Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

Máster Universitario en
Ingeniaría Industrial

Trabajo de Fin de Máster

Sistema de navegación autónoma para un vehículo eléctrico.

Autor/a: Adrián Martín Gutiérrez

Tutor/a: Jonay Tomas Toledo Carrillo

9 de marzo de 2023

La publicación de este Trabajo Fin de Máster solo implica que el estudiante ha obtenido al menos la nota mínima exigida para superar la asignatura correspondiente, no presupone que su contenido sea correcto, aunque si aplicable. En este sentido, la ULL no posee ningún tipo de responsabilidad hacia terceros por la aplicación total o parcial de los resultados obtenidos en este trabajo. También pone en conocimiento del lector que, según la ley de protección intelectual, los resultados son propiedad intelectual del alumno, siempre y cuando se haya procedido a los registros de propiedad intelectual o solicitud de patentes correspondientes con fecha anterior a su publicación

Agradecimientos.

Quería expresar mi más sincero agradecimiento a todas las personas que han sido parte de mi proyecto y han contribuido a su éxito. En primer lugar, quiero agradecer a Jonay por haber sido mi tutor y por toda la ayuda que me ha brindado durante el desarrollo del proyecto. Su orientación y consejos han sido invaluables. También quiero agradecer a mi compañero Carlos por su apoyo en cada punto del proyecto.

Agradezco profundamente a mi familia por su incondicional apoyo y motivación en todo momento, han sido un gran pilar en mi vida y en este proyecto en particular.

No puedo dejar de mencionar al equipo de FSULL, del cual he formado parte durante mis años en la universidad de La Laguna. A todos los que han pasado por el equipo durante estos años quiero agradecerles por su colaboración y ayuda en este proyecto, y por todo lo que me han enseñado. De manera especial, quiero dar las gracias a mis compañeros Aaron y Carlos, y a los profesores Carmelo, Viana, Andres, Vanesa y nuevamente a Jonay por su contribución y ayuda no solo en el equipo de FSULL si no en toda mi etapa Universitaria y lograr que se pudiera construir el primer prototipo de Formula Student de la Universidad, un sueño que tuve desde primero de carrera.

Por último, pero no menos importante, quiero expresar mi gratitud a mis amigos, quienes han estado a mi lado durante estos años y sin su apoyo, no habría sido capaz de completar esta etapa de mi vida.

Licencia

Resumen

Este proyecto se centra en el diseño de un sistema autónomo en ROS para que cumplimentara las pruebas de la competición de Formula Student Driverless. Este se trata de la continuación de mi TFG aplicado en un entorno real. Para su implementación se ha utilizado una silla de ruedas, ya que el vehículo de Formula Student de la Universidad de la Laguna no contaba con los sistemas de seguridad necesarios para probar este en él.

Para el desarrollo del sistema autónomo ha sido necesario las búsqueda y montaje de sensores como el LIDAR, los encoders de las ruedas, el controlador de los motores de la silla, etc. Esto nos han permitido desarrollar a través algoritmos implementados en nodos de ROS funcionalidades que se encarguen de la parte de Percepción, Localización, mapping, generación de trayectorias y control. Estos han sido desarrollados principalmente en C++ y algunos solo están implementados de forma teórica por falta de tiempo y problemas en el montaje.

El sistema diseñado nos permitió unos resultados positivos siendo bastante eficiente y preciso para la detección de colores. Para una implantación real en el vehículo sería necesario hacer varias mejoras y probar los algoritmos en situaciones reales.

Palabras clave: Sistema autónomo, ROS, silla de ruedas, Formula Student, Percepción, LIDAR, conos, colores, localización, SLAM, EKF, trayectorias, controlador PID.

Abstract

This project focuses on the design of an autonomous system in ROS to carry out the tests of the Formula Student Driverless competition. This is a continuation of my final degree project applied in a real environment. For its implementation, a wheelchair has been used, since the Formula Student vehicle of the University of La Laguna did not have the necessary safety systems to test it on.

The development of the autonomous system has required the search and assembly of sensors such as LIDAR, wheel encoders, wheelchair motor controller, etc. These have allowed us to develop functionalities through algorithms implemented in ROS nodes that take care of Perception, Localization, Mapping, Trajectory Generation, and Control. These have been developed mainly in C++ and some are only implemented theoretically due to time constraints and problems with assembly.

The designed system yielded positive results, being quite efficient and accurate for color detection. For a real implementation in the vehicle, several improvements would be necessary, and the algorithms would need to be tested in real situations.

Keywords: Autonomous system, ROS, wheelchair, Formula Student, Perception, LIDAR, cones, colors, localization, SLAM, EKF, trajectories, PID controller.

Índice general

Índice de Figuras

Capítulo 1

Introducción

El desarrollo de sistemas autónomos en vehículos es una de las áreas más interesantes y dinámicas en la industria automotriz. En este contexto, hemos decidido abordar el desafío de desarrollar sistemas autónomos para un vehículo de Formula Student de La Universidad de La Laguna. Sin embargo, debido a la falta de sistemas de seguridad adecuados en el coche, se ha optado por utilizar una silla de ruedas como primer acercamiento al sistema real.

Es importante destacar que este trabajo es la continuación de mi TFG el cual trataba principalmente en diseñar el sistema autónomo para realizar las pruebas en simulación a través de gazebo. Sin embargo, en este proyecto nos hemos centrado en diseñar un sistema de navegación y detección de conos que funcione en un entorno real y según la normativa de Formula Student. A parte de estos muchas partes también fueron desarrolladas en conjunto para un trabajo final de grado [1]

La implementación exitosa de estos sistemas en una silla de ruedas proporcionará un primer acercamiento al sistema real y permitirá evaluar su viabilidad en un entorno controlado antes de su aplicación en un vehículo de Formula Student. En resumen, este trabajo representa una oportunidad única para abordar el desafío de desarrollar sistemas autónomos en vehículos de forma real.

1.1. Formula Student

La Formula Student, también conocida como Fórmula SAE, es una competición entre estudiantes universitarios de todo el mundo, cuyo objetivo es promover la excelencia en el campo de la ingeniería. Los miembros de cada equipo deben diseñar, construir y desarrollar un monoplaza, ya sea eléctrico o de combustión. Tiene como objetivo fomentar el trabajo en equipo y la colaboración interdepartamental de las distintas áreas de conocimiento para el diseño, fabricación, gestión económica y marketing de un vehículo de competición.

La competición nació en Estados Unidos como Fórmula SAE a partir de la *"Society of Automotive Engineers"*. La primera edición se realizó en 1980 y a partir de ahí se extiende hasta el día de hoy en el que existen competiciones por todo el mundo (China, Australia, España, Brasil, etc.) y cuenta con más de 700 equipos de universidades de todo el mundo. En el año 2010 se añade la categoría eléctrica. Posteriormente en 2017 nació la categoría de coche autónomo, además a partir de 2022 todos los vehículos deben ser eléctricos y participar en dos pruebas dinámicas de forma autónoma. Podemos dividir la competición en 4 etapas:

1. **Pruebas Estáticas:** Se encuentran divididas en 3 eventos principales.
    - *Design Event*. Consiste en realizar una presentación de las características

Figura 1.1: Panorámica de los equipos participantes en Formula Student Spain 2022, Fuente: Formula Student Spain (`https://www.formulastudent.es/`)

e innovaciones de los monoplazas.

- *Cost and manufacturing*. En este evento se presentan todos los costes relativos al vehículo.
- *Business Plan*. Se presenta un plan de negocio simulando que se quiere vender el coche a un equipo de competición.

2. ***Scrutineering.*** Durante esta fase se realizan pruebas a todos los vehículos para comprobar que cumplen con la normativa y sean seguros para poder competir en las pruebas dinámicas. Estas verificaciones no son iguales para todos los coches por lo que usaremos distintos acrónimos. DV para los vehículos autónomos, CV vehículos de combustión, EV Vehículo eléctrico, DV-CV Vehículo autónomo de combustión y DV-EV Vehículo autónomo eléctrico. Estas pruebas son las siguientes:

   - *Pre-Inspection*.
   - *Accumulator Inspection*(DV-EV y EV).
   - *Electrical Inspection* (DV-EV y EV).
   - *Mechanical Inspection (DV, CV y EV)*.
   - *Driverless Inspection* (DV).
   - *Tilt* (DV, CV y EV).
   - *Rain Test* (DV-EV y EV).
   - *Noise Test* (CV y DV-CV).
   - *Brake Test* (CV, DV y EV).

3. **Las Pruebas Dinámicas:** Son las pruebas donde se mide el rendimiento y el funcionamiento del vehículo. Son las siguientes:

   - *Skidpad* (DV, CV y EV). Es una prueba que se realiza en un circuito en forma de 8 y que se usa para medir la eficiencia del chasis y las suspensiones.
   - *Acceleration* (DV, CV y EV). Esta prueba se usa para medir la aceleración de los monoplazas.
   - *Autocross* (DV, CV y EV). Esta consiste en dar una vuelta a un circuito de 1.5 km. Sobre todo, se usa para medir la velocidad de cada monoplaza a una vuelta.
   - *Endurance* (CV y EV). Es una prueba de resistencia en la que se deben recorrer aproximadamente 20 Km en el menor tiempo posible.
   - *Trackdrive* (DV). En esta prueba el vehículo debe completar 10 vueltas a un circuito de forma completamente autónoma.
   - *Efficiency Event* (DV, CV y EV). En este test se mide la eficiencia de los vehículos de combustión y eléctricos durante la *Endurance* y los driverless en la *Trackdrive*.

4. ***Post Event Inspection***. Este evento se realiza justo después de la última prueba y se verifica que el coche sigue cumpliendo con la normativa.

1.1.1. Características de los eventos driverless

Los eventos realizados en la categoría driverless tienen características que los diferencian de los eventos de las demás categorías. Estas diferencias son de gran importancia ya que influyen directamente en el diseño del sistema autónomo y se encuentran descritos en la Reglas [5] y el Handbook de la competición [4]. Empezaremos por las marcas que deberán llevar los circuitos cuyas características son las que vemos a continuación:

- La pista está marcada con conos.
- Los bordes izquierdos de la pista están marcados con pequeños conos azules.
- Los bordes derechos de la pista están marcados con pequeños conos amarillos.
- Los carriles de salida y entrada están marcados con pequeños conos naranjas.
- Se colocarán grandes conos naranjas antes y después de las líneas de inicio, finalización y cronometraje.
- Si no se define lo contrario, la distancia máxima entre dos conos en la dirección de conducción es de 5 m. En las curvas, la distancia entre los conos es menor para una mejor indicación.
- Las líneas de inicio, finalización y cronometraje, así como las zonas alejadas del equipo de mantenimiento del tiempo, están marcadas con pintura roja, naranja o rosa.

- Además, para el *Skidpad* y el *Trackdrive*, las líneas de límite de pista a cada lado de la pista y los carriles de entrada/salida pueden marcarse con pintura amarilla, verde o blanca.

- No hay líneas de límite de seguimiento para la aceleración y la prueba del Sistema de frenos de emergencia (EBS).

- Los equipos se les darán datos acerca del circuito.

- Los conos usados durante la competición los podemos ver en la tabla 2.2 y son del fabricante Wemas.

			
Cono grande naranja	Cono pequeño naranja	Cono azul	Cono amarillos

Tabla 1.1: Tipos de conos [4]

Para superar cada evento dinámico es necesario que el vehículo siga un procedimiento para cada una de las pruebas, tal y como vemos a continuación:

- **Procedimiento *Skidpad***: Cada equipo tendrá dos intentos. El vehículo comenzará 15 m detrás de la línea cronometraje. La señal de *GO* enviada desde el RES(Remote Emergency System) dará la aprobación para comenzar. El vehículo entra perpendicular a la figura de ocho y dará una vuelta completa en el círculo de la derecha. La próxima vuelta continuará en esta y será cronometrada. Inmediatamente después de la segunda vuelta, el vehículo entrará en el círculo izquierdo y realizará el mismo proceso descrito anteriormente. Inmediatamente después de la segunda vuelta el vehículo saldrá del circuito y deberá realizar una parada completa 25 m después de la línea de cronometraje. El circuito donde de esta prueba lo podemos ver en la figura 2.3.

- **Procedimiento *Acceleration***: Cada equipo tendrá dos intentos. El vehículo comenzará 0.30 m detrás de la línea cronometrada. La señal de *GO* enviada desde el RES dará la aprobación para comenzar. El tiempo comenzará una vez cruce la línea de cronometraje se parará después de la línea de final, tendrá 100 m para realizar la parada completa. EL circuito lo podemos ver en la Figura 2.4.

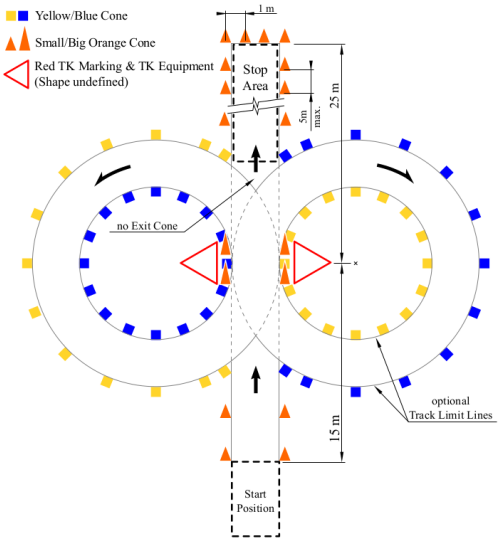


Figura 1.2: *Skidpad* configuración base [4]

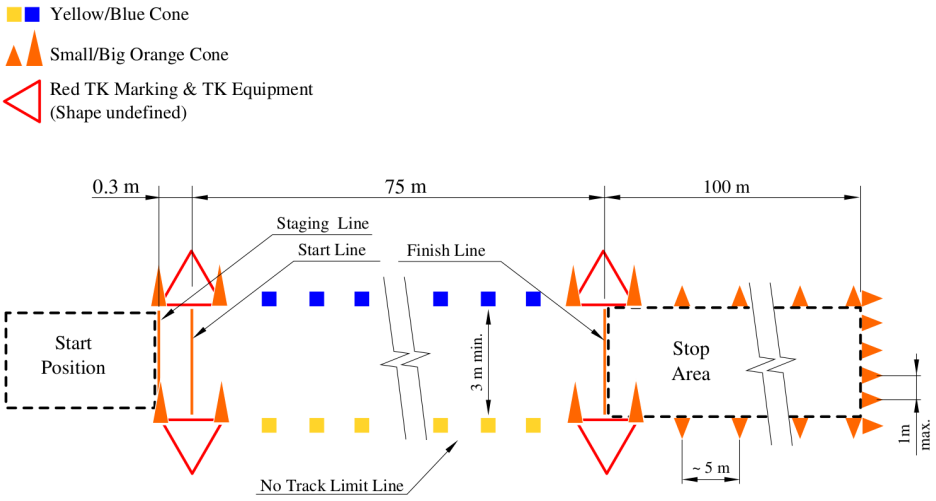


Figura 1.3: *Acceleration* configuración base [4]

- **Procedimiento *Autocross***: Cada equipo puede reconocer el circuito caminando pero sin equipamiento, además los datos que se obtengan antes del primer intento están completamente prohibidos. El vehículo comenzará 6m detrás de la línea cronometraje. La señal de *GO* enviada desde el RES dará la aprobación para comenzar. El tiempo comenzará una vez cruce la línea de cronometraje y realizará una vuelta al circuito, realizando una parada completa 30m después de la línea de meta.

- **Procedimiento *Trackdrive***: Cada equipo puede reconocer el circuito de la misma manera que el *Autocross* . La señal de *GO* SE envía desde el RES y dará la condición para que el monoplaza empiece. El tiempo comenzará una vez cruce la línea de cronometraje y realizará 10 vueltas al circuito. Una vez las

complete deberá realizar una parada completa 30 metros después de la línea de meta. El vehículo puede ser descalificado si la velocidad media durante las primeras 3 vueltas es inferior a 2.5 m/s o en sí en las demás es inferior a 3.5 m/s.

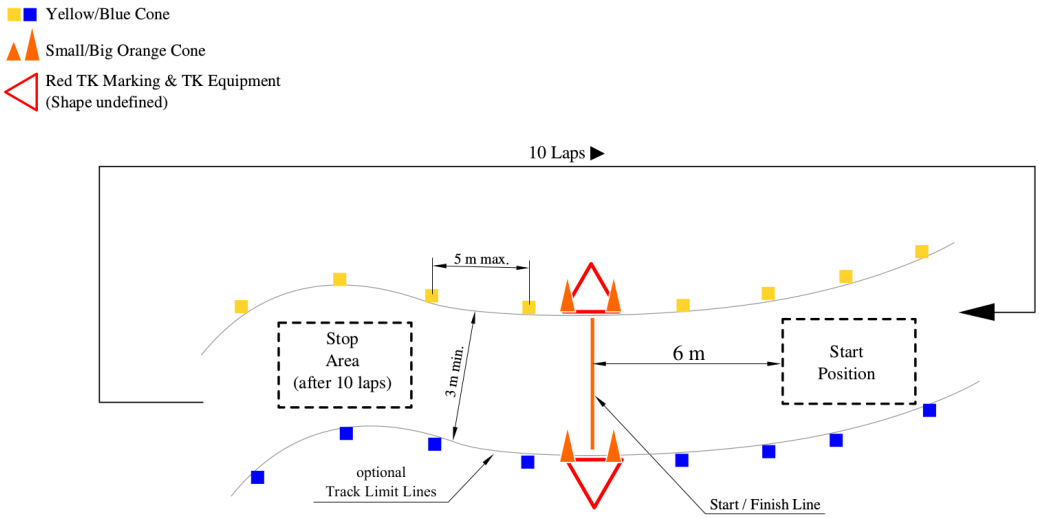


Figura 1.4: *Trackdrive* configuración base [4]

1.2. Formula Student Universidad de La Laguna (FSULL-Dynamics)

El equipo de Formula Student nació en 2017, con la motivación de ser el primer equipo de Canarias en participar en esta competición. Durante el año 2022 el equipo participó en su primera competición, la cual fue Formula Student Spain celebrada en el Circuito de Castellolí.

El equipo actualmente está formado por más de 15 miembros de diferentes grados y másteres de la Universidad de La Laguna, aunque en su mayoría pertenecen a los grados de Ingeniería Electrónica Industrial y Automática e Ingeniería mecánica.

