \ z]^T$$

Para el cálculo de n , se realiza una descomposición en valores singulares (SVD) de la nube de puntos considerada perteneciente al plano en dicha iteración. Esta proporciona las tres direcciones principales de dispersión. Al ser una superficie plana, el vector n , perpendicular al plano, indica cuál es la dirección de menor varianza y corresponde al menor valor singular.

Una vez calculado n , d se halla directamente sustituyendo s en la ecuación, siendo s la media de todos los puntos de la nube considerada suelo en dicha iteración.

ANÁLISIS DEL CÓDIGO

El código, ha sido enteramente realizado en Python. Este se compone de dos códigos fundamentalmente, uno siendo el algoritmo de filtrado de suelo propiamente, *DelphiMejorado*, y el otro compuesto por todas las otras funciones de apoyo para testear el algoritmo y visualizar su eficacia en el ordenador, *GroundPF*.

DELPHI MEJORADO

A continuación, se presenta una explicación de cada una de las funciones del código:

1. `__init__`: al tratarse de un objeto, esta función inicializa los parámetros o atributos de dicho objeto. Contiene valores predeterminados, los cuales se han



comprobado que funcionan bien en términos genéricos, pudiendo ser modificados. La modificación de los parámetros se detallará a lo largo de esta sección.

2. **ExtractInitialSeeds:** como su nombre indica, esta función extrae los puntos semilla para el cálculo del primer plano, correspondiente a la primera iteración. A partir de los puntos ordenados en cota z creciente, toma los N_{lpr} puntos de menor cota z y tras esto realiza la media de dichos puntos: lpr . Por último, devuelve aquellos puntos cuya cota z no difiera más de Th_{seeds} de la cota de lpr .
3. **main:** esta función, a partir de las semillas proporcionadas por **ExtractInitialSeeds** y la nube de puntos **point**, devolverá dos arrays: **pg** conteniendo los puntos pertenecientes al suelo y **png** el resto de los puntos pertenecientes a **point**.

Para ello, en un bucle de N_{iter} iteraciones, realizará el siguiente proceso:

- a. SVD de los puntos considerados **pg** en dicha iteración. Cálculo del vector normal a dicho plano.
- b. Redefinir **pg** como el array de puntos cuya distancia ortogonal al plano de la SVD sea menor que Th_{dist} .

GROUNDPF

A continuación, se presenta una explicación de cada una de las funciones del código:

1. **__init__:** al igual que para **DelphiMejorado**, esta función inicializa los atributos del objeto, con valores predefinidos que deberían funcionar correctamente en condiciones de uso genéricas.
2. **preProcesamientoROS:** función para ordenar los puntos recibidos desde ROS en un array de puntos. Los puntos son leídos desde ROS en forma de vector continuo de puntos (los puntos van seguidos uno detrás de otro con sus respectivas coordenadas de la forma: [x1,y1,z1,x2,y2,z2,etc.]
3. **segmentation:** esta función realiza un filtrado por distancia al Lidar (según D_{max}) y por altura (según Z_{max}) y un posterior segmentado. El objetivo del filtrado es eliminar todo punto que esté a una distancia del Lidar que no merezca la pena considerar o a una altura que inevitablemente puedan formar parte del suelo. Para la referencia de la altura se toma el punto con cota z más pequeña de toda la nube de puntos.

En cuanto a la segmentación, lo primero que realiza la función es calcular la distancia r y el ángulo azimuth. Ambos valores podrían ser transmitidos como parámetros, modificando ligeramente las líneas de código. Posteriormente, según el número de segmentos definido por $N_{divisions}$, situará a los puntos de la nube pertenecientes a estos en arrays, los cuales serán devueltos como salida en la lista **sectorized**.



El proceso para situar los puntos por sector se hará de forma que el bucle encargado de recorrer los puntos y situarlos en segmentos sea cada vez de menor tamaño para aligerar el tiempo de procesado.

4. **Display3d**: esta función proyecta dos imágenes:
 - a) Contiene todos los puntos, tanto *ground* como *nonGround*, tras el filtrado por altura y distancia.
 - b) Contiene únicamente los puntos *nonGround*, para comprobar la eficacia del algoritmo.
5. **main**: esta función es capaz de realizar el filtrado de suelo completo. Está programada para trabajar a partir de un fichero *.csv* para los tests. Deberá ser modificada para que tome los datos de ROS. Aquellos segmentos cuyo número de puntos sea inferior a 10 no serán considerados (para poder realizar una correcta SVD), eliminando dichos sectores. Al tomar el Lidar un número muy elevado de puntos, es muy baja la probabilidad de que dichos puntos pertenezcan a un cono.

PARÁMETROS PARA SELECCIONAR

En el código **GroundPF** el cuál es usado para realizar las funciones conjuntas de filtrado de altura, distancia y suelo, existen varios parámetros que se pueden modificar, y que afectan directamente al rendimiento y precisión del algoritmo.

Estos parámetros son:

- **Umbral de altura [th_z]**: este parámetro determinará la altura a partir de la cual los puntos serán eliminados. Esta altura se calcula sumando el valor de *th_z* a la altura mínima presente en la nube de puntos detectada por el Lidar.
- **Número de iteraciones [n_iter]**: como se ha comentado anteriormente, el algoritmo funciona de forma recursiva calculando el plano n veces para cada segmento, resultando como plano de suelo el plano final. En función de la orografía, podremos modificar este parámetro. Un valor mayor aportará mayor precisión, pero ralentizará el rendimiento.
- **Número de puntos semilla [n_lpr]**: este parámetro determina el número de puntos que compondrán las semillas el primer valor medio N_{lpr} , a partir del cuál se tomarán las semillas para el primer plano.
- **Umbral de las semillas [th_seeds]**: este umbral determinará la distancia máxima a la que podrán encontrarse los puntos del primer N_{lpr} (definido en el parámetro precedente) para componer las semillas objeto de la primera SVD y por tanto del cálculo del primer plano.



- **Umbral de distancia al plano** [th_dist]: este umbral tiene la misma función que *th_seeds*, utilizándose para todos los N_{lpr} sucesivos.
- **Número de divisiones** [n_divisions]: define el número de divisiones en las que se segmentará por ángulo azimuth. Definirá por tanto también el número de planos que conjuntamente compondrá el plano final. Un valor más alto otorgará mayor precisión pero requerirá de mayor tiempo de computación.
- **Azimuth máximo** [a_max]: este parámetro determina, a partir de la referencia del Lidar, cuál es el azimuth máximo para ejecutar un filtro por ángulo, en el caso en que fuese necesario. Podrá ser utilizado, por ejemplo, para que el Lidar tome únicamente los puntos que se encuentran delante del dispositivo. Si se buscase definir un rango entre dos ángulos, habría que añadir al algoritmo un nuevo parámetro, denominado por ejemplo *a_min*.
- **Umbral de distancia** [d_max]: define la distancia máxima al Lidar a partir de la cual los puntos serán eliminados.

3.3.4 RESULTADOS

El algoritmo fue testeado en un frame proveniente de una prueba realizada en el Insia, en la que se situaron cuatro conos sobre la pista, y se registró un recorrido con el Lidar a través de Veloview, registrando así archivos de tipo pcap, que pueden ser transformados a sus frames correspondientes en formato csv.

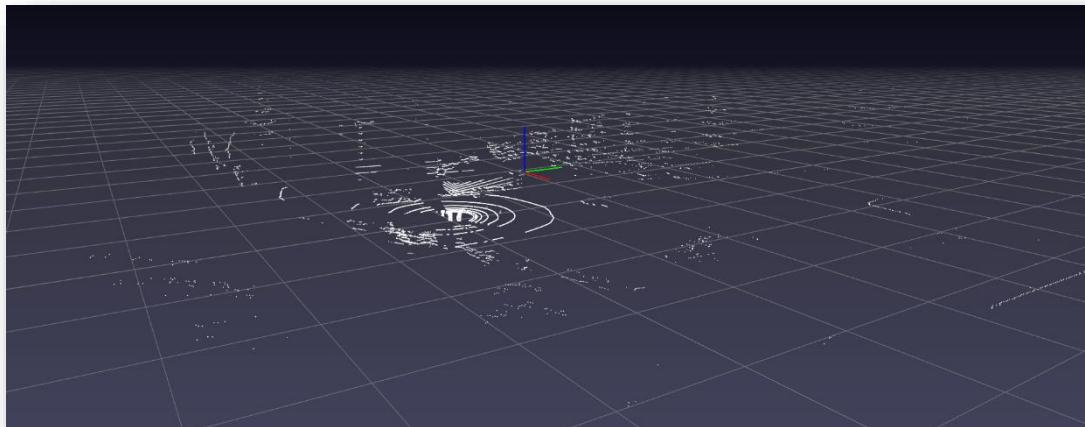


Figura 8: Frame completo visualizado en Python

Con ayuda de la librería pptk [11], se ha podido visualizar en Python la nube de puntos contenida en el archivo Csv, como vemos en Figura 8.

Esta nube de puntos está compuesta por 13049 puntos, los cuales se irán filtrando hasta obtener únicamente el cono de la competición sobre la pista.

Para ejecutar el algoritmo, se testearon diversas combinaciones de parámetros. Para no sobrecargar el documento, se presentarán a continuación los parámetros escogidos para su ejecución, siendo estos valores que otorgaban resultados óptimos, y se presentarán algunas variaciones y sus consecuencias.

Los parámetros establecidos en el algoritmo para la prueba que se presenta a continuación son, por tanto:

- Th_z = 2
- N_iter = 2
- N_lpr = 20
- Th_seeds = 0.05
- Th_dist = 0.05
- N_divisions = 12
- A_max = (2*np.pi)



- $D_{\max} = 7$

Estos valores fueron establecidos como parámetros por defecto del algoritmo, si en su utilización en un momento concreto no se define su valor, al otorgar resultados precisos.

Al ejecutar el algoritmo, se realizan filtros por distancia y por altura:

- Filtro por distancia: se filtra toda la información a una distancia del Lidar mayor de 7m. Se consideró esta distancia al aportar un margen suficiente al algoritmo de control para posibles rectificaciones, pudiendo ser cambiada cada vez que se ejecute el algoritmo.
- Filtro por altura: al buscar que únicamente resulten conos sobre la pista, se aplica un filtro de altura, recogiendo la altura mínima de la nube completa de puntos y filtrando todo aquello por encima de 1m de esta, donde quedarán recogidos todos los conos de la competición (ya que tienen una altura menor de 0.35m), además de otra información no relevante. De igual forma, este parámetro puede ser modificado cada vez que se ejecute el algoritmo, pudiendo adaptarlo a diferentes condiciones de la pista.

Además de estos filtros, se establece:

- Número de divisiones = 12: esto provoca que el espacio alrededor del Lidar será segmentado en sectores de 30° de amplitud, de los cuales partirán 12 planos que compondrán el plano final.
- Número de iteraciones = 2: se comprobó que en el dataset en el Insia 1 iteración prácticamente bastaba para obtener un plano representativo del suelo. Sin embargo, se estableció el valor 2, para definir mayor seguridad de cara a la utilización del algoritmo en posteriores ocasiones.

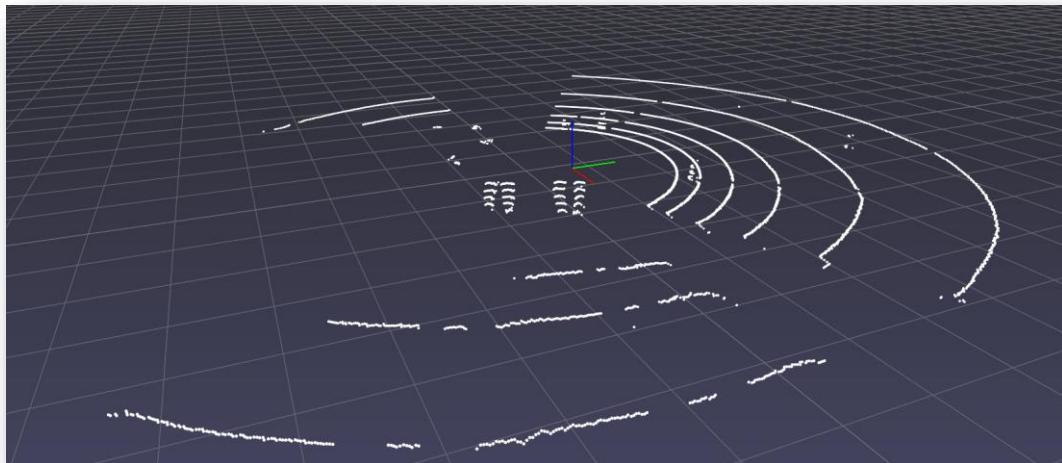


Figura 9: Nube de puntos de la fase Ground Filter tras filtros de altura y distancia

Como observamos en la Figura 9, tras aplicar los filtros de altura y distancia, resultan 5044 puntos, un 39% de los iniciales, habiendo eliminado gran cantidad de información irrelevante y permitiendo así una mayor velocidad de ejecución del algoritmo.

Observamos también en Figura 9 que, en tres sectores, dos a los lados del Lidar y un tercero situado detrás no hay ningún punto a menos de 7 metros. Los sectores de los lados se deben a un posible defecto del registro de los puntos en Lidar, ya que tampoco se encuentran en el fragmento original, sin filtros aplicados. El tercer sector, de mayor amplitud, se debe a situar el Lidar en la parte frontal de un carro a una altura de 1.3m aproximadamente, de forma que sus impactos se encuentran en el propio carro, como se puede observar en las nubes concentradas en el centro de la imagen.

Lo siguiente que realiza el algoritmo es el filtrado de suelo, tras el cual se obtienen los resultados de la Figura 10:

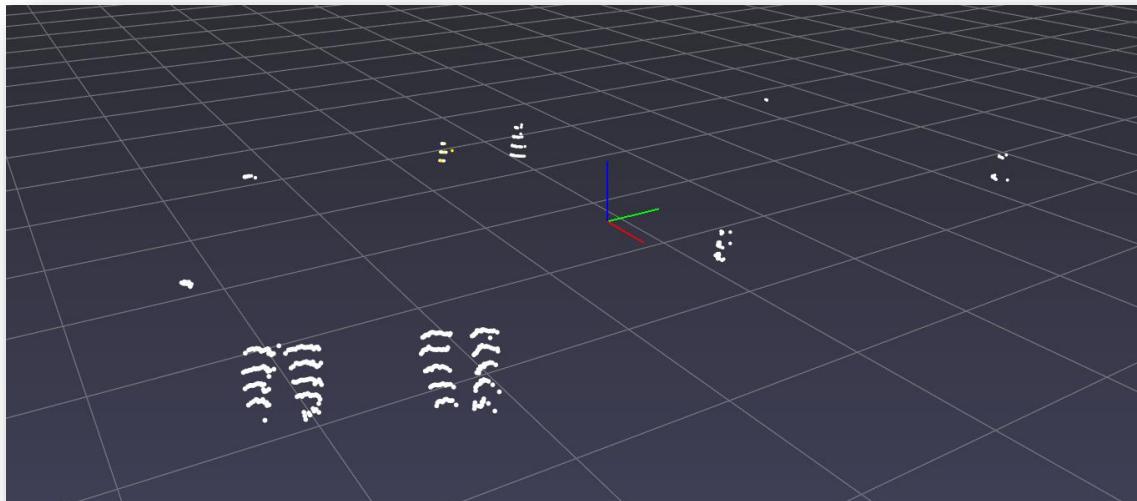


Figura 10: nube de puntos de la fase Ground Filter tras aplicar el filtrado de suelo

El número de puntos resultante es por tanto de 442, un 3% del total de puntos que entran a esta primera fase del algoritmo. En la figura observamos que únicamente restan sobre la pista los 4 conos, segmentos de las piernas de dos personas y otras nubes de menor tamaño de las que no se pueden extraer conclusiones.

Si representamos el plano obtenido por el algoritmo, resulta (Figura 11):

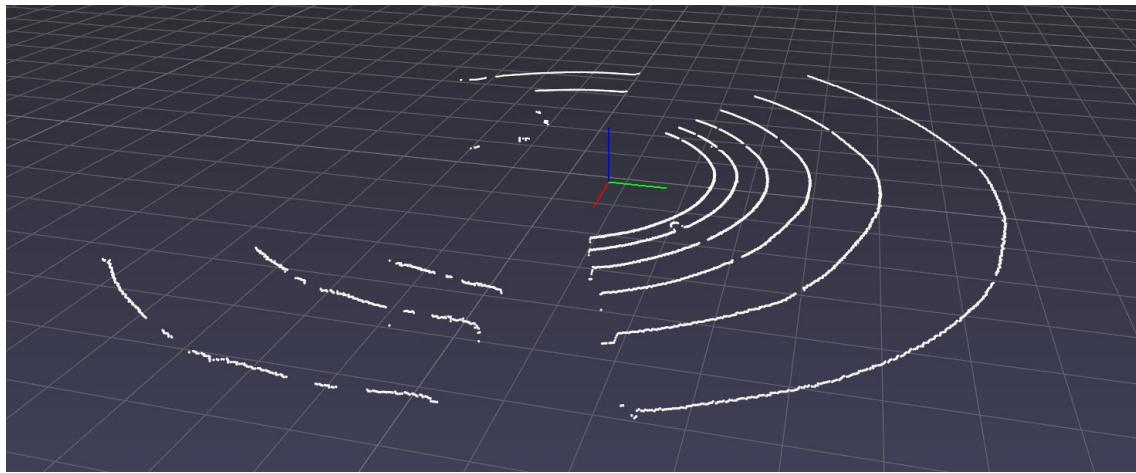


Figura 11: Plano de suelo obtenido por la fase de filtrado de suelo con los parámetros por defecto

El plano de suelo está compuesto por tanto por 4607 puntos, la mayoría de los puntos que componían el segmento tras los filtros de altura y distancia, como era lógico. Observamos que el Lidar detecta como puntos de suelo los impactos en el carro que tratábamos anteriormente. Esto se debe a que en los segmentos que ocupan la franja donde se encuentran estos puntos, no hay puntos de suelo, y por tanto el algoritmo



computa esos puntos como si se tratase de suelo. No pudiendo ser estas nubes conos en ninguna circunstancia, su filtrado no define un problema en primera instancia.

Si buscamos más detalle en la efectividad del algoritmo, podemos representar algunos de los resultados obtenidos modificando algunos parámetros:

- Si modificamos el número de divisiones:

Primero, lo establecemos en **n_divisions=1** suponiendo que no necesitamos segmentar por ángulo: el algoritmo no funciona, no siendo capaz de calcular el plano.

Si incrementamos este valor a **n_divisions=2**, obtenemos un plano de suelo compuesto por 1112 puntos, pudiéndose observar en la Figura 12 que no corresponde al plano de suelo.

Si de nuevo lo incrementamos a **n_divisions = 5**, obtenemos un plano de suelo compuesto por 3781 puntos, pudiéndose observar en la Figura 13 de nuevo que contiene puntos que no pertenecen al suelo como restos de partes corporales.

El número de divisiones no puede ser por tanto de un valor bajo al no permitir un cálculo adecuado del plano de suelo. Por otro lado, un valor excesivo también proporcionaría problemas, si existiesen segmentos con pocos puntos, en los que registraría puntos que no pertenecen al suelo como puntos de suelo. Otorgando el valor **n_divisions=12**, se guarda como valor por defecto.

NOTA: el valor del número de divisiones no afecta al tiempo de computación del algoritmo al recorrerse la nube de puntos una única vez para asignar los puntos a los diferentes sectores.

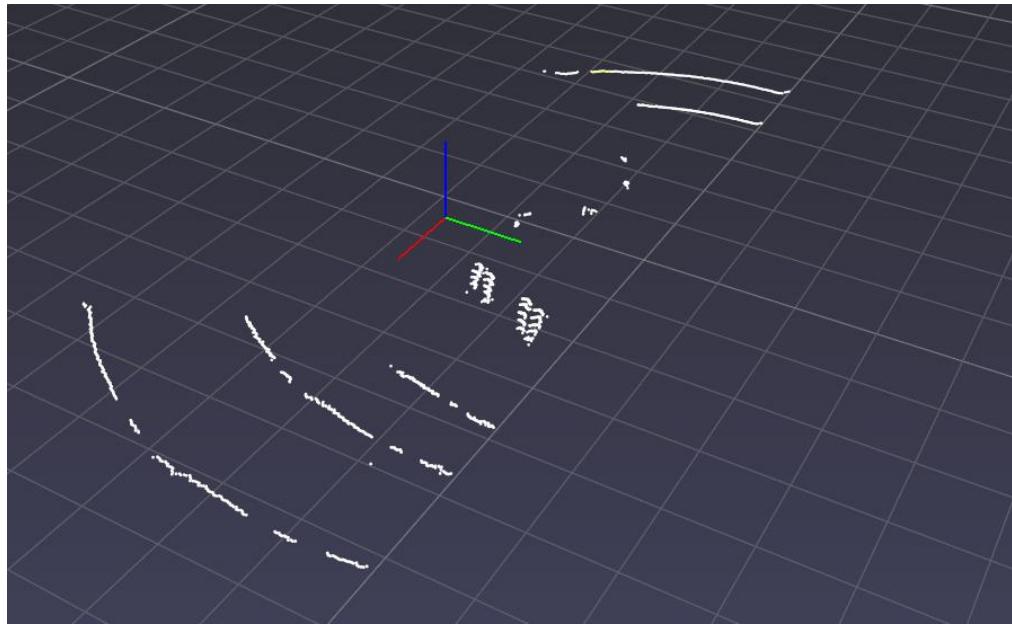


Figura 12: Plano de suelo obtenido para $n_divisions=2$

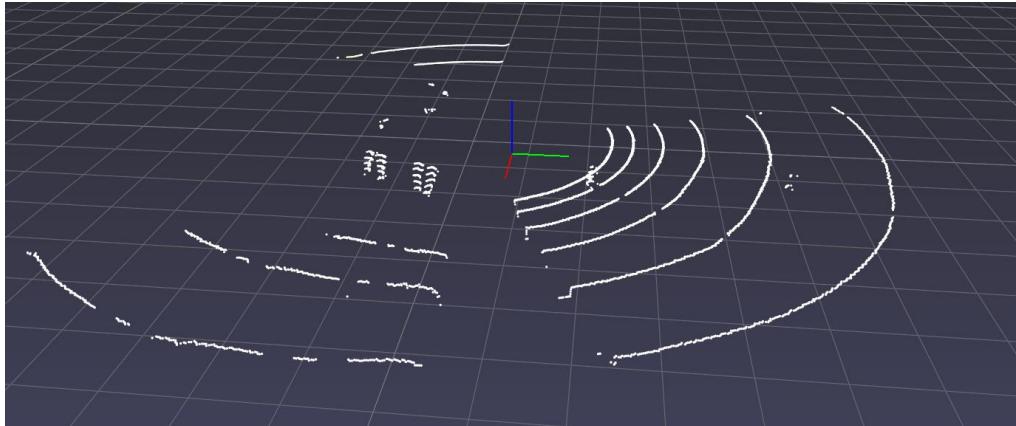


Figura 13: Plano de suelo obtenido para $n_divisions=5$

- Si modificamos el número de iteraciones:

Recordamos que habíamos establecido **n_iter=2**, que resultaba en un plano compuesto por 4607 puntos, por lo que vamos a probar otros valores y comparar sus resultados:

Si establecemos **n_iter=1**, obtenemos un plano de 4602, diferenciándose por tanto en menos de un 0.01% del original. Este resultado lo podemos observar en la Figura 14.

Incrementando el valor a **n_iter=3**, **n_iter=4**, **n_iter=5**, etc. Los planos obtenidos son todos de 4607 puntos, por lo que no merece la pena incrementar el valor.



Aunque $n_iter=1$ muestra buenos resultados para este frame, se guarda $n_iter=2$ como valor por defecto por seguridad.

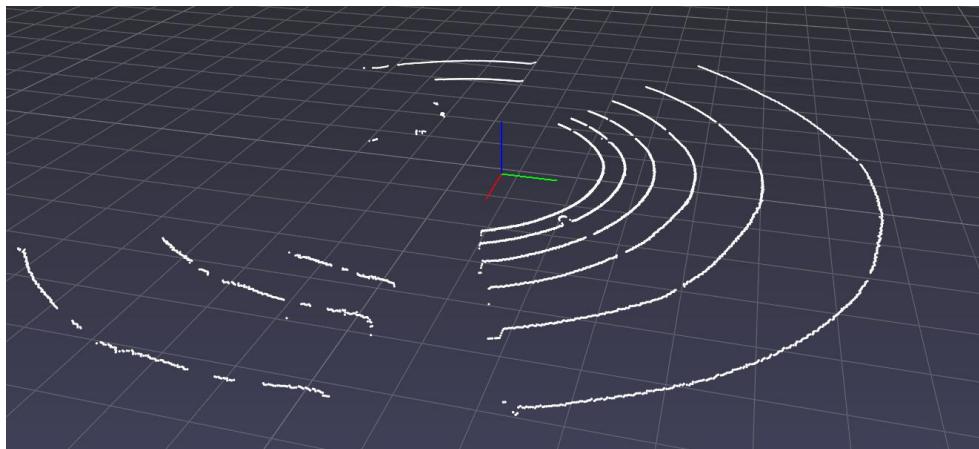


Figura 14: plano de suelo obtenido para $n_iter = 1$

Se podrían obtener resultados equivalentes modificando el resto de los parámetros. Los valores escogidos mostraron una precisión como la mostrada en la ejecución del algoritmo. Estos valores están sujetos a cambios en función de su precisión en la ejecución con datasets reales.

3.3.5 CONCLUSIONES

El funcionamiento del algoritmo de filtrado de altura, distancia y suelo es correcto, resultando en un filtrado del 97% de los puntos del dataset registrado en una prueba de campo en el INSIA. En él se observa que el suelo, principal objetivo del algoritmo, se encuentra totalmente filtrado.

El funcionamiento del algoritmo a tiempo real en un dataset equivalente al de la competición podrá ser evaluado en la etapa de implementación en ROS2. En esta implementación se obtendrá una medida real del tiempo de ejecución al que puede funcionar y de los resultados que proporciona, pudiéndose pulir los parámetros introducidos al algoritmo y establecidos como valores por defecto.



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



3.4 CLUSTERING

3.4.1 RESUMEN

Una vez la nube de puntos ha sido filtrada por altura y distancia y el suelo ha quedado filtrado, la nube restante corresponderá a los conos y otros objetos que puedan estar situados sobre la pista. Esta nube de puntos estará disociada en diferentes conjuntos o subnubes distanciados en mayor o menor medida entre ellos. El algoritmo de clustering busca asociar cada una de las subnubes de puntos contenidas en esta última a objetos diferenciados.

En esta sección se detalla cómo se realizó el algoritmo de clustering con Lidar, partiendo de un algoritmo conocido de aprendizaje no supervisado, denominado DBSCAN. Una explicación de los objetivos del código, seguida por su desarrollo detallado, será concluida con resultados a lo largo de la sección.

3.4.2 OBJETIVOS

1. Separación precisa de las nubes de puntos en sus diferentes subnubes de puntos constituyentes.
2. Alta frecuencia de ejecución del algoritmo. Siendo la velocidad de giro estándar del VLP16 de 600rpm (10Hz) se busca aproximarse lo mayor posible a esta última.
3. Mínima complejidad de computación, para dedicar los recursos de hardware a otras funciones del sistema. Este objetivo va muy ligado al objetivo 2.

3.4.3 DESARROLLO

ELECCIÓN DEL ALGORITMO



Para realizar una división de una nube de puntos genérica en las nubes de puntos que la componen, se emplean los algoritmos de Inteligencia Artificial (Machine Learning) denominados como aprendizaje no supervisado. Estos, según sus correspondientes parámetros, caracterizarán la nube de puntos, subdividiéndola en grupos respondiendo a unos criterios.

Algunos de los algoritmos más importantes para realizar un proceso de clustering son:

- Hierarchical clustering: busca crear una jerarquía entre los grupos.
- K-means: es un método de agrupamiento que tiene como objetivo la partición de un conjunto de n observaciones en k grupos en el que cada observación pertenece al grupo cuyo valor medio es más cercano. Se debe especificar de antemano el número de clusters a crear.
- Mixture models: el modelo de mezclas gaussianas también se utiliza para crear clusters de diferentes tamaños y estructuras, y requiere de una especificación del número de clusters inicialmente.
- DBSCAN: este método se basa en un agrupamiento en función de la densidad espacial, siendo capaz de diferenciar los clusters de los puntos que componen ruido, no perteneciendo a ningún cluster. Funciona bien para clusters de densidad parecida.
- OPTICS Algorithm: este método es equivalente a DBSCAN, funcionando mejor para clusters de diferente densidad, la mayor debilidad de DBSCAN.

Justo a esto, se presentan las características que van a tener las nubes de puntos de los conos:

- Se desconoce inicialmente el número de conos que hay en cada frame.
- Los conos pertenecen todos a la misma jerarquía, no siendo necesaria una distinción de esta última.
- Los conos tienen una densidad de puntos parecida al ser objetos equivalentes.

Por tanto, como no existe jerarquía y se desconoce el número de conos que se debe detectar en un frame, se descartan los algoritmos: Hierarchical clustering, Mixture models y K-means.

Entre los algoritmos restantes, OPTICS y DBSCAN, ambos cuentan con un tiempo de procesado similar [12], y deberían ser equivalentes para la tarea propuesta. En el caso de los conos, al tener una densidad de puntos parecida, parece más razonable comenzar utilizando DBSCAN y ajustar sus parámetros para que detecte con precisión los clusters de dicha densidad. Si su resultado no fuese el esperado, podría implementarse OPTICS u alguna otra opción funcional en lugar de DBSCAN.