El proyecto se encuentra en la fase de mejora para el año 2023 y se espera volver a asistir a las competición de Formula Student Spain. Los objetivos del equipo son:

- Realizar mejoras del vehículo claves en su rendimiento.
- Ampliar los conocimientos que adquirimos en los grados y másteres de la universidad.
- Concienciar de un respetuoso uso del medio ambiente.
- Lograr otra gran participación en el año 2023.
- Completar la totalidad de las pruebas dinámicas y estáticas durante la competición.

La idea principal de este proyecto dentro del equipo es desarrollar un sistema autónomo el cual pueda ser usado por el equipo para poder participar en las pruebas referentes al apartado driveless de Formula Student.

Figura 1.5: Primer coche del equipo de FSULL-Dynamics, Fuente:FSULL-Dynamics

1.3. Silla Vermerien

Para la implementación del sistema autónomo, ahora mismo el monoplaza del equipo no cuenta con los sistemas de seguridad adecuados y su implementación de primeras en este sería una tarea altamente compleja. Por eso, se decidió que como primera toma de contacto con el sistema se empezará la implementación del sistema autónomo en la realidad con la silla de ruedas Vermerien. Esta nos da unas características que nos permiten realizar un buen acercamiento al modelo del coche,que se utilizara para sistema autónomo final.

1.4. Objetivos del proyecto

Este trabajo tiene como objetivo la implementación de sistemas de percepción, localizaciones y control que nos permitan implementar un sistema autónomo para la silla Vermerien. Para ello, ha sido importante que tenga las mismas características que el que irá en el vehículo final y nos permitiría realizar las pruebas dinámicas de los vehículos driverless. Para hacer el diseño, ha sido necesario implementar un sistema de percepción, algoritmos para la localización, mapping, planificación de trayectorias, y un sistema de control para que el vehículo siga la ruta establecida. Además ha sido desarrollado en ROS (Robotic Operating System), utilizando características del simulador FSSIM (`https://github.com/AMZ-Driverless/fssim`) del equipo suizo AMZ de la Universidad de Zúrich.

Figura 1.6: Silla Vermerien, Fuente:FSULL-Dynamics

1.4.1. Requerimientos

- El vehículo debe ser capaz de trabajar con la información de los sensores y generar respuestas adecuadas para completar los objetivos de cada prueba.
- Los circuitos están construidos con conos azules en el lado izquierdo, amarillos en el derecho y en la línea de salida naranjas.
- El sistema de percepción deberá distinguir entre los conos naranjas, azules y amarillos que delimitaran el circuito.
- El sistema de mapping deberá posicionar los conos detectados en los lugares donde se encuentran.
- El planificador de trayectorias locales debe permitir al vehículo dar una vuelta completa al circuito de *Autocross* y *Trackdrive* para completar dichos eventos.
- El sistema de control deberá permitir el movimiento del vehículo para completar los eventos correctamente.
- El vehículo deberá tener una CPU (Unidad central de procesamiento) y una GPU (Unidad de Procesamiento Gráfico) capaz de procesar la información recibida de los sensores y enviar una respuesta acorde.

1.4.2. Especificaciones

- Para la percepción, el vehículo usará un LIDAR (Laser Imaging Detection And Ranging)
- Para determinar su posición, el vehículo dispondrá de un medidor de velocidad en cada una de las ruedas(WSS).
- Los sensores y actuadores se alimentan a 24V.

- Se usará el modelo de un robot diferencial para este primer acercamiento, que es el más similar para la silla de ruedas.

Capítulo 2

Tecnologías utilizadas

2.1. ROS (Robotic Operating System)

El Robot Operating System (ROS) [11] que como su propio nombre indica, es un sistema operativo, con una estructura flexible usada para programar el software de un robot. Está compuesta de una colección de herramientas y librerías que tienen como objetivo simplificar la tarea de crear un sistema complejo y robusto en una amplia variedad de robots.

Figura 2.1: Logo ROS [11]

La característica principal de este sistema, es permitirnos trabajar con procesos en paralelo de manera síncrona y asíncrona. Para ello, utiliza una estructura de maestros, nodos, mensajes y tópicos. En la Figura 2.2 podemos ver un ejemplo de ésta.

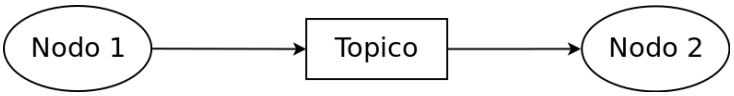


Figura 2.2: Ejemplo de cómo es la estructura de ROS

Para el diseño del Sistema Autónomo se decidió usar ROS por varios motivos:

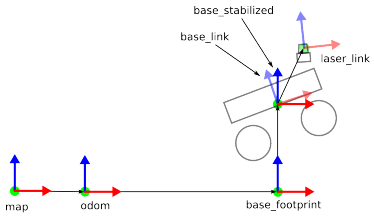
- Trabaja con sistemas de tiempo real.
- Presenta un entorno multilenguaje, es decir, permite programar en distintos lenguajes.
- Su arquitectura permite trabajar de forma modular, dedicando cada nodo a una función y comunicándolos entre sí. Esto ayuda a simplificar la complejidad del sistema.
- Contiene un sistema de mensajes IDL (Lenguaje de descripción de interfaz).
- Tiene herramientas muy útiles para el desarrollo de cualquier robot, como:
    - Rviz [13]. Ofrece la visualización de la información de los sensores y el modelo del coche en URDF (Unified Robot Description Format) en tres dimensiones. Nos permite analizar todos los datos de los sensores, lo que ayuda a detectar malas configuraciones de estos. En la Figura 2.3a podemos ver su logo.

- Rosserial. Es un protocolo para usar mensajes, tópicos, nodos y servicios de ROS usando un dispositivo que se comunique con la CPU a través de un puerto serie. De esta manera podemos usar placas como Arduino para el control o lecturas de sensores de un Robot.
- Gazebo [6]. Es un simulador en 3D que nos permite realizar pruebas con una representación virtual de nuestro robot sin tener la necesidad de disponer el real. En la Figura 2.3b podemos ver su logo.
- Librería Transform [8]: Nos permite coordinar y actualizar todas las transformadas de otros sistemas de referencia (tf y tf2). En la Figura 2.3d podemos ver un ejemplo.
- Point cloud Library (PCL) [14]. Es una librería con la que podemos realizar tareas de procesamiento de nubes de puntos y procesamiento de geometrías en 3D, lo cual es muy útil de cara a obtener la información de LIDAR. En la Figura 2.3c podemos ver su logo.
- rqt [12]. Es una estructura que tiene varias herramientas que nos permiten desde ver los nodos y tópicos que son publicados y por quién, hasta hacer gráficas con información de sensores. En la Figura 2.3e podemos ver un ejemplo.

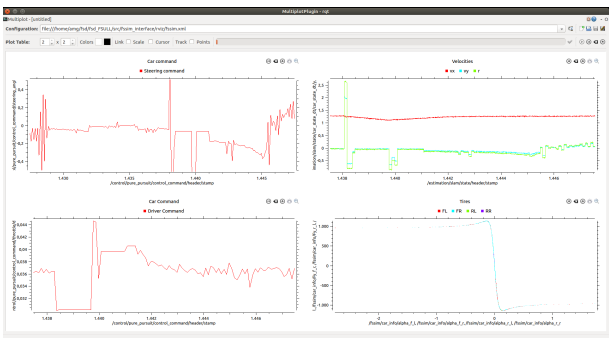
(a) Logo Rviz [13]

(b) Logo Gazebo [6]

(c) Logo PCL [14]



(d) Ejemplo tf [8]



(e) Ejemplo rqt [12]

Figura 2.3: Herramientas ROS

Para el aprendizaje de este entorno se usó el Libro *Learning ROS for Robotics Programming*[3].

2.2. Controlador Sabertooth

El Controlador Sabertooth es un driver usado para dos motores y que nos permite hasta picos de 50 A. Ademas se comunica por serial, lo que nos permite un uso fácil para el control de los motores.

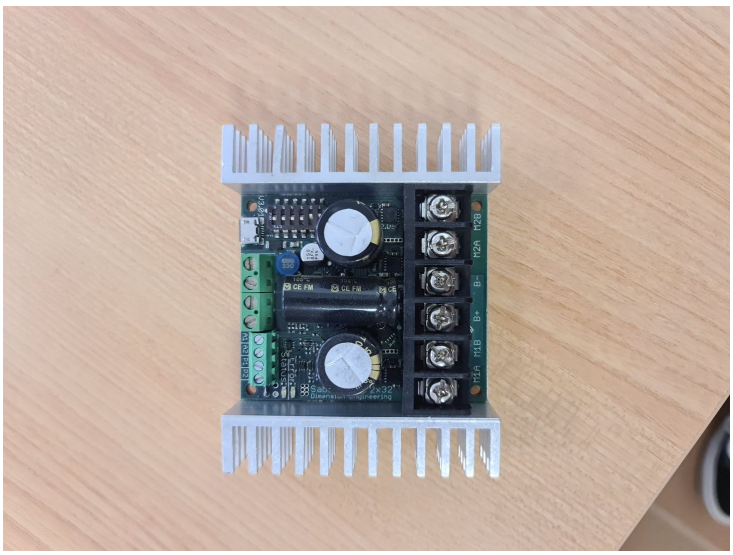


Figura 2.4: Placa de control Sabertooth

2.3. FSSIM

FSSIM (Formula Student SIMulator) es un simulador para vehículos autónomos de Fórmula Student. Fue desarrollado por el equipo suizo AMZ, de la Universidad de Zúrich. Ahora mismo este equipo es el mejor en la categoría de Formula Student Driverless.

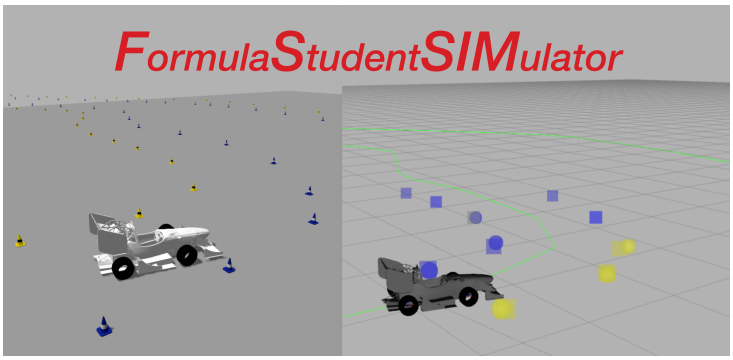


Figura 2.5: Logo FSSIM [7]

Para mi TFG se usó este simulador como base del sistema, debido a que nos permitía

realiza algunas aproximaciones que nos permiten centrarnos en desarrollar un primer sistema autónomo simplificado. Para este proyecto se usaron algunas funciones y librerías ya que al final es la base primaria de nuestro sistema autónomo.

2.4. EUFS Autonomous Simulation

EUFS Autonomous Simulation es un simulador para vehículos de Formula Student Driverless creado por el equipo de la Universidad de Edimburgo. Utiliza, principalmente, para simular el modelo ADS-DV, el cual se presta durante la competición de Formula Student UK, para que los equipos compitan con él y puedan probar así él sistema autónomo de su coche. En la Figura 2.6 lo podemos ver.

Figura 2.6: Modelo ADS-DV, Fuente: Formula Student UK(`https://www.imeche.org/events/formula-student`

Capítulo 3

Monoplaza y Silla Vermerien

Antes de describir cómo se implementó el sistema autónomo es importante conocer sobre lo que vamos a trabajar y saber así cuáles son las limitaciones y cómo debemos enfocar el sistema global. Por ello en este capítulo veremos la descripción del monoplaza fabricado por equipo de Formula Student de la ULL y la silla sobre la que se implementara el sistema.

3.1. Descripción del monoplaza

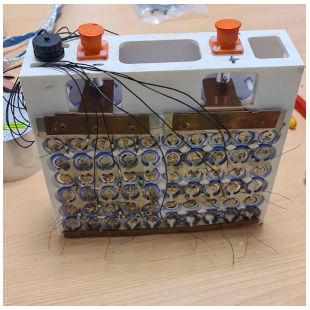
El vehículo será de tracción trasera y tiene un único motor de 80 KW, el Ermax 228, en la Figura 3.1a. Luego cuenta también con el inversor/controlador Bamocar D3 que lo podemos ver en la Figura 3.1b.

(a) Ermax 228, Fuente: FSULL-Dynamics

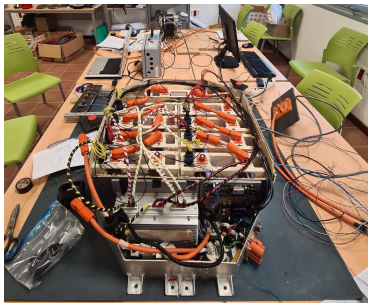
(b) Bamocar D3, Fuente: FSULL-Dynamics

Figura 3.1: Tren de potencia

Por otro lado, las baterías del coche disponen de las celdas 21700 de 4000mAh conectadas en una configuración de 108 en serie y 5 en paralelo, 540 en total, obteniendo así un almacenamiento de 7.7KWh. Aunque para este año la disposición cambiará teniendo ahora 108 serie y 5 paralelo.



(a) Pack de baterías



(b) Caja de baterías

Figura 3.2: Sistemas de baterías

Los sistemas electrónicos en el vehículo, los podemos dividir en dos:

- **Los sistemas de seguridad.** Son aquellos que se encargan de cumplir la normativa y hacer que el vehículo sea seguro. Un ejemplo de esto es el TSAL (Tractive System Active Light), que hace que se encienden unas luces que determinan si el bajo o el alto voltaje está activo, o el BMS que se encarga de la gestión de las baterías.
- **La telemetría.** Es la encargada de la obtención de los datos de los distintos sensores del monoplaza. Para la comunicación de estos datos se usa el protocolo Bus CAN. Es importante destacar que la ECU (Electronic Control Unit) está diseñada por el equipo a través de la ATMEGA328.

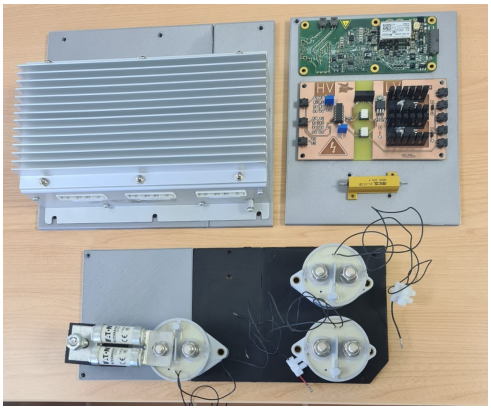


Figura 3.3: Circuito de seguridad, BMS, etc. Fuente: FSULL-Dynamics

Pasando ahora a la parte mecánica del vehículo, este incorpora un diferencial autoblocante y dispone de un chasis tubular. Además, los frenos son de disco flotante y las llantas de 10 pulgadas. En la Figura 3.4 podemos ver el ensamblaje de todos los componentes del vehículo, excepto la columna de dirección.

Figura 3.4: Ensamblaje de los componentes en el Chasis, Fuente: FSULL-Dynamics

La aerodinámica se hará en fibra de vidrio y para su diseño se busco que fuera lo más simple posible, ya que es el primer vehículo que se fabricaba. En la Figura 3.10 podemos ver una de las diversas simulaciones en SolidWorks que sirven para validar este diseño.

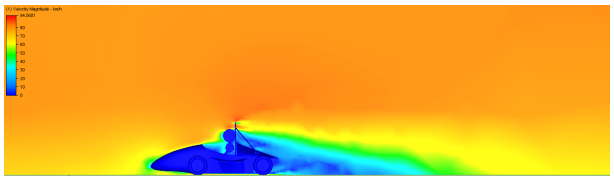


Figura 3.5: Simulación Aerodinámica con Piloto, Fuente: FSULL-Dynamics

Como características generales, nuestro vehículo tiene un peso final sin piloto de 280 Kg, tiene una distancia entre ruedas de 1.53m, un par máximo de 730N y un coeficiente de drag de 0.4 y de downforce de 0.1, con piloto.

Figura 3.6: Vehículo de FSULL-Dynamics para la temporada 21-22, Fuente: FSULL-Dynamics

3.1.1. Descripción del Hardware Sistema Autónomo en el monoplaza

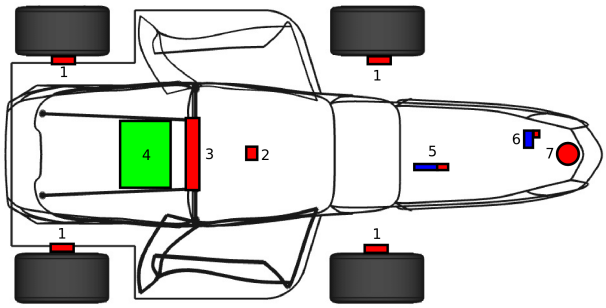


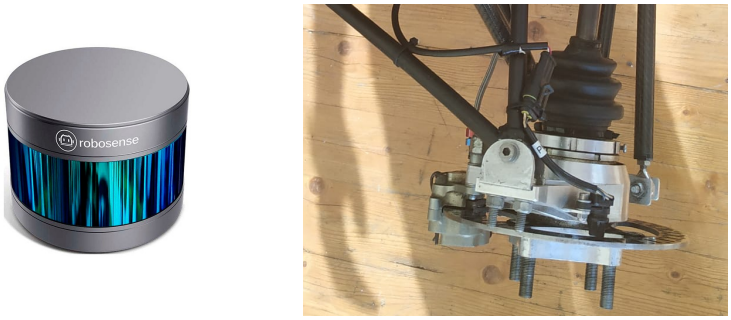
Figura 3.7: Hardware: 1 WSS, 2 IMU, 3 Cámaras estereoscópicas, 4 Driverless box (En su interior tiene la CPU y GPU), 5 Sistema de dirección, 6 Sistema de frenado, 7 LIDAR

- ## Actuadores

  - **Sistema de frenado** Para el sistema de frenos por normativa se usan cilindros neumáticos que nos permiten que el freno pueda hacer una parada de emergencia en menos de 10 m. La fuente de energía sería un cartucho de CO2. Además, esto nos permitiría que se pueda desconectar fácilmente para el modo de conducción manual.
  - **Sistema de dirección.** Para esto la mejor solución es montar paralelamente a la columna de dirección un servomotor. Debe tener una fuerza de, al menos, 30 Nm que es aproximadamente la fuerza que se necesita para mover el vehículo cuando está parado.

- ## Sensores

  - **LIDAR.** El LIDAR irá posicionado en la parte baja del morro delantero del vehículo y se usará para obtener la posición y color de los obstáculos en la pista. Se usará el RS-LIDAR-16 o el Velodyne HDL-32E pero esto lo veremos más adelante(Figura 3.8a)
  - **WSS (Wheel speed sensor).** Se utiliza también para el sistema de estimación. En nuestro caso el sensor no se colocará como vemos en la Figura 3.8b, si no que iría en los palieres que salen del del diferencia, en la parte trasera del vehículo. Esto se decidió así para tener una mejor resolución de cara a una mejor estimación de la posición del coche.



(a) RS-LiDAR-16[2] (b) Wheel speed sensor

Figura 3.8: Sensores para la estimación, Fuente: FSULL-Dynamics

  - **Encoder para el sistema de dirección.** Suele venir con el servomotor y se usa para cerrar el bucle de control y llegar a las posiciones de consigna que queramos según el sistema de control.
  - **Sensor de presión para el sistema de frenado.** Estos sensores se usarán con el mismo cometido que el anterior encoder, cerrar el bucle de control.

3.2. Descripción de la silla Vermerien

La silla usada como modelo para el vehículo ha sido la silla Vermerien Foster 3, la cual tiene dos baterías de plomo de 12v conectadas en series de 80 Ah. Además nos permite alcanzar una velocidad de hasta 30 km/h.

Figura 3.9: silla Vermerien Foster 3

3.2.1. Adaptación del modelo planteado para el monoplaza en la silla de ruedas

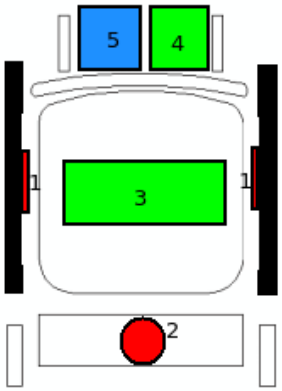


Figura 3.10: Hardware: 1 WSS, 2 LIDAR, 3 Driverless box (Ordenador portátil con GPU y CPU), 4 Arduino para lectura y estimación de posición , 5 Controlador de motores di.

Como la silla de ruedas es más similar al modelo de un robot móvil diferencial que el modelo de la bicicleta, que es el que suelen usar los vehículos, hay sensores y actuadores que se han descrito anteriormente que no hemos necesitado utilizan. Lo utilizado fue lo siguiente:

- **Sensores**
  - Lidar.

Figura 3.11: Montaje del LIDAR Robosense RS-LiDAR-16 en silla Vermerien

  - WSS.

Figura 3.12: Montaje de encoders en ruedas de la silla

- **Actuadores** Los únicos actuadores con los que contará la silla, son los dos motores de cada una de las ruedas, las cuales dentro de ellas contienen también el freno.

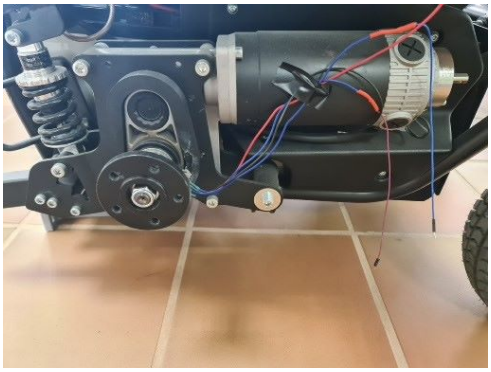


Figura 3.13: Motores eléctricos de la silla de ruedas

Por último, queremos destacar que hemos usado un ordenador portátil como la CPU y GPU principal de nuestro robot que va montado en una estructura que se fabricó para la silla. Esto lo podemos ver en más detalle en la Figura 3.14.

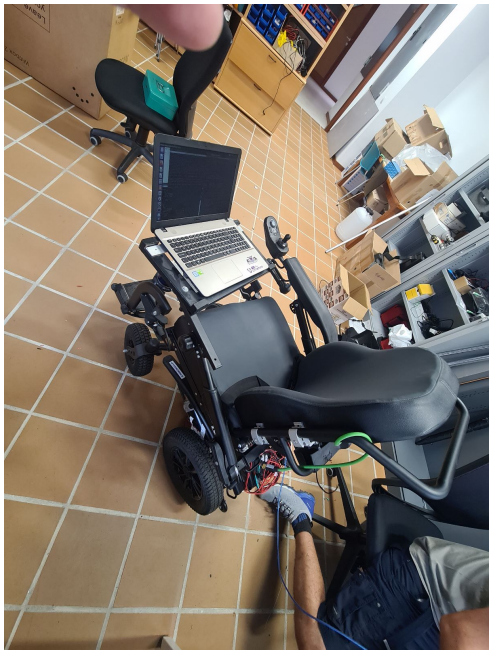


Figura 3.14: CPU y GPU de nuestro robot

Para las lecturas de los encoders(WSS) y mandar los comandos de control al Sabertooth y crear un nodo de ROS, se hizo a través de un Arduino mega.

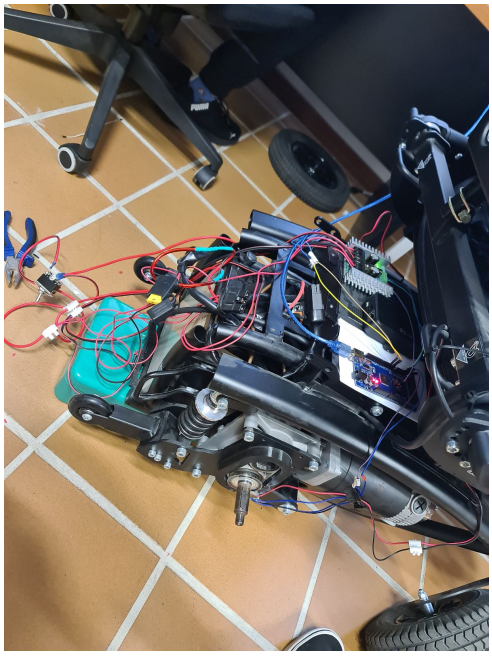


Figura 3.15: Montaje de sensores y placas de control en silla Vermerien

3.2.2. Software

Para conseguir una integración de los sensores y actuadores del vehículo se implementará un software que incluirá 3 bloques principales que trabajan en conjunto entre sí.

- **Percepción.** Obtiene la información del LIDAR la cual transforma en la posición y color de los obstáculos detectados. Esta información se enviará directamente al bloque de estimación. Se pueden ver más detalles de este apartado en el Capítulo 4.

- **Estimación y mapping.** Crear un mapa con las detecciones de la percepción y estimar la velocidad y posición del vehículo. Para cumplir con este propósito será necesario que apliquemos un estimador de velocidad. Además, nos hará falta un algoritmo de SLAM (Simultaneous Localization And Mapping), con el que obtendremos las posiciones de los obstáculos, a partir de las cuales se generarán las trayectorias. Este apartado lo podemos ver con más detalle en el Capítulo 5.

- **Control y planificador de trayectorias.** Este bloque tiene dos objetivos principales, por un lado, planificar la trayectoria y realizar una acción de control al vehículo para que siga el recorrido. Es importante destacar que el planificador global dispondrá de dos modos de funcionamiento. El primero está adecuado para el *Trackdrive* y *Autocross*, donde no se conocen de antemano las posiciones de los conos y el vehículo tiene que moverse primero con un planificador local e ir generando la ruta global.

  El otro modo está preparado para la *Skidpad* y el *Acceleration* donde los circuitos siempre son los mismos y directamente le enviamos la trayectoria global. En el Capítulo 6 veremos cómo se realiza la planificación de trayectorias, el tipo de *tracking* y control que se emplea.

  En general el planificador desarrollado está pensado principalmente para no conocer la trayectoria y generarla a partir de la posición de los conos.

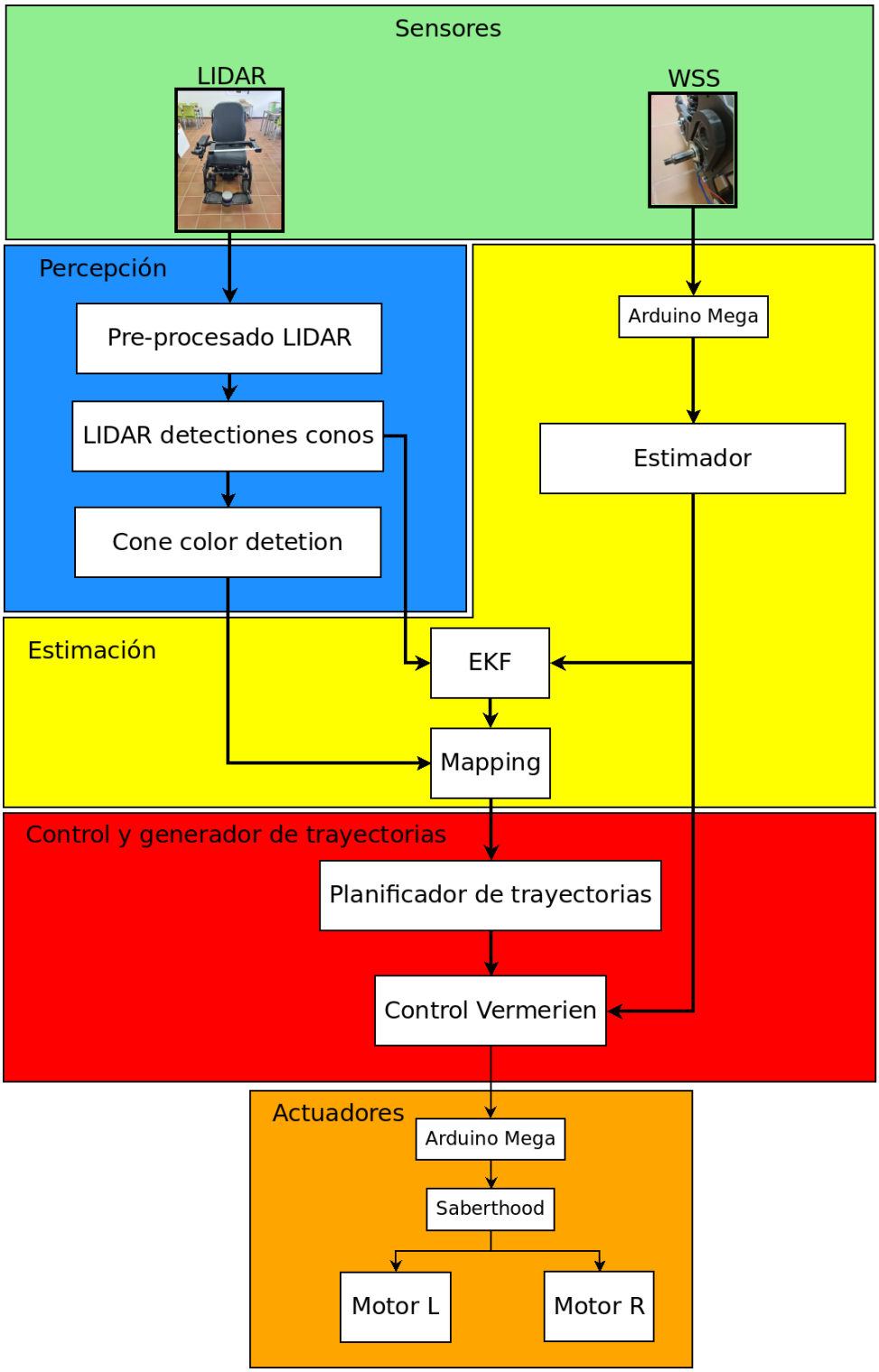


Figura 3.16: Arquitectura del software

Capítulo 4

Percepción

La detección de obstáculos es una función crítica para el desarrollo de sistemas autónomos en vehículos, ya que permite a estos sistemas evitar colisiones con objetos externos y garantizar la seguridad, robustez y rendimiento de nuestro sistema. En este capítulo, se explicará cómo se desarrolla la detección de obstáculos a través de un LIDAR (Light Detection and Ranging).

Un LIDAR es un sistema de sensores que utiliza pulsos láser para medir la distancia a objetos en su entorno. La detección de obstáculos con un LIDAR se basa en la medición de la distancia a los objetos y la generación de un mapa tridimensional de su entorno. Este mapa se utiliza para identificar obstáculos potenciales y calcular la trayectoria más segura para evitarlos.

El proceso de detección de obstáculos con un LIDAR generalmente consta de los siguientes pasos:

- Adquisición de datos: El LIDAR envía pulsos láser hacia el entorno y mide la distancia a los objetos mediante el tiempo de retorno de la luz reflejada.
- Procesamiento de datos: Los datos adquiridos se procesan para crear un mapa tridimensional del entorno. Este mapa se utiliza para identificar obstáculos potenciales. En este capítulo sobre todo nos centraremos en este punto.
- Planificación de trayectoria: La información obtenida sobre los obstáculos se utiliza para planificar la trayectoria más segura para evitarlos.

En nuestro caso utilizaremos el LIDAR para tres funciones principales que son, la detección de conos y sus respectivos colores y el SLAM(Simultaneous Localization and Mapping). También es importante destacar que la detección de obstáculos con un LIDAR no es infalible y puede verse afectada por factores como la interferencia ambiental, la presencia de objetos en movimiento y la resolución limitada del sensor. Por lo tanto, es importante que el procesamiento de los datos se haga de la manera más correcta posible y permita disminuir estos problemas que sobre todo nos afectarán a la hora de detección de colores.

4.1. Pre-procesamiento de datos

El pre-procesamiento de datos, consiste en aplicar una serie de filtros para eliminar la información que no es importante dentro de la detección. Nuestro principal objetivo en este paso ha sido quedarnos únicamente con las detecciones de conos, para lo cual se usan varios filtros que veremos a continuación.

Primero aplicamos un filtro voxel el cual consiste en agrupar puntos cercanos en un solo voxel (volumen virtual), y manteniendo un solo punto representativo de

este grupo. Esto se hace para reducir el tamaño de la nube de puntos y mejorar la eficiencia del procesamiento, ya que muchos algoritmos trabajan más eficientemente con un número menor de puntos.

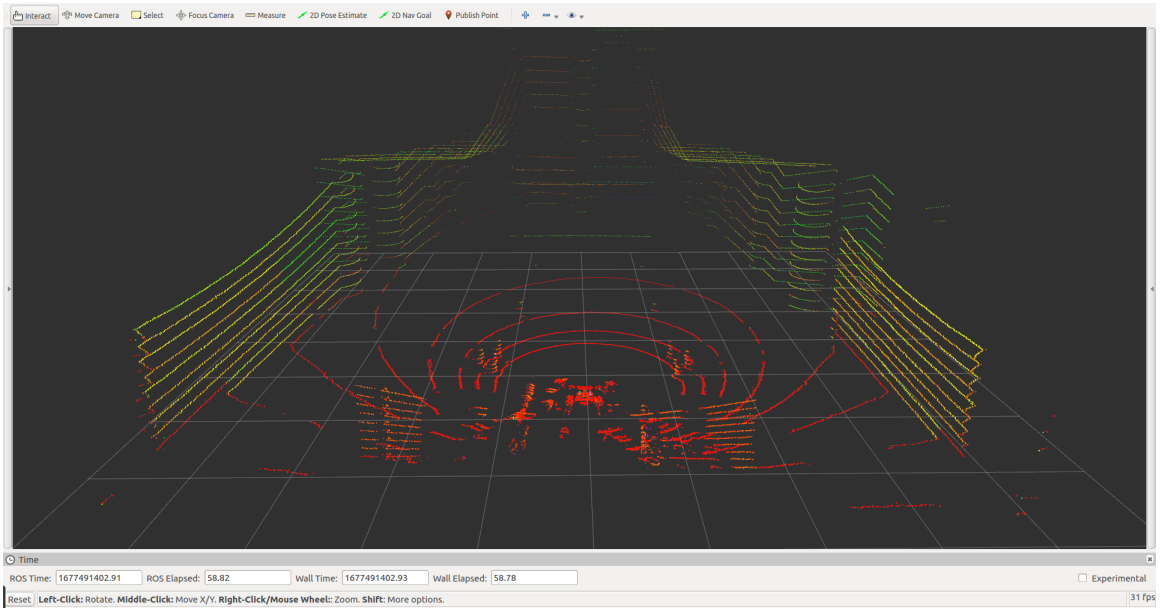


Figura 4.1: Lidar antes de preprocesado con RoboSense rs-lidar-16

Posteriormente, eliminamos la información no importante como las detecciones que hace el LIDAR. Para esto en cada punto, se aplica una serie de condiciones para determinar si el punto es elegible para ser incluido en un nueva nube de puntos. Estas condiciones básicamente es que se encuentre en la región requerida para hacer las detecciones y acotar las detecciones a una distancia máxima.