POLITÉCNICA

UNIVERSIDAD
POLÍTÉCNICA
DE MADRID



EXPLICACIÓN DEL ALGORITMO

[13] El algoritmo de clustering se basa fundamentalmente en el algoritmo llamado DBSCAN (Density-based Spatial Clustering of Applications With Noise). DBSCAN es un algoritmo de agrupamiento basado en densidad, ya que encuentra un número de grupos comenzando por una estimación de la distribución de densidad de los nodos correspondientes.

Detectar clusters de diferente densidad, tamaño y forma puede ser muy complicado sobre todo si en la nube de puntos hay ruido y outliers (puntos que no deberían pertenecer a ningún cluster). Para gestionar dichos datos, Martin Ester, Hans-Peter Kriegel, Jörg Sander y Xiaowei Xu propusieron DBSCAN [14].

El objetivo principal de DBSCAN es localizar regiones de gran densidad separadas entre sí por regiones de baja densidad. La pregunta es, ¿cómo medir la densidad de una región? Se proponen dos definiciones:

1. **Densidad de un punto P:** Número de puntos dentro de un círculo de radio ϵ alrededor de este. El vecindario del punto se define por tanto como:

$$N(p) = \{q \in D \mid dist(P, q) \leq \epsilon\}$$

2. **Densidad de una región:** para cada punto en un cluster, el círculo de radio ϵ alrededor de este contiene al menos un número mínimo de puntos (MinPts).

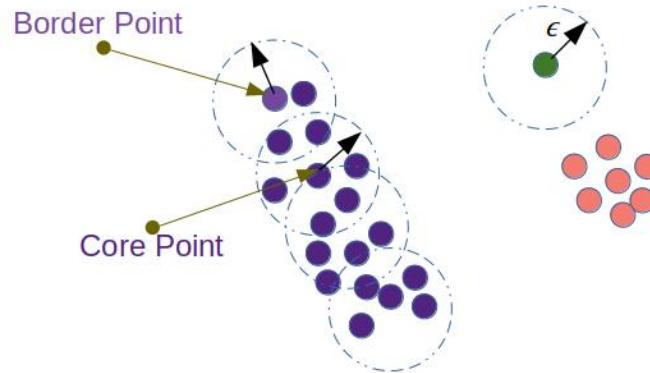
Además, se definen:

1. **Núcleo de una región:** Una vez definida la densidad, afirmamos que un punto es núcleo de una región densa si:

$$|N(p)| \geq \text{MinPts}$$

2. **Punto frontera de una región:** punto que contiene un número inferior a MinPts en su ϵ -vecindario (N) pero contenido en el vecindario de otro punto núcleo de la región.

3. **Punto ruido:** el resto de los puntos.



$$N_{Eps}(p) = \{q \in D \text{ such that } dist(p, q) \leq \epsilon\}$$

$$\epsilon = 1 \text{ unit}, \text{MinPts} = 7$$

Figura 15: Imagen explicativa de los tipos de puntos [13]

En la Figura 15 se muestra una representación visual de los tipos de puntos que pueden diferenciarse en este algoritmo.

Esta aproximación puede presentar fácilmente un problema. Si para incluir los puntos frontera en el cluster se reduce demasiado MinPts, podrían incluirse también los puntos ruido, dando lugar a errores. Para evitar esto, se proponen dos nuevas definiciones:

1. **Directamente alcanzable:** un punto a es directamente alcanzable desde un punto b si:

- $|N(b)| \geq \text{MinPts}$; Es decir, b es un punto núcleo.
- $a \in N(b)$;

De aquí deducimos que la noción de directamente alcanzable no es simétrica, puesto que, aunque en este caso a será directamente alcanzable desde b , b no lo será desde a debido a que $|N(a)| \leq \text{MinPts}$.

2. **Alcanzable densamente:** un punto a es alcanzable densamente desde un punto b con respecto a ϵ y MinPts , si existe una cadena de puntos b_1, b_2, \dots, b_n , donde $b_1 = b, b_n = a$, tal que b_{i+1} es directamente alcanzable desde b_i .

Tampoco se trata de una propiedad simétrica.

3. **Densamente conectados:** si dos puntos frontera pertenecen al mismo cluster pero no comparten el mismo punto núcleo, entonces decimos que dichos puntos están densamente conectados si existe un punto núcleo desde el cual son densamente alcanzables.

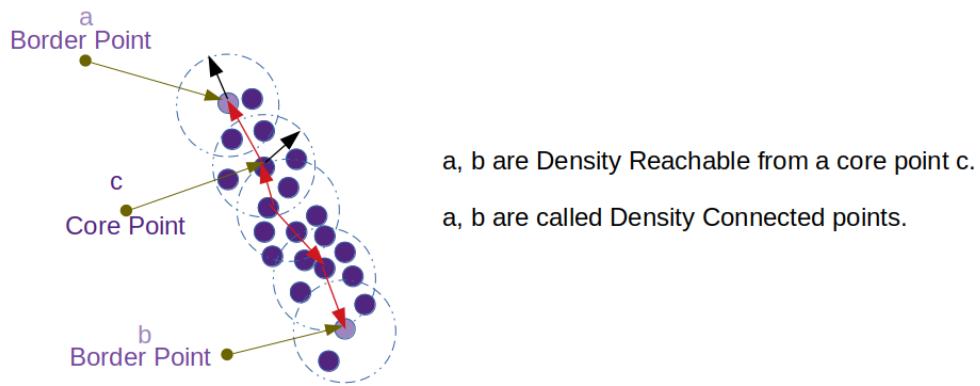


Figura 16: Dos puntos frontera, a y b, densamente conectados desde el punto núcleo c [13]

En la Figura 16 se ejemplifica de forma gráfica la situación en la que dos puntos se encuentran densamente conectados.

Una vez presentados los conceptos principales del algoritmo DBSCAN, presentamos sus pasos principales:

1. El algoritmo comienza con un punto arbitrario que no ha sido visitado aún. Se calcula su vecindario con el parámetro ϵ .
2. Si este punto contiene **MinPts** en su vecindario, la creación del cluster comienza. Si no se cumple dicha condición, el punto se etiqueta como ruido. En dicho caso, este punto puede ser encontrado luego como vecindario de un punto diferente y pasar a formar parte de un cluster.
3. Si un punto es catalogado como núcleo, todo su vecindario, junto con sus respectivos vecindarios (en el caso de que también sean puntos núcleo) son añadidos al clúster.
4. El proceso prosigue hasta que el cluster densamente conectado ha sido definido completamente.
5. El proceso se reinicia con un nuevo punto que puede ser parte de un nuevo cluster o ser catalogado como ruido.

Habiendo definido el algoritmo, dos desventajas claras salen a la luz:

- Si la base de datos la componen clusters de diferente densidad, DBSCAN fracasa en su segmentación, ya que el clustering depende de ϵ y Minpts, los cuales se mantienen constantes para todos los clusters.



- Si el programador no conoce bien las características de la base de datos, puede resultar complicado establecer los valores de ϵ y Minpts, proceso que puede requerir múltiples iteraciones.

¿Cómo ha sido implementado el algoritmo?

Por suerte, existe una librería que integra el algoritmo a partir de un array o matriz de puntos. Se trata de Scikit-learn [15], una biblioteca para aprendizaje automático de software libre en Python.

ANÁLISIS DEL CÓDIGO

De igual forma que para el algoritmo de filtrado de suelo, el código ha sido enteramente realizado en Python.

El código parte de las nubes de puntos catalogada como “no suelo”, salida de la ejecución de GroundPF, tal y como se explicaba en la sección de **Ground Filtering**. Es entonces cuando se ejecuta la función **DBSCAN().fit()**. [15] Dicha función toma como parámetros **ϵ** y **MinPts**. En cuanto a la entrada de la nube de puntos, esta debe ser un array compuesto por puntos con sus tres coordenadas. Es decir, de la forma: $[[x_1, y_1, z_1], [x_2, y_2, z_2], \dots, [x_n, y_n, z_n]]$.

Una vez computada la función, asignada a una variable llamada por ejemplo **db**, asignamos a una nueva variable llamada **labels** el conjunto de etiquetas de los clusters mediante **db.labels()**. **labels** será entonces un array de etiquetas con la misma longitud que la nube de puntos entrada de DBSCAN. Cada elemento será entonces la etiqueta que referencia el cluster al que pertenece cada punto, de la forma: $[l_1, l_2, \dots, l_n]$.

Los puntos ruido llevarán como etiqueta **-1**, siendo filtrados posteriormente al determinar el número de clusters en la asignación de la variable **nclusters**.

En la parte inferior del código encontramos una sección cuyo objetivo es representar los clusters en la pantalla coloreados de forma diferente.

PARÁMETROS PARA SELECCIONAR

Como se ha ido comentando, existen fundamentalmente dos parámetros que se pueden modificar en el algoritmo, **ϵ** y **MinPts**, correspondientes a las definiciones presentadas



precedentemente. Su determinación se realiza cuando se conoce la nube de puntos con la que se va a tratar, de forma analítica o empírica.

En este caso se ha procedido a su determinación de forma empírica, para que la segmentación en clusters según las distancias y tamaños similares a los de la competición diese resultados precisos.

Por ello, se determinaron como valores por defecto:

- $\epsilon = 0.4$
- $MinPts = 4$

3.4.4 RESULTADOS

El algoritmo fue testeado en un frame proveniente de una prueba realizada en el Insia, en la que se situaron cuatro conos sobre la pista, y se registró un recorrido con el Lidar a través de Veloview, registrando así archivos de tipo pcap, que pueden ser transformados a sus frames correspondientes en formato csv.

Para analizar los resultados, partimos de la nube de puntos proporcionada por la fase de filtrado de suelo, mostrada en la Figura 17:

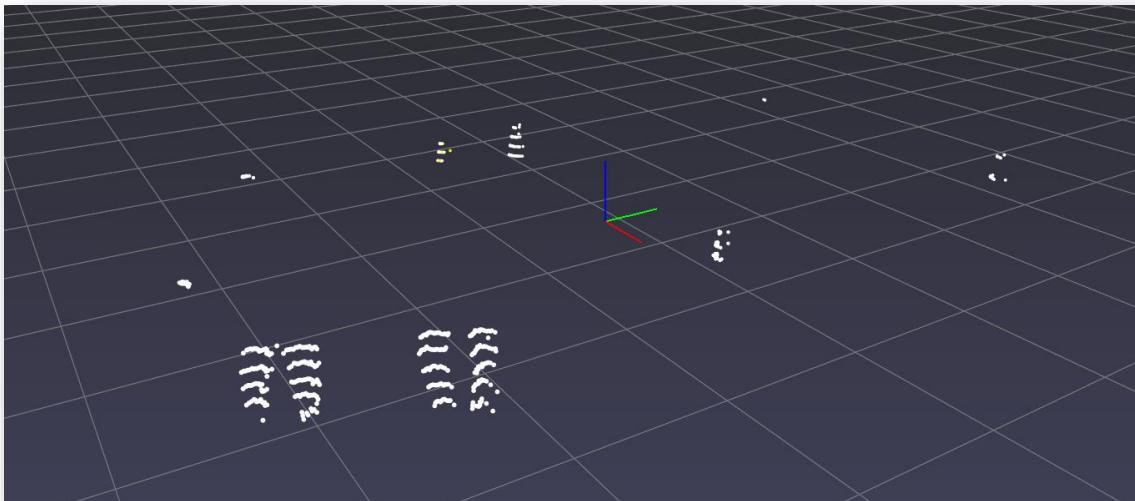


Figura 17: nube de puntos de la fase Ground Filtering tras aplicar el filtrado de suelo

Sobre esta nube, compuesta por 442 puntos, se ejecuta la fase de clustering, brindando los siguientes resultados (Figura 18):

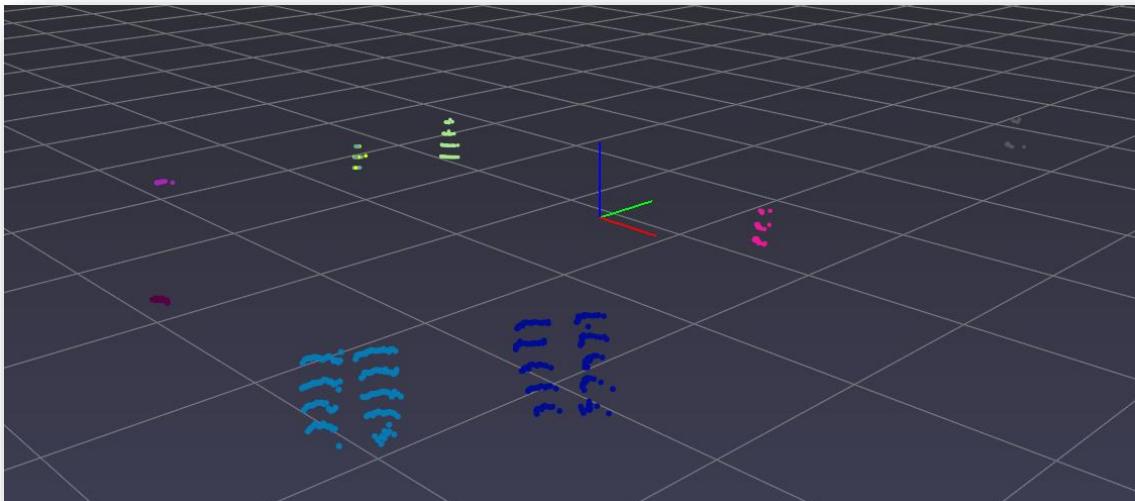


Figura 18: nube de puntos de la fase Clustering tras aplicar la división por subnube de puntos

La nube de puntos completa obtenida contiene 439 puntos. Los 3 que restan han sido considerados como ruido por el algoritmo, no componiendo de esta forma ningún cluster o subnube de puntos.

Respecto a los 439, se subdividen en 8 subnubes de puntos, de colores diferentes, como se puede observar en la figura.

Observamos en la figura cómo los parámetros seleccionados por defecto dan buen resultado para la segmentación. Su modificación tendría implicaciones, tanto en el número de clusters detectados, como en aquellos puntos considerados como ruido.

Estos parámetros toman un valor pequeño debido a que, en condiciones reales, existirán conos a diferentes distancias y establecer una variable **MinPts** de mayor valor podría provocar que los conos a mayores distancias no fuesen detectados como clusters. Además, el objetivo de esta sección es separar las nubes de puntos, sin ser preocupante que resten clusters de correspondientes a objetos de mayor o menor tamaño que los conos, ya que estos serán tratados en filtros posteriores.

Para verificar esto, introducimos en el algoritmo **MinPts** = 20, obteniendo 405 puntos en los que se ha perdido la información de uno de los conos, como se puede observar a la derecha (Figura 19):

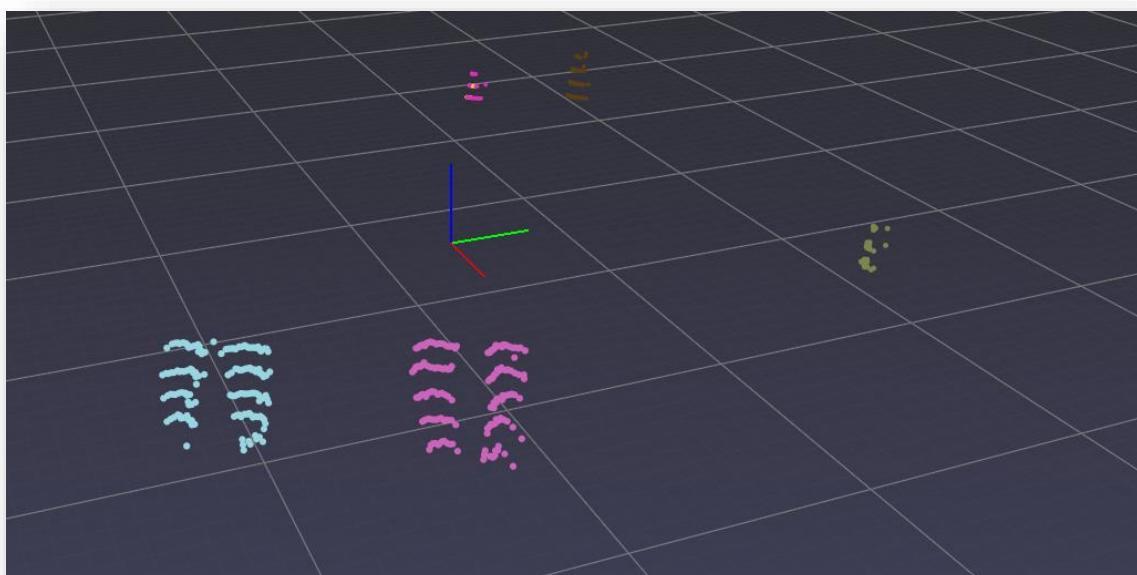


Figura 19: Clustering con $\text{MinPts} = 20$

Si introducimos un valor de **MinPts** = 10, resultan 434 puntos (6 menos que para $\text{MinPts}=4$), y apareciendo el cono de la derecha (Figura 20).

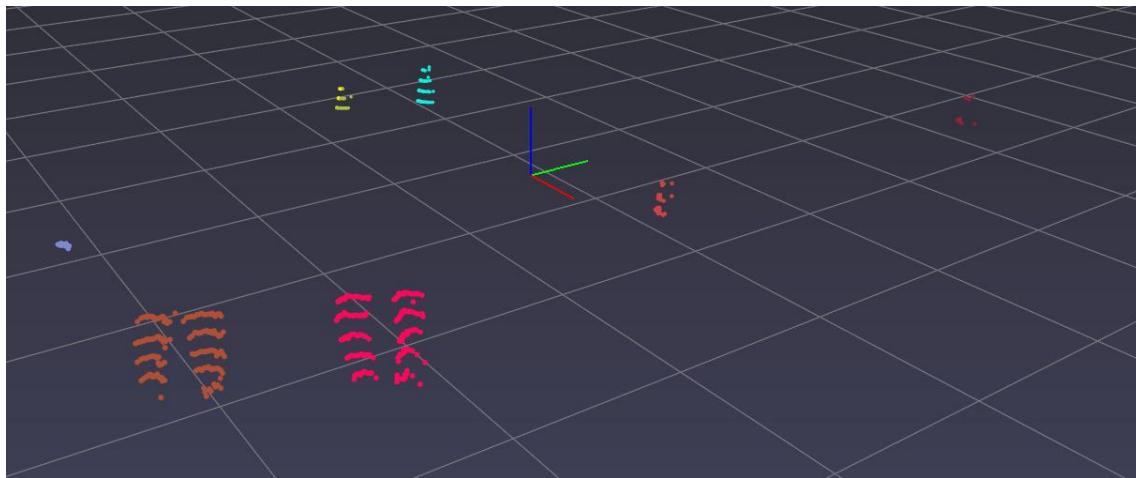


Figura 20: Clustering con $MinPts = 10$

Aumentar los valores de ϵ y $MinPts$ tendría sentido si para el caso analizado, hubiese clusters considerados diferentes (dos conos diferentes, o un cono y una persona, por ejemplo) y que el algoritmo los estuviese asociando a la misma subnube de puntos. Como no es el caso, aumentar los valores únicamente puede resultar en conos detectados como ruido, algo que se busca evitar para conservar toda la información posible. Por ello, se conservan los valores definidos como valores por defecto, por su buen resultado, a expensas de ser modificados si en la implementación en ROS2 no ofreciesen la misma precisión.

3.4.5 CONCLUSIONES

Con los resultados obtenidos en un frame registrado en el INSIA, la fase de clustering distingue claramente los diferentes objetos situados sobre la pista.

Para tener información acerca de una correcta ejecución de este proceso en tiempo real debe implementarse el algoritmo en ROS/ROS2 y ejecutarlo en un dataset equivalente a lo que pudiera tenerse el día de la competición.



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



3.5 RECONSTRUCTION

Se procede a la

3.5.1 RESUMEN

Como el principal objetivo del equipo de “Driverless” es diferenciar los conos de la pista, se entiende que todos los conos estarán apoyados en el suelo. Por lo tanto, si en la etapa de *ground filtering* se eliminó la parte de suelo de uno cono, en este algoritmo se reconstruirá.

Por lo tanto, una vez asociadas las subnubes de puntos a objetos diferenciados (*clustering*), es el momento de obtener la parte de suelo que se le quitó a estos objetos. Este algoritmo se encarga de eso: reconstruir el suelo de cada uno de los clústeres obtenidos en el algoritmo *clustering*.

Esto servirá para posteriormente, en el siguiente módulo de detección de conos, detectar con mayor precisión si un clúster se trata de un cono, o no.

3.5.2 OBJETIVOS

1. Reconstrucción de tamaño personalizable del suelo de los clústeres.
2. División del algoritmo en funciones simples para su mejor comprensión.
3. Representación 3D de los clústeres con su suelo reconstruido.

3.5.3 DESARROLLO

Explicación del Algoritmo

Se presenta el algoritmo *ObtainClusterGround.ipynb*, el cual se apoya en una librería muy básica llamada *SmallestEnclosingCircle.py* [16]. La función que nos interesa de este programa es *make_circle*, que calcula el círculo más pequeño que encierra un conjunto de puntos arbitrarios en un plano.

Esto nos sirve para delimitar los límites del clúster para reconstruir su suelo. El círculo que engloba a todos los puntos de un clúster proyectados en el plano xy, se vería como en la Figura 21, en color rojo.

La función `make_circle` tendrá, como entrada, un clúster. Sus salidas serán 3 variables: la coordenada X del círculo, la coordenada Y del círculo y el radio R del círculo.

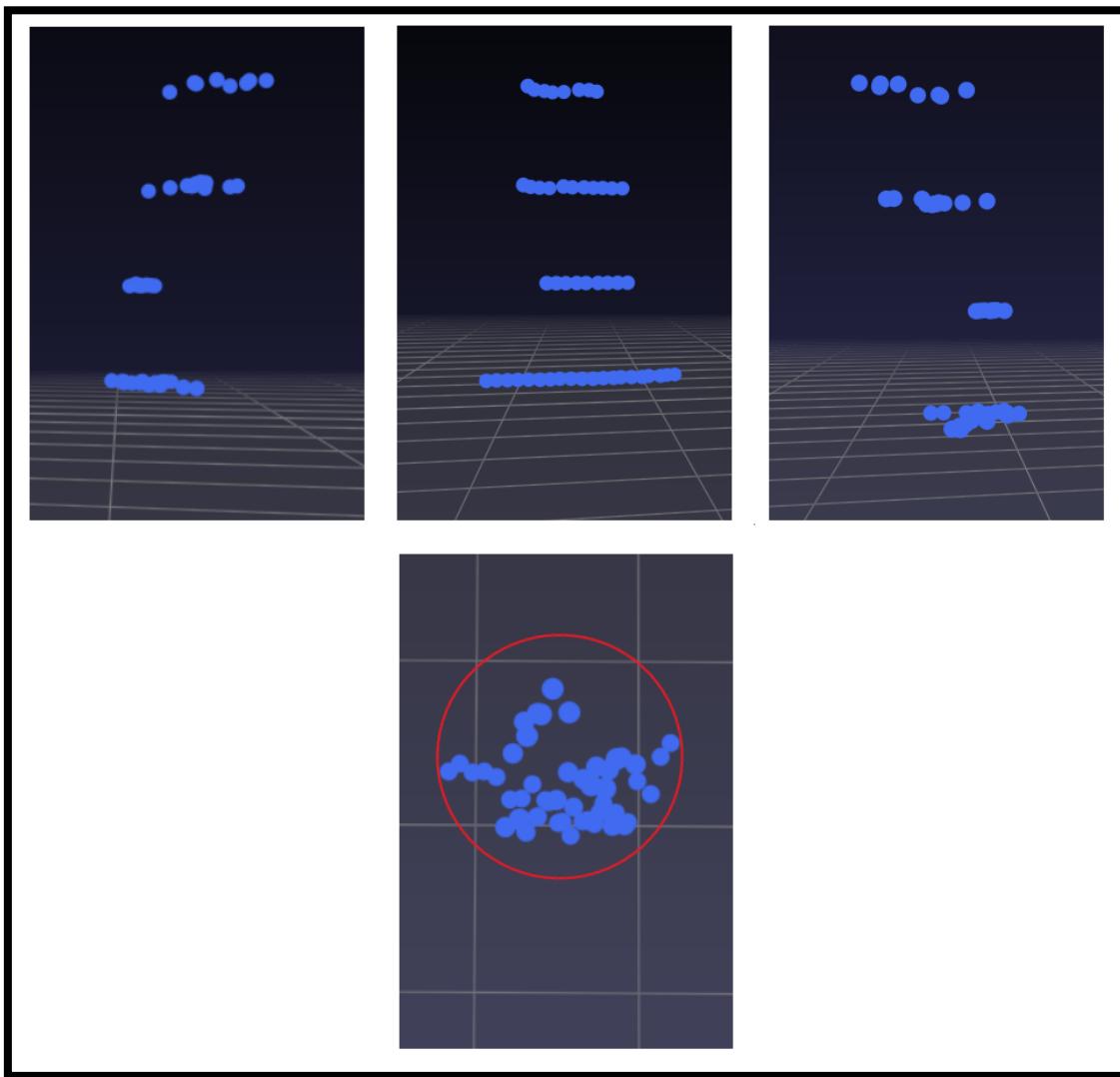


Figura 21. Representación en sistema diédrico del círculo.

Una vez obtenido el círculo que engloba los puntos del clúster, como se indica en la Figura 21, se pasa a obtener los puntos que estén dentro de ese límite del array de puntos de suelo `pg` (obtenido en el módulo `DelphiMejorado.ipynb`). Estos puntos serán los puntos de suelo del clúster, que en el caso de un cono (que es lo que más nos interesa), formarían la base de este.

Estos puntos de suelo, pertenecientes al clúster objetivo, se unirán a este, obteniendo así un array denominado `ClusterWithGround`.

Si repetimos este mismo proceso para cada uno de los clústeres de nuestra nube de puntos, se podrán guardar todos en la misma variable `ClusterWithGround`, logrando así un array de clústeres con suelo.



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



Finalmente se representan en 3D todos los clústeres reconstruidos.

Análisis del Código

El código, enteramente realizado en Python, se encuentra en el Anexo.

En este apartado se analizará el código, línea por línea, para su correcta comprensión.

De igual forma que para el algoritmo de *clustering.ipynb*, el código ha sido enteramente realizado en Python y se encuentra en el anexo.

Inicialización de parámetros de entrada

El código parte del array denominado *conjunto*, que contenía todos los clústeres generados con el algoritmo *clustering.ipynb*, y del array *pg*, contenido también en el mismo módulo.

Además, se inicializa la variable *th*, que indica el porcentaje de tamaño extra que podrían tener los círculos que engloban cada clúster (elegido por el usuario).

Por último, se inicializa el array *ClusterWithGround* con tamaño igual al número de clústeres que se tenga en el array *conjunto*.

Funciones del algoritmo

Para la correcta comprensión del algoritmo, se crearon las siguientes funciones previas:

1. EncloseCluster: Función que, a partir de un clúster de entrada, devuelve las coordenadas y el radio del círculo que encierra a la nube de puntos proyectada en el plano xy.
2. DistanceToCenter: Función que devuelve la distancia que hay entre el centro del círculo anteriormente hallado y un punto de *pg*(la nube de puntos que se etiquetó como suelo).
3. ValidGround: Esta función tiene como entrada el círculo hallado, y usa *DistanceToCenter* para encontrar todos los puntos de *pg* que están dentro del círculo previamente hallado. Estos puntos los devuelve dentro de un array al que se le llama *ClusterGround*.
4. ObtainGround: Función que, a partir del array de clústeres (*conjunto*), de los puntos de tierra (*pg*) y de un parámetro *th* que modifica la distancia máxima al centro del círculo para poder pertenecer al suelo del clúster, entrega como salida el array *ClusterGround*.



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



5. UnirClusters: Función que, a partir de un clúster (*conjunto[i]*) y su tierra (*ClusterGround*), une ambos para formar un sólo array de puntos (*ClusterWithGround*).

Cuerpo del algoritmo

El cuerpo del algoritmo *ObtainClusterGround* básicamente se basa en un bucle que usa las funciones anteriormente detalladas para cada uno de los clústeres del array *conjunto*.

Representación gráfica de los puntos

Para que se represente cada clúster con un color distinto y pueda ser más visual para el usuario, se hace uso de atributos de color, de la misma forma que se usó en el módulo *clustering*.

3.5.4 RESULTADOS

Finalmente se presenta, en la **Figura 23**, la comparación entre los clústeres sin suelo y con suelo. La optimización del algoritmo, como ya se ha comentado, se puede realizar ajustando el parámetro de distancia máxima al centro del cluster. Vemos cómo, a partir de la salida de *clustering*, y tras la aplicación del algoritmo, se consigue una reconstrucción del clúster con un aumento del 23,3% de los puntos.