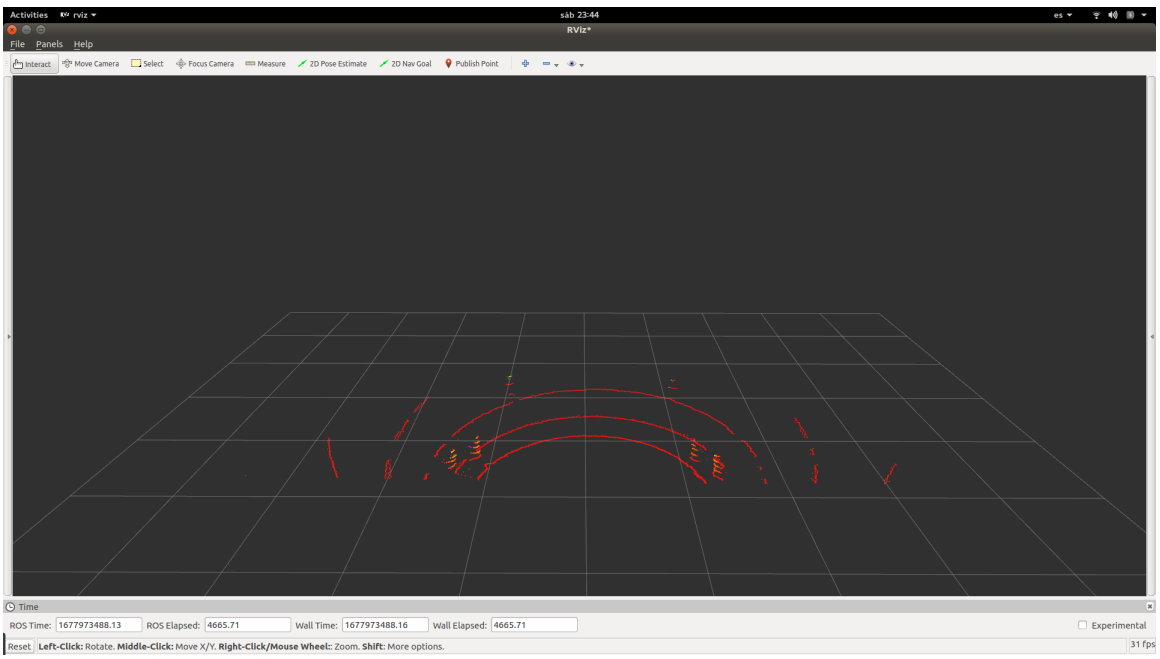


Figura 4.2: Eliminación de información no importante con RoboSense rs-lidar-16

Para lograr esto se desarrolló el código que vemos aquí:

```
void perception_handle::returnConePosition (const sensor_msgs::PointCloud2
    &lidar_value){
      pcl::PointCloud<pcl::PointXYZI> cloud_filtred;
      pcl::PointCloud<pcl::PointXYZI>::Ptr cloud1(new pcl::PointCloud<pcl
          ::PointXYZI>);
      pcl::fromROSMsg(lidar_value, cloud_filtred);
      for(int i = 0 ;i< cloud_filtred.points.size(); i ++){
      if(cloud_filtred.points[i].x > 0.8 && cloud_filtred.points[i].x*
          cloud_filtred.points[i].x + cloud_filtred.points[i].y*
          cloud_filtred.points[i].y < 25){
             cloud1->points.push_back(cloud_filtred.points[i]);
      }
      }
      .
      .
```

Luego eliminamos el suelo y cualquier pared cercana a las detecciones. Para esto utilizando el algoritmo SAC (Model-Based Random Sample Consensus) para segmentar un plano en una nube de puntos. Esto nos permite guardar los índices de los puntos en la nube de puntos, que cumplen con el modelo de plano establecido por el algoritmo SAC. Luego, se pueden extraer esos índices utilizando la clase "pcl::ExtractIndicesz se puede filtrar la nube de puntos original para obtener solo los puntos que cumplen con el modelo. Este algoritmo lo podemos implementado a continuación :

```
void perception_handle::returnConePosition (const sensor_msgs::PointCloud2
    &lidar_value){
      .
      .
      // Remove ground and wall
      pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients
          );
      pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
      pcl::SACSegmentation<pcl::PointXYZI> seg;
      pcl::ExtractIndices<pcl::PointXYZI> extract;


      seg.setModelType (pcl::SACMODEL_PLANE);
      seg.setMethodType (pcl::SAC_RANSAC); //0.05-> velodyne 32
      seg.setDistanceThreshold (0.05); //0.1 -> velodyne 16
      // Con este bucle indicaos el numero de veces para eliminar el suelo
      for(int i = 0 ;i< 3; i ++){
      seg.setInputCloud(cloud1);
      seg.setOptimizeCoefficients (true);
      seg.segment(*inliers, *coefficients);

      extract.setInputCloud(cloud1);
      extract.setIndices(inliers);
```

```
extract.setNegative(true);
extract.filter(*cloud1);
//extract.filter(cloud_p);
//Hacemos que nos muestre un error en caso de que no detecte
   superficie plana y salga del bucle
if (inliers->indices.size () == 0)
ROS_ERROR_STREAM("Could not estimate a planar model for the given
   dataset.");
break;
}
.
.
```

Esto lo realizamos en un bucle que itera varias veces dependiendo de la configuración que se quiera dar, para que la segmentación y la extracción se realice varias veces en la misma nube con lo que conseguimos extraer varios planos de una nube de puntos.

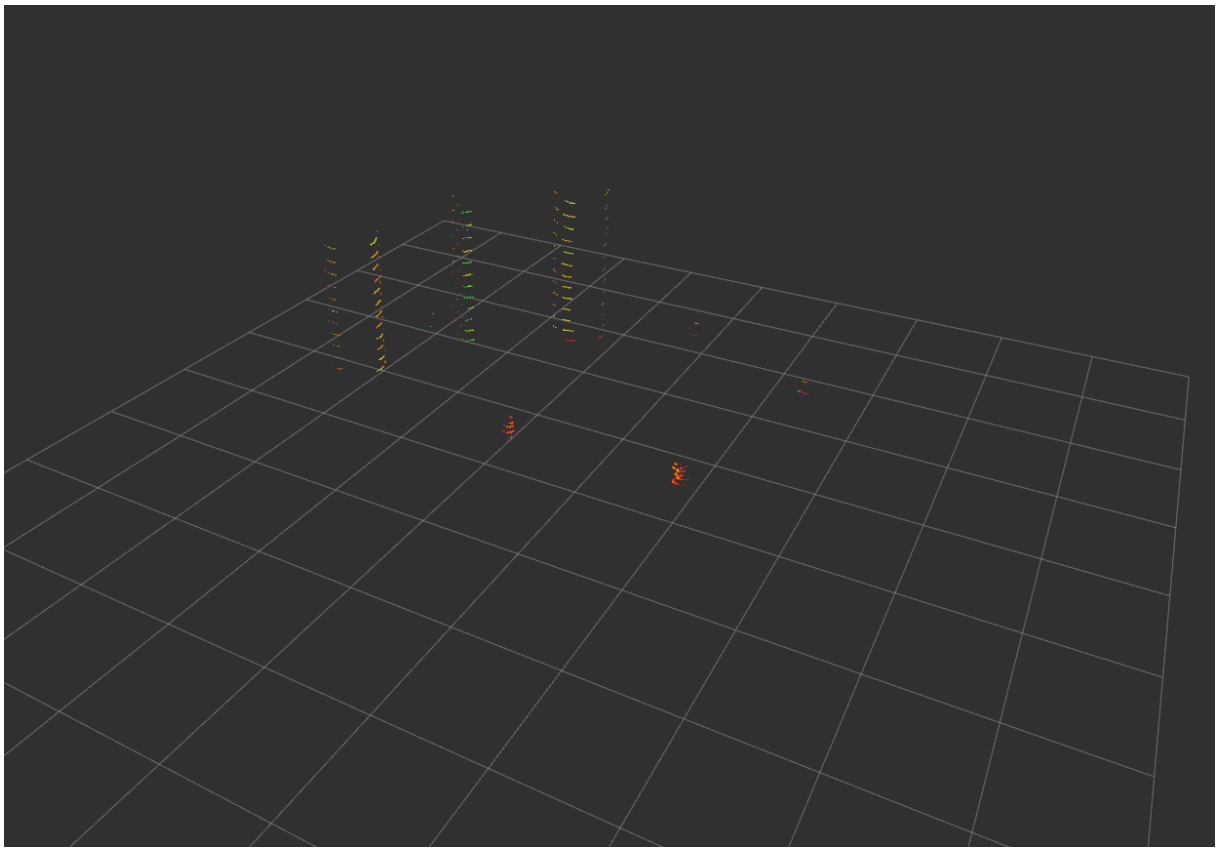


Figura 4.3: Resultado del algoritmo SAC con RoboSense rs-lidar-16

Por último, realizamos la clasificación de la información para lo cual usamos un árbol de búsqueda "KdTree", la cual consiste en una estructura de datos que nos permite hacer búsquedas de puntos cercanos en un espacio de varias dimensiones. El algoritmo funciona dividiendo el espacio en cajas (o regiones) recursivamente hasta que cada región contiene solo un punto. Estas regiones se organizan en un árbol jerárquico, y la búsqueda se realiza comparando el punto de interés con la

región en la que se encuentra para determinar a qué subregión debemos ir. Este proceso se repite hasta encontrar la región que contiene el punto buscado. Para su uso se establecen los parámetros de tolerancia de cluster, tamaño mínimo y máximo. Una vez realizado esto, ya tenemos la información de los conos segmentada y preparada para la detección de posición y posteriormente de color.

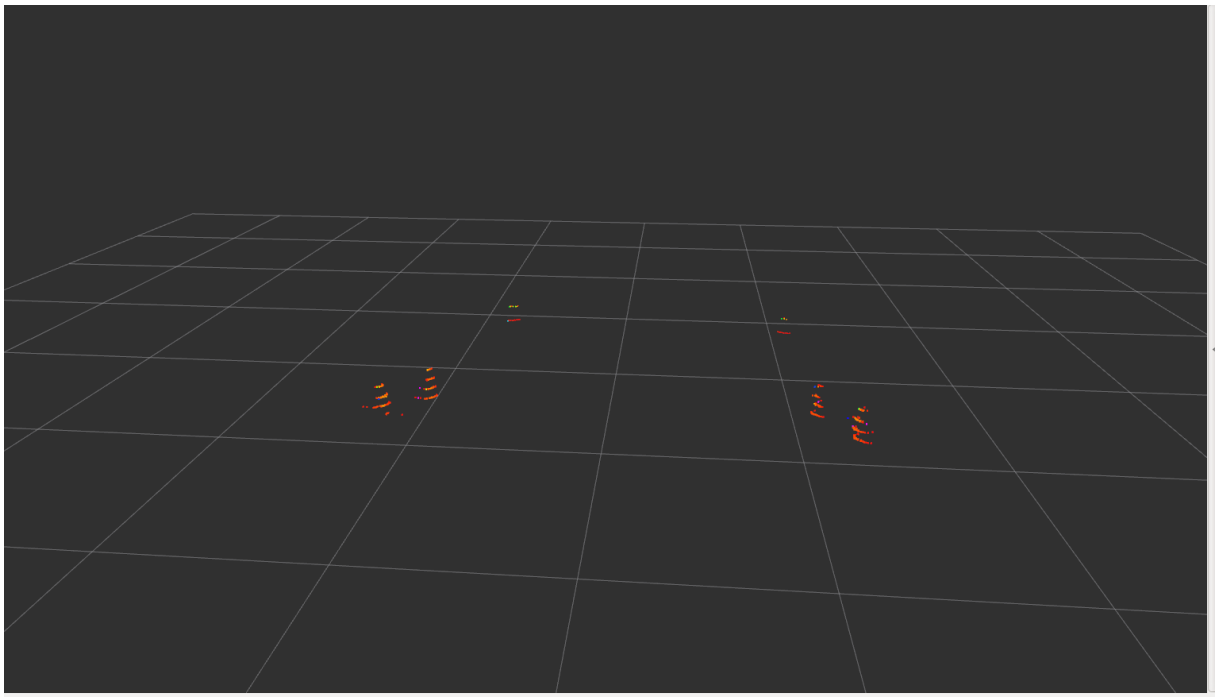


Figura 4.4: Nube de puntos con solo la información de los conos, después de "KDTree", con RoboSense rs-lidar-16

Aquí podemos ver como se ha realizado este código :

```
void perception_handle::returnConePosition (const sensor_msgs::PointCloud2
    &lidar_value){
      .
      .
      .
      //Dection the positions of the cones applying a clustering code
      // Creating the KdTree object for the search method of the
         extraction
      pcl::PointCloud<pcl::PointXYZI> cones;
      pcl::search::KdTree<pcl::PointXYZI>::Ptr tree (new pcl::search::
         KdTree<pcl::PointXYZI>);
      tree->setInputCloud (cloud1);

      std::vector<pcl::PointIndices> cluster_indices;
      pcl::EuclideanClusterExtraction<pcl::PointXYZI> ec;
      ec.setClusterTolerance (1); // 2->16 Velodyne
      ec.setMinClusterSize (10); // 10->16 Velodyne
      ec.setMaxClusterSize (1500);// 250 -> 16 Velodyne
      ec.setSearchMethod (tree);
      ec.setInputCloud (cloud1);
```

```
      ec.extract (cluster_indices);

      pcl::PointCloud<pcl::PointXYZI>::Ptr cloud_cluster (new pcl::
         PointCloud<pcl::PointXYZI>);

      for (std::vector<pcl::PointIndices>::const_iterator it =
         cluster_indices.begin (); it != cluster_indices.end (); ++it){
      //cones.points.push_back(cloud_filtred.points[it])
      for (std::vector<int>::const_iterator pit = it->indices.begin ();
         pit != it->indices.end (); ++pit)
            cloud_cluster->push_back((*cloud1)[*pit]); //*

      cloud_cluster->width = cloud_cluster->size ();
      cloud_cluster->height = 1;
      cloud_cluster->is_dense = true;
      }
      //percepion_ es la funcin para detectar los puntos en los que se
         encuentran los conos y sus colores
      perception_.cones(*cloud_cluster, cones_);
      //Publicamos la nueva nuve de puntos filtrada para verla y poder
         depurar el codigo
      pcl::toROSMsg(*cloud_cluster, lidarValueF_);
      lidarValueF_.header = lidar_value.header;
  }
```

4.2. Detección de conos

Después de segmentar la información de la nube de puntos, el siguiente paso es detectar la distancia a la que se encuentran estos conos. Esto será crucial para el posterior algoritmo de generación de trayectorias para que la silla se pueda desplazar sola a través de estos. Para ello, hemos realizado un código para encontrar su posición promedio a través de todos los puntos generados en las coordenadas x, y, z. Es importante decir que la detección de los conos se realiza en dos partes. En primer lugar, se usa un bucle *for* anidado para comparar cada punto en la nube de puntos con cada uno de los puntos de la nube segmentada y si la distancia entre dos puntos es menor que un valor se considera que ese punto pertenece a un cono y los que pertenecen al mismo se suman y se promedian para obtener la posición. En segundo lugar, el programa verifica si el cono encontrado es nuevo o si ya ha sido detectado previamente. Para esto, comparamos las posiciones de los conos encontrados con las posiciones de los conos ya detectados, almacenados y si esta coincide se considera que es repetido y no se agrega a la lista de conos. El código que realiza este proceso lo podemos ver a continuación:

```
void perception::cones(pcl::PointCloud<pcl::PointXYZI> lidar_filter,
                       visualization_msgs::MarkerArray &cone_detections_)
                          {
                       //fsd_common_msgs::ConeDetections &
```

```
                        cone_detections_){
    //Definimos las variables
    pcl::PointXYZI cones;
    pcl::PointCloud<pcl::PointXYZI> cones_cloud = lidar_filter;
    pcl::PointCloud<pcl::PointXYZI> detections;
    pcl::PointCloud<pcl::PointXYZI> cones_points;
    std::vector<fsd_common_msgs::Cone> clusters;
    .
    .
    .
    .
    for(int i = loop;i< lidar_filter.points.size(); i ++){
          //cones.clear();
          cones.x = 0;
          cones.y = 0;
          cones.z = 0;
          cones.intensity = 0;
          points = 0;
          new_cone = true;
          cones_points.clear();
          cones_organization.clear();
          intensity.clear();
          //ROS_INFO_STREAM("------------NEW Cone
             ----------------------");
          for(int j = loop;j< cones_cloud.points.size(); j ++){
          if(std::hypot((lidar_filter.points[i].x-cones_cloud.points[j].
             x),(lidar_filter.points[i].y-cones_cloud.points[j].y)) <
             0.5){
                cones.x = cones_cloud.points[j].x + cones.x;
                cones.y = cones_cloud.points[j].y + cones.y;
                cones.z = cones_cloud.points[j].z + cones.z;
                points++;
          }
          }
          if(points>20){

          //Aqui luego veremos el codigo encargado para obtener el
             gradiente de colores

          cones.x = cones.x/points;
          cones.y = cones.y/points;
          cones.z = cones.z/points;
          cones.intensity = cones.intensity/points ;

          //ROS_INFO_STREAM("Points adding to map: " << cones);
          //ROS_INFO_STREAM(lidar_filter);


          if(cone_detections_.markers.size() != 0 ){
```

```
                for(int i = 0; i < cone_detections_.markers.size(); i++)
                    {
                if(std::abs(cones.x - cone_detections_.markers[i].pose.
                    position.x) < 0.01
                && std::abs(cones.y - cone_detections_.markers[i].pose.
                    position.y) < 0.01
                && std::abs(cones.z - cone_detections_.markers[i].pose.
                    position.z) < 0.01){
                       new_cone = false;
                }
                }
            }
```

4.3. Detección de colores de conos

La detección de colores ha sido clave a la hora de tomar la decisión acerca de que LIDAR utilizar, si el RoboSense rs-lidar-16 o el Velodyne HDL-32E. Cada uno de estos sensores tiene sus propias fortalezas y debilidades en cuanto a la detección de color, por lo que es importante comprender cómo estas características afectan la precisión y eficacia de la detección de objetos. En este apartado, exploraremos en profundidad las capacidades de detección de color y cómo estas influyen en la elección entre ellos y las diferentes técnicas que hemos explorado para ello.

Se ha experimentado con diversas técnicas para lograr una detección precisa y eficiente de los colores presentes en una escena. Una de las estrategias utilizadas es la media de las intensidades en las detecciones. Este enfoque consiste en calcular la intensidad promedio de las detecciones de los conos y utilizar esa información para clasificar los conos por colores. Este enfoque se basa en la idea de que cada objeto refleja luz de manera diferente dependiendo de la distancia a la que se encuentre, lo que resulta en una intensidad diferente en la detección de LIDAR. Por lo tanto, al promediar la intensidad de las detecciones para un objeto específico, se puede obtener una representación media de la intensidad que se asocia al color de un objeto determinado en nuestro caso conos. Aunque esta técnica es simple y eficiente, también presenta algunos desafíos difíciles de superar y que veremos a continuación.

Con esta idea de las intensidades en un primer momento se pensó en utilizar un sistema de lógica difusa para determinar cada uno de los colores para lo cual se adquirieron datos de las detecciones realizadas por el LIDAR. Para desarrollar este algoritmo se debería tener datos que incluyan información sobre la posición y la intensidad de los puntos detectados, con los cuales se realicen una serie de reglas que describen las características de los conos. Luego se realiza un entrenamiento del sistema de cara a poder optimizar las reglas para mejorar la precisión del sistema y por último, se deberá de implementar y probar su funcionamiento. En nuestro caso se descartó esta vía una vez habíamos establecido unas primeras reglas debido a que es muy difícil obtener una representación precisa de la intensidad media para un objeto específico cuando tenemos gran cantidad de variabilidad en la intensidad de las detecciones que no podemos ni medir ni controlar de ninguna manera, como por ejemplo la interferencia ambiental, como la presencia de sombras o la variabilidad

de la luz ambiental. Como vemos en la Figura 4.5 y en capítulo de Introducción2.5 las pruebas cuentan con unas especificaciones muy concretas y con unos conos específicos. Para esta prueba escogimos un cono Azul tal y como se ve en la imagen.

Figura 4.5: Imagen mientras se adquirían datos para implementar la lógica difusa, con Velodyne HDL-32E

Por otro lado, en las tablas 4.1 y 4.2 podemos ver los resultados de medias de intensidad en conos obtenidas con el programa para diferentes distancias en días diferentes y con condiciones diferentes.

Distancia(m)	Intensidad
0.72	9.4 - 9.8
1.03	5.5 - 5.65
1.33	4.9 - 5.2
1.65	4.9- 5.4
1.79	7.8 - 8.3
1.94	12.5 - 13.1

Tabla 4.1: 10 de noviembre a las 14:15

Distancia(m)	Intensidad
0.72	16.2 - 17.3
1.03	15.2 - 18.3
1.33	12.6 - 15.2
1.65	9.2- 11.2
1.79	13.2 - 14.1
1.94	11.9 - 12.6

Tabla 4.2: 11 de noviembre a las 17:45

Como podemos observar en ambas tablas este método es muy complicado su uso por como indicamos antes su alta variabilidad a efectos externos los cuales no podemos controlar.

Otro método que pensamos y que nos puede permitir una mejor detección de color es un algoritmo que detectara el gradiente de intensidad. Como podemos ver en la Figura 4.6 los conos al tener diferentes bandas estos nos dan diferentes gradientes y escalas del mismo dependiendo si es un cono azul o uno amarillo.

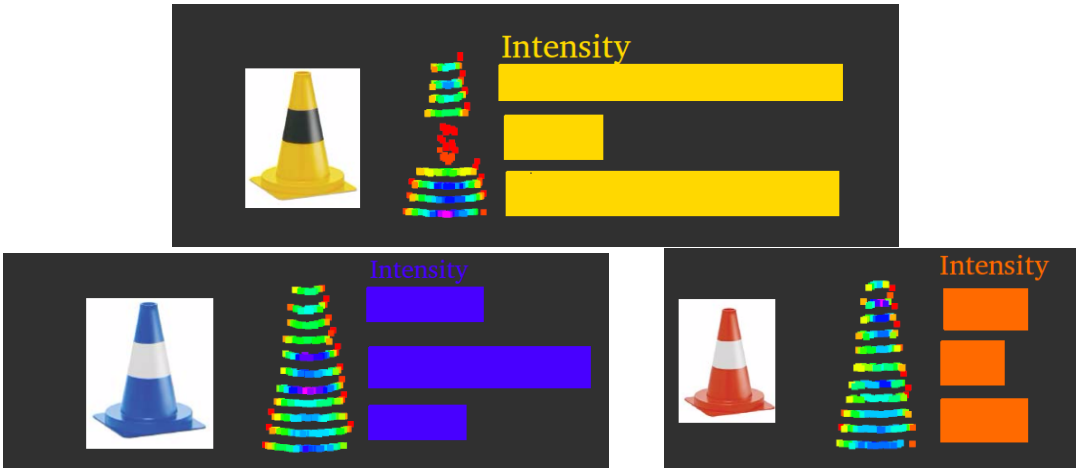


Figura 4.6: Gradiente de intensidades en los conos detectadas con el Velodyne HDL-32E

Para ello, lo que hicimos un código en el que básicamente lo que hace es organizar la nube de puntos de los valores más bajos al alta y una vez realizado esto obtenemos la media de intensidades según cada altura y calculamos los gradientes entre cada punto de altura. Esta información nos permite obtener las pendientes de intensidad entre los puntos mas cercanos del suelo hasta los mas alto y con las diferencias entre de pendientes somos capaces de determinar si sera un cono de color azul, amarillo o naranja. Como vemos en la Figura 4.6 los colores que cuestan mas diferenciar es entre el azul y naranja, pero como nos dan pendientes diferentes somos capaces de diferenciarlas. Para hacer escoger los valores en los que se encuentra cada color de cono fuimos obteniendo los valores más típicos de pendientes para cada detección por separado. A continuación podemos ver en más detalle cómo se hizo el algoritmo para realizar este proceso:

```
void perception::cones(pcl::PointCloud<pcl::PointXYZI> lidar_filter,
   visualization_msgs::MarkerArray &cone_detections_){
      .
      .
      cones_organization.clear();

      for(int j = loop;j< cones_cloud.points.size(); j ++){
      //0.38 velodyne 32
      //0.5 velodyne 16
      if(std::hypot((lidar_filter.points[i].x-cones_cloud.points[j].x),(
         lidar_filter.points[i].y-cones_cloud.points[j].y)) < 0.5){
            .
            .
            cones_organization.push_back(cones_cloud.points[j].z);
```

```
            points++;
      }
      }
      if(points>20){
      //---Organizamos la nube de puntos---
      std::sort(cones_organization.begin(), cones_organization.end());
      point_b = cones_organization[0];
      for(int j = 0;j< cones_organization.size(); j ++){
            if(std::abs(point_b-cones_organization[j]) > 0.01){
            in = 0;
            inten = 0;
            //ROS_INFO_STREAM("intensity "<< j );
            for(int a = 0;a< cones_points.points.size(); a ++){
                  if (std::abs(cones_points.points[a].z-cones_organization
                      [j]) < 0.01)
                  {
                  inten = cones_points.points[a].intensity + inten;
                  in++;
                  }

            }
            inten = inten/in;
            z_points.push_back(cones_organization[j]);
            intensity.push_back(inten);
            point_b = cones_organization[j];
            }

      }
      .
      .
      .
      if (new_cone == true){
            markers.header.frame_id = "rslidar";
            .
            .
            //Calculamos la pendiente

            mMin_i=(-intensity[0] + *std::min_element(intensity.begin(),
               intensity.end()))/(-z_points[0] + z_points[z_min]);
            mMin_f=(*std::min_element(intensity.begin(), intensity.end())
               + intensity[intensity.size()])/( -z_points[z_min] +
               z_points[z_points.size()]);

            mMax_i = (-intensity[0] + *std::max_element(intensity.begin(),
                intensity.end()) )/(-z_points[0] + z_points[z_max]);
            mMax_f= (-*std::max_element(intensity.begin(), intensity.end()
               ) + intensity[intensity.size()])/( -z_points[z_max] +
               z_points[z_points.size()]);
```

```
            //Mensajes de depuracin
            ROS_INFO_STREAM("-----------------------------------------"
                );
            ROS_INFO_STREAM("Inicial "<< std::abs(mMax_i/mMin_i));
            ROS_INFO_STREAM("Final " << std::abs(mMax_f/mMin_f));
            float ini = std::abs(mMax_i/mMin_i);
            float fin =std::abs(mMax_f/mMin_f);

            //Asignamos cada color para cada detecion de cono

            if((ini > 1 && ini < 5) && fin > 1.5 && fin < 2.5){
            cono_naranja = True;
            cono_amarillo = False;
            cono_azul = False;


            }
            else if(ini < 0.5 && fin > 10){
            cono_naranja = False;
            cono_amarillo = True;
            cono_azul = False;

            }
            else{
            cono_naranja = True;
            cono_amarillo = False;
            cono_azul = False;

            }
      }
      }
}
```

Como podemos observar ante una detección que no esté en ningún rango le asignamos el color naranja. Esto lo hacemos así, ya que los conos naranjas se usan como puntos base pero no para construir el mapa, lo que hace que en caso de duda sea mejor detectarlo de esta manera y que no afecte a la generación del mapa. Además finalmente para enviar las señales y construir el mapa organizamos la información de cada cono en objetos virtuales que se conocen como "markers", que sobre todo los usamos para visualizar la posición y orientación de los conos en rviz. Aquí lo podemos ver en más detalle:

```
void perception::cones(pcl::PointCloud<pcl::PointXYZI> lidar_filter,
                       visualization_msgs::MarkerArray &cone_detections_)
                           {
                       //fsd_common_msgs::ConeDetections &
                           cone_detections_){
      if (new_cone == true){
      markers.header.frame_id = "rslidar";
```

```
      //markers.header.frame_id = "velodyne";
      markers.header.stamp = ros::Time();
      markers.ns = "my_namespace";
      markers.id = k;
      markers.type = visualization_msgs::Marker::SPHERE;
      markers.action = visualization_msgs::Marker::ADD;
      markers.pose.position.x = cones.x;
      markers.pose.position.y = cones.y;
      markers.pose.position.z = cones.z;
      markers.pose.orientation.x = 0.0;
      markers.pose.orientation.y = 0.0;
      markers.pose.orientation.z = 0.0;
      markers.pose.orientation.w = 1.0;

      for(int j = 0; j < intensity.size(); j++){
            if(intensity[j] == *std::max_element(intensity.begin(),
               intensity.end())){
            z_min = j;
            }
            if(intensity[j] == *std::max_element(intensity.begin(),
               intensity.end())){
            z_max = j;
            }
      }

      if(cono_azul){

            markers.color.a = 1.0;
            markers.color.r = 0.0;
            markers.color.g = 0.0;
            markers.color.b = 1.0;
            ROS_INFO_STREAM("Azul ");

      }
      else if(cono_amarillo){

            markers.color.a = 1.0;
            markers.color.r = 1.0;
            markers.color.g = 1.0;
            markers.color.b = 0.0;
            ROS_INFO_STREAM("Amarillo");

      }
      else if(cono_naranja){
            markers.color.a = 1.0;
            markers.color.r = 1.0;
            markers.color.g = 0.64;
            markers.color.b = 0.0;
            ROS_INFO_STREAM("Naranja");
```

```
      }

      markers.scale.x = 0.2;
      markers.scale.y = 0.2;
      markers.scale.z = 0.2;
      k++;
      .
      .
      .
```

Además de esto, después se aplicó un filtro de mediana, para lo cual se asignó un valor numérico a cada color y nos quedamos con las detecciones más comunes. Aparte de esto el programa para evitar .ºverflow"de información va eliminado la primera detección cada 100 valores de color. El algoritmo encargado de esto lo vemos por aquí:

```
void perception::cones(pcl::PointCloud<pcl::PointXYZI> lidar_filter,
                       visualization_msgs::MarkerArray &cone_detections_)
                          {
                       //fsd_common_msgs::ConeDetections &
                          cone_detections_){
      .
      .
      .
      std::vector<float> mediana_v;
      bool newCone = true;
      for(int k =0; k< conesMean_.markers.size();k++){
      if(std::abs(std::hypot((conesMean_.markers[k].pose.position.x -
         markers.pose.position.x),
            (conesMean_.markers[k].pose.position.y - markers.pose.position.
               y))) < 0.5){
            newCone = false;
            c = k;
            break;
      }
      }

      if(!newCone){
      if(markers.color.g == 0 ){ //Azul
            meanColor_[c].push_back(1);
      }
      else if (markers.color.g == 1){ //Amarillo
            meanColor_[c].push_back(2);
      }
      else{ //Naranja
            meanColor_[c].push_back(3);
      }

      for(int y = 0; y < meanColor_[c].size();y++){
            mediana_v.push_back(meanColor_[c][y]);
```

```
        }
        sort(mediana_v.begin(), mediana_v.end());
        int n = mediana_v.size();
        mediana = (n % 2 == 0) ? (mediana_v[n/2 - 1] + mediana_v[n/2]) / 2 :
            mediana_v[n/2];

        if(mediana >= 1 && mediana <= 1.5 ){ // Azul
              markers.color.a = 1.0;
              markers.color.r = 0.0;
              markers.color.g = 0.0;
              markers.color.b = 1.0;
        }
        else if (mediana > 1.5 && mediana <= 2.5 )// Amarillo
        {
              markers.color.a = 1.0;
              markers.color.r = 1.0;
              markers.color.g = 1.0;
              markers.color.b = 0.0;
        }
        else{// Naranja
              markers.color.a = 1.0;
              markers.color.r = 1.0;
              markers.color.g = 0.64;
              markers.color.b = 0.0;
        }
        }

        else {
        conesMean_.markers.push_back(markers);
        meanColor_.push_back(fila_empty);

        }

        if(meanColor_[c].size() > 100)
        meanColor_[c].erase(meanColor_[c].begin());


        cone_detections_.markers.push_back(markers);
        detections.points.push_back(cones);

        b++;
```

Es importante destacar que las variables usadas se pueden configurar de diferentes formas dependiendo del LIDAR o resultados que se quieran conseguir. Se ha seleccionado el LIDAR Velodyne HDL-32E para el proceso de detección de colores y posición de los conos debido a sus mejores resultados. Esto es lo que se esperaba, ya que tiene una mayor resolución. En la Figura 4.7 podemos ver las detecciones sin el preprocesado que como vemos nos da una mejor resolución que el de 16.

Para las pruebas que se realizaron con la silla en funcionamiento se usó el

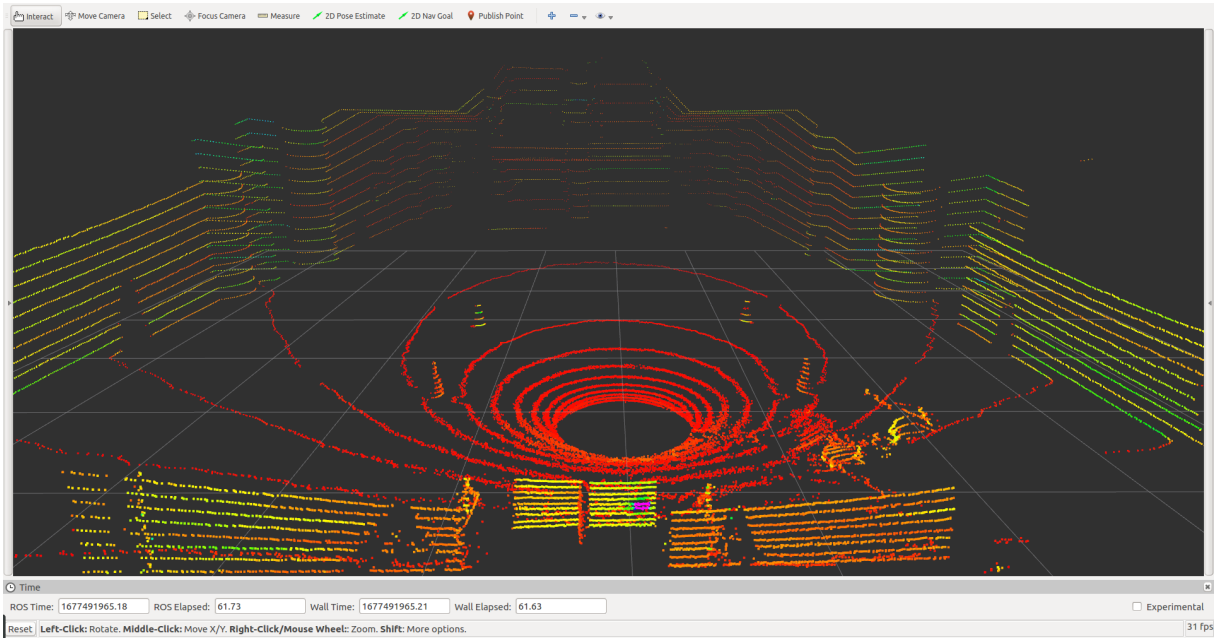


Figura 4.7: Velodyne HDL-32E antes de preprocesado

Robosense pese a esto, ya que se encontraba ya montado en la silla lo cual facilitó el proceso para hacer las pruebas.

En la figura 4.9 podemos ver el resultado de aplicar todas estas técnicas simulando que la silla está en la posición de salida del circuito de una de las pruebas y en la Figura 4.8 como estaba dispuesto la silla y el circuito en la realidad.

Figura 4.8: Disposición en la realidad de salida en Acceleration

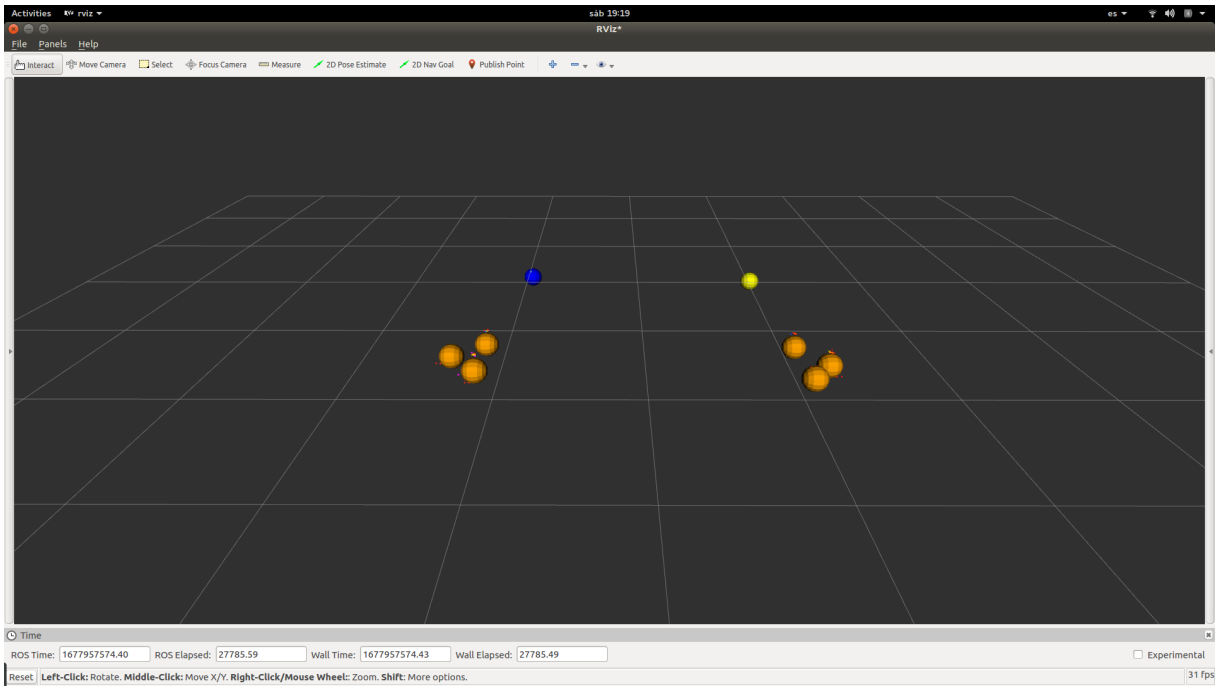


Figura 4.9: Resultado de detección de obstáculos y los colores con RoboSense rs-lidar-16

Capítulo 5

Estimación y mapping

Después de haber detectado la ubicación y color de los conos que pueden servir de obstáculos para nuestro vehículo, es importante llevar a cabo una estimación precisa de la localización del vehículo en el espacio y de los objetos que se encuentran en su entorno. Este proceso es crucial para poder desarrollar un sistema de navegación autónoma fiable y seguro.

En este apartado, abordaremos los conceptos teóricos y los algoritmos necesarios para realizar esta estimación. Cabe destacar que este apartado se enfocará principalmente en el aspecto teórico, ya que debido a la falta de tiempo y errores en la sujeción de los encoders, no fue posible implementar los algoritmos de manera completa y obtener los datos necesarios para aplicarlos. La idea principal para el desarrollo de esta parte del sistema es desarrollar un sistema SLAM (Simultaneous Localization and Mapping) haciendo fusión sensorial de las detecciones del LIDAR y el WSS (Wheel Speed Sensor) que son los encoders que se encuentran en ambas ruedas de la silla de ruedas. Para ello se desarrollaron diversos módulos de ROS con el objetivo de que trabajaran en conjunto.