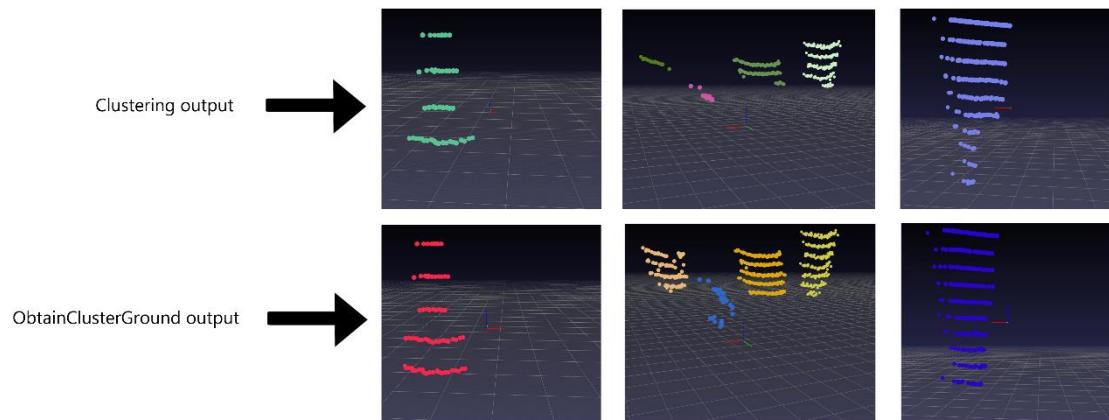


Figura 23. Salida de clústeres sin suelo (arriba) y con suelo (abajo)



3.6 Non-cone filtering

3.6.1 RESUMEN

Esté módulo parte del output de la fase de reconstrucción, que corresponde con los cluster una vez reconstruidos y tiene el objetivo de diferenciar si un cluster es un cono o no es un cono.

Se ha ideado un método para que se pueda saber si un cluster es un cono a partir del número de puntos que debería tener un cono a la distancia que se encuentre del LIDAR.

Tras haber reconstruido en la fase previa la geometría completa de los conos con los puntos de suelo que podrían haber sido del cono, se llega a un número de puntos que forman el cono que es conocido.

En esta memoria se explicará cuál es el método para llegar a saber con exactitud el número de puntos que compone el cluster de un cono.

3.6.2 Objetivos

1. Detectar si los clusters con suelo son conos o no.
2. Filtrar los clusters no conos y crear un array definitivo con los clusters conos.
3. Representación 3D de los clústeres conos.

3.6.3 Desarrollo

EXPLICACIÓN DEL ALGORITMO

Las fases en las que se divide este algoritmo son;

1. Importar datos de clusters con suelo.
2. Buscar el centroide de dichos clusters y la altura mínima del suelo.
3. Hallar la distancia del centroide al centro de coordenadas (LIDAR).
4. Aplicar la ecuación para hallar el número de puntos que corresponden a un cono a esa distancia.
5. Filtrar con un rango todos los clusters por encima y por debajo de ese número de puntos.
6. Generar un nuevo array de puntos exclusivamente pertenecientes a conos.
7. Representar en 3D y con colores los conos.

INICIALIZACIÓN DE PARÁMETROS DE ENTRADA

Se importan todas las librerías anteriormente usadas además de la librería *math*, ya que se van a utilizar algunas de sus funciones y se importa el parámetro pi de esta biblioteca.



Además, se importan los arrays generados de puntos de suelo y puntos de no suelo de la librería Ground Point Filtering. Se utilizarán los puntos de suelo para hallar la altura mínima del mismo. Esto se debe a que el LIDAR toma los datos desde una altura y el suelo se queda en una coordenada negativa. Puesto que los conos siempre se apoyan en el suelo, se obtiene dicha coordenada.

Se importa el array ClusterWithGround, el cual contiene todos los puntos de los clusters con suelo y se inicializa la variable de un threshold que se utilizará posteriormente para la detección del cono con un filtrado.

FUNCIONES DEL ALGORITMO

Para el funcionamiento del algoritmo se han creado las siguientes funciones:

6. Zground: Esta función se encarga de, a partir de los puntos de suelo, extraer la mínima coordenada de altura, es decir, en el eje z, para tomarla de referencia par hacer las operaciones posteriores.
7. EncloseCluster: Esta función, ya utilizada en la etapa anterior para rodear los clusters se utiliza, en este caso, para obtener su centroide.
8. DistanceToLidar: Esta función se utilizará para, a partir de las coordenadas de los centroides de los clusters, hallar las distancias de estos al centro de coordenadas.
9. Expectedconepoints: Esta función es la encargada de, a partir de la distancia a la que se encuentra cada cluster, generar el número de puntos que debería tener un cluster que fuese cono a esa distancia. Para esto, se tiene en cuenta, tanto la resolución del Lidar, horizontal y vertical, como la altura y anchura del cono normalizado utilizado en la competición.

Para esto, se utiliza la siguiente ecuación:

$$\text{expectedpoints} = \frac{1}{2} * \left(\frac{h_c}{2 * d * \tan\left(\frac{r_v}{2}\right)} \right) * \left(\frac{w_c}{2 * d * \tan\left(\frac{r_h}{2}\right)} \right)$$

Siendo h_c la altura del cono, w_c la anchura, d la distancia del centroide al centro de coordenadas, r_v la resolución vertical del LIDAR y r_h la resolución horizontal del LIDAR.

CUERPO DEL ALGORITMO

El cuerpo del algoritmo se ha dividido en dos partes, un primer bucle que utiliza las funciones previamente definidas para obtener el número de conos que hay. Al obtener el número de conos, se puede crear el array Clusterwithcone con los espacios suficientes para asociarle las coordenadas de los puntos del cono en un bucle posterior.

El segundo bucle tiene la función de introducir en Clusterwithcone todos los puntos correspondientes al cluster cono.

Esta estructura de un array de puntos de coordenadas (x,y,z) dentro de otro array se utiliza para organizar el array de manera de que todos los cluster siempre estén presentes en arrays diferentes dentro de un mismo array.

REPRESENTACIÓN GRÁFICA DE LOS PUNTOS

Se utiliza la misma expresión que en las librerías previas ya que se asocian una serie de atributos a cada cluster y posteriormente se introducen todos los clusters en un array que se visualizará a continuación con la librería pptk.

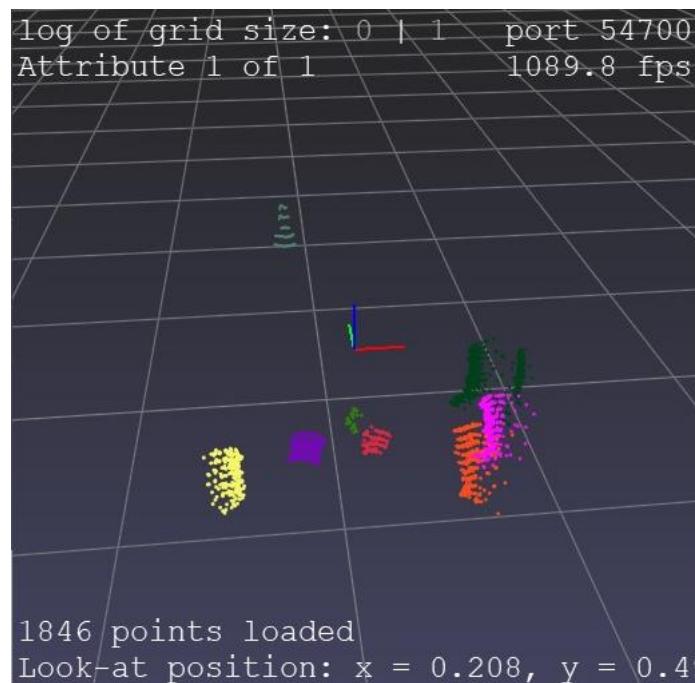


Figura 24. Representación de los Clusters antes de aplicar el Non-Cone Filter.
Fuente: Elaboración propia.

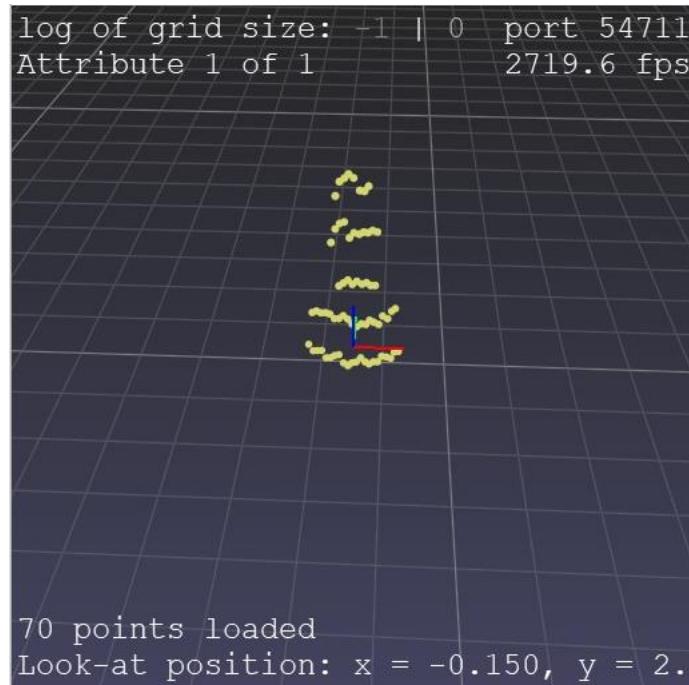


Figura 25. Representación de los Clusters después de aplicar el Non-Cone Filter. Fuente: Elaboración propia.

3.6.4 RESULTADOS

El resultado ha sido la creación de un array que contiene puntos que corresponden exclusivamente a conos. Igualmente, se puede observar que la reducción en el número de puntos es considerable pasando de 1846 a los 70 correspondientes a un cono. De la misma manera, se ha desarrollado el código para garantizar su ejecución en tiempo real y que posteriormente, en la fase de implementación en ROS2 se garantice su buen comportamiento.



3.5 IMPLEMENTACIÓN EN ROS2

3.5.1 RESUMEN

Esta sección recorre la implementación de los algoritmos en ROS2. La implementación en Python sirve de referencia para un primer ajuste de los códigos, pero no determina el comportamiento de estos en tiempo real y condiciones de competición.

Además, para conectar la detección con Lidar a la parte de visión y de control, es necesaria la utilización de un sistema como ROS/ROS2 que integre todas las partes y las distribuya en nodos, algoritmos que se ejecuten en bucle en tiempo real y se envíen la información entre ellos.

Para explicar el proceso, esta sección comenzará por el análisis de la elección de ROS2 frente a ROS, para luego tratar la implementación del sistema de detección con Lidar en ROS2.

3.5.2 OBJETIVOS

Los objetivos de esta sección comprenden:

1. Comprender por qué se eligió ROS2 frente a ROS para la implementación en tiempo real de los algoritmos de detección por Lidar en INGENIA-FSAE.
2. Entender por qué se implantó ROS2 en Ubuntu, estando dicho sistema disponible para Windows, sistema operativo que resulta más intuitivo.
3. Comprender los pasos que permiten implantar y conectar los diferentes algoritmos de detección, presentados uno a uno anteriormente
4. Incorporar un dataset real, equivalente al de la competición de FSUK
 - Ground-filtering
 - Clustering
 - Reconstruction
 - Non-Cone Filtering
5. Probar los algoritmos en tiempo real con dicho dataset y extraer resultados y conclusiones



3.5.3 DESARROLLO

3.5.3.1 ELECCIÓN ENTRE ROS Y ROS2

En primer lugar, resulta conveniente analizar las ventajas e inconvenientes de ROS y ROS2, para determinar en qué sistema trabajar [17].

ROS2 surge en 2015 con el objetivo de mejorar la seguridad de las máquinas que implementan este sistema, así como su funcionamiento en tiempo real. Por ello, vamos a analizar aquellas diferencias más significativas entre ambas versiones.

- **Lenguajes**

ROS2, a diferencia de ROS, permite trabajar en Python 3.5 y versiones de C++ como la 13 o la 14. ROS se encuentra limitado a Python2.7 y C++11. Muchas librerías han dejado o dejarán pronto de dar soporte a versiones antiguas de Python o C++ lo que convierte esta diferencia en clave a la hora de generar nuevos códigos.

- **Sistemas operativos**

ROS puede implementarse en Linux y, con ciertas dificultades, en MacOS. ROS2 permite implementarse en Windows, SO de mayor familiaridad con los usuarios. En el caso del INGENIA, no habíamos trabajado con Linux, luego resultaba más intuitivo operar en Windows.

- **Centralizado vs Distribuido**

En ROS es necesario definir un **Master** a la hora de ejecutar el sistema, al ser este de tipo **centralizado**. Por otro lado, ROS2 es de tipo distribuido, por lo que se pueden ejecutar los nodos siendo estos los que alertan al resto de nodos de su aparición o desaparición.

- **TCPROS vs DDS**

ROS2 hace uso de DDS (Data Disribution Service), un framework de arquitectura **publisher/suscriber** ideal para ROS2 y necesario para aplicaciones en tiempo real.



Existen más diferencias no comentadas debido a que no fueron consideradas en el momento de definir qué sistema utilizar por considerarse de menor relevancia.

Sin duda, ROS es una versión más madura, estable y con herramientas completamente funcionales. Al mismo tiempo, los errores que aún no han sido corregidos están en su gran mayoría documentados y permiten su correcta resolución. Es por ello por lo que en el ***UpmRacing-Driverless*** se utiliza ROS como sistema para implantación de los sistemas de detección y control.

Sin embargo, siendo el INGENIA una oportunidad de innovación y sin poseer conocimientos iniciales de ROS/ROS2, esta segunda versión aporta más frescura y puesta al día de requerimientos como un correcto funcionamiento en tiempo real. Además, permite su implantación en Windows, lo que permitía estudiar únicamente el funcionamiento de ROS2, reduciendo la dificultad de formar al equipo de INGENIA en Ubuntu al mismo tiempo.

Por estos motivos, se decidió optar por la formación en ROS2 y su implantación en Windows en una primera instancia. A continuación, evaluaremos dicho recorrido por Windows y el porqué de la transición posterior a Linux.

3.5.3.2 IMPLANTACIÓN DE ROS2 EN WINDOWS

Como comentábamos, el primer paso fue instalar ROS2 en Windows. Dicha instalación se realiza de forma sencilla gracias a los tutoriales presentes en la propia web de ROS2 [18]. Como se menciona en dicha web, la wiki de ROS2 se encuentra en fase Beta, por lo que pueden aparecer ciertos errores que no hayan sido corregidos aún.

ROS2 se puede instalar en cualquiera de sus versiones. Las más recientes y que a día de hoy tienen soporte, son *ROS2 Dashing Diademata* y *ROS2 Eloquent Elusor*. Cualquiera de las dos debería ser equivalente, aunque estando la primera lanzada en mayo de 2019 y la segunda en noviembre de 2019, la primera se encuentra más completa a nivel de documentación, soporte y librerías disponibles.

Lo siguiente es realizar los tutoriales de la propia wiki de ROS2 para Windows [18]. Dichos tutoriales están planteados, hasta ahora, para ROS2 Eloquent Elusor. Estos resultan ser una guía básica y completa en el aprendizaje de ROS2.

Sin embargo, ya en tutoriales que tratan funcionalidades básicas de *ROS/ROS2* como *rqt*, el sistema comienza a dar errores. La investigación de dichos errores en internet deriva en fuentes oficiales que indican una migración incompleta a Windows por parte



de los desarrolladores y por tanto en algunos casos no podrán solucionarse hasta dentro de algún tiempo.

Algunos de dichos errores pueden llegar a solucionarse mediante variables de entorno y otras carpetas del sistema de Windows que sin un conocimiento relativamente avanzado de informática no resulta conveniente modificar.

Estos sucesos llevaron a un bloqueo en la implantación y aprendizaje de ROS2 en Windows que llevaron a la toma de una nueva decisión: ROS2 en Linux o ROS en Linux.

3.5.3.3 ELECCIÓN DE ROS2 EN UBUNTU

Ante la nueva decisión planteada, volvimos a evaluar las diferencias entre ambos sistemas. Tras varios tutoriales de ROS2, el conocimiento inicial de cómo funcionaba el sistema parecía indicar que su implantación en un sistema operativo no conocido (Ubuntu) sería más sencilla que la implantación de un sistema como ROS desconocido por completo. Tampoco existía certeza de que dichos errores que habían bloqueado el desarrollo no se fueran a reproducir en Ubuntu. No obstante, tras una búsqueda exhaustiva en internet acerca de aquellos errores, dichas fuentes (foro oficial de ROS2 [19]) indicaban que dichos fallos eran un problema de Windows.

Por tanto, la decisión fue comenzar la instalación e implementación de ROS2 en Linux. La instalación de Linux se llevó a cabo a través de una máquina virtual, en concreto Virtual Box [20]. Siendo los algoritmos en tiempo real muy demandantes de recursos, conviene establecer los parámetros de forma que dicha máquina virtual tenga a su disposición la mayor parte de los recursos disponibles en el sistema.

Tras la instalación de Linux, procedimos a la instalación de ROS2 y al aprendizaje en base a los tutoriales que se encuentran en su Wikipedia [18]. Esta vez, los errores surgidos eran de una magnitud inferior y podían resolverse tras búsquedas en internet.

En los tutoriales de ROS2, para la aplicación de detección en Lidar, debe ponerse especial énfasis en la creación de un **publisher** y un **subscriber**. Los tutoriales de la plataforma alcanzan por el momento un nivel básico y como veremos más adelante la creación de un **publisher** y un **subscriber** personalizado necesita de herramientas adicionales.

3.5.3.4 IMPLANTACIÓN DE LOS ALGORITMOS DE DETECCIÓN EN ROS2



Una vez instalada la máquina virtual (Virtual Box), el sistema operativo Ubuntu dentro de esta y ROS2 (versión *dashing*, más completa por el momento y con soporte vigente), el sistema está listo para comenzar a implantar los algoritmos.

El *workspace* listo para ejecutar se encontrará adjunto junto con este documento y permitirá agilizar el proceso, su nombre es Lidar2, en referencia a un *workspace* para trabajar con detección en Lidar en la versión 2 de ROS.

A continuación, se presentan las etapas que conducen a la creación de los diferentes paquetes (*packages*) y nodos (*nodes*) que se comunicarán en el sistema, permitiendo ejecutar una comunicación en tiempo real.

3.5.3.5 ANÁLISIS DE LOS PAQUETES PRESENTES

En el *workspace* Lidar2, encontramos fundamentalmente dos *packages*.

3.5.3.5.1 VELODYNE

Velodyne consiste en un *package* preestablecido, descargado de internet [21] y consiste fundamentalmente en un *driver* para poder conectar el Lidar a los nodos que crearemos en el otro *package* (*py_pubsub*). Dicho *driver* permite conectar el Lidar y extraer los datos en tiempo real, transformándolos a formato *PointCloud2*, formato muy común en el tratado de nubes de puntos y con el cuál trabajaremos en los algoritmos de detección con Lidar.

Este paquete se encuentra compuesto de los siguientes nodos:

3.5.3.5.1.1 Velodyne_driver

Se encarga de recoger los datos del Velodyne (en nuestro caso, el VLP16 de 16 capas) y combinarlos en un mensaje por revolución. El mensaje de salida será de la forma */velodyne_paquets* (*/velodyne_msgs/VelodyneScan*), datos sin transformar que serán tratados por otros nodos de este mismo paquete.

Si en lugar de contar con el dispositivo Lidar en dicho momento se cuenta con un archivo pcap de una grabación precedente (dicho formato se puede generar por ejemplo grabando con el VLP16 a través del software *Veloview* [22]), también podrá ser utilizado con este



driver, cambiando el parámetro *pcap* del mismo y estableciendo la ruta a la grabación en cuestión. Dicha herramienta fue utilizada en una primera instancia para poder testear los algoritmos en diferido al no tener acceso a pruebas reales del Lidar por una situación de confinamiento.

Para cambiar dicho parámetro debemos acceder a la ruta `~/velodyne_driver/config/VLP16-velodyne_driver-node-params.yaml` y añadir la ruta al archivo tras la línea que dice *pcap*:

También podemos cambiar este parámetro en el *node* (archivo *driver.cpp*) donde tomará este archivo por defecto al modificar el parámetro correspondiente.

3.5.3.5.1.1 Velodyne_pointcloud

Este nodo se suscribe al mensaje publicado por *Velodyne_driver* y convierte los datos en formato *raw de Lidar* en formato **pointcloud2**, publicando un nuevo mensaje de tipo *sensor_msgs/PointCloud2* en el *topic velodyne_points*. Este será el *topic* al que suscribiremos el primer algoritmo del pipeline de detección para comenzar a tratar la nube de puntos.

3.5.3.5.1.1 Velodyne_msgs

Este último *nodo* no lo utilizamos de momento en la detección. Permite separar los mensajes de *velodyne_points* en *n* mensajes, siendo *n* el número de capas del Velodyne, en nuestro caso 16.

Por último, cabe destacar que se creó un archivo de tipo **launch** para el paquete Velodyne (el cuál se encuentra junto con el resto de los archivos tipo **launch** del package), titulado *velodyne-all-nodes-VLP16-launch-custom.py* y que permite lanzar todos los nodos del paquete *Velodyne* con la configuración que hemos establecido previamente.

3.5.3.5.2 PY_PUBSUB

Este paquete ha sido creado expresamente para incorporar todos los nodos que componen el pipeline de detección con Lidar. Su nombre hace referencia a algo que no resulta muy evidente realizar con Python en ROS2 en una primera instancia y que desarrollaremos a



continuación: crear un nodo que esté suscrito a un *topic* y tras un proceso publique mensajes en otro *topic* diferente.

Para desarrollar estos nodos se utilizó como base los ejemplos básicos de publicador-suscrito en el repositorio GitHub oficial de ROS2 [23]. De entre las opciones propuestas, se tomó la versión *~local_function* para ambos casos (*publisher-suscriber*), siendo esta la que permitirá combinar ambos códigos y conseguir un nodo que esté suscrito y publique al mismo tiempo.

Los códigos que se encuentran dentro de este paquete y su funcionamiento correspondiente son:

3.5.3.5.2.1 DelphiMejorado

Este código, explicado en la sección de *Ground Filtering*, resulta de apoyo a modo de librería para realizar el filtrado de suelo.

3.5.3.5.2.2 GroundPF

De forma análoga a *DelphiMejorado*, este código define una clase *GPFV* que permitirá al nodo que viene a continuación realizar el filtrado de suelo. Este código se encuentra desarrollado de igual forma en la sección de *Ground Filtering*.

3.5.3.5.2.3 Ground_pubsub

Este código corresponde al nodo encargado de realizar el filtrado de suelo. Para ello, sigue el siguiente proceso.

En primer lugar, define una variable global denominada *PointCloud2ToPub*, utilizada a lo largo del código para modificar el mensaje que será publicado al final de este.

A continuación, se define la función *callback(msg)* la cuál será llamada cada vez que el nodo lea un mensaje. En esta función, se transforman los puntos en formato PointCloud2 a una lista que contiene la nube completa. Esto se puede realizar gracias al código *point_cloud.py* el cuál será explicado a continuación.

Una vez obtenida la lista de puntos, se trata con los algoritmos de filtrado de suelo para luego transformarse a formato *PointCloud2*, una vez más con apoyo del código *point_cloud.py*.



A continuación, encontramos la suscripción al *topic velodyne_points* y, tras esta, la función ***timer_callback***. Esta última función será llamada por el nodo de forma periódica (con periodo definido por *timer_period*, que resultará ser la frecuencia definida por el programador para la publicación de los mensajes), publicando un mensaje en el *topic nonGround_points*, donde se encontrarán los puntos que no pertenecen al suelo.

NOTA: como margen de mejora, se podría definir el ***timer_period*** de forma que no publique tras un cierto tiempo sino cada vez que tenga un mensaje que publicar. Por el momento, dicho tiempo corresponde a una frecuencia cuyo valor no ha sido todavía establecido (hablaremos de dicha frecuencia en los resultados de la implementación en ROS2), siendo la frecuencia del Lidar de 10hz, frecuencia objetivo si el algoritmo tiene la velocidad suficiente..

3.5.3.5.2.4 Point_cloud

Este código procede de la librería contenida en la documentación de ROS [24]. Este archivo contiene funciones de gran utilidad para transformar las nubes de puntos en formato ***PointCloud2*** a listas de puntos con el formato *[[Xm, Yn, Zn, In, Rn], ...]*, siendo las tres primeras variables las coordenadas de los puntos, la cuarta la capa a la que corresponde el punto (de entre las 16 del Lidar, en nuestro caso) y por último el quinto la reflectividad devuelta tras el impacto. Para hacer funcionar dicho código con ROS2, en las funciones de leer nubes de puntos (*read_points*, etc.) se comentó la sentencia que comenzaba por “***assert isinstance...***”. La cuál estaba pensada para devolver un error en caso de introducir un formato diferente de ***Pointcloud2*** en ***ROS*** y que no necesitamos para nuestro caso en el que aseguramos una entrada del formato ***PointCloud2*** y en ROS2.

3.5.3.5.2.5 Clustering_pubsub

Este código se estructura de forma similar a ***ground_pubsub***, siendo el responsable de realizar la etapa de ***clustering***. La metodología que describe el funcionamiento del algoritmo se explica en la sección de ***clustering***. Por tanto, este código se suscribe al *topic nonGround_points*, publicado por ***Ground_pubsub***, transforma el formato ***PointCloud2*** en una lista de puntos como la especificada en la explicación de ***point_cloud***, aplica el algoritmo desarrollado en Python y vuelve a transformar la lista a formato ***PointCloud2***, publicando los clusters a la vez en el *topic clustered_points*. El objetivo de este nodo es comprobar si se realiza correctamente la función de clustering.



Por ello, antes de publicar la nube de puntos, se cambia la reflectividad de los puntos pertenecientes a cada cluster a un valor igual para los puntos pertenecientes al mismo cluster y diferentes entre clusters. De esta forma, visualmente se podrá comprobar si el algoritmo de segmentación no supervisada funciona con precisión.

3.5.3.5.2.6 Enclose_circle

Este código corresponde a una librería extraída de internet y explicada en la sección **Reconstruction**. Será empleada en el nodo ***pipeline***, explicado a continuación.

3.5.3.5.2.7 Pipeline

Este *nodo* es el más importante de todo el *package*. Contiene todos los algoritmos/etapas de detección con Lidar integradas en el mismo *nodo*. Esto permite ahorrar problemas de comunicación entre *nodos* y probar la calidad del sistema completo de forma sencilla. Al integrar todos los algoritmos en el mismo *nodo*, se pierde rendimiento que se ganaría repartiéndolo en diferentes *nodos*, como se ha hecho con ***ground_pubsub*** o ***clustering_pubsub***.

Este código, se suscribe al *topic* ***velodyne_points*** al igual que lo hacía ***ground_pubsub*** y transforma la nube de puntos mediante la secuencia de fases en serie:

Filtrado de suelo > Clustering > Reconstruction > Cone Detection

Como salida publica en el *topic* ***clustersWithGround*** una nube de puntos con lo que debería corresponder únicamente a los conos sobre la pista.

3.5.3.6 DATASETS REALES CON ROS2

Con el objetivo de testear los algoritmos en una base de datos equivalente a lo que correspondería al día de la competición, a parte del *package* ***velodyne*** que nos permitía utilizar los archivos ***pcap*** grabados con nuestro Lidar, se buscaron archivos de tipo ***rosbag*** en internet. Se encontró un repositorio de GitLab [25]. De aquí se pueden descargar incluso *datasets* de la propia competición de FSUK, idénticos a las condiciones el día de la competición.



Estos **datasets** se encuentran en formato ***rosbag***, muy común en ROS y que encuentra su segunda versión en el formato ***rosbag2***, para ROS2. Los archivos de tipo rosbag no se pueden abrir con ROS2 directamente, por lo que hace falta generar un puente ROS1-ROS2 para abrir el archivo en ROS, suscribirse al *topic* publicado por este último con ROS2 y registrar la información en un archivo de tipo ***rosbag2***, con el que se podrá trabajar posteriormente. Para ello se hace uso de la propia documentación de ROS2 [26], donde hace referencia a los requisitos necesarios para llevar el proceso a cabo, con sus referencias para lograr dichos requisitos. Tras esto último, describe el proceso completo hasta lograr el archivo ***rosbag2***.

3.5.4 RESULTADOS

Para evaluar los resultados de la implantación del algoritmo completo de detección con Lidar en ROS2, realizaremos una integración sucesiva de las distintas fases, analizando los resultados que estas nos proporcionan.

De esta forma, se llevarán a cabo 4 pruebas diferentes, según el nivel de integración de las cuatro fases que constituyen por el momento la detección con Lidar:

- Ground Filtering
- Ground Filtering + Clustering
- Ground Filtering + Clustering + Reconstruction
- Ground Filtering + Clustering + Reconstruction + Non-Cone Filtering

Para los tests, se utilizará el archivo ***rosbag2***, (obtenido como explicamos anteriormente), archivo correspondiente a un registro de un archivo de Lidar equivalente al de la competición en FSUK [25]. Este archivo será ejecutado mediante rosbag2 play -r 10, indicando una frecuencia de 10Hz para la publicación de mensajes en el *topic* velodyne_points, misma frecuencia que la del lidar.

Si observamos lo indicado para este archivo [25], fueron tomados en FSUK2018 usando el vehículo ADS-DV proporcionado por IMechE. Se utilizó un Lidar Velodyne VLP-16 a 10Hz de frecuencia tomando datos en un rango de 360 grados. Se indica que tanto los restos de coche como las ruedas deben ser filtradas.

3.5.4.1 GROUND FILTERING

El algoritmo de Ground Filtering, junto con el filtro de altura y distancia, está implementado en ground_pubsub. Se ejecuta el nodo, estableciendo un tiempo de publicación de 5Hz, debido a que un tiempo superior da error en la ejecución por no ser suficiente en varias ocasiones.

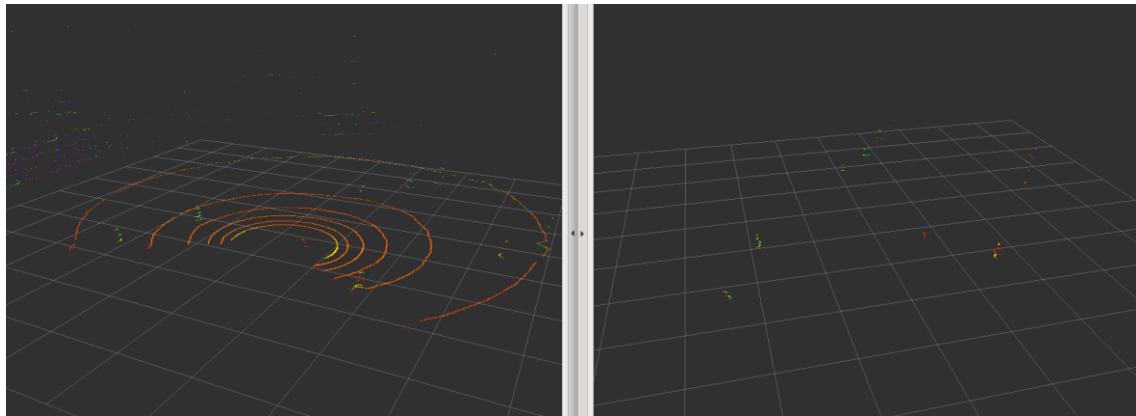


Figura 26: Frame de Ground_pubsub a 5 hz

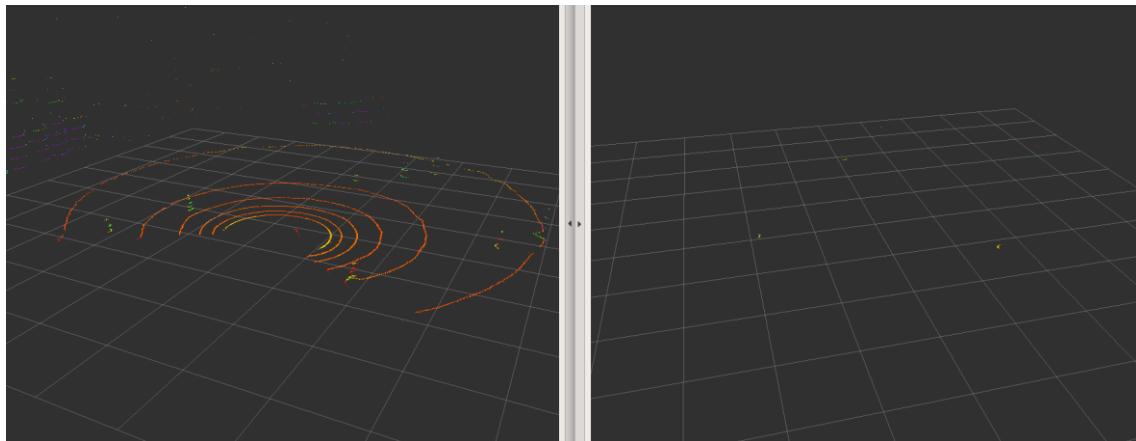


Figura 27: Frame de Ground_pubsub a 5 hz

En las figuras (Figura 26 y Figura 27), tomadas de la ejecución del algoritmo, detectamos que el filtrado de suelo se realiza con gran precisión. Sin embargo, si observamos el algoritmo ejecutarse, podemos apreciar en algún frame uno de los dos siguientes errores:

1. Fracción de suelo que resta sobre la pista, no detectada como suelo.
2. Gran parte de los conos se consumen al filtrar el suelo, como detectamos en la segunda imagen.

Para solventar estos problemas, que pueden proceder de un desajuste de los parámetros del filtrado de suelo, no optimizados para este algoritmo, un ajuste de estos últimos puede resultar en mejores resultados.

Para solventar este problema, se incrementa el parámetro ***n_iter*** a cuatro (cambiando el número de iteraciones a 4 en lugar de 2), y el error de aparición de suelo sobre la pista aún se reproduce en algunas ocasiones. Al ser ocasiones muy puntuales y dar lugar a un cluster de suelo de gran tamaño, este será filtrado por posteriores fases del algoritmo, por lo que es un error solventable.

No merece la pena continuar incrementando el número de iteraciones, puesto que esto realentizaría el algoritmo, sin aportar mejores resultados.

Otra opción que se plantea para solventar el problema es incrementar el parámetro ***th_dist*** que hace referencia al grosor del plano final de 5 cm a 10 cm. Esto solventa el problema, pero incrementa la desaparición de gran parte de información de los conos sobre la pista, algo que no resulta conveniente.

Para corregir estos errores se plantea una tercera opción, que aún no ha sido implementada. Cuando se analizó el algoritmo, se segmentó el espacio en sectores en función del ángulo azimuth. Existía la posibilidad de segmentar también estos sectores en función de la distancia, algo que no se llevó a cabo por que se fiera el caso de no existir suficiente información en un sector para constituir un plano. Resultaría conveniente implementar esta funcionalidad y analizar los resultados.

3.5.4.2 GROUND FILTERING + CLUSTERING

Para testear ambas fases a la vez, tenemos que ejecutar los nodos de ***Ground_pubsub*** y de ***clustering_pubsub*** al mismo tiempo, ambos con frecuencia de 5hz. La frecuencia de ***clustering_pubsub*** no podrá ser menor que la de ***ground_pubsub*** al depender de los datos publicados por este *nodo*.

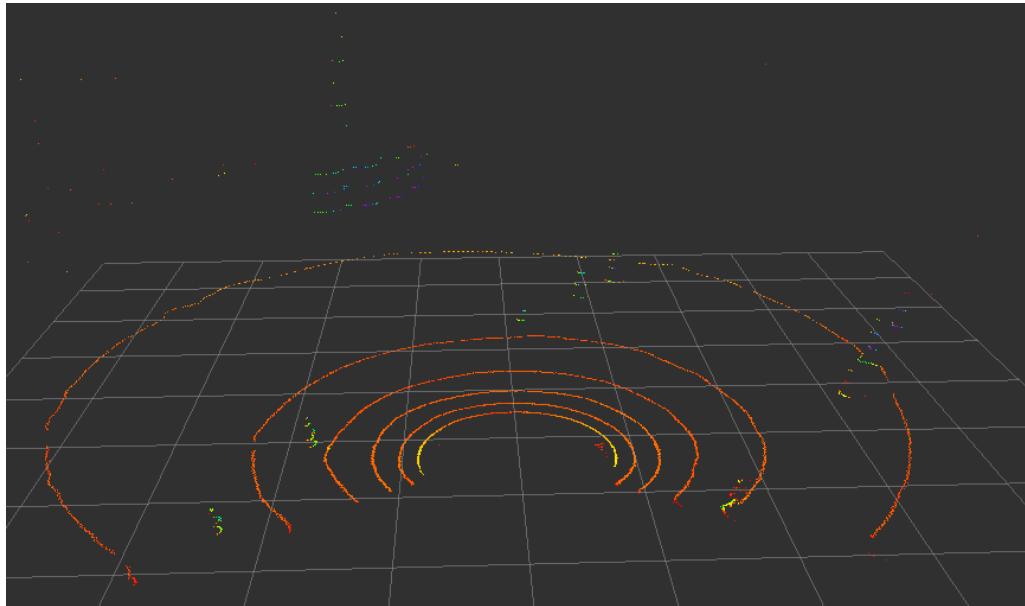


Figura 28: Frame de Ground Filtering + Clustering Dataset

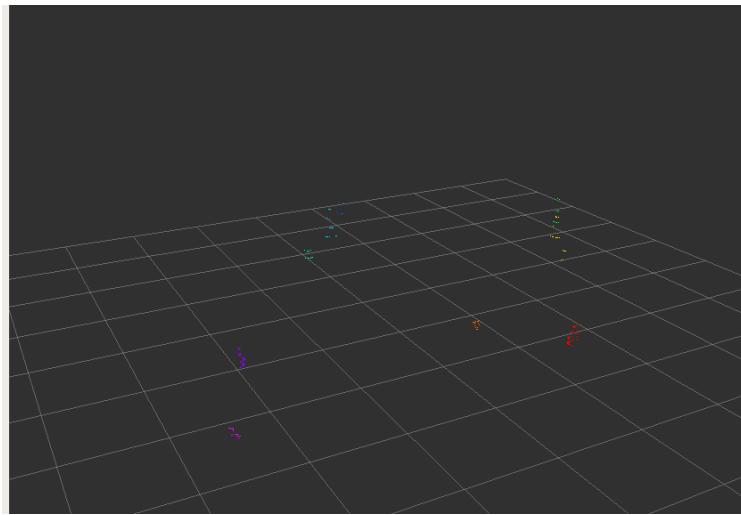


Figura 29: Frame de clusters obtenidos de Ground Filtering + Clustering

En esta ocasión, se han separado las figuras del *dataset* y el resultado de las fases combinadas del algoritmo para una mejor visualización.

Observamos en la Figura 28 la entrada del algoritmo y en la Figura 29 una correcta segmentación de la nube de puntos en sus correspondientes subnubes. Los conos quedan diferenciados en *clusters* diferentes.

Si observamos la ejecución del *dataset* completo, se reproduce el problema anterior de los puntos de suelo sobre la pista, esta vez asociados a 1-3 *clusters* diferentes. El problema sigue sin ser grave al no haber terminado el filtrado completo y reproducirse en situaciones esporádicas y en magnitud reducida (únicamente una pequeña porción de suelo en ciertas ocasiones). Si este persiste al final del algoritmo, deberán modificarse las distintas fases para solventarlo.

3.5.4.3 GROUND FILTERING + CLUSTERING + RECONSTRUCTION

En este punto, se testearán ya tres fases del algoritmo. Al haberse integrado a partir de este punto todas las fases en el código ***pipeline***, se creará una máscara para la última fase (Non-cone filtering) y se visualizarán por tanto las tres primeras con la ejecución de un único nodo.

Se prueba una ejecución del algoritmo a 5hz, acumulando los parámetros definidos anteriormente.

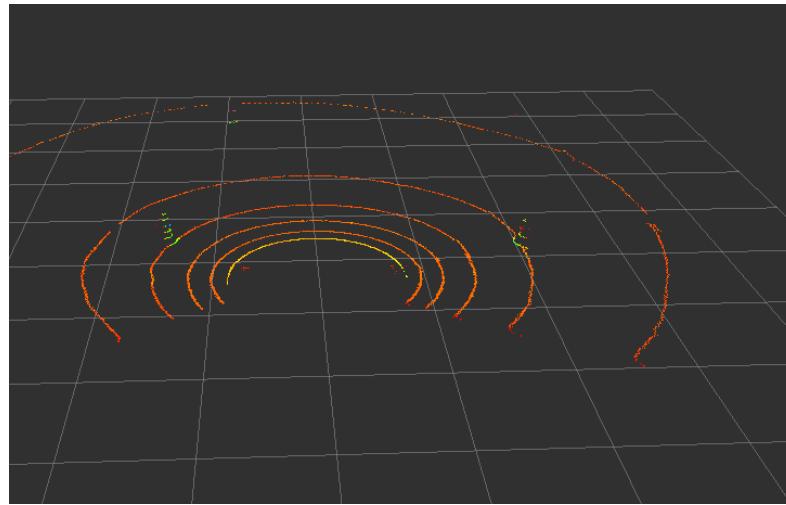


Figura 30: Frame de GroundFiltering + Clustering + Reconstruction Datatest

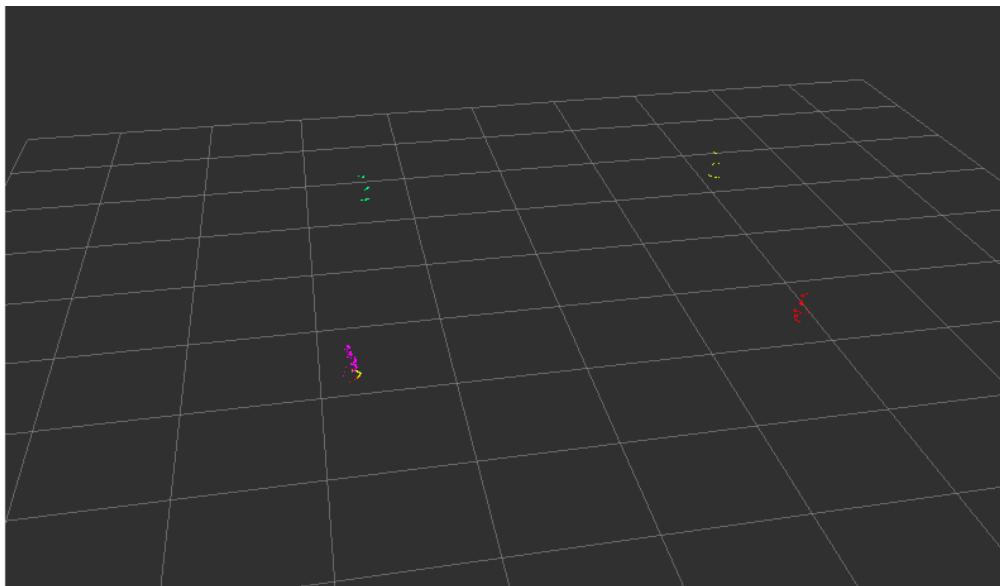


Figura 31: Frame de Ground Filtering + Clustering + Reconstruction

Al ejecutar el algoritmo, observamos en la Figura 30 la entrada del algoritmo y en la Figura 31 una correcta reconstrucción de los conos con la información que se había perdido al eliminar el suelo. Esto es visible fácilmente en el *frame* expuesto, en el cono de la esquina inferior izquierda. Ejecutando el *dataset* completo, se observa que este comportamiento se reproduce de forma correcta durante todo el *dataset*.

3.5.4.4 GROUND FILTERING + CLUSTERING + RECONSTRUCTION + NON-CONE FILTERING

Para esta última fase, se va a ejecutar el nodo pipeline completo, comprendiendo todas las fases y constituyendo el algoritmo de detección con Lidar completo, a falta de incluir una última fase de distinción en la posición de los conos (izquierda/derecha).

El algoritmo se ejecuta a 5Hz, dando problemas a frecuencias superiores, aunque pudiendo realizarse en algunos casos.

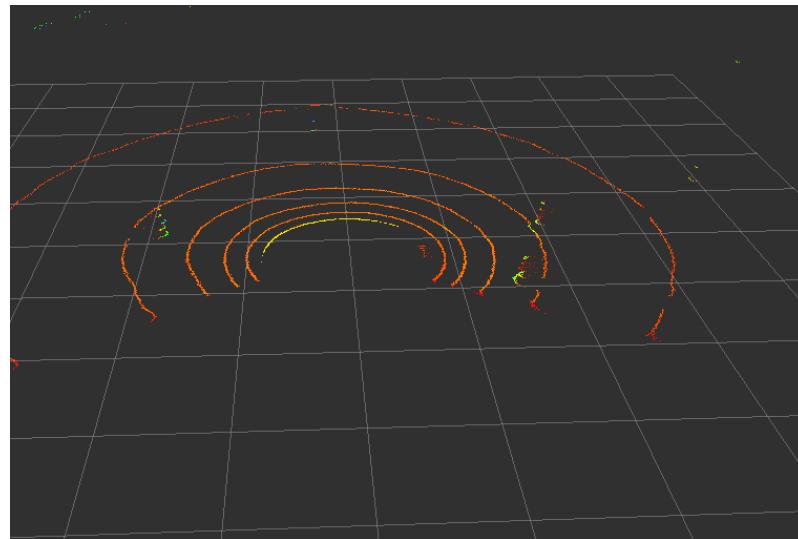


Figura 32: Frame de Ground Filtering + Clustering + Reconstruction + Non-cone filtering dataset

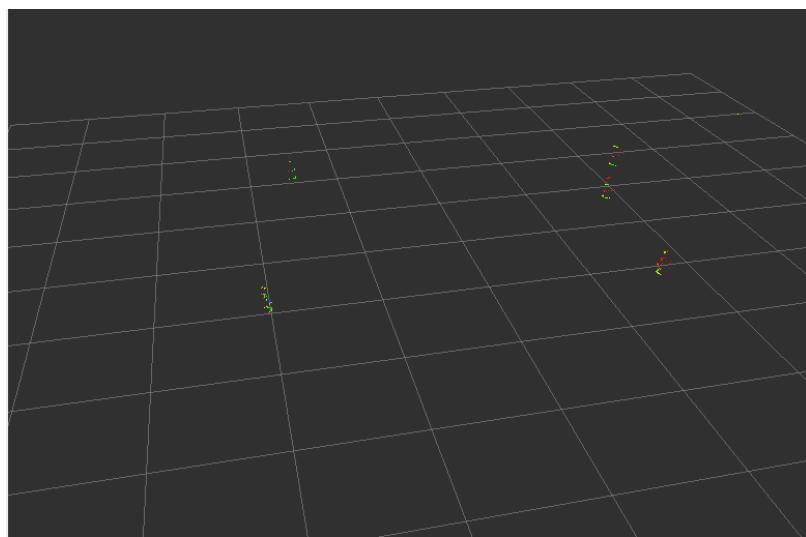


Figura 33: Frame de Ground Filtering + Clustering + Reconstruction + Non-cone filtering



En esta última integración, al ejecutar el algoritmo, observamos en la Figura 32 y en la Figura 33 que lo único que tiene salida de este son conos, solventándose por completo el problema de la aparición de suelo, quedando este filtrado en la última fase.

A continuación, implementamos en el algoritmo una función que nos dirá la máxima distancia a la que reconocemos los conos. La máxima distancia a la que estos son detectados ronda los 6.95m, algo óptimo teniendo en cuenta que el algoritmo filtra la información más lejana de 7m.

Si cambiamos el parámetro del filtro de distancia a 10m (cambiando en *GroundFiltering, d_max = 10*), el algoritmo detecta conos a 9.80 metros del Lidar, por lo que este parámetro tiene gran importancia en la distancia máxima a la que se detectan los conos.

Si en una tercera prueba, establecemos el valor en *d_max=15*, los conos más alejados detectados vuelven a estar a 9 metros, por lo que este valor no aportaría información útil nueva y por el contrario realentiza el algoritmo.

La mayoría de los valores de los conos detectados se encuentra entre 3 y 9 metros, por lo que se registra 10 metros como la nueva distancia máxima de detección del algoritmo.

3.5.4.5 CONCLUSIONES

Tras el análisis de resultados, se extraen conclusiones y puntos de mejora para dar mayor rendimiento y precisión al algoritmo:

- En tanto que una primera versión, el resultado de salida de los conos se encuentra por encima de lo que se podría esperar. Lo único que resta a la salida son conos. Sin embargo, hay conos detectados en un *frame* que no se detectan en el *frame* siguiente, pese a estar a una distancia similar del Lidar. Esto presenta una necesidad de optimización del algoritmo y sus parámetros, para ganar la precisión que le falta.
- Sería conveniente testear el algoritmo en un sistema con altas prestaciones, ya que la máquina virtual no funciona de la misma forma que un sistema con Ubuntu instalado como sistema operativo en disco. Es conveniente por tanto realizar la transición y reproducir los resultados.
- El algoritmo se ejecuta a una frecuencia de 5Hz, la mitad de la frecuencia de publicación del Lidar y frecuencia objetivo del proyecto. Es por tanto necesario optimizar los códigos, eliminando complejidades redundantes, de cara a obtener mejores resultados de precisión.



3.6 FUSION

3.6.1 RESUMEN

En la detección del coche autónomo de *driverless*, existen dos medios de detección: Lidar y Cámara (visión). Ambos son equivalentes, utilizando diferentes tecnologías y se desarrollan en paralelo para lograr una solución más robusta y sólida una vez los datos de ambos dispositivos se hayan fusionado.

Esta sección trata precisamente de la fusión de los datos cámara-Lidar. En concreto, nos centraremos en la fase de fusión desarrollando un algoritmo para el Lidar, quedando a la espera de la otra mitad, realizada por la subdivisión de cámaras y que permitiría el intercambio de información de ambos dispositivos bajo el mismo marco de referencia.

Como en códigos anteriores, comenzaremos por una explicación de los objetivos, para proseguir con un desarrollo, finalizando con los resultados obtenidos y las conclusiones extraídas.

3.6.2 OBJETIVOS

- Desarrollar un algoritmo de detección de ciertos patrones con alta precisión (decenas de milímetros).
- Elegir un método de fusión que convenga a ambas partes, Lidar y cámara.
- Construir un marco de referencia común a ambos algoritmos.

3.6.3 DESARROLLO

3.6.3.1 ELECCIÓN DEL ALGORITMO

Como se encuentra expuesto en los objetivos, el primer reto es encontrar una vía que permita fusionar la información de ambos dispositivos, que seamos capaces e implementarla en el equipo y que de lugar a una precisión de cm.

El objetivo final es construir una referencia común entre los datos registrados por el Lidar y la cámara. Para ello podríamos tomar medidas de la traslación y rotación de ambos dispositivos, uno respecto del otro, al situarlos sobre el monoplaza. Sin embargo, esto no siempre resulta sencillo y puede ser fuente de múltiples imprecisiones. Para evitarlo, desarrollaremos un algoritmo que permita realizar este proceso de forma automática, encontrando esas coordenadas de traslación y rotación y generando un marco de referencia común a ambos dispositivos.



Existen múltiples vías para realizar una fusión de cámara-Lidar, destacando entre ellas las siguientes, según fuentes de prestigio que han desarrollado algoritmos equivalentes:

- Calibración automática extrínseca (3D-3D) [7]: en este artículo se propone un método de calibración extrínseca, tomando un marco de referencia compuesto por un rectángulo de madera/plástico al que se le realizan cuatro incisiones circulares. El objetivo trata de detectar tanto en el Lidar como en la cámara el centro de dichos círculos, y construir a partir de ahí el marco de referencia.
- Calibración online automática [27]: en este artículo se realiza un análisis de la metodología disponible para realizar la fusión Lidar-Cámara. Tras ello, proponen un método de calibrado automático en cualquier escenario, sin necesidad de objetos patrón, de ahí la referencia al término online. Afirman que el proceso toma varios minutos hasta completar la calibración. La ventaja de este algoritmo es su capacidad de detectar pequeñas rotaciones o traslaciones en la colocación de los sensores y corregir los parámetros de calibración a tiempo real.
- Calibración extrínseca para radar, cámara y Lidar [28]: en este artículo se propone un método similar al que encontrábamos en el primer informe, mediante un marco de referencia, posibilitando además la integración de un radar a dicha calibración, algo no realizado en el UpmRacing. Parte de la misma base por tanto que el primer algoritmo, siendo este más completo al permitir integrar más sensores.
- Calibración Lidar-cámara usando correspondencias 3D-3D entre puntos [29]: en este artículo, se propone una vez más una calibración extrínseca, utilizando distintos marcos de referencia, que deben ser colgados con el objetivo de detectar sus aristas y vértices. Además, propone la utilización de ArUco markers, un tipo de marco similar a lo que puede ser un código Qr.

Además de estos artículos, que sirvieron como una referencia para conocer algunos de los métodos más utilizados en el sector, existen muchos otros, referenciados en gran parte en estos mismos.

Tras un análisis, con el objetivo de realizar una primera versión funcional y explicada lo mejor posible en el propio artículo para una implantación que diera una base para la fusión cámara-Lidar, se escogió la Calibración automática extrínseca (3D-3D) [7], desarrollada en la Universidad Carlos III de Madrid. La calibración online automática se descartó por su complejidad para una primera versión y esta era la calibración extrínseca que se adaptaba de mejor manera a nuestro caso. El marco de referencia no necesita estar



en una posición elevada y con una primera toma de datos ya se pueden obtener resultados precisos.

3.6.3.2 INTRODUCCIÓN A LA METODOLOGÍA

Una vez elegida la metodología, comienza el desarrollo e implementación del algoritmo, en base a lo expuesto en el artículo [7]. La trasformación se encuentra definida por un conjunto de 6 parámetros:

$$\text{Referencia} = (t_x, t_y, t_z, \phi, \Theta, \psi)$$

Siendo t_x, t_y, t_z las traslaciones respecto de los tres ejes (x, y, z) y ϕ, Θ, ψ correspondientes a los ángulos de giro respecto de los tres ejes. La transformación se puede llevar a cabo respecto a la referencia de la cámara o del Lidar, de forma reversible. Como comentábamos, en esta sección nos centraremos en la realización de la mitad de la fusión, siendo esta únicamente la parte correspondiente al Lidar.

Para la calibración, se utiliza un único marco de calibración, el cuál debe ser percibido de forma constante por ambos sensores durante el proceso de calibración, sin cambios en la posición de este o de los sensores.

El marco consiste en una plataforma rectangular con cuatro círculos perforados, distribuidos de forma simétrica. Estos círculos actúan como diferentes figuras visibles a la vez por la cámara estéreo y el Lidar, siendo necesario por tanto que ambos sensores tengan una visión completa del marco de referencia durante la calibración. Además, en el caso de Lidar se especifica que al menos dos rayos intercepten en cada círculo, para una precisión correcta.



Figura 34: Ejemplo de marco propuesto



Más allá de estos requisitos, no hay suposiciones adicionales acerca de la posición de los sensores o del marco, no necesitando estar este último alineado con ningún eje.

NOTA: se recomienda que el marco tenga textura para facilitar el proceso de reconocimiento de la cámara estéreo.

La calibración se divide en dos fases: una segmentación o reconocimiento del marco de referencia en ambos sensores y un posterior registro para estimar la transformación entre ambas referencias. En el caso de esta sección, únicamente se desarrollará la primera fase y haciendo referencia únicamente al Lidar.

3.6.3.3 DESARROLLO DEL ALGORITMO

El algoritmo se dividirá en cuatro fases, que serán explicadas a continuación. El algoritmo se encuentra repartido en cinco códigos:

1. FiltradoInicial: correspondiente a la primera fase.
2. SegundoFiltrado: correspondiente a la segunda fase.
3. TercerFiltrado: correspondiente a la tercera fase.
4. CuartoPasoCentroides: correspondiente a la cuarta fase del algoritmo.
5. Transformación: código que hace referencia a los cuatro anteriores e integra el proceso completo de fusión.

3.6.3.3.1 BUSQUEDA DEL PLANO

Los primeros pasos corresponden a extraer los puntos pertenecientes a discontinuidades en el marco de referencia. Para ello, se filtrarán lo máximo posible los puntos registrados por el Lidar y se buscará en estos una forma geométrica plana. El realizar un marco plano nos beneficiará en esta etapa para lograr encontrar una referencia plana, algo que no resulta difícil con Lidar y que se encuentra ampliamente documentado en internet. Se deberá aplicar esta medida de forma que no afecten a los resultados otras superficies planas como pueden ser muros o paredes.

Lo primero que realiza el código Transformación es importar el archivo CSV correspondiente a los datos registrados con *Velodyne* a través de *Veloview*. Estos datos serán un registro del Lidar con el marco de referencia situado a una distancia que será necesario medir de forma aproximativa.

Una vez importado, lo primero que realiza el algoritmo es un filtrado del suelo, para evitar que este sea reconocido como plano, en base a lo expuesto en la sección **GroundFiltering**, cuyos códigos son importados.



Tras ello, se ejecuta el primer filtrado, llamando al código ***FiltradoInicial***. Se llama en concreto a la función ***initial_filter*** de este último, que toma como parámetros:

- Distancia a la que se encuentra el marco de referencia (*distBoard*) con una precisión de medio metro necesaria.
- Los puntos recogidos en la medida del Lidar, para ser filtrados.

Lo primero que hace el código es por tanto eliminar mediante un filtro de distancia todos aquellos puntos que se encuentren fuera de un rango de un metro del valor ***distBoard*** especificado, de la forma:

$$\text{Puntos Conservados} = P(r) \in \text{distBoard} \pm 0.5m$$

Tras esta fase, los únicos puntos restantes deberían ser prácticamente los pertenecientes al marco, no debiendo existir otros planos, al realizarse la prueba en un espacio abierto y sin muros o paredes. Lo siguiente es encontrar el plano contenido en los puntos que resulta, mediante la metodología del algoritmo RANSAC [30]. Este algoritmo es ampliamente utilizado y se especializa en detectar planos en una nube de puntos. Es capaz de encontrar planos en nubes de puntos siendo capaz de descartar aquellos puntos ruido que interceden en la correcta determinación del plano.

Este algoritmo, a partir de unos puntos semilla de la nube de puntos, genera un primer plano. Acto seguido, compara el resto de los puntos de la imagen mediante un threshold y calcula un nuevo plano, que será guardado si el último resultado es mejor que el anterior.

La sección del código que implementa el algoritmo RANSAC en base a los puntos proporcionados es extraída de un repositorio de GitHub [31] y se adapta tomando únicamente las funciones necesarias (***fit_plane_LSE_RANSAC*** principalmente). Los parámetros propuestos se conservan a la espera del resultado que proporcionan.

Esta función proporciona:

- El plano obtenido, representado por sus parámetros mediante un array de la forma [a,b,c,d] de $ax+by+cz+d=0$
- La lista de puntos que entran dentro del umbral especificado en el parámetro ***inlier_thresh***.

Con esta fase realizada, pasaríamos a la segunda, debiendo haber obtenido un resultado como el de la Figura 35, representada en el artículo base de este algoritmo [7].