5.1. Modelo cinemático de la silla de ruedas

El modelo que hemos usado para describir el movimiento de la silla de ruedas y poder determinar su posición según su movimiento ha sido el modelo de un robot móvil diferencial. Esto debido a que este tipo de modelos se utiliza en robots móviles que tienen la capacidad de moverse independientemente una rueda de la otra.

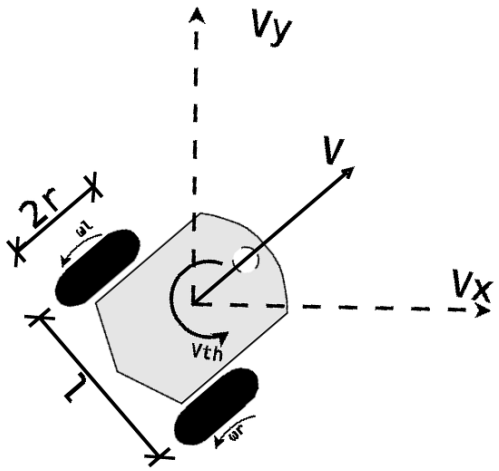


Figura 5.1: Modelo cinemático del robot diferencial

El modelo es básicamente una representación matemática que describe la relación entre las velocidades de las ruedas y la posición y orientación del robot en el espacio. Este consiste en considerar el robot como una estructura rígida que se mueve en un plano, y se basa en la relación entre las velocidades lineales de las ruedas y la velocidad angular del robot en su centro de masa. Este modelo matemático es capaz de predecir la posición y orientación del robot a partir de la velocidad de sus ruedas, y viceversa.

La principal ventaja del modelo cinemático de un robot diferencial es que proporciona una forma sencilla y rápida de controlar el movimiento del robot. Dado que el modelo se basa en la relación entre las velocidades de las ruedas y la posición y orientación del robot, es posible controlar el movimiento del robot ajustando la velocidad de cada rueda de forma independiente. Esto permite al robot seguir trayectorias curvas y sortear obstáculos de manera efectiva. Para su implementación se desarrolló el siguiente código en Arduino:

```
void updateOdom() {
  double dt, dleft, dright, dx, dy, dxy_ave, dth, vxy, vth;

  /* Get the time in seconds since the last encoder measurement */
  dt = (odomInfo.encoderTime - odomInfo.lastEncoderTime) / 1000.0;

  /* Save the encoder time for the next calculation */
  odomInfo.lastEncoderTime = odomInfo.encoderTime;

  /* Calculate the distance in meters travelled by the two wheels */
  dleft = (leftPID.Encoder - odomInfo.prevLeftEnc) / ticksPerMeter;
  dright = (rightPID.Encoder - odomInfo.prevRightEnc) / ticksPerMeter;
  odomInfo.prevLeftEnc = leftPID.Encoder;
  odomInfo.prevRightEnc = rightPID.Encoder;

  /* Compute the average linear distance over the two wheels */
  dxy_ave = (dleft + dright) / 2.0;



  /* Compute the angle rotated */
  dth = (dright - dleft) / wheelTrack;



  /* Linear velocity */
  vxy = dxy_ave / dt;



  /* Angular velocity */
  vth = dth / dt;
```

```
  /* How far did we move forward? */
  if (dxy_ave != 0) {
       dx = cos(dth) * dxy_ave;
       dy = -sin(dth) * dxy_ave;
       /* The total distance traveled so far */
       odomInfo.linearX += (cos(odomInfo.angularZ) * dx - sin(
       odomInfo.angularZ) * dy);
       odomInfo.linearY += (sin(odomInfo.angularZ) * dx + cos(
       odomInfo.angularZ) * dy);
  }

  /* The total angular rotated so far */
  if (dth != 0)
       odomInfo.angularZ += dth;

  /* Represent the rotation as a quaternion */
  geometry_msgs::Quaternion quaternion;
  quaternion.x = 0.0;
  quaternion.y = 0.0;
  quaternion.z = sin(odomInfo.angularZ / 2.0);
  quaternion.w = cos(odomInfo.angularZ / 2.0);



}
```

El código básicamente es la implementación de lo que explicamos del modelo cinemático es decir como vemos la función comienza calculando el tiempo transcurrido desde la última medición del encoder, y luego calcula la distancia que ha recorrido cada rueda desde la medición anterior. También calcula la distancia promedio lineal recorrida por las dos ruedas y el ángulo rotado por el robot.

Posteriormente calcula la velocidad lineal y angular del robot en función de los cambios de distancia y tiempo, con lo que luego obtiene la distancia que ha avanzado el robot en el eje x y en el eje y. Finalmente obtenemos el ángulo total girado por el robot y lo representa como un cuaternión para posteriormente publicar la orientación y la velocidad del robot en el tópico .ºdom". También a modo de depurar se publica la posición de los encoders de las ruedas en otro tópico.

5.2. Modelo cinemático para el coche

Para la implantación en el prototipo de vehículo, este es uno de los puntos básicos que se debe modificar. Probablemente sea necesario añadir más sensores como GPS para mejorar el posicionamiento y utilizar técnicas de filtros extendidos de Kalman, etc. Para ello sería necesario un estudio detallado de su posible implementación.

El modelo básico utilizado para describir los movimientos de un vehículo se suele simplificar al conocido como Bicycle Kinematic Model. Este es un modelo matemático simplificado que describe el movimiento de un robot en dos dimensiones utilizando

dos ruedas, una delantera y una trasera, como la bicicleta. Este se basa en dos variables: la velocidad lineal del robot, v, y la velocidad angular, Ω.

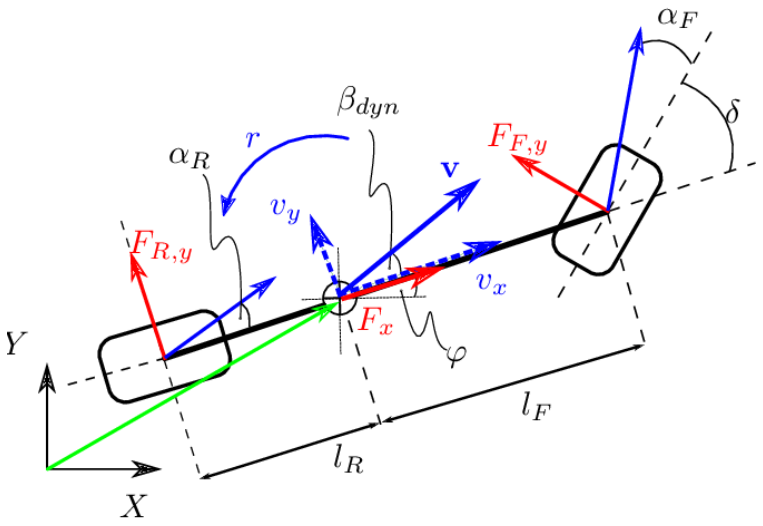


Figura 5.2: Dynamic bicycle model [7]

El modelo asume que las dos ruedas del robot están ubicadas en los extremos opuestos de un eje, y que la rueda delantera está en una posición fija mientras que la rueda trasera puede girar en cualquier dirección. Para describir el movimiento del robot, el modelo cinemático de la bicicleta utiliza dos ecuaciones. La primera ecuación describe la velocidad lineal del robot en términos de la velocidad angular y el radio de giro, R. Esta ecuación es la de la cinemática del avance:

$$v = \omega R \tag{5.1}$$

La segunda ecuación describe la velocidad angular del robot en términos de la velocidad lineal y el ángulo de dirección, , que es el ángulo entre la dirección del robot y la dirección del eje de las ruedas traseras. Esta ecuación se conoce como la ecuación de la cinemática de la dirección:

$$\omega = \frac{v\tan(\delta)}{L} \tag{5.2}$$

Donde L es la distancia entre los ejes de las dos ruedas.

Estas dos ecuaciones pueden combinarse para describir el movimiento del robot en términos de las entradas de velocidad angular de las ruedas izquierda y derecha. Esto se logra a través de la siguiente ecuación:

$$v = \frac{R}{2}(\omega_l + \omega_r) \tag{5.3}$$

$$\omega = \frac{R}{L}(\omega_r - \omega_l) \tag{5.4}$$

donde ω_l y ω_r son las velocidades angulares de las ruedas izquierda y derecha, respectivamente. Este modelo es una versión simplificada ya que se puede volver más complejo como vemos en la siguientes ecuaciones de movimiento que podemos

ver en más detalle en artículo [7].

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\varphi} \\ \dot{v}_x \\ \dot{v}_y \\ \dot{r} \end{pmatrix} = \begin{pmatrix} v_x \cdot cos(\delta) - v_y \cdot sin(\delta) \\ v_x \cdot sin(\delta) + v_y \cdot cos(\delta) \\ r \\ \frac{1}{m}(mv_y r + (F_{R,x} - F_{F,y} \cdot sin(\delta)) \\ \frac{1}{m}((F_{R,y} + F_{F,y} \cdot sin(\delta) - mv_y r) \\ \frac{1}{I_z}(F_{F,y} \cdot l_F \cdot cos(\delta) - F_{R,x} \cdot l_R) \end{pmatrix} \tag{5.5}$$

Donde la m es la masa, I_z la inercia, l_R y l_F la distancia delantera y trasera respecto al centro de masas, $F_{R,y}$ y $F_{F,y}$ es la fuerza lateral de la rueda delantera y trasera. Esta última representa las fuerzas de contacto entre la superficie y el neumático, para obtener estos se usa una simplificación del Pacejka tire model[9].

Por tanto, el modelo cinemático de la bicicleta es un modelo matemático simple que describe el movimiento de un robot móvil de dos ruedas utilizando dos variables: velocidad lineal y velocidad angular.Esto lo hace perfecto para describir el movimiento del vehículo según el giro de las ruedas. Por tanto para implementarlo en el Formula Student, se tendría que implementar un nodo de ROS como el que vimos en el robot diferencial que nos permita sustituirlo por el nuevo modelo y así estimar la velocidad y posición del robot para poder diseñar un sistema de control.

5.3. Fusión sensorial para mejorar la estimación de la posición

Para mejorar la estimación de la posición de la silla, se ha decidido aplicar fusión sensorial con la información del LIDAR y los encoders de las ruedas. Para esto se aplicara un Filtro Extendido de Kalman que será el encargado de esta tarea, permitiéndonos así generar una estimación precisa de la posición y orientación del vehículo móvil y los objetos en su entorno y generar así un mapa preciso lo que nos permitirá un mejor resultado en las pruebas que queremos realizar. Básicamente un Filtro extendido de Kalman es una extensión del Filtro de Kalman que nos permite utilizar técnicas de estimación utilizando sistemas dinámicos no lineales, es decir, teniendo en cuenta factores como el rozamiento o el propio ruedo que tienen los sensores.

Para desarrollarlo deberemos crear un nodo que se dedique exclusivamente a esta tarea. Para hacerlo podemos usar el paquete ROS `"robot_pose_ekf"`. Este se encargará de realizar la fusión sensorial de los encoders y las detecciones del LIDAR. Para ello se utilizará un modelo de movimiento para relacionar la posición y velocidad de la silla de ruedas con las velocidades de las ruedas detectadas por la odometría. Este modelo es básicamente lo que hemos descrito anteriormente y usaremos posición y orientación como entrada a nuestro algoritmo. Luego, se utiliza la información de los conos obtenida del LIDAR para actualizar la estimación de la posición y orientación de la silla de ruedas. Esta comparación permite corregir la estimación de la posición de la silla de ruedas y mejorar la precisión de la estimación. Este proceso se realiza iterativamente a medida que se reciben nuevas medidas de LIDAR y odometría, permitiendo que la estimación de la posición de la silla de ruedas

se vaya refinando con el tiempo. Este paquete también proporciona herramientas para ajustar los parámetros del filtro EKF, como la varianza del ruido del modelo de movimiento y observación, la varianza de las mediciones del LIDAR y los encoders y la frecuencia de actualización del filtro EKF. Es importante que se haga una buena configuración de estos parámetros para lograr una buena precisión en la estimación de la posición y orientación del vehículo. Una vez tenemos esto ya podemos proceder a la generación de un mapa del circuito por donde deberá circular el vehículo.

5.4. SLAM y generación del mapa de coste

SLAM (Simultaneous Localization and Mapping) es una técnica utilizada en robótica para construir un mapa del entorno y estimar la posición del robot en tiempo real. El objetivo del SLAM es permitir que un robot se mueva en un entorno desconocido mientras construye un mapa de ese entorno y se localiza en el mapa. Para crear este mapa lo ideal es utilizar el paquete ROS "gmapping"que nos proporciona una implementación del SLAM para construir un mapa de entornos en 2D utilizando un LIDAR y la posición y orientación generada por el nodo de fusión sensorial.

Además si queremos mejorar la calidad del mapa, se puede realizar una corrección de bucle cerrado. La corrección de bucle cerrado implica la detección de lugares en el mapa donde la silla de ruedas ha pasado antes y la corrección de la posición y orientación estimadas en esos lugares. Esto ayuda a reducir los errores acumulativos y mejora la precisión del mapa.

Además este mapa será fundamental de cara a la generación de trayectorias que veremos más adelante. Para ello deberemos asignar valores de costo a diferentes partes del entorno en función de su dificultad relativa para navegar. Por ejemplo, en nuestro caso como el circuito queda definido por los conos que son de color azul y amarillo entre cada uno de estos conos generaremos unos valores de costo de manera que impida que la silla se pueda salir del circuito o llevarse algún cono.

Para implementar este en ROS aparte del paquete "gmaping"deberemos usar paquete `"costmap_2d"` que básicamente nos proporciona una estructura para construir un mapa de coste en 2D basado en los datos de sensor y la geometría del robot.También debemos hacer un archivo de configuración YAML para especificar los parámetros de configuración del mapa de coste, como la resolución, el tamaño del mapa y los rangos de valores de coste. Básicamente estos valores dependerán de la prueba que realice nuestro vehículo.

Capítulo 6

Control y planificador de trayectorias

La planificación de trayectorias y el control son componentes críticos en el diseño de sistemas robóticos que pueden navegar en entornos complejos y desconocidos, como un robot diferencial. La planificación de trayectorias se refiere al proceso de generar una secuencia de posiciones y orientaciones a través del tiempo que permiten que el robot llegue a su destino de manera segura y eficiente, mientras que el control se refiere al proceso de enviar comandos a los actuadores del robot para seguir la trayectoria planificada y mantener el robot en movimiento. Es importante que destaquemos antes de empezar que este apartado igual que el anterior tiene un desarrollo muy teórico debido que como no se pudo establecer una localización ni generar un mapa no pudimos preparar los algoritmos para generación de trayectorias.

Para este caso es muy importante también diferenciar las pruebas, ya que como pudimos ver en el capítulo 2.5,son diferentes por lo que es necesario definir diversas "misiones"para cada una de las pruebas.

6.1. Planificador de trayectorias

La planificación de trayectorias es una parte fundamental de cualquier sistema de navegación autónomo y se utiliza para garantizar que nuestro robot se mueva de manera segura y eficiente hacia su destino deseado. La planificación global y la planificación local son los dos enfoques principales que utilizaremos para cumplir nuestros objetivos. Una parte característica de nuestro caso particular es cómo definir el punto objetivo, ya que lo que a nosotros nos interesa es dar vueltas en un circuito. Para ello, lo que se ha pensado es ir generando puntos objetivos cambiantes a medida que avanzamos en el circuito, es decir, el primer punto será el generado por la primera pareja de conos y a partir de ahí iremos variando este a la más lejana que vamos detectando.

6.1.1. Planificador Local

Dentro de la robótica existen varios planificadores locales de trayectorias. Para el proyecto que precede este, que básicamente consistía en la simulación del sistema autónomo con el Formula Student, se implementó un algoritmo de triangulación debido a su buen funcionamiento y simpleza. Lo que hace nuestro algoritmo es emparejar los conos que se encuentran a una distancia del coche menor de 6 m, luego se obtiene la recta que une cada pareja, calculando así el punto medio que es el que usaremos como punto objetivo para la planificación.En la Figura 6.1 podemos ver la planificación local a través de este método.

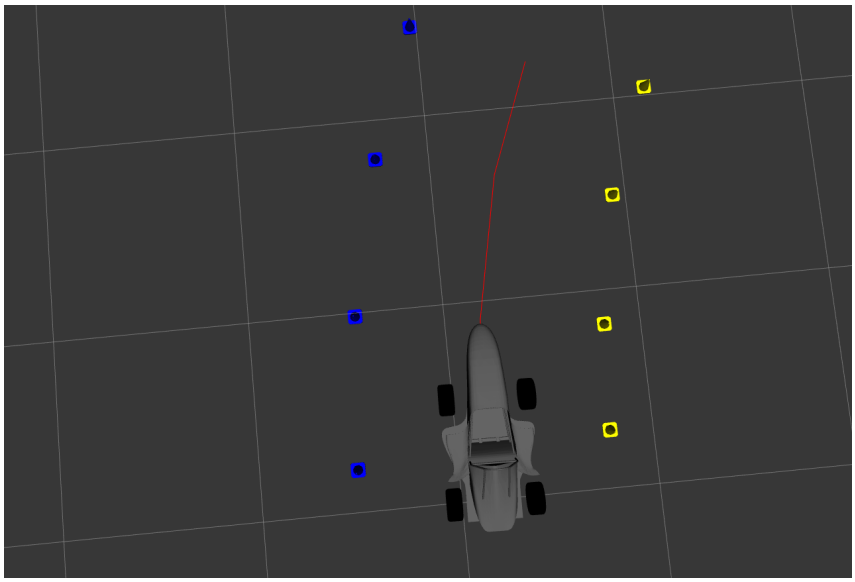


Figura 6.1: Planificador local implementado

En el siguiente código podemos ver cómo se realizó la implementación de este:

```
for(int i = 0; i < map.cone_yellow.size(); i++){
     const auto it_blue = min_element(map.cone_blue.begin(), map.
        cone_blue.end(),
     [&](const fsd_common_msgs::Cone &a,
     const fsd_common_msgs::Cone &b) {
           const double da = fabs(hypot(map.cone_yellow[i].position.x - a.
              position.x,
                 map.cone_yellow[i].y - a.position.y));
           const double db = fabs(hypot(map.cone_yellow[i].position.x - b.
              position.x,
                 map.cone_yellow[i].position.y - b.position.y));

           return da < db;
     });

     geometry_msgs::PoseStamped pose;

  if(fabs(hypot(it_blue->position.x - map.cone_yellow[i].position.x,
                 it_blue->position.y - map.cone_yellow[i].position.y )) <
                     6){

     float angulo = atan2((it_blue->position.y - map.cone_yellow[i].
        position.y),
           (it_blue->position.x - map.cone_yellow[i].position.x));
     float d1 = fabs(hypot(it_blue->position.x - map.cone_yellow[i].
        position.x,
                 it_blue->position.y - map.cone_yellow[i].position.y ));

     pose.pose.position.x = (d1/2)*cos(angulo) + map.cone_yellow[i].
```

```
            position.x;
        pose.pose.position.y = (d1/2)*sin(angulo) + map.cone_yellow[i].
            position.y;
        pose.pose.position.z = 0.0;

        if(fabs(hypot(it_blue->position.x - state.car_state.x
                    ,it_blue->position.y - state.car_state.y )) < 8){
              path_.poses.push_back(pose);
        }
    }
```

Sin embargo, pese a que este sistema nos diera buenos resultados de cara a la simulación estudiando el sistema real y viendo sus limitaciones hemos visto que este algoritmo es necesario darle más complejidad de cara que nos de unos mejores resultados. Además este algoritmo no tenía en cuenta ningún tipo de mapa de coste y cogía directamente las detección desde los conos.