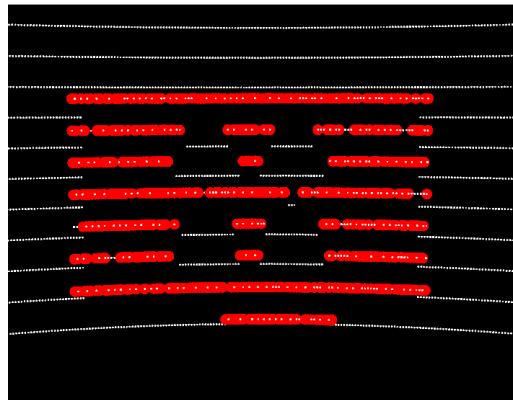


Figura 35: Resultado referencia de la primera fase de la calibración [7]

NOTA: se recomienda que el umbral (`inlier_thresh`) introducido en el algoritmo RANSAC tenga un valor muy pequeño ya que el marco compone un plano fino y esto aumentará en principio el rendimiento del algoritmo para descartar el resto de los puntos que no pertenecen a este marco.

3.6.3.3.2 BÚSQUEDA DE DISCONTINUIDADES

Una vez hemos filtrado todos aquellos puntos que no pertenezcan al marco, el siguiente paso consiste en filtrar todos aquellos puntos que no pertenezcan a discontinuidades. Este paso se encuentra matemáticamente descrito en el artículo de referencia [7]. Para implementarlo, hacemos uso del segundo código, llamado: **segundoFiltrado**.

Con la salida de los puntos de la primera fase, mediante la función `classify_layer`, la cuál clasifica los puntos según la capa a la que pertenezcan (de las 16 que componen el VLP16) en orden ascendente, y dentro de cada capa ordena los puntos según su ángulo azimuth en orden ascendente.

La siguiente función llamada de **segundoFiltrado** es `depth_discontinuity`, la cuál filtra todos aquellos puntos cuyo punto inmediatamente a la izquierda y a la derecha no están a más distancia del Lidar que dicho punto. Esta distancia se parametriza con el valor `th_discont`, que permite ajustarse en el algoritmo e indica el margen de diferencia distancia tolerable para dichos puntos y que siga considerándose que no hay discontinuidad, eliminándose.

Además, para evitar la posibilidad de que puntos muy cercanos se hayan detectado con profundidades diferentes y siendo discontinuos, se pasa el filtro de la función `azimuth_discontinuity` que filtra aquellos puntos que tengan puntos cercanos a menos

de un valor de azimuth determinado por *th_azimuth*, parámetro de la función. Esta función también filtra los bordes, al establecer que puntos contiguos con una diferencia de más de 300 grados se filtren, como es el caso de los bordes.

Tras esta etapa, el resultado debería ser similar al de la Figura 36.

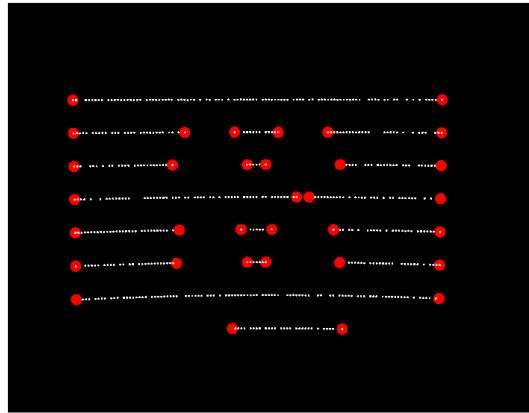


Figura 36: Resultado esperado tras eliminar discontinuidades [7]

3.6.3.3.3 SEGMENTACIÓN DE CÍRCULOS

Llegaríamos a la tercera fase del algoritmo, donde el objetivo es que únicamente resulten los cuatro círculos del marco de calibración, cuyos centros serán usados como puntos clave para la matriz de transformación. Para ello únicamente se conservan las capas con un número compatible de puntos para que tengan un círculo interceptado.

Para realizar esta etapa nos servimos del código *TercerFiltrado*. Este contiene la función *circle_points*, a la que entran los puntos filtrados de la segunda fase.

Esta función conserva únicamente aquellas capas que contienen más de dos puntos, ya que como vemos en la anterior figura, aquellas que contengan un círculo tendrán 3-4 puntos y las que no lo contengan tendrán 2. Se comprueba que el número de puntos de las capas que dicen tener un círculo multiplicadas por cuatro sea inferior al número de puntos que son considerados puntos de círculos porque si no no habría suficientes puntos para realizar la última etapa.

El resultado debe ser equivalente al de la Figura 37.

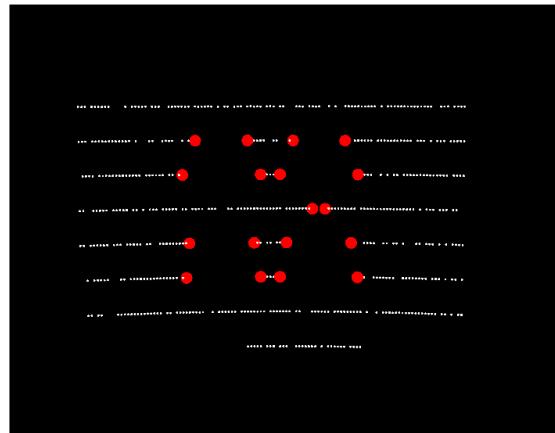


Figura 37: Resultado esperado de la tercera etapa del algoritmo de fusión [7]

3.6.3.3.4 OBTENCIÓN DE LOS CENTROS

Llegados a la última etapa, queda obtener los centros de los cuatro círculos del marco. Para dividir la nube de puntos en cuatro subnubes compuestas por cada uno de los círculos. Para ello, se utiliza el código **CuartoPasoCentroides**. Este código hace uso del algoritmo de aprendizaje no supervisado KMEANS. Este algoritmo se expone en la sección de *Clustering* y resulta conveniente en esta etapa al conocer el número de clusters en los que queremos dividir la nube y ser de gran rapidez y precisión.

Para ejecutarlo, volvemos a hacer uso de la librería sklearn [15], la cual nos da también los centros de los círculos una vez ha distinguido cuatro *clusters* en la nube de puntos.

El resultado del algoritmo debería ser equivalente al de la Figura 38, lo cual será comprobado a continuación en resultados.

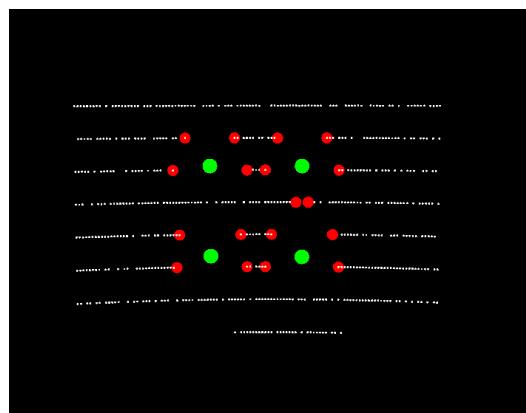


Figura 38: Resultado de aplicar la metodología de calibración en Lidar [7]



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



3.6.4 RESULTADOS

Para evaluar y calibrar el algoritmo, se realizó un marco de cartón en el INSIA. Se hizo de cartón para constituir una primera versión y dar un indicativo acerca del rendimiento del algoritmo. Los resultados pueden observarse en la Figura 39.



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



Figura 39: Marco de cartón para test de calibración

Tras esto, se realizó una prueba con el Lidar en la que se situó el marco a dos metros y medio de distancia, en posición vertical, a la misma altura que el Lidar. No pudieron tomarse medidas precisas de la colocación de ambos dispositivos para una posterior cuantificación milimétrica debido a unas condiciones temporales que lo impedían.

Una vez tomados los datos con el Lidar, podemos visualizar su salida en *Python*.

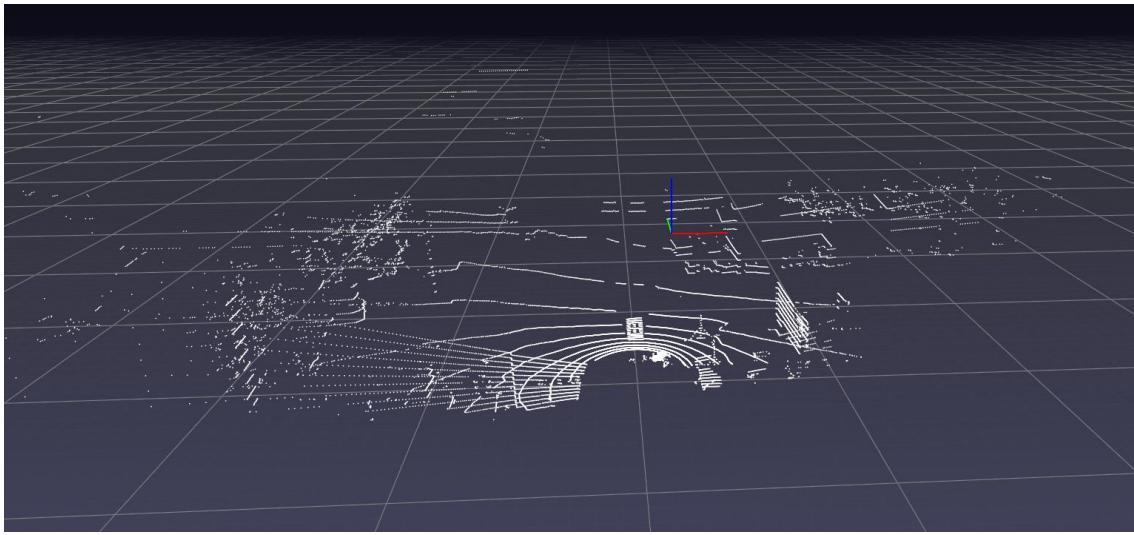


Figura 40: Toma de datos con Lidar algoritmo calibración

Si hacemos zoom en la Figura 40, podemos observar en la Figura 41 con mayor detenimiento el marco de cartón utilizado para la calibración, dispuesto en forma vertical.

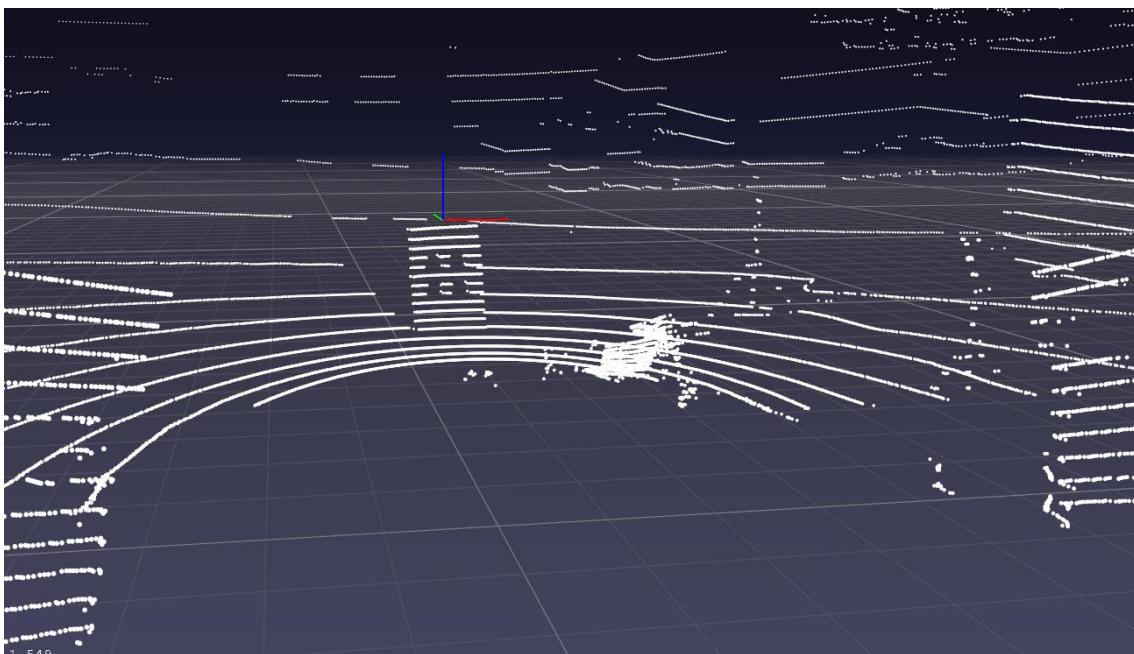


Figura 41: Marco para la calibración visualizado en Lidar

Sobre esta base, vamos a ir ejecutando las diferentes fases el algoritmo.



3.6.4.1 BÚSQUEDA DEL PLANO

Lo primero que ejecuta el algoritmo es el filtrado de suelo, distancia y altura. El filtro de altura se sitúa en 4 metros con el parámetro `th_z`, y el filtro de distancia en 7metros, valor por defecto del código GroundPF y útil en este caso, donde el marco se sitúa a 2.5 metros del Lidar.

La salida de este algoritmo se puede observar en la Figura 42:

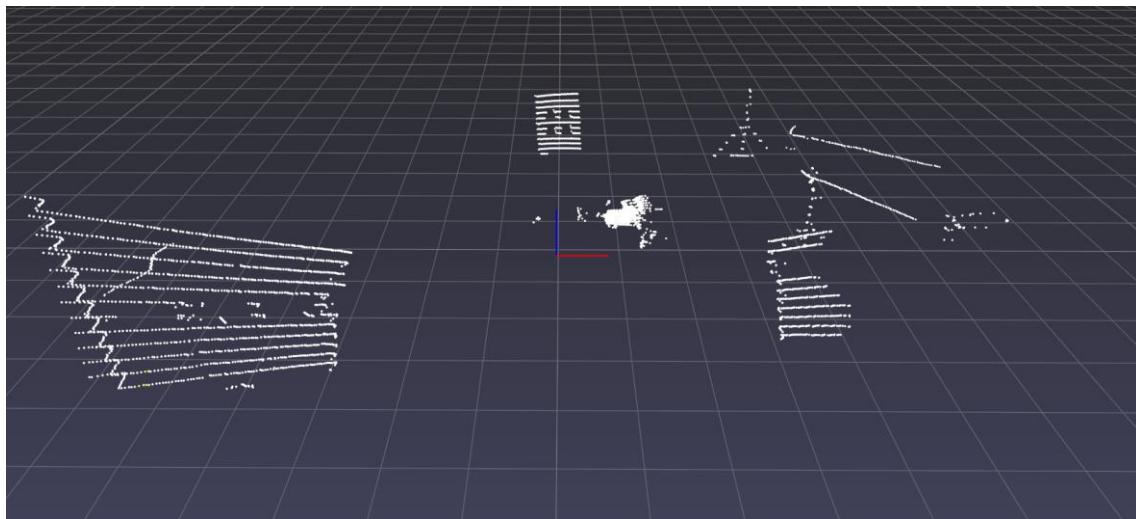


Figura 42: Salida de GroundPF en la primera fase de calibración

El número de puntos se ha reducido de 18787 a 4500, filtrando por tanto un 76% de los puntos originales.

Sobre esta base ejecutamos la primera fase tal y como se indicaba en el desarrollo, con la ayuda del código **FiltradoInicial**. Los resultados, que pueden observarse en la Figura 43, corresponden a 731 puntos, todos ellos del marco de calibración, habiéndose conseguido filtrar el resto de información.

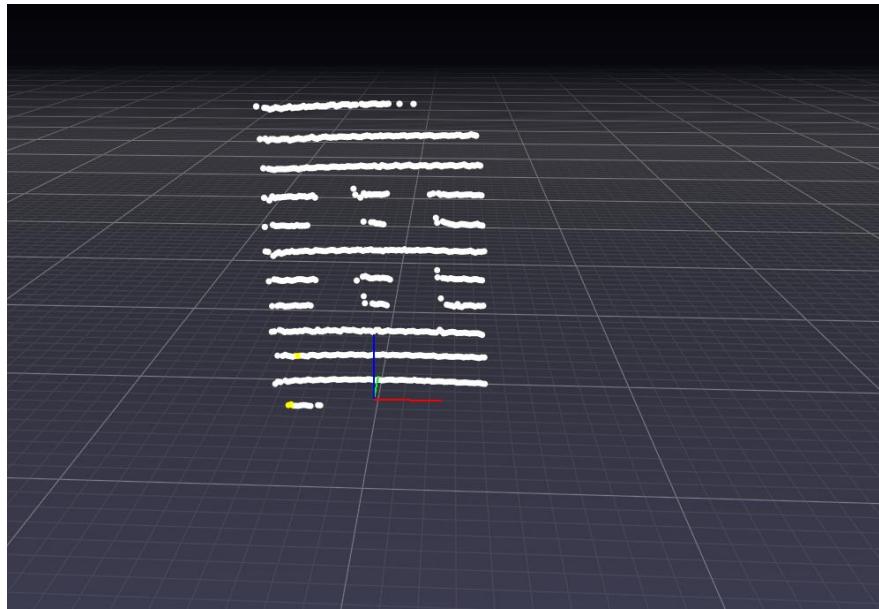


Figura 43: Salida de la primera fase del algoritmo de Calibración

En esta fase se han filtrado por tanto un 96% de los puntos que existían originalmente en la nube de puntos registrada por el Lidar.

3.6.4.1.1 BÚSQUEDA DE DISCONTINUIDADES

En esta segunda fase, ejecutada con ayuda del código **SegundoFiltrado**, se aplica en primer lugar el filtro por profundidad, representando lo obtenido en la Figura 44,

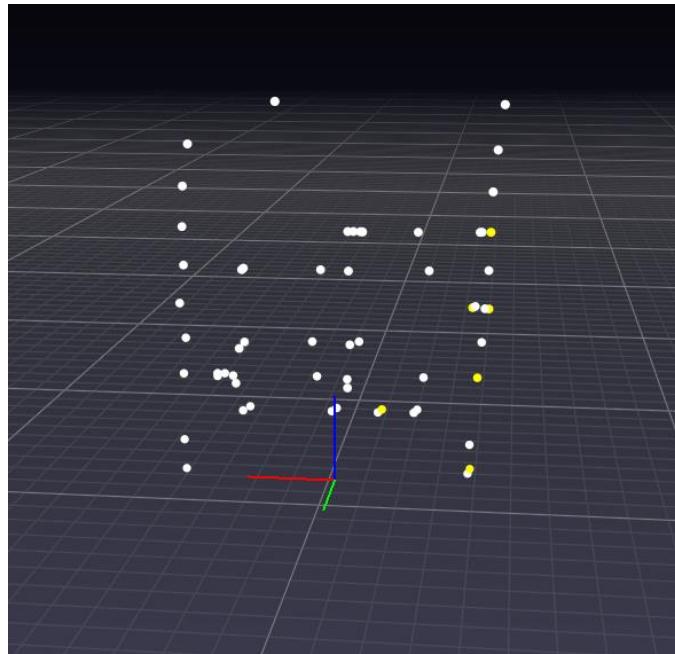


Figura 44: Salida del filtro de profundidad de la segunda fase en calibración

La salida es de 58 puntos y detectamos el problema por el que en el desarrollo se explica la aplicación de un filtro por azimuth. Existen puntos del marco que han sido detectados a profundidades diferentes pese a pertenecer al mismo marco. Eso se puede deber a que las pruebas fueron tomadas con malas condiciones climatológicas y se pudo no cumplir la condición de no movimiento del lidar ni del marco durante la calibración.

Para solventar este problema, se aplica el filtro de azimuth, tal y como se explica en el desarrollo, obteniendo lo nuevamente representado en la Figura 45: 19 puntos, donde apenas restan los correspondientes a bordes del marco y bordes de los círculos. Esta corresponde a la salida de la segunda fase de filtrado, donde se han filtrado un 97,4% de los puntos de entrada, que eran 731.

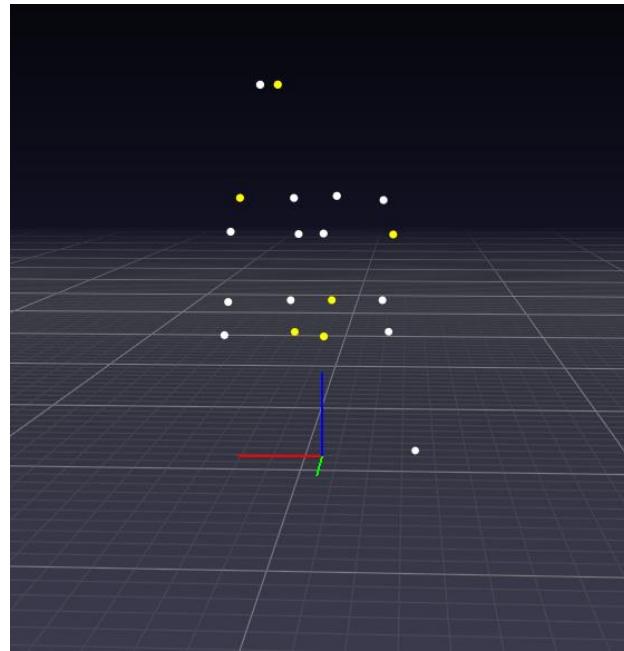


Figura 45: Salida de la segunda fase de calibración

3.6.4.1.2 SEGMENTACIÓN DE CÍRCULOS

La tercera etapa se ejecuta con la ayuda de código **TercerFiltrado**, según lo indicado en desarrollo. La salida de esta tercera fase corresponde a lo representado en la Figura 46. Un conjunto de 16 puntos que únicamente corresponden a los bordes de los círculos, habiendo desaparecido el resto de los puntos del marco que pudieran restar.

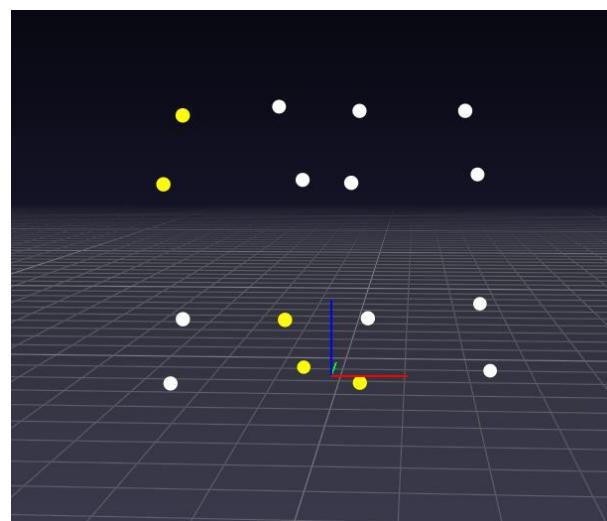


Figura 46: Salida de la tercera fase de calibración

3.6.4.1.3 OBTENCIÓN DE LOS CENTROS

Esta cuarta etapa encuentra los cuatro clusters contenidos en la nube de puntos y calcula sus centros, dando como resultado los cuatro centros, visualizado en la Figura 47.

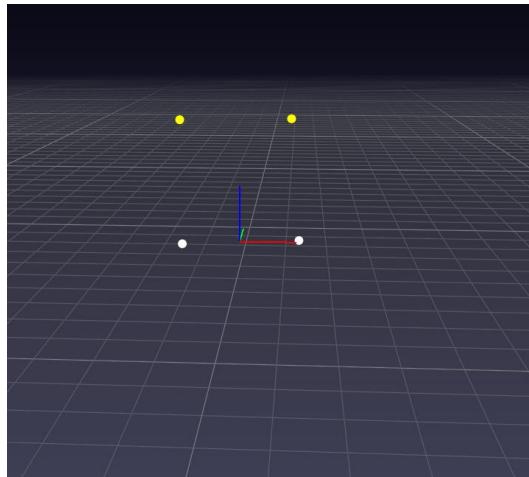


Figura 47: Salida del algoritmo de calibración

Para visualizar mejor esta salida, la superponemos sobre el marco original, obteniendo en la Figura 48:

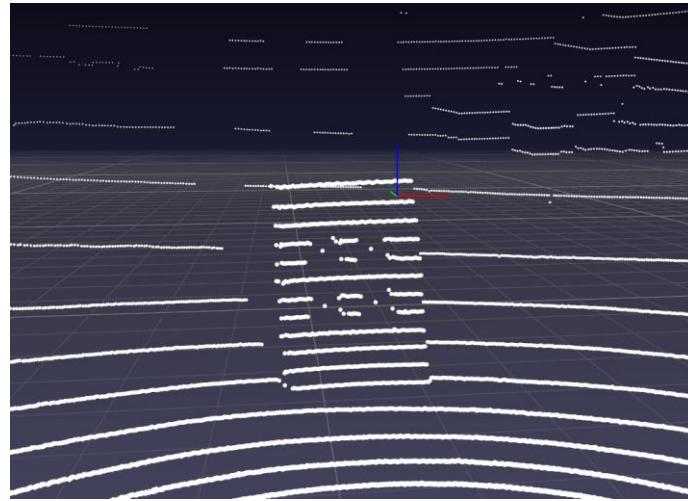


Figura 48: marco de calibración con los centros hallados

3.7. SÍNTESIS DE LA DETECCIÓN CON LIDAR

Una vez llegados a este punto, habiendo finalizado la sección de detección con Lidar, creamos esta sección a modo de síntesis o resumen de toda la sección 3, para dar un mejor entendimiento en caso de que, al estar tan compartimentado, se haya podido perder la unidad estructural que reside tras el algoritmo de detección.

El primer lugar, se han desarrollado las cuatro fases de detección implementadas hasta la fecha:



Comenzando por un filtrado de altura y de distancia, se elimina la mayor parte de la información que no resultará de utilidad. Tras esto, un filtrado de suelo (todo esto en *Ground Filtering*) permite eliminar de la nube de puntos el suelo y que de esta forma resten únicamente los objetos que se sitúan sobre este. Los muros deberían estar filtrados por el filtro de distancia en condiciones de competición (de ahí que únicamente deban restar los objetos) aunque su permanece alguna sección de un muro no sería un problema ya que se filtraría en las siguientes fases.

La segunda fase, *clustering*, distingue los diferentes objetos contenidos en la nube de puntos. Con esto ya tenemos la nube de puntos dividida en sub-nubes diferenciadas.

La tercera fase, *reconstruction*, completa cada *cluster* con aquellos puntos pertenecientes a este que habían sido filtrados en *Ground Filtering*. Esto permite recuperar aquella información útil que podría haber sido perdida, y únicamente aquellos puntos del plano de suelo que pertenezcan al clúster al situarse debajo de este, no el resto de los puntos del suelo que continuarán filtrados.

Por último, el algoritmo de *non-cone filtering* compara el número de puntos de cada *cluster* reconstruido al número de puntos que debería tener si se trata de un cono de la competición. Con esto, se filtran aquellos objetos que no resulten ser un cono de la competición, restando únicamente estos últimos.

Faltaría una quinta fase, que no ha sido desarrollada y que distinguiría entre los conos que se encuentran a la izquierda y los que se encuentran a la derecha del coche, información importante para el algoritmo de control. Esto se distingue por colores en la división de visión con cámara estéreo y, aunque podría no resultar esencial aquí, convendría desarrollarla para ganar esa redundancia que hemos buscado a lo largo de todo el proyecto para que la fusión de ambos sensores resulte en una precisión incrementada.

Destacar que, hasta este punto, las fases que han sido explicadas se habían desarrollado en Python y testeadas para un único frame, ya que en los IDES que utilizábamos (Spider y Júpiter) y en Windows, no se puede testear su funcionamiento frente a un flujo constante de frames. Para ello se llevó a cabo la implantación en ROS2, que constituye toda la etapa siguiente. Con esta testeamos tanto todos los frames que habíamos registrado nosotros en el INSIA como un dataset registrado en la propia competición e idóneo por tanto para adaptarnos a las condiciones de esta última.

Las fases fueron testeadas de forma gradual hasta llegar a una versión que incorporaba todas ellas. En esta última versión se observó que la única salida del algoritmo eran los conos de la competición, por lo que se validó su funcionamiento para ser la primera iteración que se realizaba. Sin embargo, se observó a su vez que en ciertos frames había una desaparición de algunos conos que no deberían haberse filtrado.

Habiendo referenciado los algoritmos con artículos de prestigio, los algoritmos utilizados deberían acondicionarse a la robustez de lo obtenido en dichos artículos, al tratarse de la misma metodología. Es por ello por lo que consideramos se debe prestar atención a la optimización de los diferentes parámetros de los algoritmos. No contamos con el tiempo para realizar dicha optimización y consideramos que mejorará la precisión de la detección hasta obtener unos



resultados más sólidos, con una probabilidad mucho menor de encontrar una desaparición de algún cono.

Por último, sabemos que la información del Lidar debe fusionarse con la de la cámara estéreo. Estando ambos dispositivos situados en diferentes puntos del vehículo, sus sistemas de referencia son diferentes. Para fusionar ambas informaciones, se debe crear una correspondencia entre cada punto registrado en ambos dispositivos, para lo que es necesario referenciar ambos sistemas de referencia a uno común. Realizar esto de forma manual puede ser realmente complicado al ser necesaria tanta una matriz de rotación como una de traslación, constituyendo ambas la matriz de transformación final.

Es por ello por lo que se utiliza un método que calibrará automáticamente ambos dispositivos, una vez situemos un marco como el explicado delante de ambos dispositivos. Para mejorar la precisión, se debe realizar este proceso más de una vez, obteniendo así una matriz de transformación que resulte la media de la original. Esta última etapa no se ha realizado, a la espera de que se realice la parte del calibrado en la división de visión con cámara estéreo. Si no se ha realizado es porque no se contaba competir este año con Lidar en la competición al no poder contar con este dispositivo y se priorizaron otras tareas en el equipo.



4. CONTROL

4.1 RESUMEN

En este **apartado** se explicará el funcionamiento del control utilizado en el vehículo autónomo de la temporada 2019/2020. Se hará uso de un control longitudinal de velocidad constante y un control lateral sobre el volante para el seguimiento de una trayectoria. El algoritmo usado para el control lateral es el Stanley Controller, el cual fue creado por la universidad de Standford para la competición *DARPA Challenge*.

4.2 OBJETIVOS

El objetivo de este trabajo consiste en crear el control lateral y longitudinal de tal forma que el vehículo sea capaz de seguir una trayectoria la cual se va creando dinámicamente en lo que pasa por un camino delimitado por conos. De esta forma, el vehículo va detectando los conos y a su vez va creando una trayectoria entre ellos. Esta información llega al control, que se encarga de que el vehículo la siga lo más fielmente posible. Para ello, se definen los siguientes objetivos:

- Establecer un control longitudinal que permita una circulación a velocidad constante, la cual se alcance de forma suave. Incluir un sistema de frenada progresivo una vez se haya recorrido el circuito.
- Lograr un control lateral del vehículo, que permita seguir una trayectoria definida con el menor error de posicionamiento posible y que sea estable, de forma que no alcance el objetivo con movimientos bruscos que puedan comprometer el comportamiento dinámico del vehículo.
- Crear una trayectoria a seguir durante el propio movimiento del vehículo a partir de la posición de los conos detectados con la cámara. Estos conos deben leerse de tal forma que la trayectoria creada al final tenga sentido y no se desvíe del camino, lo que supondría un fallo en la prueba.
- Leer datos del vehículo proporcionados por el GPS y la IMU, así como la posición de los conos otorgada por la cámara o el Lidar. Además, se debe conectar el programa con el vehículo de tal forma que sea capaz de enviar las órdenes de giro



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



INDUSTRIALES
ETSII | UPM

de volante y de pedal de freno o acelerador con una frecuencia suficiente para el correcto funcionamiento.

- Sincronizar todas las señales enviadas y leídas para que trabajen a una misma frecuencia, de tal forma que diferentes órdenes no se asocien a medidas que corresponden a tiempos ya pasados.



4.3 DESARROLLO

Para el desarrollo del control del vehículo, en primer lugar, se ha partido de una trayectoria ficticia “invisible”. Esto es, se quiere que el vehículo vaya de un punto a otro a través de diferentes “*checkpoints*”, los cuales en un primer lugar se conocen.

Con estos puntos conocidos se debe crear una trayectoria y que en una primera instancia ha sido mediante un “*spline* cúbico” que pase por todos los “*checkpoints*”. De esta forma, el primer objetivo ha sido el crear un algoritmo que sea capaz de seguir esa trayectoria.

El control debe ir enfocado a una parte longitudinal, en la que se actúa sobre la aceleración y el frenado y una parte lateral, en la que se actúa sobre el volante para corregir la trayectoria

Para probar el algoritmo, en primer lugar, se ha optado por simulación, en la cual se obtiene la posición del vehículo por medio de las ecuaciones cinemáticas de un punto. Como se puede observar, se comienza por la idea más simple para poder comprobar si el algoritmo seleccionado funciona correctamente. Una vez simulada correctamente, se va creando un modelo más complejo.

Una vez se han logrado resultados satisfactorios en simulación, el siguiente paso es probar el algoritmo en un vehículo real. Este cambio supone numerosas modificaciones en el algoritmo, ya que ahora no se pueden utilizar cálculos de ningún tipo para suponer diferentes características. Así mismo, se deberán leer ciertos parámetros del vehículo para poder operar correctamente con el algoritmo. Para la comunicación del código con el vehículo se usará ROS (*Robot Operating System*).

Una vez conseguida la implementación del algoritmo en un vehículo real y realizadas de forma satisfactoria diversas pruebas, el siguiente paso es trabajar con una trayectoria no ficticia, esto es, con una trayectoria que ha sido detectada con los sensores y por lo tanto se cree el camino mientras el vehículo va recorriendo los puntos ya creados. Esto supondría el final del algoritmo de control y permitiría al vehículo ajustarse a una trayectoria y recorrerla de forma completamente autónoma.

El código utilizado ha sido realizado con *Python* debido a su compatibilidad con ROS y por su sencillez para empezar a trabajar, ya que ningún miembro lo conocía de antemano.



4.3.1 CONTROL LONGITUDINAL

El primer paso para el control del vehículo es su comportamiento longitudinal. Debido a que el coche utilizado es proporcionado por la competición y no contamos con él para la realización de ensayos, la intención es la de circular a baja velocidad en una primera instancia. Como la velocidad es baja, se busca que el circuito de la competición se realice a velocidad constante, de forma que el vehículo únicamente tenga que frenar cuando termine el recorrido.

Así pues, para lograr que el vehículo circule a una velocidad constante, se hará uso de un controlador PID, en el cual la velocidad objetivo es constante durante todo el recorrido y la velocidad actual se lee del vehículo por medio del *framework ROS*.

$$v(t) = K_p e(t) + K_i \int_0^t e(t) dt + K_d \frac{de}{dt}$$

Siendo $e(t)$ el error entre la velocidad objetivo y la velocidad en el instante t . Las constantes K son las acciones proporcional, derivativa e integral, que determinarán la busqueda con la que se alcanza la velocidad objetivo, la oscilación que se da hasta que se estabiliza en el valor objetivo y la precisión adquirida. De esta forma, se obtiene $v(t)$, que es la velocidad en cada instante, que es la orden que se mandará a los actuadores.

De esta forma, es necesario elegir un valor para las acciones proporcional, integral y derivativa que permitan una aceleración adecuada, sin demasiadas oscilaciones y que llegue al objetivo de forma precisa. Sin embargo, debido a que la precisión para alcanzar una velocidad en concreto no es estrictamente necesaria, en una primera instancia se trabajará únicamente con acción proporcional.

La utilización de una velocidad adaptativa a las condiciones del terreno se ha aplazado hasta contar con un vehículo propio o hasta poder realizar las suficientes pruebas en un vehículo real como para conocer la precisión de todas las maniobras realizadas y así adentrarse en un algoritmo de este tipo.

Como en la competición no se necesita modificar los controladores de bajo nivel, la orden mandada es la de velocidad objetivo, ya que será el propio código del coche el que realice las transformaciones necesarias para que el vehículo alcance dicha velocidad.



4.3.2 CONTROL LATERAL

El control lateral busca establecer la posición correcta del vehículo respecto a una trayectoria. Existen diferentes algoritmos para lograr este objetivo y de entre todos se ha optado por el control Stanley, el cual es un controlador en bucle cerrado [32].

Este algoritmo fue creado por la universidad de Stanford para la competición DARPA, en la cual un vehículo autónomo *off-road* debe seguir un trayecto esquivando diferentes obstáculos durante más de 100 millas de recorrido. Se define como un algoritmo que no depende de superficies rectas y que aporta una gran estabilidad global y buena precisión de seguimiento de la ruta (en la competición se tuvo un error de posición menor a 0,1 m). Se basa en tener en cuenta las distancias de la pista al eje delantero, en vez de a todo el vehículo, con la intención de que el eje de la dirección sea el que se encuentre en la ruta. Además, lleva muy poco coste computacional.

El motivo de la elección de este algoritmo por parte del equipo es el resultado que obtuvo en la competición, así como el logro que obtuvieron de no chocar con ningún obstáculo, lo que muestra el alto nivel de seguimiento de ruta que proporciona el algoritmo y es esencial a la hora de seguir una ruta delimitada por conos, donde el espacio libre entre el vehículo y éstos puede llegar a ser inferior a medio metro en cada lado.

Como este algoritmo se ha utilizado para competir en la *FS-AI DDT Class*, en la cual el vehículo es proporcionado por la competición, hemos establecido trabajar a una velocidad constante y baja, dado que la organización no proporciona apenas información sobre el vehículo y adentrarse en un modelo dinámico no sería la mejor opción.

Por lo tanto, se va a trabajar con un modelo cinemático, ignorando las inercias y aceleraciones a las que se ve sometido el algoritmo. Si se lograsen numerosas pruebas reales se podría establecer en un futuro un modelo dinámico parametrizado de forma que se adapte a diferentes posibles vehículos.

En primer lugar, se debe proporcionar una ruta a base de puntos al programa, que vendrían a ser por ejemplo el punto medio entre cada pareja de conos azul-amarillo. A partir de estos puntos, se crea la trayectoria uniendo todos los puntos por medio de un *spline* cúbico. Una vez creado el *spline*, se divide en subdivisiones aún mucho menores para garantizar una mayor precisión entre los puntos que se otorgan inicialmente. De esta forma, se puede pasar de unos 20 puntos de ruta dados inicialmente, a más de 1000 por medio de las subdivisiones, creando un sistema de seguimiento mucho más preciso, ya que el algoritmo trabaja con estas subdivisiones, **Figura 49**.

Una vez creada la trayectoria, el algoritmo comienza a funcionar. El vehículo va siguiendo todas las subdivisiones en orden de forma que el vehículo no retroceda o se salte partes

del circuito. Así, el vehículo tendrá como objetivo la subdivisión más cercana y una vez que la ha pasado cambia de objetivo a las siguientes. Sin embargo, el algoritmo permite que, si el vehículo empieza a mitad de la trayectoria, su punto objetivo se encuentre también en la mitad, es decir, no obliga al vehículo a realizar el recorrido desde el principio.

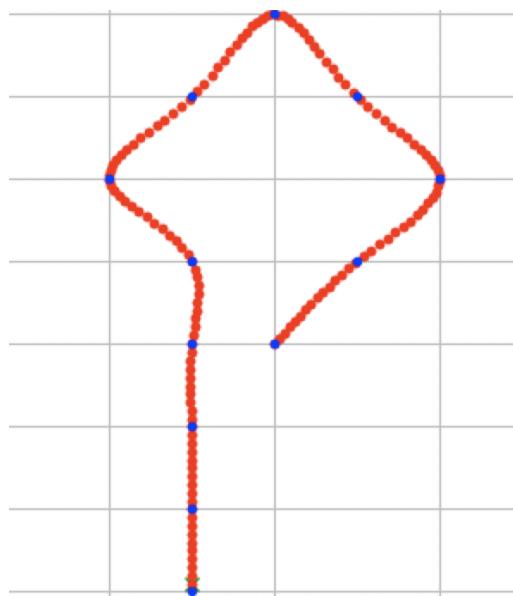


Figura 49: Creación de la ruta a partir de los puntos medios de los conos

En la figura se pueden observar en azul los puntos iniciales otorgados al programa (los puntos medios de los conos) y en rojo las subdivisiones creadas a partir de unir los puntos azules por medio de un *spline* cúbico.

El funcionamiento del algoritmo se basa en la minimización de dos errores para lograr colocarse encima de la trayectoria, donde estos son nulos, Figura 50.

- El *heading error* es el error de orientación del coche. Se obtiene como la diferencia de orientación entre el vehículo y la derivada al *spline* en el punto objetivo al que se está dirigiendo el vehículo. Esta orientación se ha denominado como *yaw angle* y está referido en unos ejes globales, siendo 0° en la dirección *X* y 90° en la dirección *Y*. Es el giro del vehículo sobre el eje *Z*. Por lo tanto, este error es **nulo** cuando el vehículo circula de forma paralela a la trayectoria.

$$\delta_{\text{heading}} = \theta_{\text{trayectoria}} - \theta_{\text{vehículo}}$$

Donde $\theta_{\text{trayectoria}}$ es la derivada de la trayectoria en el punto objetivo del vehículo y $\theta_{\text{vehículo}}$ es el ángulo de guiñada del vehículo.

- El *cross-track error* mide la distancia en paralelo al vehículo del eje delantero al punto de la trayectoria objetivo. Si el vehículo no se encuentra paralelo a la trayectoria, se proyecta la distancia al punto objetivo sobre la normal a la dirección de éste. Por lo tanto, este error es **nulo** cuando el vehículo se encuentra sobre la trayectoria.

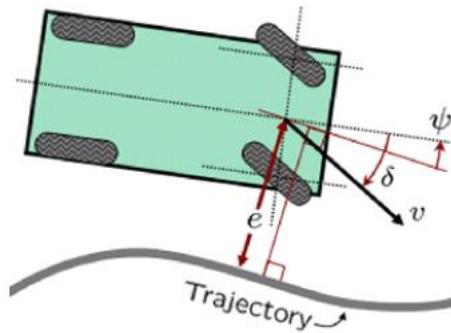


Figura 50: Esquema de los errores Heading y crosstrack

En la Figura 50, se puede observar un esquema de los errores del vehículo. Por un lado, e muestra el error *crosstrack* o de distancia en paralelo. Por otro lado, la diferencia entre los ángulos de guiñada es el *crosstrack*. La suma de ambos errores transformada a ángulo, en el caso del *crosstrack* y como se comentará más adelante, es δ , el ángulo de guiado de las ruedas del vehículo.

Como se puede observar, ambos errores son independientes entre sí, siendo cada uno responsable de diferentes pautas sobre la posición. El *heading error* busca orientar el vehículo de forma adecuada y el *crosstrack* busca posicionarlo en el lugar adecuado.

El *heading error* proporciona un error en ángulo, mientras que el *crosstrack* lo proporciona en distancia. Debido a que se actúa sobre el ángulo de guiado del vehículo, es necesario que ambos estén referidos a ángulo para así indicar cuantos grados hay que girar el volante.

Por lo tanto, es necesario transformar el *crosstrack error*. Para ello, se utiliza como información dicho error en distancia y la velocidad del vehículo. Con ambos datos se busca que a bajas velocidades proporcione un mayor “error en ángulo” y por lo tanto mande una orden de un giro más cerrado y que a altas velocidades las maniobras sean más suaves. Es por eso por lo que se usa la siguiente fórmula:

$$\delta_{crosstrack} = \tan^{-1}\left(\frac{k \cdot \text{crosstrack_error}}{v}\right)$$

Se introduce una constante k para poder ajustar la brusquedad y suavidad del giro según se desee. De esta forma, la orden de giro de volante que se manda es la suma de ambos errores, siguiendo un sentido positivo antihorario.

$$\delta = \delta_{heading} + \delta_{crosstrack}$$

De esta forma, el algoritmo se muestra robusto frente a diferentes posibilidades. Si el vehículo se encuentra sobre la ruta, pero no es paralelo a ella, el *crosstrack* será nulo, pero el *heading error* no, que será el que corrija la orientación. Si el vehículo se encuentra fuera de la ruta, pero aproximándose a ella, los errores de *heading* y *crosstrack* tendrán signos diferentes y será uno de los dos el que predominará sobre el otro, que se corregirá más adelante.

El controlador Stanley actúa en bucle cerrado, recibiendo como realimentación el estado del vehículo, es decir, su posición en 3 grados de libertad, su velocidad y la posición del volante. En base a esta información del anterior ciclo, calcula cuánto debe girar el volante en el ciclo actual.

El funcionamiento del algoritmo de control se puede resumir en la siguiente **Figura 51**. En primer lugar, se recibe la información del estado del vehículo. Después en base a ella se define el punto objetivo más cercano. En base al punto elegido se calculan los errores *heading* y *crosstrack* y se manda la orden de giro de las ruedas. En el siguiente ciclo, se vuelve a analizar la posición y se vuelve a empezar.

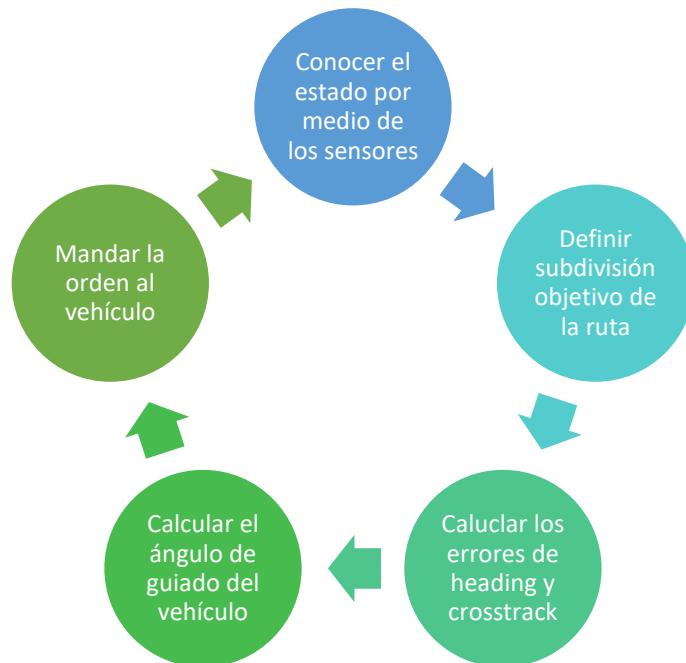


Figura 51: Esquema de funcionamiento del algoritmo de control lateral

4.4 RESULTADOS



4.4.1 SIMULACIÓN

Como ya se ha mencionado anteriormente, el objetivo de la simulación es comprobar el funcionamiento del algoritmo con las condiciones menos exigentes. De esta forma, se ha introducido únicamente un modelo cinemático del vehículo, lo que viene a ser ignorar las diferentes inercias del vehículo y las aceleraciones a las que se ve sometido.

La simulación es una parte muy importante ya que permite corregir errores que de no haberlo hecho y haber probado directamente sobre un vehículo real podrían haber provocado errores graves.

En primer lugar, el control longitudinal se ha utilizado un PID para alcanzar una velocidad deseada, que será constante para la simulación. Nuevamente, la simulación nos permite corregir de forma adecuada los parámetros del PID para obtener la respuesta que se desea.

El control lateral se considera como un ángulo *delta*, el cual está referido respecto al eje longitudinal del vehículo. Este ángulo delta es el que debe girar el vehículo. De esta forma, la orientación del vehículo viene definida por el ángulo *yaw angle*, que es el de orientación del propio vehículo, o **ángulo de guiñada**. Así, se definen las siguientes ecuaciones cinemáticas, que permitirán conocer el estado del vehículo durante la simulación.

$$\begin{cases} x = x_0 + v \cdot \cos \text{yaw} \cdot dt \\ y = y_0 + v \cdot \sin \text{yaw} \cdot dt \end{cases}$$

De esta forma, según la velocidad que lleve el vehículo y la orientación de éste, se conocerá la posición sobre el plano en referencia global. El término *dt* hace referencia a la duración de cada ciclo. Se desea que el programa trabaje a una frecuencia de 10 Hz, por lo que *dt* será de 0,1 segundos.

Para la obtención del ángulo *yaw*, se utiliza la siguiente fórmula.

$$\text{yaw} = \text{yaw}_0 + \frac{v}{L} \tan \delta \cdot dt$$

Siendo δ el ángulo de guiñada de las ruedas del vehículo con respecto al eje longitudinal y L la batalla del vehículo. Esta fórmula se puede deducir a partir del modelo de la bicicleta de un vehículo que se encuentra girando y tiene un centro de rotación sobre el cual realiza la curva.

El procedimiento de simulación consistió en una evolución de dificultad. Para ello, en primer lugar, se probaron trayectorias rectas, en las que el vehículo comenzaba en el primer punto. Posteriormente, a dicha trayectoria se le incorporaba una curva, para así observar la respuesta del algoritmo mediante el controlador Stanley.

Cuando el vehículo debe realizar una trayectoria recta y además empieza exactamente en ella, el algoritmo funciona perfectamente. Si se encuentra además orientado de forma correcta, los errores de orientación y de posicionamiento son nulos, por lo que no se actúa sobre el volante. Como se puede observar, este caso es prácticamente improbable en la realidad.

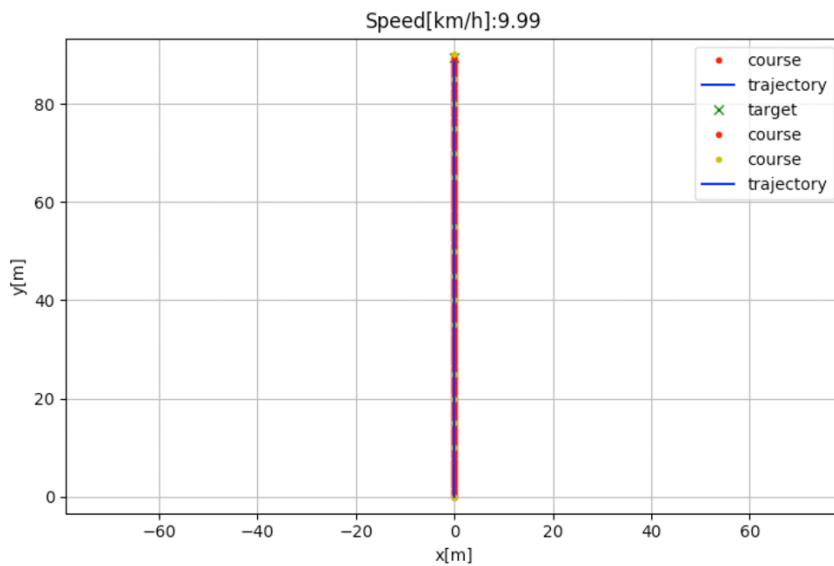


Figura 52: Simulación en trayectoria recta empezando en ella

La prueba se realiza sin ningún problema, como se puede ver en la Figura 52. En amarillo se encuentran los puntos que se han dado para la trayectoria que se quiere seguir, en rojo. Por último, en azul se observa la trayectoria seguida por el vehículo virtual.

Al introducir una curva en el recorrido, el control lateral debe actuar, lo que nos permitió observar la precisión seguida por parte del algoritmo.

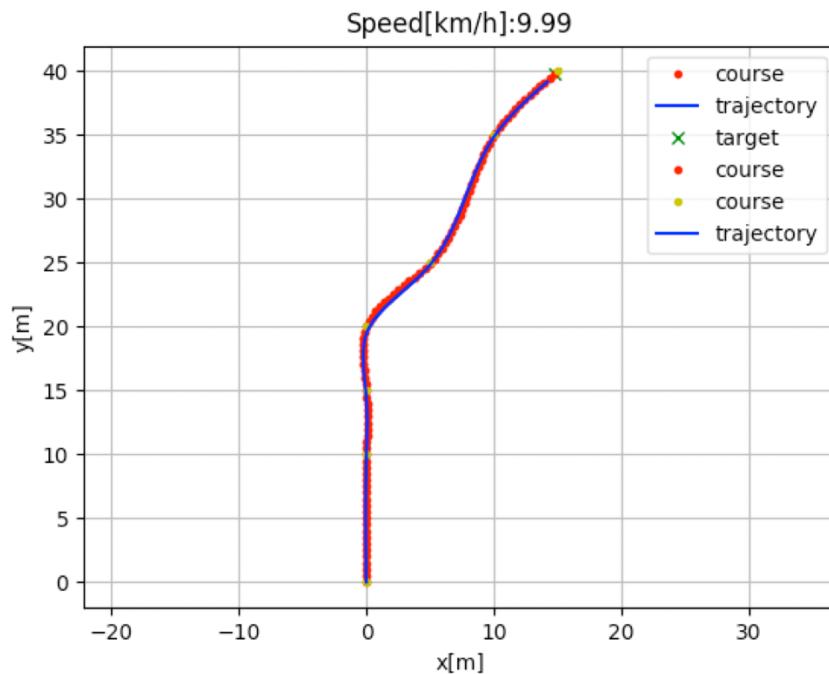


Figura 53: Simulación en trayectoria curvilínea, comenzando dentro de ella

Esta prueba exigía una mayor dificultad, pero se puede observar como el algoritmo es capaz de resolverla a la perfección, Figura 53. Para medir la precisión de la prueba, se ha utilizado el error medio cuadrático (*mse*), para comprobar el error existente entre la trayectoria seguida y la deseada. El *mse* obtenido para esta prueba es de 0,04 m. Esto indica que la precisión aún con curvas es muy alta. Además, se muestra por gráfica el error de posición en cada instante.

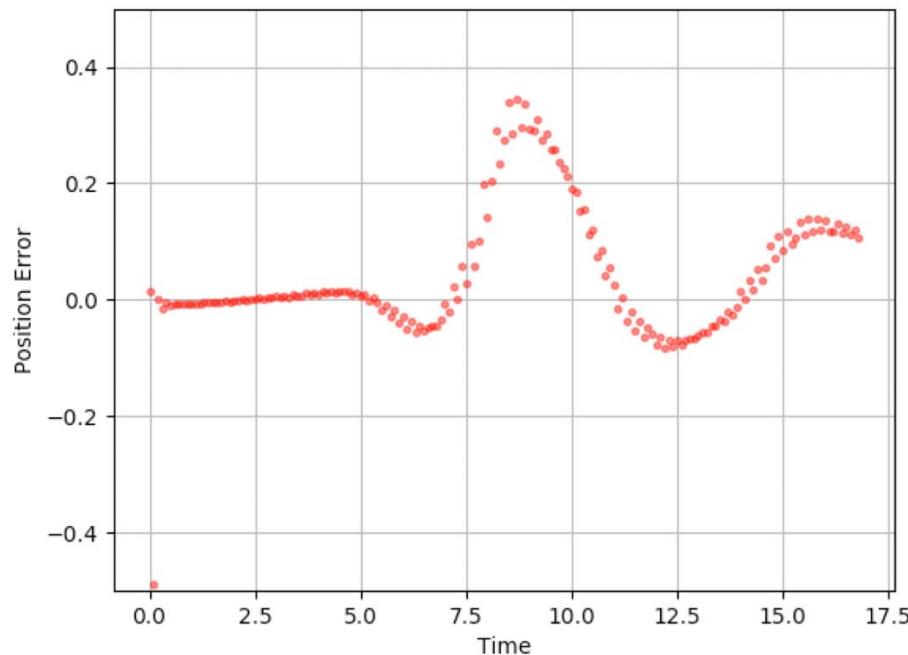


Figura 54: Error de posición del vehículo en trayectoria curvilínea

Se puede observar cómo al tomar la curva se produce un incremento del error, pero que es rápidamente solucionado, de forma que el error vuelve a minimizarse, **Figura 54**. En la siguiente figura, , se prueba la misma trayectoria anterior, pero comenzando a 2,5 metros del punto inicial.

De esta forma, y continuando con una dificultad creciente de los ensayos, se decide probar en un caso en el que el vehículo no empiece encima de la trayectoria. Esto es algo que se puede dar de forma muy probable en la realidad, ya que posicionar el vehículo en el punto donde se desea es muy complicado.

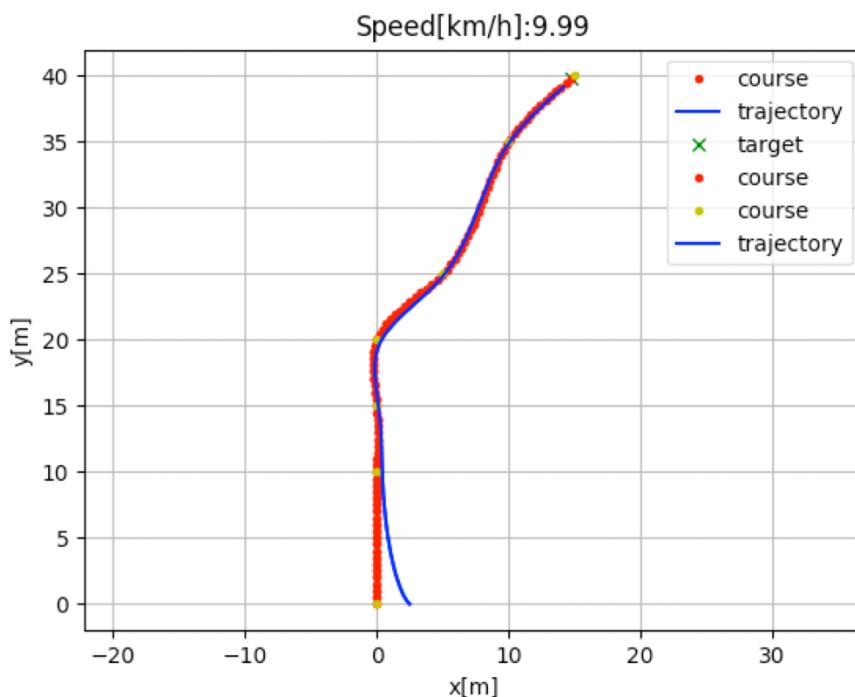


Figura 55: Trayectoria curvilínea comenzando a 2,5 m de ella

Se observa cómo va aproximándose a la trayectoria deseada de forma progresiva y como antes de 15 metros ya ha encauzado el camino como si no hubiese comenzado desviado, **Figura 55**. En este caso, el mse es notablemente mayor, ya que se debe corregir una desviación de 2,5 m. Se obtiene un $mse = 0,37$ m. El error de posición en cada instante es el siguiente.