Para esto se decidió hacer un pequeño estudio de las generaciones de trayectorias locales que podríamos desarrollar. Se estudiaron principalmente dos, la primera es el algoritmo de estrella o A* que es usada por Verdino, proyecto realizado por el Grupo de Robótica de la ULL, y que consistía en un carro de golf que se mueve de forma autónoma [10]. Este es un algoritmo de búsqueda de camino óptimo que se utiliza para encontrar el camino más corto entre dos puntos en un grafo ponderado. Además debemos generar una función heurística para estimar el costo restante para alcanzar el objetivo desde un nodo dado.

El segundo que estudiamos es usado por otros equipos de Formula Student y se trata del Gaussian Random Paths(GRP). Este básicamente es un método de generación de trayectorias locales para robots móviles que se basa en la aleatoriedad y en el uso de distribuciones Gaussianas. Este algoritmo tiene como objetivo encontrar una trayectoria local que conecte el punto de inicio con el punto final de una forma suave y que tenga una alta probabilidad de éxito en cuanto a la evitación de obstáculos. Para implementarlo deberemos definir los paramentos de la distribución gaussiana, los cuales podemos ir ajustando para un mejor funcionamiento. Después generamos una muestra aleatoria utilizando una función de generación de números aleatorios a través de una semilla específica y con esto obtendremos la primera posible trayectoria para nuestro robot. Una vez hecho esto el algoritmo deberá verificar la viabilidad de la misma, es decir, si no choca con obstáculos en el entorno. Para esto, se utiliza el mapa de costos que se ha generado previamente. Además se verifica la calidad de la misma utilizando una función de costo teniendo en cuenta diferentes factores, como la longitud de la trayectoria, la cantidad de curvas pronunciadas, la distancia a obstáculos, etc. Un código sencillo donde podemos ver su implementación puede ser el siguiente:

```
    // Definir los parmetros de la distribucin Gaussiana
    float mu = 0.0; // media
    float sigma = 0.2; // desviacin estndar

    for(int i = 0; i<100; i++){
```

```
// Generar una muestra aleatoria de la distribucin Gaussiana
float x = mu + sigma * generateRandomNumber();

// generateRandomNumber() es una funcin que genera un nmero
   aleatorio entre 0 y 1
float y = mu + sigma * generateRandomNumber();

// Verificar la viabilidad de la trayectoria
bool isPathClear = true;
//Primero determinamos si chocamos con algn obstculo
// Convertir las coordenadas (x, y) a ndices de celda en el mapa

int a = (int) (x / resolution);
int b = (int) (y / resolution);

// Verificar si la celda correspondiente a la trayectoria est
   ocupada

if (map[i][j] == 1) {
      return false; // Trayectoria inviable
}

// Verificar si la celda y sus vecinos estn libres

for (int ii = i-1; ii <= i+1; ii++) {
      for (int jj = j-1; jj <= j+1; jj++) {
      if (map[ii][jj] == 1) {
            isPathClear = false; // Trayectoria inviable
      }
      }
}

if (isPathClear) {
      // Evaluar la calidad de la trayectoria
      float distance = sqrt(pow(x - goalX, 2) + pow(y - goalY, 2));

      // Calcular la distancia mnima entre el punto actual y los
         obstculos
      float minDistance = FLT_MAX;

      for (const auto& obstacle : obstacles) {
      float d = sqrt(pow(x - obstacle.x, 2) + pow(y - obstacle.y, 2)
         ) - obstacle.radius;
      if (d < minDistance) {
      minDistance = d;
      }
      }

      float cost = distance + obstacleWeight * minDistance;
```

```
// evaluatePath() es una funcin que calcula la calidad de la
   trayectoria
if (cost < bestCost) {

 // Se ha encontrado una nueva mejor trayectoria
      bestCost = cost;
      bestPathX = x;
      bestPathY = y;
}
}
```

Para nuestro caso creemos que posiblemente el algoritmo de GRP nos de unos mejores resultados pero hasta que no se realice una aplicación práctica no sabemos con claridad cuál de los anteriores será mejor.

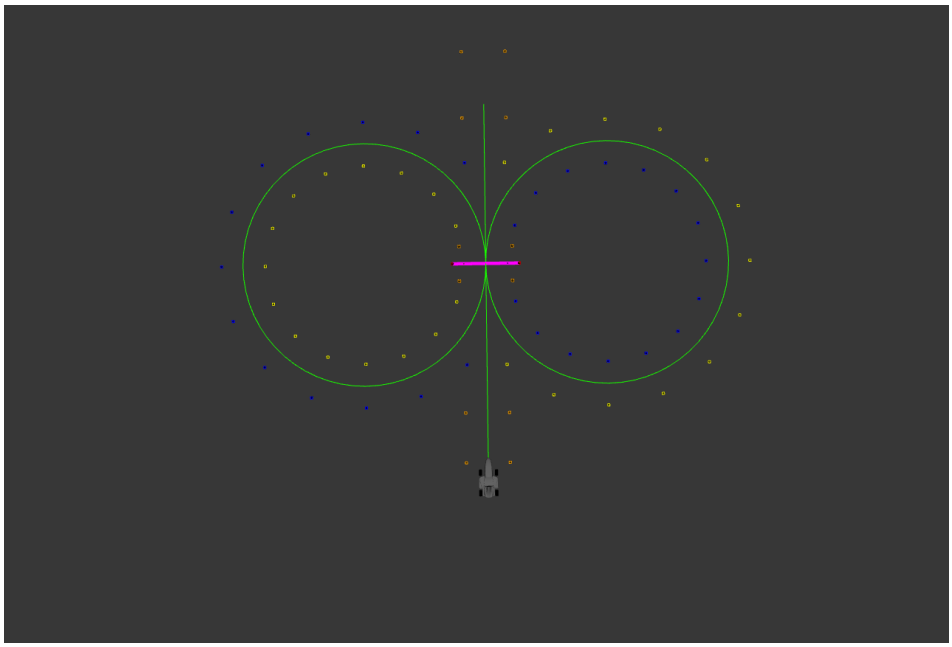
6.1.2. Planificador global

El sistema de planificación global es de vital importancia en el sistema autónomo, ya que se encargará de que el vehículo complete las pruebas. Para obtener los puntos del planificador global se aplicaron dos métodos según el tipo de prueba. En la prueba del *Trackdrive* no tenemos ningún mapa de coste por lo que será necesario ir construyendo en la primera vuelta a medida que vamos avanzando por él a través del planificador local. Por otro lado las pruebas de *Skidpad* y *Acceleration* ya conocemos el mapa de coste, por lo que podemos generar una trayectoria global e ir corrigiendo con la local en los casos necesarios. En el trabajo anterior, este algoritmo era muy poco óptimo porque iba generando esta a través de los puntos por los que iba pasando, que en el caso de la simulación era suficiente, pero que en la realidad nos daría muchos problemas para poder implementarlo realmente por lo que debemos buscar otro algoritmo para generar esta. Este lo podemos ver en el siguiente código:

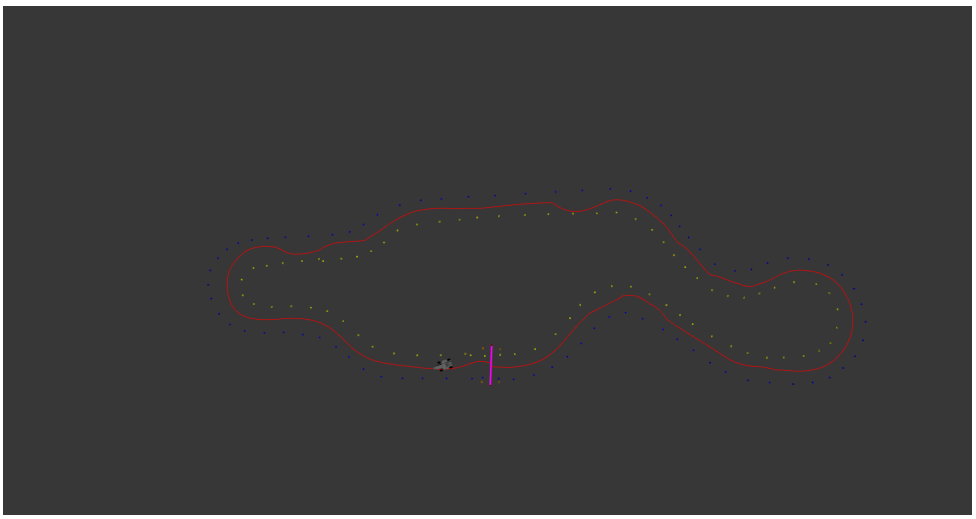
```
if(lap_ == 1 && mission_.mission == "trackdrive"){

geometry_msgs::PoseStamped pointCar;
pointCar.pose.position.x = state_.car_state.x;
pointCar.pose.position.y = state_.car_state.y;
pointCar.pose.position.z = 0;
path_planing_.poses.push_back(pointCar);
}

else if(mission.mission== "skidpad"){
double pi = std::acos(-1.0);
float radio = 9.125;
int i = -16;

while(i < 0){
      pose.pose.position.x = i;
```

```
            pose.pose.position.y = 0;
            path_planing.poses.push_back(pose);
            i = i + 3.2;
      }

      for(int vuelta = 0; vuelta < 2; vuelta++){
            double ang = pi/2;

            while(ang > -3*pi/2){
            pose.pose.position.x = radio*cosf(ang);
            pose.pose.position.y = radio*sinf(ang)-9.125;
            path_planing.poses.push_back(pose);
            ang = ang - 2*pi/60;
            }
      }

      for(int vuelta = 0; vuelta < 2; vuelta++){
            double ang = pi;
            //ROS_WARN_STREAM(ang);
            while(ang > -pi){
            pose.pose.position.x = radio*sinf(ang);
            pose.pose.position.y = radio*cosf(ang)+9.125;
            path_planing.poses.push_back(pose);
            ang = ang - 2*pi/60;
            }
      }

      i = pose.pose.position.y;

      while(i < 14.0){
            pose.pose.position.x = i;
            pose.pose.position.y = 0;
            path_planing.poses.push_back(pose);
            i = i + 3.2;
      }
      }

      else if(mission.mission == "acceleration"){
      int i;
      while(i < 75.0){
            pose.pose.position.x = i;
            pose.pose.position.y = 0;
            path_planing.poses.push_back(pose);
            i = i + 1;
      }
      }
```

En la Figura 6.2b y 6.2a podemos apreciar el circuito generado por los dos métodos descritos anteriormente.



(a) Resultado del planificador de global para *Skidpad*



(b) Resultado del planificador global para un circuito de *Trackdrive*

Figura 6.2: Planificador global

Como hemos dicho antes este método nos puede funcionar de cara a la simulación de hecho como ya dije anteriormente en el trabajo anterior no se había generado un mapa de coste algo que es fundamental de cara a la implementación del sistema en la realidad, además que este nos ayudara mucho mas a mejorar el algoritmo que aplicamos en el trabajo anterior. Una de las posibilidades de cara a mejorar esta planificación global es cambiar la resolución del mapa de coste y generar una planificación global con el mismo algoritmo que generamos las trayectorias locales.

Una vez tengamos implantado un algoritmo de este tipo ya podemos ir generando objetivos para que vayan siguiendo esta planificación global para eso usamos un algoritmo de tracking que consiste en que el vehículo persiga un punto en el camino que se encuentra a cierta distancia, en la Figura 6.4 podemos apreciar esto.

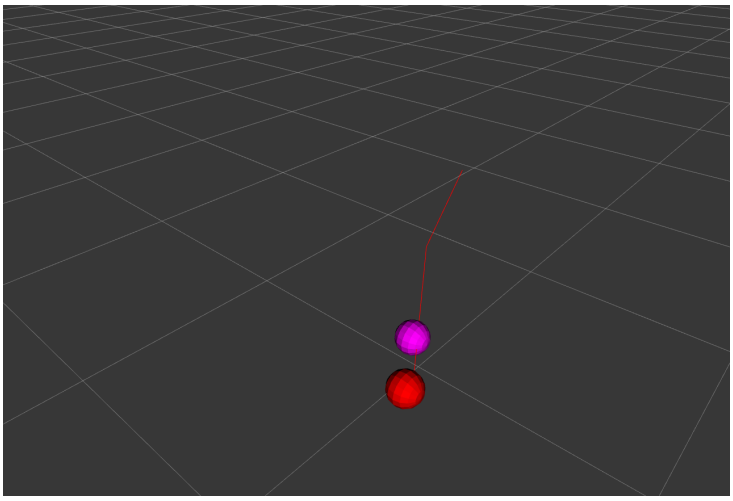


Figura 6.3: El punto rojo es el objetivo que persigue el vehículo y el rosado será el siguiente

Este sistema se suele comparar con la forma que tenemos los humanos de conducir ya que tenemos que mirar un punto hacia delante y dirigirnos hacia él. Como ya vimos antes en nuestro sistema estos puntos a los que debe dirigirse son los correspondientes al planificación local o global dependiendo de la prueba. Para implementar este sistema en el vehículo, se obtiene el ángulo entre el robot y el punto objetivo, este será el valor de consigna que usaremos para ir moviendo nuestro robot.

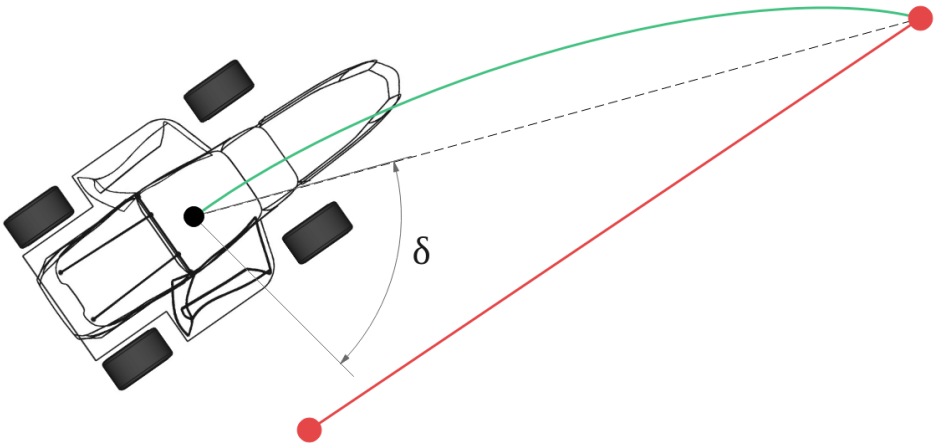


Figura 6.4: En rojo vemos la trayectoria del vehículo y en verde la trayectoria que seguirá para alcanzar el punto objetivo

6.2. Control

En el campo de la robótica móvil, el control es un aspecto clave para lograr un desempeño óptimo y seguro del robot en su entorno. El control adecuado de un robot móvil puede mejorar su capacidad para navegar con precisión, evitar obstáculos, realizar tareas complejas y lograr una mayor eficiencia energética. Además, el control también juega un papel fundamental en la seguridad, ya que permite al robot responder de manera efectiva a situaciones imprevistas y garantizar que su comportamiento sea predecible y confiable. Para nuestro caso se ha decidido aplicar un control PID, que como sabemos es un método de control clásico muy utilizado en la industria y la robótica que se basa en el uso de tres términos: Proporcional, Integral y Derivativo. El control PID se utiliza para ajustar y mantener una variable de salida (como la velocidad, posición o temperatura) en un valor deseado (referencia), teniendo en cuenta las desviaciones y cambios en la variable de entrada(sensores) y la respuesta del sistema en tiempo real.

En nuestro caso hemos decidido usar esto debido a su facilidad para poder aplicarlo. Además en nuestro caso está aplicado directamente en el microcontrolador arduino así que únicamente debemos mandar los comandos de velocidad a través del tópico para que la silla se mueva y controlador actué. El código para implementarlo lo podemos ver a continuación:

```
/* PID routine to compute the next motor commands */
void doPID(SetPointInfo * p) {
      long Perror;
      long output;

      Perror = p->TargetTicksPerFrame - (p->Encoder - p->PrevEnc);

      // Derivative error is the delta Perror
      output = (Kp * Perror + Kd * (Perror - p->PrevErr) + Ki * p->Ierror)
          / Ko;
      p->PrevErr = Perror;
      p->PrevEnc = p->Encoder;

      output += p->output;

      if (output >= MAXOUTPUT)
      output = MAXOUTPUT;
      else if (output <= -MAXOUTPUT)
      output = -MAXOUTPUT;
      else
      p->Ierror += Perror;

      p->output = output;
}

/* Read the encoder values and call the PID routine */
void updatePID() {
```

```
  leftPID.Encoder = robot->leftMotor->read();
  rightPID.Encoder = robot->rightMotor->read();

  /* Record the time that the readings were taken */
  odomInfo.encoderTime = millis();
  odomInfo.encoderStamp = nh.now();

  if (!moving) /* If we're not moving there is nothing more to do */
       return;

  /* Compute PID update for each motor */
  doPID(&leftPID);
  doPID(&rightPID);

  robot->setSpeeds(leftPID.output, rightPID.output); /* Set the motor
      speeds */
}

void updatePID() {
  leftPID.Encoder = robot->leftMotor->read();
  rightPID.Encoder = robot->rightMotor->read();

  /* Record the time that the readings were taken */
  odomInfo.encoderTime = millis();
  odomInfo.encoderStamp = nh.now();

  if (!moving) /* If we're not moving there is nothing more to do */
       return;

  /* Compute PID update for each motor */
  doPID(&leftPID);
  doPID(&rightPID);

  robot->setSpeeds(leftPID.output, rightPID.output); /* Set the motor
      speeds */
}
```

Se realizaron diversas pruebas de este controlador con la silla y pudimos comprobar con una primer ajuste de las variables proporcional, integral y derivativa que tenía un buen comportamiento, aunque en algunas superficies el controlador tendía a fallar ya que las ruedas de la silla patinaban. En el caso del vehículo tenemos pensado utilizar este mismo controlador aunque es posible que también tengamos que explorar otros usados por otros equipos como puede ser el MCP(Model Predictive Control) que básicamente usa un modelo matemático del sistema para predecir el comportamiento futuro y optimizar el control de la salida del sistema en tiempo real. A diferencia del controlador PID, el MCP es capaz de tener en cuenta las limitaciones y restricciones físicas del sistema, como las limitaciones de velocidad, aceleración, torque, entre otras. El proceso de control en el controlador MCP se realiza en dos

pasos principales: en primer lugar, se genera un modelo matemático del sistema que se va a controlar, y en segundo lugar, se utiliza este modelo para predecir el comportamiento futuro del sistema y determinar la acción de control óptima.

Capítulo 7

Conclusiones y líneas futura

En este proyecto final de máster, se ha llevado a cabo el diseño de un sistema autónomo para el vehículo de Formula Student de la Universidad de La Laguna. Para su desarrollo, se utilizó una silla de ruedas Vermerien, lo que permitió la aplicación de técnicas de robótica en un entorno real. Se emplearon distintos algoritmos para la detección de conos, la localización, fusión sensorial, SLAM, generación de trayectorias y control de la silla, con una estructura de nodos proporcionada por ROS.

Pese a que se lograran todos los objetivos planteados en este trabajo se tienen que hacer todavía una serie de mejoras para hacer que el sistema sea mas robusto y obtener mejores resultado. Estas las podemos ver a continuación:

- Fijar de una mejor forma los encoderes en las ruedas de la silla.
- Implementar los sistemas de EKF y la generación del mapa de coste, realizando varias pruebas con pequeños circuitos.
- Aplicar modelos de control mas complejos pero que nos permitan resultados mas óptimos y un mejor rendimiento cuando se realizan las pruebas.
- Estudiar si es suficiente con las detecciones del LIDAR y los Encoders para la localización del vehículo.
- Implementar circuitos de control que nos hagan de RES(Remote Emergency Stop) para las pruebas de la silla.
- Mejorar el cableado actual de la silla para que sea más robusto.
- Estudiar si es necesario mejorar la precisión en las detecciones de colores usando cámaras estereoscópicas.
- Poner en los nodos archivos de configuración yaml para ajustar los filtros de percepción, generación de mapa de coste, trayectorias y control sin tener de ejecutar todo el código.
- Implementar el sitema autónomo en el Formula Student haciendo los cambios necesarios en el modelo y el control del vehículo.

En conclusión, este proyecto ha permitido la aplicación de distintas tecnologías y técnicas en el desarrollo de un sistema autónomo y ha establecido una base sólida para el equipo de FSULL-Dynamics para la implementación completa del sistema en el monoplaza real y competir en la modalidad de Formula Student Driverless.

Capítulo 8

Conclusions and Future lines

In this Master's final project, an autonomous system for the Formula Student vehicle of the University of La Laguna was designed. For its development, a Vermerien wheelchair was used, which allowed the application of robotics techniques in a real environment. Different algorithms were employed for cone detection, localization, sensor fusion, SLAM, trajectory generation, and chair control, with a node structure provided by ROS.

Although all the objectives in this work were achieved, there is still room for improvement to make the system more robust and achieve better results. These improvements are listed below:

- Improve the fixing of the encoders on the wheelchair wheels.
- Implement EKF systems and cost map generation, performing several tests with small circuits.
- Apply more complex control models that allow for more optimal results and better performance during testing.
- Study whether LIDAR and encoders' detections are sufficient for vehicle localization.
- Implement control circuits that act as RES (Remote Emergency Stop) for wheelchair testing.
- Improve the current wiring of the wheelchair to make it more robust.
- Study whether it is necessary to improve color detection accuracy using stereoscopic cameras.
- Place configuration YAML files in the nodes to adjust perception filters, cost map generation, trajectories, and control without having to execute all the code.
- Implement the autonomous system in the Formula Student vehicle by making the necessary changes to the model and vehicle control.

In conclusion, this project has allowed the application of different technologies and techniques in the development of an autonomous system and has established a solid foundation for the FSULL-Dynamics team to implement the system in the car fully

Bibliografía

[1] Carlos Javier Alvarez Casablanca. “TFG: Sistema de Captura de Datos y Localización Automática para un Vehículo Eléctrico”. En: (2012).

[2] ROS Components. “RS-LiDAR-16”. https://www.roscomponents.com/es/lidar-escaner-laser/251-rs-lidar-16.html?search_query=LIDAR&results=42.

[3] Enrique Fernandez et al. *Learning ROS for Robotics Programming - Second Edition*. 2nd. Packt Publishing, 2015. ISBN: 1783987588.

[4] “FSG Competition Handbook 2020”. https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FSG20_Competition_Handbook_v1.0.pdf.

[5] “FSG Rules 2020”. https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FS-Rules_2020_V1.0.pdf.

[6] “Gazebo”. http://gazebosim.org/.

[7] Juraj Kabzan et al. “AMZ Driverless: The Full Autonomous Racing System”. En: *ArXiv* abs/1905.05150 (2019).

[8] “Libreria TF ROS Wiki”. http://wiki.ros.org/tf.

[9] Hans B. Pacejka y Egbert Bakker. “THE MAGIC FORMULA TYRE MODEL”. En: *Vehicle System Dynamics* 21.sup001 (1992), págs. 1-18. DOI: 10.1080/00423119208969994. eprint: https://doi.org/10.1080/00423119208969994. URL: https://doi.org/10.1080/00423119208969994.

[10] Grupo de Robótica de La ULL. “Safe and Reliable Path Planning for the Autonomous Vehicle Verdino”. En: *IEEE Intelligent Transportation Systems Magazine* 8.2 (2016), págs. 22-32. DOI: 10.1109/MITS.2015.2504393.

[11] “ROS”. https://www.ros.org/.

[12] “rqt ROS Wiki”. http://wiki.ros.org/rqt.

[13] “Rviz ROS Wiki”. http://wiki.ros.org/rviz.

[14] “The Point Cloud Library”. https://pointclouds.org/.

Apéndice A

Requisitos de normativa

Para comprobar los requisitos de la normativa hemos ido al apartado de vehículos driverless, de las reglas de Formula Student Germany 2020 [5].

- **DV1.1 Base del coche**: Los requerimientos y restricciones generales para los vehículos autónomos, son los mismos que los demás vehículos, ya sean de combustión o eléctricos.
- **DV1.2 Comunicaciones Inalámbricas**: Está prohibido cambiar parámetros, mandar comandos o hacer cualquier modificación en el software a través de comunicaciones inalámbricas. Recibir información del vehículo en una sola dirección si está permitido. Durante los eventos dinámicos, las comunicaciones inalámbricas pueden estar limitadas o ser interrumpidas. El único dispositivo que está permitido que envía comandos a través de comunicaciones inalámbricas es el Remote Emergency System (RES) descrito en la norma DV1.4. El uso de GPS está permitido, pero no tiene un lugar asegurado durante la competición.
- **DV1.3 Datta Logger**: Los organizadores darán data logger estandarizado, que deberá ser instalado en los vehículos durante la competición. Más adelante las especificaciones del data logger y los requerimientos del Hardware y el interfaz del software, se podrán encontrar en el Handbook de la competición. La intención del data logger es entender y reproducir el estado del sistema en caso de error. Esto incluye un set básico de señales definidas en el Handbook y el conjunto de señales que monitoriza el EBS (Emergency Brake System), para garantizar la redundancia y detectar errores.
- **DV1.4 Remote Emergency System (RES)**: Todos los equipos deben estar equipados con el RES especificado en el Handbook. El sistema deberá consistir en dos partes, el control remoto y el módulo del coche. Además, el RES tiene dos funciones adicionales:
  - Hacer saltar el Shutdown Circuit (SDC) cuando el botón de parada emergencia remoto es pulsado.
  - Implementar las comunicaciones del coche para el control de carrera.
    - El control de carrera es el encargado de enviar la señal de *Go* al coche.
    - La señal *Go* reemplaza las banderas verdes.

  El módulo RES del coche deberá estar directamente integrado en SDC con un relé conectado en serie con los botones de apagado. La antena del RES debe montarse sin obstruir y sin interferir en partes próximas (otras antenas, etc).

- **DV1.5 Shutdown Circuit (SDC)**: Si el SDC es abierto por el Sistema Autónomo (AS) o por el RES, sólo se podrá resetear manualmente (con otro botón fuera del vehículo, próximo al ASMS (Autonomous System Master Switch) o vía al LVMS (Low Voltage Master Swich) (apagando y encendiendo de nuevo). El SDC puede ser cerrado por el AS (Autonomous System), si las siguientes condiciones se cumplen:
  - Conducción manual: Si este modo se activa, el AS comprobará que el EBS está desactivado (No posible actuación del EBS).
  - Conducción autónoma: Si este modo está activo, el ASMS está activado y hay suficiente presión en los frenos (los frenos están cerrados).
- **DV2.1 Señales**: Todas las señales del AS son consideradas como Señales Críticas (SCS).
- **DV2.2 Master Switch del AS**: Los actuadores para el giro y el freno deben activarse con el LVMS y el ASMS. Cuando el ASMS está Off, deben cumplirse lo siguiente:
  - No debe girar, frenar o acelerar, por el sistema autónomo.
  - Los sensores y la unidad de procesamiento pueden estar operativos.
  - El coche debe estar disponible para ser empujado, según la norma A7.
  - Debe ser posible pilotar el vehículo en modo manual.

  Después de encender el ASMS, el coche puede empezar a moverse y los frenos deben mantenerse cerrados (Estado: *As ready*, figura 2.1) hasta la señal de *Go* enviada desde el RES (Estado: *As driving*, Figura A.1)
- **DV2.4 Actuación de giro**: El sistema de actuación de giro solo debe funcionar si el vehículo se encuentra en el estado RD (Ready to Drive), aunque puede mantenerse activo durante una frenada de emergencia. El movimiento manual debe ser posible sin pasos de liberación manual, y solo mientras el ASMS se encuentra apagado. Él vehículo solo debe realizar una transición de estado si se cumplen todas las condiciones y hasta que se complete la transición, los ASSI (Autonomous System Status Indicator) deben indicar el estado inicial.
- **DV2.4 Definiciones de estados del sistema autónomo**: El sistema autónomo deberá contar con los estados y transiciones de estados que podemos ver en la figura 2.1. El actuador de giro sólo podrá tener los siguientes estados:
  - *Unavailable*: La fuente de potencia de los actuadores está desconectada, el manejo manual es posible.
  - *Available*: La fuente de potencia está conectada y los actuadores pueden responder a los comandos del AS acorde con la norma 2.3.1

  El freno (Service Brake) tiene los siguientes estados:
  - *Unavailable*: La fuente de potencia de los actuadores está desconectada, el freno manual es posible.

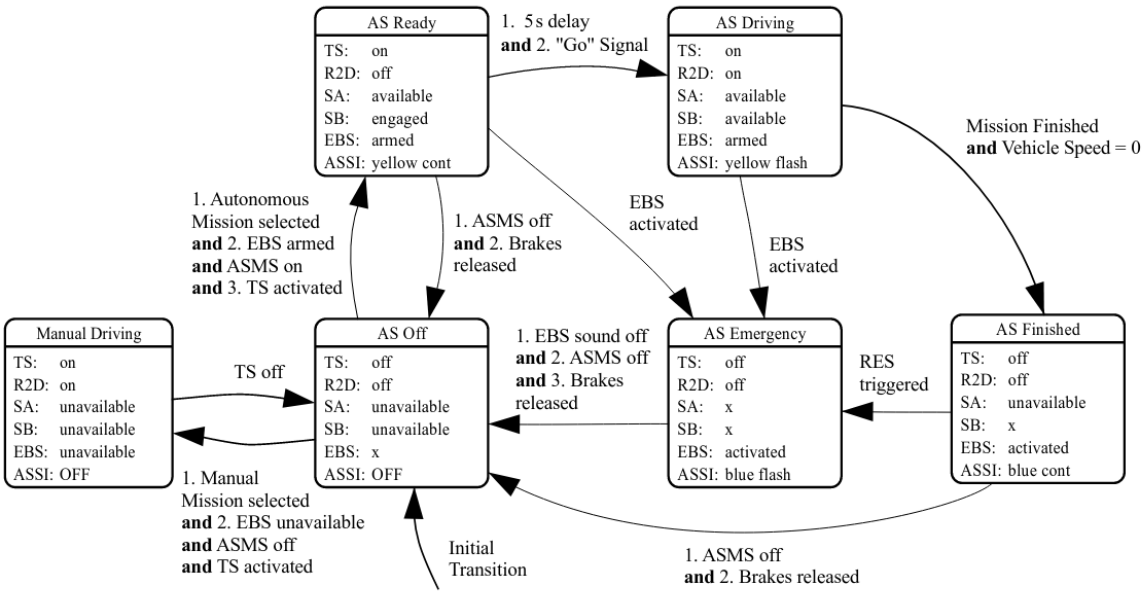


Figura A.1: AS máquina de estados [5]

- *Engaged*: Previene al coche de rodar, se pulsa el freno un 15 %
- *Available*: La fuente de potencia está conectada y los actuadores pueden responder a los comandos del AS.

El EBS solo tiene los siguientes estados:

- *Unavailable*: La fuente de potencia de los actuadores está desconectada, el freno de emergencia no es posible.
- *Armed*: Inicia la maniobra de frenado de emergencia inmediata si el SDC está abierto o si se interrumpe el suministro de LVS %
- *Activated*: Los frenos son cerrados y la potencia del EBS es cortada. Los frenos solo se pueden liberar después de realizar los pasos manuales.

- **DV2.5 Indicador del status del AS (ASSIs)**: El vehículo debe tener tres indicadores luminosos para saber en qué estado se encuentra el AS, los podemos ver con detalle en la tabla 2.1. Estas luces deberán encontrarse justo por detrás de main hoop. Además el *AS Emergency* deberá emitir un sonido con las siguientes características:

  - On-/Off de 1 Hz a 5 Hz.
  - Un ciclo de trabajo del 50 %.
  - Un sonido entre 80 dBA y 90 dBA como máximo.

- Duración de entre 8s a 10s después del AS Emergency.

AS Off	AS Ready	AS Driving	AS Emergency	AS Finished
off	yellow continuous	yellow flashing	blue flashing	blue continuos

Tabla A.1: Iluminación según estado[5]

- **DV2.6 Misiones Autónomas**: El sistema autónomo deberá implementar las siguientes misiones:

  - *Aceleration*.
  - *Skidpad*.
  - *Autocross*.
  - *Trackdrive*.
  - *EBS test*.
  - *Inspection*.
  - *Manual driving*.

  La misión seleccionada deberá ser indicada en el AMI (Autonumus Mission Indicator).

- **DV3.1 Requerimientos técnicos del sistema de emergencia de frenado**: El vehículo debe tener equipados un EBS, el cual debe ser alimentado por el LVMS, ASMS, RES y un relé el cual estará alimentado por el SDC. Además, el EBS solo debe utilizar sistemas pasivos con almacenamiento mecánico de energía. La pérdida energía eléctrica debe conducir a una maniobra directa de frenado de emergencia. Debe ser parte al sistema hidráulico de frenado y el freno manual (por el pedal de freno) deberá ser desactivado por el AS. También es necesario que sea fácil de desactivar para los Oficiales. Por último, es necesario decir que no está permitido el uso de accesorios de presión en los circuitos neumáticos críticos de función del EBS y de cualquier otro que utilice el mismo almacenamiento de energía.

- **DV3.2 Funcionamiento seguro del sistema de emergencia de frenado**: Debido al carácter crítico de seguridad del EBS, el sistema debe permanecer completamente funcional o el vehículo debe pasar automáticamente al estado seguro en caso de que se produzca un solo fallo. En este estado seguro, el freno está pulsado para prevenir que el coche se mueva y se abre el SDC. Se debe realizar una verificación inicial en la que se garantice que el EBS y su redundancia puedan aumentar la presión de frenado tal y como se espera antes de que el AS cambie a *As ready*.

- **DV3.3 Funcionamiento seguro del sistema de emergencia de frenado**: El sistema deberá tener un tiempo de reacción que no exceda de 200 ms y la deceleración debe ser superior a $8 m/s^2$.

- **DV3.4 Sensores y Componentes, Montaje**: Todos los sensores deben estar colocadas de modo seguro en el vehículo. Además, no deben estar colocados en contacto con el casco del piloto y deberán posicionarse en la superficie del vehículo. Los sensores se pueden montar con una distancia máxima de 500 mm por encima del suelo y menos de 700 mm hacia adelante de la parte delantera de los neumáticos delanteros (ver figura A.2).
  Las antenas que actúan exclusivamente como tales con el lado más largo <100mm pueden sobresalir del vehículo. Para los componentes detrás del compartimento del conductor, se acepta un voladizo del 25 % del volumen de su caja delimitadora.

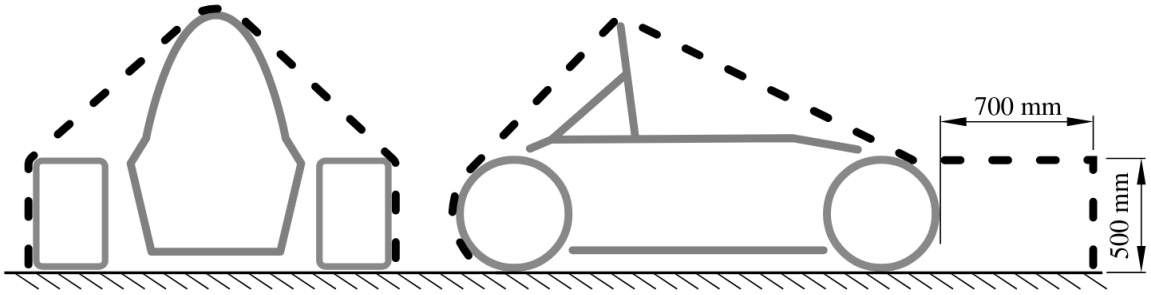


Figura A.2: Envoltura donde se pueden montar los sensores[5]