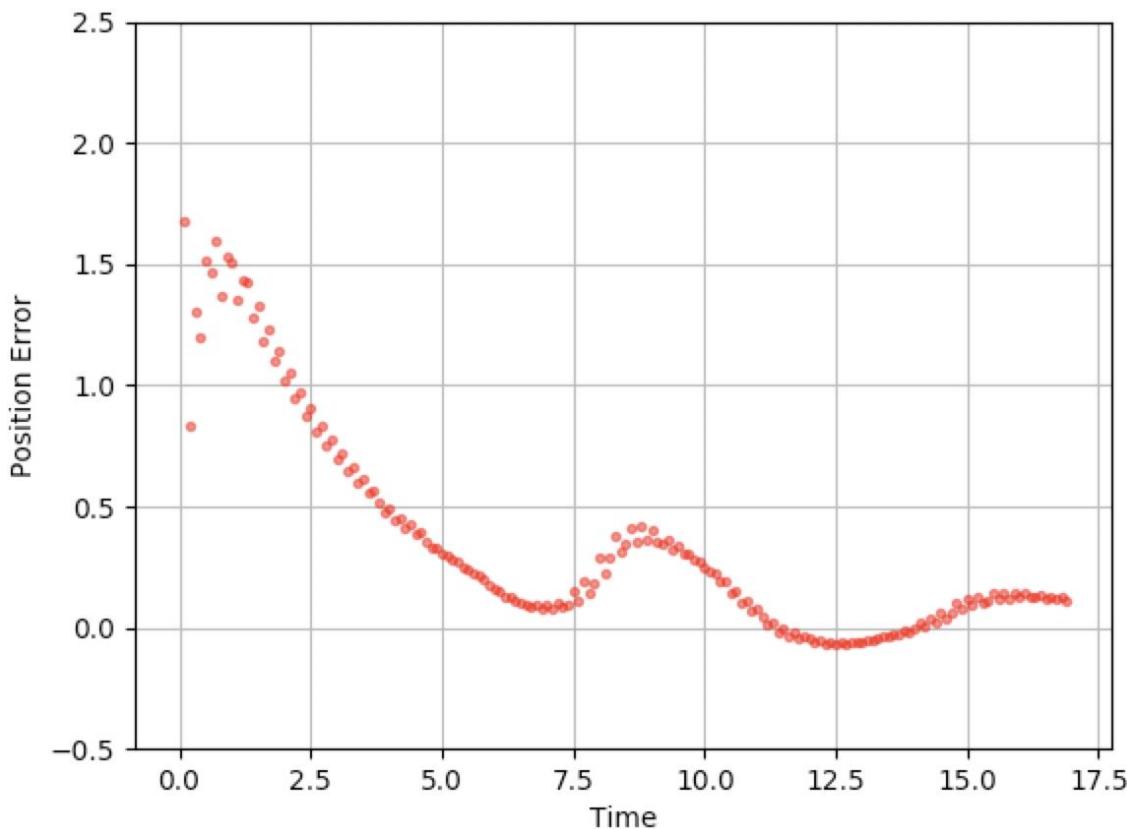


Figura 56: Error de posición en trayectoria curvilínea comenzando a 2,5 m de ella

Donde se puede ver muy bien cómo el algoritmo va corrigiendo el error de posición hasta que se estabiliza antes de llegar a la curva, Figura 56. Una vez probados diferentes modos, se observó que el vehículo no era capaz de seguir trayectorias en las cuales se formasen “lazos”, es decir, donde la trayectoria cortase un punto por donde ya había pasado, como se puede observar en la Figura 57.

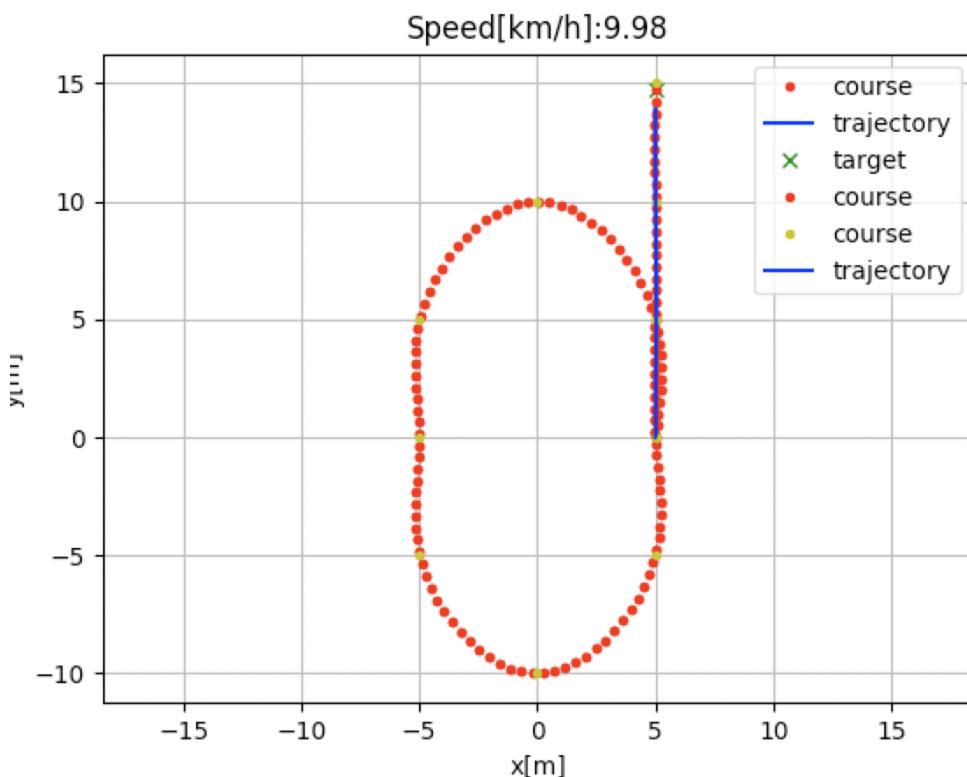


Figura 57: Prueba fallida en trayectoria con un lazo

Ver qué parte del código era el que daba este comportamiento fue algo que llevó más tiempo del esperado. En teoría el algoritmo debe ir de punto rojo a punto rojo en orden, de forma que no se salten partes del circuito. Sin embargo, se observó que existía una colisión entre esto y el método que se utilizó para que el coche encauce la ruta si no se encuentra en ella.

Este segundo método hacía que el vehículo decidiese como su objetivo a aquel punto que se encontrase a la menor distancia, por lo que, si un vehículo iniciaba a la mitad de la trayectoria, el punto más cercano se encontraría lo más seguro que en paralelo a su eje longitudinal. Sin embargo, si se producía un lazo en el circuito, el vehículo entraría adecuadamente en la intersección, pero seguramente el siguiente punto objetivo no sería el consecutivo en orden, si no el punto más cercano al final de la ruta, lo que haría que el vehículo omitiese el lazo y por lo tanto la prueba no fuese correcta.

Para corregir este comportamiento, se hizo que el algoritmo comprobase constantemente a que objetivo estaba enfocando y mirase en un intervalo de hasta 30 subdivisiones. Por lo tanto, al llegar a un lazo y estando por ejemplo en el target 150, observaría que sus dos puntos más cercanos serían el **target** 151 y el **target** 400, por lo que elegiría el 1511 y por lo tanto realizaría el circuito al completo.

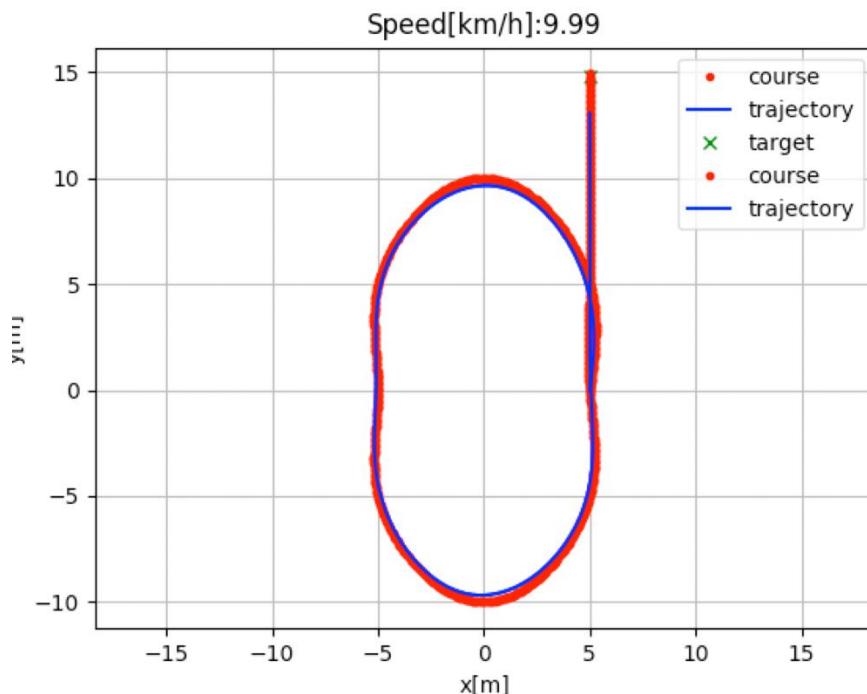


Figura 58: Prueba satisfactoria en trayectoria con un lazo

Al realizar la prueba con el código correspondiente añadido, se observa como se supera de forma satisfactoria el circuito en forma de lazo y por lo tanto supone un avance más de cara a probar el vehículo en un test real, Figura 58.

El siguiente paso es comenzar a trabajar con las rutas que se podrían utilizar en el test real, pero con el simulador. Para ello, se tienen un total de 12 rutas, de las cuales 6 son en el circuito hallado en el INSIA. Este test tiene unas ciertas particularidades.

Para poder trabajar en esos circuitos, se van a utilizar las coordenadas reales que se tendrían allí. Esto significa que es necesario obtener las coordenadas GPS del circuito y más concretamente las del punto inicial de la ruta. Posteriormente se deben transformar estas coordenadas al plano cartesiano para adecuarlas al programa. Con esto se busca observar si trabajar con coordenadas de magnitudes muy elevadas (más de 10^6 metros para cada una).

Además, es necesario aprender a trabajar con unas nuevas librerías de *Python*, que permiten leer mapas y transformar recorridos a vectores bidimensionales. De esta forma, no es necesario crear rutas “a mano”, si no que a partir de ahora se pueden leer rutas dibujadas sobre un mapa. Un ejemplo de una ruta ya creada es la siguiente, Figura 59.



Figura 59: Trayectoria en el circuito del INSIA

La cual fue posteriormente simulada para comprobar el comportamiento. En simulación se obtuvieron los siguientes resultados, Figura 60.

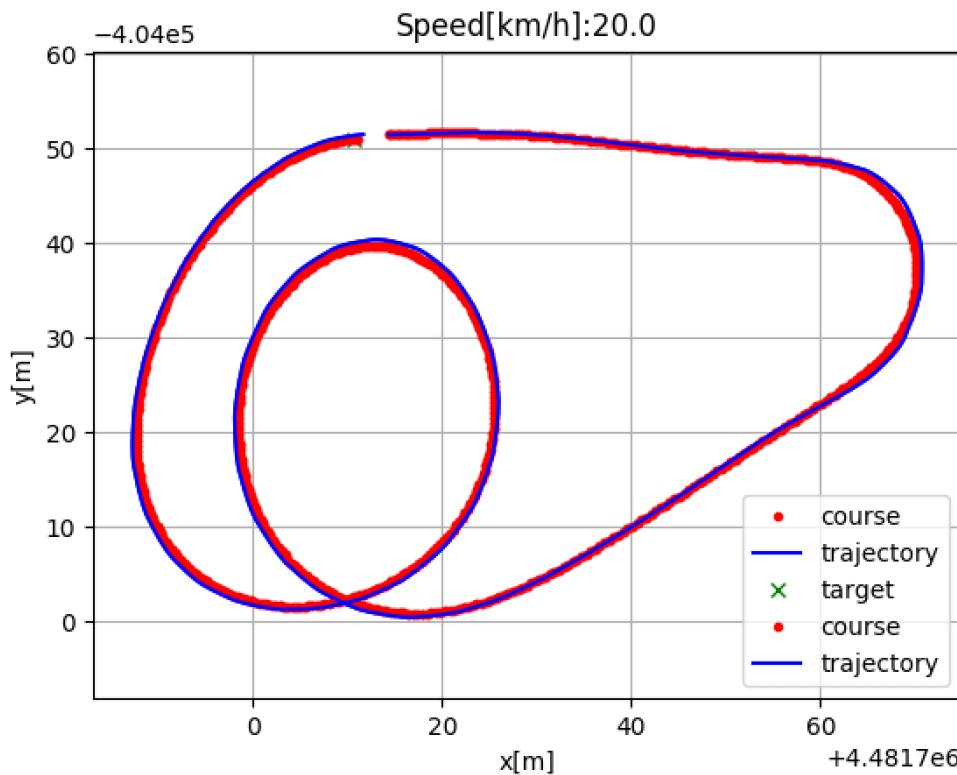


Figura 60: Simulación de trayectoria real

Al realizar la prueba observamos que la trayectoria se sigue de forma correcta, al igual que el resto de las simulaciones realizadas con las trayectorias inventadas. Se obtiene un $mse = 0,2$ m. También se observa que a mayor velocidad, menor precisión en la toma de las

curvasCabe destacar que el ensayo se realizó a 20 km/h y el error de posición en cada instante fue el siguiente, Figura 61.

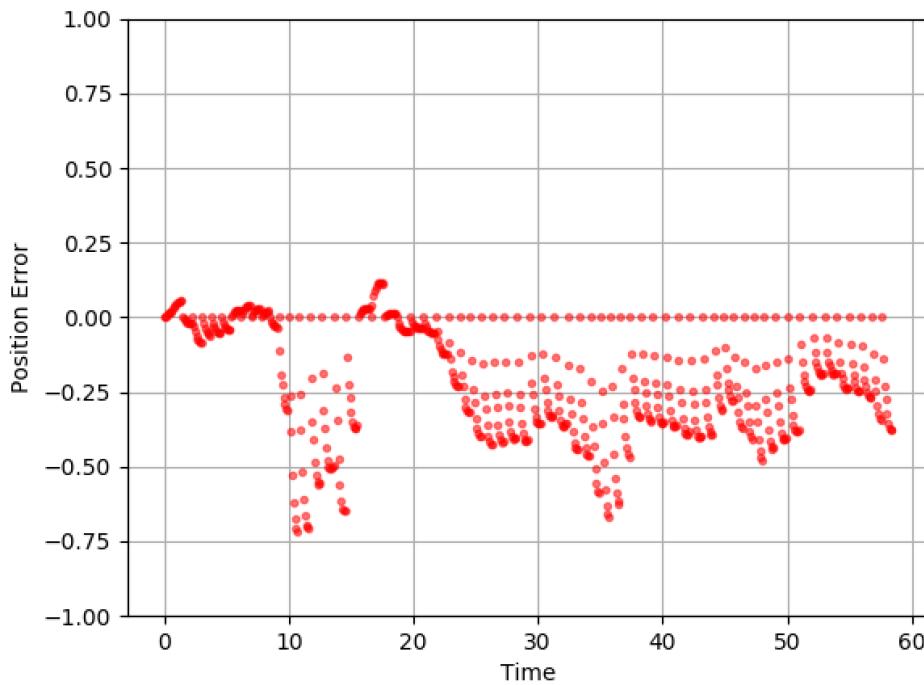


Figura 61: Error de posición en simulación de trayectoria real

Se puede observar como se producen cambios ligeramente bruscos en el comportamiento y eso se debe al enfrentamiento entre la componente *heading* y el *crosstrack*, la cual se comentará más adelante.

Tras todas estas simulaciones realizadas, se concluye que el algoritmo está preparado para comenzar su adaptación a un vehículo real, con todo lo que esto supone.

4.4.2 PRUEBA EN VEHÍCULO REAL

Para la prueba en un vehículo real, es necesario realizar varias adaptaciones al código para que funcionen de forma correcta. En primer lugar, en simulación se utilizaban diversos cálculos matemáticos para conocer el estado del coche. Bien sus coordenadas cartesianas o su orientación, ambas por medio de la velocidad, necesitan plantearse de otra forma.

Si se utilizan cálculos para suponer ciertos estados del vehículo, las posibles diferencias con el sistema real (ya que siempre las habrá), provocan que ciclo a ciclo se vaya acumulando error, lo que termina en que al cabo de varios segundos el vehículo se ha desviado notablemente de su trayectoria.



Para la posición del vehículo, se tomará la lectura del GPS en cada ciclo. Lo mismo ocurre con la velocidad. Para el ángulo de guiñada del coche, ya no se puede utilizar la misma fórmula que se usaba en la simulación. En una primera instancia, para obtener dicho ángulo se utilizó trigonometría. Esto es, se comprobaba la posición actual del vehículo y se miraba la de una distancia atrás (aproximadamente 30 cm). Una vez se tienen las coordenadas de ambos puntos, se obtiene la recta que los une y se calcula la pendiente, es decir, la tangente. Dicho ángulo sobre el eje X será el ángulo absoluto de guiñada del vehículo.

Una vez realizadas dichas modificaciones, es necesario empezar a trabajar con la comunicación ordenador-vehículo. Esto se realiza por medio de ROS (*Robot Operating System*). Aquí no se ahondará en información sobre el funcionamiento de ROS, si no que sólo se comentarán las partes que fueron necesarias añadir al código para lograr la correcta comunicación.

ROS utiliza una estructura de módulos *Subscriber* y *Publisher* en forma de nodos. Así, un nodo suscrito a otro podrá recibir la información que éste publica (*Publisher*). Un nodo puede ser a la vez *Publisher* y *Subscriber*.

Analizándolo de esta forma, existe cierta información del vehículo que el código debe leer, es decir, a la que se debe suscribir. Esta información es la señal de GPS, la velocidad del vehículo y el ángulo de giro del volante. A su vez, el código debe mandar las respuestas a los actuadores, que en este caso serán los *Publisher* de velocidad del vehículo y la comanda de giro de volante.

Se debe crear una función que contenga la denominación de los diferentes nodos que se quieren crear. Un ejemplo es el siguiente.

```
rospy.Subscriber('/gps/position', Vector3, callbackgps)
```

Donde se puede ver cómo el código se está suscribiendo al nodo publicador de la señal GPS. Esta información la ejecuta por medio de la función del código *callbackGPS*, la cual se activará siempre que se lea una señal de GPS.

Continuando con la lectura del GPS, en la función *callbackGPS* se recibe como entrada un vector de dos dimensiones que contiene la latitud y la longitud de la posición. Por lo tanto, en la función dicha información debe transformarse a coordenadas cartesianas. Dichos datos se incluyen en la función *update* para actualizar la posición del vehículo en el código.

Una vez creadas las diferentes funciones *Callback*, ya tenemos toda la información leída de los sensores de forma correcta. El siguiente paso es transmitir nuestra información al vehículo. Dentro del bucle principal del programa es donde se calcula el ángulo de volante



que se debe girar y se regula la velocidad a la que se debe ir. Es por eso por lo que debe ser en ese mismo bucle donde se envíe esa información al vehículo.

```
pub.publish(speed)
```

La estructura es más simple y solo es necesario definir la variable como un *Publisher*. De esta forma, el vehículo deberá ser el que se suscribe a este nodo para poder recibir la información.

ROS funciona únicamente en *Linux*, por lo que es necesario utilizar dicho sistema operativo. Más concretamente, se ha utilizado *Ubuntu 18.04 LTS Version*, la cual goza de soporte actualmente.

Una vez realizado el tutorial introductorio de ROS para establecer los directorios de trabajo, será necesario empezar a incluir todos los archivos.

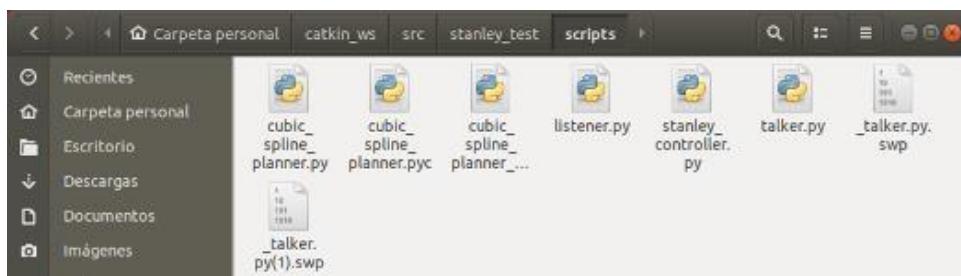


Figura 62: Archivos que se deben encontrar en la carpeta de scripts

Al realizar el tutorial introductorio, se creará automáticamente la carpeta *Catkin_ws*, que es el espacio de trabajo de ROS. Dentro, se crearán tres carpetas: *Bild*, *Devel* y *Src*, será dentro de esta última donde introduciremos la carpeta del código, que a su vez contendrá tres subcarpetas: *Include*, *Scripts* y *src*. Dentro de *Scripts* es donde incluiremos los archivos de *Python*, Figura 62.

Para poder poner en funcionamiento el programa, se debe abrir el *Terminal* de *Linux*. En una ventana debe ejecutarse *roscore*, que es el master que permite conectar los nodos entre sí. Después, en otra ventana, deben ejecutarse los siguientes códigos, Figura 63.

```
inigo@inigo-VirtualBox:~$ cd catkin_ws
inigo@inigo-VirtualBox:~/catkin_ws$ source ./devel/setup.bash
inigo@inigo-VirtualBox:~/catkin_ws$ rosrun stanley_test_coche stanley_controller
new.py --file /home/inigo/Documentos/maps/insia/demo_den.rou
```

Figura 63: Códigos para ejecutar el programa en ROS

Rosrun es el comando que se utiliza para correr programas por medio de ROS. Después se debe indicar el paquete que se quiere ejecutar y a continuación el archivo. Adicionalmente, se incluye un comando que es el de cargar el circuito. Para ello se debe poner la ruta del mapa en el ordenador.



De esta forma ya está todo listo. Cabe recalcar que todo esto debe realizarse en el ordenador del propio vehículo y es ahí donde se le deben dar permisos para que controle el volante y el acelerador.

Para las pruebas reales se contó con un vehículo Mitsubishi IMIEV, proporcionado por el INSIA, el cual se encuentra totalmente automatizado, por lo que sólo debíamos introducir el algoritmo y conectarlo a las diferentes partes del vehículo. Dicho vehículo se puede observar en la Figura 64.



Figura 64: Mitsubishi IMIEV utilizado en las pruebas reales

Sin embargo, las primeras pruebas no fueron satisfactorias. Se dieron numerosos problemas para conectar de forma correcta el vehículo al programa y se dieron numerosos errores en la lectura de los sensores. Una vez subsanados, se pudo iniciar. Nuevamente, se dio otro error. El vehículo al comenzar la trayectoria le exigía al volante girar el máximo posible todo el rato, lo que significaba irse contra el muro.

Este comportamiento trajo verdaderos quebraderos de cabeza. Ya que constantemente que se probaba el vehículo no seguía la trayectoria que debía. Poder solucionar errores en un sistema real no es tan fácil como hacerlo en una simulación. No se puede repetir el ensayo infinitas veces cambiando pequeños parámetros y vuelta a empezar. Es por eso por lo que fue necesario trabajar con una nueva herramienta que proporciona ROS, el *rosbag*.

Rosbag es un comando que permite guardar todas las variables que se han realizado durante una prueba. Esto significa que guarda todas las señales que se han comunicado por medio de los *Subscriber* y los *Publisher* y permite realizar la misma prueba, pero sin estar en el vehículo, lo que vendría a ser como una “simulación”.



De esta forma, tras la primera prueba fallida, llegó un tiempo de pruebas por medio de *rosbag*. Para poder observar el archivo, es necesario abrir una tercera ventana en el *Terminal* y en ella ejecutar el siguiente comando, **Figura 65**.

```
intigo@intigo-VirtualBox:~/Documentos/tests$ rosbag play sae_2020-02-24-10-52-28.bag
```

Figura 65: Comando para ejecutar un *rosbag*

Al hacerlo, veremos en la misma ventana múltiples datos y un contador de tiempo que muestra los segundos del ensayo real. Por lo tanto, debemos ir de nuevo a la segunda ventana y ejecutar el mismo código que cuando se probó el vehículo real. Es necesario señalar que, para visualizar los datos, fue necesario añadir código para la visualización de gráficas. En ellas, dibujamos la trayectoria seguida y todos los puntos del GPS. Además, por medio de diversos *print* trabajamos observando qué valores tomaban según qué variables.

Sin lugar a duda, el siguiente paso fue uno de los que más complejidad tuvo. Del ensayo contábamos únicamente con unos pocos segundos de simulación, en los que se observaba cómo el vehículo comenzaba correctamente, pero al cabo de unos segundos se perdía completamente.

La búsqueda del error y la prueba de diversas hipótesis fue un proceso largo y tedioso que duró algo más de dos meses. Finalmente, se observó que uno de los posibles focos de error era el código que permitía realizar lazos en el circuito. Debido a que la trayectoria que se probó iba y volvía por el mismo punto, se creaba un conflicto junto al código que permitía incorporarse a una trayectoria a la mitad. De esta forma, el algoritmo creía que se encontraba en la parte final del trayecto y buscaba darse la vuelta, ya que está en una dirección equivocada. Es por eso por lo que cuando se hizo la prueba el vehículo pedía un giro de volante demasiado elevado, entre otras cosas. Esta última modificación, junto a otras varias dejaron el código listo para otra prueba.

El 24 de Febrero de 2020 realizamos la primera prueba satisfactoria del código en un vehículo real. Las condiciones del día eran de cielo despejado y carretera seca, por lo que no se esperaban comportamientos anómalos de ese estilo. Se contaba con un sensor GPS situado en el eje delantero del vehículo y al tratarse de un ensayo a cielo abierto no se esperaban grandes errores en el establecimiento de la posición del vehículo. El vehículo fue capaz de realizar la trayectoria exigida, aunque no la pudo completar de la mejor forma posible, como se muestra en la siguiente **Figura 66**.

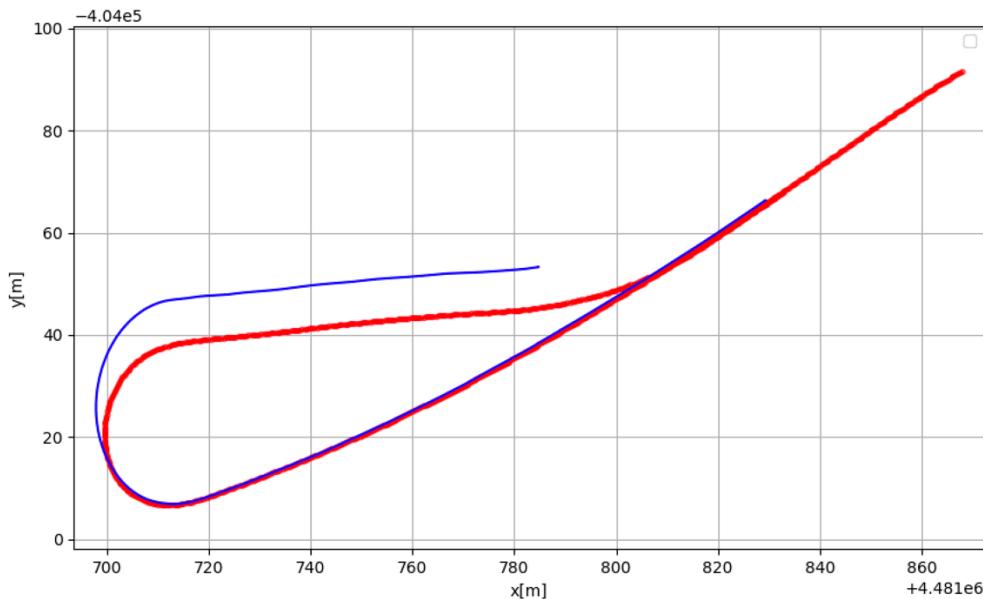


Figura 66: Trayectoria seguida en la prueba real

El vehículo comenzó cerca de la trayectoria, pero no en ésta exactamente. De ahí fue capaz de seguir la línea correctamente hasta llegar a la curva. Una vez ahí, comenzó la curva de forma correcta pero como se puede observar en la Figura 66 se desvió de la trayectoria. Esto se debe a que se impuso una limitación al giro del volante de 180º para evitar dar volantazos. Sin embargo, como se puede observar, fue esta limitación la que hizo que no se pudiese tomar una curva igual de cerrada que la que se quería. A partir de ahí el programa fue acumulando un error hasta pasó la curva.

A partir de la salida de la curva, se esperaba que el algoritmo trabajase para situar al vehículo encima de la curva, ya que no tenía que hacer un giro tan cerrado esa vez. Sin embargo, el programa consideró que se encontraba en la trayectoria y realizó un recorrido paralelo al que debería.

En las siguientes figuras se muestran algunas imágenes tomadas durante la realización del ensayo.



Figura 67: Inicio del ensayo real

En la Figura 67, se puede observar el inicio del ensayo. En la parte derecha se puede observar el ordenador, que es el que nos mostraba por pantalla diferentes datos con los que nos podíamos asegurar del correcto funcionamiento del algoritmo.



Figura 68: Fotograma de la prueba real durante la toma de una curva

En la Figura 68, se puede observar el giro del volante realizado por la orden del vehículo. Es notable destacar que el volante se encuentra girando 180°, que fue el valor límite que se le impuso a éste para las primeras pruebas. La imposición de este límite se comentará más adelante.



Figura 69: Fotograma de la prueba real al final del ensayo

En la Figura 69, se puede ver el final del ensayo. Como se ve, el vehículo iba dirigido hacia el césped, algo que desde luego no era el comportamiento esperado. Este fotograma corresponde a la parte del ensayo donde circula en paralelo a la ruta.

Por lo tanto, se observa que la prueba fue un éxito en parte. Por un lado, el algoritmo funcionó y permitió que el vehículo siguiese una ruta de forma fiel. Debido a un error nuestro al colocar la limitación de volante tan baja, hizo que el ensayo no fuese satisfactorio.

No obstante, el hecho de que el vehículo circulase de forma paralela a la vía, pero a una distancia considerable de esta era un error lo suficientemente grave como para darnos a entender que algo del código no funcionaba bien. Por lo tanto, tras la prueba se volvió a trabajar con *rosbag* y la grabación realizada ese día.

En primer lugar, nos centramos en analizar la parte del recorrido que se realizó de forma correcta, esto es, el inicio, junto a la recta y la entrada en curva. Se trabajó asilando ese recorrido, obteniendo lo siguiente, **Figura 70**.

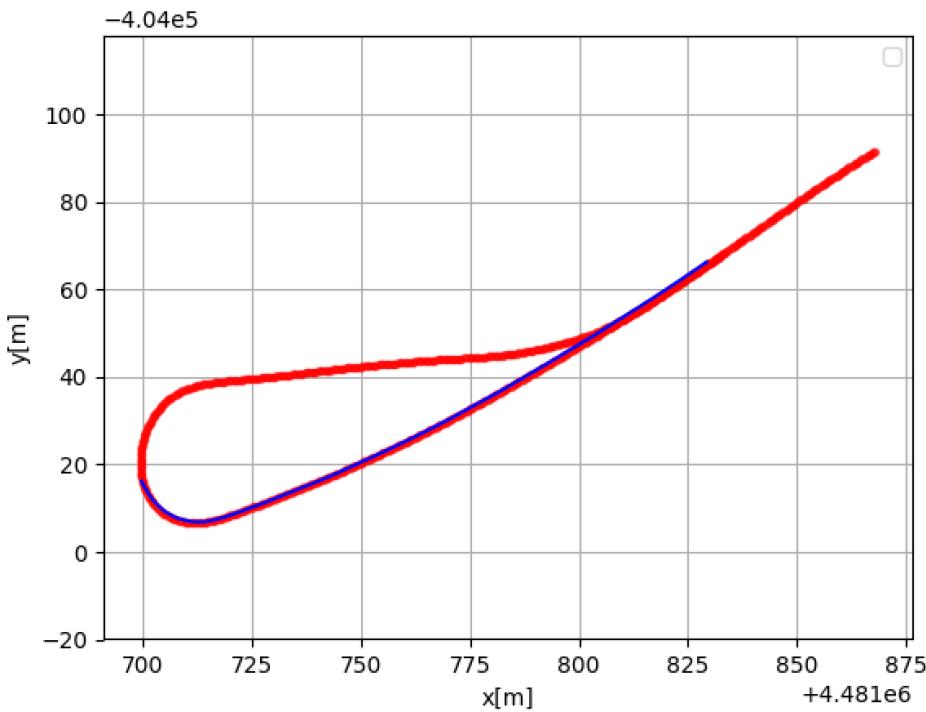


Figura 70: Parte de la trayectoria de la prueba real seguida correctamente

Para observar cómo funcionaba el algoritmo, obtuvimos el error de posición para cada ciclo, que es la diferencia en distancia entre el eje delantero del vehículo y el punto más cercano de la ruta. Sacamos por gráfica los puntos y obtuvimos lo siguiente, Figura 71.

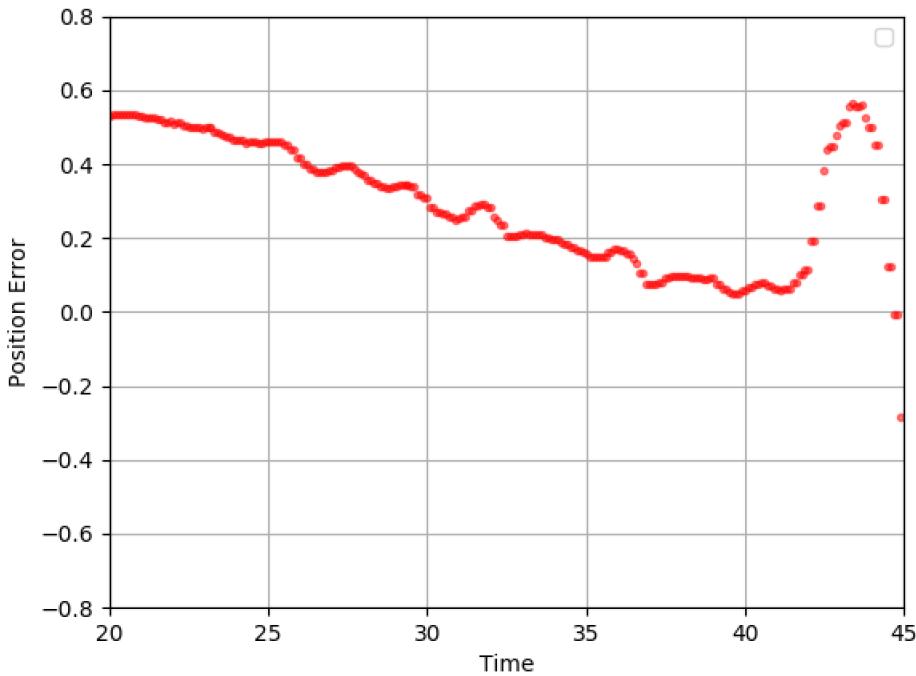


Figura 71: Error de posición del vehículo en la prueba real en la parte de la trayectoria seguida de forma correcta



El ensayo tuvo una duración de aproximadamente 60 segundos. El vehículo comienza a moverse en el segundo 20, que es cuando se empieza a tener en cuenta el error. En el segundo 45 es cuando el vehículo no es capaz de seguir la ruta debido a la limitación del volante.

Se observa como el ensayo comienza con un error de posición inicial de más de medio metro. Esto se debe a que el vehículo comenzó a medio metro de distancia de la ruta. Sin embargo, a medida que avanza el tiempo, se puede ver que el error va decreciendo, lo que muestra que el algoritmo funciona correctamente y va aproximándose a la ruta y por lo tanto reduciendo el error. En el segundo 40, el vehículo se encuentra prácticamente en la ruta. Después cuando comienza el giro vemos un pico en la gráfica, lo cual es normal, ya que el vehículo reacciona al cambio de dirección y pese a que en un inicio se desvía ligeramente, después rectifica y de nuevo disminuye el error. Lamentablemente, ahí es cuando tiene el giro limitado y el vehículo empieza a ver aumentado el error por no poder girar lo suficiente.

4.4.3 MODIFICACIONES TRAS LA PRUEBA REAL

Una vez comprobado que el algoritmo funciona correctamente, es necesario comprobar qué parte del código provocaba que el vehículo circulase de forma paralela, pero sin aproximarse a la trayectoria. Se analizó el *rosbag* y las numerosas variables que se tienen en cuenta para ver si podía darse el caso de que se ignorase esa parte de la ruta y únicamente fuese recto o si ignoraba el *crosstrack error*. Tras diversas pruebas, se observó que el error surgía por el *crosstrack error*.

Como ya se comentó al hablar del funcionamiento del control Stanley, éste se basaba en la minimización de dos errores: el *heading error* y el *crosstrack error*. Ambos errores había que reflejarlos como un ángulo de volante que corrigiese dicha situación. El *heading error* al ser un error de ángulo, la corrección en ángulo de rueda era simplemente el opuesto. El *crosstrack error* es un error de distancia, por lo que fue necesario convertirlo a ángulo. Recordando lo explicado anteriormente, la transformación era la siguiente.

$$\delta_{crosstrack} = \tan^{-1}\left(\frac{k \cdot cross_track_error}{v}\right)$$

La *k* se utilizó para determinar la brusquedad del giro. Cuanto mayor era el giro, el algoritmo reaccionaba con mayores ángulos a la diferencia de posición lateral. Si a una misma velocidad y posición, la *k* era 1, $\delta_{crosstrack}$ era menor que si *k* era un valor mayor.

Cuando se realizaron las primeras pruebas reales, se observaba que el vehículo realizaba movimientos muy grandes constantemente, de un lado para el otro, lo que daba una

sensación de inestabilidad. Es por eso por lo que se optó por reducir el valor de dicha k , con lo que se vio que la respuesta era más relajada y en un principio se supuso que eso sería correcto.

No obstante, poner un valor para k demasiado bajo tenía un problema que no vimos hasta la última prueba. Ese valor tan bajo hacía que la aportación del giro de volante gracias al *crosstrack* error fuese muy pequeña y por lo tanto si en algún punto *heading* y *crosstrack* emitían órdenes diferentes, esto es, el *crosstrack* decía de girar a la izquierda porque el vehículo se encontraba a la derecha de la pista, pero el *heading* pedía girar a la derecha porque el coche estaba mal orientado, por la actuación de la k predominaba por mucho la orden proveniente del *heading*.

De esta forma, si el vehículo estaba orientado al igual que la trayectoria, pero se encontraba separado de ésta, el vehículo tendería a acercarse a esta. Sin embargo, al hacerlo, el *heading* error notaba que la dirección que se tenía no era la correcta y mandaba la comanda opuesta.

Esto explica por qué en el trayecto que el vehículo circuló correctamente, se estabiliza en 0,1 m y no minimiza el error. También explica las variaciones de pendiente del error a lo largo de su descenso, ya que se encontraban difiriendo ambos errores, pero terminaba predominando uno.

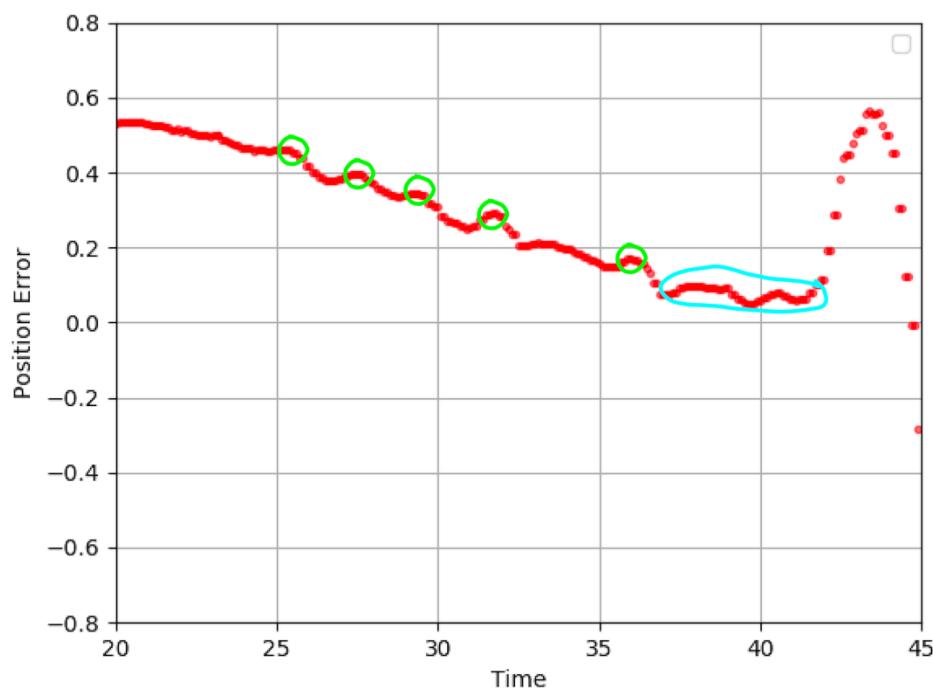


Figura 72: Análisis del error de posición en la prueba real



En la **Figura 72** se pueden observar los picos provocados por el enfrentamiento entre ambas órdenes (verde). Además, se muestra cómo antes del giro ya se produjo el hecho de que el vehículo circulase en paralelo a la vía, pero sin aproximarse completamente (azul).

Introducir esta corrección en el valor de la k , conllevaba otra modificación. Al tener un valor más grande, lo más seguro es que mandase ángulos de giro de volante más bruscos, como ya vimos en ensayos anteriores. Ejemplo de esto era pasar de 0 a 100º de volante en un ciclo, lo que eran movimientos demasiado bruscos.

Por lo tanto, lo siguiente fue controlar esas órdenes de volante. Se pensaron en diversas maneras de hacerlo, pero al final se optó por una muy simple pero efectiva. Se impondría un giro máximo de volante por ciclo, por lo que, si el vehículo recibía órdenes de un giro demasiado grande, lo realizaría en varios ciclos. Eso recortaba algo la precisión, pero era necesario para evitar volantazos.

Estos últimos cambios no se pudieron probar en una nueva prueba real, debido a la crisis sanitaria a la que nos enfrentamos. Se consideró que los buenos resultados en simulación serían una muestra del correcto funcionamiento y se pasó el enfoque a otras partes del código.

Había más partes del código que se podían perfilar sin necesidad de realizar pruebas. Por lo tanto, se cambió de enfoque por la parte de control. Ya se tenía un código que era capaz de seguir una ruta que el vehículo ya conocía de antemano de forma correcta. Así que el siguiente paso era tratar de asemejarlo al caso real.

En la prueba de la competición no se conocería la posición de los conos de antemano, por lo que la ruta que se tomaba en la parte de control no se conocería completamente, si no únicamente la ruta cercana y por delante al vehículo.

Para ello, en primer lugar, se tomó una ruta conocida y se creó una función para segmentarla y leer únicamente los tres primeros puntos a partir del vehículo. Así, cada vez que pasaba de un punto al siguiente, admitía uno más en su lista. El principal objetivo de esto fue comprobar que la creación de una ruta en cada ciclo funcionaba correctamente y no saturaba el sistema, lo cual fue demostrado. A continuación, se creó otra función para que, en vez de tomar los datos de una lista conocida, fuesen unos datos que se leyesen de otro nodo, más concretamente de los de detección y que serían los conos detectados y el camino a seguir, de forma que se fuese cerrando el camino poco a poco, pero a una velocidad suficiente para que el vehículo pudiese circular correctamente.



4.5 CONCLUSIONES

Tras el proceso seguido de simulación y objetivo de lograr una prueba en un vehículo real, se ha logrado un algoritmo que es capaz de seguir de manera correcta diferentes trayectorias y ateniéndose a diferentes situaciones como pueden ser curvas, empezar el recorrido por la mitad del trayecto o empezar desviado y retomar la ruta correctamente.

Tras numerosas pruebas de simulación se ha analizado la precisión del algoritmo y se ha observado que posee un error muy bajo de posición a lo largo del circuito. Con la posibilidad de la realización de las pruebas reales, se ha podido avanzar en un terreno muy complicado en el que la solución de los errores resulta mucho más compleja que en la simulación. Pese a que la última prueba no fue completamente satisfactoria, se puede concluir que el algoritmo funciona correctamente en una situación real y los errores que se detectaron se pueden considerar subsanados. De esta forma se espera que la próxima vez que se pruebe el algoritmo en un vehículo, la prueba se complete sin errores inesperados.

Así pues, se definen las siguientes conclusiones a los objetivos propuestos:

- Se ha logrado establecer un control longitudinal de velocidad constante, así como un sistema de frenada que actúa cuando el vehículo ha finalizado el recorrido, deteniéndolo progresivamente.
- Se ha logrado establecer un control lateral del vehículo por medio de actuación sobre el volante y se ha probado en numerosas trayectorias, tanto en simulación como en pruebas reales, observando que su funcionamiento es correcto.
- Se ha logrado crear una trayectoria que ha de seguir el algoritmo de control para permanecer dentro del circuito. Esta trayectoria nace a partir de los conos detectados.
- Se ha leído de forma satisfactoria la información proporcionada por el coche, necesaria para conocer el estado del vehículo y que el algoritmo funcione correctamente.
- Se ha sincronizado toda la información, tanto enviada como recibida por el vehículo para que todo funcione a una misma frecuencia, de forma que no se solapen instrucciones de un comando con mediciones de otro distinto.



5. CONCLUSIONES DEL PROYECTO Y DIFICULTADES ENCONTRADAS

A continuación, se exponen las diferentes conclusiones que han sido extraídas a lo largo de la realización del proyecto, así como las dificultades encontradas durante la duración completa de este:

- **Definición del alcance:** para el proyecto se partió sin base sólida de años anteriores sobre la cuál poder sustentarse. Pese a existir un trabajo realizado por un grupo de Ingeniería dos años anterior, los métodos empleados en el desarrollo del vehículo autónomo diferían en gran medida de aquellos métodos propuestos y utilizados por los equipos con mejores resultados en competiciones anteriores de formula student Driverless. Al ser nuestra primera vez en un proyecto de tales dimensiones y no tener demasiado conocimiento inicial sobre la detección por LiDAR o el control del vehículo, el hecho de que los papers de los mejores participantes de la competición tuvieran un planteamiento distinto al proyecto INGENIA anterior, complicó la definición de los objetivos del proyecto en un primer momento, así como su alcance y los caminos a seguir. Se comenzó tratando de abarcar todas las fases del vehículo al mismo tiempo, sin rumbo definido, en lugar de partir de una idea sencilla e ir progresando a partir de ella, estrategia que se implantó más adelante.
- **Codificar en Python:** para la realización de los códigos se eligió el lenguaje Python. Este cuenta con gran soporte a nivel mundial y resulta más intuitivo para su aprendizaje que C++. Sin embargo, sólo uno de los cuatro integrantes contaba con nociones básicas de este y los otros tres ninguna. Supuso dos retos el proceso de aprendizaje de este para comenzar a implementar los diferentes algoritmos que componen las fases tanto de detección como de control realizadas en el proyecto:
 - El mayor reto de este nuevo lenguaje fue comprender perfectamente el mundo de las librerías, entender que la mejor forma de trabajar es a partir de módulos ya creados y adaptándolos a la aplicación propia del proyecto.
 - Por otro lado, existen diferentes entornos de desarrollo de Python, también llamados IDEs. La búsqueda del mejor IDE que proporcionase la mayor flexibilidad y, además, facilidad de programación nos resultó un complicado.

Gracias al proyecto se ha aprendido en gran medida el funcionamiento de este lenguaje, llegando, finalmente, a grandes resultados. Sin embargo, existió algún



caso donde se bloqueó la implantación de algún algoritmo por el desconocimiento acerca de la metodología para codificarlo. Esto se produjo tanto en la fase de realización del algoritmo de filtrado de suelo, donde se partió de dos algoritmos y únicamente fuimos capaces de implantar uno de ellos, como en la fase de clustering, donde se planteó un problema de las mismas características dando por perdida una segunda versión de esta que prometía mayor rendimiento a igual precisión.

- **Detección de Conos:** La segunda dificultad fue la de encontrar el método que ofreciera los mejores resultados para la detección de los conos. En un principio se planteó proceder mediante *machine learning*, pero tras dos semanas de estudio se decidió pasar a la detección por código, ya que encontramos muy difícil el aprendizaje de las redes neuronales y su aplicación al proyecto.
- **Control del vehículo:** existen numerosos métodos para lograr el control del vehículo, por lo que buscar un algoritmo que fuese preciso y asegurase un seguimiento constante fue necesario. Además dicho algoritmo debería ser posible de programar y funcionar en un sistema real.
- **Trabajo con sistemas reales:** la inexperiencia en trabajar con sistemas reales supuso más barreras de las esperadas. No se tenía noción de qué características que daban por hechas en una simulación no se podían extrapolar a un caso real. El error acumulado y plantear un buen posicionamiento del vehículo fueron cruciales. Además, para cada ensayo real era necesario tener todo muy bien controlado, para que no sucediese ningún fallo que pudiese tener consecuencias graves. El tiempo requerido para cada ensayo era mucho mayor que en una simulación, por lo que se alargaba el proceso mucho.
- **Aprendizaje de ROS|ROS2:** para el proyecto se ha aprendido tanto el funcionamiento de ROS como el de ROS2, al existir una migración existente entre ambas versiones y una indefinición clara acerca de cuál es óptimo utilizar, con diferentes ventajas e inconvenientes. A esto se suma el aprendizaje de Linux, tanto para ROS donde resulta obligatorio como para ROS2, donde la implantación en Windows planteó barreras infranqueables por falta de desarrollo de la versión. Finalmente se ha conseguido una base en ambas versiones, logrando realizar sistemas en tiempo real en ambas e incluso sistemas que comuniquen ROS con ROS2.
- **Confinamiento:** El confinamiento planteó un nuevo reto, al no contar con el dispositivo Lidar para hacer tests reales de los algoritmos ni con un vehículo para realizar tests del algoritmo de control. En el caso del Lidar, se logró encontrar un dataset equivalente al de la competición que resultó de gran utilidad y permitió realizar tests de los algoritmos equivalentes a los que podrían haber sido realizados



POLITÉCNICA

UNIVERSIDAD
POLITÉCNICA
DE MADRID



en las instalaciones del INSIA. En el caso del control del vehículo, llegó en el momento en el que se estaban haciendo más progresiones en el sistema real del vehículo, lo que supuso un gran jarro de agua fría. Todos los avances se paralizaron y no se pudo continuar con esa parte, por lo que fue necesario avanzar con otras diferentes y aparcar ese módulo



6. LÍNEAS FUTURAS

Se plantean algunas líneas futuras que, según cual sea, pretenden ampliar el proyecto para alcanzar nuevos objetivos o, en otros casos, mejorar la eficiencia y eficacia del proyecto.

6.1 Líneas futuras enfocadas a la mejora de eficiencia y eficacia de lo ya realizado:

- Simulación mejorada: Se plantea poder mejorar la simulación de ROS ya que, actualmente, la simulación se está llevando a cabo en máquinas virtuales y con ordenadores portátils de bajo consumo. El objetivo de cambiar la simulación a un hardware mejor y ejecutarla directamente en el sistema operativo es el de mejorar la rapidez de cálculo y con ello la eficacia del código.
- Optimización de los códigos: En segundo lugar, se plantea la optimización de los códigos programados en Python. Esto tiene dos objetivos importantes dentro del proyecto:
 - 1) La simplicidad de éste para que su ejecución consuma menos recursos y sea más rápido de ejecutar y, por ende, se necesite menos potencia de hardware para ejecutarlos.
 - 2) La optimización de los parámetros de cada módulo, para optimizar su precisión. Esta mejora se haría mediante varias pruebas con múltiples *datasets*, para conseguir la mayor precisión posible ajustando los parámetros a los valores que mayor mejores resultados ofrecieran.
- Mejora de la precisión: respecto a los resultados obtenidos en la parte de control, se quiere mejorar la precisión obtenida en el ensayo real por medio de ajuste de diversas variables, así como una estimación del estado del vehículo más precisa.

6.2 Líneas futuras enfocadas a la ampliación del proyecto:

- Detección de posición de los conos: Habría un módulo intermedio entre la detección de los conos y el control. Este algoritmo debería indicar la posición de los conos, identificando si se encuentra a la derecha o la izquierda del vehículo, para que Control pudiera entender por dónde tiene que conducir al vehículo. Para este módulo, se programaría un algoritmo de aprendizaje automático supervisado (Supervised Machine Learning). “En una tarea supervisada, se puede medir inmediatamente qué tan bien se desempeña el modelo en los datos de entrenamiento, porque se proporcionan los resultados óptimos, los objetivos” [32].
- Fusión del LiDAR y la cámara: El paso siguiente a la detección de conos con LiDAR es el de realizar la fusión de esta salida junto con la detección de conos con cámara. Esto implica desafíos debidos a problemas de nivel de adquisición de



datos, que incluyen: diferencias en las unidades físicas de medida, diferencias en las resoluciones de muestreo y diferencias en la alineación espaciotemporal [33]. La fusión puede ser de dos tipos: intrínseca o extrínseca. Se considera la idea de realizar una fusión extrínseca, mediante un proceso de transformación de un cuerpo rígido entre los sistemas de coordenadas de los dos sensores [34]. La calibración extrínseca se realiza con un objeto externo, para hacer coincidir las correspondencias entre los dos sensores.

- Acoplamiento de la parte de detección con la parte de control: La integración de los dos módulos grandes que componen al proyecto se deberá hacer de forma que las salidas de Detección sean lo más limpias y precisas posibles para que Control pueda trabajar sin problemas con ellas.
- State Estimation: se quiere realizar un buen método para conocer la posición del vehículo de manera más precisa, utilizando varios sensores y fusionando la información entre ellos. Actualmente sólo se utiliza GPS, lo que disminuye la precisión adquirida y no permite ajustarse a la trayectoria de forma tan precisa como aporta el algoritmo, desaprovechándolo, por lo tanto.
- Generación de trayectoria robusta: se quiere crear un algoritmo que a partir de un número de conos diferente entre izquierda y derecha sea capaz de proporcionar una ruta que circule entre ambos.



7. REFERENCIAS

- [1] «AMZ,» ETH Zürich, [En línea]. Available: <http://driverless.amzracing.ch/>. [Último acceso: 2020 febrero 11].
- [2] I. B. Manterola, E. G. Sola y V. S. Prieto, «Validación del LIDAR y simulación de un vehículo autónomo de competición,» de *Asignatura INGENIA: Ingeniería de automoción. Diseño, fabricación, ensayo y demostración de un vehículo para la competición Formula SAE*, Madrid, 2018, p. 17.
- [3] A. R. López, «LiDAR cone detection as part of a perception system in a Formula Student Car,» Barcelona, Universitat Politècnica de Catalunya, 2019.
- [4] D. Zermas, N. Papanikolopoulos y I. Izzat, «Fast Segmentation of 3D Point Clouds: A Paradigm on LiDAR Data for,» ResearchGate, 2017.
- [5] M.-. Himmelsbach, H.-J. Wuensche y F. v. Hundelshausen, «Fast Segmentation of 3D Point Clouds for Ground Vehicles,» IEEE, 2010.
- [6] N. Gosala, A. Böhler, M. Prajapat, C. Ehmke, M. Gupta, R. Sivanesan, A. Gawel, M. Pfeiffer, M. Böhurki, I. Sa y R. D. a. R. Siegwart, «Redundant Perception and State Estimation for Reliable Autonomous Racing,» 2018.
- [7] C. Guindel, J. Beltrán, D. Martín y F. García, «Automatic Extrinsic Calibration for Lidar-Stereo,» 2017.
- [8] I. Clarke, «OXTS,» [En línea]. Available: <https://support.oxts.com/hc/en-us/articles/115004181525-Hardware-integration-with-Velodyne-LiDAR>. [Último acceso: 2020 mayo 06].
- [9] «<https://pointcloudlibrary.github.io/documentation/>,» Creative Commons Attribution 3.0. [En línea].
- [10] D. Zermas y I. a. P. N. Izzat, «Fast Segmentation of 3D Point Clouds: A Paradigm on LiDAR Data for Autonomous Vehicle Applications.,» 10.1109/ICRA.2017.7989591., 2017.
- [11] «<https://github.com/heremaps/pptk>,» HERE Europe B.V, 2011-2018. [En línea].
- [12] M. M. B. H.-P. K. J. S. Mihael Ankerst, «OPTICS: Ordering Points To Identify the Clustering Structure,» de *Management of Data*, Philadelphia PA, 1999.
- [13] S. Bhattacharyya, «<https://towardsdatascience.com/dbscan-algorithm-complete-guide-and-application-with-python-scikit-learn-d690cbae4c5d>,» Towards Data Science, 9 Junio 2019. [En línea].



- [14] M. Ester, H.-P. Kriegel, J. Sander y X. Xu, «A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise,» de *AAAI*, 1996.
- [15] ScikitLearn, «<https://scikit-learn.org/stable/>,» Scikit, 2007-2019. [En línea].
- [16] Project Nayuki, «nayuki.io,» 20 06 2018. [En línea]. Available: <https://www.nayuki.io/page/smallest-enclosing-circle>. [Último acceso: 01 04 2018].
- [17] BasqueGeek, «<https://geekgasteiz.wordpress.com/2018/11/01/ros2-vs-ros-1-migramos/>,» GeekGasteiz, 1 Noviembre 2018. [En línea].
- [18] «<https://index.ros.org/doc/ros2/>,» ros-infrastructure. [En línea].
- [19] «<https://answers.ros.org/questions/>,» ros-infrastructure. [En línea].
- [20] «<https://www.virtualbox.org/>,» ORACLE. [En línea].
- [21] J. Whitley, «<https://github.com/ros-drivers/velodyne/tree/dashing-devel>,» The Autoware Foundation, Noviembre 2019. [En línea].
- [22] «<https://www.paraview.org/veloview/>,» ParaView. [En línea].
- [23] «<https://github.com/ros2/examples/tree/master/rclpy>,» ros-infrastructure. [En línea].
- [24] «https://docs.ros.org/api/sensor_msgs/html/point__cloud2_8py_source.html,» ros-infrastructure, 2008. [En línea].
- [25] I. Georgiev, «<https://gitlab.com/eufs/datasets>,» Edinburgh University Formula Student, 2020. [En línea].
- [26] «https://github.com/ros2/rosbag2_bag_v2/blob/master/README.md,» ros-infrastructure. [En línea].
- [27] J. Levinson y S. Thrun, «Automatic Online Calibration of Cameras and Lasers,» Stanford Artificial Intelligence Laboratory, Standford.
- [28] J. Domhof, J. F. P. y K. a. D. M. Gavrila, «An Extrinsic Calibration Tool for Radar, Camera and Lidar,» de *2019 International Conference on Robotics and Automation (ICRA)*, Montreal, Canada, 2019.
- [29] A. Dhall, K. Chelani, K. Vishnu Radhakrishnan y M. Krishna, «LiDAR-Camera Calibration using 3D-3D Point correspondences,» 2017.
- [30] M. Y. Yang y W. Förstner, «Plane Detection in Point Cloud Data,» 2010.
- [31] hctr, «https://github.com/htcr/plane-fitting/blob/master/fit_plane_LSE.py,» 2018. [En línea].



- [32] S. Hochreiter, «www.bioinf.jku.at,» Winter Semester 2014. [En línea]. Available: http://www.bioinf.jku.at/teaching/current/ws_mlstvl/ML_supervised.pdf.
- [33] V. D. Silva, J. Roche y A. Kondoz, «[arxiv.org](https://arxiv.org/ftp/arxiv/papers/1710/1710.06230.pdf),» 15 08 2018. [En línea]. Available: <https://arxiv.org/ftp/arxiv/papers/1710/1710.06230.pdf>. [Último acceso: 01 05 2020].
- [34] J. Li, «FUSION OF LIDAR 3D POINTS CLOUD WITH 2D DIGITAL CAMERA IMAGE,» Oakland, 2015.
- [35] M. I. Valls, H. F. Hendrikx, V. J. Reijgwart, F. V. Meier, I. Sa, R. Dubé, A. Gawel, M. B. Siegwart y Roland, «Design of an Autonomous Racecar: Perception, State Estimation and,» AMZ.